Towards Effective and Efficient Free Space Management*

Mark L. McAuliffe
University of Wisconsin—Madison
mcauliff@cs.wisc.edu

Michael J. Carey
IBM Almaden Research Center
carey@almaden.ibm.com

Marvin H. Solomon
University of Wisconsin—Madison
solomon@cs.wisc.edu

Abstract

An important problem faced by many database management systems is the "online object placement problem"—the problem of choosing a disk page to hold a newly allocated object. In the absence of clustering criteria, the goal is to maximize storage utilization. For main-memory based systems, simple heuristics exist that provide reasonable space utilization in the worst case and excellent utilization in typical cases. However, the storage management problem for databases includes significant additional challenges, such as minimizing I/O traffic, coping with crash recovery, and gracefully integrating space management with locking and logging.

We survey several object placement algorithms, including techniques that can be found in commercial and research database systems. We then present a new object placement algorithm that we have designed for use in Shore, an object-oriented database system under development at the University of Wisconsin—Madison. Finally, we present results from a series of experiments involving actual Shore implementations of some of these algorithms. Our results show that while current object placement algorithms have serious performance deficiencies, including excessive CPU or main memory overhead, I/O traffic, or poor disk utilization, our new algorithm consistently demonstrates excellent performance in all of these areas.

1 Introduction

This paper presents a new solution to an old and deceptively simple-sounding problem known as the "online object placement problem," the problem of choosing a disk page to hold a newly allocated object. Object placement has been an important problem since the days of the first flat-file database systems and continues to be important today. Yet to our knowledge, the database literature contains no studies of object placement from the standpoint of storage utilization and allocation performance.

Placing variable-size objects onto fixed-size pages[1] is an example of the online Bin Packing problem, for which simple, well-known algorithms provide reasonable utilization in the worst case and excellent utilization in common cases [JDU+74]. However, the storage requirements of a database system introduce significant extra challenges, such as minimizing I/O traffic, coping with crash recovery, and gracefully integrating space management with locking and logging. In particular, *reserved space*, space freed by transactions that have deleted persistent objects but have not yet committed, introduces an additional object placement obstacle.

This paper surveys current solutions to the object placement algorithm and introduces a new object placement algorithm known as HY(n,u). We present the results of a performance study based on an implementation of HY and other object placement algorithms in Shore [CDF+94], an object-oriented database system under development at the University of Wisconsin—Madison. This performance study highlights several of the features of the new algorithm. Among these features are:

Superior packing: HY(n,u) achieves excellent space utilization on all tested workloads.

Speed: HY(n,u) consistently runs as fast as the fastest of the algorithms in our study.

Simplicity: Our Shore implementation of HY(n,u) consists of fewer than 600 lines of C++ code. In addition, HY(n,u) requires no persistent data structures beyond those found in most commercial database systems.

Low memory usage: In our tests, we found 200 bytes of main memory sufficient to drive a two-gigabyte file.

I/O performance: When placing objects on pages, HY(n,u) avoids unnecessary I/O operations by

*This research is sponsored by the Advanced Research Project Agency, ARPA order number 018 (formerly 8230), monitored by the U.S. Army Research Laboratory under contract DAAB07-92-C-Q508.

[1] In this paper, we consider only objects that are smaller than one page.

maintaining a high degree of buffer pool locality and by avoiding reads of full and nearly-full pages.

Flexibility: The u parameter is a target disk utilization. It can be adjusted to trade off creation speed against density of packing (and hence I/O traffic for read-only traversals).

The rest of this paper is organized as follows. Section 2 surveys file organizations currently in use in commercial and research database systems. Section 3 describes data structures and object placement algorithms commonly used for free space management in database systems. Section 3 also introduces two new algorithms: an algorithm called NF(WH) (Next-Fit with Witnesses and Histograms) that fixes some but not all of the problems with existing algorithms, and HY(n,u), which is a hybrid of NF(WH) and another algorithm. Section 4 describes the Shore implementation of the object placement algorithms examined in our performance study. Section 5 presents the results of our performance study, and Section 6 summarizes and discusses our results.

2 File Organizations in Database Systems

Existing relational database systems typically use one of two file organizations as the primary storage structures for storing relations: heap files or B$^+$ trees. In some systems, the records of a relation that has a primary key are stored in the leaf pages of a B$^+$ tree indexing that key. Not only does this organization provide a fast access path for locating records by key, it also provides a mechanism for allocating space: the page on which a new record must be placed is completely dictated by its key. If that page is full, it may be split (using the usual B$^+$ tree insertion algorithm) or an overflow page may be chained from it, but there is no possibility of placing the record on some other existing leaf page. While this organization makes space allocation straight-forward, it does not lead to particularly good space utilization: Johnson and Shasha show typical space utilization for B$^+$ trees to be between 39% and 70% [JS93].

In a B$^+$ tree file organization, records do not have fixed physical addresses; they may be moved from page to page as other records are inserted into or deleted from the relation. The lack of fixed addresses presents a problem in the design of secondary indices, which need to "point" to tuples of the relation. Some systems, such as Sybase [Kir93], point to records via their physical addresses within the B$^+$ tree, requiring costly updates to all secondary indices whenever a record is moved. Other systems, such as Tandem's NonStop SQL [Tan87], use the primary key itself as a "pointer," incurring an extra level of index traversal on each lookup through a secondary index.

A more common file organization in current relational systems is a heap file, which is simply an unordered set of pages. A new record can be placed on any heap file page containing sufficient free space. The heap organization decouples a record's placement from its contents, allowing more flexibility in space allocation and clustering. It raises the question, however, of how to choose pages for the allocation of new records. Most commercial systems optionally allow a *clustering index* to be used to guide the placement of new records. A clustering index is a B$^+$ tree index on a distinguished attribute, called the *clustering attribute*, which is usually declared when the relation is created. The clustering attribute need not be a unique key, but it should correlate with common access patterns.

To insert a record into a relation with a clustering index, the system probes the index to find the page or pages containing other records with the same value as the new record (or a neighboring value) for the clustering attribute. The new record is placed on one of these pages or a nearby page, if possible. If the record cannot be placed on or near the page indicated by the clustering index, then any page of the file, typically the last page, is used. Over time, the file's clustering degrades as a result of these stray insertions, and also because of records that outgrow their original page and must be moved to a new page. When this degradation becomes severe, the file's clustering can be restored using a file reorganization utility.

Free space management for heap files has seen little attention in the relational database literature. In some circles, it is felt that free space management is not very important for commercial systems, as system administrators will simply buy enough disks to suit their needs (an attitude applauded by disk drive manufacturers). Furthermore, it is felt that object placement should be guided primarily by clustering decisions. However, in some relations, no attribute presents itself as a logical choice for a clustering attribute. Such relations are best stored as unordered heap files: the overhead of using and maintaining the clustering index is avoided, and, as our experiments will show, the file can see excellent space utilization.

