Versus: a Model for a Web Repository

João P. Campos Mário J. Silva

XLDB Research Group
Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749-016 Lisboa, Portugal
[jcampos,mjs]@di.fc.ul.pt

Abstract

Web data warehouses can prove useful to applications that process large amounts of Web data. Versus is a model for a Repository for Web data management applications, supporting object versioning and distributed operation. Versus applications control the distribution, and the integration of data. This paper presents the design of Versus and our prototype implementation.

Keywords: Web data repository, versioning, distributed database.

1 Introduction

The Web is a great personal enhancement tool, but the amount of data available is so vast that its true potential can only be harnessed with tools specialized in aiding users find, sort, filter, summarize and mine this data.

To handle large amounts of information, applications need bandwidth. With today's limitations, applications wouldn't be able to solve user queries in due time, because it would take them too long to download the data.

Pre-fetching the information (anticipating user interaction) and storing it would be a reasonable solution: getting a copy of all the needed information is very expensive (both on time and bandwidth usage), but saved data can then be reused by several applications and users.

A Web robot can be used to seek, download and store large portions of Web contents. However available Web robots are either expensive and proprietary [1, 8], outdated [9], or both [16]. Solutions for storing collected Web data are tightly coupled with the robots used, and, being proprietary, are not readily available for usage by other applications.

In addition, to efficiently implement Web applications that deal with Web data, we may need scalable storage, capable of holding large amounts of data, with a high throughput.

The motivation for this work is that we couldn't find a storage offering high performance meta-data management (like serverless filesystems [2] do for data) with an interface to manage web meta-data. Our goal is to provide support for automatically perform the following functions:

Retrieval of large quantities of data from the Web. This may represent a huge computational effort, requiring advanced techniques to address scale problems. Applications retrieving and saving data are usually built tightly coupled with the storage system used. Hence, the storage framework for Web data should be highly scalable, allowing the distribution of the loading processes among a network of processors.

Manage meta-data about Web resources. Most applications built on Web data require both the documents retrieved from the Web and the meta-data available about these documents, such as the URL where the document was retrieved, its last modification date, or MIME-type. The storage system must provide methods for storing and retrieving these meta-data elements along with the documents.

Save historic data. History may be relevant. While some applications won't care about old unavailable documents, some others might be interested in looking at how a portion of the Web was some time ago. The storage must provide access methods enabling user applications to specify what they want to see in respect to time.

This paper presents an implementable model for a Web data repository satisfying these functional and architectural requirements and the implementation of a working prototype that serves as its proof of concept. Versus is the name used for the model developed for storing and managing Web data. In the text, we also designate the developed prototype system as Versus. The paper is organized as follows: next section presents some work related with Versus; section 3 presents the Versus model for a distributed repository; section 4 details our prototype implementation and section 5 presents the conclusions and future work.

2 Related Work

Version models are a powerful means of representing evolution of objects over time. The emphasis on versioning systems research was on supporting Computer Aided Design (CAD) systems. The design process is slow: complex objects are developed by teams of designers, each of whom designs independent parts. Parts are integrated to form the whole. Eventually some parts are redrawn and some parts are reused from previous projects.

Web data collection is similar to CAD engineering design: data is collected at different times (due to bandwidth constraints) and may be related (through the link structure) or integrated with other data to form complex objects, like pages or sites. Some parts (pages) of the collected Web may be revised, recollected and related with old (already stored) parts. The Web grows everyday, revealing new pages to integrate in the global picture.

Version models provide semantic extensions to support the organization of engineering data [14], including unified concepts for managing and structuring information changing over time. Versus uses some of the defined concepts, such as workspaces, versions and check-out/check-in operations.

Web-based Distributed Authoring and Versioning (WebDAV) is a set of extensions to the HTTP protocol that enable users to collaboratively edit and manage files on remote Web servers [18, 13]. WebDAV implements long lasting locks, preventing two users from writing the same resource without merging changes. WebDAV servers are not designed for holding the amount of data we aim to hold with Versus. A WebDAV interface could, in principle, be developed for Versus.

Web repositories are data stores designed to hold Web data. Most were developed to support search engines, storing the data needed to build indexes or compute rankings. Some implementations hold large portions of the Web, and their architecture is designed to hold the entire visible Web. WebBase [10] is a repository of web pages designed for maintaining a large shared repository of data downloaded from the Web. The main focus of WebBase is optimizing data access, storing all the meta-data in a separate database management system. From our experiments we found that meta-data management can be a bottleneck to the system performance. We couldn't find details about how WebBase manages meta-data other than it is saved on a relational database (is it centralized or distributed?) WebBase is specifically tailored for supporting a Web crawler.

AIDE [7, 3] is a difference engine that allows users to track changes on Internet pages; WebGUIDE [6] is a system for exploring the changes storage system, offering a navigational tool to analyze the differences in Web pages over time. The difference engine is supported by a centralized versioning repository that stores versions of documents so that they are available for comparison in the future. Data is saved in this repository in Revision Control System (RCS) [17] format. Meta-data is saved in a relational database.

