Analyzing and Improving Table Space Allocation

Shanchan Wu1 Yefim Shuf2 Hong Min2 Hubertus Franke2 Bala Iyer3
Frances H Villafuerte3

1 Department of Computer Science, University of Maryland, College Park, MD, USA
Email: wsc@cs.umd.edu
2 IBM T.J. Watson Research Center, Yorktown Heights, NY, USA
Email: {yefim,hongmin,frankeh}@us.ibm.com
3 IBM Silicon Valley Lab, San Jose, CA, USA
Email: {balaiyer,francesv,jwatts}@us.ibm.com

Abstract

Space allocation is a fundamental operation performed by a database management system (DBMS) when it inserts a record into a table. A good space allocation algorithm quickly locates and reserves enough space for a record, places it closer to its related records, and utilizes the available space. Satisfying these conflicting requirements is challenging and trade-offs are carefully balanced by well-chosen heuristics. As a DBMS evolves over time, especially a commercial DBMS, its space allocation algorithm gets more sophisticated and complex and relies on many heuristics. Technological changes, new applications, and greater data volumes render many legacy heuristics ineffective. These factors hinder understanding of space allocation behavior under many workload conditions and make it difficult to enhance the algorithm without causing performance regressions for some of the workloads.

To facilitate research and study the performance of a table space allocation algorithm of a modern DBMS in real-world workload scenarios, we build an extensible simulation framework. We analyze algorithm behavior and make surprising observations. We use the findings to further improve the existing algorithm by proposing algorithm enhancements and showing their benefits with respect to key performance metrics. In conclusion, the proposed framework has been effective in research to understand the performance, improve the space allocation algorithms, and to guide the developers of a commercial DBMS.

1 Introduction

For a database management system (DBMS), the ability to insert a record quickly and efficiently is critical. To maximize disk space utilization while achieving high performance, database designers use various strategies for storing records. As improvements in processor speed continue to outpace improvements in disk access time, I/O is increasingly a major bottleneck in systems and especially in large DBMS (Hsu et al. 2001). An efficient space allocation strategy that minimizes I/O frequency is essential. One example is to maintain data clustering so that disk seeks can be reduced for sequential reads. However, a mix of inserts and deletes can easily cause disk space fragmentation.

Designing a table space allocation strategy that performs well for various workloads is challenging. The tradeoffs like “add more empty pages to a table space” or “fully utilize the existing space” are not easily explored via the back of an envelope analysis. Another requirement for a space allocation algorithm is to have fewer contentions in a highly concurrent transaction processing environment. We are not aware of any framework for space allocation algorithms research.

In this paper, we study a representative table space allocation algorithm of a commercial DBMS and its variations. To compare different space allocation strategies, we build a simulation tool. The tool is used to quantitatively answer “what if” questions that arise during a space allocation strategy design and identify the strengths and weaknesses of the strategy. The tool is also used to pinpoint when record insert performance degrades. We use the tool to analyze the algorithm under various conditions representing real workload scenarios and to evaluate our algorithm enhancements. It can be used to evaluate the implication of using Solid State Disks (SSDs) (Agrawal et al. 2008) whose performance characteristics are different than Hard Disks (HDs). Simulating various input patterns and their effect on the insertion algorithm is a complicated modeling task and the tool that helps accomplishing it has a considerable practical value.