In the world of object-oriented database systems, there is much less information available about system internals. We conjecture that many commercial object-oriented database systems use unclustered heap files, but we cannot prove it from the open literature. However, two recent research object-oriented systems with which we are familiar, Shore [CDF$^+$94] and its predecessor, Exodus [CDG$^+$90], use this organization. Heap files are a natural choice for object-oriented systems, as the navigational access patterns common in such systems suggest clustering together objects of assorted types based on structural relationships. Even if objects are stored in type extents, a type may not have any attribute suitable for use as a clustering attribute.

3 Free Space Management in Heap Files

The remainder of this paper is concerned with the management of free space in heap files. The next three subsections survey different aspects of free-space management. The first subsection describes persistent data structures used for free-space management. The second subsection describes current object placement algorithms and introduces two new algorithms, including HY(n,u). The final subsection describes *reserved space*, a potential pitfall for object placement algorithms.

3.1 Data Structures for Free Space Management

Every system of which we are aware uses one of two methods for keeping track of free space: space maps or free lists. Space maps are used by many systems, including Rdb [HE95], DB2 [IBM89], and Starburst [HCL+90], each of which calls them by a different name. In this paper, we use the term FSIP (Free Space Inventory Page) used by Mohan and Haderle [MH94][2]. FSIPs are pages placed at well-known positions of a volume or file that contain summary information about the amount of free space in each of a set of other pages. While it is possible for FSIP entries to contain the exact number of free bytes on each page, it is more common for them to record approximate information. The database system thus defines a small number of free-space classes, and the FSIP indicates to which class each page belongs.

Some systems simply classify pages as "open" or "closed" according to whether they have enough free space to allocate a "typical" record. With this scheme, the FSIP needs only one bit per entry. Other systems classify pages by the amount of free space on the page. In Shore, we divide pages into 15 such categories and store four bits of state information per page[3]. Storing approximate information allows more compact FSIPs, saving space and improving locality, but more importantly it implies that the class of a page changes less frequently, decreasing the overhead of updating (and logging changes to) the FSIPs. We have adopted the term *free-space bucket* (or just *bucket*) to refer to the set of pages belonging to the same free space class. Thus, for a given set of *threshold* values t, free-space bucket b refers to those pages whose free-space value f satisfies $t_b \leq f < t_{b+1}$. Figure 1 shows a five-page file and its associated FSIP.

The other structure commonly used to keep track of free space is a free list. A free list is a list of identifiers of pages that are likely to be able to hold a new object ("open" pages, in the terminology of our previous discussion). A page is entered onto the free list when it is first allocated, or when object deletions cause the page to cross a predetermined free-space threshold. Free lists are used in commercial systems much less frequently than FSIPs. In fact, Oracle is the only system we know that uses them [Stü95]. Free lists can be tricky to maintain in a transactional environment. In particular, aborting a transaction that has made modifications to a free list can be difficult if other transactions have made subsequent modifications to the list.

We have decided to focus our efforts on FSIP-based systems, as they are more common than free lists, and because the methods used to maintain them in a transactional environment are well understood and well documented [MH94].

3.2 Algorithms for Free Space Management

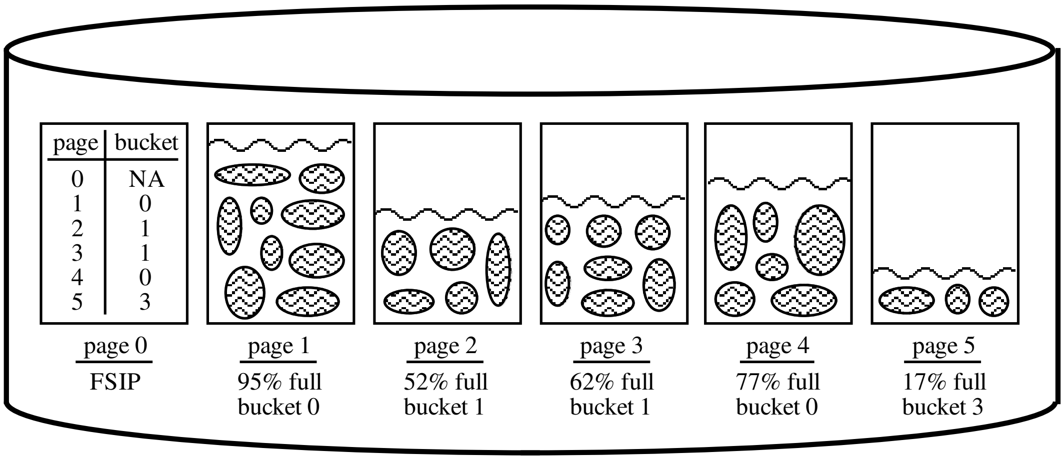
There are two conflicting goals for a space-allocation algorithm: it should run fast, and it should waste very little disk space. The simplest and fastest algorithm is to allocate new records out of one page until no more will fit, and then move to a new (empty) page. We call this algorithm *Append Only* (AO). AO can unnecessarily leave pages partially filled when a large record request forces it to move to a new page, even though subsequent requests for smaller records could be used to fill the previous page more completely. This effect can be mitigated by allowing new objects to be allocated from any of the n most recently added pages, instead of just the last one. We call this variant AO(n).

Under-filled pages can also be caused by record deletions, or by moving a record when it outgrows its original page. Over time, the cumulative effect of these under-filled pages is *file bloating*, which wastes disk space and unnecessarily increases the I/O cost of file scans. As shown in Appendix A, AO's disk utilization can be expected to approach $1/H_k$ in the limit, where k is the number of objects that fit on a page and H_k is the k^{th} harmonic number. For example, for 256-byte objects allocated from 8k-byte pages, utilization drops to $1/H_{32} \approx 25\%$, and for 32-byte objects, it drops to $1/H_{256} \approx 16\%$. Many commercial systems provide a reorganization utility that consolidates under-filled file pages, releasing any pages that become completely empty in the consolidation process. However, these utilities tend to be highly disruptive to normal system operation.

At the other end of the spectrum from AO are algorithms that look for unused space on existing file pages before resorting to adding a new page. An example of this kind of algorithm is *First-Fit* (FF), which searches the pages of the file sequentially from beginning to end until a suitable page is found. FF can be shown to have worst-case space utilization within a factor of 1.7 times optimal [JDU+74], and in practice it frequently achieves utilization well over 90%. On the other hand, it is an n^2 algorithm: each addition of a new page to the file is preceded by an unsuccessful scan of all of the pages added previously. The use of FSIPs allows algorithms like FF to eliminate from consideration pages

[2] We prefer the Rdb term "SPAM" (SPAce Map) page, but were afraid of being sued by Hormel.

[3] Shore's FSIPs are per-volume data structures, rather than per-file. The 16th code is therefore used to indicate pages that do not currently belong to any file.



bucket number	percent full
0	75-100
1	50-75
2	25-50
3	0-25

Figure 1: Persistent file data structures for a system that uses four free-space buckets (shown at right).

with insufficient space without actually reading them from disk, but as the measurements reported later will demonstrate, even the overhead of scanning the FSIPs is significant for large files.

