The Starburst Long Field Manager

Tobin J. Lehman
Bruce G. Lindsay

IBM Almaden Research Center

Abstract

Starburst is an experimental database management system prototype whose objectives include extensibility, support for knowledge databases, use of memory-resident database techniques, and support for large objects. We describe the structure of the Starburst long field manager, which was designed to manage large database objects such as voice, image, sound and video. The long field manager uses the buddy system for managing disk space, which allows it to allocate a range of small to very large disk extents (buddy segments) quickly and efficiently. When possible, a small number of large buddy segments are used to store long fields, which allows the long field manager to perform fewer disk seeks when transferring large objects to and from disk. The long field manager uses shadow-based recovery for long field data and write-ahead-log recovery for long field descriptor and allocation data. Internal space management synchronization is enforced by a combination of long-term and instantaneous locks.
Keywords: Starburst, relational database, long fields, storage methods, buddy system.

1 Introduction

The main goal of the Starburst experimental database management system (DBMS) is to provide a flexible platform on which research personnel can perform DBMS experimentation well into the 1990s. So far, this experimentation has been directed toward improving the structure and performance of the internals of the DBMS, *e.g.*, new recovery and logging mechanisms [Rothermel 89], extensible data management [Lindsay 87], new optimizer methods [Lee 88] [Lohman 88], new query processing methods [Haas 89], new methods for handling data types [Wilms 88], and new signature access methods [Chang 89].

As user-interface technology advances, user interfaces will make more use of multimedia presentations by manipulating voice, sound, image, video and animation. From the database perspective, each of these multimedia objects will require large amounts of storage. Furthermore, presenting images, playing sound, producing video signals, and displaying animation in real time provide significant challenges for a relational database.

The Starburst long field manager was designed with these applications in mind. The long field manager design goals include high data bandwidth with low CPU overhead and efficient storage management techniques. We have combined existing storage management techniques from database systems and file systems in a novel way to achieve these goals. The rest of the paper describes the design of the Starburst long field manager and is organized as follows.

Section 2 discusses the goals and the motivation behind the design of the Starburst long field manager, along with previous work in the area of data storage. Section 3 presents the storage design of the long field manager, including the space allocation data structures and algorithms. Section 4 discusses concurrency control and recovery for long fields. Section 5 summarizes the work and gives the project implementation status. Last, appendix A describes the SQL interface and long field operations.

For space considerations, this version of the long field paper has omitted some details and many diagrams. For more details, the interested reader should obtain the corresponding IBM technical report.

2 Starburst Long Field Manager Design Motivation

2.1 Design Goals

Three main goals drove the design of the Starburst Long Field Manager:

1. Storage allocation and deallocation must be efficient. We expect long field sizes to be large, on the order of 100 megabytes. It is important that the allocation mechanism minimize the time spent allocating and deallocating space to hold a long field.
2. The long field manager must have excellent I/O performance; long field read and write operations should achieve I/O rates near disk transfer speeds.
3. Long fields must be recoverable, but the recovery mechanism must not substantially slow down the long field operations.

To see how we might achieve these goals, we look at existing work in storage systems.

2.2 Previous Work in Storage Systems

Storage management of some kind is required by almost all large software systems. Overall, database systems and file systems have spent the most effort on designing fast and efficient storage mechanisms.

Database Systems

Past database systems were designed to manage simple facts — values that could be represented in fields of 255 bytes or less. Larger fields, such as those containing thousands or millions of bytes, presented problems to the database record manager, so the large fields were typically implemented *via* a separate long field mechanism.

The first SQL relational database system, System R [Astrahan 76], supported long fields with lengths up to 32,767 bytes. The System R long field manager divided long fields into a linked list of small manageable segments, each 255 bytes in length. Operations were restricted to reading and writing entire long fields; partial reads or updates were not supported. Later, an extension to SQL was proposed that provided operators for manipulating long fields [Haskin 82]. A new interface, the long field cursor, provided the ability for partial reading and updating of long fields. Along with the language extension, a storage mechanism was proposed that stored long fields as a sequence of 4 kilobyte data pages (rather than the previous scheme of a linked list of 255 byte records). The maximum length of a long field in extended SQL was about 2 gigabytes [SQL 85].

The Wisconsin Storage System (WiSS) [Chou 85] concurrently developed a similar mechanism for storing long fields. A WiSS long field was split into 4 kilobyte data pages, called *slices*. To reduce internal fragmentation, a *crumb*, a partially filled slice managed similarly to a database record, was used to hold the last segment of a long field if it did not occupy a full slice. A long field was represented by a directory of slices, plus a crumb. WiSS long fields had a size limit of 1.6 megabytes.

A more recent database system, EXODUS [Carey 86], stores all data objects in a general-purpose storage mechanism that can handle objects of any size — the limit is imposed by the amount of physical storage available. EXODUS uses a data structure that was inspired by the ordered relation data structure proposed for use in INGRES [Stonebraker 83]. The data structure is basically a B^+ Tree indexed on byte position within the object, with the B^+ Tree leaves as the data blocks. Objects smaller than a page are stored as single records (in a sense, a degenerate case of the $B+$ Tree structure — a single node). The EXODUS data structure is designed more for random access than for sequential access, although the leaves (the data blocks) may be configured to comprise several sequential disk pages if scan performance is being emphasized.

None of the database systems we've examined have emphasized high performance with respect to reading and writing large objects. Typically, performance issues are restricted to transaction processing systems involving "normal" database fields, those fields that are smaller than 255 bytes in length. The majority of the work involving the movement of large amounts of data at high speed has been in the area of file systems.

File Systems

File systems were used to manage large objects before database management systems existed. File systems associated with batch-oriented operating systems were designed to keep data stored either contiguously or clustered in physical extents, as data was read and written sequentially at high speed. Files with sequentially allocated disk blocks had excellent I/O performance, but sequential allocation caused problems with disk space due to external fragmentation. The next best thing to sequential allocation was physically clustered allocation in extents, or cylinder groups.

