Defining Virtual Views of Electronic Resources using Declarative Queries

Mark.B.Dixon
Leeds Metropolitan University
Leeds, LS6 3QS
England

ABSTRACT
This paper describes a mechanism which allows multiple views of underlying electronic resource structures to be created. The aim is to address problems faced by users when trying to navigate file system structures defined by third parties. A framework has been developed which supports the definition and evaluation of an appropriate solution. The framework includes a query language that allows for the construction of user defined views using a declarative style grammar. Several deployment architectures, which support practical application of the proposed framework, have been developed and form the basis of an initial implementation.

General Terms
Dynamic Resource Retrieval, Declarative Query Execution, File Systems.

Keywords
Query language; virtual views; document retrieval; declarative.

1. INTRODUCTION
Modern day computer systems provide us with the ability to store and retrieve enormous amounts of information. Web-based access to electronic resources is now a trivial matter. Web-servers often provide controlled access to an underlying file system, the management of which is typically handled using the mechanisms provided by the operating system. The filing cabinet metaphor which prescribes the representation of information as files within a hierarchy of folders is now an accepted standard. The approach has been replicated across many storage and distribution domains, including the Internet, as evidenced by the presence of hierarchical paths within Uniform Resource Identifiers (URIs) [1]. The use of such a simple hierarchical structure has clearly been an overwhelming success, but is there a better way of organising *our* information?

As individuals we typically organise file system structures to suit our own particular requirements. The way we actually decide upon these structures depends on many factors including personal preference, historical patterns, accepted standards or convention. Over time however our own personal requirements can of course change, making this single structure less meaningful in different situations. A much bigger problem however is that information sources are very commonly shared. Once an individual is required to navigate through a structure defined by someone else, they lose the ability to recall the decision making process that went into creating the structure in the first place. Hence, when faced with an alien file system structure, most users typically resort to a combination of common sense and guess work while navigating.

The simple fact is that a single hierarchical file system structure does not suit the needs of all its potential users. In fact a single file system structure often does not even support the needs of the user who created the structure in the first place. The creators of the Unix file system clearly recognised this problem and partially addressed it through the support for symbolic links [2]. These allow files and directories to simultaneously appear within several appropriate, and often more meaningful, locations within a single hierarchical structure. Support for short-cuts within window manager desktops is also a way of addressing the difficulties in identifying commonly used resources within a large hierarchical file system.

The real way to address this problem however is to provide a mechanism that allows for the presentation of a single underlying file system structure in a manner that suits the needs or a particular user or class of users. In essence what is required is the ability to create one or more *Virtual Views* of an underlying file system. This concept is diagrammatically represented in Figure 1.

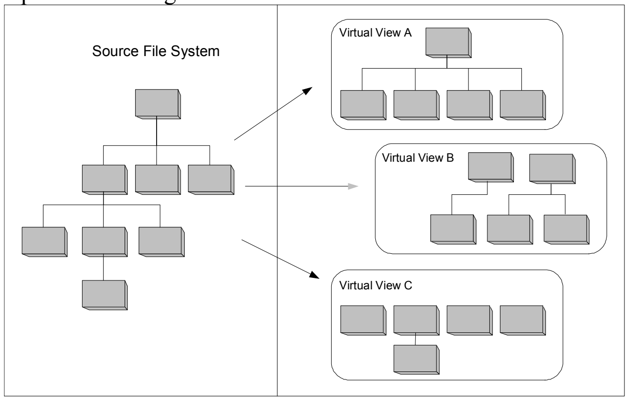

Figure 1: Virtual views of an underlying file system

2. RELATED WORK
The idea of presenting a virtual representation of the underlying file system is not a new concept of course. In fact virtual file systems have been in existence since the mid 1980's, for example Sun Microsystems developed the Virtual File System (VFS) and vnodes [3]. Traditional virtual file systems tend to be hard-coded abstractions, based on underlying concrete file systems, which provide a consistent API that supports a uniform mechanism of file access and control. The virtual views

approach extends this idea by allowing ad-hoc abstractions to be defined and evaluated dynamically.

As well as low level operating system level solutions there are an increasing number of user level applications designed to help access resources in a manner not restricted by their storage location. Many of these are indexing based search approaches and include Apple spotlight [4] and their associated Smart Folders, Microsoft's Windows Search [5] and Linux tools such as Beagle and Beagle++ [6]. This work drops between these user level applications and the very concrete implementation ideas associated with kernel virtual file systems. Also the ideas associated with aggregation of disparate resources, supported in a very rudimentary way by technologies such as RSS feeds [7], also have a bearing on the virtual views approach.

