479

Storage Structures for Versions and Alternatives

Randy H. Katz and Tobin J. Lehman
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706

ABSTRACT: We identify the roles played by versions and design alternatives in an engineering database. The obvious way to implement versions is to maintain each in a separate collection of files. Because several versions must be kept on-line in a design environment, the approach leads to large disk requirements. We develop B-tree based storage structures to encode versions as "negative" differential files. Our objective is to keep the disk requirements small. We discuss the effect of enormous amounts of cheap archival storage (write-once optical digital disks) on the proposed structures. We are implementing versions in the Wisconsin Storage System (WiSS), an experimental database component under development at the University of Wisconsin-Madison.

1. Introduction

Until recently, the database research community has been concerned primarily with providing effective data management facilities for conventional data processing applications, viz., banking, accounting, and so forth. New directions include database facilities for non-traditional problem domains, such as scientific data analysis [BORA82], the automated office [ELLI80], and computer-aided design [HASK82, KATZ82].

We are particularly interested in applying database methods to support design activities. A design management system is an extension of a database system for handling the information about the design of complicated "engineered" artifacts, e.g., large software systems, multi-author documents, and integrated circuits. Although the structures developed here are of use in any

of these domains, we will concentrate on integrated circuits. Artifacts as complicated as these are created by teams of designers simultaneously working on different pieces of a design. In addition, a design may be specified in several different design representations. For example, an integrated circuit design is viewed simultaneously as a geometric layout, a transistor network, or a logic circuit. Besides controlled sharing and multiple design representations, the environment should also support design versions and the exploration of tentative alternative designs. In this paper, we are concerned with how a design data management system implements versions and alternatives.

A design evolves over time. An _in-progress_ version is a design "under construction." Designers modify the in-progress version until it is ready for release. They can create _alternatives_, which are hypothetical variants of an in-progress version. These permit designers to explore experimental design solutions without making changes to the in-progress version. Several alternatives can be merged to create a new in-progress version. When a design is ready for release, it is first made _effective_ for testing and internal distribution. Effective versions cannot be updated without once again becoming in-progress. Effective versions become _released_ when the designed object is sent into the field. Released versions can be _archived_ and later _restored_ when immediate access is necessary.

The question addressed here is how to use existing database techniques and storage structures to support design versions and

alternatives. The remainder of this paper is organized as follows. In the next section, we present a model of how design versions and alternatives are used in the design process. In section 3, we describe the desirable features of a version mechanism, and evaluate previously proposed techniques to support versions and alternatives in section 4. We propose a new storage structure for versions in section 5, where the goal is to support many on-line versions while keeping disk requirements small. In section 6, we describe how new technology, such as write-once optical digital disks, affects version management. We give our conclusions and implementation status in section 7.

2. Design Administration

The design life cycle and operations for manipulating versions and alternatives is shown in figure 2.1. A new "in-progress" version is _created_ from an initial version. Designers simultaneously update in-progress data. Alternative design versions are created by issuing a _create-alternative_ command. Changes to alternative designs do not affect the in-progress version. Mechanisms based on intention locking within a design hierarchy are used to insure exclusive designer access to parts of the design [HASK82, KATZ82]. Therefore, the part of the design updated by a designer, whether the in-progress version or one of its alternatives, will not change underneath him, although "surrounding" design parts might be modified by concurrent activity. Alternatives are created over disjoint sets of data (i.e., parallel subtrees of a design hierarchy), and can be merged to form a

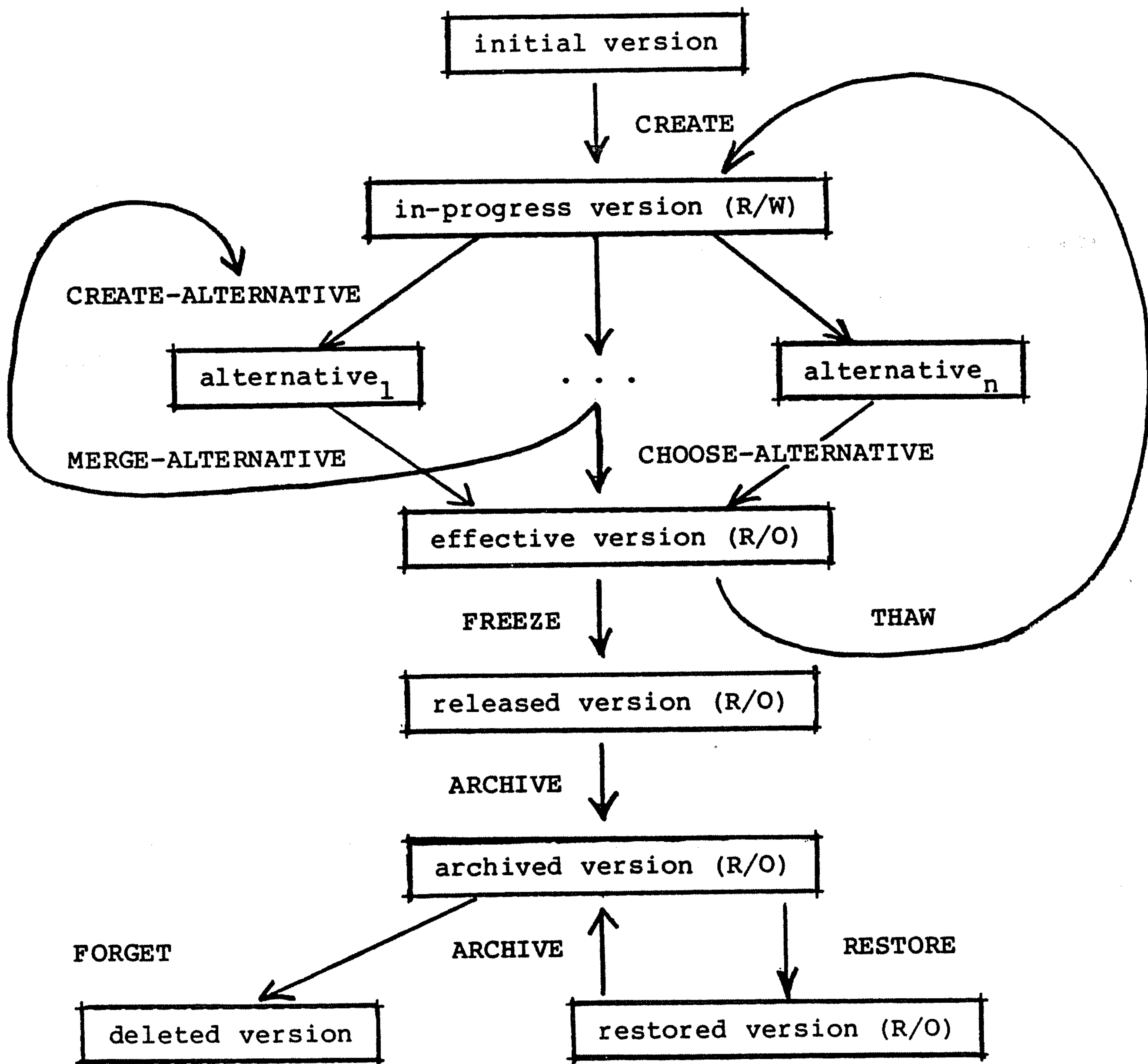


Figure 2.1 -- Design Life Cycle

new alternative with a merge-alternative command. An effective version is formed by choosing an alternative (one of which is the in-progress version). Effective versions are read-only and

represent a pre-release version of a design, suitable for testing before actual release. An effective version can be updated only if it is first thawed back into an in-progress version. Otherwise, it can be frozen into a released version. Releases can be archived and later restored. Ancient versions are removed from the archive by being forgotten. Once forgotten, a version can never be reaccessed. Usually, direct access is required for the last released version, all effective versions created since the last release, and the current in-progress version.