The Internet Archive [12, 4, 15] goal is to build an Internet library for offering access to collections in digital format. The main focus is on long term preservation of selected contents and offering access to collected items.

We have presented other research on topics related to Versus. A comparison between Versus and the systems presented is out of the scope of this paper and is of little practical interest as they all have a small overlap with Versus with respect to functionality.

3 Versus Model

Processing large quantities of information in a Web data-warehouse involves integration of data from multiple sources, indexing, summarizing and mining Web data. The key for scaling up these heavy data processing operations lies in distributing the load among several processors, parallelizing the tasks to perform. However, this distribution must be supported by a storage system that can cope with the new complexities introduced, such as partitioning the work into units, physical distribution of data and scheduling work units among the processors, provisioning of methods for accessing distributed data and, finally, the fusion of the independently processed parts to form coherent views.

Our approach is based in a versions and workspaces model for data, enabling parallelization of applications processing large collections of Web pages. Versus follows this approach, supporting concurrent updates, versioning and distribution.

3.1 A Typical Usage Scenario

One example of an application with high data interaction is a distributed Web crawler. In a typical implementation, each thread, running on a separate processor, is responsible for collecting documents from certain parts of the Web; in the end, the crawler delivers an integrated archive with the collected documents. The running context of such an application is depicted in Figure 1. Each thread, when initializing, would get from the storage server the roots of the crawl (the pages where to start looking for links). Crawling the Web consists of iteratively downloading pages, extracting the links referencing other pages, downloading these pages and so on. During the crawl, threads would exchange data through the repository's storage server to ensure that each document is not processed more than once. When each of the threads finishes, it uploads the documents obtained to the repository's storage server, making them available to other applications.

3.2 Requirements

We identified the following main requirements for a web data repository:

- Support partitioning of the work into disjoint units that can be processed concurrently;
- Support concurrent updates to disjoint subsets of the data;
- Support integration of results from processed units;
- Enabling threads working on separate units to exchange information so that applications can avoid duplicate processing;

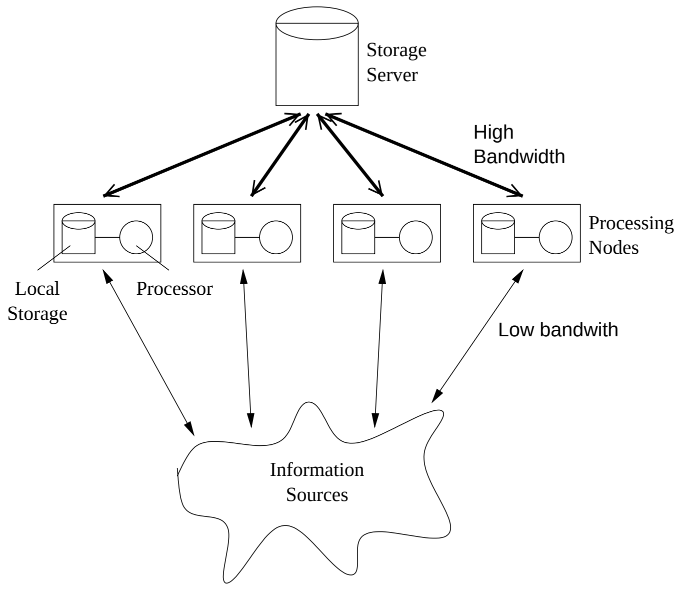

Figure 1: Running context of applications using Versus. The *storage server* holds data to be shared by the several transactions of the running application. Each transaction runs in a *processing node* and has an associated storage, where data processed locally is kept. The time lost in data transfer between the processing nodes and the storage server is compensated by parallel data processing.

- Support storage of large amounts of data, ultimately archiving a very large portion of the entire Web;
- Enable reading of stored information while other transactions process updates;
- Support periodic partial updates to stored information, refreshing stale data items while maintaining their relationships to other items;
- Reuse storage when new documents are equal to a previously collected version;
- Enable views over past states of data, providing the time dimension in stored data.

3.3 Assumptions

The design of Versus is based on the following assumptions:

1. Information spaces can be partitioned into disjoint subsets that can be processed with a high degree of independence;
2. The performance overhead introduced by intra-thread communication for synchronization of the non independent part of the computation is largely compensated by the parallel execution of the threads.
3. Applications provide the repository with a function to partition the data into processing units and a function to reconcile conflicting data generated within different units.

Independence among working units is application specific. Assumption 2 implicitly states that Versus is most suited for applications that can profit from parallel processing.

3.4 Concepts

We now present the main concepts of the Versus model.