There are a number of projects that have addressed the middle ground associated with this work. FiST is a language which aims to ease the development of file systems [8]. The approach provides an intermediate language capable of describing a file system that can be compiled into modules for multiple platforms. This removes the need to write low-level kernel specific code in order to implement new file systems. Although operating at a lower level of abstraction, the idea of allowing definition of file system through an independent abstract language is a similar approach to the virtual views concept described in this paper.

In order to realize the true potential of organizing data in multiple structures suited to individual needs, the presence of meta-data to help describe the resources is necessary. Additionally the ability to use expressions to define multiple views is fundamental. Two approaches that exhibit such functionality are the Semantic File System [9] and the Logic File System [10]. The SFS introduces the concept of *virtual directories* which are interpreted using queries. The attributes used however (i.e. the meta-data) cannot be assigned as part of the query. Also, navigation into a virtual directory that is the result of a query is not possible. The LFS is an object based approach, where logical descriptors are associated with objects allowing expressions to be defined for supporting navigation and querying. An interesting concept specified as Future directions of this work suggests treating directories and files in a similar way, thus allowing navigation into files as well as directories.

The Nebula File system [11] was designed specifically to support information management. Nebula implements files as sets of attributes and allows the user to assign attributes to files, in fact this ability is fundamental to the design of this particular system. It also supports the concept of *views* in the same way as suggested by the virtual views approach. As with the Semantic File System however, navigating the result of a query is not possible.

Spyglass is a file metadata search system that is specially designed for large-scale storage systems [12]. The approach taken by Spyglass is to build indexes and use snapshots that allow for fast searching of large amounts of metadata. The system is designed for scalability since the work recognises the fact that the size and complexity of today's storage systems make it difficult to manage available files. The system uses several interesting techniques to help deal with the requirement of large scale storage support. Some of these techniques could be applied to the virtual view query execution described within this paper. However, the idea of indexing meta-data information in order to improve performance may be in conflict when performing on-demand access to remote web-based resources, hence there is not a direct synergy between this work and the approach described within this paper.

Damasc is a file system where rich data management services for scientific computing are provided as a native part of the file system [13]. Users of the system can use declarative queries and updates over views of underlying files. An additional layer on the file system is used which exposes the contents of files through which views can be defined and used for queries and updates. This work was undertaken specifically to support large scale data processing for scientific applications, whereas the virtual views approach is designed to be a general purpose solution. Hence there are both similarities and differences between this work and the virtual views discussed within the paper.

3. REQUIREMENTS FOR A VIRTUAL VIEW SYSTEM

A system that supports presentation of virtual views should have the ability to dynamically identify resources of interest via the evaluation of a set of instructions. A virtual view should therefore be describable using a definition which provides these instructional details. Such a definition can be described as a *virtual view query*. An evaluation engine can then process these queries on request to produce a set of results.

By examining the related work, further research and development was undertaken in order to identify the primary requirements of such a system. Identification and justification of specific requirements is necessary in order to highlight fundamental issues which must be suitably addressed by an implementation. The primary requirements are discussed below. These often address the weaknesses of existing approaches. Arguably there are of course many other requirements which may be identified. Those described however were seen as fundamental to the successful development of an effective solution.

3.1 On Demand Query Evaluation

Since the evaluation of a view query results in a virtual structure which is derived from one or more other structures, each evaluation should be performed on request. This is necessary to ensure up to date information is extracted from the original sources. This is not to say implementations are prohibited from using caching mechanisms in order to improve performance. It is important however to recognise the fact that virtual views aim to dynamically construct result sets on request, rather than explicitly store and retrieve resources directly.

3.2 Support for Structural Representation

The results obtained from the evaluation of a virtual view query must be able to emulate traditional hierarchical structures. This ensures that large sets of results can be effectively managed in a manner to which users are accustomed. Typically these structures will be specifically tailored to the requirements which motivated the development of the view query rather than being a replica of the original sources. The ability to be able to navigate into sub-structures created as the results of other queries must be supported.

3.3 Storable, Reusable and Configurable Queries

It must be possible to store view queries so that evaluation can be repeated as required. A query must be reusable in the construction of new queries since there are undoubtedly situations which have common requirements and therefore common solutions. There must also be a mechanism provided which allows different parameter values and context information to be passed to generic definitions. The ability to reuse, combine and refine existing definitions will ease the construction of new virtual views.

3.4 Multiple Source Extraction

When a virtual view query is evaluated it must be capable of constructing a view which contains results obtained from more than one source. This allows virtual views to act as information aggregators, where resources from a diverse set of sources are presented in a single manageable view. Ideally the resources should be accessible from a number of distinct storage providers, including local storage, mapped network drives and Internet based documents.