Effective versions are design checkpoints. Since the correction of some design errors can generate many new ones (i.e., the debugging process "diverges"), effective versions provide the capability of backing up a design to a consistent version known to have passed a certain battery of tests.

Designers prefer to try several alternative approaches for solving a design problem. For example, he may want to examine two circuits that perform the same function, yet one consumes less area (but more power) than the other. The design system should make it possible for the designer to carry forward multiple alternative designs in parallel. If several independent alternatives appear promising, they can be merged into a single alternative. An alternative is chosen for combination with the current in-progress version to form a new effective version. Because alternative designs are tentative, changes to an alternative do not affect the current working copy of a design until it is chosen as an effective version.

While a conventional database has a database administrator to control its definition and use, a design database must have a design administrator. This person or group is the only user permitted to issue create, choose-alternative, merge-alternative, restore, and forget operations. Individual designers may only create alternatives or update the in-progress version.

3. Design Goals for Structures to Support Versioned Files

A design version is a consistent representation of a design at a particular time from the viewpoint of the design process. A design version is implemented by the collection of synchronized versioned files that store data about the designed object. Versions are created explicitly by the design administrator, not by individual update operations as in [REED79]. Existing database systems provide no special mechanisms for creating and managing versions of files. However, such support is critical for successful design data management.

We evaluate storage structures for versions on the basis of three criteria:

(1) Does the structure encode versions with a minimum amount of (record) redundancy? This is achieved by only storing new images of records when they change. A record is stored once for each set of versions for which it has gone without update.

(2) Can all versions of a logical record be obtained quickly (record oriented access)? This can be achieved by clustering a current record together with its previous versions. Alternatively, structures can be used to link together a record's current version and its old versions. An example of this type of access is an application that needs the history of how a part of a circuit design has evolved over time.

(3) Can all records within a version be obtained quickly (version oriented access)? All records within a particular version can either be clustered or linked together to insure fast access. For example, an application may need to access the version of a design as it existed in May 1981.

These goals cannot be satisfied simultaneously. For example, it is impossible to physically cluster a file both by version and by logical record without introducing replicated data. For the applications and design environment we have in mind, the overriding consideration is to encode versions with minimum redundancy. This allows us to increase the number of versions kept on-line for a given amount of available disk space. Further, most accesses are directed to the current version, with significantly less access to older versions. We want excellent support for version oriented access, with record oriented access a secondary consideration. Support for version oriented access via clustering cannot be achieved without replication, because some records remain unchanged across versions. Clustering by versions would require that these unchanged records appear with every version

that contains them. Minimum encoding would be lost.

A version scheme must also support alternatives. Alternatives differ from other kinds of versions in that (1) several current alternative versions can exist simultaneously (there is only one current in-progress, effective, or released version at a time), and (2) updates to an alternative are not made to the in-progress version until the two are explicitly combined (alternatives are tentative). Designers working on an alternative do not see the changes made by other designers working on parallel alternatives. Once an alternative is chosen, the others are discarded.

We will propose a method that encodes versions with a minimum number of records. Current versions of all logical records are clustered together for faster access to these frequently accessed records. Older versions are collected together without clustering. An auxiliary index structure supports both record and version oriented access.

4. Previous Approaches for Version Management

Three approaches have been proposed to support some form of versioning: "file level versions", shadow pages (page level versions), and differential files (a form of record level versions). Database snapshots [ADIB80] are similar to versions, but not the same. Snapshots are read-only windows on the database that are refreshed periodically and automatically. Effective and released versions are also read-only, but alternative and in-progress

versions are not. Snapshots are created by time-triggered events, while versions must be created explicitly by the design administrator.

The obvious approach to implement versioned data is based on file level versions.* In this method, a version is represented by a named collection of files, in which each file name is extended with a unique version number. Thus, file DESIGN.1 contains the original version, while files DESIGN.2, DESIGN.3, ... contain subsequent versions of the design. Alternative versions are accommodated by extending the naming scheme. If there are two alternatives for version 2, then the files are named DESIGN.2.1 and DESIGN.2.2. The approach does not use disk space effectively. It is difficult to justify the allocation of precious disk space to alternatives that are not selected to become an in-progress version.

Although versions are not encoded minimally, fast access within a version is supported by clustering each within its own file. Additional structures are needed to interrelate records across versions. We reject this approach because of its poor storage utilization.

The second method is the <u>shadow page</u> mechanism of [LORI77]. It reduces the storage overhead of keeping multiple versions of design data on-line by only storing pages that change across

* OS/360 supported the notion of generation data sets, which is a similar concept.

versions. The first update to a record on a page creates a new version of the page. The design database is a collection of page maps over data pages. A new version is created from the current version by first creating a copy of its page maps. Updates are not performed in place. Instead, a new page slot is allocated, the changed page (i.e., the "shadow") is written to this slot, and the page map of the new version points to it. The storage requirements to keep versions on-line is the sum of the sizes of the old version, the pages that have been modified, and the page maps.

The shadow page method has been proposed as a recovery mechanism, not as a technique for versions. Consequently, certain features are lacking that are crucial for their support. Accessing a version of a record or all records within a version is efficiently implemented by the page maps, but versions are not clustered (except for the original). Since different versions of the same record are assigned the same virtual address, it is not obvious how to reuse virtual addresses once a logical record, with old versions still on-line, has been deleted. Page versions do not lead to a minimum encoding for record versions. Further, for large databases, the page maps can grow large enough to require their own mechanisms for paging and swapping.

The third method for representing design versions is by differential files [SEVE76]. A new version is represented by a base file (old version) and a change file (the differential file). Fast version access to the base file/old version is supported by

clustering, but the new version remains unclustered. Without auxiliary structures, access to record versions is slow. To find a record, we must first check if it is in the differential file. If it isn't, we must search the base file. In the worse case, twice as many reads are needed to find the records in the new version that have not changed (mechanisms, such as the Bloom filter, have been proposed to reduce the probability of encountering the worst case). An advantage is that the differential file mechanism achieves minimum record encoding of versions.