Early IBM operating systems such as DOS had file systems that left disk allocation up to the user; the user specified the number and location of disk extents required to hold the file. Later systems, such as IBM's OS/MVS, used hints from the user to plan for initial storage amounts as well as for future growth. IBM's CMS file system running under VM provided both automatic sizing and placement. Using the reasoning that blocks recently written will be read soon, the CMS file system allocated extents (disk tracks) that were close to the current position of the virtual disk arm.

The DEMOS file system [Powell 77], designed for CRAY computers, used the notion of physically clustered 4 kilobyte disk pages to enhance I/O performance. The inventors realized that sequential scanning of files, as opposed to random access, was the more frequent mode of access, so they designed accordingly. When a new disk block was added to the end of a file, it was allocated as physically close to the last block as possible. In order to prevent multiple files being written simultaneously from confusing the system, DEMOS also used a preallocation strategy to allocate a set of blocks for a file rather than single blocks.

The original UNIX file system [Ritchie 74] used 512 byte data pages that were allocated from random disk locations. The inventors seemed to follow the reasoning that, compared to batch systems, timesharing systems have less strict disk organization requirements. For example, with multiple users making disk requests, disk traffic can appear random. Even when two clients have contiguously allocated files, if their read requests are interleaved the disk arm can oscillate between the two file areas and eliminate any advantage of file clustering. The UNIX random page allocation scheme provided only medium-level performance at best, as virtually every disk page fetch operation resulted in a disk seek. Stonebraker noted that this results in poor performance for database systems that use the UNIX operating system [Stonebraker 81].

The UNIX Fast File System [McKusick 84] is an improvement over the original UNIX file system because it uses the idea of physical clustering of disk pages. The Fast File System uses larger data pages that are allocated from cylinder groups, thus sequential scan performance is improved significantly. For large files, a maximum of a megabyte of space is allocated from each cylinder group. The justification for this is that no single file should get all the space in a cylinder group, and that a megabyte of data between disk seek operations produces a respectably high transfer rate. To reduce internal fragmentation, the last large page can be split into smaller pages (fragments).[1]

The file system used by the Dartmouth Time Sharing System (DTSS) [Koch 87] uses the binary buddy system [Knuth

[1] Many page and fragment sizes are mentioned in [McKusick 84], but a common choice has been a 4 kilobyte page and a 1 kilobyte fragment.

75] for space allocation. The binary buddy system provides DTSS with variable size disk extents whose sizes are powers of two (units are disk pages).

When the file size is known in advance, DTSS allocates its disk extents in a manner that minimizes internal fragmentation. DTSS uses the bit representation of the size of the file (in pages) to determine the sizes of the extents needed to contain the file. Using N extents for file storage, the size of the file (in pages) is rounded up to the smallest binary number that contains N or fewer ones. For example, using 3 for N, 23_{10} (10111_2) would be rounded up to 24_{10} (11000_2) (one segment of 16 pages and one segment of 8 pages), 75_{10} (1001011_2) would be rounded up to 76_{10} (1001100_2) (one segment of 64 pages, one segment of 8 pages and one segment of four pages), and 231_{10} (11100101) would be rounded up to 256_{10} (100000000_2) (one segment of 256 pages). When the file size is not known in advance, DTSS allocates extents increasing in size until the file is contained. Later, the storage for the file is reallocated using the scheme described above once the file size is known.

Lessons Learned

Most database systems are quite primitive in their treatment of long fields. They treat the long fields as single values; although larger than simple scalar values, the long fields are read or written in a single operation. Those few database systems that handle long fields somewhat gracefully still do not pay particular attention to performance.

File system designers have addressed the I/O performance problem for large files and have found effective solutions. A file system using large disk extents can provide better performance than file systems using physically clustered disk pages, such as the UNIX Fast File System. A comparison of DTSS against the UNIX Fast File System shows that DTSS is much more efficient in allocating disk space, and through the use of large disk extents supplied by the buddy system, DTSS is able to sustain a high disk throughput with much less CPU usage. DTSS uses an average of three extents to hold a file, although it is able to allocate as many as twenty-four extents per file. The average number of extents per file is a system parameter, and is chosen to minimize the number of disk seeks per file scan, while providing a maximum of 3 percent disk space loss to fragmentation.

A survey of four systems using DTSS showed that the median static file size ranged between 6.4 kilobytes and 25 kilobytes, and that 98 percent of the files were under 64 kilobytes [Koch 87]. Using an average of three disk extents to hold files with these sizes was reasonable, but larger sizes, such as 100 megabytes, would probably require both larger extents and a larger number of them. The Starburst long field manager does exactly that.

3 The Long Field Manager Storage Design

3.1 Storage Design Overview

Figure 1 presents an overview of the data structures used for long field storage management. Given that it is not practical to store a long field directly in a relation[2], the next best thing is to store a long field descriptor in the relation. The long field descriptor is a *single level* directory of disk extents. We chose a size limit of 255 bytes for the long field descriptor because it is stored in a tuple as a field value, and since tuples may not span pages, it is important to minimize the size of the descriptor.

[2]IBM's Database 2 (DB2) actually does this, but it also restricts the size of a long field to 32 kilobytes—the size of the largest DB2 data page.

The variable disk extents, called *buddy segments*, are taken from *buddy spaces* which are large fixed-size sections of disk that are reserved for long field use. Buddy spaces are taken from an even larger (variable size) portion of disk, labeled *DB Space* in figure 1. DB Space is a Starburst term for an area of disk typically used to hold a set of tables or indices. A buddy space comprises an allocation page and a data area. The allocation page describes the state and size of the individual buddy segments in the buddy space data area. Buddy segments contain only data—no control information—and therefore (given suitable system facilities) the data can be transferred directly from disk to an application's buffers.

