Palo Alto Research Center

The Yggdrasil Project: Motivation and Design

Robert B. Hagmann

XEROX

The Yggdrasil Project: Motivation and Design

Robert B. Hagmann
Xerox Palo Alto Research Center

CSL-91-13 October 1991 [P91-00140]

Abstract: The Yggdrasil Project has been concerned with the design, building, and operation of a large scale, persistent information storage server. The novel parts of this system are that it supports multiple data models, merges file server and database server implementation, scales both in record size and server total size, has set based properties, and provides non-navigational access to hypertext.

The main type of information to be stored in the system is literature. This is the principle form of stored human information. The electronic form of literature is the evolution of printed literature that is current today. This information differs from the typical database (that stores "data") or from a file server in the size of objects, object semantics, interrelationships, lifetimes, criticisms, histories, and types of information to be stored.

This paper describes several parts of the design and implementation of Yggdrasil. The paper concentrates on the novel parts of the design such as data model integration, a storage design for datum of vastly varying sizes, delayed indexing, and alerters.

CR Categories and Subject Descriptors: H.2.2 **[**Database Management**]:** Physical Design; H.2.4 **[**Database Management**]:** Systems; H.2.7 **[**Database Management**]:** Database Administration — Logging and recovery

Additional Keywords and Phrases: Hypertext, Object-Oriented Database

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

1. Introduction

The Yggdrasil Project has been concerned with the design, building, and operation of a large scale, persistent information storage server. This paper discusses several of the issues and aspects of the design and implementation of Yggdrasil. The prime audience for the paper is intended to be people interested in database systems implementation.

The project came out of the focusing of the corporate policy of Xerox Corporation on "Document Processing." Xerox corporate strategy requires multimedia, large scale, unstructured, robust, and efficient servers for the storage and retrieval of documents. Yggdrasil was a project in Xerox's Palo Alto Research Center, Computer Sciences Laboratory (CSL) that addressed many of the research issues needed to provide such a service. A prototype server was built, but was never used except by the developers.

The cost of secondary storage has continued to decrease over the years. In addition, the range of size of individual objects that people wish to store has grown One storage object in CSL is a 160 megabyte file. The simple yet elegant hierarchal naming systems such as we have in file systems do not work well for all kinds of data. Nor were the concepts and existing implementations of relational database systems a panacea: they is missing several features that make it difficult for it to model some applications (e.g., good solutions for repeating groups, hypertext, stored procedures, transitive closure, large objects, ...). Object-oriented databases are a current research topic, but significant problems arise in their performance, locking, security, evolution, archiving, execution model, and robustness.

Although much of the research in database systems is involved with increasing the semantics of the system, many of the potential clients of these systems simply want data storage. They want the database to "store the bits and get out of the way." Semantics will be provided by the applications. The database must be fast, efficient, and robust.

The name Yggdrasil was chosen because of its definition [Amer81]:

> **Ygg|dra|sil** *n.* **Also Yg|dra|sil.** *Norse Mythology.* The great ash tree that holds together earth, heaven, and hell by its roots and branches. [Old Norse, probably "the horse of Yggr" : *Yggr*, name of Odin, from *yggr*, variant of *uggr*, frightful (see **ugly**) + *drasill*, horse.]

CSL has a history of naming projects from the Sunset New Western Garden Book [Suns79]. The project was meant to be infrastructure so that it "holds together" many other things. It supports versions so "roots and branches" is relevant.

The theme of this paper is the solutions to new problems encountered when expanding storage requirements in several dimensions at once. The reader will learn about data model integration, a storage design for datum of vastly varying sizes, delayed indexing, and alerters. The paper concentrates on the novel parts of the design and gives minimal treatment to the more standard

parts of the system. Parts of the system that were designed but not built are not discussed since no experience was gained in the construction or operation of those parts of the system. These parts included the query system, tertiary storage issues, availability, compression, and client object caching.

The lessons learned from this experience were:

integration of data models is important and feasible

all databases should strive to have file server interfaces for some of the stored data

generic parts of data models include properties, sets, naming, and containers

separate locking name spaces for different aspects of objects can help concurrency and file server implementation

versioning is important and must be done on the server

external interfaces to servers should publish permanent names for objects and anchors

This paper is organized as follows. Section 2 provides a historical perspective on this work. Section 3 is "Seven Key Ideas and Rationale" and deals with the basic data model and the reasons for choosing these ideas. Section 4 has the basic overview of the system design. It provides the understanding of the overall design so that the pieces that are discussed later fit together better. Section 5 is "Data Model Integration" and shows how the server was built to support multiple data models. Allocation and clustering of magnetic disk storage is discussed in Section 6. Yggdrasil normally performs its index updates in a delayed and as soon as possible (ASAP) manner. Section 7 discusses the implementation and justification for this. Section 8 discusses a simple triggering mechanism called Alerters. Section 9 discusses related work. The paper last section is for conclusions.

2 Historical perspective

The storage, presentation, organization, access, creation, tailorability, finding, and contributing to the store of knowledge of mankind will be changing. That is, *literature* will have a new form. Part of this change will be the underlying database support the new literature. This paper presents the motivation and design aspects for a project that dealt with a database for the new literature. It had a data model and was of the scale needed for departmental sized organizations to deal with the new literature.

Literature today is thought of as a collection of writings that have been printed. Literature distills the knowledge of mankind. At various times, literature was passed orally, was restricted to "scholars," was *not* supposed to evolve, or was only available to the rich. Today, printed literature is widely available and about half of the world's adult population is literate.

Literature will evolve away from printing solely on paper. Movies, radio, telephone, television, recorded video tape, recorded audio tape, electronic mail, FAX, CD ROM, and high bandwidth communication will (continue) to provide alternative dissemination methods to printing. By scanning old format printed material into an electronic form, and performing page recognition and natural language processing, old literature will evolve into an electronic form. With electronic literature, there is a further broadening of the availability of information.

The character of the literature also will change. A piece of literature can be designed to be viewed as a hologram. The piece can be active: it can run simulations, perform database queries, compute graphics, or compute sound. A document can be shown at the level of understanding of the reader.

Literature will change faster. Much of what will be published will be of low quality. Criticism and reviews will also be published. Criticism and reviews must be used to filter the literature so that it is manageable.