Differential files provide a natural implementation of alternatives [STON80, STON81]. For example, an alternative A is represented by the view A = (B U I) - D, where B is the in-progress version, I is an insertion differential file, and D is a deletion differential file. Since changes to A are placed in I or D, B is never modified by an update to A. The scheme cannot support inserted records that are identical to a previously deleted record. This is only circumvented by introducing a unique identifier for each logical record.

A major deficiency of differential files is that the old version is clustered, while the new version is not. We propose a "negative" differential file to provide better support for versions. I and D differential files are created when a new in-progress version is created from a released or initial version. Changes are made to the base file in-place and recorded in the appropriate differential files (note the resemblance to an undo recovery log). The old version is defined by the new version and

the differential files as OLD = (NEW U D) - I. The advantage of this representation is that the new version is clustered, while old versions can be accessed (slowly) without the need for excessive disk space. This is particularly effective when (1) versions do not vary much, and (2) the old version is referenced infrequently (but not so infrequently that it might as well be archived and restored as needed).

The Source Code Control System (SCCS) of UNIX* uses a differential file approach for versions of text files [ROCH75]. Each version or alternative is represented as a difference ("delta") file over the previous version. Versions are encoded minimally. Before a version is read or updated, it is first extracted into a work file, thus clustering the desired version. The extraction process is time consuming, since the complete chain of delta files from the initial version to the extracted version must be accessed. Since SCCS is concerned with text files and not databases, there is no support for record oriented access. No special capabilities are provided for version archive or restore. Alternatives are supported by parallel delta files.

5. A New Storage Structure for Version Management

5.1. Requirements for Version Management

We review the desirable properties of a version support mechanism, and indicate how the structures of this section

---

* Trademark of Western Electric Corporation.

provide them:

(1) Minimum redundancy: only changed records are stored for each new version. Versioned files are arranged as negative differential files.

(2) Record oriented access: all versions of a logical record can be accessed. Each logical record is uniquely identified. An auxiliary index structure supports access to all on-line records with the same logical record id.

(3) Version oriented access: all records within a version can be accessed. The structure used in (2) is also used here. The index indicates which of each logical record's several versions was the one existing when each file version was created. The records are extracted into a working file to obtain clustering.

5.2. Version Storage Structure

A logical record is associated with a collection of physical records that represent it at different times in the life of a design file. The relationship between a record and its versions should be independent of physical addresses, since we would like to reuse the physical address space after a logical record has been deleted. We associate a system generated surrogate with a logical record, and embed it in each of its record versions. Once a surrogate has been assigned to a logical record, it may never be reassigned. Record oriented access is supported by a surro-

gate index over the record versions. The index is implemented with a B+-tree structure [COME79], commonly supported by database system access methods.

The size of the surrogate is application dependent. A four byte surrogate is sufficient to identify approximately 4.3 billion different logical records per file. For a one million record file, this permits over four thousand completely rewritten versions of the file! Longer surrogates are needed if: (1) a design file contains a large number of records, (2) versions are created frequently, or (3) a large percentage of the file changes from version to version. For the types of databases under investigation, these are rare situations.

Our storage structure for version support is organized as follows. Each physical record is identified by a unique record ID (RID), which serves as its physical address within the database. The B+-tree index is keyed on logical record surrogates. The leaf pages contain entries that associate a surrogate with the RIDs of the logical record's versions. Since a RID is unique across the collection of files which make up the database, the index can span several files.

A versioned file is supported by three internal files: the _history index_ as described above, the current version file (henceforth called the _current file_), and the old version file (henceforth called the _history file_). The entry associated with a key within leaf pages of the history index is called a _version_

history. A version history contains a RID for its logical record's version in each version of the file. We will describe some techniques for compressing the version history in the next section. The current versions of all records are clustered together in the current file, while old on-line versions of records are contained in the history file. Although the current and history files could be combined, we obtain better performance by keeping the current version clustered in its own file (see figure 5.1).

When a versioned file is first created, the history index, current file, and history file are empty. As new records are added to a file, their surrogates are entered into the history index and the record itself is inserted in the current file. The first update to a record after a new version is created is not performed in place. The old record is first copied from the current file to the history file, then the new record overwrites it in the current file. The version history is updated to point to the new current record and the newly moved previous record. Subsequent updates occur in the current file. Note the similarity to the shadow page method. However we use index structures that have been designed for very large files and residence on secondary storage. This is not the case for page maps.

Old versions can be accessed directly within the history file, but better performance is obtained by first extracting the desired version from the current and history files into a working file. The current version is always kept in the current file.