3.2 The Buddy System

Discussion

The buddy system works well at managing block sizes that differ by several orders of magnitude [Knuth 75]. It is known for fast allocation and automatic coalescing of blocks on deallocation, but it has also been reported to suffer from poor space utilization due to fragmentation [Bromley 85, Chowdhury 87, Lloyd 85, Page 86, Peterson 77]. Many of the negative reports about the buddy system come from those trying to use the buddy system to allocate a *single* block of storage rather than a set of blocks. All buddy systems (binary, weighted, and fibonacci) have poor storage utilization due to external fragmentation when single extents are used to store data.

However, when multiple blocks are used, both internal and external fragmentation can be reduced greatly. The bounded fragmentation work done by Bryant and Franaszek [Bryant 85] at IBM's Yorktown Research Laboratory shows that the buddy system can be used effectively to address the problems of internal fragmentation and external fragmentation. Since the binary buddy system can supply buddy segment sizes in 2^N units (the units typically being disk pages), it is possible to allocate the correct sizes of buddy segments so that there is less than a single disk page lost to internal fragmentation for any long field.

Therefore, despite the Buddy System's reputation for poor space utilization, it is our choice as the space allocation method for the long field manager because we plan to use it effectively. We use a set of blocks in a range of sizes (including the smaller sizes) to reduce external fragmentation and trimming of blocks (into smaller buddy system blocks) to reduce internal fragmentation.

Description

We refer to a variable-size segment allocated with the buddy system as a *buddy segment*. In the Binary Buddy System, the size of a buddy segment is a power of two, where the units are disk pages. Segments of the same size sharing ancestors are considered buddies (*e.g.* segments 0000 and 0001 are buddies at the 1 unit level, segments 1000 and 1100 are buddies at the 4 unit level). Calculating the address of a buddy for a particular size is straightforward in the binary buddy system. The address of a segment XOR'd with its size gives the address of its buddy. For example, to find the

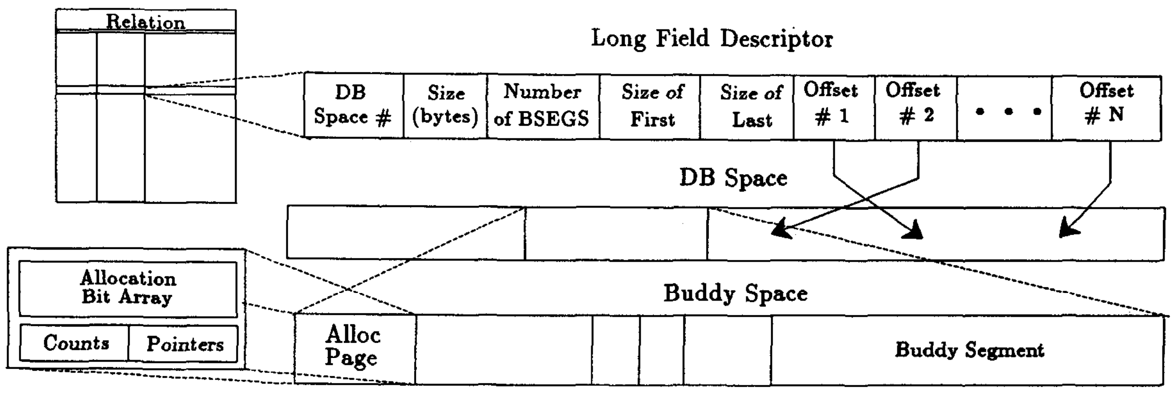


Figure 1: A Storage Overview of the Long Field Manager

size 4 buddy of block 1100, take the XOR of 1100 and 100, which gives 1000 — the size 4 buddy of block 1100 is 1000.

3.3 The Starburst Long Field Descriptor

A long field descriptor completely describes the storage of a long field. The descriptor is small, less than 255 bytes, yet it holds the disk storage information for long fields as large as 100 megabytes.[3] The descriptor contains a directory of buddy segment pointers, which allow direct access to long field data pages. The long field descriptor makes use of the property that the variable size segments are powers of two, thus segment sizes can be expressed as $log_2(size)$, rather than $size$ which would require more storage.

3.4 Long Field Manager Algorithms

Creating a Long Field

Creating a long field involves: creating a long field descriptor, allocating buddy segments to hold the long field, and setting the array of offsets in the long field descriptor to point to the buddy segments. A long field may be allocated with or without prior knowledge of size. When the eventual size of a long field is not known *a priori*, successive segments allocated for storage double in size, (*e.g.* the first segment is a single page (1024 bytes), then the next segment is two pages (2048 bytes), then 4 pages, then 8 pages and so on until the maximum size of 2048 pages (2 megabytes) is reached). Once the segment size reaches the maximum, a sequence of maximum size segments is used until the entire long field is stored, up to the maximum long field size (120 megabytes). When the size of a long field is known before creation, maximum segments are used, rather than increasing sizes. The use of doubling sizes has three advantages:

1. It does not unnecessarily allocate large blocks that would then be broken into smaller blocks during the trimming process. (Long Field Trimming is explained in the next section.)
2. It uses the smaller sizes, thus reducing external fragmentation.
3. It avoids storing size information for each segment in the long field descriptor, since the segment size is implicit given the size of the first segment and the known pattern of growth.

[3]The design allows us to support up to approximately 400 megabytes if necessary. If even larger fields are needed, then the application must manage a set of maximum length long fields.

Sometimes, the exact size of a long field is not known, but an approximate size is available. Given a hint of approximate size from the long field append function, the long field manager will start with an appropriately sized first segment and allocate doubling segment sizes from there.

When allocating buddy segments for a long field, the allocator attempts to allocate buddy segments from the same buddy space, which is roughly similar to a contiguous group of cylinders. Therefore, we expect that disk seeks will be small even between buddy segments.