Insert performance is one of the most challenging issues in real-life usage of DBMS. Ability to complete new data imports in a given time frame and to do so in a space usage efficient manner is a key characteristic of a well performing DBMS. Our work addresses key challenges in solving this important problem. This paper makes the following contributions:

- It presents an extensible framework for simulating space allocation algorithms and evaluating them with respect to various performance metrics on multi-threaded workloads. The framework is used as a testbed to explore ideas for improving space allocation algorithms and gain insights into how such algorithms behave in real-world workload scenarios.
- It presents observations with practical performance implications. We find that when record insertion is guided by a clustered index, a random record sequence can be inserted faster if it is pre-sorted in the clustered index key order. We show that pre-sorting records is a way to improve the cluster ratio. We also show that providing each thread with a distinct starting point when searching for space can reduce contentions.
- It proposes space allocation algorithm enhancements and quantifies benefits. We show one enhance-
The rest of the paper is organized as follows: in section 2, we present related work; in section 3, we describe the organization of a table space and a table space allocation algorithm of a modern DBMS; in section 4, we describe our simulation framework; we use the framework to exercise several workload scenarios to study the space allocation algorithm in section 5; in section 6, we propose algorithm enhancements and experimentally show their benefits; in section 7, we discuss how the performance metrics collected by our framework can be applied to analyze the actual cost of a record insert in a real system; we summarize in section 8.

2 Related Work

Research on table space allocation algorithms, while being important to the database community, has seldom been presented in the literature. Although there has been work on space management in 1996 (McAuliffe et al. 1996), in spite of technological changes, emergence of new applications, and demands for rapid loading of high volumes of data, the problem of space allocation has not been given much attention since (McAuliffe et al. 1996) and must be revisited. McAuliffe et al. (McAuliffe et al. 1996) studied object placement algorithms, their storage utilization and allocation performance with regard to clustering. Their work focused on free space management in heap files. They noted that many object placement algorithms have serious performance deficiencies, including excessive CPU or memory overhead, I/O traffic, or poor disk utilization. Compared to (McAuliffe et al. 1996), we focus on multi-threaded workloads. We use a more comprehensive set of performance parameters for analyzing performance costs from two orthogonal aspects: one is dictated by the underlying hardware and DBMS, and one is algorithm driven.

Our goal is to improve DBMS performance by reducing the CPU cost of allocating space for a record and creating a better data layout reducing I/O delays. There has been work on improving I/O performance from different angles: disk access optimizations, prefetching, architectures for storing large volumes of data, and layout optimizations (Hsu et al. 2005). Disk drive models were studied (Ruemmler & Wilkes 1994) and I/O simulation tools were developed. Sorting of RIDs was used to reduce I/O for bulk deletes (Gärtner et al. 2001). To our knowledge, we are the first to build a research tool for studying the effects of input patterns on the insertion algorithm behavior.

Data prefetching was shown to reduce synchronous I/O operations. Hsu et al. (Hsu et al. 2002) examined the logical I/O reference behavior of the peak production database workloads from ten of the worlds largest corporations and analyzed factors that affect how these workloads respond to different techniques for caching, prefetching, and write buffering. Wilson et al. (Wilson et al. 1995) discussed the design of dynamic memory allocators. Due to the popularity of Flash memory as a data storage medium, some began designing Flash-based DBMS (Lee & Moon 2007). Our tool can be used to explore algorithms for systems with Flash-based storage.

There has been work on improving DBMS performance by reducing resource access contentions, mostly at the transaction level. To ensure data integrity, locking schemes were proposed (Felber & Reiter 2002, Silberschatz & Kedem 1980). Several concurrency control techniques were investigated (Felber & Reiter 2002, Reuter 1982, Gawlick & Kinkade 1985). The contentions addressed in the literature are contentions on accesses to records and indexes of databases, and not contentions during table space allocation, which is this papers focus. The methods to improve concurrency and space utilization by space reservation and tracking are described in (Mohan & Haderle 1994).

3 Table Space And Table Space Allocation Algorithm

Figure 1 shows table spaces and tables in a database. Since table spaces reside in database partition groups, the table space selected to hold a table defines how the data for the table is distributed across database partitions. A single table space can span several containers. Containers define physical storage for a table space. A container can be a file system directory, a file with a preset size, or a raw device such as an unformatted disk, a disk partition, or a logical volume. Multiple containers from one or more table spaces can be created on the same physical disk. For better performance, each container can reside on a different disk.