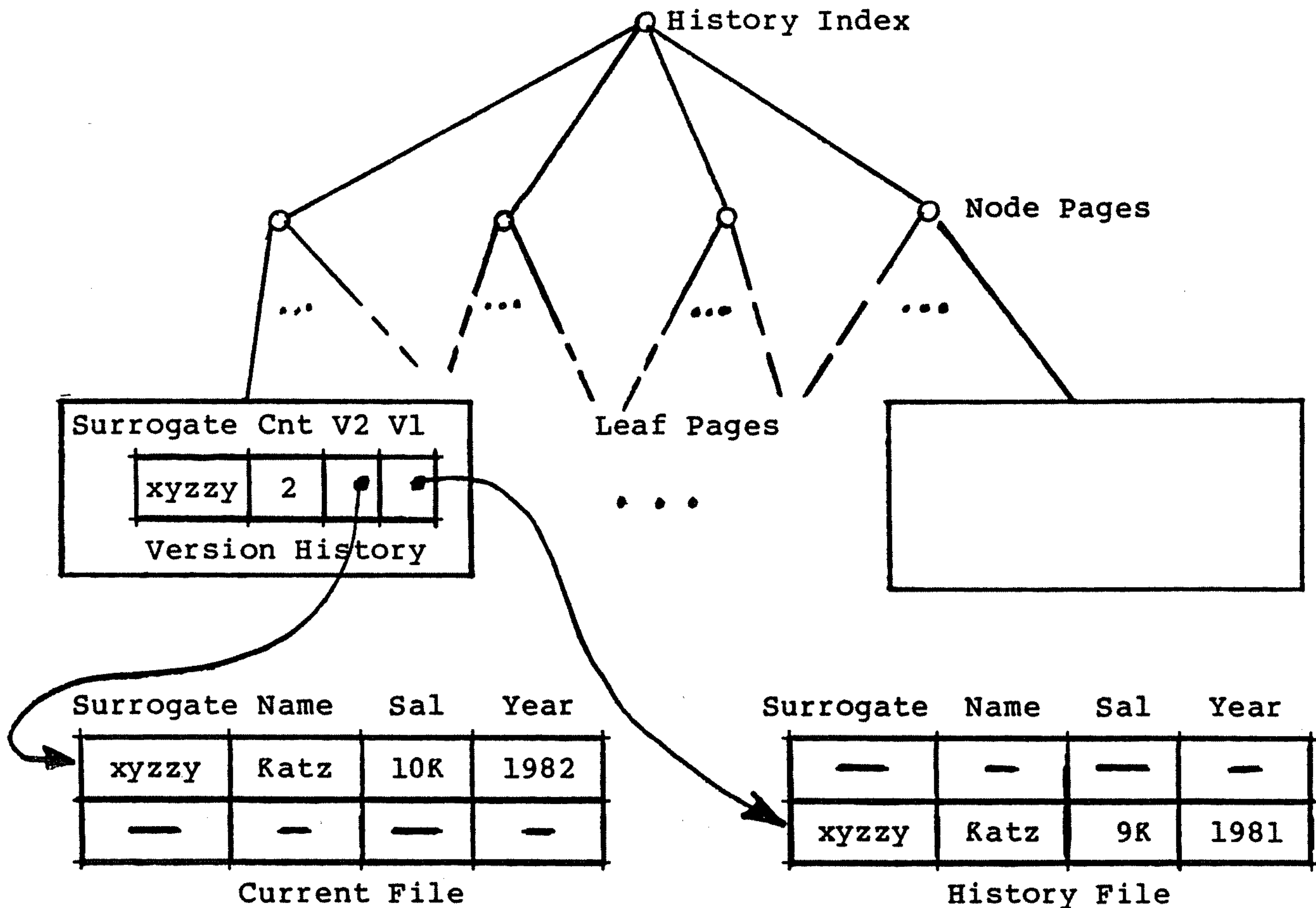


Figure 5.1 - Version Control Storage Structure

However, to reduce the overhead of updating the current file and history index, the current version is also copied into the working file before an editing session. Updates are batched and merged into the current file when the session is over. This is the approach taken by SCCS, except that here the current rather than the original version is directly accessible. To extract an old version, the version histories are scanned to obtain the RIDs

of the appropriate record versions. The records are accessed and copied into the working file. Indices over the working file can be constructed during this copy operation.

A performance problem arises because records are accessed in surrogate order rather than in physical address (RID) order, since we scan the leaves of the history index to do the extraction. A solution is to first copy the RIDs into a temporary file, sort them, and then access the records of the current and history files for copying. Since RIDs are small (typically 8 bytes), even for a large database the RID list need not be prohibitively large. Ideally, surrogates should be allocated in RID order, but even if this were possible, the ordering relationship between surrogate and RID would be lost once a record moves from the current file to the history file.

The purpose of the working file is to batch updates to the current version, and to cluster the records of a particular old on-line version for better access. A surrogate is allocated for a record when it is first inserted into the working file. Whenever it is modified, inserted, or deleted, an entry is simultaneously placed in an associated change file (redo log). This is a combination insertion/deletion differential file indexed by surrogate. Only the most recent update for each record is kept in the change file.

When update activity has ceased, the current file is updated by merging the change file into the history index. If a new ver-

sion is being created by the merge, then each version history is expanded by an entry for a new current record version, and the old current becomes the new previous version. The previous version consists of the records in the current file at the point at which the file was extracted into the working file. The details are given in figure 5.2. CurrentVersion(Surrogate), PreviousVersion(Surrogate), and CurrentFile(Surrogate) refer to the current version entry and previous version entry in the version file, and the record within the current file respectively. Otherwise the entries in the change file are used to update the records in the current file directly. The algorithms for archive and restore of old versions are given in figure 5.3. If several

---

Algorithm MergeChangeFile

```
FOR EACH Surrogate S in ChangeFile DO
   IF ChangeFile entry is new THEN
      Insert change file record into CurrentFile
      Insert S into HistoryIndex
      CurrentVersion(S) <- record inserted in CurrentFile
      (all)PreviousVersion(S) <- NULL
   ELSE IF ChangeFile entry is delete THEN
      Copy CurrentFile(S) to HistoryFile
      Delete CurrentFile(S)
      CurrentVersion(S) <- NULL
      PreviousVersion(S) <- record moved to HistoryFile
   ELSE
      Copy CurrentFile(S) to HistoryFile
      CurrentFile(S) <- ChangeFile(S)
      CurrentVersion(S) <- new CurrentFile record
      PreviousVersion(S) <- record moved to HistoryFile
```

Figure 5.2 -- Merge Change File into Versioned File

```
Algorithm ArchiveVersion

FOR EACH Surrogate S IN HistoryIndex DO
   Extract RID of desired version from version history
   IF not NULL THEN
      Copy record into archive
      Remove entry from version history
      IF no other entry contains RID THEN
         Remove from current/history file

Algorithm RestoreVersion

FOR EACH record IN archived version DO
   Insert archived record into history file
   IF surrogate is not in HistoryIndex THEN
      insert surrogate into index
   Entry for restored version within HistoryIndex
     points to inserted record
```

Figure 5.3 -- Version Archive/Restore Algorithms

versions are archived and then restored, the total on-line storage requirements may increase, since records that span versions are not identified and combined during restore.

Several types of read access to versioned data are supported by the above structures. The version histories provide the capability to find any on-line version of a record given its surrogate. The cost is only that to traverse the index, plus an extra access to the current or history file to get the desired record. Since the surrogate is embedded in every record version, we can access other versions of the same record starting with the surrogate within a particular version. Index structures can be built over specific versions (including the current version) or across

a subrange of versions (including all on-line versions). Queries can be expressed over the current version (the default), a specific on-line version, a subset of on-line versions, or all on-line versions. Placing the current versions in the same file allows the records of the current version to be clustered for good sequential access. The current file can be structured in any way desired for fast access to the current version. The working file can be structured similarly for extracted versions.

The working file and change file structures also support design alternatives. An in-progress version is extracted into the working file, all updates are performed there in-place and posted to the change file. Since an in-progress version is updated independently of the alternatives defined over it, a designer working with an alternative periodically refreshs his copy of the in-progress version in the working file. His change file must be merged back into the working file to recreate the changes made in the alternative. This is expedited if an index on surrogate is built over the working file. An alternative is incorporated into the in-progress version by merging the alternative's change file into the current file as described above.

In the next subsection, we present the operations for manipulating the structures in more detail, suggesting some optimizations that make the approach more suitable for actual implementation.

5.3. Implementation Details for Version Manipulation

There are a few problems with the storage structure described above. First, if version histories are not removed from the history index after all versions of the logical record have been archived or deleted, then the structure can grow intolerably large. Some mechanism is needed to reclaim space within the index. Second, the creation of a new version as described above causes every version history to be accessed and expanded by a new entry, which is a large overhead. Version histories should be expanded only when they are needed. Finally, we need compression techniques to keep the version histories as compact as possible.

These problems are solved by associating additional information with the versioned file, and by being more clever in how we encode version histories within the history index. Associated with each file is a surrogate count (SCnt), which is the largest surrogate allocated so far, a version number (V#), which is the number of the largest version created to date, a version count (VCnt), which is the number of versions currently on-line, and a version map (VMap), which is a bit map with a "1" for those versions still on-line, and a "0" for archived versions (this is an implementation detail -- a table could be used for the version map, but would take up more space).