Long Field Trimming

To reduce *internal fragmentation*, the last long field segment is trimmed to the nearest page boundary whenever an application closes a long field (typically upon transaction commit). Consider how a segment of 16 pages would be trimmed to hold a string of bytes that occupy 11 pages. The sequence of segment sizes needed is straightforward; it is the binary representation of the number of pages needed to hold the long field. 11_{10} is 1011_2, thus there's a segment of size eight, a segment of size two, and a segment of size one. The remaining segments (size one and four) are released. After trimming, the last segment is logically a *set* of buddy segments, not a single segment. However, since they are contiguous, the last buddy segment pointer in the long field descriptor may still point to the first buddy segment in the set of segments. This representation may change if append operations (explained below) are performed on a long field that has been trimmed.

Appending to a Long Field

We believe that most long field applications, such as those involving voice, image, sound, or video, will read and write whole long fields and not perform update operations that affect only parts of long fields. There are some applications, however, that may find it necessary to perform partial updates to a long field. An application using the long field mechanism to contain a log file, for example, might perform append operations on a long field.

In most cases, the last segment of a long field will be trimmed to some fraction of its original size (to reduce in-

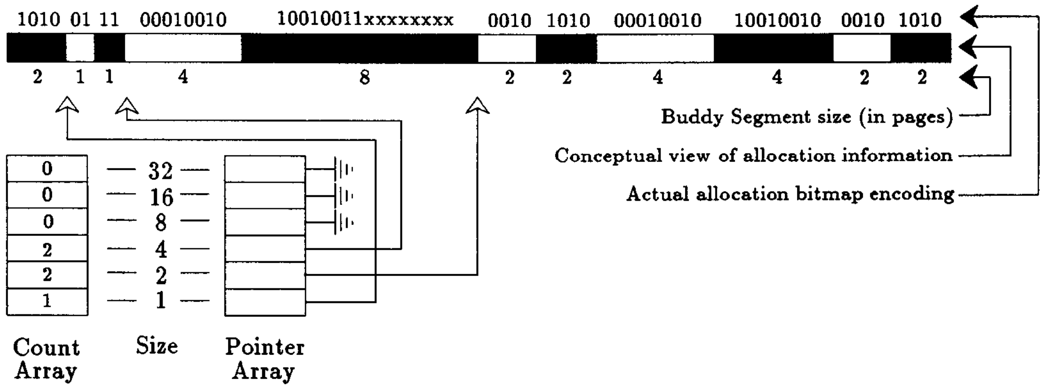

1010 01 11 00010010 10010011xxxxxxxx 0010 1010 00010010 10010010 0010 1010
2 1 1 4 8 2 2 4 4 2 2
Buddy Segment size (in pages)
Conceptual view of allocation information
Actual allocation bitmap encoding
0 0 0 2 2 1
32 16 8 4 2 1
Count Array
Size
Pointer Array

Figure 2: Allocation page structure

ternal fragmentation). In keeping with the design of the long field descriptor, an append operation that required more space than the existing trimmed segment would require a new segment with a size equal to the original size of the trimmed segment. The data from the trimmed segment would be copied to the new segment, and then the new data would be written to the new segment. If the new segment had extra space after the append operation, it would also be trimmed. Clearly, this would not be an efficient mechanism, as appending to very long fields would result in several megabytes of data being copied for each append operation.

Instead, when the long field manager receives an append request for a few bytes on a large object, it modifies the long field descriptor to handle appends more gracefully. The last segment offset in a long field descriptor typically points to a trimmed buddy segment, which consists of several smaller contiguous segments that are treated as a single segment. When an append operation is performed on a long field, the set of segments making up the trimmed segment is broken into separate segments, and the segment pointer for the trimmed segment becomes a directory for the new set of segments. Hence, just as the descriptor uses increasingly larger blocks for unknown sizes, it uses decreasingly smaller sizes for frequent append operations.

3.5 Long Field Manager Space Allocation

Long Field Descriptor Parameters

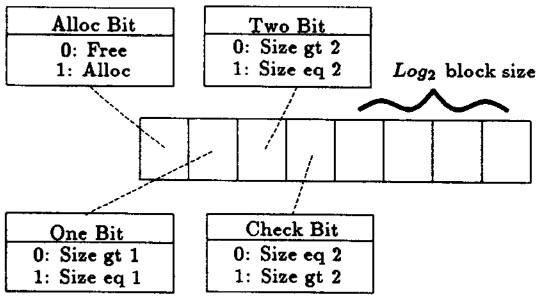
As we stated earlier, the long field descriptor must be minimized. The first five fields use 14 bytes, leaving about 240 bytes for the array of segment offsets (80 entries). Using a conservative estimate, a long field may have both increasing segment sizes, starting with a single page segment, and may also have decreasing segment sizes (because of appending). Using a 1 kilobyte data page, a maximum buddy segment size of 2 megabytes (2^{11} pages), would result in up to 11 slots being used for increasing sizes (the 11 slots hold a total of approximately 2 megabytes), up to 11 descriptor slots being used for decreasing sizes (also approximately 2 megabytes), and 58 slots being used for maximum size segments (116 megabytes), for a total of about 120 megabytes. (Notice that a long field composed of only maximum size buddy segments would hold more, but we chose to set the maximum long field length to the minimum of the possible maximum sizes.) Since the maximum length of the long field descriptor is fixed, the maximum length of a long field varies with the maximum buddy segment size. A maximum buddy segment size of 1 megabyte would result in a maximum long field size of 62 megabytes, a maximum buddy segment size of 4 megabytes would hold a long field of 232 megabytes and a maximum buddy segment size of 8 megabytes would hold a long field of 448 megabytes. Our current design, based on a 4 kilobyte allocation page, does not support buddy segments larger than 8 megabytes.