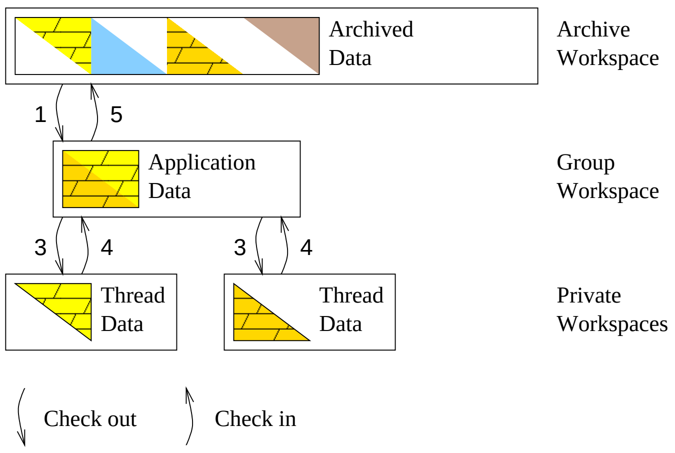

Figure 2: Versus supports three classes of workspaces: archive, group and private workspaces. The figure depicts an archive workspace holding a data set partitioned in several subsets. Applications check-out to the group workspace only the data sets they will use. Threads concurrently check-out subsets of the data, process them, and check them back in.

3.4.1 Workspaces

Workspaces are well bounded and independent environments where application threads can apply transactions to subsets of the data to be processed, minimizing interaction with other data subsets being processed by other clients.

We define three kinds of workspaces: private, group and archive.

Private workspace: provides storage to application threads. Private workspaces are independent of one another, and may reside in different processors. Each parallel thread that accesses the repository and generates results for an application should instantiate a private workspace of its own.

Group workspace: integrates partial results generated by clients on private workspaces. Each application (possibly with several threads of execution) processing archived data should instantiate a group workspace. Conflicts may arise when consolidating data from several private workspaces into a group workspace. Versus handles the conflicts using the methods provided by the application that generated them.

Archive workspace: stores data permanently. It keeps version history for the data and is able to reconstruct earlier views of data. The archive workspace is an append only storage: data stored in the archive workspace can't be updated or deleted.

Data is passed from one workspace to another via *check-out* and *check-in* operations through the following steps (see figure 2):

1. When an application is started, it instantiates a new group workspace, checking-out data it will need from the archive workspace;
2. The application forks n parallel threads;
3. Each of the parallel threads starts its own private workspace and checks out one of the data subsets;
4. When finished with one subset, the thread checks in the results into the group workspace and restarts with another data subset;

5. When the application finishes, the results in the group workspace are checked-in into the archive workspace.

3.4.2 Layers

Versus sees its data as a collection of objects that can be versioned, organizing them in layers. A *layer* is a storage unit capable of holding one single version of each object stored in a workspace. Each workspace may contain objects from several layers.

Each workspace has an *active layer*. All objects that are added to the workspace are associated to the active layer. Workspaces can't save objects in layers other then the active layer.

A Versus repository may be set to increment the active layer number automatically or manually. If set to automatically increase the layer number, the current layer is incremented whenever a new version of an object that already exists in the current layer is added; then the new version is added to the new layer. If the repository is set to manually increment the current layer number, then any addition of an object that already exists in that layer is denied and an error is raised.

Layers are represented by integers monotonically incremented in a repository, they store the time dimension of data, showing the partial order of object manipulation operations within the repository; for example, in a manually incremented repository, one application knows that any two objects stored in the same layer are contemporary, meaning that they were both inserted into the repository when that layer was the active layer.

3.4.3 Versions

Version models allow the storage of several instances of the same objects as saved in different instants over time. This is very useful for storing the evolution of the state of objects, enabling applications to see views of the represented world at different points in time. As the Web can't (and shouldn't) be represented at once, saved representations of it can't be easily refreshed. The application of the version model to Web data is very useful because it allows the refreshing of parts of the represented data known to be stale, maintaining coherence between fresh and non fresh data. Furthermore, applications can choose to work with different views over data: for instance, a search engine built on top of the repository may use the latest available version of each document, while a web difference engine can choose to read all versions of a document to track how it was changed.

Versus assumes that if any two versions have the same id, then they both are versions of the same object. As all versions have an associated layer number, which is unique for every version of any given object, two versions of one object have distinct layer numbers, and the order of the layer numbers can be used to derive which of these is the oldest version.

3.4.4 Objects and Associations

Versus is designed to process webs of objects that can be viewed as labeled graphs, where nodes are object instances and edges are associations between them. Edge labels denote association types.

Objects saved in a Versus repository are modeled as having an associated name, a property set and a stream of data. Streams of data are to be saved in a filesystem, and their management is external to Versus. An object o is represented in Versus as a tuple $o(name, \{properties\}, stream)$. The object name is the identifier of an object and can't be changed.

Objects may be related to each other by oriented, typed associations, modeling the rich associations that exist in the real world between objects. A relationship R of type t from object

a to object b is represented as a tuple $R(a, b, t)$, where a is the anchor of the relation, b its target and t is the association type.

