WebBase : A repository of web pages

Jun Hirai Sriram Raghavan Hector Garcia-Molina Andreas Paepcke
{hirai, rsram, hector, paepcke}@db.stanford.edu
Computer Science Department
Stanford University
Stanford, CA 94305

Abstract

In this paper, we study the problem of constructing and maintaining a large shared repository of web pages. We discuss the unique characteristics of such a repository, propose an architecture, and identify its functional modules. We focus on the storage manager module, and illustrate how traditional techniques for storage and indexing can be tailored to meet the requirements of a web repository. To evaluate design alternatives, we also present experimental results from a prototype repository called ***WebBase***, that is currently being developed at Stanford University.

Keywords : Repository, WebBase, Architecture, Storage management

1 Introduction

A number of important applications require local access to substantial portions of the web. Examples include traditional *text search engines* [Google] [Avista], *related page* services [Google] [Alexa], and *topic-based search and categorization* services [Yahoo]. Such applications typically access, mine or index a local cache or *repository* of web pages, since performing their analyses directly on the web would be too slow. For example, the Google search engine [Google] computes the PageRank [BP98] of every web page by recursively analyzing the web's link structure. The repository receives web pages from a *crawler*, which is the component responsible for mechanically finding new or modified pages on the web. At the same time, the repository offers applications an access interface (API) so that they may efficiently access large numbers of up-to-date web pages.

In this paper, we study the design of a large shared repository of web pages. We present an architecture for such a repository, we consider and evaluate various implementation alternatives, and we describe a prototype repository that is being developed as part of the *WebBase* project at Stanford University. The prototype already has a collection of around 25 million web pages (amounting to about 210GB of HTML) and is being used as a testbed to study different storage, indexing, and data mining techniques. An earlier version of the prototype was used as the backend storage system of the Google search engine. The new prototype is intended to offer parallelism across multiple storage computers, and support for a wider variety of applications (as opposed to just text-search engines). The prototype does not currently implement all the features and components that we present in this paper, but the most important functions and services are already in place.

A web repository stores and manages a large collection of data "objects," in this case web pages. It is conceptually not that different from other systems that store data objects, such as file systems, database management systems, or information retrieval systems. However, a web repository does not need to provide a lot of the functionality that the other systems provide, such as transactions, or a general directory naming structure. Thus, the web repository can be optimized to provide just the essential services, and to provide them in a scalable and very efficient way. In particular, a web repository needs to be tuned or targeted to provide:

***Scalability*:** Given the size and the growth of the web [LG99], it is paramount that the repository scale to very large numbers of objects. The ability to seamlessly distribute the repository across a cluster of computers and disks is essential. Of particular interest to us is the use of *network disks* to hold the repository. A network disk is a disk, containing a processor, and a network interface that allows it to be connected directly to a network. Network disks provide a simple and inexpensive way to construct large data storage arrays, and may therefore be very appropriate for web repositories. (A donation of a large number of network disks by Quantum Systems has made it possible for us to explore this strategy for assembling large web repositories.)

***Streams*:** While the repository needs to provide access to individual stored web pages, the most demanding access will be in bulk, to large collections of pages, for indexing or data mining. Thus the repository must support *stream* access, where for instance the entire collection is scanned and fed to a client for analysis. Eventually, the repository may need to support ordered streams, where pages can be returned at high speed in some order. (For instance, a data mining application may wish to examine pages by increasing modified date, or in decreasing page rank.)

***Large updates*:** The web changes rapidly [LG99]. Therefore, the repository needs to handle a high rate of modifications. As new versions of web pages arrive, the space occupied by old versions must be reclaimed (unless a history is maintained, which we do not consider here). This means that there will be substantially more space compaction or reorganization than in most file or data systems. The repository must have a good strategy to avoid excessive conflicts between the update process and the applications accessing pages.

***Expunging Pages*:** In most file or data systems, objects are explicitly deleted when no longer needed. However, when a web page is removed from a web site, the repository is not notified. Thus, the repository must have a mechanism for detecting obsolete pages and removing them. This is akin to "garbage collection" except that it is not based on reference counting.

In this paper we study how to build a web repository that can meet these requirements. In particular,

- We propose a repository architecture that supports the required functionality and high performance. This architecture is amenable to the use of, but does not require, network disks.
- We present alternatives for distributing web pages across computers and disks. We also consider different mechanisms for staging the new pages provided by the crawler, as they are applied to the repository.
- We consider ways in which the crawler and the repository can interact, including through batch updates, or incremental updates.
- We study strategies for organizing the web pages within a "node" or computer in the system. We consider how space compaction or reorganization can be performed under each scheme, and we discuss how to select optimal disk "bucket" sizes.
- We present experimental results from our prototype, as well as simulated comparisons between some of the approaches. This sheds light on the design choices that are available.

Our goal is to cover a wide variety of techniques, but to keep within space limitations, we are forced to make some restrictions in scope. In particular, we do make the following assumptions about the operations of the crawler and the repository. Other alternatives are interesting and important, but simply not covered here.

- We assume that the crawler is *incremental* [CGM00] and does not visit the entire web each time it runs. Rather, the crawler merely visits those pages that it believes have changed or been created since the last run. Such crawlers scale better as the web grows.
- The repository does not maintain a temporal history (or versions) of the web. In other words, only the latest version of each web page must be retained in the repository.
- The repository stores only standard HTML pages. All other media and document types are ignored by the crawler.
- Finally, indexes are constructed using a snapshot view of the contents of the repository. In other words, the indexes represent the state of the repository between two successive crawler runs. They are updated only at the end of each crawler run and not incrementally.

The rest of this paper is organized as follows. In Section 2, we present an architectural description of the various components of the repository, while in Section 3 we concentrate on one of the fundamental components of the architecture- namely the *storage manager*. In Section 4 we present results from experiments conducted to evaluate various options for the design of the storage manager, while in Section 5 we survey some related work. Finally, we conclude in Section 6.

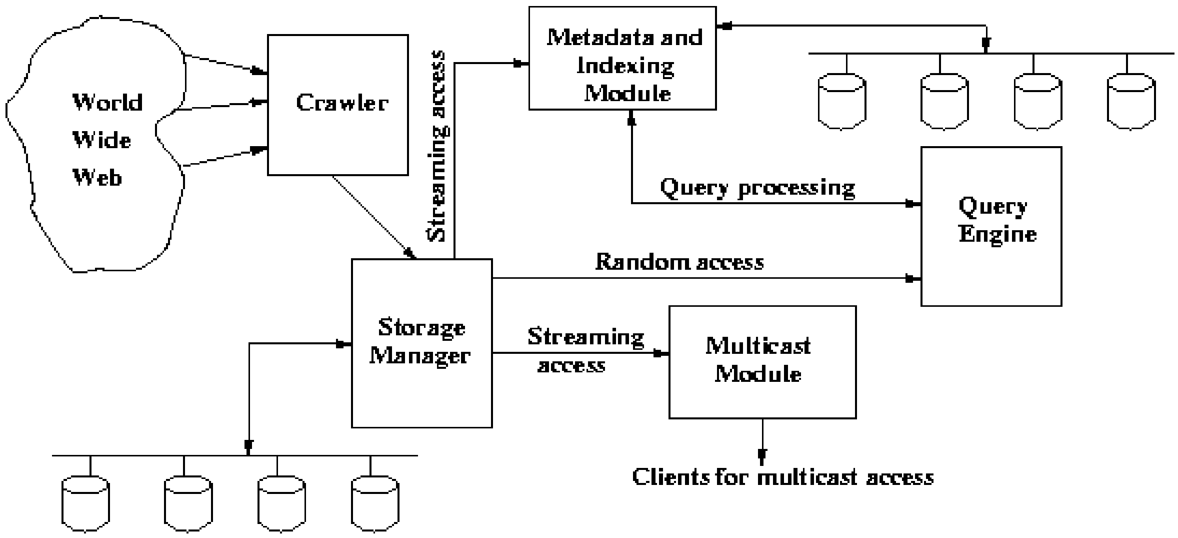

Figure 1: WebBase Architecture

2 Architecture

Figure 1 depicts the architecture of the WebBase system in terms of the main functional modules and their interactions. It shows the five main modules - the crawler, the storage manager, the metadata and indexing module, the multicast module, and the query engine. The connections between these modules represent exchange of information in the indicated directions.