Page Allocation and Bitmap Encoding

Space in a long field DB Space is controlled by *allocation pages* stored in the long field DB Space. Recall from Figure 1 that the DB Space is divided into *buddy spaces*, and each buddy space contains an allocation page at the beginning of the space. Each allocation page controls about 16 megabytes of disk storage. Figure 2 shows the structure of an allocation page and the allocation bitmap representation. The allocation page controlling a buddy space has three parts: the allocation bitmap, the buddy segment Count Array, and the buddy segment Pointer Array.

The Count Array and the Pointer Array increase the efficiency of the search for a buddy segment. The Count Array shows the number of buddy segments of each size available in the space controlled by an allocation page, so that an allocator can determine immediately if it should look in a given allocation page for a given segment size. The Pointer Array provides a place to start looking for a segment of a particular size, so rather than starting each search from the beginning of the bitmap, the Pointer Array shows the first place where an available segment was last seen (by anyone updating the allocation page). On some occasions, the Pointer Array may point to a segment that is available, but on other occasions, the segment may have recently been allocated, hence the Pointer Array actually provides a *hint* rather than an absolute location. However, the pointer for a particular buddy size is guaranteed to be at least a correct starting point for a search for that size buddy segment. As segments are allocated, the Count Array is updated and the Pointer Array is updated to point to the location of the newly allocated segment (this is how the segment pointer moves forward). As segments are freed, the Count Array is updated and the Pointer Array is set to point to the segment closest to the beginning (this is how the segment pointer moves backward). Upon initialization, all of the pointers in the Pointer Array point to the first available segment of each size.

An obvious method for representing allocation informa-



Examples

Bits	Meaning
11	A segment of 1 page (Allocated)
01	A segment of 1 page (Free)
1010	A segment of 2 pages (Allocated)
0010	A segment of 2 pages (Free)
10010100	A segment of 16 pages (Allocated)
00011000	A segment of 256 pages (Free)

Figure 3: Allocation bitmap encoding

tion in bitmap form would be to use a single bit for each page, where a '1' in a page's bit position means that the page is allocated and a '0' means that the page is free. With the buddy system, however, more information is needed: namely, the size of the buddy segment (in pages). We use two bits per disk page so that we can represent both the allocation status and the size of the buddy segment.

Two bits are sufficient for representing the size and allocation status of a single page buddy segment. When a buddy segment is larger than a single page, for every page in the buddy segment there are two additional bits available for size and status information. Eight bits are sufficient for representing the allocation status and size of segments that range from 1 page to 32768 pages.[4] Therefore, when eight bits are available, that is, when a segment is at least four pages long, the general purpose representation is used. When the segment size is one or two pages, then the specific representation for one or two pages is used. Figure 3 shows the bit encoding used to store the size and allocation information of various sizes of buddy segments. Notice that at most eight bits per segment are used to hold information, while any remaining bits are left unused.[5]

Encoding the size in the page allocation bitmap gives a considerable advantage when looking for a particular segment size. First, at most eight bits must be examined, regardless of the segment size, to determine the segment size and allocation status. Second, when a segment is freed, a simple arithmetic operation on the freed segment's address gives the address of the buddy which can then be checked immediately for possible coalescing.

Also, because of the buddy system, a segment of size 2^N can appear only on boundaries where the address is 0 modulo 2^N. Thus, looking at the size of a segment on those boundaries provides an immediate direction of where to look next.

With our allocation bitmap encoding, the bitmap *must* be read on *valid* segment boundaries. This invariant is maintained by the pointer array, which guarantees that the starting place for a search is valid. Without this invariant, an allocator searching the bitmap could easily become confused. Notice in Figure 2 that the segment of size 8 has 16 bits available, but uses only 8 bits. The trailing 8 bits do not carry any useful information, and could mislead an allocator that tried to read the bitmap on that nonsegment boundary.

4 Concurrency and Recovery for Long Fields

Concurrency control is managed by the database system at the record level, hence there is no user-level concurrency control mechanism needed for the long field manager. Locking the record that contains a long field descriptor also locks the long field associated with that record.

We examined the two main techniques for recovery of long fields: logging and shadow techniques. Shadowing [Lorie 77] is a recovery technique in which current page contents are never overwritten. Instead, new pages are allocated and written, while the pages whose values are being replaced are retained as shadow copies until they are no longer needed to support the restoration of the system state due to transaction rollback. Logging [Gray 81] is a well-known database recovery technique which supports state restoration and reconstruction of values that are updated in place by recording in a system log the old and new values of the updated data items. We chose the shadow technique for long field data because of its simplicity [Traiger 82] and its potential for causing fewer I/O operations. However, since the shadow technique does not provide protection against media failures some additional measures are necessary; the long field data can be stored on duplexed disks or logged. Duplexed disks allow higher throughput, but they are costly. Logging is less expensive, but it allows less throughput and has the additional problem of potentially degrading the performance of the entire system. We are currently investigating these alternatives.

When the Starburst long field manager updates (inserts or deletes) a long field, it creates a new copy of the updated long field data segments (which it transfers directly to disk) and logs the changes to the long field descriptor and the long field allocation page(s). The replaced long field data segments remain as shadow copies until the updating transaction commits. When a long field data value is deleted or replaced, the allocation information for the corresponding space in the long field file is marked free, but the allocation information itself is locked against *all* transactions (even the deleting transaction) until the deleting transaction completes, thus ensuring that deletes will be undoable. Should a delete operation need to be undone, the deleted space will be reclaimed and the long field descriptor will be reset to reference its original set of data segments. Locking the freed storage presents an interesting problem, since at the time the space is freed, the buddy system may auto-

[4]The number 32768 could be pushed to 262144, since we don't need to represent 2^0, 2^1, or 2^2, as they're done with a different encoding. However, we didn't need to add this extra bit of complexity because we plan on using a maximum of only 2048 pages in our design.