Since literature will become electronic, the structure of it can change. There will be non-linear documents (hypertext). Access to literature will also change: querying and filtering agents will find information for us and hide junk from us.

One of the keys to having literature evolve is to have an appropriate database. To be useful, it has to be large and persistent. It has to be able to store text files, object files, page description language files, editable documents, video, audio, as well as the normal "data" that is stored in databases (integers, dates, floats, etc.). The Yggdrasil project addressed the database issue by designing and building a database server whose's underlying data model is hypertext, that has the scale and scalability needed, which supports multiple data models, and has the "systems" aspects done well.

3. Seven Key Ideas and Rationale

This section presents the seven basic ideas for the system. None of these ideas are new. The value of this project is in bringing these ideas together in one place, applying them to the literature problem, and implementing them in a robust manner. The ideas are nodes, labeled links, properties, containers, set oriented non-navigational access, names, and versions.

3.1 Nodes (documents or objects without semantics)

In keeping with the theme of literature, nodes in the database are also called documents. A node may not have a real world analogy that the reader might consider a document. The node's contents might be the number three. The node might be a source or object "file."

The system was designed to handle a large number of nodes (e.g., millions). A node can be of

any size: one byte long, or a gigabyte. Nodes (or node parts) were to be archived to on-line tertiary storage (e. g., an optical disk jukebox).

Nodes have a *contents*. The contents of a node is a *primitive value*: it is a type and a value (of that type). Only a dozen or so types are understood by the server. For example, strings (any length), dates, floats, object identifiers (or Document Identifiers or a *DID*), and integers (32 bit and infinite precision) are supported. In addition, other types are stored without the server understanding what the value means. For example, ODA documents and compressed scanned images are stored, but the server treats them as uninterpreted bytes. The client code, not running on the server, is expected to provide the semantics for these types. The type space is fairly large (almost 32 bits) and is administrated by a database administrator without much support from the system.

The notion of an object, node, document, entity, or file seems pervasive in many of the areas that are of interest. To support hypertext, file servers, and object-oriented databases then this seems like the right place to start with the data model.

3.2 Labeled links

Nodes can be connected by links (hypertext) [Bush45, Engl63, Nels81]. A link is a directed pointer between two nodes. The link has a label, which is just a string. An *inlink* or a *to link* are links that point at a node, while an *outlink* or a *from link* are links leaving a node.

Nodes can efficiently determine both their inlinks and outlinks. Although links are directional, they can be followed in either way. Links are themselves nodes. Links were binary in the implementation, but would have been expanded to be multi-headed and multi-tailed if the project continued. Anchors (link source or destinations of smaller granularity than an object) are not in the system, but should have been.

Not only is the notion of node important, but also the relationships between nodes. Links are a very flexible and simple notion. Binary relations between nodes is the simplest form of these, yet it is capable of modeling most relationships. Links are directional since most relations are not reflexive.

Links are nodes (or at least can be reified as nodes) for two reasons. First, sometimes the dual view of the "schema" is important. That is, the entities and relationships are backwards: the entities should be relationships and the relationships should be entities. Without the links being nodes this is hard to model. Second, it is sometimes useful to have links as nodes. For example, assigning properties to links can be useful.

3.3 Nodes have properties (attributes)

Nodes have a set of *attributes* or *properties*. Both terms are used interchangeably. The attributes of a node are a set of *attribute name* and *attribute value* pairs. The attribute name is just a string that identifies the attribute. The attribute value is a set of fields. Each field has a *field name* and a *field value*. The field value is a set of primitive values. Each of the above sets may be ordered or unordered.

So the attributes are a set with the elements identified by *attribute name*. Each attribute has a value that is a set of fields. Fields are set valued where these values are primitive values. So the attributes are three levels of sets.

Having no restriction of the number of levels of sets made constructing a query language much harder. In contrast, only two levels made modeling of some applications difficult and less efficient. The modeling difficulties seemed particularly true for the types of objects, literature, for which the system was designed. Three levels of sets is a compromise that is powerful in capturing most applications yet easier to implement than full generality. Note that relational systems have only one level of sets: the fields of a relation are single valued.

With three levels, the common case of an auxiliary table used to simulate repeating groups can be modeled as part of the object. The benefit is the elimination of the otherwise useless auxiliary table. It also seems better to keep the private properties of an object with the object, and not create auxiliary objects. Objects are then are more self contained, so less of the structure of the database is hidden in processing and is easier to store together on disk. Not all cases of the use of auxiliary tables are eliminated. Where the auxiliary table serves some other purpose (e.g., it is used to indicate common structure) or if the repeating group is itself multi-level, then an auxiliary object is preferable to putting the data "inline" the object.

If the sets are ordered, then they give the n-ary relations as a field value (see next subsection). Ordering is also a natural for many applications. For documents we want to see chapter 1 first. Relational systems are guaranteed to be unordered.

Naming of attributes is common in systems we are trying to supplant or make persistent. File servers have named attributes. Object oriented systems have named slots. Hence, attributes have names.

Fields were named to allow for finer granularity in naming of attributes. These names are optional.

3.4 Containers group nodes

Yggdrasil is designed to hold millions or billions of nodes. Without some help, it is easy to get "lost in (hyper) space." Both databases and file systems have grouping mechanisms. Relational

databases use relations to group similar nodes together. File systems have (hierarchical) directories that are a well accepted improvement over a flat name space. Again, related items are placed in the same group. However, the items are no longer similar (e.g., .c files and .o files). Some sort of context or grouping is needed by any large system.

Containers are the context mechanism in Yggdrasil. A container is a (possibly ordered) set of nodes. A container is associated with a *root node* of the container (analogous to directory in a file system). A node can belong to any number of containers. A container can be the sub-container of any number of containers. Loops are allowed in the sub-container relation (i. e., it is a directed graph, not just a DAG).

3.5 Set oriented non-navigational access

One of the powers of relational databases is the non-navigational, relational access to data. Relations are just sets of similar items. The access to data is non-navigational in that the client characterizes what the matching data looks like, as opposed to providing the systems with an (imperative) program to get the data. Sometimes this is called non-procedural access.

Both navigational and non-navigational access are desirable. Hypertext systems are outstanding for browsing (navigating), but are not good at finding starting points for browsing.