Version histories are compressed in two ways: (1) only on-line versions are recorded in the history -- there is no need to maintain the RIDs of records that have been archived, and (2) if

a record has not changed across several recent versions, then the history will contain the same RID within the current file for each of these -- store this RID once and make each version history variable length, with the understanding that the most recent version in a record's history is also the same as its most recent version in the file's current version (and all versions between). The details follow.

The version history is organized as in figure 5.4. The Cnt is the number of record versions indicated by a particular version history (Cnt $\leq$ VCnt $\leq$ V#). V_i, ..., V_1 are RIDs of records that represent the versions of the record identified by the Surrogate. A RID of NULL indicates that the logical record has no image for that version.

A version history can be deleted from the history index whenever all its on-line versions are NULL. The on-line versions are those versions i such that VMap[i] = "1". The condition is checked whenever a record is deleted or a version is archived. Even though the history is dropped from the index, the surrogate

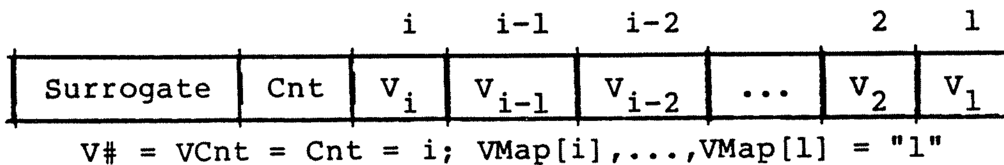


Figure 5.4 -- Version History Format

cannot be reallocated, since a version of the record may be restored later.

Cnt is the count of entries in the version history for a particular record, while VCnt is the total number of versions currently on-line. Logically, every history should have VCnt entries, but if a record has not been updated for several versions, Cnt will be less than VCnt. The last VCnt - Cnt versions of this record are identical to the most recent version stored in the history, since the record has not changed in the meantime. The version history is expanded whenever the record it describes is updated.

We use VMap to eliminate holes in version histories that occur when versions are archived. At any point in time, VMap is V# bits long and V# exceeds VCnt. VCnt bits within VMap are "1" to indicate which are the on-line versions. In the naive implementation of version histories, the RID of version k is found in the kth entry. Suppose that some versions older than k have been archived, and that their entries have been removed from the history. If there are j on-line versions preceeding k, the RID of version k is found in the $j+1^{th}$ entry ($k \geq j+1$). For example, consider the version history of figure 5.5, with four versions, of which three are on-line. The RID for version V_4 is found as follows. The sum of VMap[1] through VMap[3] is 2. Therefore, RID of V_4 = $(2+1)^{th}$ entry in the version history. In general,

		3	2	1
Surrogate	Cnt	V_4	V_2	V_1

V# = 4; VCnt = Cnt = 3;
VMap[4],VMap[2],VMap[1] = "1"
VMap[3] = "0"

Figure 5.5 -- Version History with V_3 Archived

$$\text{RID of } V_k \leftarrow \left(\sum_{j=1}^{k-1} \text{VMap}[j]\right)+1^{th} \text{ entry of the version history}$$

This version encoding keeps version histories small since only on-line versions are recorded. An additional method to reduce the size of histories is to compress the entries in individual histories. The technique is effective since records are typically left unchanged across versions. Rather than dedicate an entry for each version, the history can be encoded as a RID followed by a repetition factor. The decoding of the compressed data is straightforward.

5.4. Evaluation

We call our version method record level versions, to distinguish it from the other approachs of section 3. First we compare record level versions and file level versions on the basis of on-line storage costs. We develop a simple cost model to determine when our approach uses less storage than file versions.

Since file level versions are unlikely to conserve on storage costs, it comes as no surprise that for a large number of situations, our approach is superior. A similar analysis is performed for record level versions and page versions. For all but a few extreme cases, record versions is still superior.

5.4.1. Record Level Versions and File Level Versions

Record level versions keep to a minimum the on-line disk space for a version's data, at the cost of some associated overhead for the history index. Fortunately, the size of the index grows very slowly in comparison to the number of records. We develop a simple model of the on-line storage requirements of a versioned file to compare file level versions and our method. Let $\underline{r}$ be the number of records in the file, $\underline{s}$ be the size of each record, $\underline{d}$ be the fraction of changed records from one version to the next, and $\underline{n}$ be the number of non-current versions on-line (e.g., a file with only the current version on-line has $n = 0$). We assume that the size of the file is stable and does not grow, i.e., that records are modified but not inserted or deleted.

The storage cost to keep n versions on-line using file level versions is:

$$n * r * s,$$

while the cost for record level versions is:

$$k * (12 * r + 8 * d * r * n) + d * r * s * n.$$

The latter is derived by assuming that the index entry for each record is 12 bytes long (4 byte surrogate + 8 byte RID), except for those records that have been modified. Their entries are an additional 8 bytes long, i.e., they contain the RID of the record's previous version. There are d*r such records. We assume that because of the techniques of section 5.3, only the modified records have a completely expanded entry in the history index. The term d*r*s is the marginal increase per version in the size of the history file. The cost of the internal nodes of the B-tree is captured by the factor k. Assuming 400 <key, pointer> pairs per internal B-tree node,

$$k = \sum_{i=0}^{\infty} 1/(400^i) \approx 1.00251$$

Thus the storage for internal nodes is approximately .251% of the storage for the history index leaf pages.

The values for which both approaches yield the same cost is determined by setting the two formulae equal and solving for $\bar{s}$, the <u>breakeven</u> <u>record</u> <u>size</u>:

$$n * r * \bar{s} = k * (12 * r + 8 * d * r * n) + d * r * \bar{s} * n$$

$$(5.1) \qquad \bar{s} = k * (12 + 8 * d * n) / (n * (1 - d))$$

S values computed for selected values of n and d are shown in figure 5.6. These are plotted in figure 5.7. Equation (5.1) reveals that the breakeven record size is independent of the

$\underline{d}$	$\bar{s}$ ($\underline{n}$=$\underline{1}$)	$\bar{s}$ ($\underline{n}$=$\underline{2}$)	$\bar{s}$ ($\underline{n}$=$\underline{3}$)
0.0	12	6	4
0.1	15	8	5
0.2	17	10	7
0.3	21	12	9
0.4	25	15	12
0.5	32	20	16
0.6	42	27	22
0.7	59	39	32
0.8	92	62	52
0.9	192	132	112
1.0	∞	∞	∞