3.4.5 Partition and Data Units

A *partition* of a workspace is defined as the division of the workspace into disjoint subsets. We call each of the subsets forming the partition a *strict data unit*, or simply a strict unit.

3.4.6 Predicates

When checking out data from one workspace to another, applications specify the disjoint subset of the data (objects and versions) to be checked out.

If applications had to enumerate the objects to check-out one by one, they would have to know in advance the objects' identifiers. This may turn out impossible to some applications. In Versus, applications specify sets of objects to check-out using predicates. A predicate is a function $Pred_A$ that, given an object o returns *true* when o belongs to A.

$Pred_A$ is not a *belongs-to* operation. The application of a predicate to all objects in the workspace defines the unit. On check-out, the repository tests the supplied predicate against candidate objects and returns those that satisfy the predicate. For instance, if one thread wants to perform a transaction on all objects whose identifier starts with letter d, it provides a function to the repository that returns *true* if an object starts with d and *false* otherwise. The repository then evaluates that function on all objects of the workspace to find out which are to be checked-out.

Predicates must be defined by Versus applications because only applications have the knowledge of how their data can be processed in independent subsets.

Predicates defined over one workspace must obey to two invariant conditions:

1. No object in a workspace can satisfy two different predicates simultaneously.
2. Every object in a workspace will satisfy a predicate for the lifetime of all applications that operate on the workspace.

Invariant 2 implies that predicates can't depend on object attributes that are updated by the application and should be functions of object properties that are invariant (such as the name).

3.4.7 Strict Data Units

A strict data unit represents a set of data that can be checked out by a transaction. Partitions vary according to the predicates given. As predicates are application defined, the size of the data units is application dependent. The minimum check-out granularity is ultimately a single object. Invariant 1 implies that data units defined by a partition are disjoint. The union of all strict data units in a workspace always represents a set of objects contained in the workspace.

3.4.8 Working units

A working unit is a container used to check-out a strict data unit from one workspace to another and checking the results of the operations executed on the objects of the working unit back in.

A valid working unit definition would consist in creating a data unit for each letter and making all objects whose first letter of their identifier match the working unit letter part of the corresponding data unit. This definition would always generate 26 working units (one for each letter), independently of being applied to an empty workspace or to a workspace with thousands of objects to partition. This working unit definition complies with both repository invariants:

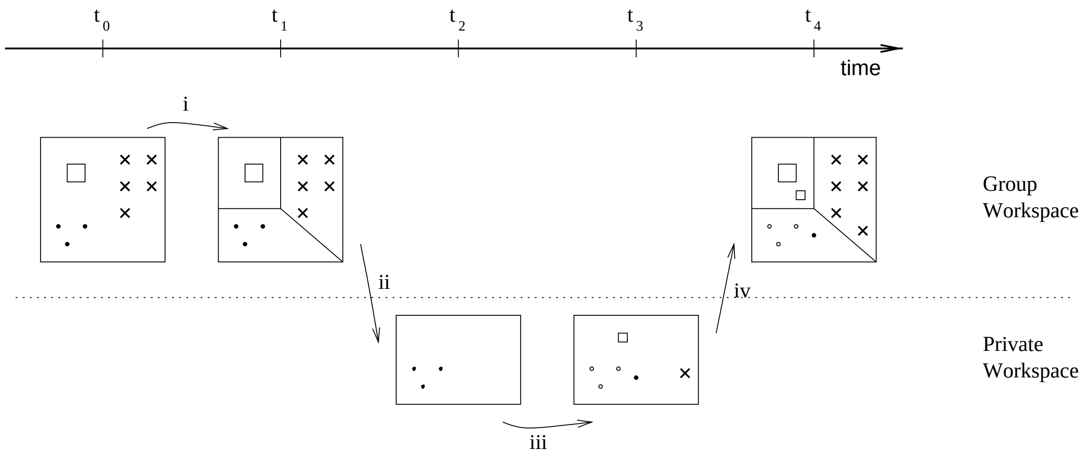

Figure 3: Illustration of data processing within Versus. Initially, at t_0, there is a group workspace (square) with objects of different shapes. Then, the following operations are performed on the workspace: *i)* application of a partition based on object shape (represented by the lines that partition the square in three parts); *ii)* check-out of the working unit containing the circles data unit; *iii)* private workspace objects are updated and three new objects (a circle, a cross and a square) are inserted; *iv)* data is checked back in the original group workspace.

an object with a given identifier will only match one starting letter; and as the identifier will always have the same first letter, it will always belong to the same data unit and will be always checked out to the same working unit.

3.4.9 Loose Data Units

A loose data unit is a strict data unit plus all objects for which there is a relationship between versions belonging to the strict data unit and other versions added to the working unit. Objects can only be added to a working unit if they satisfy the predicate defining the strict data unit that originated it, or if they are directly related with objects in the strict data unit.