Containers can be configured to automatically maintain indices. For any attribute, all nodes that can be reached in the transitive closure of the containment graph starting at the container are indexed. Index maintenance is specified by the "owner" of the subtree or by the database administrator. An index at a level replicates data of indices at lower levels. The design of the query system uses the notion of container, so that often the indices can be used during query evaluation.

3.6 Documents can be named

Any node can act as a directory. It is natural to expect the root nodes of containers to be directories, but the system does not make this restriction. A special system maintained property provides the name and node binding. Not all nodes have to be named and a node can have many names.

Naming can also help with the "lost in (hyper) space" problem. However, a major consequence of providing naming is that Yggdrasil can provide a view of a database that is a file system or file server. A file server front-end can be built that talks a file server protocol, and provides access to some of the data in the server. Links are not modeled in all file server protocols, so they usually disappear from the file server view. A particular file server protocol may have a restricted notion of attributes (e. g., a fixed number of them of fixed types) or the protocol may not

type the contents of files. Directories may or may not have contents. In any case, the file server front-end provides a view of the database that is suitable for programs that are designed to talk to a file server.

A few front-ends were anticipated. NFS, Xerox's NS Filing Protocol, and ANSI/CCITT/OSI Filing and Retrieval were all candidates for front-ends. A server is not restricted to only one front-end filing protocol.

One prime benefit of this is to unify file systems and databases. There is one permanent storage mechanism. Different views are provided for different software. It is possible to maintain information in the database, and "publish" it via the naming system.

3.7 Versions and alternatives for nodes

Nodes can have have *versions*, *revisions*, and *alternatives*. A node is really a tree of states for the node. A version is a modification of a state that produces a new state. An alternative is a new branch added to an existing node. States can differ in any of contents, properties, links, containers to which it belongs, and its value as a container. Major changes along a branch are called versions while minor changes are called revisions. Changes are (usually) applied at the leaves of the tree. A new version or revision is built whenever a transaction commits that changes some part of the node. The version or revision is added to the version chain for the current alternative. A new version collapses the revisions between it and the previous version.

Versions and alternatives are a requirement for many systems. File servers files, documents, and CAD objects all benefit from having versioning. The distinction between versions and revisions is more subtle. Revisions are intended to model changes that are minor. It is important to commit the changes, but after a short time the revisions are irrelevant.

4. Design Structure: The Basics and Overview

4.1 Foundations

4.1.1 Mach, Camelot, and the hardware platform

The operating system of choice for this project was Mach [Rash86], the transaction layer was Camelot [Spec87], and the hardware platform were the SPARC based machines from SUN. Unfortunately, the existing release of Mach during the design and implementation of the system did not run on SPARC's. While waiting for Mach to be ported, a Mach/Camelot emulator was built to implement those features of Mach and Camelot needed by Yggdrasil, but running over SunOS.

4.1.2 PCR

The Portable Common Runtime (PCR) provides many useful services for its clients [Weis89]. It has lightweight threads, numerous file descriptors, garbage collection, dynamic loading, and load state management.

4.1.3 The Cedar Programming Language and PCedar

Cedar is a programming language and programming environment [Swin86]. It is a language of the Algol family that features strong typing, data abstraction, monitors, lightweight threads, garbage collection, single virtual address space, generic references, lists, atoms, procedure variables, and exceptions.

Many of the basic abstractions needed by Yggdrasil already exist in the programming environment. There are some 2 million lines of code, so that there is much to use. Garbage collection and lists were particularly useful.

4.2 Storage management

The system design has three levels of on-line storage. In current technology they correspond to main memory, magnetic disk, and optical disk juke box. In the design of a three level system with migration of objects between the two stable, slower levels, an identifier for an object cannot be mapped directly to the secondary or tertiary storage. After all, the object might move. Forwarding pointers are inconvenient. So the design has a storage map, indexed by DID, for all objects. This is called the *DID Map*. This map encodes the secondary and/or tertiary storage used to store the stable disk resident form of all objects. It also encodes some other metadata such as where different versions of the objects are stored; slot numbers (for small objects); use, modify, and backup times; and compression technique used (if any). Objects that have large storage needs for naming, contents, or other properties use a separate component file for each large part of the object.

By providing a map, an external identifier for an object can be provided that identifies the object over all the time that the object exists. Since the object might migrate between storage levels, external identifiers that, in any way, encode the storage address are undesirable. The combined benefit of permanent external identifiers, full flexibility in storage location and media, and centralized storage of some object metadata justifies the cost of the DID Map. Extensive caching made the time cost of the DID Map not too large.

The real design challenges in secondary storage management are clustering, migration between stable storage levels, small object and big objects in the same system (objects vs. files), and buffering policies. The design for Yggdrasil makes a contribution in small object and big objects allocator and clustering (see Section 6).

4.3 Naming

Every container has a "root" node which represents the container. A system maintained property of any object, including the root node for a container, is naming. Naming maps a string name to a DID.

The name matching may be case sensitive and may include versions. The naming property is system maintained so as to preserve several invariants about naming.

The naming information is kept in a separate file for the object (see Section 6.3). This file is used as the permanent storage for a B-Tree for the names mapped from the context of the object. Although naming is a property, the disk data representation is not a list associated with the object.

For NFS, containers and directories are one and the same. Names are only allowed from containers (directories). Lookup is case sensitive and the highest (most recent) version is always found.

4.4 Locking

The existing design uses object locks. For the initial system, this makes sense since some locking is already at the object level (e.g., flock in 4.3 BSD). Many OODBMS's use object locks. The design admits page level and finer granularity locks.

But with object locks certain operations are unnaturally restricted. Linking to an object should not require a lock on the object. Adding a name to a directory should not lock the underlying object and the whole directory.

Locks are split for an object. There are object locks, directory locks, and meta locks. The object locks cover the contents and the (normal) properties. These are the externally visible locks. Directory locks are for the naming code. Meta locks handle the system maintained properties such as links and container membership.

There are the usual read and write locks, but the design also has browse locks. A browse lock is a breakable read lock. When broken, the agent performing the transaction is notified (see Section 8 on Alerters). The transaction does not abort.

4.5 Volatile and Stable Objects, Change Lists, and Object Update

Objects exist in two forms: volatile (in memory) and stable (on disk). A cache of volatile objects is kept. When a cache miss occurs, the DID is looked up in the DID Map and the object (or at least the small part of its metadata) is read from disk, parsed into a volatile form, and added to the cache. Large objects are buffered while small objects are read completely.