Thus AO runs quickly, but fails to provide reasonable space utilization, while FF gives excellent utilization, but has unacceptably poor runtime performance. We now describe two algorithms that attempt to bridge the gap between FF and AO. Neither of these algorithms requires persistent data structures other than the FSIPs described above. They therefore have no effect on disk organization, log traffic, or crash recovery. The first algorithm is derived by adding three modifications to FF.

The first modification is well-known from main-memory allocation algorithms. Starting each search at the beginning of the file is a bad idea, since it tends to create a concentration of full pages near the beginning, leading to longer and longer searches over full or nearly full pages before finding a suitable page. Instead, we maintain a (transient) cursor, which records the page last used for allocation and start the next scan from that point, wrapping around to the beginning of the file when we reach the end. With this modification, the algorithm is generally called *Next-Fit* (NF).

In the classical (main memory) case, an unsuccessful complete scan of memory is a catastrophic failure; it indicates that memory is exhausted. In our case, we can "recover" by allocating a new page to the file. However, NF, like FF, will only add a page after an expensive, fruitless search. Thus in the worst (and common) case where there are mostly creations and few deletions, the algorithm is still order n^2. Our second modification is designed to avoid these fruitless searches. Recall that the FSIPs partition data pages into buckets, depending on how full they are. The modified algorithm maintains a transient histogram h that records a count of the number of pages in each bucket h_b. Before searching the file for a page with enough space to satisfy an allocation request, the modified algorithm consults the histogram to determine if any such pages exist. More precisely, when processing a request for n bytes, the algorithm checks for the existence of a histogram bucket b, with free-space threshold t_b, such that $h_b > 0$ and[4] $t_b \geq n$. Note that the histograms are only a "hint;" a transaction may consult the histogram and learn that an appropriate page exists, only to have the space on that page stolen by another transaction before it completes its search. We could avoid this problem by serializing searches, but it is clearly better simply to suffer the cost of the unnecessary search in this infrequent situation. We call the Next-Fit algorithm augmented with histograms NF(H).

While histograms are useful for avoiding searches that are doomed to fail, they do nothing to speed up successful searches. Our third modification is to add *witnesses.* Associated with each bucket is either the page identifier (physical address) of any page known to be in that bucket, called a "witness" for that bucket, or an *unknown* indication, meaning that no such page is currently known. The resulting algorithm first looks for a bucket that represents pages with sufficient free space and has a known witness. If such a bucket is found, then its witness is used to satisfy the request. Otherwise, we fall back on an exhaustive search of FSIPs, as in NF and NF(H). If satisfying the request causes the witness page to move to a different bucket, then the witness value for the original bucket is set to *unknown.* If the witness value for the page's new bucket is *unknown*, then the page can be used as a witness for that bucket. The collection of witness values forms a cache of pages with known amounts of free space. Witness values are refreshed when a page changes buckets, when a

[4]Because the free-space information is imprecise, bucket $b-1$ may contain pages that have enough free space to satisfy the request. All algorithms implemented in this study are conservative in the sense that they only consider pages whose FSIP value indicates that they are *guaranteed* to have enough space. The space wasted by this policy is negligible with sufficiently many buckets.

new, empty page is allocated, or when an exhaustive scan encounters an FSIP value for a bucket with an *unknown* witness (regardless of whether or not that page satisfies the scan). The Next-Fit algorithm augmented with both witnesses and histograms is called NF(WH). Figure 2 shows the witnesses and histograms maintained by NF(WH) for the file shown in Figure 1.

	0	1	2	3
witnesses:	4	1	NA	5

	0	1	2	3
histogram:	2	2	0	1

Figure 2: Transient data structures used by NF(WH) for the file shown in Figure 1. Note that because the file has no pages in bucket two, there is no witness for that bucket.

As our performance results will show, NF(WH) works quite well for a variety of workloads. Its packing is as good as FF, yet it runs nearly as quickly as AO. For workloads that generate large numbers of free-space holes, however, NF(WH) works very poorly. In its zeal to fill all holes, no matter how small, NF(WH) reads pages from disk even if those pages can accommodate only a single new object. It therefore generates excessive numbers of I/O requests, as well as exhibiting poor buffer-pool and disk locality. This problem is not unique to NF(WH); any algorithm that tries to maximize space utilization is subject to this weakness.

Our solution to this problem is a *hybrid* algorithm, HY(n,u), that combines variants of AO(n) and NF(H). Like AO(n), HY(n,u) maintains a (transient) data structure, called the *pid cache*, which records the page identifiers (pids) of up to n pages together with the amount of free space on each of them. As long as the file's space utilization remains above a pre-set threshold, given by the u parameter, HY(n,u) behaves like AO(n) and only examines pages in the pid cache before allocating a new page. However, if the file's utilization drops below u, HY(n,u) mimics NF(H); after failing to find a suitable page in the pid cache, it uses a histogram to determine whether to search for an existing file page on which to place the new object. This determination also depends on the value of u. HY(n,u) only considers pages whose utilization is less than u, and will only instigate a search if such a page exists. Note that the histogram maintained by NF(H) contains enough information to compute a quite precise estimate of the file's space utilization.

Once a page has been located and the new object has been placed on it, the page is considered for addition to the pid cache. If the page was already in the cache, HY(n,u) simply updates its free space information. Otherwise, if the page has more free space than a page already in the cache, it is added, replacing the fullest page in the cache. Figure 3 shows the witnesses and histograms maintained by HY(2,u) (pid cache size of two) for the file shown in Figure 1.

pid cache:		0	1
	page:	4	5
	bucket:	0	3

	0	1	2	3
histogram:	2	2	0	1

Figure 3: Transient data structures used by HY for the file shown in Figure 1. In this example, HY uses a two-entry pid cache, which currently contains pages four and five, the two most recently allocated pages.

3.3 Reserved Space

When a transaction deletes an object, the database system must ensure that if that transaction is rolled back, it will be able to restore the deleted object. Database systems that rely on physical record identifiers, in particular, may need to take special measures to allow deleted objects to be restored to their original pages. If exclusive-mode page locks are obtained for record deletions, then no further actions are necessary to fulfill this requirement: the lock prevents other transactions from making any modifications to the page. However, if intention-mode page locks are obtained (in conjunction with exclusive-mode object- or record-level locks), then a method called *space reservation* is typically used to fulfill this requirement. Space reservation sets aside a certain number of bytes on a given page for use only by a specific transaction. If the transaction commits without having used the reserved space, the reservation expires and the reserved space is released.