3.5 Scalable Query Evaluation

Virtual views aim to rearrange and effectively manage a possibly large number of resources from a diverse set of sources. Therefore a supporting system must be designed to be scalable. This is typically achievable by applying query evaluation in a parallel fashion whenever possible. Work has been undertaken which shows how declarative type queries can be implemented using a distributed service-based architecture [14]. This work included the development of a service-based distributed query processor. This processor is capable of factoring out, as services, the functionalities related to the construction and execution of distributed query plans. Although this work was aimed mainly at data management, the same techniques could be applied to building an equivalent virtual view query execution engine. The ability to automatically distribute query execution into a service-based architecture could help address the problem of evaluating complex and computationally expensive queries.

3.6 Transitive Resource Discovery

Rather than simply identify resources that are readily available from existing file systems, virtual view queries should have the ability to identify resources via other resources. For example, a virtual view query may refer to an HTML file via a URI. The query processing engine should have the ability to process this file and identify any referenced resources, eventually adding these to the returned set. This effectively allows navigation into files as well as directories.

3.7 Resource Property Values

During query processing it should be possible to examine available property values in order to aid construction of the desired results, e.g. the type of a file. It should also be possible to associate new properties values with specific resources of interest. This ability supports the association of arbitrary meta-data with resources, thus providing a flexible mechanism for communicating with higher level applications and allowing the development of more expressive queries. The fact has been highlighted that multiple ontologies must be supported when associating meta-data with information sources [15]. Specifically, the problem of shared vocabulary across multiple ontologies must somehow be taken into account. The ability to assign extrinsic meta-data in the form of property values to information sources during the construction of virtual view queries allows this issue to be addressed, since the context in which the query is being used can help define the most meaningful ontology.

4. EXAMPLE IMPLEMENTATION

In order to help validate the requirements and to provide a test-bed for query development an initial implementation of a virtual view system was developed. This reference implementation was developed in the Java programming language to ease deployment across a number of platforms and architectures. The two primary elements of this implementation are the query processing engine (QPE) and the virtual view query language (VVQL).

The query processing engine was designed to ensure maximum flexibility within various application frameworks. This was achieved by creating a core system that contains an evaluation model which may be populated via XML. By default the results of evaluation are also generated as XML. The existence of this single core engine allows for the development of a suite of related tools which leverage the query processing capabilities in various ways. Queries are identified to the query processing engine via a URI. This supports the re-use of common queries and queries provided by 3rd parties. The queries identify the location of the various resources to be included within the generated view. The host file system and resources are interrogated as necessary and the resulting virtual views are output in an appropriate format.

One of the first additions to the core engine was the virtual view query language (VVQL) parser. This was developed using the Java Compiler Compiler (JavaCC) [16, 17]. This component allows for the development and evaluation of queries using a simple declarative language. Although the XML representation of queries represents a canonical definition, the ability to specify queries in a more concise manner drastically improves query understanding and development. Several simple *result presenters* were also developed which allow for output from the QPE in formats other than XML. This core architecture is depicted in Figure 2.

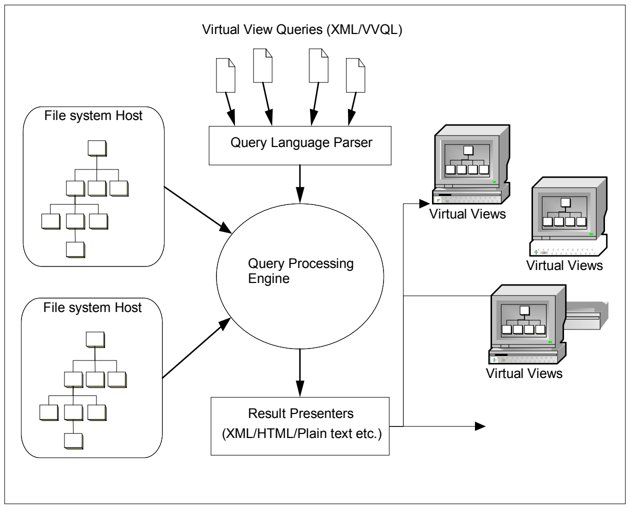

Figure 2: Virtual view implementation architecture

The first higher level tool which was built on top of the QPE and VVQL parser was a simple interactive shell. This allows for simple testing and query experimentation. This interactive shell program also provides a simple command line interface to the QPE which can be easily driven from other higher level tools. A screen shot of the interactive shell program is shown in Figure 3. Within this example the simplest possible VVQL query has been executed.

```
-------------------------------------------------------------
Virtual View Query Shell v0.1 - (c) Mark Brian Dixon 2008.
Type "exit" to quit
-------------------------------------------------------------
vvq> show *;

file:/E:/workspace/Webository/.classpath
file:/E:/workspace/Webository/.project
file:/E:/workspace/Webository/.settings/
file:/E:/workspace/Webository/.svn
file:/E:/workspace/Webository/bin/
file:/E:/workspace/Webository/Docs/
file:/E:/workspace/Webository/src/
----------------------------------------
Execution complete - 7 resources returned
vvq>
```