To modify an object, an appropriate lock is first obtained. Modifications are kept as a list associated with the volatile object. Large parts of objects are handled specially: only the pages

changed are noted as modifications. Subtransaction commit [Moss81] is just a list operation: the modifications of the subtransactions are appended to the lists of its parent. Modification lists make the implementation of subtransactions easy. Lists are a language primitive in Cedar and they are used freely in the implementation. Since Cedar has a garbage collector, lists are garbage collected after use.

Transaction commit is divided into three phases: precommit, initiate Camelot commit, and either a commit or an abort (from Camelot). To start a commit, precommit applies the changes to the object. This is complicated by objects having multiple lock types (see above). Concurrent transactions can be updating the same object. To avoid anomalies and deadlock, but still have good parallelism during commit, changes are applied in a partial order. All objects to be updated in a transaction are hashed in such a way that all conflicting updates hash to the same value. Update of the same DID is a conflict, but so is update of two DID's that are stored on the same secondary storage page. The necessary buckets of the hash for a transaction are latched in order. Holding the latch, DID's matching the hash value are latched and the changes applied. When all objects for the bucket are changed, the bucket latch is dropped. The object locks are kept until the commit record is buffered. If a bucket latch cannot be obtained, the transaction waits. Deadlock cannot occur since the hashing imposes a total order. Normally, little blocking occurs due to the hashing. Multiple concurrent, non-conflicting updates are allowed. Once precommit is over, the system asks Camelot to commit the transaction.

4.6 Delayed index management

Yggdrasil provides for facilities to build indices in containers. An index is built over the properties or contents of an object. Only values that are interpreted by the server may be indexed. The objects that are indexed are exactly those in the transitive closure of the container relation seeded with the container. Indices are built selectively based on systems administrators direction. See Section 7 for more information.

4.7 Query system

Part of the design of the system is to have non-navigational access to the data in the system. This involves extending a relational query language with the native data model of the server (e.g., links). Since the query system was never built, so that the principles of the system were not verified.

The major aspects of the query system were:

Sets: Sets of nodes are what is manipulated. A container is an example of a set of nodes.

Relational operations on sets: The normal set and relational operations are provided for sets

of nodes.

Dereferencing links: Links may be dereferenced either the the link itself or to the resulting object.

Attribute and field selection: Attributes and fields may be selected and operated upon using conventional operations.

Transitive closure: Links may be followed to build a transitive closure for a query.

Expressions, Conditions, and Aggregation

5. Data Model Integration

5.1. Introduction

Interoperability with different persistent storage systems is a key problem in database systems. Users and applications need to use multiple persistent storage systems. Often this problem is not even recognized, and the user or application takes on the interoperability burden.

To see the problem, consider the example of an electronic mail system for a user. Mail is sent and routed to the user. The user's mail system accesses the mail from the mail server. It then eliminates duplicates, inserts the new mail into a full text database, updates a relational database of messages, marks interest in attachments (file, video, or whatever), adds an icon to the desktop, and stores the message in a folder (directory) on a file server. All of this should be done under transaction control. Note the variety of the persistent storage systems involved. They may have different protocols, commit logic, and naming. It is rare that a common commit protocol is used between the multiple persistent storage systems. The persistent storage systems do not share coherent notions of backup. The availability of the system is the product of the individual availabilities. Coherence and consistency are hard to enforce.

Other tools available to the user may not be used in their accustomed manner with the mail. Suppose that the mail was completely stored in a relational database. Most printing software is file based: the document formatters take a file and print it. To print mail, a file with the contents has to be created and the file is then printed. The editor, spelling checker, keyword finder, and grammar checker also may only expect files.

The main problem is that the application has to take on the full burden of interoperability. Useful components have been combined to give a service. But every application (or user) has to provide interoperability for itself.

This section presents a way to address part of this problem. In Yggdrasil, a single server supports multiple data models layered over a common storage system.

The method of persuasion of this section is to relate the design, implementation, and testing

experience of building a system that has data model integration.

5.2. Motivation and Solution Strategy

5.2.1 Why worry about integration?

Integration has usually been missing from persistent storage systems. If there is a problem, why is it not more widely recognized?

One answer is that since different persistent storage systems have evolved separately, the desire for integration and the feasibility of integration has not been appreciated. Within a particular type of persistent storage system, interoperability is often recognized. Standard protocols for connecting Distributed Relational Database systems are being developed. But interoperability between different data models for data base systems or interoperability between different file servers using different protocols is rare. The problem has been addressed for particular data models, not for persistent storage systems as a whole.

There are also "legacy" applications or databases. These are existing software or databases that provide some useful function. They are mostly fixed in their use. It is impractical to change them. These systems do not provide for integration so that they must be integrated by the user or application writer.

5.2.2 Where can integration be done?

The simplest answer is nowhere. Integration of services is left to the user.

Often the burden of interoperability is taken by the application. It deals with the data model and interface issues. Every application has to do this for itself.

One area that has had some research is that of accessing multiple databases from a common interface. The implementation of the interface handles the interoperability problems.

Finally, much of the interoperability burden can be taken on by the persistent storage server. The system supports multiple data models over the same data.

5.2.3 Why not Federated Databases?

One area of research has been Federated Databases. An additional layer or view is typically added on top of several existing databases. Integration is a key concept (see [Daya84]), but it is hard to achieve. The existing databases have different schemas. They may be very structurally different and not be mutually consistent. The databases may not cooperate during commit, so additional inconsistencies may occur. The databases are not backed up in a consistent manner. Loading backup of one database will destroy invariants. Performance can be a problem. Controlling many independent databases and maintaining invariants (e.g., referential integrity across databases) is hard.

If the underlying databases use different data models, additional problems arise. The tendency is to have a database that is the ``greatest common divisor'' of the database models. The federation layer has to mask the deficiencies and differences between the models to get something useful.

By having one system that incorporates multiple models, the tendency is to avoid the inconsistencies. Performance, backup, restore, invariant maintenance, and integration are better using data model integration than in Federated Databases.

5.2.4 Which models?

Yggdrasil uses three different data models. Some of these, particularly the file server data model, are not normally recognized as a data model. A brief discussion of data models is in order.

Date [Date86] defines a data model as consisting of three components: a collection of object types, a collection of operators, and a collection of general integrity rules. This is a fairly broad definition. File servers do have a data model.