The crawler module is responsible for retrieving pages from the web and handing them to the storage management module. The crawler periodically goes out to the web to retrieve fresh copies of pages already existing in the repository, as well as pages that have not been crawled before. The storage module performs various critical functions that include assignment of pages to storage devices, handling updates from the crawler after every fresh crawl, and scheduling and servicing various types of requests for pages. Our focus in this paper will be on the storage module, but in this section we provide an overview of all the components.

The metadata and indexing module is responsible for extracting metadata from the collected pages, and for indexing both the pages and the metadata. The metadata represents information extracted from the web pages, for example, their title, creation date, or set of outgoing URLs. It may also include information obtained by analyzing the entire collection. For instance, the number of incoming links for each page (coming from other pages), or citation count, can be computed and included as metadata. The module also generates indexes for the metadata and for the web pages. The indexes may include traditional full text indexes, as well as indexes on metadata attributes. For example, an index on citation count can be used to quickly locate all web pages having more than say 100 incoming links. The metadata and indexes are stored on separate devices from the main web page collection, in order to minimize access conflict between page retrieval and query processing. In our prototype implementation simple metadata attributes are stored and indexed using a relational database.

The query engine and the multicast module together provide access to the content stored in the repository. Their roles are described in the following subsection.

2.1 Access Modes

The repository supports three major access modes for retrieving pages:

- Random access
- Query-based access
- Streaming access

Random access: In this mode, a specific page is retrieved from the repository by specifying the URL (or some other unique identifier) associated with that page.

Query-based access: In this mode, requests for a set of pages are specified by queries that characterize the pages to be retrieved. These queries may refer to metadata attributes, or to the textual content of the web pages. For example, suppose the indexing module maintains one index on the words present in the title of a web page and another index on the hypertext links pointing out of a given page. These indexes could together be used to respond to a query such as: "Retrieve the URLs of all pages which contain the word Stanford in the title and which point to http://www-db.stanford.edu/". The query engine (shown in Figure 1) is responsible for handling all such query-based accesses to the repository.

Streaming access: Finally, in the streaming access mode, all, or at least a substantial portion, of the pages in the repository are retrieved and delivered in the form of a data stream directed to the requesting client application. This access mode is unique to a web repository and is important for applications that need to deal with a large set of pages. For example, many of the search applications mentioned in Section 1 require access to millions of pages to build their indexes or perform their analysis. In particular, within the WebBase system, the streaming interface is used by the indexing module to retrieve all the pages from the repository and build the necessary indexes.

The multicast module in Figure 1 is responsible for handling all external requests for streaming mode access. In particular, multiple clients may make concurrent stream requests. Therefore, if the streams are organized properly, several clients may share the transmitted pages. Our goal for the WebBase repository is to make the streams available not just locally, but also to remote applications at other institutions. This would make it unnecessary for other sites to crawl and store the web themselves. We believe it will be much more efficient to multicast streams out of a single repository over the Internet, as opposed to having multiple applications do their own crawling, hitting the same web sites, and on many occasions requesting copies of the same pages.

Initially, WebBase supports stream requests for the entire collection of web pages, in an arbitrary order that best suits the repository. WebBase also supports *restartable streams* that give a client the ability to pause and resume a stream at will. This requires that state information about a given stream be continuously maintained and stored at either the repository or the client, so that pages are not missed or delivered multiple times. Stream requests will be extended to include requests for subsets of pages (e.g., get all pages in the ".edu" domain) in arbitrary order. Eventually, we plan to introduce order control, so that applications may request particular delivery orders (e.g., pages in increasing page rank). We are currently investigating strategies for combining stream requests of different granularities and orders, in order to improve data sharing across clients.

2.2 Page identifier

Since a web page is the fundamental logical unit being managed by the repository, it is important to have a well-defined mechanism that all modules can use to uniquely refer to a specific page. In the WebBase system, a page identifier is constructed by computing a signature (e.g., checksum or cyclic redundancy check) of the URL associated with that page. However, a given URL can have multiple text string representations. For example, *http://www.stanford.edu:80/* and *http://www.stanford.edu* both represent the same web page but would give rise to different signatures. To avoid this problem, we first *normalize* the URL string and derive a *canonical representation*. We then compute the page identifier as a signature of this canonical representation. The details are as follows:

- **Normalization:** A URL string is normalized by executing the following steps:
  - Removal of the protocol prefix (*http://*) if present
  - Removal of a *:80* port number specification if present (However, non-standard port number specifications are retained)
  - Conversion of the server name to lower case
  - Removal of all trailing slashes ("/")
- The resulting text string is ***hashed*** using a signature computation to yield a 64-bit page identifier.

The use of a hashing function implies that there is a non-zero collision probability. Nevertheless, a good hash function along with a large space of hashes makes this a very unlikely occurrence. For example, with 64 bit identifiers and 100 million pages in the repository, the probability of collision is 0.0003. That is, 3 out of 10,000 repositories would have a collision. With 128 bit identifiers and a 10 billion page collection, the probability of collision is 10^{-18}. See [CGM98] for more discussion and a derivation of a general formula for estimating collisions.

3 Storage Manager

In this section we discuss the design of the storage manager. This module stores the web pages on local disks, and provides facilities for accessing and updating the stored pages. The storage manager stores the latest version of every page retrieved by the crawler. Its goal is to store *only* the latest version (not a history) of any given page. However, two issues require consideration :

- *Consistency of indexes:* A page that is being referenced by one or more indexes must not be removed from the repository even if a later version of the same page has been retrieved by the crawler. For all such pages, two versions might need to temporarily co-exist until the indexes can be modified. This requirement impacts the functioning of the various update schemes and we defer a discussion of this issue to Section 3.3.
- *Expunging pages:* The storage manager is free to expunge pages that no longer exist on the web. Since the crawler does not explicitly indicate what pages have been removed from web sites, it is the responsibility of the storage manager to ensure that old copies of non-existent pages are periodically expunged.