Figure 5.6 -- Breakeven Record Sizes

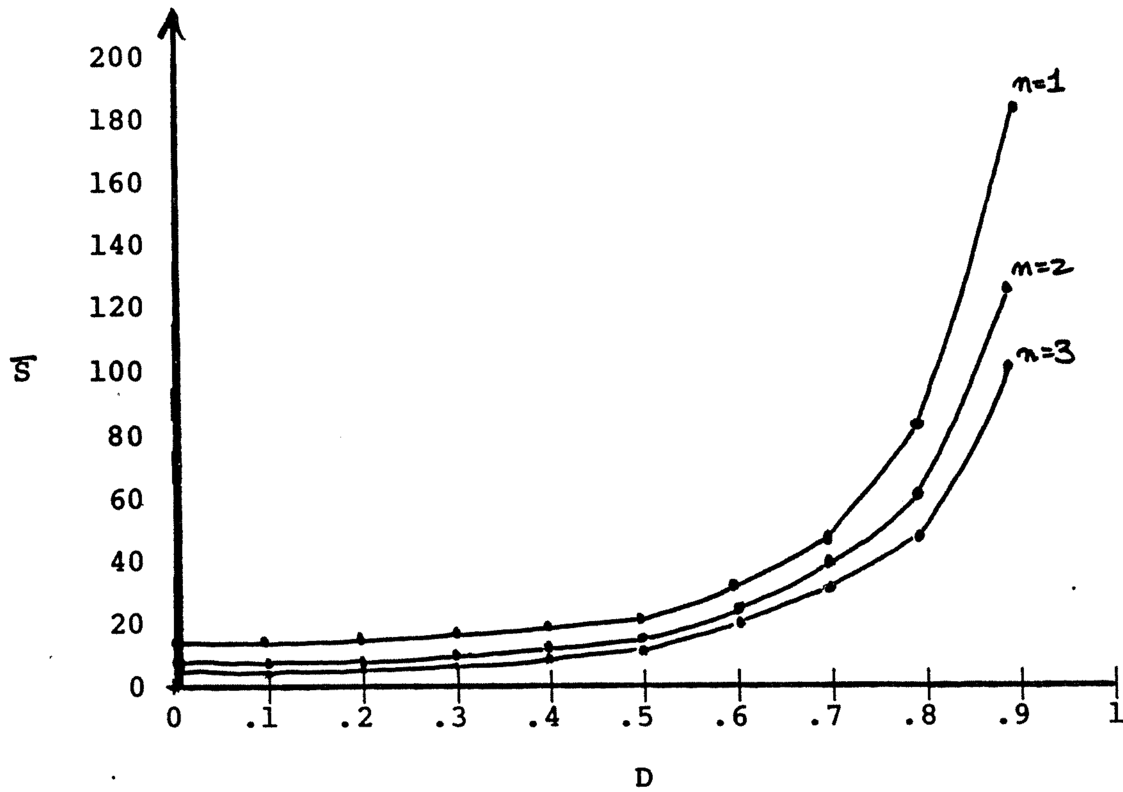


Figure 5.7 -- Breakeven Record Size Curves

number of records in the file. This is because the formulation is based on marginal increases in space requirements.

The graphs of figure 5.7 are explained as follows. Above each curve is the region in which record level versions yield a lower on-line storage requirement than file level versions. Thus, for the case where n=1 and d=0, record level versions require less space when the record size is <u>greater than</u> 12 bytes. As n increases, the breakeven sizes decrease, and as d increases, the breakeven sizes increase.

The use of the history file saves on-line space, but the space savings is not always enough to recoup the overhead of the history index. The storage cost of the history index is independent of the record size. The longer the individual records, the more space is saved by not replicating the unchanged records, and the easier it is to amortize the cost of the index. Thus, our approach is not well-suited for small records. As n increases, the history file saves even more space, at a small incremental increase in index size. Thus, the breakeven size decreases. Similarly, as d increases, the amount of space saved by the history file decreases. It becomes more difficult to amortize the cost of the index. For example, the whole file has changed when d = 1. Hence, the history file is as large as a complete file level version, with the extra space overhead of the history index. The result is that no record size, no matter how large, will make record versions attractive. Note that file level versions will need some indexing mechanism to access versions of the same record across version files. Since we have not included this in our analysis, record level versions may be even better than as indicated above.

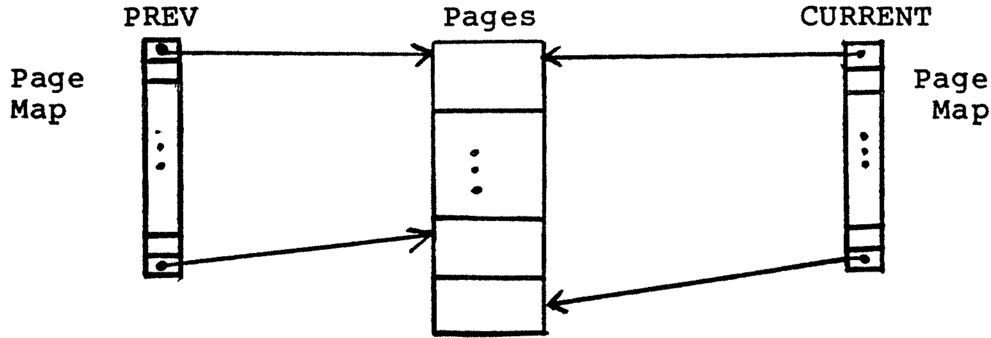
To summarize the trends, record level versions is advantageous when (1) records are not small, (2) the amount of change that a file undergoes between versions is relatively small (10-30%), and (3) several versions are typically kept on-line. We believe that these conditions exist in the design environment.

5.4.2. Record Level Versions and Shadow Pages

There is a resemblance between record versions and shadow pages as described by Lorie [LORI77]. Shadows support two versions of data, i.e., previous and current pages. Our technique supports arbitrarily many versions at the record level. The shadow page scheme maps an address into the appropriate physical page by page maps, while we map surrogates (essentially logical addresses) into the appropriate record via the history index (see figure 5.8). Shadows could be extended to maintain a page map per version to support more than two on-line versions. However, since versions of the same record have the same virtual memory address, we still need a mechanism to manage version histories of deleted records, so we can reuse the freed address space. In the following analysis, we only examine the case where two versions (current and previous) are kept on-line.

Record versions have more control overhead, because of the history index and associated version histories. However, the complete size of the data portion of a versioned file will be smaller because of the smaller version granularity. With page versions, records that have not been updated are stored redundantly across versions. Access costs are higher in ours because of the index traversal, while the cost for shadows is only the single level of indirection through the page maps. Further, because the allocation of surrogates is not clustered, frequent faults among index pages may arise. However, page versions also forfeit the clustering among shadow and original pages. The

Shadow Versions (Pages)