[5]Those bits would be used if the segment is split into smaller segments.

matically coalesce the released buddy segment into a larger buddy segment, which could possibly be coalesced into an even larger buddy segment, and so on. The newly freed (and possibly coalesced) buddy segment may have a different address from the originally freed buddy segment. As a result, simply locking the address of the released buddy segment is not adequate.

Our solution uses a set of exclusive-mode locks, intention-mode locks and instant conditional locks. Allocate (A) and intention allocate (IA) locks are *instant conditional* locks; they are not actually held like regular locks. Setting an instant conditional lock on a segment simply tests the lock status of the segment; it does not actually lock a segment, nor does it cause the requester to block in the event the segment is already locked. When an allocate lock is denied, the requester simply moves on to the next available buddy segment and tries again. An allocation bitmap structure is protected by an allocation page latch while a requestor attempts to set allocate locks, so mutual exclusion of multiple requestors is guaranteed.

When a buddy segment is freed, a release lock (R) is placed on its address and size, and an intention release lock (IR) is placed on all of its ancestors. These locks are held until transaction completion. When a transaction attempts to allocate a buddy segment, it sets instant conditional intention allocate (IA) locks on all ancestors of the intended buddy segment, and then sets an instant conditional allocate (A) lock on the intended buddy segment. The lock matrix is as follows:

		Lock Held		
		None	IR	R
Lock Requested	IR	OK	OK	n/a
	R	OK	n/a	n/a
	IA	OK	OK	NO
	A	OK	NO	NO

If an ancestor is locked with a release (R) lock, the intention allocate lock will be denied, as the ancestor must remain unallocated until its holding transaction completes. If the intended buddy segment is locked with either an intention release lock (IR) or a release lock (R) then all or part of the buddy segment must remain unallocated until its holding transaction completes.

An important aspect of any hierarchical locking scheme is the number of lock granularities. With 12 sizes of buddy segments (our current design), releasing a buddy segment of size 1 would require setting 1 release lock and 11 intention release locks. Rather than set a lock at each level, locks are set at only certain specified levels (granularities). For example, Figure 4 shows how we might set locks only at segment sizes 2 and 8 in a system with only 5 sizes. Setting locks at only certain levels trades concurrency for overall locking cost, as each lock may cover more segments than just the segment requiring a lock. In Figure 4 for example, segment 0011 is locked along with segment 0010, and pages 1000–1011 are locked along with segment 1100.

Related Work in Recovery

Numerous other methods have been used for supporting recoverable long fields or long sequential byte strings. An IBM database product, DB2, stores long fields directly in normal database records. DB2 records may not cross page boundaries, and since the largest page size in DB2 is 32 kilobytes, the maximum long field size in DB2 is 32 kilobytes. Recovery of long fields is the same as for other types — write-ahead logging. Another IBM database product, SQL/DS, has a long field manager that stores the long field value in a sequence of blocks, each block being small enough to fit into a database page. Each of these blocks has the format of a database record and the blocks are linked into a linear list whose head is in the database record which logically contains the long field. The values of the block of the long field are recovered using the "standard" database log techniques. As a result, the contents of an updated long field are written both to the log and to the database.

The remaining related work in recovery comes from the domain of "transactional file systems". A useful survey of several file systems which support transactions can be found in [Svobodova 84]. The predominant approach for these systems is the use of *shadow* copies which is similar to our approach. Most of the systems rely upon specialized techniques to recover control and allocation information following a failure. XDFS and CFS [Mitchell 82] both use shadow pages but depend upon *careful replacement* of control and allocation information to insure failure recovery. Both of these systems require extensive scanning of secondary storage to validate and reconstruct control information.

Paxton [Paxton 79] describes a shadow based file system which uses a form of careful replacement for control information known as *intentions lists*. Paxton's approach to storing long byte string values is similar to our method in that it uses shadow page replacement as the basic recovery mechanism. However, the use of intentions lists as the mechanism for transaction commit requires at least 3 extra write operations to reliably update the control information associated with the long byte string value. Our method requires only 1 extra (log) write of control information (the long field descriptor) to recoverably update a long field.

The Alpine file system [Brown 85] logs file content updates and copies file content from the log to the file following the commitment of the transaction modifying the file. This leads to 3 I/O operations per file page modified by the transaction. While file content changes in Alpine use a redo only log, file allocation operations are propagated immediately to secondary storage and use an undo log to support transaction abort. The disadvantages of the Alpine approach include the copying needed to propagate logged changes to the file proper and the need for synchronous I/O to log the update file control information. Our method writes long field content changes only to the long field file blocks and relies upon database recovery logging for control and allocation information recovery which implies that these changes do not need to be written synchronously and can be piggybacked on other log I/O operations.

The work closest to our method of long field recovery is the new file system for the Cedar Operating System [Hagmann 87]. The new Cedar file system uses the shadow technique to recover file content pages and uses a log to recover changes to file control information. File data pages are not logged and are vulnerable to media failure. Allocation data is not logged either, but it can be recovered by scanning the extent tables of all files. The logging and recovery granularity is the page and the log is a redo only log. Reservation of space for released pages is accomplished by postponing the allocation table updates until transaction commit. Our method uses record-level (sub-page) logging and recovery and uses the log for recovery of both control information

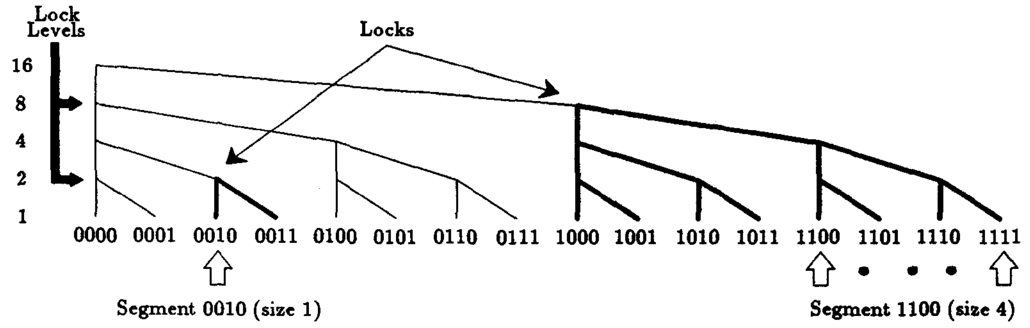


Figure 4: Sparse locking of buddy segment sizes.

and allocation data.