Figure 3: Interactive virtual view query shell

5. VIRTUAL VIEW QUERY LANGUAGE

The virtual view query language (VVQL) is a declarative language that allows for the definition of virtual views. The view definitions are constructed as query operations which are evaluated against an underlying file system, or the results of other virtual view queries. In many respects VVQL is comparable to SQL with regard to how it evaluates queries against an underlying dataset. The difference being of course that the dataset for SQL is stored within relational tables rather than within a hierarchical file system. The existence of this similarity had an effect upon the syntax design. Keywords were chosen carefully to ensure end-users did not confuse the VVQL with common SQL syntax with which they may already be familiar.

A basic query written in VVQL has the following form:

```
show <resource_definition>
[when <condition>]
[let <property_assignments>]
[<qualifiers>]
;
```

The <resource_definition> element can refer to resources directly or via nested queries and referenced queries. Direct references often include wildcards to allow for pattern matching. Indirect references allow for the re-use of existing queries. The <when> element allows conditions to be specified that filter the results obtained. The <let> element allows for the assignment of arbitrary property name and value pairs to the returned results. The <qualifiers> provide additional post-processing options such as sorting and grouping. A very simple query which creates a virtual view containing only files with a document type suffix can be specified as follows.

```
show "*.doc";
```

During query processing a current working location (CWL) is maintained which is used to give a context to any relative resource locations. Hence both absolute and relative resource references can be included within a query. The ability to specify relative resource locations is particularly important when defining generic reusable queries.

The VVQL supports the ability to combine multiple sub-query expressions. At the simplest level this allows for the aggregation of resources from multiple sources. Combination of multiple sub-queries can also be applied using set theory type operations. For instance, support for *union*, *intersection* and *difference* is provided. The most powerful combination operation however is the *sequence* operator. This allows the query language to support navigation and resource discovery through a number of arbitrary paths. This is an essential capability when considering many resources are typically stored within hierarchical structures. Queries which support recursive evaluation are particularly effective when using the sequence operation.

The VVQL support combination of sub-queries using the following form:

```
show <sub_query> {<combination_op> <sub_query>};
```

The <combination_op> element specifies the type of combination operation to be applied. In effect this acts as an operator and the surrounding <sub_query> elements act as the operands. An example of aggregation is shown below. This combines three news resources into a single view. The **plus** keyword is used to request the resource aggregation operation.

```
show "http://www.bbc.co.uk"
plus "http://www.cnn.com"
plus "http://www.independent.co.uk/news"
;
```

The sequence operation takes the output of one sub-query and uses it as the input into the next sub-query, in a manner similar to the pipes which support the Unix tool philosophy [18]. A good example of the sequence operator being applied is shown below. This recursive query may be used to flatten a hierarchy of resources into a single horizontal structure. Since this query is recursive and hence refers to itself it must be stored within a file, in this case this query is called "flat.vvq". The **then** keyword is used to request the resource sequence operation.

```
show * plus (show * when isDir() then
ref "./flat.vvq");
```

This query works by first identifying all resources within the current working location (CWL). It then identifies all directory type resources within the CWL and passes these via the sequence operation back into itself. The presence of the sequence operation effectively causes the CWL to traverse down all possible branches of the hierarchy. The order in which the resources are identified could be changed by restructuring of the query or by the use of an appropriate qualifier.

The ability to divide queries into multiple sub-queries provides an opportunity for parallel query evaluation. With the exception of being applied against the sequence operator, all sub-queries which appear as operands may be evaluated concurrently. This allows multi-threading to be utilized, which is important when accessing resources from multiple remote locations via networks. The ability to access resources concurrently reduces the inevitable latency which would occur if only sequential evaluation was possible.

The sequence combination operator partially resolves the requirement for transitive resource discovery. To fully realize support for the transitive concept however requires an additional capability within the query language. This is achieved through the provision of a *processing* token. When used this token specifies that further processing is required on the identified resources prior to them being added to the set of results. The nature of this token's behaviour led to the decision to use the @ character for this purpose. This suggests that resources located 'AT' the identified resource, rather than the resource itself, are the real items of interest.