Figure 3 represents the relationship between working units and workspaces. At check-out, a working unit is identical to the strict working unit: all the checked out objects satisfy the predicate originating the unit. At check-in, there might be objects (like the new square in the example of Figure 3) that don't satisfy the predicate (*"is a circle?"* in the example). Loose data units are the data units in this condition.

3.5 Operations

So far we have seen that, to update an object, an application checks out the working unit that contains the object into a private workspace, modifies the object and then checks the working unit back in. The intuition behind this mode of operation is that if we have a massive processing on a large collection of objects, we can make it concurrently by copying the objects into separate data stores, have them manipulated while isolated from the collection, and then reconcile them with the collection. We now present the semantics of these operations on workspaces.

3.5.1 Operations on data and conflict generation

Addition of new objects to a working unit while isolated would be very restricted if this were possible only with objects within the strict data unit checked out. For example, consider a crawler collecting pages from the Web, working on a private workspace that checked out a working unit for all objects of a given site; if, when downloading one of the Web pages it finds a link to some page on another site, how would it save that reference? Not within the partition, because it doesn't satisfy the predicate. It would not be able to check-out the proper working unit either, because it can't handle two working units at a time.

To mitigate this problem, we allow for data that doesn't belong to the current strict data unit (the one previously checked out) to be conditionally inserted within the working unit, enlarging it into a loose data unit. Insertion is allowed for objects that, albeit not belonging to the strict working unit, are directly associated with objects that are within the strict working unit. On the other hand, insertion is always allowed for objects belonging to the strict working unit.

Inserting or updating object belonging to the working unit does not generate conflicts as objects in data units are checked out to one workspace at a time, no two parallel threads concur to use the same objects. However, conditional insertion of objects that don't belong to the strict working unit may generate conflicts, because two parallel processes might insert the same object while isolated. When reconciling the data, conflicts must be automatically resolved by an application-supplied code.

3.5.2 Check-out

Transactions check-out a data unit from one workspace, called the source workspace, into another, called the target workspace. They determine what to check-out by applying the predicate associated with the working unit to the source workspace.

The check-out operation for a given unit defined by a predicate takes one argument, the source workspace to check-out, and generates two workspaces: the source workspace after the check-out and the target workspace.

The check-out operation is defined only if the unit to check-out is not already in use. The only modification to the source workspace is that the unit is added to the set of units currently in use. The target workspace will contain all the objects of the source workspace that satisfy the predicate, plus all the relationships from the source workspace where both the anchor and target objects are checked out.

Check-out doesn't copy relationships from objects that belong to the checked out unit to objects that don't belong to the corresponding unit at the target workspace. Hence, threads working on the target workspace don't have access to these relationships. Applications that require access to these relationships should define a partition that generates units big enough to contain them.

Transactions can only check-out data from one working unit at a time. As check-out doesn't copy relationships involving objects outside the strict data unit to the more private workspace, applications operating on the private workspace will only see relationships among objects in the workspace.

3.5.3 Conflict resolution

Implementation of a conflict resolution policy in the repository would force all the applications to use it, even if it is not appropriate to their needs. To satisfy the specific needs of Web applications the model defines a conflict manager interface that applications must implement to solve the conflicts while saving conflicting data.

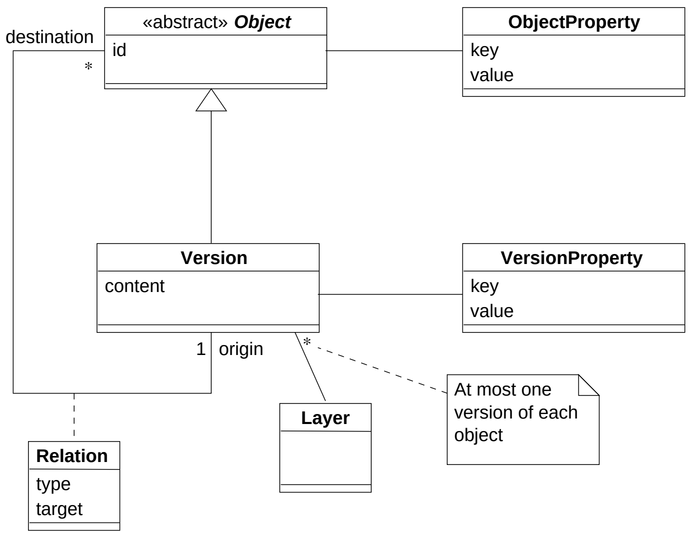

Figure 4: Class model for the data handled by the repository.

Versus applications must implement a conflict management function, that, given two candidate objects, decide which should be saved in the repository. The result may be one third object generated by merging the two candidates. The decision is application driven.

3.5.4 Check-in

Applying an operation to the data in one workspace is equivalent to partitioning the workspace, checking out each of the data units, applying the operation to each of the working units and then checking them in. As this is true for operations that don't need to see relations between objects in different partitions, the repository is suited for serving applications for loading large amounts of data, allowing the parallelization of the process.