5 Summary

We designed the Starburst long field manager to manipulate long fields, with sizes in the 100 megabyte range, with great speed and efficiency. The long field manager uses the buddy system to manage disk space, which is compatible with growth and trimming algorithms that increase storage efficiency. Long field data segments contain only user data, which allows the long field manager to transfer data directly from disk storage to application buffers without copying data to an intermediate system buffer.

Recovery for long fields is based on shadows for long field data and write-ahead-log for long field allocation and descriptor information. The use of shadows prevents old and new values from both being saved — only the new pages need to be recorded. Locking of buddy segments for space reservation purposes uses exclusive-mode locks, intention-mode locks, and instant conditional locks in order to update the allocation information in place and perform concurrent space coalescing without allowing other transactions to see the updates to the allocation tables until the holding transaction completes.

Long Field Manager Status

The majority of the Starburst long field manager design and implementation took place during the summer of 1985. After some preliminary testing validated the low levels of the space allocation mechanism, further implementation on the Starburst prototype was postponed until several other necessary components were finished. We are just now finishing the implementation of the long field manager. We plan to run numerous benchmarks, comparing our long field manager performance to that of available file systems. One special benchmark in our plans involves storing a full length feature film (a Fred Astaire video — uncolorized) using the long field manager and then playing the video in its entirety in an application window. An uncompressed 2 hour video digitized for a 1 megabyte screen would be approximately 216 gigabytes, but using video compression hardware, we calculate that a 2 hour video will require approximately 500 megabytes.

The basic design of the Starburst long field manager was also implemented for IBM's Operating System/2 (OS/2) and incorporated in IBM's OS/2 Extended Edition Database Manager.

6 References

[Allchin 80] J. E. Allchin *et al.*, "FLASH: A Language-Independent, Portable File Access System," *Proc. ACM SIGMOD*, (May 1980).

[Astrahan 76] MM. Astrahan *et al.*, "System R: Relational Approach to Database Management," *ACM TODS*, Vol. 1, No. 2 (June 1976).

[Bromley 80] A. G. Bromley, "Memory Fragmentation in Buddy Methods for Dynamic Storage Allocation," *Acta Informatica*, Vol. 14, pp. 107–117 (1980).

[Brown 85] M. Brown *et al.*, "The Alpine File System," *ACM TOCS*, Vol. 3, No. 4 (November 1985).

[Bryant 85] R. Bryant and P. Franaszek, "Storage Allocation Policies via Bounded Fragmentation," *IBM Research Presentation*, Summer 1985.

[Carey 86] M. J. Carey *et al.*, "Object and File Management in the EXODUS Extensible Database System," *Proc. 12th VLDB*, (August 1986).

[Chang 89] W. Chang and H. Schek, "A Signature Access Method for the Starburst Database System," *Proc. 15th VLDB* (August 1989).

[Chou 85] H-T Chou *et al.*, "Design and Implementation of the Wisconsin Storage System," *Software Practice and Experience*, Vol. 15, No. 10 (October 1985).

[Chowdhury 87] S. K. Chowdhury and P. K. Srimani, "Worst Case Performance of Weighted Buddy Systems," *Acta Informatica*, Vol. 24, pp. 555–564 (1987).

[Eswaran 76] K. P. Eswaran *et al.*, "The Notions of Consistency and Predicate Locks in a Database System," *CACM*, Vol. 19, No. 11 (November 1976).

[Gray 75] J. Gray *et al.*, "Granularity of Locks in a Shared Data Base," *Proc. VLDB* (September 1975).

[Gray 81] J. Gray *et al.*, "The Recovery Manager of the System R Database Manager," *ACM Comp. Surveys*, Vol. 13, No. 2 (June 1981).

[Haas 88] L. Haas *et al.*, "Extensible Query Processing in Starburst," *Proc. ACM SIGMOD* (May 1989).

[Hagmann 87] R. Hagmann, "Reimplementing the Cedar File System Using Logging and Group Commit," *Proc. SOSP* (November 1987).

[Haskin 82] R. L. Haskin and R. A. Lorie, "On Extending the Functions of a Relational Database System," *Proc. ACM SIGMOD* (June 1982).

[Knuth 75] D. E. Knuth, *"The Art of Computer Programming. Vol 1, Fundamental Algorithms,"* Addison-Wesley, Reading Mass., 1969.

[Koch 87] P. D. L. Koch, "Disk File Allocation Based on the Buddy System," *ACM TOCS*, Vol. 5, No. 4 (November 1987).

[Lee 88] M. Lee *et al.*, "Implementing an Interpreter for Functional Rules in a Query Optimizer," *Proc. 14th VLDB* (August 1988).

[Lindsay 87] B. Lindsay *et al.*, "A Data Management Extension Architecture," *Proc. ACM SIGMOD* (May 1987).

[Lohman 88] G. Lohman, "Grammar-Like Functional Rules for Representing Query Optimization Alternatives," *Proc. ACM SIGMOD* (May 1988).

[Lorie 77] R. Lorie, "Physical Integrity in a Large Segmented Database," *ACM TODS*, Vol. 2, No. 1 (March 1977).

[Lloyd 85] E. L. Lloyd and M. C. Loui, "On the Worst Case Performance of Buddy Systems," *Acta Informatica*, Vol. 22, pp. 451–473 (1985).