Figure 2 shows the structure of a table space. A table space contains multiple segments. Each segment contains a number of pages. The typical page types are: a header page, a space map page (SMAP), a compression dictionary page (if data is compressed), and a data page. The header page describes the table space. A space map page identifies data pages with enough free space for new records. Each space map page contains a specific range of pages. A SMAP uses an indicator (several bits) for each data page to indicate the level of free space on that page. When a table space does not have enough available space to accommodate a new record, an extension operation may occur, in which case, new space is allocated at the end of a table space.

There is an index structure (an index tree) to keep track of the order and locations of records in a table. The index is updated once a record is inserted into a table. The number to the left of each page in Figure 2 indicates page...
number of that page in a table space. The location of a record is defined by page number of the data page where the record is located and the offset of the starting point of the record in the data page.

The table space allocation algorithm operates as follows:

1. For a new record to be inserted, look up the record’s key in an index tree of the table to find a desired location (i.e., a candidate data page) for the record to be placed at. If the record’s key value does not exist in the index, the nearest key value in the index is used for identifying the candidate page.
2. If the placement in step 1 fails, find space within the same segment where the candidate data page is located.
3. If failed in step 2, search from the first segment that has free space forward to the last segment covered by the same space map page. Note: A reference to the first segment that has free space is updated when necessary.
4. If failed in step 3, go to the last segment of the table space. Search from the first page of the last segment to the last page of the last segment.
5. If failed in step 4 and if allocating new space will not cause an extension, then allocate a new page.
6. If failed in step 4 and if allocating new space will cause an extension, then do an exhaustive search from the first segment that has free space to the end of the table space. A reference to the first segment that has free space is updated when necessary.
7. If failed in step 6, allocate a new page with an extension.

An ideal table space allocation algorithm should be able to quickly find enough free space for a record, waste no space, and maintain data clustering which is measured by “cluster ratio” to indicate how closely the records' physical placement ordering matches the logical ordering of certain keys. To maintain a high cluster ratio, the algorithm tries to place a record in the order of following vicinities if possible: index directed candidate page, or within the same segment of candidate page, or in a segment covered by the same space map page.

A good starting point to analyze performance of a space search algorithm is to identify major factors influencing its performance. The major factors for record insertions are I/O operations and contentions. The I/O operations include reading space map pages (SMAP) and data pages from a disk storage system. Since the size of a buffer pool is finite and is usually much smaller than the size of a corresponding table, a larger number of (random) page fetching operations usually results in more misses in a buffer pool and more I/O operations. To reduce I/O operations, each thread maintains in memory one recently used data page and one SMAP page. When a thread accesses a data page or a SMAP that is already in memory, an I/O operation is avoided. Once a thread fetches and successfully uses a new data page or a new SMAP page, it maintains them in memory to speed up access.

In a SMAP, the number of bits representing how much space is free on each data page is small (for space efficiency) and the information on free space is not frequently updated (to increase concurrency). So even if a SMAP indicates that a data page may have enough free space, when the data page is fetched, there might not be enough free space for a new record.

The following are some performance metrics used to evaluate the algorithm in terms of CPU and I/O activities and contentions:

- The percentage of records that are inserted into an initial candidate data page identified via an index tree lookup.
- The number of pages checked in a SMAP before a successful insert.
- The number of data pages and SMAP pages that are fetched before finding free space to insert a record.
- The number of page latches or locks that are contention with during a space search.

Our simulation tool tracks these and other performance metrics.

4 The Simulation Framework

In this section, we describe the architecture of the simulation framework / tool. To reduce memory footprint, the tool does not store the content of records, but only keys and record sizes. The tool inputs record sequences and outputs statistical results. The input sequences are generated by an input workload generator (or by using the instrumentation features of a DBMS). It generates various input streams with configurable attributes.