Techniques for releasing reserved space fall into two categories: *eager* and *lazy*. In an eager technique, a transaction re-visits each page containing deleted objects at commit time, updating the free-space and reserved-space counts on the page. It also updates the corresponding FSIP entry if necessary. In a lazy method, a transaction that reserves space on a page simply places enough information on the page to allow future transactions to determine whether the reservation is still in effect. Lazy techniques normally also update a page's FSIP entry at the time the object is deleted, rather than waiting until later. Lindsay, Mohan, and Pirahesh [LMP86] describe the lazy method used in Starburst [HCL$^+$90]. Mohan and Haderle [MH94] describe a newer lazy technique and survey other implementation techniques related to space reservation.

Lazy methods for releasing space reservations avoid the eager techniques' costly revisiting of pages at commit time, but they introduce a new problem: updating a page's FSIP entry at the time of the object deletion places a value into the FSIP entry that may mislead an object placement algorithm, making it believe that there is at least a certain amount of free space on the page, when in reality, some of that free space is currently reserved. In general, object placement algorithms such as NF(WH) and FF, which are very eager to fill free-space holes, encounter reserved space more often than lazier algorithms and therefore suffer greater performance degradation. At the other extreme, AO almost never encounters reserved space.

In the case of HY, there is a choice as to whether object deleters should put pages with reserved space into the pid cache. If they do not, creators are protected from encountering reserved space, but transactions that both delete and create objects will be prevented from reusing their own reserved space. In our implementation, a deleter will place a page into the pid cache if that page has more free space (including any reserved space) than the fullest page in the cache. However, a creator that attempts to place an object on a page in the pid cache, but fails due to reserved space, will remove that page from the cache.

4 Implementing Object Placement

Our performance study is based on an implementation of variants of five object placement algorithms in Shore. All together, these algorithms account for approximately 3000 lines of C++ code, about one-third of which are support code shared by all of the algorithms. Of the remaining 2000 lines, about 550 are specific to HY, 600 are specific to NF(WH), and the rest are specific to one of the other algorithms.

In addition to HY and NF(WH), we also tested FF, AO(1), AO(8), and Best-Fit (BF). Best-Fit always allocates an object from the fullest page that can hold it. Our implementation of BF uses a simple in-memory B^+ tree to keep track of the free space in all pages of a file[5]. The large size of this data structure makes BF impractical for use in real systems, but we include it in our experiments to provide another data point on possible disk space utilization. (Theoretical results show that the worst-case utilization of BF is the same as the worst case of FF: 1.7 times optimal [JDU+74].)

We also studied an enhanced version of FF based on the object placement algorithm used in DB2/2. This algorithm searches pages starting at the beginning of the file if the current request is smaller than the immediately preceding allocation request in the same file, but starts the search with the page used to allocate the previous object otherwise [Moh95]. The idea is that a "small" request is a signal to go back and try to fill in a hole. Preliminary experiments with this algorithm indicated that while it improves upon FF, it shared most of FF's shortcomings, and its performance did not approach that of NF(WH) or HY. We have therefore omitted detailed measurements of this variant of FF from the results in the next section.

[5]This is a transient B^+ tree; it is completely different from Shore's persistent B^+ tree implementation.

The following table summarizes the algorithms investigated in this study and the abbreviations we use to identify them.

abbreviation	*full name*
AO(n)	Append Only with pid cache size n
BF	Best-Fit
FF	First-Fit
NF(WH)	Next-Fit with Witnesses and Histograms
HY(n,u)	Hybrid with pid cache size n and target utilization u

All of our algorithms use a transient *file table* that records information about recently accessed files. When a file is first encountered, the algorithms that use information from the FSIPs scan all the FSIP entries for the file to initialize their data structures (i.e., the histograms for NF(WH) and HY, and the free-space B^+ tree for BF). Although this scan can hurt the cold start performance of the system, the information in the file table can remain cached indefinitely, even across transaction boundaries.

We used FSIP entries of 4 bits each, partitioning pages into 16 buckets. In Shore, a page contains 8 Kbytes, of which 84 bytes are overhead. For buckets 0 through 13, pages in bucket b contain between t_b and t_{b+1} bytes of free space:

b	t_b
0	0
1	64
2	128
3	256
4	512
5	1024
6	1811
7	2598
8	3385
9	4172
10	4959
11	5746
12	6533
13	7320
14	8108

Bucket 14 contains pages that hold no objects and bucket 15 contains pages not currently allocated to any file. The thresholds are spaced at roughly logarithmic intervals to limit the relative error caused by quantization.

5 Performance Results

To study the performance of our set of object placement algorithms, we constructed a set of four workloads, each of which is designed to test a specific aspect of the object placement problem. The hardware platform for our tests consisted of a Digital Equipment Corporation

DECpc XL 590 (90 Mhz Pentium processor), with 32 megabytes of memory, running Solaris 2.4. The data volume was a 2.1GB Seagate ST12400N raw drive that was entirely under the control of Shore. The log was stored on a 100MB partition of a Conner CFP1060S hard drive. To avoid interference from outside sources, the machine was removed from all networks, and all unnecessary processes were terminated. A buffer pool consisting of 1000 8KB pages was used for all tests. For the hybrid algorithm, a target space utilization of 87% (approximately 7/8) was used for all experiments.

5.1 The Uniform Create Workload

Our first workload tests creation of nearly uniform-sized objects. It is designed to mimic an application that populates a database with many small objects, such as a bulk loader or an application that creates a large persistent object graph. In the absence of deletions or large variations in object size, even AO can be expected to achieve nearly perfect packing, and the extra "cleverness" of the other algorithms should incur overhead with little benefit in disk-space performance. The purpose of this test is to measure how bad those overheads are. Object sizes were random and uniformly distributed in the range of 100–300 bytes. This range was chosen because it overlaps free-space buckets 1, 2, and 3. Before beginning each test, we formatted the data volume and created a single empty file on the volume. The test program then runs a series of transactions, each of which creates 10,000 objects and commits, until the volume is completely full.

Figure 4 shows the throughput of each algorithm for this workload as a function of the number of objects in the file. For this simple workload, all of the algorithms, with the notable exception of FF (shown in Figure 5), have excellent performance. FF, on the other hand, ran so slowly that it had to be terminated after running for 12 hours, by which time it had completed less than 5% of the test. By comparison, the other algorithms completed the test in just over two hours.

FF's poor performance is easily attributed to the large number of FSIP entries that it examines. In the small portion of the test that it was able to complete, FF looked at over 4 billion FSIP entries. NF(WH), by comparison, looked at fewer than 800,000 entries over the entire test; HY was able to avoid FSIP searches altogether.

As described above, our implementation of BF maintains free-space information about all of the pages in the file in an in-memory B^+ tree. It therefore does not use FSIPs, except during the initial creation of the tree. This approach enables it to complete the test in a reasonable amount of time. However, its memory overhead makes it impractical for use in a real system: by the end of this experiment, the B^+ tree had grown to well over one megabyte in size. By comparison, no other algorithm used more than a few hundred bytes.