Briefly, these models are described below.

Hypertext

Hypertext (or Hypermedia) has an extraordinarily simple data model. It has objects and links. Objects are entities and links are relationships between objects.

OODBMS backend

Many of the proposed and actual OODBMS systems have a client server model. The server is called a backend.

The data model is often just objects with slots. The slots can contain object pointers or primitive data. There is a special slot for the contents of the object. Semantics is performed either on the client, on the server, or on both.

File Server (NFS in the implementation)

The file server data model is not normally recognized as a data model. It is somewhat degenerate as compared with traditional data models. In the UNIX File System, there are only two types of objects: directories and files. The operators are what we normally think of as system calls (e.g., create, delete, and write). The integrity rules also are pretty minimal. In the UNIX File System, loops are not allowed in the directory name space.

File servers provide a name to "uninterpreted sequence of bytes" mapping. Their strengths over normal databases is that they use naming for access, they are well integrated into many application programs (e.g., compilers, editors, drawing programs, ...), they store large objects, and they impose little structure on their data. Many of these strengths can also be viewed as weaknesses.

The file server protocol used in this work is NFS [SUN86]. This protocol was chosen because of its wide use and simplicity. Other protocols have more structure in the name space (e.g., Xerox's NS Filing Protocol (XNS) and ANSI). Others have more structure in the objects (files) themselves (e.g., FTAM). While the design for this system considered the requirements of other filing protocols (particularly XNS), in this paper we will concentrate on NFS since that is the one that was implemented.

5.2.5 Why these three models?

To make the points of interoperability and integration, at least two data models are required. The hypertext and file server data models are implemented in the prototype. The OODBMS has had extensive design work and simulation. It has not yet been implemented.

These three models are all of particular interest to Xerox and to Xerox PARC. Xerox is "The Document Company." This is part of its corporate strategy. Interconnections between documents is quite naturally modeled with hypertext. and this model imposes few restrictions. It is flexible and unstructured. In addition, PARC has been a leader in hypertext with the Notecards project [Halz87] and in object-oriented systems such as Smalltalk [Gold83].

File servers were first built at PARC. Woodstock was designed and built in 1975 [Swin79]. The Interim File Server (IFS) was built in 1977 and was used until 1991. Much of our software depends on files from file servers.

The last reason to select these three data models is that they do not all have complete, precise semantics. Hypertext and OODBMS's have not existed long enough in the database community to have the semantics that well developed. There are no standards. File servers typically only have very weak semantic guarantees.

5.3. A server with three views

5.3.1 Why a server emphasis rather than a distributed systems emphasis

Scale is important in any system design. Scale can take on several meanings. Many large servers storage will most likely exist in the future. There are issues of scale within the server and issue of scale among many servers. Since it was infeasible to pursue both of these scaling dimensions at once, scaling within a server was emphasized over scaling over many servers. The main reason for this choice was money. A single server can have an optical disk juke box to provide low cost, archival, on-line storage. The per megabyte cost of an optical disk juke box is very attractive.

None of this is intended to indicate that a Yggdrasil server is not a network citizen. The server is designed for participation in distributed transactions, dealing with external unique identifiers, and

replication of services.

5.3.2 The union of the supported data models

The server has its own data model. It is roughly the union of the three data models. Common concepts are merged and a generic primitive is used in the data model. There are objects (or nodes or files), links, names, and sets of objects (directories or relations, called containers here). Objects have set valued attributes (attributes are file properties and fields in a relation). The attributes are named by a string name. Objects have object identifiers (OID's in OODBMS's or inumber's in NFS). *Metadata* about an object is everything known about an object except its contents (e.g., properties, containers the object is a member, ...).

5.3.3 Views imposed over the data

The server provides views over the data for the various data models. These views share many properties with the views in relational systems. Not all of the data can be accessed through a view. There also is the view update problem.

5.4. Design structure

5.4.1 Naming

In Section 4.3, it was pointed out that all containers have naming maintained by the system as a property of the "root" node for the container. This information is kept in a separate file for the object with the data structure of a B-Tree.

The main problem in naming was the amount of log data. A B-Tree causes lots of bytes to be moved during an insert or delete. Since physical logging was used, a lot of log data could be generated for large directories. Particularly for NFS, a different data structure, such as hashing, could have been used that has simpler update.

But this has nothing to do with problems with data model integration. The real problems have to do with *other* file server protocols. (Remember that Yggdrasil was designed to support multiple file server protocols, not just multiple data models.) No problems with the other data models arose.

Three problems with other file server protocols were uncovered. First, are names case sensitive? If so, then "makefile" and "Makefile" are different file names. If not, then they are the same file name. A case insensitive lookup may have its invariants broken: two files have the "same" name. A second problem is what are the valid characters in names. Xerox NS Filing allows 16-bit characters in component names. Spaces are allowed. How are these presented in NFS? The final problem is how do the access control of the multiple protocols interact?

Many incompatibilities are solved simply by the view mechanism. Additional properties not expected by NFS can be hidden. NFS only uses the most recent version. Set valued attributes

select the first element of the set as the value.

Since only NFS was supported, not all of the problems had to be addressed. File names with illegal NFS names are masked. Case insensitive naming must use versions, and the versions of the case insensitive names merged (e.g., "makefile!1" and "Makefile!2"). Access control for the view determines access.

Although not much insight was gained in supporting support multiple file server protocols, there was not much problem with naming in supporting the three data models.

5.4.2 The Graph engine and Views

A navigational graph engine interface is defined in the code. The other data models are implemented as views over this interface. The interface corresponds to the data model of the server. It has the abstractions of objects, contents, properties, indices, links, containers, and names.

OODBMS and hypertext fit easily into the native data model. The most challenging view was for NFS. This was not particularly difficult, but the following problems occurred.

First of all was error reporting. The NFS protocol only defines a few errors. These do not adequately model the errors that can occur in Yggdrasil. How are commit failures reported? How are inconsistencies reported? There was not adequate extensibility in the error reporting mechanism in NFS.

Another problem was using short term transactions and locks. These naturally conflict with the locking of longer running transactions. Since short term locks had to be held, different lock modes were used for NFS naming than were used for other access to the objects. Hence, there was a separate lock space for naming apart from the normal object locking. Moderate complication of the commit logic occurred due to concurrent, no-conflicting transactions committing while they both were updating the same object (see Section 4.5).