One of the most common processing requirements using this approach is to identify resources from within HTML formatted resources. This can be done using screen-scraping type techniques as described by Schrenk [19] and Hemenway et al [20]. Although this is a fairly simple concept to understand it significantly enhances the capability of virtual views to include a much wider range of target resources from multiple locations. For example, the following query will show all resources embedded within the specified resource, rather than simply returning the resource itself.

```
show @"http://en.wikipedia.org/wiki/Fft";
```

An optional processor name may be specified following the token. If this is absent then a default processor is identified by using the type of the accessed resource to guess the most appropriate processor. In practice it may be necessary to refine the processing capabilities to filter out unwanted results. This would mean developing more sophisticated processors which had a certain level of awareness about the format of the HTML resource being processed. Although the query processor has been designed to be extendible in terms of supporting transitive processing capabilities, i.e. by providing dynamic loading of processing classes, a cleaner alternative would be to extract resources directly from a well defined publishing mechanism such as RSS feeds. This is the preferred approach since it ensures that returned resources have been specifically identified as being of public interest and grouped accordingly, which is not necessary the case in the "catch-all" approach supported by generic screen scraping. Also there are both legal and ethical considerations to take into account when using the screen scraping approach [21].

The following example would produce a list of resources extracted from the delicious.com hot list of topics.

```
show @"http://feeds.delicious.com/v2/rss";
```

Although the basic HTML screen scraping approach is rather coarse in its behaviour, it does permit an example which shows the expressive power of VVQL. This next example defines a view called "crawl.vvq" that recursively identifies all resources referenced by other resources. It is in effect a very simple web-crawler.

```
show * plus (show @* then ref "./crawl.vvq");
```

The creation of generic reusable queries require that mechanisms exist which allow a certain amount of configuration to take place during evaluation. This is supported within VVQL using a parameter passing mechanism. Actual parameter values may be passed during requests for query evaluation. These are then accessible within the query using a dollar character followed by a parameter number. For example a generic query called "types.vvq" could be defined as follows.

```
show * when type = $1;
```

This query could then be called from another query as follows.

```
show ref "./types.vvq(\"doc\")"
plus ref "./types.vvq(\"txt\")";
```

The <let> element supports the ability to assign to arbitrary property value pairs during query evaluation. Higher level tools may interrogate these property values and use them to support complex structures. For example a high level tool can attempt to read the "query" property of a returned resource, if this exists then the tool can evaluate the specified query in order to identify the contents of that resource. This provides a mechanism to support directory structures within virtual views themselves. For example a virtual view could be constructed in the following manner.

```
show "A" let query="show A*;"
plus "B" let query="show B*;"
plus "C" let query="show C*;"
…
plus "Z" let query="show Z*;"
;
```

When evaluated this query would return twenty six resources, each representing a *virtual directory*. The contents of each of these directories would then become available by evaluating the VVQL with the "query" property.

There is additional syntax supported by the VVQL which has not been covered within this short introduction. The fully defined grammar is given in the Appendix and is defined using Extended Backus–Naur Form (EBNF) [22]. A set of predefined properties and methods are available for use within filter expressions and property assignments. For example, a `System.date` property may be accessed to allow definition of queries that identify resources stamped with a specific date and time. Also the ability to include the results of nested queries within expressions is also supported.

6. DEPLOYMENT ARCHITECTURES

The underlying concepts and implementation described so far provide the basic justification and platform on which a virtual view system can be based. There is however a need to describe more abstract usage scenarios in order to identify how the virtual view approach can be practically applied in real situations. End-users after all are not going to be content with having a list of resources listed within the interactive shell. What is required is the definition of one or more deployment architectures which place the underlying platform into an application context.

From a pragmatic point of view the most obvious form of user engagement with the virtual view architecture is via a web-browser. The basic approach is to provide a web-page which allows exploring of the results obtained from evaluating predefined virtual views. Generating the results of query evaluation as HTML provides a workable solution fairly quickly. There is however a need to actually instigate the evaluation of a query in the first place. From a user's point of view this should be via a simple click of a link within the browser. At the server side an appropriate evaluation can then be performed using any one of the many technologies designed to support dynamic user interaction. Any common server side technology could be used including PHP, ASP.NET, Perl or Python. Within this project however the Java Servlet [23] infrastructure was used, since it was a fairly simple exercise to wrap the Java based QPE within a Servlet. The architecture of this particular deployment model is shown in Figure 4.

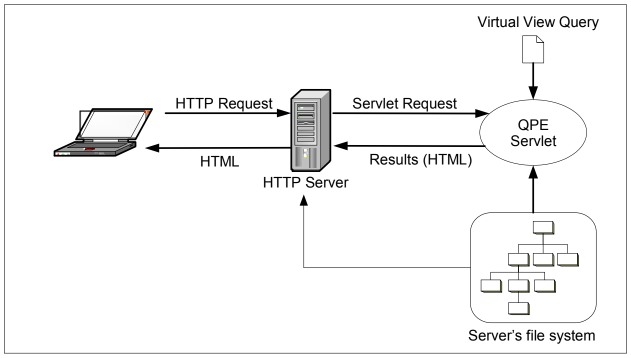

Figure 4: Server side deployment architecture

Placing the entire virtual view infrastructure at the server side does of course ensure a low cost of entry for potential users. It also provides an opportunity for information publishers to provide virtual views of their resources with very little overhead.