As expected, all algorithms achieved excellent disk

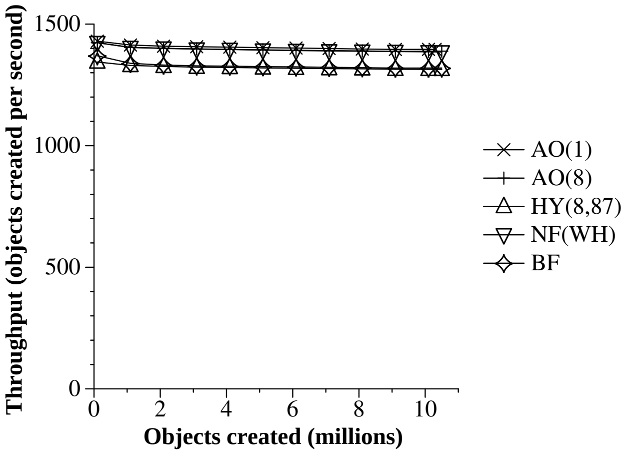


Figure 4: Uniform Create Workload: Throughput.

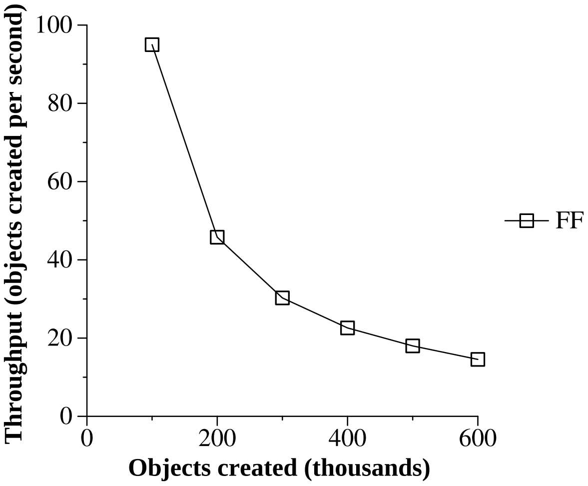


Figure 5: Uniform Create Workload: Throughput (FF only).

utilization for this simple test (see Figure 6, and note that the y-axis in this graph runs from 75% to 100%). AO(1)'s slightly poorer packing is due to its single-page pid cache. The allocation of a larger object occasionally causes AO(1) to discard the page currently in its cache, even though a smaller object might still fit on the page. Because AO(n) never again looks at a page once it has left the pid cache, this space is never used. AO(8) and HY(8,87) both avoid this problem simply by using a larger cache.

5.2 The Mixed Create Workload

Our second workload also contains only object creations, but includes a sprinkling of larger objects, such as might be found in an application that includes text objects or objects containing variable-size arrays or collections. This kind of workload also arises in database systems that cluster objects of different types together

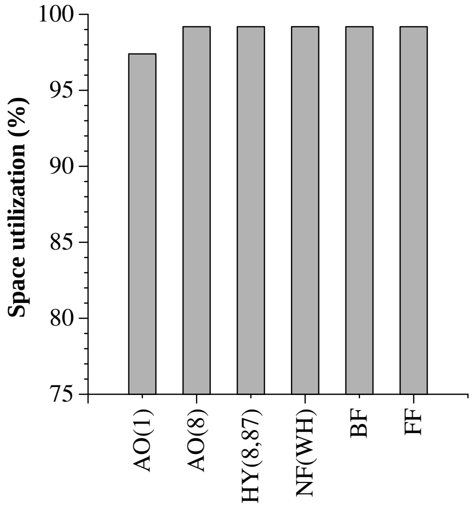


Figure 6: Uniform Create Workload: Space utilization.

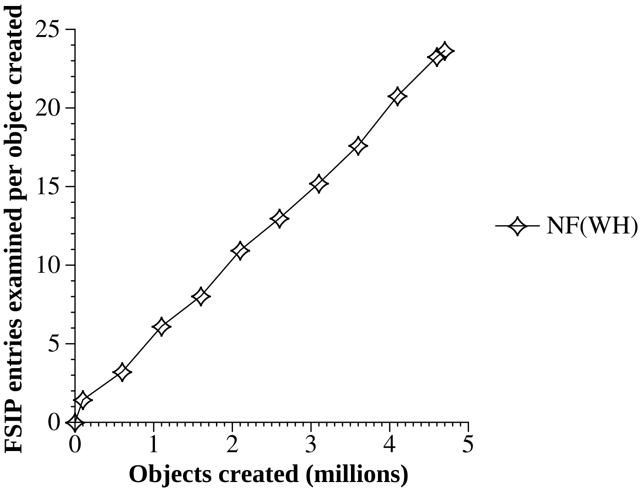


Figure 8: Mixed Create Workload: FSIP entries examined per object created.

in the same file. As in the first workload, the test program formats the data volume, creates a single empty file on the volume, and creates new objects until the volume is full. For this experiment, 95% of the new objects are 100–300 bytes in size, while the remaining 5% of the objects are 5000 bytes. This workload presents a challenge to object placement algorithms that try to maintain good space utilization: at most one of the large objects can fit on a page, yet each one leaves nearly 40% of the page unused.

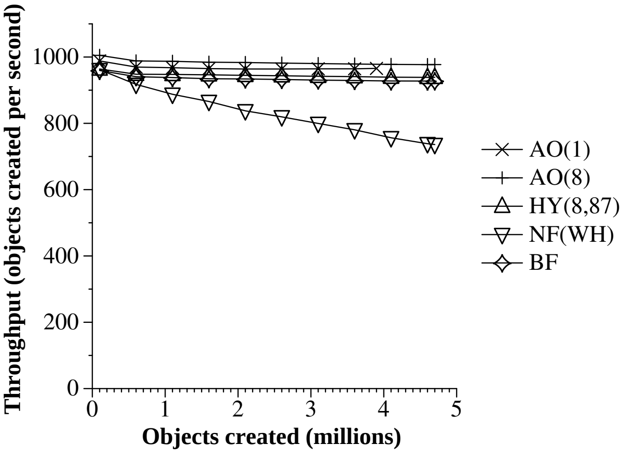


Figure 7: Mixed Create Workload: Throughput.

Figure 7 shows the throughput results for the Mixed Create workload. As with the Uniform Create workload, FF was so slow that it could not complete the test and its throughput would not show up in Figure 7. A comparison of Figures 7 and 4 shows similar performance trends, although all of the algorithms have lower throughput for the mixed workload because the larger objects fill more pages, and therefore increase the number of disk I/Os per created object.

NF(WH)'s throughput drops steadily throughput this experiment. The root of the problem for NF(WH) lies in the use of just a single witness per free-space bucket under a workload that frequently creates multiple pages in the same bucket. For example, whenever a 5000-byte object is allocated on a brand-new page, the system will place the page into bucket seven. If two of these large objects are created in a row, the algorithm is faced with two pages in bucket seven, only one of which can serve as a witness for the bucket. The other page will be dropped from the witnesses, and the only way to retrieve it is via an exhaustive search over the file's FSIPs entries. As the size of the file grows over the course of the experiment, these searches get longer and longer. This effect can clearly be seen in Figure 8, which shows the number of FSIP entries examined by NF(WH) per object created.