Does delete really delete the object? All that delete does, in NFS, is to remove the name from a directory. If it is no longer accessible (no other directory names it), then the file itself is removed. A compromise was taken in Yggdrasil. An object that has only been manipulated through the NFS interface is deleted when the last name is removed for it. To be exact, if the object is in any other container besides the directory container, it is not deleted.

The NFS protocol has an object called a cookie. It is only 32 bits long. This small size was a problem. The server could have more than a few billion objects.

Various locking games had to be played. For example, a rename between directories requires locks in both directories. The code acquires the first lock normally (i.e., with waiting), but for the second lock it attempts to get the lock without waiting. If it succeeds, all is well. If it fails, it drops the first lock and attempts the locking in the opposite order. Repeated failures or timeouts can terminate the operation.

5.5. Conclusions

The design, implementation, and testing of Yggdrasil show that data model integration for the data models hypertext, OODBMS backend, and file server is quite possible. This section used the design process of Yggdrasil to uncover the problems. While the problems were irritating and caused small glitches in semantics and implementation, nothing was insurmountable.

Databases can integrate and interoperate multiple data models. They can take on part of the interoperability burden by integrating data models.

File server interfaces should be provided by databases.

6. Magnetic disk page allocator and clustering

This section presents the magnetic disk page allocator and clustering design for Yggdrasil. Only the clustering and part of the large object handling was implemented. The system never ran with tertiary storage.

6.1 Requirements

The system had the following requirements:

Keep an object, its attributes, and its contents on a page - if it fits.

If an attribute value or the contents is bigger than a page, store it on its own page set.

Place no (real) restriction on the number of attributes or their sizes. An acceptable restriction is that an object cannot exceed the size of a tertiary storage device, tape, or platter.

Perform concurrent allocation

Cluster leaf objects near parents

Cluster links near ``from'' objects

Allocate big objects in big contiguous chunks

Have fast allocation

Good internal concurrency

Recover from crashes quickly.

6.2 The allocator and clustering

6.2.1 Splitting large objects

Clustering and small objects are related problems. For objects with large components, the system splits off these large parts of objects into different files. For an object with a large contents,

the properties and other metadata about a large object and the contents are in different files. Any property that is large (such as the naming attribute for a container) is kept in a separate file. Small objects and the metadata for large objects are both small.

6.2.2 Containers

Containers provide a natural way to organize large collections of objects. The notion of a container is generic notion that comes from contexts in hypertext [Deli87], directories in file systems, and relations in a RDBMS. The expectation is that a container holds objects that share some semantic relationship with each other and the metadata (e.g., properties) of these objects are the natural domain for queries. Hence, it is important to be able to search the metadata of the objects in a container quickly. Also, access to the metadata of one object in a container is taken as a good indication that other objects in the same container will be accessed.

All of this supports a clustering scheme that relies on the containers. Objects are clustered on secondary storage near their container. This is complicated by objects being in more than one container and the nesting of containers. For the purposes of clustering, an object has a *primary* container.

If clustering were recursive, then everything would have to be clustered near everything else. To break the circularity, container objects are never clustered. Leaf objects that are not containers themselves are clustered with their primary container.

The real test of a storage design is in a large scale, heavily used, perennial system. Yggdrasil cannot make any claims here.

6.3 Object allocation on magnetic disk

Put simply, the basic idea is to cluster what is possible. For small objects that's everything. For larger objects, this may be only the meta data, only the small parts of the meta data, or no meta data. By trying to cluster at least the meta data, searches of the metadata (e.g., properties) can be done quickly. As the size of the data and/or metadata gets so large that it interferes with clustering, this data is allocated somewhere not in the cluster.

Clustering is done for ``leaf'' objects near parent objects. Parent objects maintain a list of cylinders where some of their children are located. This is a hint for allocation. They also maintain a list of hints for partial pages that have some room on them that also contain children of the parent.

For example, a large object (a "file") can have it's metadata clustered. When the file is opened and read, the file can be read in large runs from disk. These runs are not necessarily near the metadata. For large objects these disk I/O's are not expensive since they are being amortized over many bytes transferred.

6.4 Allocating space on magnetic disk

6.4.1 Page allocation

There are three maps stably kept on disk for allocation. They are:

allocated/free page map: a bitmap indexed by page number

partially allocated page map: a bitmap indexed by page number - 0 means not a partial page, 1 means a partial page

approximate size free: a byte map indexed by page number - the high order bits of the free space on the page

Each of the stable maps has a volatile copy called the shadow map. Allocations are done to the shadow map prior to transaction commit and then performed on the stable map during commit. Deallocation follows a different order of update: stable modifications are made at commit and shadow modifications after commit completes.

To allocate some pages, the system looks in the *shadow* allocated/free page map for a run of bits big enough based on the clustering for the object. It sets the bits on in the shadow for the pages it gets. It locks the byte(s) where it allocated the pages (Camelot logging only logs bytes, not bits, so there can be anomalies). Any further allocations look in the shadow so they will not reallocate these page(s). Nor can other transactions allocate pages from the bytes that have been locked. This is non-blocking: other pages are allocated instead. At precommit, the system logs old value/new values from the stable and the shadow maps, respectively. Then the bits are set in the stable bitmap. At commit, the system does nothing but drop locks. If abort occurs, the system undoes the allocations from both maps and then drop locks.

To free some pages, they are just remembered in the file object during the transaction. Neither the shadow or stable maps are modified. At precommit if the lock can be obtained, the bits are turned off in the stable map, and old value/new values for the shadow and the stable maps, respectively. If the lock cannot be obtained, a transaction is forked to turn the bits off. If commit succeeds, the bits are freed in the shadow. If abort occurs, normal recovery fixes the stable bitmap. Locks are dropped after commit and abort. The forked transaction to turn the bits off can block on the lock as long as needed. This does not guarantee that the delete actually occurs: a crash may destroy the forked transaction. But there is only a loss of a few pages and only in the case of conflict and a crash, which should be rare.

Why does this work? The shadow's bits are set until the page can be used by another transaction. So the bits are set early by allocation and turned off late during freeing. Only pages that are free no matter what transactions commit or abort will have 0 bits. Keeping the allocated and freed runs associated with the file, which in turn is associated with the transaction, allows for

the redo/undo processing on the bitmaps. By locking bytes, we guarantee that allocators do not interfere with each other. Deallocators can interfere with other deallocators or with allocators. When they interfere, the bytes that interferes gets their updates forked. The system never waits for a lock during precommit, commit, or abort.