Each thread in this multi-threaded tool operates on a separate input sequence. Many parameters such as time to insert a record, wait time for a latch, the properties of a table space, are configurable. An index is implemented as a B-tree. To find a candidate page from an index, we look for a record with the same key, or the nearest higher key, or the nearest lower key. The index is updated once a record is inserted. We implemented a Lock-Manager to simulate concurrent access issues and contentions.

Figure 3 describes the structure of the simulation framework.

The tool is used as follows. After the input generator generates desired sequences of input records, they are saved. Then the simulator is started to concurrently process input sequences with multiple threads and collect statistical data for performance metrics. At the end of a simulation, the state of a table space can be saved on a disk. The next time, if we want to continue from a particular state of a table space, we can configure the tool to load the saved state of a table space and process more input streams. This lets us to use identical initial states for different experiments.

5 Experimental Analysis Of The Algorithm

In this section, we analyze the characteristics and performance of the algorithm with different workloads. Intuitively, it is faster to find space for a new record when a table space has a substantial amount of free space than when a table space is almost filled. To study how the performance metrics of the algorithm change when the state of a table space changes, we simulate a real banking application example where a table space grows from being sparse to being full. We find a particular state when performance metrics deteriorate sharply and propose techniques to mitigate this performance problem. We also investigate...
how the characteristics of input record sequences affect algorithm performance. The study of this scenario will help us make decisions on whether or how to preprocess input sequences before inserting records. For this purpose, we compare and analyze performance metrics for the ordered record sequences and unordered record sequences.

5.1 A Banking Application Example

Let us consider a database design for a representative banking application workload. The data is organized by a clustering index on a data attribute such as account number. There is a fair amount of free space left on each page. When there is account activity, the banking application closes the old account record by updating its ending timestamp, and inserts a new record for the same account, preferably near the account record that has just been closed.

For this type of a database and application design, it is expected that the table space, which initially has a lot of free space, will grow full over time as more transactions are processed and more records are inserted. The application expects table space re-organizations and extensions to be performed periodically to space out records in the table space. However, when some accounts are more active than others, free space around these accounts becomes scarce and new records corresponding to these accounts are placed elsewhere. This prolongs a space search process for those records. The quality of data clustering degrades.

We simulate this workload and investigate performance issues when the account access pattern is skewed. The table space is pre-populated with records corresponding to all bank accounts, with every page having some free space. After that, concurrent streams periodically insert records into the table space. When there is sufficient free space, a record can be inserted quickly. When the available space decreases, it takes longer to place a new record.

5.1.1 Experimental Analysis

The initial state of a table space is created by inserting a sequence of records with non-duplicated sequential keys corresponding to all account numbers. Each page is left with 80% of free space (20% of space is occupied by account data). We use the 80/20 rule and designate 20% of accounts as very active accounts that generate 80% of account activities. The remaining 80% moderately active accounts generate 20% of activities. We create input sequences representing this skewed account activity pattern. Each sequence consists of sub-sequences representing daily bank activities. Each daily sub-sequence covers 2.5% of distinct accounts and 365 sub-sequences (365 days) make a long sequence of records. The records in each sub-sequence can fill 0.5% of space of the initial table space.

The performance metrics of the experiment are shown in Figure 4 through 8. In Figure 4 through 7, x-axis value corresponds to a particular sub-sequence (a particular day). For example, x=50 is the 50th subsequence of the 50th day. Figure 4 shows where the records are inserted into (candidate pages, other pages in the same segment, corresponding to algorithm steps described in Section 5) over time, represented by a percentage of records in a subsequence. For example, when x = 1 (the first day), almost 100% of records are placed on candidate pages. When x is around 160, the initial table space is almost full. Figure 5 shows the average number of pages checked in SMAP per record insertion. Figures 6 and 7 show the average number of fetching operations per record insertion. Figure 8 shows that the cluster ratio is worse in the final state compared to the initial state.