The reintegration of working units' data previously checked out from a workspace has to consider the existence of new data that might conflict with the already existing data.

Check-in is a function that takes two workspaces W' and W_x and returns a third workspace, resulting from checking W_x into W'. Its effects are:

1. The resulting set of objects consists of those objects created before the check-out plus:
   - Objects created before check-out that belong to the strict unit, but were updated during isolated operations;
   - Objects identified by the resolution of conflicts between new objects and those that existed before and don't belong to the unit;
   - The remaining objects, those created after check-out, that satisfy the predicate.
2. The resulting relationships are all the relationships that existed before the check-out, minus relationships from updated versions, plus new relationships.
3. The lock created when the unit was checked out is released.

3.6 Data Model

Figure 4 shows the UML class model of the data handled in a Versus repository. We have the following main classes:

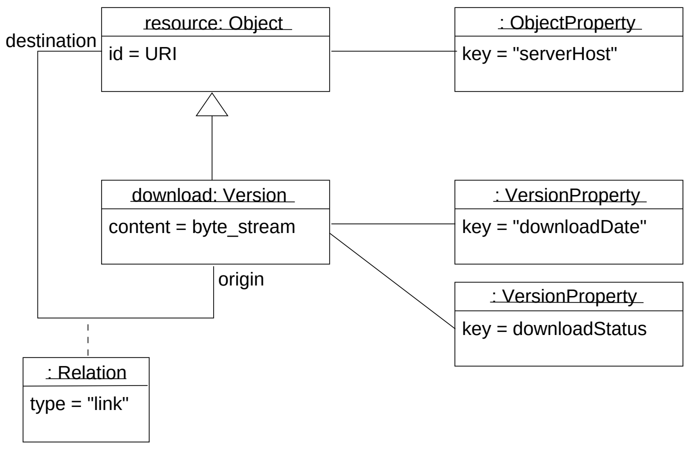

Figure 5: Simple model of the Web represented using the class model of figure 4.

- A *version* is a snapshot of an object at a given instant in time. It is saved within a repository *layer* and its *contents* represent the state of the object. An example of a version is a downloaded web page.
- A *layer* is a storage unit capable of holding, at most, one version of each object.
- *VersionProperties* are meta-data properties of this version of the object. Each property has a *key*, its name, and a *value*. An example of a version property is the download date of a Web page.
- *Objects* are an abstraction to aggregate all versions of a given real world object.
- *ObjectProperties* are properties common to every version of the same object.
- *Relations* are directed associations from instances of objects to objects or to their instances. If the target field of the relation object is void, the target of the association is any instance of the object, to be chosen by the application. If the target of the relation is a version identifier, the target is that version.

This class model is very general and enables the management of several kinds of saved data. A simple model for data downloaded from the Web can be represented using the Versus class model, as shown in Figure 5:

- A *resource* is an object reference it doesn't contain any data, it is just a pointer to a place on the Web;
- A *download* represents the act of attempting to download the resource. If the download results in a stream of bytes, that stream is saved in the *contents* field;
- Each download attempt has a download status, which is the log of the result: successful, failed, temporary unavailable, and so on. That status may be recorded in the value field of property *downloadStatus*; similarly, the *download date* can be recorded in the corresponding property.
- *Resources* can have properties that don't change between downloads, like *serverHost*, which represents the host that serves the resource.
- Downloaded data may reference other downloaded data using HTML links. These are represented as *relations*. If there is a link from a downloaded resource to a resource not yet downloaded, the target attribute remains void; if both resources are downloaded, then the target attribute is filled with the target download of the link.

Applications using this model can extend it to their own needs for saving whichever meta-data they need by implementing additional properties.

3.7 System Architecture

The repository interface was designed as a library that runs within the client application process: there is no process running "a repository server".

To enable users to build applications Versus provides an API, specifying the interface that applications must implement to access the repository to distribute their operation using the Versus model.

Calls to repository operations result in operations on meta-data, on data, or both. Data is stored in a filesystem, visible by all the workspaces: we assumed that other distribution techniques could be applied to data management (such as distributed file systems, or serverless file systems). As a result, from the versus perspective, all workspaces (private, group or public) share the same data store. Meta-data is stored in relational databases. Both data and meta-data storage can be supported by external processes.

A generic system architecture diagram, illustrating the duality between the data and meta-data management is shown in Figure 6. Meta-data is stored in relational databases. As we want to distribute meta-data management through several processors running the same application, we must have, at least, as many databases as processors. Data is shared in one filesystem.

As a workspace is the basic distribution unit, every workspace must have its own database. A workspace may be visible from several applications, running in distinct processors; in this case, there has to be only one database, and every interface using the workspace mapped onto the database must connect to it.