6.4.2 Slots and slot update

The system uses a slot table on pages holding small objects (or small parts of big objects). These slots are known through the DID Map and are not visible outside the server.

When an (apparently) small object part is to be written, either for the first time or re-written, the object value is written into non-recoverable VM. At commit, the size of the object parts can be accurately computed. If the object is new or no longer fits on its old page, the allocator is called (see the next section).

Assuming that we have a small object (or a small part of a large object), then the page for the object is then locked. If the lock would block and forms a cycle, then the lock attempt fails and the object part is moved off to a different page. If this transaction commits, then a new transaction is forked to remove the old object part. This extra transaction may be lost due to a crash, but we just loose storage. The DID Map and the values it points to are valid. Deadlock is not possible in this code.

6.4.3 Design for clustering

Objects are created in a container. The container provides the principle hook for clustering. Objects are clustered ``near'' the root object of the container.

When trying to allocate a large object part, a list of ``cluster near'' objects and ``segment, page, and size'' ranges is given to the allocator. Similarly, allocating a small object part, a list of ``partial page'' objects and ``segment, page, and size'' ranges is given to the allocator. As pages are allocated or filled they are added or deleted from the proper list.

Pages are grouped into sequences called *cylinder groups* [Leff89]. A cylinder group is a set of cylinders that are very near each other. It takes a very short time to seek from any cylinder to another in the group.

For some parents, there is a list of other groups where there children already have been located. Once this hint list is also exhausted, it is time to try a completely different group. The group is selected randomly from the groups that are somewhat empty.

At boot, the cylinder groups are arranged into lists of how full they are. Separate lists are kept for groups past the "little object low water mark" and before the "big object high water mark" [Hagm86].

7. Index maintenance

Yggdrasil provides for facilities to build indices in containers. An index is built over the properties or contents of an object. Only values that are interpreted by the server may be indexed. The objects that are indexed are exactly those in the transitive closure of the container relation, initially seeded with the container. Indices are built selectively based on systems administrators direction. They are used by the query system to speed up queries.

Whenever an document attribute is modified, the indices to this attribute have to be updated. Also, whenever container membership is changed, indices have to be updated.

Index update in commercial databases is quite complex. High concurrency of update and many indices per relation make this a hard problem to do well.

The indices are maintained by ASAP update, and hence are not guaranteed to be atomic with respect to the transaction that made the update. It is intended that the update proceed very quickly, but not using the client's transaction. The updates do have transaction guarantees, but not inside the client's transaction.

This makes the index update much easier. It is much easier to get good parallelism, the code is much shorter, and the code is easier to read.

The semantics for ASAP update are somewhat strange in the database community, but they model many real world activities. For example when a phone is installed, "information" does not immediately have the number. When a book is put in a library, the "card catalog" may not be atomicly updated. If a document is inserted into the database, why would the same user immediately query to find it? The user already knew about the document. The index update will happen quite soon, but it is not atomic with the insert.

The point of this is to argue that ASAP index updates have acceptable semantics for the principle uses of the database. There also an interface that can be used to delay informing a client of the completion of an update until all indexing activity for the update has been finished.

The DID for the document, its parent container DID's, and the add, mod, or delete information are appended to a "index update list." This list is kept in recoverable storage as a circular list with a start and an end pointer. One of the last things a transaction does before commit is to update this list (if needed).

A set of processes do the deferred updates from the "index update list". Each takes successive items off of the list. It tries to do the updates needed. If it cannot acquire a lock, it goes on to the next item. The update will be tried again later.

The goal is to do the update recursively for all needed objects, indices, or containers. However, the rules of the game can change. That is, while value that is indexed is changing, the containment membership may concurrently be changing.

So updates go in the following phases (cyclically). The first phase starts with all the containers and indices that has to be done is found. Dependence on update on the data in the object is detected (e. g., the parent objects of a container are important to index update due to a changed attribute). As objects are modified by index update, they are time stamped. The update keeps track of the largest timestamp seen. When it appears that the update is complete, the second phase starts. The dependencies are examined again. If a larger timestamp is found than was current during the first phase, the first phase is restarted. If no additional containers and indices are found, the update is done. Otherwise these containers and indices are processed as above.

Upon recovery, the index update list is processed again. Some updates may be redone, but this is idempotent.

8. Alerters

Alerters are a simple form of triggers. At some event on a Yggdrasil server, a "message" will be sent. Only simple events related to locking were supported. Events can include insert, delete, read, or update of some class of objects. The alerter will cause a message for some (probably) external entity to be queued. The alert will specify an object identifier for the object causing the alert, and optionally with a tag specified by the client. Alerts can either be one-shot or continuous. Alerts will be either for an ongoing connection or for an external server.

Alerters can have a lifetime of a transaction, an incarnation of the server, an incarnation of the remote client, or be persistent over both client and server crash.

9. Comparison to other work

It is impractical to compare Yggdrasil with all related work. Yggdrasil touches on too many areas of file systems, hypertext, operating systems, and databases. In this section only a few systems are mentioned.

Yggdrasil had three main roles: a file server, a hypertext storage system, and an object-oriented database backend. Yggdrasil made no contributions to file servers as such, but rather in integrating file servers with other forms of persistent data storage.

Hypertext originated in Vannevar Bush's 1945 classic article [Bush45]. Other notable early works are by Ted Nelson [Nels81] and Douglas Engelbart [Engl63]. An excellent survey of hypertext systems can be found in [Conk87]. One major hypertext project was done a PARC. This is the Notecards system [Halz87]. There is a conference devoted to Hypertext.

Yggdrasil does not make any contributions to the hypertext model. What it attempted to do was to be a long-term, persistent, very large storage system for hypertext. It differs in scale and implementation from nearly all other systems. One closely related system is Neptune [Deli86].

The concept of containers in Yggdrasil is modeled in part after the notion of Contexts for hypertext [Deli87].

Object-oriented databases are an active area within the database community. Many systems have been built. A few of them are GemStone [Maie87], Orion [Wilk90], O_2 [Deux90], and Iris [Wilk90].

The concept of versions is common in many systems. A good book on the subject is [Katz85]. One system that uses versions for programming environments in DOMAIN [Lebl85].