Figure 9 shows the space utilization results for the Mixed Create Workload (note again that the y-axis runs from 75–100%). Except for AO(1), all of the algorithms have excellent space utilization. As in the Uniform Create workload, AO(1)'s poorer packing is due to the variation in object sizes. However, in the current workload, the effect has become much more pronounced, due to the inclusion of much larger objects.

FF, NF(WH), and BF have the best space utilization in this test, although their improvement over AO(8) and HY(8,87) is less than 1%. In each case, this small improvement in packing is achieved at great cost: FF (not shown in Figure 7) and NF(WH) have poor runtime performance, while BF uses an excessive amount of memory (again, BF's B^+ tree grows to well over one megabyte). Incidentally, AO(1)'s poorer packing explains why that algorithm's curve ends earlier than the others in Figure 7—it fills the disk after creating fewer objects.

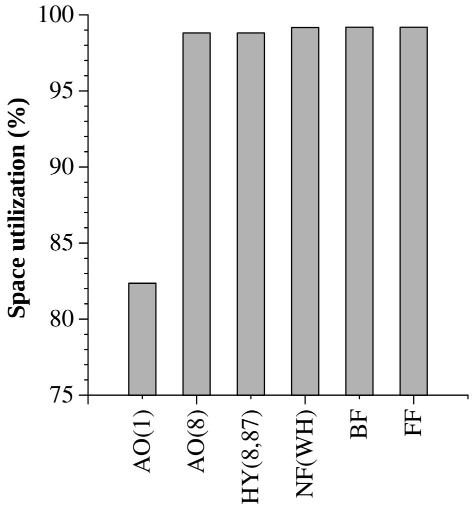


Figure 9: Mixed Create Workload: Space utilization.

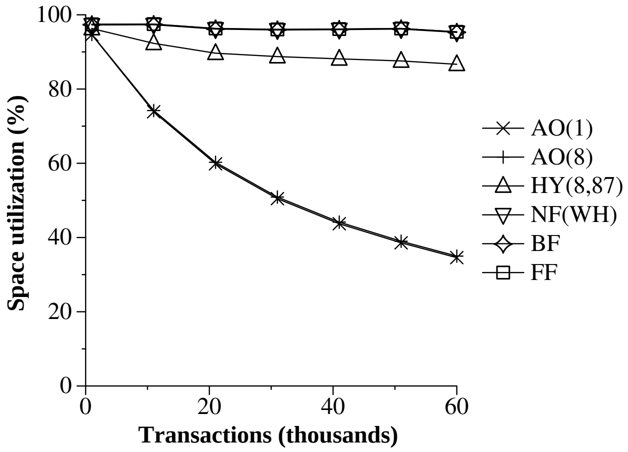


Figure 11: Create-Delete Workload: Space utilization.

5.3 The Create-Delete Workload

Our third workload consists of a mixture of creations and deletions of nearly homogeneous small objects. Each experiment ran 60,000 transactions in sequence against a file that was initially populated with 200,000 objects of 100–300 bytes each (totaling approximately 40MB). In all cases, the initial file was created with AO(8) using the homogeneous workload, so that the objects were very tightly packed (90% utilization). Each transaction was an object creator or an object deleter with independent and equal probability, with the number of objects created or deleted uniformly distributed over the range 8–16. Since the creation and deletion rates were equal, the number of objects in the file remained roughly constant. Objects to be deleted were chosen with equal probability from all objects in the file.

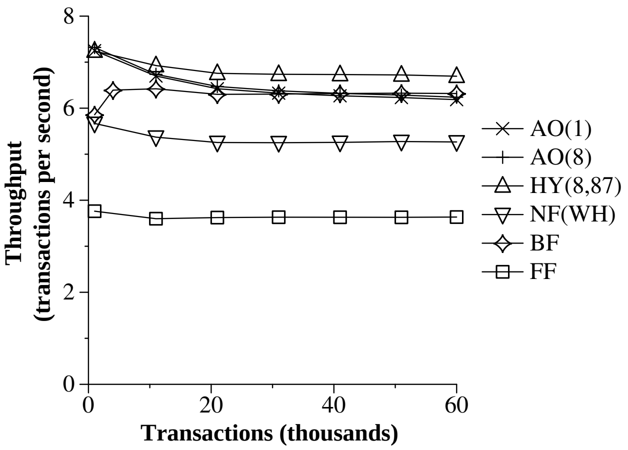


Figure 10: Create-Delete Workload: Throughput.

Figure 10 shows transaction throughput for the Create-Delete workload, and Figure 11 shows space utilization. This workload tests an algorithm's ability to fill free-space holes caused by object deletions. One would expect that AO, as the only algorithm that does not try to fill these free-space holes, would easily outperform the other algorithms in this test. Thus, the surprising result in Figure 10 is that AO's performance is matched, and even bettered, by algorithms that do fill holes.

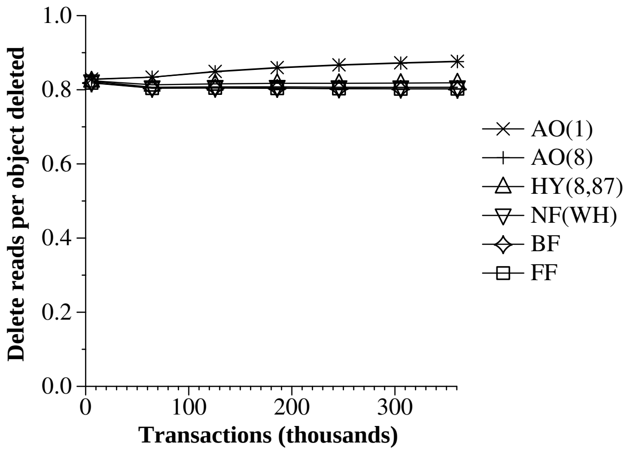


Figure 12: Create-Delete Workload: Delete Reads.

This result can be explained by examining the number of "delete reads" (pages read from disk to delete objects from them) for each algorithm. Since there is no spatial locality in deletions, deleter transactions have a very poor buffer-pool hit rate. Figure 12 shows the "deleter miss rate," the number of delete reads per object deleted. Initially, the file contains 5059 pages, and the buffer pool holds 1000 pages. Therefore, all of the deleters initially see a miss rate of approximately 80%. During the test, most of the algorithms are able to control the growth (in pages) of the file, as suggested by Figure 11, and therefore see the same miss rate throughout the test. AO, on the other hand, allows the file to grow to almost three times its original size

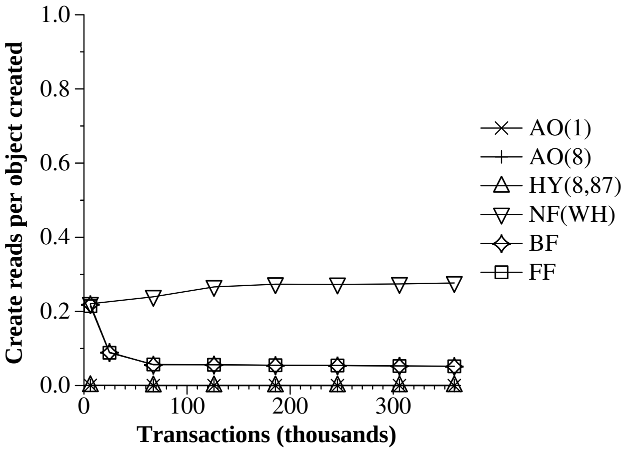


Figure 13: Create-Delete Workload: Create Reads.