Record Versions

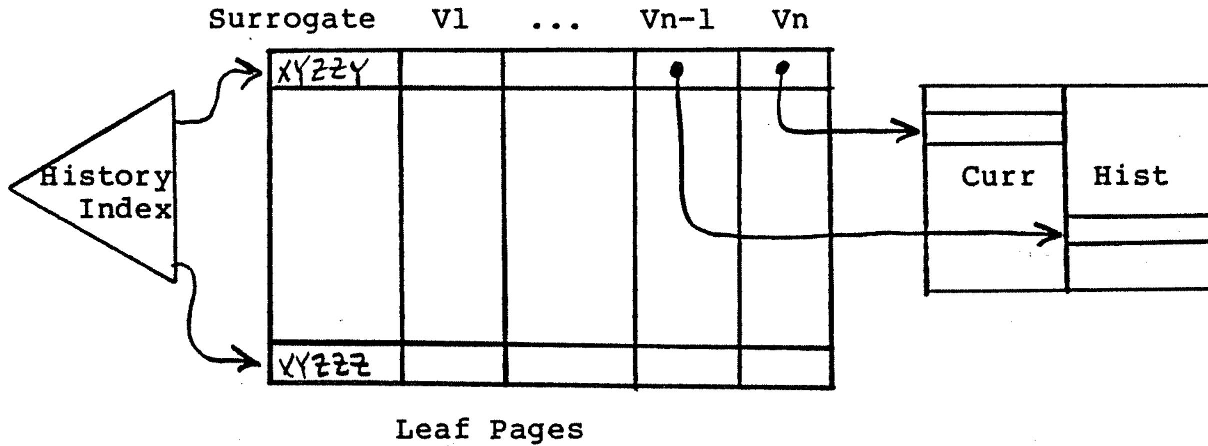


Figure 5.8 -- Comparison of Shadow Versions and Record Versions

working and change files are proposed, in part, to relieve the performance overhead of accessing versioned data through the history index.

The total storage cost for the shadow page method is the cost of the page map plus the cost of the new pages added to the

file. The storage cost, as a function of the number of records updated, is:

(5.2) $8 * r * s / p + p * F(x)$

where r and s are as above, p is the page size, x ($x \le r$) is the number of records updated, and F(x) is the number of pages containing the x updated records. We use Bernstein's approximation of the Yao function [BERN81] to compute F:

$$p * F(x) = \begin{cases} p * x, & x \le r*s/2p \\ 1/3 * (p * x + r * s), & r*s/2p \le x \le 2r*s/p \\ r * s, & 2r*s/p \le x \end{cases}$$

When the number of records updated is less than half the number of pages, each record access results in another page fault. In the range between half and twice the number of pages, less than one page is accessed for each record as we begin to hit previously accessed pages. When more records than twice the number of pages are updated, then every page will have been accessed.

We can reexpress the storage cost for record versions as a function of the number of updated records as well:

(5.3) storage cost $= 12 * k * r + (8 * k + s) * x$

Superimposing the two curves results in a graph of the form of figure 5.9.

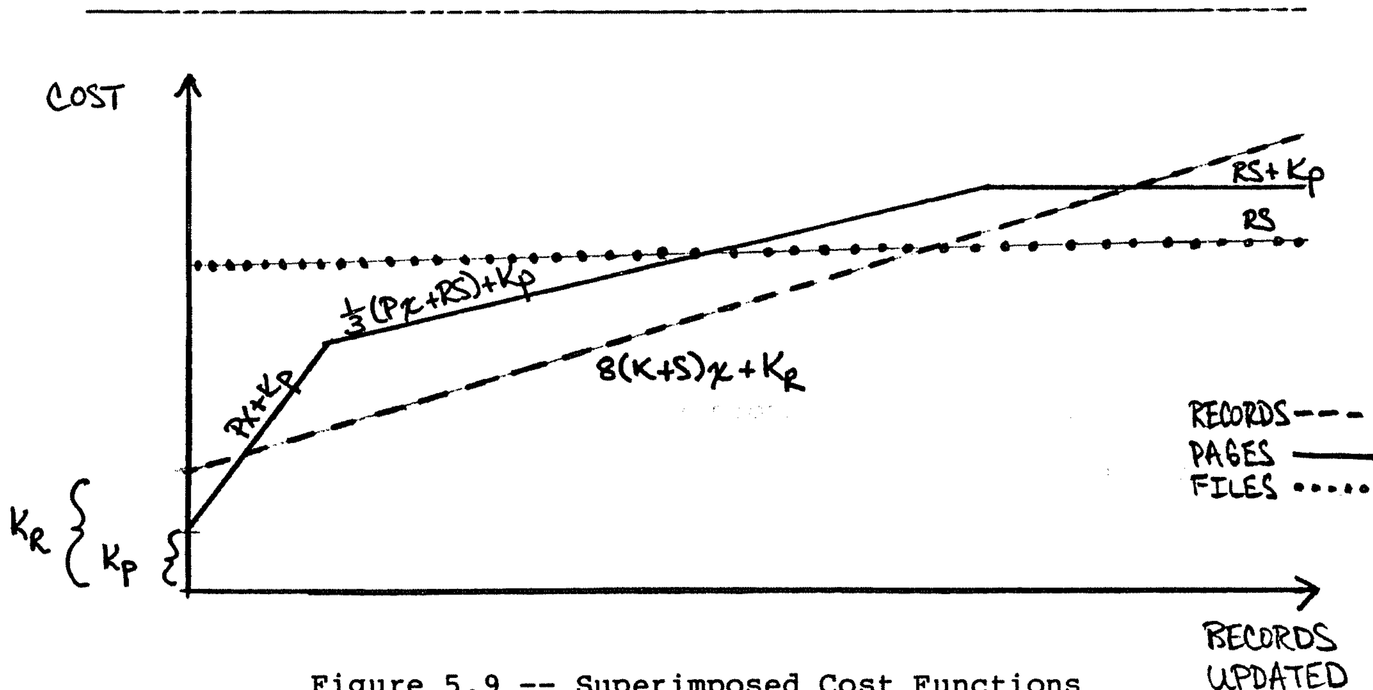


Figure 5.9 -- Superimposed Cost Functions

Observe that the cost functions intersect at two points. Initially the shadow page method has a much smaller overhead, since the space for the page maps is small compared with the history index. As records are updated, the cost of shadow pages grows more rapidly as whole pages are added to the file instead of individual records. Our experiments indicate that the first intersection point (low point) occurs very close to the origin.

Eventually every page of the file has a new version, and subsequent updates incur no additional storage cost. However, these updates continue to add to the cost of record versions. Eventually the curves cross again (high point) as the initial overhead of the history index once again comes into play.

For a fixed size database (4 MBytes) and page size (4000 Bytes), we have computed the following points of intersection for different choices of r and s in figure 5.10. As the number of records increase, the low point increases. When expressed as a fraction of the total number of records (d), the low point is very small, i.e., much less than .01.

To summarize, the record version approach is superior to shadow pages in terms of storage utilization unless either very few records are updated (less than 1%) or very many records are updated. Although we believe that few records are updated between versions, we believe that enough are to justify the choice of record level versions.

6. Effects of New Technology on Version Management

Optical digital disks are a new technology that promise to dramatically increase the amount of on-line storage available to a computer system. The write-once nature of the medium makes it

r	s	low point (d)		high point
2000	2000	5	(.0025)	1959
4000	1000	11	(.0028)	3937
6000	500	23	(.0038)	5716
10000	400	30	(.0030)	9548
12000	333	36	(.0030)	11354
14000	285	42	(.0030)	13131

Figure 5.10 -- Sample Intersection Values

ideal for archival purposes. Current optical disk densities are about 2400 million bytes per disk surface (40,000 tracks by 486,000 bits per track) [MAIE82]. An optical disk pack of ten surfaces has the same capacity as eighty conventional 300MByte disks.