[Mitchell 82] J. Mitchell and J. Dion, "A Comparison of Two Network-based File Servers," *CACM*, Vol. 25, No. 4 (April 1982).

[McKusick 84] M. K. McKusick, W. N. Joy, S. J. Leffler and R. S. Fabry, "A Fast File System for UNIX," *ACM TOCS*, Vol. 2, No. 3 (August 1984).

[Ousterhout 85] J. K. Ousterhout *et al.*, "A Trace-Driven Analysis of the UNIX 4.2 BSD File System," *Proc. 10th SOSP* (December 1985).

[Page 86] I. P. Page, "Improving the Performance of Buddy Systems," *IEEE Trans. on Computers*, Vol. C-35, No. 5 (May 1986).

[Paxton 1979] W. Paxton, "A Client-Based Transaction System to Maintain Data Integrity," *Proc. 7th SOSP* (December 1979).

[Peterson 77] J. L. Peterson, "Buddy Systems," *CACM*, Vol. 20, No. 6 (June 1977).

[Powell 77] M. L. Powell, "The DEMOS File System," *Proc. 6th SOSP* (November 1977).

[Ries 79] D. Ries and M Stonebraker, "Locking Granularity Revisited", *ACM TODS*, Vol. 4, No. 2 (June 1979).

[Ritchie 74] D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *CACM*, Vol. 17, No. 7 (July 1974).

[Rothermel 89] K. Rothermel, C. Mohan, "ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions," *Proc. 15th VLDB* (August 1989).

[SQL 85] R. Lorie and J. Daudenarde, "Design System Extensions User's Guide," (April 1985).

[Smith 81] A. J. Smith, "Algorithms and Architectures for Enhanced File System Use," *Experimental Computer Performance Evaluation*, pp. 165–193 (1981).

[Stonebraker 81] M. Stonebraker, "Operating System Support for Database Management," *CACM*, Vol. 24, No. 7 (July 1981).

[Stonebraker 83] M. Stonebraker *et al.*, "Document Processing in a Relational Database System," *ACM TOFS*, Vol. , No. 2 (April 1983).

[Svobodova 84] L. Svobodova, "File Servers for Network Based Distributed Systems," *ACM Comp. Surveys*, Vol. 16, No. 4 (December 1984).

[Traiger 82] I. L. Traiger, "Virtual Memory Management for Database Systems," IBM Research Report RJ3489 (41346) (May 25, 1982).

[Wilms 88] P. Wilms *et al.*, "Incorporating Data Types in an Extensible Database Architecture," Procs. of 3rd Int. Conf. on Data and Knowledge Bases, Jerusalem, (June 1988).

[Verhofstad 78] J. S. M. Verhosfstad, "Recovery Techniques for Database Systems," *ACM Comp. Surveys*, Vol. 10, No. 2 (June 1978).

A The SQL Interface and Long Field Operations

In order to meet its extensibility goal, Starburst must accept the incorporation of new data types, new storage methods, and new access methods in a straightforward manner. The long field, as we have defined it, is a new storage method, and the "large unstructured byte stream" is a new data type. The creator of a new data type defines the interface to instances of the data type, defines the operations on instances of the data type, and specifies the storage method.

In the Starburst project, the large byte stream is the first data type to be implemented that does not fit into the standard mold for data types. Unlike scalar values such as integers, floating point numbers, or even the original System R long field, a byte stream value is not manipulated directly. The user of a byte stream type first selects a "handle" on it through an SQL SELECT statement or a CURSOR operation.

In general, an application may define a "handle" for any type instance. A handle for a type instance is similar to an open file descriptor; it is used to reference that type instance when performing operations on it. Functions can manipulate type instances *via* a handle, or in some cases, they can manipulate the type instance itself. Instances of scalar types, for example, can be manipulated directly, but byte streams may be manipulated *only* though a handle.

Defining the external interface to byte streams requires two steps. First, we must extend the standard SQL language with a special "apply" operator to provide a mechanism for calling the new operations. Second, we define a set of operations that manipulate byte streams.

A.1 The SQL Interface

In order to extend the System R interface to long fields, Haskin and Lorie proposed an extended definition of cursors to manipulate long fields [Haskin 82]. The proposed extension to SQL applied specifically to long fields, and was not usable by other data types. The APPLY function, as suggested here, is a general interface that allows an application to call operations defined on instances of any extended data types.

We extend SQL with an APPLY operation, in addition to its SELECT, UPDATE, INSERT, and DELETE operations. Although we have not yet determined the exact syntax of the APPLY operation, the general idea is that an application will use the APPLY operation when it asks the database system to manipulate instances of extended data types. Historically, database systems simply store data, and leave the manipulation up to the application. With long fields, however, it is more efficient to have the database perform some of the operations.

An example of the APPLY operation is:

```
EXEC SQL APPLY Read( :lf_handle, :position,
:count, :buffer );
```

A.2 Operations

When an application opens a byte stream through an SQL SELECT or CURSOR operation, it is given a handle that refers to the open long field. The application then supplies that handle when calling the operations defined for long fields. The supported byte stream operations are:

Clear(lf_handle) Clear the long field and set its value to NULL. (We make the distinction between a NULL value and zero length.)

Truncate(lf_handle, position) Delete the bytes from <position> to the end of the long field.

Read(lf_handle, position, count, BUFFER) Read <count> bytes from the long field at <position> and put them into <BUFFER>.

Append(lf_handle, count, BUFFER, hint) Append <count> bytes, taken from <BUFFER>, to the end of the long field. If the long field is zero length or NULL, then use <hint> as a value for estimated size. <hint> is used when the long field is being written over several append operations and the actual size of the long field is not known.

Replace(lf_handle, position, count, BUFFER) Overwrite <count> bytes at <position>, taken from <BUFFER>.

Length(lf_handle, length) Return the length of the long field in <length>.