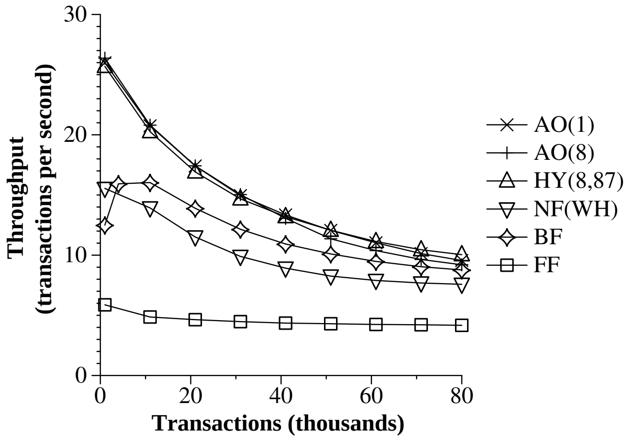


Figure 14: Concurrent Create-Delete Workload: Transaction throughput.

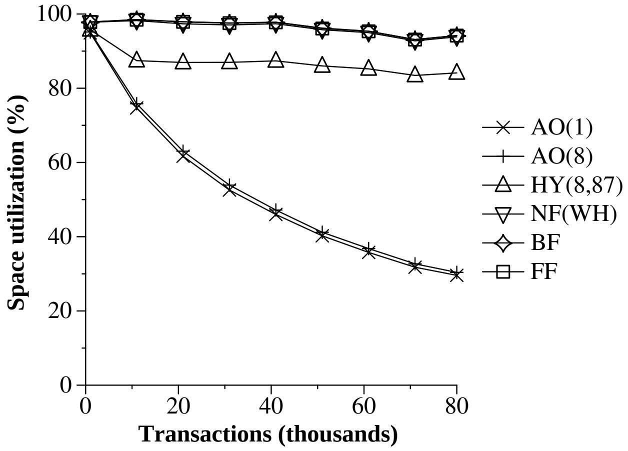


Figure 15: Concurrent Create-Delete Workload: Space utilization.

(approximately 14000 pages), and its miss rate therefore grows to about 88%. If the objects in the file were evenly distributed, we would expect to see a miss rate of 93% for the given file and buffer pool sizes. However, since AO only places new objects on new pages, newer pages tend to be fuller than older pages, and are therefore more likely to be the target of a deletion.

While AO sees an increase in delete reads, the other algorithms suffer the cost of "create reads" (existing pages read from disk to add objects to them), but as Figure 13 shows, this cost is only significant for NF(WH), which is willing to read a page from disk even if that page only has room for one more record. The other algorithms either avoid this cost by filling pages before they migrate out to disk (FF and BF), or amortize it by reading pages that can hold multiple new objects (HY).

5.4 The Concurrent Create-Delete Workload

Our final workload is a multi-threaded version of the Create-Delete workload and is designed to measure the performance of concurrent object creations and deletions, and in particular, the effects of reserved space. As with the single-threaded Create-Delete workload, we begin with a tightly-packed file containing 200,000 objects of 100–300 bytes each. We then create four threads, each of which runs the Create-Delete workload of Section 5.3.

Figures 14 and 15 show throughput and space utilization for the Concurrent Create-Delete workload. Comparing Figures 10 and 14, we see that the use of four threads initially yields nearly four times the throughput of the single-threaded versions, but performance drops off as the experiment proceeds. The primary reason for the loss in efficiency is buffer thrashing due to the random object deletions. Nonetheless, the speedup is still between 1.25 and 1.5 for most algorithms by the end of the experiment, and the relative performance of the algorithms for this workload, in both throughput and space utilization, is similar to the previous workload.

A factor not present in the single-threaded workload is reserved space. Despite the shortness of the transactions in this workload, some of the algorithms encounter many pages with reserved space. Recall from Section 3 that under a lazy space reservation scheme, a page's FSIP entry indicates the total amount of free space on the page, including its reserved space. Creator transactions may therefore be misled into trying to place new objects on the page, only to discover that some of the free space is currently reserved. We call these occurrences "reserved-space collisions."

Figure 16 shows the number of reserved-space collisions per object created. The high number of collisions for FF and BF is a result of those algorithms' eagerness to fill small free-space holes. Because these algorithms try to keep all pages 100% full at all times, and because this workload rarely deletes more than one record from a given page in any one transaction, pockets of reserved space usually appear on pages that have little or no other free space. HY(n,u) avoids this problem because it never considers a page for allocation unless its occu-

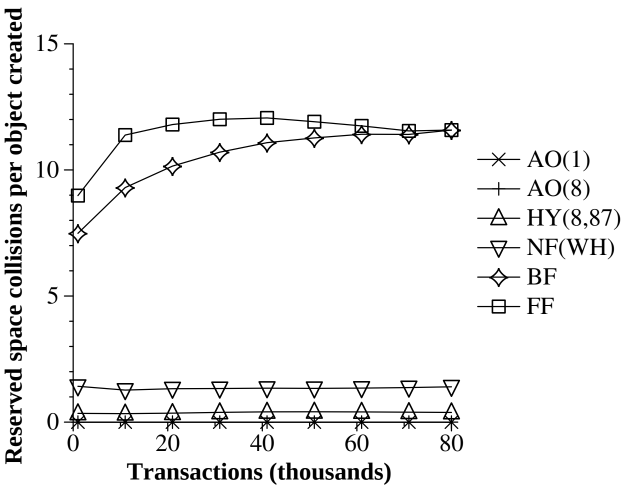


Figure 16: Concurrent Create-Delete Workload: Reserved-space collisions.

pancy is less than u (unless it is in the pid cache). It is therefore much less prone to reserved-space collisions.

6 Conclusions

Solutions to the object placement problem must take into account special issues that arise in database systems, such as locking and logging, I/O traffic, and reserved space. We have surveyed solutions to the object placement problem with respect to these issues and presented the results of a performance study that examines how each of these solutions performs under various workloads. Our performance study is based on an implementation of several object placement algorithms in Shore, an object-oriented database system under development at the University of Wisconsin—Madison.