The mapping of workspaces to databases must respect the visibility rules previously defined:

- Every object using the repository can view the archive workspace;
- Only objects within an application can view the group workspace initialized by the application;
- Only one thread within the application can view its own private workspace.

A possible implementation of this policy consists of:

- The archive workspace is implemented as one database, optimized to hold large amounts of data and perform big transactions; every application should know how to access this database to use the archive data.
- A group workspace is implemented as a new database, created whenever an application is started; this workspace will only last while the application is running. The application should check-in the data held in the workspace before finishing, because the database will be then discarded. This database will be visible only by the application.
- A private workspace is implemented as a new database created by an application thread. This database is optimized to quickly perform small transactions, holding limited amounts of data. It is discarded when the thread ends.

4 Results

We built a prototype implementation of the Versus model using the Java environment. The meta-data management was written in SQL and is as much DBMS independent as possible. All DBMS dependent constructs were placed in separate functions that can be easily edited for adding support for new DBMSs.

The prototype is built upon the following major software components::

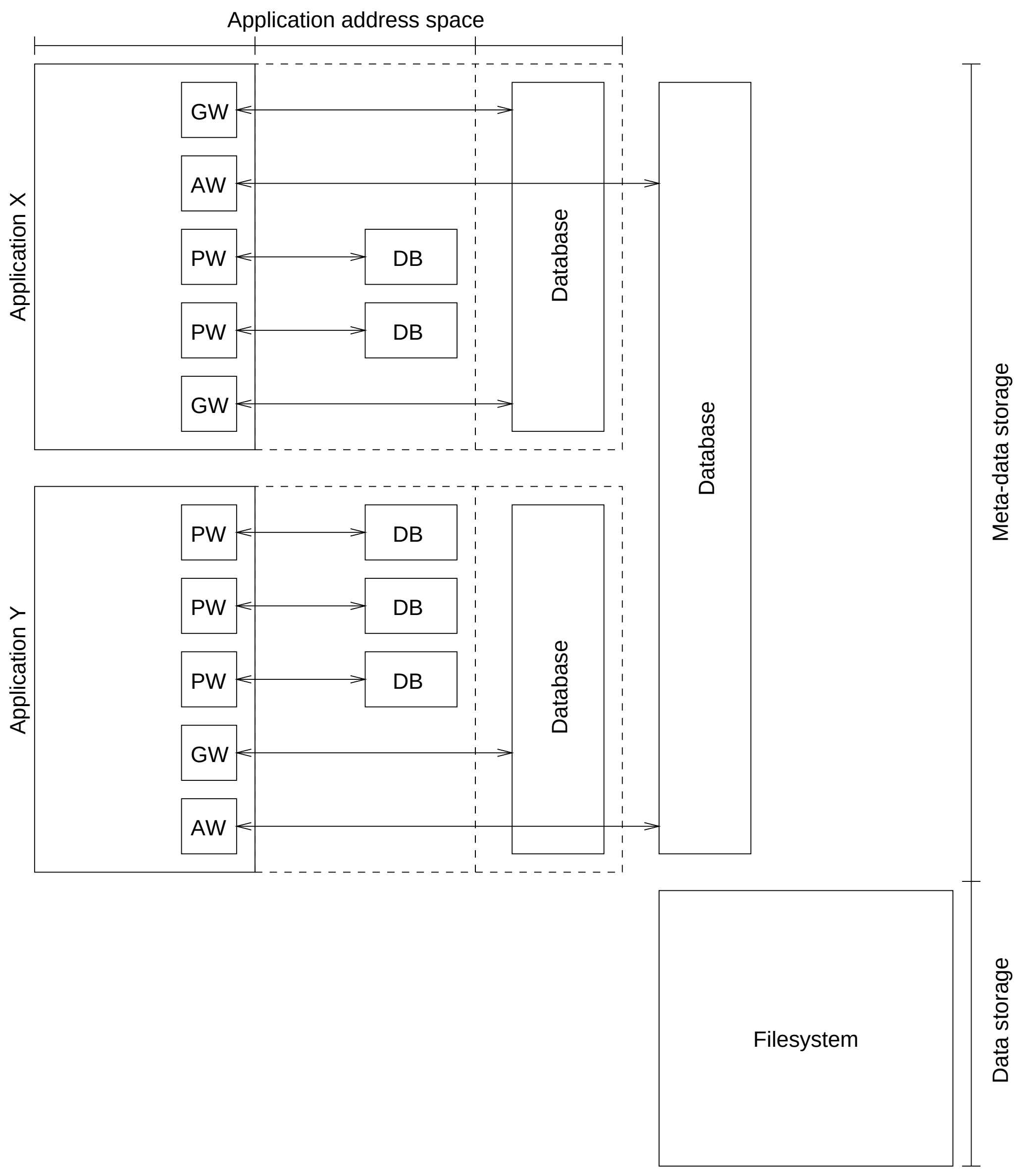

Figure 6: Data and meta-data management in Versus. AW, GW and PW are the archive, group and private workspace interfaces, respectively. Each workspace has its own database; several interfaces to one workspace share the database containing meta-data. All workspaces share the data storage. The application process includes the linked libraries, and, optionally, the databases of the private or private and group workspaces.