The experimental results show that during the transition of a table space from the almost full state to the full state, the performance metrics are significantly worse. A few sparsely located remaining free space slots in a table space cause a long exhaustive search. After the transition state, the few sparsely located remaining free space slots are filled and the indicator to the first segment that contains available space is shifted to the location near the end of a table space, which reduces the cost of an exhaustive search. A table space may also be transitioning from the full state to the almost full state because of deletion operations. The deletion operations can create sparsely located free slots and the algorithm will try to find those empty spots when new records are inserted. This can lead to a long exhaustive search.

5.1.2 Mitigating Performance Problems

To mitigate performance problems during the transition state, we evaluate ideas that use heuristics to avoid an exhaustive search. One is to avoid an exhaustive search by anticipating and detecting symptoms of the pre-transition state proactively. Another is to stop performing an exhaustive search if a table space is almost full.

Avoiding an Exhaustive Search by Anticipating and Detecting Symptoms of the Pre-Transition State Proactively:

For this banking application workload example, if it can be detected by inspecting performance metrics (using a performance reporting facility) that a transition state will occur soon, then we can reorganize a table space proactively before more records are inserted. Reorganizing involves sorting all data in a table space, repopulating the table space with added space, and leaving a reasonably high percentage of free space on each page. For example, suppose the detected time is day 150 (i.e., 10 days before...
a table space is full). Once a table space is reorganized, we continue inserting the remaining data, i.e., data of day 151 through day 365. The results are shown in Figure 9 through Figure 13.

A comparison of Figure 4 to 8 with Figure 9 to 13 indicates that performance metrics improved significantly (after reorganizing a table space shortly before the transition state). The cluster ratio in the final state is better – it improved over the one without proactive reorganization. Of course, reorganization comes with its own cost in terms of time and space. So the cost/benefit of reorganization and its impact on data insert performance and query performance need to be considered and balanced.

**Avoiding an Exhaustive Search of an Almost Full Table Space:**

There is an alternative to the previously proposed idea of performing reorganization in the pre-transition state. During the insertion, after detecting that a table space is almost full and anticipating that allocating new space will cause an extension, the algorithm can be changed to skip an exhaustive search, and directly allocate a new page with an extension. When a table space is almost full, the probability that a new record will be inserted into an initial candidate page is low. This is one of the heuristics we can use to detect whether a table space is almost full. We set a threshold of 30% to perform our experiment, i.e., a table space is almost full if no greater than 30% of records in a particular time window (e.g., one day) are inserted into candidate pages. The results are in Figure 14 through Figure 18.

Comparing the sizes of the final table space (after all records were inserted) in the original algorithm and in the modified algorithm, we notice that the difference in table space sizes is very small, 0.99989 : 1. This means that applying this scheme will not cause much more space to be consumed. There are no pulses in Figures 15 through 17. In contrast, in the original algorithm, the pulses appear in Figures 5 through 7. These pulses indicate that substantially more work needs to be done to find free space during the corresponding time period. Other than the pulses, the values of other parts of the curves are similar. So the scheme improves performance, with respect to performance metrics, during the transition state of a table space while maintaining performance metrics during other states of a table space.

**5.1.3 Discussions**

The behavior anticipated by an application is to insert data into the pages dictated by an index, the “candidate pages”. When the percentage of candidate page placements becomes very small, extensions and re-organizations should be performed to add more disk space, re-cluster records, and space out records in a table space. Failing to anticipate this transition early enough, as shown, could lead to a big performance degradation which is followed by a steady state of sub-optimal record insert performance.

To mitigate the problem using approaches proposed in section 5.1.2, we can collect performance metrics during the insertion process to predict an upcoming transition state. Several statistical events can be used to indicate the approach of a transition state: (1) the percentage of records that can be inserted into candidate pages decreases quickly, (2) the percentage of records that are inserted into other pages in the same segment where the candidate page is located increases and then decreases, (3) the percentage of records inserted into other segments covered by the same SMAP increases and then decreases, and then (4) the percentage of records that are inserted during an exhaust-
tive search increases sharply. When an approaching transition state is predicted by these indicators, corresponding actions can be taken such as to advise a database administrator to perform a table space re-organization or start an automatic online table space re-organization.