An alternative approach to supporting user interaction is to provide a client side standalone graphical file manager type application. Such an application is essentially equivalent to the multitude of available file managers bundled with OS desktop environments. The difference being of course that information regarding the displayed files and directories is actually derived at run time via query execution, rather than being accessed from the underlying file system. When a virtual view query file is accessed via an existing file manager, file type associations may be used to execute the virtual view file manager to allow navigation through the virtual view. A prototype virtual view file manager application has been built. This approach has not been pursued to any great length however since it has been identified that the tighter integration of a virtual view file system into the host OS may make such an application redundant. This suggestion is discussed further within the conclusions section.

One of the main problems with both of the suggested deployment architectures is that of resource accessibility. Resources which are located on file systems directly accessible by the QPE are easy to identify. Also resources which are identified through a fully defined URI are also typically accessible via the specified protocol. The problem appears however when attempting to identify groups of resources by using pattern matching of wildcards. Many common protocols such as HTTP [24] do not support such operations. i.e. a web server typically does not allow the entire contents of a directory to be listed.

There are a few ways of solving this problem. One approach could be to interrogate remote file systems via the Java Servlet mechanism described above. Although this is perfectly feasible a more robust approach is to deploy remote server applications which are capable of accessing the required file systems directly. The application to achieve this already exists of course, in the form of the Query Processing Engine. Therefore a simple protocol has been developed which allows the driving of the QPE via a network connection. This type of architecture, as shown in Figure 5, effectively creates a network of communicating VVQ servers, further addressing scalability issues of large queries. As a query is being processed any reference to remote file system contents not directly available through the standard protocol may be passed for evaluation to a remote VVQ server. This is achieved by stripping the appropriate parts from a URI in order to identify the remote server host.

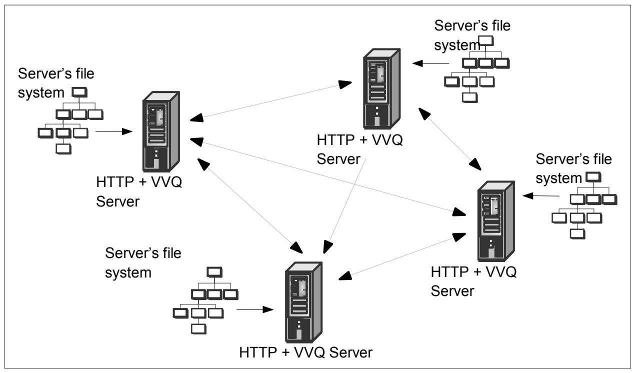

Figure 5: Multiple VVQ server architecture

Any machine which provides a VVQ server is essentially granting permission to external users to create their own personal views of the machine's file system. Hence, it is of course necessary to restrict the part of the file system which can be actually accessed by the VVQ server, otherwise the contents of the entire file system could be discovered. In practice this is achieved by the setting of a ROOT mount point in a similar way to a typical http-server. Also it should be noted that a VVQ server only provides a list of available resource URIs rather than the resources themselves. It therefore only makes sense to provide access to resources which may be ultimately accessible via another protocol such as HTTP.

Both the QPE wrapper Servlet and the multi-threaded QPE server were developed by extending the underlying core engine and parser. Most of the code used within the different deployment architectures is therefore shared. Other deployment architectures, which have not yet been developed, are also clearly possible. These include a web-based application which allows the construction of virtual view queries via higher level mechanisms. Additionally more user friendly presentation skins could be developed using technologies such as Adobe Flash, JavaFX or Silverlight which display resources in a variety of ways.

7. CONCLUSIONS AND FUTURE WORK

The successful implementation of an example system which provides a usable test-bed clearly indicates the viability of the virtual view approach. This paper has only scratched the service in terms of what is achievable. The VVQL is a simple yet powerful language that is capable of expressing extremely complex semantics in a very concise manner. These capabilities combined with the multitude of possible deployment architectures means that there are probably many other possible applications. Also the development of related products such as browser plug-ins which are VVQ protocol aware would encourage user interaction.

The ubiquitous presence of query processing aware servers across large public networks such as the Internet would provide an environment in which users could navigate 3rd party file systems in a manner chosen by themselves, rather than the provider of the file system. This would allow a consistent view of information to be obtainable no matter how or where it was stored.