- The data management was implemented using NFS, a network file system [5].
- The DBMS supporting group and archive workspaces is an Oracle 8i server.
- The DBMS supporting the private workspace is an hsqldb server [11]. hsqldb, previously known as HyperSonicSQL is a database management system written in Java that can be configured to run in three modes: in-memory, file or client/server. Private workspaces run in-memory databases: all the data is stored in the Versus application memory space, and all the database management is run in the application run time.

5 Conclusions and Future Work

This paper presented Versus, a model for a repository for Web data, supporting versioning of objects, and distributed operation. Versus uses workspaces to distribute the data among the several processors involved in the processing of the data.

Versus assumes that data sets to process can be partitioned into disjoint units that can be processed with a certain degree of independence. Its potential for better performances comes from the assumption that the processing of independent data units generate no conflicts with data being processed in parallel by other processors. However, applications can generate conflicts (in a limited extent), as long as they provide methods for solving the conflicts when merging the data units.

Distribution is application driven, allowing client applications to specify how they want to partition their data into units that will be assigned to processors.

An API was defined to let applications access Versus. A modular architecture was proposed, setting the design for implementations of the model. A prototype implementing the Versus API as a Java library and using other off-the-shelf components to implement blocks specified in the architecture was built. A performance evaluation of the prototype was defined and prototype evaluation/optimization is under way, using a developed simulator to stress the several usage patterns of the prototype when applied to real client applications.

Versus is not directly comparable with other systems that we know, because it has a unique set of features: it is a large-scale, distributed Web data repository, supporting object versioning.

The future work on Versus includes a detailed study of Versus performance, comparing the results with other technologies currently available that, albeit not overlapping Versus in functionality, may be adapted to tackle some of the requirements of the client applications.

Other areas of interest are the study of the feasibility of developing a WebDAV interface for Versus, and the refinement of Versus to allow objects to be versioned at the XML element granularity.

References

[1] Altavista. `http://www.altavista.com`, July 2001.

[2] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems (TOCS)*, 14(1):41–79, February 1996.

[3] Thomas Ball and F. Douglis. An Internet Difference Engine and its applications. In *Proceedings of the COMPCON Spring '96*, 1996.

[4] Mike Burner. Crawling towards eternity: Building an archive of the World Wide Web. *Web Techniques Magazine*, 2(5), May 1997.

[5] B. Callaghan, B. Pawlowski, and P. Staubach. *RFC 1813: NFS Version 3 Protocol Specification.* Sun Microsystems, Inc., June 1995.

[6] Fred Douglis, Thomas Ball, Yih-Farn Chen, and Eleftherios Koutsofios. WebGUIDE: Querying and navigating changes in Web repositories. *Computer Networks*, 28:1335–1344, May 1996. In Proceedings of the 5th International World-Wide Web Conference.

[7] Fred Douglis, Thomas Ball, Yih-Farn Chen, and Eleftherios Koutsofios. The AT&T Internet Difference Engine: Tracking and viewing changes on the Web. *World Wide Web*, 1(1):27–44, 1998.

[8] Google. `http://www.google.com`, July 2001.

[9] Harvest Web indexing. `http://www.tardis.ed.ac.uk/harvest/`, July 2001.

[10] Jun Hirai, Sriram Raghavan, Hector Garcia-Molina, and Andreas Paepcke. WebBase: A repository of web pages. In *Proceedings of the Nineth World-Wide Web Conference*, 1999.

[11] hsql database engine home page. `http://hsqldb.sourceforge.net/`, July 2001.

[12] The Internet Archive: Building an 'Internet Library'. `http://www.archive.org`, July 2001.

[13] E. James Whitehead Jr. and Yaron Y. Goland. WebDAV: A network protocol for remote collaborative authoring on the Web. In Susanne Bødker, Morten Kyng, and Kjeld Schmidt, editors, *Proceedings of the 6th European Conference on Computer Supported Cooperative Work*, September 1999.

[14] Randy H. Katz. Toward a unified framework for version modeling in engineering databases. *ACM Computing Surveys*, 22(4):375–408, 1990.

[15] Z. Smith. The truth about the web. *Web Techniques Magazine*, 2(5), May 1997.

[16] Danny Sullivan. Altavista regional listings left to rot. *The Search Engine Report*, September 2001. `http://searchenginewatch.com/sereport/01/09-altavista.html`.

[17] Walter F. Tichy. Design, implementation, and evaluation of a Revision Control System. In *Proceedings, 6th International Conference on Software Engineering, September 13-16, 1982, Tokyo, Japan*, pages 58–67. IEEE Computer Society, September 1982.

[18] WebDAV resources. `http://www.webdav.org`, July 2001.