We envision optical disk technology as a way of providing an on-line window into an archive. Versions in the on-line window can be restored quickly, whereas access to versions in the off-line portion of the archive is much slower. We describe the implementation of the on-line window when we have three optical disk drives. Drives A and B hold the on-line archive, while C is a free drive used to load off-line disks whenever an off-line version is to be restored. Drive A is the "top" of the on-line archive, and B is the "bottom". When A becomes full, further writes are written to B. When it becomes full, a fresh disk is loaded on C, the disk on A is moved off-line, and B and C become the top and bottom of the archive respectively. A is available as the free drive. The scheme is described in [SVOB81].

Whenever a new version is created, the previous current version is written into the archive, whether it is to be archived or kept on-line. Only the current version is kept on conventional disk. Eventually the on-line window will become full, and a full disk's worth of data will be moved off-line. The portion of the disk that was meant to be archived will then join the archive. However, something must be done to keep the rest of the data on-line.

Associated with the optical disk archive is a directory, on conventional disk, that identifies file versions with their archive addresses (the Restore Directory). The information is needed to find a version within the archive while processing a request to restore it. Addresses are of the form <optical disk volume ID, offset>. Another on-line directory (the Window Directory) identifies which versions are in the on-line window, and whether they are "archived" or meant to stay on-line. This directory is updated whenever: (1) a version is written to optical disk (add the version to the directory), (2) a version in the on-line window changes from "on-line" to "archive", and (3) the on-line window becomes full, forcing an optical disk to be moved off-line. In the latter case, archived versions are dropped from the on-line window's directory, and all other versions are copied to the top of the archive before the disk is taken off-line.

Leaving copies of a version sprinkled around the archive is not as bad an idea as it may seem. The density of optical disk storage is so immense that the percentage of useful bits (non-redundant data) per area of media remains high. While the copy operation is time consuming, it occurs only when a disk becomes full. We believe this to be an infrequent event: the time between subsequent versions in the design environment is measured in days or weeks rather than seconds. A version can stay in the on-line window for a long time before it must be copied. During that time, designers may lose interest in a version, changing its state from on-line to archive. When the window fills up, then the

disk must be moved to archive. Many of the originally "on-line" versions will now be "archived", and will not need to be copied.

Because of the relatively long seek time of optical disks, it is unreasonable to expect to use them for random access. It is not advisable to intermix many random reads with bulk writes to the end of the archive on optical disks. However the technique of extracting a version into a working file is well-suited for optical disks. An on-line version is bulk copied into a working file on conventional disk before it can be accessed (note: optical disks provide a high data transfer rate that is suited for large volume copying). Random access on a conventional disk does not suffer from the long seek time penalties of optical disks.

To summarize, optical disk technology allows us to dispense with our requirement of minimal version encoding. With large amounts of storage available, we choose file level versions instead. The in-progress version remains on conventional disk. Old versions are extracted into a working file on conventional disk as before.

7. Conclusions and Status

We have described a storage structure for version management that combines aspects of differential files and shadows. Versions are encoded differentially while keeping the access overhead reasonable. A versioned file is represented by a current file, holding current record images, a history file, holding their old shadows, and an index, relating the records within the current and

history files. A version is extracted from this structure, updates are posted in a change file, and the changes are merged with the original file when the version is replaced. The scheme provides natural support for design alternatives.

We are in the process of implementing a storage system called WiSS (Wisconsin Storage System), with Professor David DeWitt and a group of students (Nina Anania, Jim Bowan, Heng-Tai Chou, Wei-Laung Hu, Rick Simkin, Ann Varda, Dave Ward, Ed Wimmers). WiSS will support versioned files, as well as more conventional files and access structures. The version support structures described here are constructed from B-tree indices and sequential files implemented by WiSS. We plan to build a design data management system on top of WiSS, to provide a variety of database services tailored for the engineering design environment [KATZ82].

8. Acknowledgements

Randy Katz was supported by the Wisconsin Alumni Research Fund. Tobin Lehman was partially supported by a grant from the IBM Corporation. We acknowledge the help of our colleagues, Haran Boral and David DeWitt, who willingly read earlier drafts of this paper and offered many constructive comments.

9. References

[ADIB80] Adiba, M. E., B. G. Lindsay, "Database Snapshots," Proc. 6th Intl. Conference on Very Large Databases, Montreal, Canada, (Oct. 1980).

[BERN81] Bernstein, P. A., et. al., "Query Processing in a System for Distributed Databases," ACM Trans. on Database Sys., V 6

N 4, (Dec. 1981).

[BORA82] Boral, H., D. DeWitt, D. Bates, "A Framework for Research in Database Management for Statistical Analysis," ACM SIGMOD Conference, Orlando, FL, (June 1982).

[COME79] Comer, D., "The Ubiquitous B-Tree," ACM Computing Surveys, V 11, N 2, (June 1979).

[ELLI80] Ellis, C. A., G. Nutt, "Office Information Systems and Computer Science," ACM Computing Surveys, V 12, N 1, (Mar. 1980).

[HASK82] Haskin, R. L., R. A. Lorie, "On Extending the Functions of a Relational Database System," ACM SIGMOD Conference, Orlando, FL, (June 1982).

[KATZ82] Katz, R. H., "A Database Approach for Managing VLSI Design Data," 19th ACM/IEEE Design Automation Conference, Las Vegas, NV, (June 1982).

[LORI77] Lorie, R. A., "Physical Integrity in a Large Segmented Database," ACM Trans. on Database Systems, V 2, N 1, (Mar. 1977).

[MAIE82] Maier, D., "Using Write-Once Memory for Database Storage," ACM Symp. on Princ. of Database Systems, Los Angeles, CA, (Mar. 1982).

[REED79] Reed, D., "Implementing Atomic Actions on Decentralized Data," Proc. 7th ACM SIGOPS Symp. on Operating Systems Principles, 1979.

[ROCH75] Rochkind, M. J., "The Source Code Control System," IEEE Trans. on Software Engineering, V SE-1, N 4, (Dec. 1975).

[SEVE76] Severence, D. G., G. M. Lohman, "Differential Files: Their Application to the Maintenance of Large Databases," ACM Trans. on Database Systems, V 1, N 3, (Sep. 1976).

[STON80] Stonebraker, M. R., K. Keller, "Embedding Expert Knowledge and Hypothetical Data Bases into a Data Base System," Proc. ACM SIGMOD Conference, Santa Monica, California, (May 1980).

[STON81] Stonebraker, M. R., "Hypothetical Databases as Views," Proc. ACM SIGMOD Conference, Ann Arbor, Michigan, (May 1981).

[SVOB81] Svobodova, L., "A Reliable Object-Oriented Data Repository for a Distributed Computer System," Proc. 8th Symp. on Op. Sys. Principles, Pacific Grove, CA, (Dec. 1981).