A key future aim of this project is to more closely integrate the QPE into host operating systems. This would allow virtual views to appear as part of the native file system. Initial work has already begun to create an integration of the QPE into the Linux OS using the FUSE API [25], this however makes it necessary to redevelop the QPE as a C language application. The MS Windows Explorer application is also extendible through public interfaces [26]. Linking the QPE to these extension points will allow Windows OS users to browse virtual views in exactly the same way as the host file system.

As the QPE implementation matures it may be possible to develop caching mechanisms which allow virtual views to evaluate complex queries extremely quickly. If this was developed however it should be done in a manner which does not impact the performance of a user's individual machine. Hence, a caching system is a valid extension providing it runs independently on a server.

The use of virtual views could also be adapted to help automatic generation of RSS feeds. This would involve the fairly simple development of a new results presenter component which maps the XML output into a form suitable for RSS aggregator processing.

Finally the creation of development tools which allow for the writing, profiling and debugging of virtual view queries needs to be performed. The intention is to provide an Eclipse IDE plug-in [27] that supports such features along with syntax aware editors.

The philosophy behind the creation of the virtual views approach was to place the ownership of resource presentation with the user rather than the provider. Virtual views provide the possibility of supporting multiple "role" centric views of electronic resources. Hence, the term *polymorphic file system* probably most concisely describes the overall concept.

8. ACKNOWLEDGMENTS

The author wishes to thank Prof. Colin Pattinson and Julian Old for their very useful feedback during this research project and the writing of this paper.

9. REFERENCES

[1] Berners-Lee, T., Fielding, R. and Masinter, L. 2005. Uniform Resource Identifier (URI): Generic Syntax. Request for Comments (RFC) 3986, STD 66. http://tools.ietf.org/html/rfc3986 Accessed 11 April 2011.

[2] IEEE. 2004. The ln Utility, The Open Group Base Specifications Issue 6: IEEE Std 1003.1, http://www.opengroup.org/onlinepubs/009695399/utilities/ln.html

[3] Kleiman, S. R. 1986. ''Vnodes: An Architecture for Multiple File System Types in Sun UNIX'', in Proceeding of the USENIX Association Conference, Atlanta, Georgia, pp. 238-247.

[4] Apple Incorporated, 2009. Mac 101 : Spotlight. Article: HT2531. http://support.apple.com/kb/HT2531

[5] Microsoft Corp. 2011. Windows Search 4.0. http://www.microsoft.com/windows/products/winfamily/desktopsearch

[6] Chirita, P-A., Costache, S., Nejdl, W. and Paiu, R. 2006. Beagle ++: Semantically Enhanced Searching and Ranking on the Desktop. The Semantic Web: Research and Applications Lecture Notes in Computer Science, Volume 4011/2006, 348-362, DOI: 10.1007/11762256_27.

[7] NotePage inc. 2010. RSS Specifications, http://www.rss-specifications.com/rss-specifications.htm, Accessed 11 April 2011.

[8] Zadok, Erez and Nieh, Jason. 2000. "FiST: a language for stackable file systems", in Proceedings of the annual conference on USENIX Annual Technical Conference. USENIX Association Berkeley, CA, USA.

[9] Gifford, D. K., Jouvelot, P., Sheldon, M. A., and O'Toole jr., J.W. 1991. "Semantic file systems", In Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91) (Oct.), ACM, pp. 16–25. Doi: 10.1145/121133.121138.

[10] Padioleau, Y., and Ridoux, O. 2003. "A logic file system", in Proceedings of the 2003 USENIX Annual Technical Conference (San Antonio, TX, June 2003), pp. 99–112.

[11] Bowman, C.M., Dharap, C., Baruah, M., Camargo, B. and Potti, S. 1994. "A File System for Information Management", in Proceedings of the Conference on Intelligent Information Management Systems. June. Washington, DC.

[12] Leung, A.W., Shao, M., Bisson, T., Pasupathy, S., and Miller, E.L. 2009. "Spyglass: fast, scalable metadata search for large-scale storage systems" in Proceedings of the 7th conference on File and storage technologies. pp. 153-166. USENIX Association Berkeley, CA, USA.

[13] Brandt, S., Maltzahn, C., Polyzotis, N. and Tan, W-C. 2009. "Fusing data management services with file systems", in Proceedings of the 4th Annual Workshop on Petascale Data Storage. pp. 42-46. ACM New York, NY. doi:10.1145/1713072.1713085.

[14] Lynden, S., Mukherjee, A., Hume, A.C., Fernandes, A.A.A., Paton, N.W., Sakellariou, R. and Watson, P. 2009. The design and implementation of OGSA-DQP: A service-based distributed query processor. Future Generation Computer Systems. Volume 25, Number 3, pp. 224-236. Elsevier. doi:10.1016/j.future.2008.08.003.