Traditional garbage collection algorithms reclaim space by discarding objects that are no longer referenced. The inherent assumption is that all data objects are available for testing, so that non-referenced objects can be identified. This differs from our situation, where the aim is to detect, as soon as possible, whether an object has been deleted in a remote location (a web site) so that it can be similarly deleted from a local copy (the repository). To do this cleanup, the storage manager associates two numerical values with each page in the repository - *allowed lifetime* and *lifetime count*. Allowed lifetime represents the time a page can remain in the repository without being refreshed or replaced. When a page is crawled for the first time, it's lifetime count is set to the allowed lifetime. Also, whenever a new version of the page is received from the crawler, the lifetime count is again reset to the allowed lifetime. Otherwise, the lifetime count of all pages is regularly decremented to reflect the amount of time for which they have been in the repository. Periodically, the storage manager runs a background process that constructs a list of URLs corresponding to all those pages whose lifetime count is about to reach 0. It forwards the list to the crawler, which attempts to visit each one of those URLs during the next crawling cycle. Those URLs in the list for which no pages are received from the crawler during the next update cycle are removed from the repository. If the crawler indicates that it was unable to verify the existence of a certain page, possibly because of network problems, then that page is not expunged. Instead, it's lifetime count is set to a very small value to ensure that it will be included in the list next time around.

Note that the crawler has it's own parameters [CGM00] for deciding the periodicity with which individual pages are to be crawled. The list provided by the storage manager is only in addition to the pages that the crawler already intends to visit.

For scalability, the storage manager must be distributed across a collection of ***storage nodes***, each equipped with a processor, one or more disks, and the ability to connect to a high-speed communication network. (For WebBase, each node can either be a network disk, or a regular computer.) To coordinate the nodes, the storage manager employs a central *node management server* . This server maintains a table of parameters describing the current state of each storage node. The parameters include:

- Total storage capacity, occupied space, and free space on each node
- Extent of fragmentation on each storage device
- Current state of the node - possible states include *down*, *idling*, *streaming*, and *storing* (the significance of each of these states will become clear once the update operations are presented)
- Number of outstanding requests for page retrieval and their types (random access, query-based, or streaming mode)

Based on this information, the node management server allocates storage nodes to service requests for stream accesses. It also schedules and controls the integration of freshly crawled pages into the repository. In the remainder of this section we discuss the following design issues for the storage manager:

- ***Distribution of pages*** among the storage nodes (Section 3.1)
- ***Organization of pages*** on each storage device for maximum efficiency during streaming and random access (Section 3.2)
- ***Update mechanism*** to integrate freshly crawled pages into the system (Section 3.3)

3.1 Page distribution across nodes

We consider two policies for distributing pages across multiple storage nodes.

- **Uniform distribution:** All storage nodes are treated identically; any page can be assigned to any of the nodes in the system.
- **Hash distribution:** The page identifier (computed as the signature of the URL as described in Section 2.2) is used to decide the allocation of pages to storage nodes. Each storage node is associated with a range of identifiers and contains all the pages whose identifiers fall within that range.

The hash distribution policy requires only a very sparse global index to locate the node in which a page with a given identifier would be located. This global index could in fact be implicit, if we interpret some portion (say the high order n bits) of the page identifier as denoting the number of the storage node to which the page belongs. In comparison, the uniform distribution policy requires a dense global index that maps each page identifier to the node containing the page. On the other hand, by imposing no fixed relationships between page identifiers and nodes, the uniform distribution policy simplifies the addition of new storage nodes into the system. With hash-based distribution, this would require some form of "extensible hashing". For the same reason, the uniform distribution policy is also more robust to failures. Failure of one of the nodes, when the crawler is providing new pages, can be handled by allocating all new incoming pages to the remaining nodes. With hashing, if an incoming page falls within a failed node, special recovery measures will be called for.

3.2 Organization of pages on disk

Each storage node must be capable of efficiently supporting three operations: page addition, high-speed streaming, and random page access. In this subsection we describe two ways to organize the data within a node to support these operations: *hash-based organization* and *log-structured organization*. We defer an analysis of the pros and cons of these methods to a later section where we describe experiments that aid in the comparison.

3.2.1 Hash-based organization

Hash-based organization treats each disk as a collection of hash buckets. Each bucket is small enough to be read into memory, processed, and written back to disk. Each bucket stores pages that are allocated to that node and whose identifiers fall within the bucket's range. Note that this range is different from the range of page identifiers allocated to the storage node as a whole according to the hash distribution policy of Section 3.1. We assume that the hash-based organization uses a fixed hashing scheme with bucket overflows being handled by separately allocated *overflow buckets*. Buckets that are associated with successive ranges of identifiers are also assumed to be physically continuous on disk (excluding overflow buckets if any). Also, we assume that within each bucket, pages are stored in increasing order of their identifiers. Note that at any given time, only a portion of the space allocated to a hash bucket will be occupied by pages - the rest will be free space. Given such an organization, let us consider the three fundamental operations:

Random page access: Random page access can be supported without any additional local index to map a page identifier to the physical location. It is straightforward to identify the bucket containing the page, to read it into memory, and then conduct a main-memory search to locate the required page.

Streaming: A streaming request asks for all or a subset of the pages at the node. If no particular order is required, then streaming can be supported efficiently by sequentially reading the buckets from disk (in the order of their physical locations) into memory, and transmitting the pages to the client. The effective streaming rate will be some fraction of the maximum disk transfer rate, with the fraction being determined by the amount of utilized space in each bucket.

Page addition: The performance of hash-based organization for this operation depends on the order in which the pages are received. If the crawler sends new pages in a purely random order, then each page addition will require one read of the relevant bucket, followed by an in-memory update, and then a disk write to flush the modified bucket back to disk. Space used by old, unwanted pages can be reclaimed as part of this process.

On the other hand, if pages are received in the **order** of their page identifiers, a more efficient method is possible. (This means that the crawler, or whoever is sending the new pages, must somehow order the pages before transmitting them to the storage node.) In particular, as each bucket is read from disk, a batch of new pages can be added, and then written to disk. (The in-memory addition is simple since the incoming and the stored pages are in order.) If main memory is available, more than one bucket can be read into memory and merged with the incoming pages, allowing each disk operation to be amortized among even more pages.

3.2.2 Log-structured organization

The log-structured page organization is based on the same principles as the Log-structured File System (LFS) described in [RO91]. New pages received by the node are simply appended to the end of a log, making the process very efficient. To be more specific, the storage node maintains either two or three objects on each disk:

- A large *log* that occupies most of the space available on disk and which includes all the pages allocated to that disk as a single continuous chunk
- A *catalog* that contains one entry corresponding to each page present in the log. A typical catalog entry includes the following information:
    - Identifier of the page in question
    - Pointer to the physical location of the page within the log
    - Size of the page
    - Status of the page (*valid, marked,* or *deleted* - the semantics of these states will be clear once the update strategies are discussed)
    - Timestamp denoting the time when the page was added to the repository
- If random access to a page is required, then a local *B-tree index* that maps a given page identifier

to the corresponding location of the page, is also maintained.

For typical network or PC disk sizes and average web page sizes, the number of pages in the log is small enough that only the leaves of the B-tree need to reside on disk. Therefore, from now on, we will assume that only one disk access is required to retrieve an entry from the B-tree index.

Page addition: New pages are appended to the end of the log. If we assume that the catalog and B-tree do not necessarily have to be kept continuously up to date on disk, then batch mode page addition is extremely efficient since it involves successively writing to contiguous portions of the disk. The required modifications to the catalog are buffered in memory and periodically flushed to disk. The page addition rate improves with increase in the amount of main memory available for buffering these modifications. Log space must eventually be compacted, to remove old, unwanted pages. Also, once page addition completes, the B-tree index must be updated.

Random access: Requires two disk accesses, one to read the appropriate B-tree index block and retrieve the physical position of the page, and another to retrieve the actual page.