Clustering is a common technique in database systems. For example, ORION used clustering based on complex objects [Kim87] and Chang and Katz used structural relationships and inheritance [Chan89]. Where Yggdrasil is different is that clustering is based on containers.

10. Conclusions

This paper describes the historical foundation, goals, motivations, and basic system design for Yggdrasil. This project contributions are in the selecting existing ideas to build an artifact. The artifact is a large scale, full featured hypertext system with set oriented non-navigational access. Another major theme of the project is the integration of databases and file servers.

The lessons learned from this experience were:

integration of data models is important and feasible

all databases should strive to have file server interfaces for some of the stored data

generic parts of data models include properties, sets, naming, and containers.

separate locking name spaces for different aspects of objects can help concurrency

versioning is important and must be done on the server

external interfaces to servers should publish permanent names for objects and anchors

Acknowledgements

William Jackson, Brian Oki, and Marvin Theimer all participated in the Yggdrasil project. This work was possible due to the support of the author's co-workers in the Computer Sciences Laboratory at Xerox's Palo Alto Research Center and of Xerox Corporation.

References

[Amer81] *The American Heritage Dictionary of the English Language*, W. Morris, Ed. Houghton Mifflin. 1981.

[Bush45] V. Bush, ``As We May Think,'' *The Atlantic Monthly*. Vol. 176, No. 1, pp. 101-108,

July, 1945. Reprinted in *CD ROM The New Papyrus*, S. Lambert and S. Ropiequet eds., Microsoft Press, 1986.

[Chan89] E. Chang and R. Katz. "Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in an Object-Oriented DBMS," *Proceedings of the 1989 SIGMOD International Conference on the Management of Data*, Portland, June 1989, 348-357.

[Conk87] J. Conklin, *A Survey of Hypertext*. MCC Technical Report Number STP-356-86, Rev 1.

[Date86] C. J. Date. *An Introduction to Database Systems*, Volume I. Fourth Edition. Addison-Wesley. 1986.

[Deli86] N. Delisle and M. Schwartz. "Neptune: a Hypertext System for CAD Applications," *Proceedings of SIGMOD '86*, May. 1986, 132-143.

[Deli87] N. Delisle and M. Schwartz. "Contexts - A Partitioning Concept fro Hypertext," *ACM Transactions on Office Information Systems*, **5**, 2, April 1987.

[Deux90] O. Deux et al. "The Story of O_2," *IEEE Transactions on Knowledge and Data Engineering*, **2**, *1*, March 1990.

[Engl63] D. Engelbart "A Conceptual Framework for the Augmentation of Man's Intellect," in *Vistas in Information Handling*, Volume 1, P. Howerton and D. Weeks, ed., Spartan Books, London, 1963.

[Gold83] A. Goldberg and D. Robson. *Smalltalk-80 The Language and its Implementation*, Addison-Wesley. 1983.

[Katz85] R. Katz. *Information Management for Engineering Design Applications*. Springer-Verlag. 1985.

[Hagm86] R. Hagmann. "Reimplementing the Cedar File System Using Logging and Group Commit," *Proceedings of the Eleventh Symposium on Operating Systems Principles*, Aug. 1987, 155-162.

[Hala87] F. Halasz, T. Moran, and R. Trigg. "Notecards in a Nutshell," *CHI+GI 87*.

[Kim87] W. Kim, J. Banerjee, H. Chou, J. Garza, and D. Woelk. "Composite Object Support in an Object-Oriented Database System," *Proceedings OOPSLA '87*, Oct. 1987, 118-125.

[Kim90] W. Kim, J. Garza, N. Ballou, and D. Woelk. "Architecture of the ORION Next-Generation Database System," *IEEE Transactions on Knowledge and Data Engineering*, **2**, *1*, March 1990.

[Lebl85] D. Leblang, R. Chase, and G. McLean. "The DOMAIN Software Engineering Environment for Large Scale Software Development Efforts," *Proceedings 1st International Conference on Computer Workstations*, San Jose, California, Nov. 1985.

[Leff89] S. J. Leffler, M. K. McKusick, M. Karels, and J. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, May, 1989.

[Maie87] D. Mainer, and J. Stein. ``Development and Implementation of an Object-Oriented DBMS'' *Research Directions in Object-Oriented Programming*, B. Shriver and P. Wegner, ed., MIT Press, 1987. Also appears in [Zdon90].

[Moss81] E. Moss. *Nested Transactions: An Approach to Reliable Computing*. M. I. T. Report MIT-LCS-TR-260, 1981.

[Nels81] T. Nelson. *Literary Machines*. T. H. Nelson, Swarthmore, PA., 1981.

[Rash86] R. Rashid. ``Threads of a New System,'' *Unix Review*, **4**, 8, Aug. 1986.

[Spec87] A. Spector, et al. *Camelot: A Distributed Transaction Management Facility for Mach and the Internet - An Interim Report*. Carnegie-Mellon Report CMU-CS-87-129, June 1987.

[SUN86] *Network File System Protocol Specification*, Version 2, Revision B, Feb. 1986.

[Suns79] *Sunset New Western Garden Book*. Lane. 1979.

[Swin79] D. Swinehart, G. McDaniel, and D. Boggs. ``WFS: A Simple Shared File System for a Distributed Environment,'' *Operating Systems Review*, **13**, 5, November 1979.

[Swin86] D. Swinehart, P. Zellweger, R. Beach, and R. Hagmann. ``A Structural View of the Cedar Programming Environment,'' *ACM Transactions on Programming Languages and Systems*, **8**, 4, October 1986.

[Weis89] M. Weiser, A. Demers, and C. Hauser. ``The Portable Common Runtime Approach to Interoperability,'' *Twelfth ACM Symposium on Operating System Principles*, which is printed as *Operating Systems Review*, **23**, 5, December 1989.

[Wilk90] K. Wilkinson, P. Lyngback, and W. Hasan. ``The Iris Architecture and Implementation,'' *IEEE Transactions on Knowledge and Data Engineering*, **2**, 1, March 1990.

[Zdon90] S. Zdonik and D. Maier, ed., *Readings in Object-Oriented Database Systems*. Morgan Kaufman, 1990.

XEROX

The Yggdrasil Project: Motivation and Design

Robert B. Hagmann

CSL-91-13