[15] Mena, E., Illarramendi, A., Kashyap, V. and Sheth, A.P. 2000. OBSERVER: An Approach for Query Processing in Global Information Systems Based on Interoperation Across Pre-Existing Ontologies. Distributed and Parallel Databases, Volume 8, Number 2, pp. 223-271, DOI: 10.1023/A:1008741824956 . Springer.

[16] Java.net. 2009. Java Compiler Compiler - The Java Parser Generator, Version 5.0, https://javacc.dev.java.net Accessed 11 April 2011.

[17] Copeland, T. 2009. Generating Parsers with JavaCC, second ed., Centennial Books, Alexandria, VA.

[18] The Linux Information Project (LINFO). 2006. The Unix Philosophy: A Brief Introduction. Aug. http://www.linfo.org/unix_philosophy.html, Accessed 11 April 2011.

[19] Schrenk, M. 2007. Webbots, Spiders, and Screen Scrapers. No Starch Press. San Francisco.

[20] Hemenway, K and Calishain, T. 2004. Spidering Hacks: 100 Industrial-Strength Tips & Tools. O'Reilly.

[21] Adler, K.A. 2003. "Software Helps Users Access Web Sites But Activity by Competitors Comes Under Scrutiny", New York Law Journal, June 9th, American Lawyer Media. Available via reprint.

[22] ISO. 1996. Extended BNF, Syntactic Meta-language, http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip. ISO/IEC 14977. Accessed 11 April 2011.

[23] Coward, D., Yoshida, Y. 2003. JSR-000154 Java Servlet 2.4 Specification, Java Community Process, http://jcp.org/aboutJava/communityprocess/final/jsr154/index.html Accessed 11 April 2011.

[24] Network Working Group. 1999. HTTP/1.1, June. http://www.w3.org/Protocols/rfc2616/rfc2616.html Accessed 11 April 2011.

[25] Source Forge. 2010. FUSE: Filesystem in User*space*. http://fuse.sourceforge.net/ Accessed 11 April 2011.

[26] Rensin, D. 2006. Windows Shell: Create Namespace Extensions for Windows Explorer with the .NET Framework, MSDN Magazine, http://msdn.microsoft.com/en-us/magazine/cc188741.aspx Accessed 11 April 2011.

[27] Clayberg, E. and Rubel, D. 2006. Eclipse: Building Commercial-Quality Plug-Ins, 2nd Edition. Addison-Wesley.

APPENDIX

```
(* Virtual View Query Language, EBNF Grammar *)
(* (c) 2008-2011 M B Dixon *)

rootQuery = query , ";" ;

query = "show" , subQuery { ("plus" | "then" |
"union" | "intersect" | "minus" ) , subQuery } ;

subQuery = ["unique"] , [ "@" , ["pi:" , name] ]
, ( resourceQuery | nestedQuery | refQuery ) ,
["when" , expression] , ["let" ,
propertyAssignment , {"," , propertyAssignment}]
, {qualifier} ;

resourceQuery = ( queryURI | "*" ) ;

nestedQuery = "(" , query , ")" ;

refQuery = "ref" , queryURI , [ "(" , paramList
, ")" ] ;

qualifier = ( "sort_by" | "group_by" ) , "(" ,
paramList , ")" ;

propertyAssignment = name , "=" , expression ;

expression = subExpression ,  {binaryOperator ,
subExpression} ;

subExpression = ( integerNumber | realNumber |
char | string | "true" | "false" | date | "(" ,
expression , ")" | "(" , query , ")" |  ("!" |
"-" | "~") , subExpression |  "$" ,
[integerNumber] | "null" | "system" | name , [
"(" , paramList , ")" ] ) ;

date = ? any valid US format date ? ;

paramList = [ expression , {"," expression} ] ;

binaryOperator =  ( "+" | "-" | "/" | "*" | "%"
| "&&" | "||" | "<=" | "<<" | ">=" | ">>" | "=="
| "!=" | "<" | ">" | "&" | "|" | "^" | "." |
"in" | "like" | "regexp" ) ;

name = ? any letter ? , { ? any letter ? | "0" |
digit } ;

queryURI = string ;

string = '"', {? all visible chars ?-'"'}, '"' ;

char = "'", ? all visible char ? - "'", "'" ;

integerNumber = ( "0" | digit , {("0", digit)} )
| ( "0x" , ("0" | digit | hexDigit), {("0" |
digit | hexDigit)} ) ;

realNumber =  ( "0" | digit ) , {("0", digit)} ,
"." , ( "0" | digit ) , {("0", digit)} ,
[exponent] ;

digit = ("1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9") ;

hexDigit = ("a"|"b"|"c"|"d"|"e"|"f") ;

exponent = ( "E" | "e" ) , [ ("+" | "-") ] , (
"0" | digit ) , {("0", digit)} ;
```