Page streaming: If no particular streaming order is required, then the log-structured organization is very efficient in streaming out pages as the log merely has to be read sequentially. Note that this assumes that all the pages in the log at the time of streaming are "valid". If this is not true, then additional disk accesses will be needed to examine the catalog and discard pages whose status flag is not set to "valid".

3.3 Update Schemes

We assume that updates proceed in cycles. The crawler collects a set of new pages, these are incorporated into the repository, the metadata indexes are built for the new snapshot, and a new cycle begins. Under this model, there is a period of time, between the end of a crawl and the completion of index rebuilding, when older versions of pages (that are being referenced by the existing index) need to be retained. This will ensure that ongoing page retrieval requests, either through query-based access or streaming mode access, are not disrupted. Thus, we classify all pages in the repository as:

- **Class A:** Old versions of pages (referenced by the current active indexes) whose newer versions already exist in the repository.
- **Class B:** Unchanged pages - those pages for which only one version exists because they were not crawled between the time the index was last built and the time the latest crawl was executed.
- **Class C:** These include pages not seen before, as well as new versions of pages whose older versions already exist as class A pages. In other words, all the pages received from the crawler during a crawling cycle are class C pages.

Note that even though we use the word "version" in our discussions, we are not actually comparing the two copies of a page (one in the repository and the other retrieved by the crawler during the latest crawl) to find out if the content has changed. Rather, we treat every fresh crawl of the web as defining new versions for all the pages. Therefore, whenever the crawler retrieves a page corresponding to a certain URL, if the repository contains a page with the same URL, we treat the crawled page as a new version (class C page) that will eventually replace the original version (which is now a class A page) that is already in the repository. Thus, the update process consists of the following steps:

- Receive class C pages from the crawler and add them to the repository.

- Rebuild all the indexes using the class B and class C pages.
- Delete the class A pages.

If the system does not accept random or query-based page access requests until the entire update operation is complete, it is possible to exchange the order of execution of the last two steps. In that case, the class A pages need not be retained until the indexes are rebuilt. The batch update method described in Section 3.3.1 operates in this manner.

Besides the batch update scheme, we also briefly describe an incremental update scheme in Section 3.3.2. There are additional options beyond these two. For example, one could have two full copies of the repository, and alternate between them when one copy is updated. We do not discuss these other strategies here.

3.3.1 Batch update scheme

In this update scheme, the storage nodes in the system are partitioned into two sets - ***update nodes*** and ***read nodes***. The freshly crawled class C pages are stored on the update nodes whereas class B and class A pages are stored on the read nodes. By definition, the active index set (before rebuilding) references only class A and class B pages - therefore all requests for page retrieval will involve only the read nodes. Analogously, only the update nodes will be involved in receiving and storing pages retrieved by the crawler. Figure 2 illustrates the flow of data between the crawler and the two sets of nodes during the batch update process. The steps for a batch update are as follows :

1. System isolation:

1. The multicast module stops accepting new stream requests.
2. The crawler finishes adding class C pages to the update disks.
3. Queries are suspended, and the system waits for ongoing stream transfers to complete.

2. Page transfer: Class C pages are transferred from the update nodes to the read nodes, and class A pages are removed from the read nodes. The details of these operations depend on both the page organization scheme and the page distribution policy. We discuss some examples of page transfer at the end of this section.

3. System restart:

1. The class C pages stored in the update nodes are removed. If needed, the crawler can be restarted to start populating the update nodes once more.
2. All the pages from the read nodes are streamed out to the indexing module to enable index reconstruction. External requests for streaming access can be accepted provided they do not involve access to one or more indexes.
3. Once the indexes have been rebuilt, the read nodes start accepting random and query-based requests.

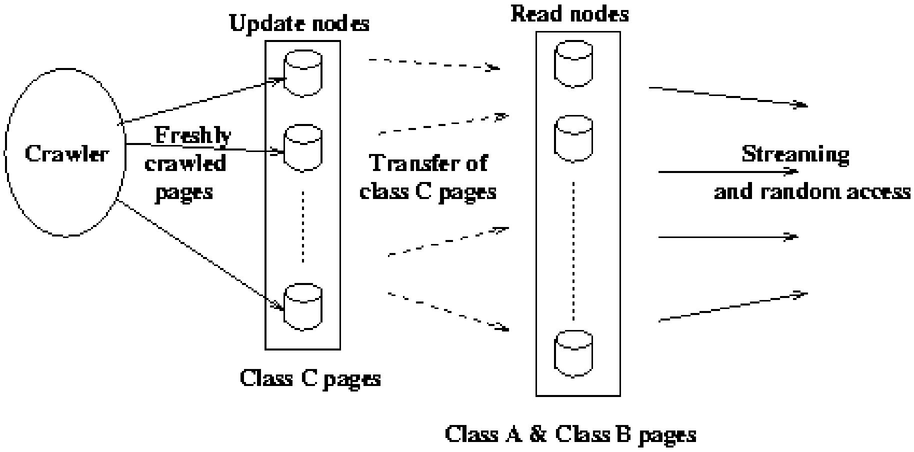

Figure 2: Batch update strategy

The exact mechanism for the transfer of pages between update and read nodes depends on the page organization and distribution policy used in each set of nodes (both the organization and the policy could be different for the two sets). In what follows, we illustrate two possible scenarios for the transfer.

Scenario 1 - Log-structured organization and Hash distribution policy on both sets : For illustration, let us assume there are 4 update nodes and 12 read nodes. The crawler computes the identifier of each new page it obtains. Pages in the first quarter of the identifier range are stored in update node 1, the pages in the second quarter go to node 2, and so on. When the crawl cycle ends, update node 1 will sub-partition its allocated identifier range into 3 subranges, and will send its pages to the first three read disks. Similarly, update disk 2 will partition its pages to read disks 4, 5 and 6, and so on. Thus, each read disk receives pages from only one update disk. The sequence of steps for transferring pages to the read nodes is as follows:

1. Each update node i constructs lists $L_{i,j}$. List $L_{i,j}$ contains the identifiers of class C pages that are currently in i and which are destined for node j.
2. Suppose read node j receives a list of pages from the update node i. It then computes $L'_{i,j}$ = $Intersection(L_{i,j}, C_j)$ where C_j denotes the list of identifiers corresponding to pages currently stored in j (note that C_j is directly available by a scan of the catalog).
3. By definition, $L'_{i,j}$ represents the set of class A pages at read node j. The catalog entries for these pages are located and their status flags are modified to indicate that they have been "deleted".
4. Next, the read nodes go into compaction mode to reclaim space created by the deletion of these class A pages.
5. Finally, each update node begins transmitting streams of class C pages to the corresponding read nodes. Each stream contains exactly the pages that are destined for the receiving read node.
6. Once all the pages have been received, each read node, if necessary, builds a local B-tree index to support random access.