The analysis of the algorithm suggests that in addition to the I/O cost of space search, the CPU cost can be a factor. We find there can be a noticeable CPU cost associated with scanning SMAP pages which are likely cached in a buffer pool. In the algorithm we examined, a performance bottleneck associated with a space search is largely related to the number of pages visited.

5.2 Ordered vs. Random Sequences

To reduce a time window to load data into databases, we investigate whether “massaging” data prior to loading can reduce the load time. One way to pre-process data is to sort it. In this section, we investigate whether we should order records by key values before loading.

To answer this question, we first analyze the characteristics of indexing. When using an index, the index is consulted before an insertion and then updated after the insertion. At the beginning, when both a table space and an index are empty, the first record is inserted into the first page in a table space and the index is updated. When the page for the record pointed by the index is not available (due to insufficient space or held latches), a table space search algorithm is invoked. Over time, the table space grows gradually, with space near the beginning being slowly filled and leaving most of available space near the end of the table space.

Due to the characteristics of the cluster indexing, different insertion behaviors are observed with ordered and random input sequences. Since a record always gets the candidate page number of the nearest key in the index, when inserting a record of ordered sequences, the algorithm will likely first try a page (a candidate page) near the end of a table space where the pages are likely to have free space. When inserting a record of random sequences, a candidate page pointed by an index can potentially be anywhere in a table space. So we hypothesize that the algorithm finds free space faster for ordered sequences. We validate our hypothesis.

5.2.1 Experimental Analysis

We generate input sequences consisting of random permutations of distinct keys. The number of concurrently processed input sequences (N) is varied in our experiments. Each sequence in an experiment is of the same length. Each thread processes a different input sequence. We compared the results of two different types of inputs: (1) all input sequences are ordered before they are inserted; (2) all input sequences are in a random key order. The simulation results for N = 10, 25, 40 and 55 are as follows.

Figures 19 and 20 show performance metrics on where the records are inserted in a table space. It can be seen that when input records are ordered (vs. random), they are more likely to be placed on candidate pages and the algorithm is less likely to search for space in the last segment in a table space. Overall, more records in ordered sequences are placed successfully during the first three steps of a space allocation flow (described in Section 3) than in random sequence. This is an indication that the algorithm performs better on ordered input records.

Figures 21 through 24 show the comparison of the average number of page latch hits, the average number of fetching data page operations, and the average number of fetching SMAP operations, with two different types of inputs: ordered sequences and random sequences. From Figure 21 we can see that the average numbers of page latch hits per record insertion in ordered sequences and in random sequences are similar when N=10 and 20. When N increases to 40 and 55, the average numbers of page latch hits per record insertion in ordered sequences are less than those in random sequences. Figure 22 through 24 show that the average number of pages checked in the SMAP, the average number of fetching data page operations, and the average number of fetching SMAP operations per record insertion with ordered sequences are all significantly smaller (i.e., better) than those with random sequences. Altogether, the data in Figures 21 through 24 further suggests that, to min-
imize the time to insert records into a table space, it is advisable to order the records before inserting. Furthermore, Figure 25 shows that the cluster ratio of a table space is better when input sequences are pre-sorted.

6 Algorithm Enhancements

In the previous section, we studied the algorithm using inputs with different characteristics. In this section, we address issues of heavy contentions (often present in multi-threaded environments) and I/O frequency. Our goal is reducing contentions and I/O operations while maintaining or improving a cluster ratio. In this section, we propose three techniques that improve the space search algorithm. We show benefits with respect to performance metrics. The three enhancements can be combined together, but for the purpose of an analysis we evaluate them separately.