The results of our performance study indicate that simple algorithms, such as First-Fit (FF), Append Only (AO), and Best-Fit (BF) have serious deficiencies in functionality, performance, or memory usage: FF's n^2 performance makes it impractical for use in large files, while AO's inability to reuse file space presents difficulties for maintaining good space utilization and file organization. BF performed well in almost all of our tests, but uses excessive amounts of main memory.

We showed how three simple improvements to FF lead to an algorithm that avoids its n^2 behavior, but the resulting algorithm, Next-Fit with Witnesses and Histograms (NF(WH)), still falls significantly short of AO's throughput when deletions are mixed with creations. However, HY(n,u), which is a hybrid of AO and NF, exhibits excellent performance in all of our tests. At the heart of the hybrid algorithm is a tradeoff between space utilization and I/O performance. By not trying to achieve 100% space utilization at all times, the hybrid algorithm is able to provide better I/O performance, as well as better resistance to reserved space, than any of the other algorithms in our study. In addition to its excellent performance, the hybrid algorithm is attractive because of its simplicity and low memory consumption.

7 Acknowledgments

The authors wish to express their gratitude to Michael Zwilling, who provided valuable insight and assistance during every phase of this project, from the design of the algorithms to their implementation, debugging, and performance tuning. We would also like to thank those members of the industrial database community who provided us with details of the inner workings of various commercial systems: Fred Carter, Jim Gray, Bruce Lindsay, C. Mohan, Franco Putzolu, and Peter Spiro. Eric Bach pointed out the exercise in Knuth that contained the closed form for the recurrence in Appendix A. Finally, we would like to thank Hector Garcia-Molina and Jeffrey Naughton for providing the hardware platform used for our tests, and Janet Wiener for her careful reading of the paper.

References

[CDF+94] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C.K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring Up Persistent Applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 383–394, May 1994.

[CDG+90] M. J. Carey, D. J. DeWitt, G. Graefe, D. M. Haight, J. E. Richardson, D. T. Schuh, E. J. Shekita, and S. L. Vandenberg. The EXODUS Extensible DBMS Project: An Overview. In Stanley B. Zdonik and David Maier, editors, *Readings in Object-Oriented Database Systems*, pages 474–499. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[HCL+90] L. M. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. J. Carey, and E. Shekita. Starburst Mid-Flight: As the Dust Clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160, March 1990.

[HE95] L. Hobbs and K. England. *Rdb: A Comprehensive Guide*. Digital Press, second edition, 1995.

[IBM89] IBM Corp. *IBM Database 2 Version 2 Diagnosis Guide and Reference*, second edition, September 1989.

[JDU+74] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham. Worst-Case Performance Bounds for Simple One-Dimensional Packing Algorithms. *SIAM Journal of Computing*, 3(4), December 1974.

[JS93] T. Johnson and D. Shasha. B-Trees with Inserts and Deletes: Why Free-at-Empty Is Better Than

Merge-at-Half. *Journal of Computer and System Sciences*, 47:45–76, 1993.

[Kir93] J. Kirkwood. *Sybase Architecture and Administration.* Ellis Horwood, 1993.

[Knu69] Donald E. Knuth. *The Art of Computer Programming. Volume 1: Fundamental Algorithms.* Addison-Wesley, Reading, MA, 1969.

[LMP86] Bruce G. Lindsay, C. Mohan, and M. Hamid Pirahesh. Method for Reserving Space Needed for "Rollback" Actions. *IBM Technical Disclosure Bulletin*, 29(6):2743–2746, November 1986.

[MH94] C. Mohan and D. Haderle. Algorithms for Flexible Space Management in Transaction Systems Supporting Fine-Granularity Locking. In *Proceedings of the International Conference on Extending Database Technology*, 1994.

[Moh95] C. Mohan, October 1995. Private communication.

[Stü95] G. Stürner. *Oracle 7: A User's and Developer's Guide.* International Thompson Pushlishing Co., 1995.

[Tan87] Tandem Database Group. NonStop SQL: A Distributed High Performance, High Availability Implementation of SQL. In *High Performance Transaction Systems.* Spring Verlag, 1987.

A Asymptotic Utilization of Append-Only

As noted in Section 3, AO leads to poor disk utilization under workloads that combine creations and deletions. To derive an analytic estimate of the "bloating" caused by AO, we make some simplifying assumptions:

- All objects are the same size, and precisely k objects fit on a page.
- Objects are created and destroyed at the same rate, and in "batches": A page filled with k objects is added to the file, and then k objects are deleted.
- Objects to be deleted are selected randomly and uniformly from among all objects currently in the file.
- The file contains kN objects, where $N \gg k$.

Since the file would fit on N pages if it were tightly packed, the asymptotic space untilization is N/P, where P is the expected number of non-empty pages (completely empty pages are reclaimed by the system).

During each deletion phase, k objects are deleted with equal probability out of the total of kN objects in the file, so each object is deleted with probability $1/N$ and survives with probability $p = 1 - 1/N$. Consider a snapshot of the file just after a new page has been added. Number pages by age so that page 0 is the full page just added, page 1 is the page added in the previous cycle, etc. Each surviving object on page i has survived i rounds of deletion, so an object on page i is still in the file with probability p^i. Thus the probability that all k objects on page i have been deleted is $(1-p^i)^k$, and the probability that page i is non-empty is

$$\begin{aligned} E_i &= 1-(1-p^i)^k \\ &= 1-\sum_{j=0}^{k}(-p^i)^j\binom{k}{j} \\ &= \sum_{j=1}^{k}(-1)^{j+1}p^{ij}\binom{k}{j}. \end{aligned}$$

The expected number of non-empty pages in the file is the sum of these probabilities.

$$\begin{aligned} P &= \sum_{i=0}^{\infty} E_i \\ &= \sum_{i=0}^{\infty}\sum_{j=1}^{k}(-1)^{j+1}p^{ij}\binom{k}{j} \\ &= \sum_{j=1}^{k}(-1)^{j+1}\binom{k}{j}\sum_{i=0}^{\infty}p^{ij} \\ &= \sum_{j=1}^{k}(-1)^{j+1}\binom{k}{j}\frac{1}{1-p^j}. \end{aligned}$$

Since $p = 1 - 1/N$,

$$\begin{aligned} 1-p^j &= 1-\sum_{i=0}^{j}\binom{j}{i}(-\frac{1}{N})^i \\ &= \frac{j}{N}+O(\frac{j}{N^2}) \\ &\approx \frac{j}{N} \end{aligned}$$

for $N \gg j$. Thus

$$\begin{aligned} P &\approx N\sum_{i=1}^{k}(-1)^{i+1}\frac{1}{i}\binom{k}{i} \\ &= NH_k, \end{aligned}$$

where H_k is the k^{th} *harmonic number*,

$$H_k = \sum_{i=1}^{k}\frac{1}{i}$$

(see [Knu69, Section 1.2.7, Exercise 13]). Thus

$$Utilization = N/P \approx \frac{1}{H_k}.$$