Scenario 2 - Hash-based organization and Hash distribution policy on both sets : The use of a hash-based node organization allows for certain optimizations while transferring pages to the read nodes. For one, since corresponding class A and class C pages are guaranteed to be present in the same hash bucket, deletion of class A pages does not occupy a separate step, but is performed in conjunction with the addition of class C pages. The steps are as follows :

1. Each update node reads the hash buckets into memory in the order of their physical locations on disk. As a result (Section 3.2.1), the pages are read in the increasing order of their identifiers.
2. For each page retrieved from disk, the update nodes determine the read node to which the page is to be forwarded, and transmit the page accordingly.
3. Each read node begins to receive a sorted stream of pages from one of the update nodes.
4. The read nodes read their hash buckets into memory in the order of their physical locations on disk. They then execute a "merge sort" that involves both the incoming sorted stream as well the pages that they read from disk. As part of the merge sort, if a page arriving on the stream has the same identifier as one of the pages retrieved from disk, then the former is preferred and the latter is discarded. (This corresponds to replacing a class A page by the corresponding class C page). The resulting merged output is written out to disk as the modified hash buckets. Figure 3 illustrates how the merge sort is executed.

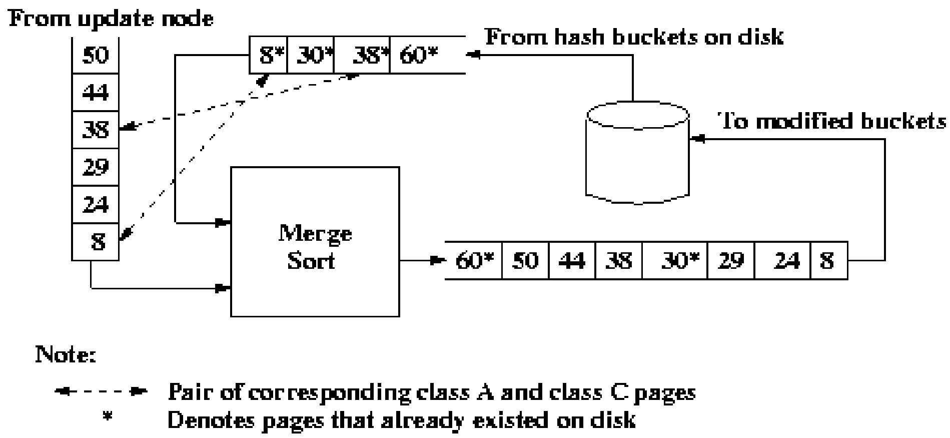

Figure 3: Addition of new pages using merge sort

Advantages of the batch update scheme: The most attractive characteristic of the batch update scheme is the lack of conflict between the various operations being executed on the repository. A single storage node does not ever have to deal with both page addition and page retrieval concurrently. Another useful property is that the physical locations of pages on the read nodes do not change between updates. (This is because the compaction operation, which could potentially change the physical location, is part of the update). This helps to greatly simplify the state information required to support restartable streams as described in Section 2.1. The required state information merely consists of a pair of values *(Node-id, Page-id)* where, *Node-id* represents some unique identifier for the storage node that contained the last page transmitted to the client before the interruption, and *Page-id* represents the identifier for that last page.

3.3.2 Incremental update scheme

In the incremental update scheme, there is no division of work between read and update nodes. All nodes are equally responsible for supporting both update and access simultaneously. The crawler is allowed to place the freshly crawled pages on any of the nodes in the system as long as it conforms to the page distribution policy. Analogously, requests for pages through any of the three access modes may involve any of the nodes present in the system. The extraction of metadata and construction of indexes are undertaken periodically, independent of the ingestion of new pages into the repository.

As a result, the incremental update scheme is capable of providing continuous service. Unlike in the batch update scheme, there is no need to isolate the system during update. The addition of new pages into the repository is a continuous process that takes place in conjunction with streaming and random access. Further, this scheme makes it possible to provide even very recently crawled pages to the clients through the streaming or random access modes (though the metadata and indexes associated with these pages may not yet be available). However, such continuous service does have its drawbacks:

Performance penalty: Performance may suffer because of conflicts between multiple simultaneous read and update operations. This performance penalty can be alleviated, to some extent, by employing the uniform distribution policy and having the node management server try to balance the loads. For example, when the node management server detects that a set of nodes are very busy responding to a number of high-speed streaming requests, it can ensure that addition of new pages from the crawler does not take place at these nodes. All such page addition requests can be redirected to other more lightly loaded nodes.

Requires dynamically maintained local index: Consider the case when the incremental update scheme is employed in conjunction with log-structured page organization. Holes created by the removal of class A pages are reclaimed through compaction which has the effect of altering the physical location of the stored pages. This makes it necessary to dynamically maintain a local index to map a given page identifier to its current physical address. This was not necessary in the batch update scheme since the physical location of all the pages in the read nodes was unaltered between two successive updates.

Restartable streams are more complicated: When incremental update is used in conjunction with hash-based page organization, the state information required to support restartable streams gets complicated, since the physical locations of pages are not preserved when there are bucket overflows. However, it turns out that in the case of log-structured organization, it is possible to execute compaction in such a way that despite incremental update, the simple state information used by the batch-update system suffices. The extended version of this paper [HRGMP99] discusses these issues in further detail.

4. Experiments

We conducted experiments to compare the performance of some of the design choices that we have presented. In this section we will describe selected results of these experiments and discuss how various system parameters influence the "best" configuration (in terms of the update strategy, page distribution policy, and page organization method) for the storage manager.

4.1 Experimental Setup

Our WebBase prototype includes an implementation of the storage manager with the following configuration:

- Update strategy: Batch update
- Page distribution policy for both update and read nodes: Hash distribution
- Page organization method used in both sets of nodes: Log-structured organization

The storage manager is fed pages by an incremental crawler that retrieves pages at the rate of approximately 50-100 pages/second. The WebBase storage manager can run on network disks or on conventional PCs. For the experiments we report here, we used a cluster of PC's connected by a 100 Mbps Ethernet LAN, since debugging and data collection are easier on PCs. In addition to the repository, we have also implemented a *client module* and a *crawler emulator*. The client module sends requests for random or streaming mode accesses to the repository, and receives pages from the read nodes in response. The crawler emulator retrieves stored web pages from an earlier version of the repository and transmits it to the update nodes at a controllable rate. Using such an emulator instead of the actual crawler provided us with the necessary flexibility to control the crawling speed, without interacting with the actual web sites.

For ease of implementation, the storage manager has been implemented on top of the standard Linux file system. In order to conform to the operating system limit on maximum file size, the log-structured organization has been approximated by creating a collection of individual log files, each approximately 512 MB in size, on each node.

To compare different page distribution and node organization alternatives, we conducted extensive simulation experiments. The simulation hardware parameters were selected to match our prototype hardware. This allowed us to verify the simulation results with our prototype, at least for the scenario that our prototype implements (batch updates, hash page distribution, log-structured nodes). For other scenarios, the simulator allows us to predict how our prototype would have performed, had we chosen other strategies. All of the performance results in this section are from the simulator, except for those in Table 4, which report the performance of the actual prototype.

We used the following performance metrics for comparing different system configurations (all are expressed in terms of number of pages/second/node):

- *Page addition rate* : This is the maximum rate at which the system is able to receive new pages and add them to the repository.
- *Random page access rate*: Random page access refers to the retrieval of a certain page from the repository by specifying the identifier associated with that page. Random page access rate is the maximum rate at which such requests can be serviced by the system.
- *Streaming rate*: refers to the rate at which all the pages in the system can be retrieved and transmitted without imposing any specific transmission order.