6.1 Reducing Contentions

6.1.1 The Observed Problem

When multiple threads try to insert records into a table space, contentions on accesses to resources can have a significant impact on performance. A thread waiting on a resource protected by a lock will have to wait for the lock to be released before it can proceed. Modern DBMS use fine grain locking and latching to reduce contentions during record insertions and updates.

After investigating the table space allocation algorithm, we found that contentions can be frequent during the search through the last segment of a table space. The following explains the reason for heavy contentions during the search through the last segment. (a) If a table space is empty or almost empty (i.e., each data page has plenty of free space), when a page is selected by an index as a candidate page, there is a high probability that a record can be inserted into that page. Even if a record cannot be inserted into that candidate page (perhaps because other threads filled it), it is still likely that the record can be inserted into a page in the same segment or a page covered by the same SMAP. In this situation, the performance is not a significant concern. (b) However, when a table space is almost full (i.e., only a few pages have enough free space for a new record), a thread will have to search through many pages before successfully inserting a record. Eventually, if no space is found, the algorithm allocates one or more new pages at the end of a table space. Hence, the pages in the last segment are more likely to have free space than pages in other segments. Therefore, when a table space is almost full, a new record is more likely to be inserted into a page in the last segment. In the original algorithm, when searching in the last segment, all threads start from the same page and are likely to find the same page with free space at the same time. The first thread that gets the page will lock, performs space checking, and inserts into it. The other threads checking the same page will contend and wait until the page is unlocked.

6.1.2 A Proposed Enhancement

We propose the following technique to reduce contentions in the last segment. Instead of letting all threads traverse through the same sequence of pages from the same starting page in the last segment, select a random page within the last segment as a start searching page for each thread. Consequently, the first page found to be available in the last segment by different threads will tend to be different. Hence the chance of all threads contending on the same page in the last segment will be reduced.

Besides reducing contentions, we also consider reducing the number of page fetching operations by assigning a random offset number for each thread when a thread starts.

6.2 Using “Recent History Lookup List”

6.2.1 A Proposed Enhancement

To minimize the waste of space, a table space allocation algorithm uses free space in a table space as much as possible before allocating more space at the end of the table space. A SMAP (space map page) is a structure that tracks the level of available space in every data page, with each data page represented by several bits in a SMAP. To find a page...
with enough free space, each thread has to scan through all SMAP bits, including those representing full pages.

One approach to speed up searching for free space is to track only pages that have free space. However, if we build a separate global structure to remember all data pages which are not full (having free space to hold the shortest record), there might not be enough memory to hold the structure and it is preferable not to store it on a disk due to the cost of I/O operations. An alternative is to keep track of a small subset of pages that have enough free space; these few pages can be stored in main memory. A page in the small subset is re-used for free space until it is full and replaced by another page. The reused page in the subset also has better locality than a page identified by scanning SMAP. Consequently, we propose a data structure to hold a few available pages and corresponding strategies to access the structure when looking for space and updating the structure. We use RHL_LIST (Recent History Lookup List) to denote this structure as pages in the list are recently found available and used for insertion.

We design the RHL_LIST structure as follows. The RHL_LIST structure is a fixed size array. Each item in RHL_LIST contains a pointer (reference) to a data page, a pointer (reference) to a SMAP page that is relevant to this data page, and a status flag indicating the status of that item in RHL_LIST. There are three possible status flags for each item: AVAILABLE, TRASH, BUSY. The AVAILABLE status of an item indicates that the page in this item is currently occupied by any thread and there is enough free space on this page. The TRASH status indicates that the page does not have enough free space and can be replaced by another available page. The BUSY status of a page indicates that the page is currently occupied by a thread.

There are three main operations on the RHL_LIST structure: (i) to update the status of an item, (ii) to get an available page through the items in the RHL_LIST, and (iii) to insert a new item into RHL_LIST with the pointer to a new page which has enough free space and a pointer to a relevant SMAP.