Note that all our performance metrics are on a per-node basis. This enables us to present results that are independent of the number of nodes in the actual system. For systems using batch-update, the inherent parallelism in the operations implies that the overall page addition rate of the system (random page access rate) is simply the per-node value multiplied by the number of update nodes (number of read nodes) if we assume that the network is not a bottleneck. For incremental update systems, the scale up is not perfectly linear because of conflict between operations. For batch update systems, an additional performance metric is the *batch update time*. This is the time during which the repository is isolated and does not provide page access services.

We present some selected experimental results below. The first set of results illustrates the optimization process that must carried out to tune the nodes to handle web data. The second set of results include a comparison of the two different page organization methods described in Section 3.2, as well as a comparison of different system configurations. The last set of results measure the performance of our implemented prototype.

4.2 Choosing a hash bucket size

For best performance, the parameters for each node storage organization (e.g., disk block sizes, memory buffer sizes) must be tuned. This tuning must be targeted for the type of data we expect (web pages, not relations or scientific data files) and its characteristics. To illustrate this process, in this section we consider the choice of "optimal" hash bucket size for a system that employs the hash distribution policy and which uses hash-based page organization at each node. The choice of a good hash bucket size must balance two conflicting requirements: large buckets lead to longer IO times, while small buckets lead to poor space utilization and hence poor streaming performance. Similar tradeoffs exist for other node organizations, but are not considered here due to space limitations.

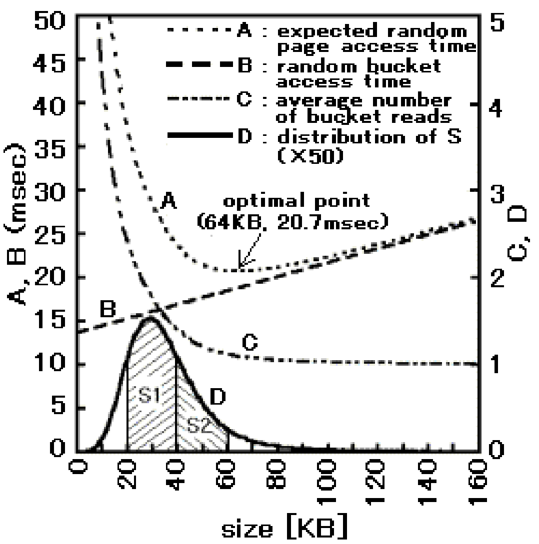

Figure 4: Optimal bucket size

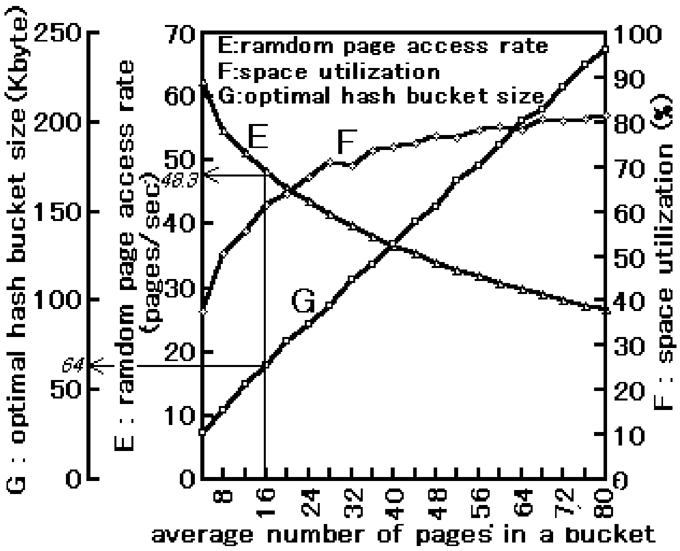

Figure 5: Space-performance tradeoff

We begin with an arbitrary choice, of 2^{20}, on the total number of hash buckets in the system. (This choice is revisited below.) Since the total range of our identifier space is 2^{64} (we are using 64-bit page identifiers), this gives each hash bucket a range of size 2^{44}. We then used a sample of $16*2^{20}$ pages (derived from our existing collection of 25 million web pages) and proceeded to assign these pages to the 2^{20} buckets based on the values of their page identifiers. This results in an average of 16 (= $16*2^{20}/2^{20}$) pages per bucket. Let *S(b)* denote the sum of the sizes of all the pages that are allocated to a bucket *b* as a result of this assignment. Then, plot *D* in Figure 4 represents the frequency distribution of *S*, expressed as a fraction of the total number of buckets (In the figure, this curve is scaled up by a factor of 50 for readability). To illustrate, suppose we wish to determine the number of buckets that have their total page sizes in the range 20 to 21 KB. Since the value of the distribution at 20 is around 0.02 (the value in the figure is 1 which yields 0.02 when scaled down by 50), the area under the curve between 20 and 21 is roughly 0.02 * (21-20) = 0.02. Therefore, the number of buckets with total page sizes in the

20-21 KB range is 0.02 * 2^{20} which is roughly 21000. The same figure also includes a plot (marked *B*) of the the average bucket access time as a function of the hash bucket size. For example, if buckets are 80KB in size, then it takes around 21.2 ms to read a bucket on our hardware.

Now, for a given hash bucket size, using distribution *D*, it is possible to determine the number of buckets that would require an overflow chain of size 1, the number of buckets that would require an overflow chain of size 2, and so on. For example, suppose we choose hash buckets of size 20 KB, then the two shaded regions S1 and S2 in Figure 4 represent the number of buckets that would require overflow chains of size 1 and 2 respectively. Therefore, we can also determine the average number of bucket reads (including overflow buckets) that would be required to randomly read a page, if the chosen bucket size was used. We performed this computation for different choices of hash bucket sizes to obtain plot *C* in Figure 4. Finally, plots *B* and *C* can be multiplied together to yield plot *A*, that depicts the variation of random page access time as a function of the hash bucket size. The optimal point in plot *A* represents the best balance between long overflow chains in the case of small buckets and high read times for reading large buckets. It indicates that if we choose to have 16 pages on an average per bucket, then we must choose a hash bucket size of 64 KB (which would give us an average random page access time of 20.7 ms).

The process described above was then repeated by choosing different values for the total number of hash buckets in the system (or equivalently, by starting with different values for the average number of pages per bucket), and calculating the average random page access time and optimal hash bucket size for each value. The result of this computation is summarized as plots *E* and *G* in Figure 5. Plot *G* shows the variation of optimal bucket size whereas plot *E* shows the variation of random page access rate (the reciprocal of random page access time). For example, the points for 16 pages per bucket correspond to our initial scenario (16 * 2^{20} pages divided into 2^{20} buckets). Hence, the *G* value at this point is 64KB for the optimal bucket size, and the *E* value is 1/20.7 = 48.3 pages per second.

Plot *F* in the same figure shows the expected space utilization as a function of bucket size. This curve was determined by using the actual pages sizes from our sample of 16 million pages. The combination of the *E* and *F* curves are a useful design aid in choosing the average number of pages per bucket for a hash-based organization. We can now clearly quantify the tradeoff between space utilization and access rate for a web repository. Clearly, as buckets grow, space utilization and streaming performance improve, but random access suffers. Depending on the requirements of the web repository, an appropriate point in this spectrum must be chosen. We stress that it is important to use a real data distribution for web pages as a starting point, so that the nodes can indeed be tuned for a web repository

4.3 Comparing different systems

In this section we present some selected results from our simulator. For simplicity we do not consider network performance, i.e., we assume that the network is always capable of handling any traffic. Performance is solely determined by the disk characteristics and the disk access pattern associated with the system configuration. The architecture of the simulator and the details of the simulation process are described in an extended version of this paper [HRGMP99].

Operation	Log-structured organization	**Hash-based organization** (1 million buckets each of size 64KB) (Average occupancy of buckets - 60%)
Streaming rate [page/sec/node]	6300	3900
Random page access rate [pages/sec/node]	35	51
Page addition rate [pages/sec/node] (pages arrive in random order)	6100	23
Page addition rate [pages/sec/node] (in random order using 10MB buffer)	6100	35
Page addition rate [pages/sec/node] (pages arrive in sorted order)	6100	1300

Table 3: Comparing page organization methods

4.3.1 Comparing log-structured and hash-based organizations

In Section 3.2 we described two node organizations, log-structured and hash-based. Table 3 compares these organizations at a single system node. Because pages are more tightly packed in a log, a log-structured node can stream pages out 62% faster than a hash-based node. However, the hash-based node does not have to use a local index for random reads, so it can read pages at a higher rate (46% higher). A log-structured node can clearly append new pages at a much higher rate than a hash-based node. Increasing the available memory to 10 MB improves the add rate for the hash-based node from 23 to 35 pages/sec., still way under the performance of the log. (The observed improvement is merely because buffering allows the use of a single disk sweep (the "elevator algorithm") to update all the hash buckets.) On the other hand, if pages are received in sorted order (by identifier), then the merging technique of Section 3.2.1 can improve the page addition performance of the hash-based organization by almost two orders of magnitude.

Hashed-log organization: The results of Table 3 suggest that in a batch system, update nodes should be log-structured to support a large throughput from the crawler, while read nodes should be hash-based to support high read traffic. However, to achieve good performance at the read nodes during update, pages transferred to them from the update nodes should arrive in sorted order. Unfortunately, a log-structured update node will have difficulty generating the sorted stream.

This suggests a hybrid node organization for the update nodes, which we call *hashed-log*. With this scheme, a disk contains a number of large logs, instead of a single log as in the pure log-structured organization. Each of these individual logs is typically about 8-10 MB in size and therefore much larger than typical hash buckets. However, since update nodes do not have to support random page access, this large size is not a problem. Like hash buckets, each log file is associated with a range of hash values and stores all pages that are assigned to that node and whose page identifier falls within that range. Pages received from the crawler are buffered in memory to the extent possible and then appended to the appropriate logs. Buffering has a more significant impact here than in pure hash-based organization. Since the number of logs is much smaller than the number of hash buckets, it is more likely that multiple incoming pages are mapped to the same log. These pages can then be written to the log in one stroke, allowing us to amortize the cost of a disk seek among all these pages. Further, page addition requires only a write of the actual page whereas in the hash-based organization, each page addition requires reading in all the pages in the bucket and then writing the whole bucket back to disk. Our experiments [HRGMP99] indicate that a hashed-log organization using 10 MB logs and using a memory buffer of size 10 MB is able to provide a page addition rate of 660 pages/sec, an order of magnitude better than that supported by a hash-based organization. During page transfer from update nodes to read nodes, logs can be read in the order of their associated hash range values, sorted in memory, and transmitted, thereby resulting in a sorted stream.

4.3.2 Comparing different configurations

To compare different system configurations, we require a notation to easily refer to a specific configuration. We adopt the following convention:

- Incr[*p*, *o*]: denotes a system that uses the incremental update scheme, policy *p* for distributing pages across nodes, and organization method *o* to organize pages within each node. Here, *p* can be either "hash" or "uniform" and *o* can be either "hash" or "log".
- Batch[U(*p1, o1*), R(*p2, o2*)]: denotes a system that uses the batch update scheme, uses policy *p1* and organization method *o1* on the update nodes, and policy *p2* and organization method *o2* on the read nodes.

For example, the system configuration for our prototype (as discussed in Section 4.1) can be represented as Batch[U(hash,log), R(hash,log)].

Table 4 presents some sample performance results for three different system configurations that use the batch update method, employ the hash distribution policy, and use hash-based page organization at the update nodes. For this experiment we assume that 25% of the pages on the read nodes are replaced by newer versions during the update process. We call this an *update ratio* of 0.25. The three configurations differ in the organization method that they employ at the update nodes. The center column gives the page addition rate supported by a single update node (derived earlier). If we multiply these entries by the number of update nodes we get the total rate at which the crawler can deliver pages. The third column gives the total time to perform a batch update of the read disks, and represents the time the repository would be unavailable to clients. The last configuration, which uses a hashed-log organization at the update nodes, provides the best balance between page addition rate and a reasonable batch update time. Note that because of the parallelism available in the batch update systems, the update time does not depend on the number of nodes but is purely determined by the update ratio.

System configuration	Page addition rate per update node [pages/sec]	Batch update time (update ratio=0.25)
Batch [U(hash, log), R(hash, hash)]	6100	5.35 hrs
Batch [U(hash, hash), R(hash, hash)]	35	1260 secs
Batch [U(hash, hashed-log), R(hash, hash)]	660	1260 secs

Table 4: Sample performance results for different configurations

4.4 Experiments on overall system performance

Table 5 summarizes the results of experiments conducted directly on our prototype. Since our prototype employs a log-structured organization on both sets of nodes, it exhibits impressive performance for both streaming and page addition. Note that the results of Table 5 include network delays, and hence the numbers are lower than those predicted by Table 3. In particular, the streaming rate is measured at our client module, and the page addition rate is what the emulated crawler sees.

Streaming rate	2800 pages/sec per read node
Page addition rate	3200 pages/sec per update node
Batch update time	2451 seconds (for update ratio = 0.25)
Random page access rate	33 pages/sec per read node

Table 5: Performance of prototype

Figure 6 plots the variation of batch update time with *update ratio* for our prototype. As before, the update ratio refers to the fraction of pages on the read nodes that are replaced by newer versions. Our prototype system uses a batch update process with stages corresponding to Scenario 2 of Section 3.3.1. Figure 6 shows how each stage contributes to the overall batch update time. (Note that the y-axis in Figure 6 is cumulative, i.e., each curve includes the sum of the contributions of all the stages represented below it). For example, for an update ratio of 0.25, we see that catalog update, page identifier transfer, and B-tree construction require only 26, 84, and 88 seconds respectively. However, compaction requires 1244 secs whereas page transfer requires 1008 seconds. The domination of compaction and page transfer holds at all update ratios. In addition, the figure shows that the time for page transfer remains almost constant, independent of the update ratio. This is because an increase in update ratio requires a corresponding increase in the number of update nodes to accommodate the larger number of pages being received from the crawler. Since page transfer is an operation that each update node executes independently and simultaneously, we are able to achieve perfect parallelism and keep the page transfer time constant. Compaction, on the other hand, exhibits a marked decrease with increase in update ratio. This is reasonable, since at higher update ratios, more class A pages are deleted, thereby leaving behind smaller amounts of data to be moved around on the read nodes during compaction.

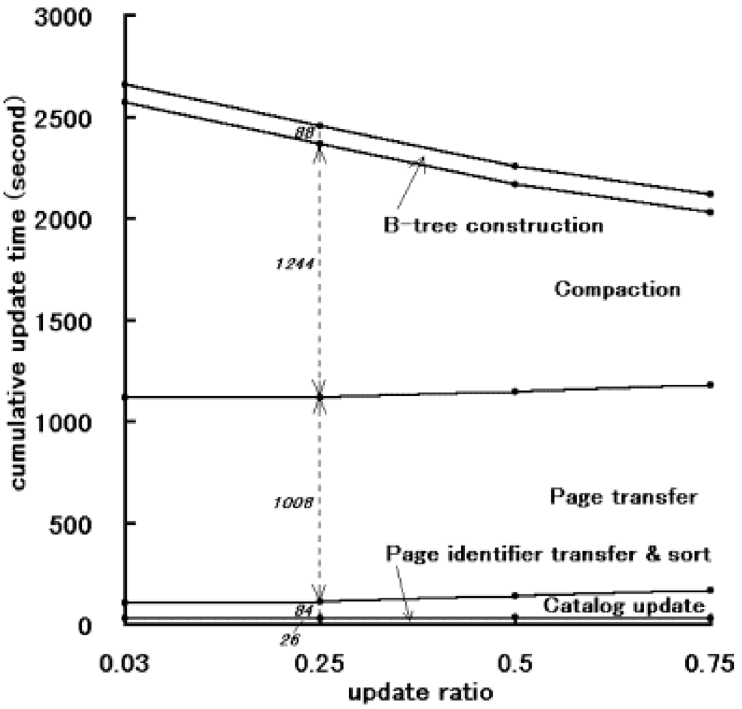

Figure 6: Batch update time of prototype

System configuration	Streaming	Random page access	Page addition	Update time
Incr [hash, log]	+	-	--	inapplicable
Incr [uniform, log]	+	--	+	inapplicable
Incr [hash, hash]	+	+	-	inapplicable
Batch [U(hash, log), R(hash, log)]	++	-	+	+
Batch [U(hash, log), R(hash, hash)]	+	+	+	-
Batch [U(hash, hash), R(hash, hash)]	+	+	-	+
Batch [U(hashed-log, hash), R(hash, hash)]	+	+	+-	+

Table 6: Relative performance of different system configurations

4.5 Summary

There is a wide spectrum of system configurations for a web repository, each with different strengths and weaknesses. The choice of an appropriate configuration is influenced by the deployment environment as well as by the functional requirements. Some of the factors that influence this choice are crawling speed, required random page access performance, required streaming performance, node computing power and storage space, and the importance of continuous service. For example, in an environment that includes a high-speed crawler, configurations that perform poorly on page addition, such as Incr[hash, hash] or Batch[U(hash,hash), R(*,*)], are not suitable. Similarly, if continuous service is essential in a certain environment, then none of the batch update based schemes presented in this paper would be applicable. Table 6 presents a summary of the relative performance of some of the more useful system configurations. In that table, the symbols ++, +, +-, -, and -- represent, in that order, a spectrum of values from the most favorable to the least favorable for a given performance metric.

5. Related work

From the nature of their services, one can infer that all web search engines either construct, or have access to, a web repository. However, these are proprietary and often specific to the search application. In this paper, we have attempted to discuss, in an application-independent manner, the functions and features that would be useful in a web repository, and have proposed an architecture that provides these functions efficiently.

A number of web-based services have used web repositories as part of their system architecture. However, often the repositories have been constructed on a much smaller scale and for a restricted purpose. For example, the WebGUIDE system [DBCK96] allows users to explore changes to the World Wide Web and web structure by supporting recursive document comparison. It tracks changes to a user-specified set of web pages using the AT&T Difference Engine (AIDE) [DB96] and provides a graphical visualization tool on top of AIDE. The AIDE version repository retrieves and stores only pages that have explicitly been requested by users. As such, the size of the repository is typically much smaller than the sizes targeted by WebBase. Similarly, GlimpseHTTP (now called WebGlimpse) [MSG97] provides text-indexing and "neighborhood-based" search facilities on existing repositories of web pages. Here again, the emphasis is more on the actual indexing facility and much less on the construction and maintenance of the repository.

The Internet Archive [IArch] project aims to build a digital library for long-term preservation of web-published information. The focus of that project is on addressing issues relevant to archiving and preservation. Their target client population consists of scientists, sociologists, journalists, historians, and others who might want to use this information in the future for research purposes. On the other hand, our focus with WebBase has been on architecting a web repository in such a way that it can be kept relatively fresh, and be able to act as an immediate and current source of web information for a large number of existing applications.

6. Conclusion

In this paper we proposed an architecture for structuring a large shared repository of web pages. We argued that the construction of such a repository calls for judicious application of new and existing techniques. We discussed the design of the storage manager in detail and presented qualitative and experimental analysis to evaluate the various options at every stage in the design.

Our WebBase prototype is currently being developed based on the architecture of Section 2. Currently, working implementations of an incremental crawler, the storage manager, the indexing module, and a query engine are available. Most of the low-level networking and file-system operations have been implemented in C/C++ whereas the query interface has been implemented in Java.

For the future, we plan to implement and experiment with some of the more advanced system configurations that we presented in Section 4.3.2. We also plan to develop advanced streaming facilities, as discussed in Section 2.1, to provide more client control over streams. Eventually, we plan to enhance WebBase so that it can maintain a history of web pages and provide temporal information.

References

[Alexa] *Alexa Incorporated*, http://www.alexa.com

[Avista] *Altavista Incorporated*, http://www.altavista.com

[BP98] Sergey Brin and Larry Page, *The Anatomy of a Large-Scale Hypertextual Web Search Engine*, Proc. of the 7th Intl. WWW Conference, April 14-18, 1998.

[CGM00] Junghoo Cho and Hector Garcia-Molina, *Incremental Crawler and Evolution of the Web*, submitted to the 9th Intl. WWW Conference, 2000.

[CGM98] Arturo Crespo and Hector Garcia-Molina, *Archival Storage for Digital Libraries*, Third ACM Conference on Digital Libraries, June 23-26, 1998.

[DB96] Fred Douglis and Thomas Ball, *Tracking and viewing changes on the web*, Proc. of the 1996 USENIX Technical Conference, January 1996.

[DBCK96] Fred Douglis, Thomas Ball, Yih-Farn Chen, and Eleftherios Koutsofios, *WebGUIDE: Querying and Navigating Changes in Web Repositories*, Proc. of the 5th Intl. WWW Conference, May 6-10, 1996.

[Google] *Google Incorporated*, http://www.google.com

[HRGMP99] Jun Hirai, Sriram Raghavan, Hector Garcia-Molina, and Andreas Paepcke, *WebBase : A repository of pages*, Stanford Digital Libraries Project Technical Report SIDL-WP-1999-0124, Computer Science Dept, Stanford University, Nov. 1999. Available at http://www-diglib.stanford.edu/diglib/WP/PUBLIC/DOC319.html

[IArch] *Internet Archive*, http://www.archive.org

[LG99] Steve Lawrence and C. Lee Giles, *Accessibility of Information on the Web*, Nature, vol. 400, July 8 1999.

[Lycos] *Lycos Incorporated*, http://www.lycos.com

[MSG97] Udi Manber, Mike Smith, and Burra Gopal, *WebGlimpse - Combining Browsing and Searching* , Proc. of the 1997 USENIX Technical Conference, Jan 6-10, 1997.

[RO91] Mendel Rosenblum and John K. Ousterhoust, *The design and Implementation of a Log-Structured File System*, Proc. of the 13th ACM Symposium on Operating Systems Principles, pages 1-15, Oct.1991.

[Yahoo] *Yahoo Incorporated*, http://www.yahoo.com