When a page is obtained from the RHL_LIST, the status of the item where the page is located in the RHL_LIST is set to BUSY. After the page is processed by a thread, the status is set to AVAILABLE or TRASH depending on the available space on that page. If the available space is greater than the maximum size of the records of the table, then it will be set to AVAILABLE, otherwise, it will be set to TRASH. The status of an item may also be changed from AVAILABLE to TRASH during the get_one_available_page() operation by a thread. When a thread traverses through the RHL_LIST, even if it finds the status of an item to be AVAILABLE, it has to check the space on the page to see whether it is indeed AVAILABLE (because the available space of that page may be changed by some threads without accessing the RHL_LIST, i.e., via other parts of searching). If the available space is smaller than necessary to hold the record, the status of that item is changed to TRASH. When a record is successfully inserted into a page and after that the available space of that page is still greater than the maximum size of the records, we try to insert a new item having a pointer to that page into RHL_LIST. The insertion will not be successful if there are no TRASH pages in RHL_LIST. This operation is inexpensive because it is in-memory and no extra objects are created. As the latch time on each item in RHL_LIST is very short, the contention on RHL_LIST items is not an obvious performance concern.

The data pages (and the corresponding SMAP pages) that are referenced via RHL_LIST are more likely to be in memory. When inserting records into those pages, we do not have to fetch them from a disk and it reduces I/O operations.

6.2  Using "Recent History Lookup List"
6.2.1  A Proposed Enhancement

To minimize the waste of space, a table space allocation algorithm and showed that referring to the recent history lookup list (RHL_LIST) improved record insert efficiency. The RHL_LIST is a structure to keep a small set of recently used data pages. Attempting to insert records directly into these pages reduces time searching for free space. The RHL_LIST is a structure to keep a small set of recently used data pages. Attempting to insert records directly into these pages reduces time searching for free space.

We generate sequences of distinct key values; each sequence having the same number of records. We conduct experiments using 10, 25, 40 and 55 sequences (concurrent threads). We compare performance metrics of the original and the enhanced algorithm using the RHL_LIST. The experimental results are shown in Figures 6-1 through 6-15. Although cluster ratios do not change, other performance metrics (the number of pages checked and the number of pages fetched during a space search) improve significantly. In Figures 6-1 through 6-15, the curves for the algorithm using RHL_LIST are almost flat, while the curves for the original algorithm are growing quickly. This demonstrates that the enhanced algorithm using RHL_LIST has better scalability.

6.2.2  Experimental Results

We generate sequences of distinct key values; each sequence having the same number of records. We conduct experiments using 10, 25, 40 and 55 sequences (concurrent threads). We compare performance metrics of the original and the enhanced algorithm using the RHL_LIST. The experimental results are shown in Figures 6-1 through 6-15. Although cluster ratios do not change, other performance metrics (the number of pages checked and the number of pages fetched during a space search) improve significantly. In Figures 6-1 through 6-15, the curves for the algorithm using RHL_LIST are almost flat, while the curves for the original algorithm are growing quickly. This demonstrates that the enhanced algorithm using RHL_LIST has better scalability.

6.2.3  Discussion

Earlier in this section, we proposed an enhancement to the allocation algorithm and showed that referring to the recent history lookup list (RHL_LIST) improved record insert efficiency. The RHL_LIST is a structure to keep a small set of recently visited data pages with free space. Attempting to insert records directly into these pages reduces time searching for free space.

As an alternative to tracking data pages with free space, we can keep (in a data structure) a set of SMAP pages each of which indicates that at least some of their data pages have free space. The modified space search algorithm first checks the candidate page, then tries to search in some SMAP pages in that structure, then tries to search in the last segment, and then proceeds as in the original algorithm. The structure to keep a set of SMAP pages needs to be kept up to date. When and how to update the structure as well as a performance analysis are left for the future work.

6.3  Reducing Search for Available Space
6.3.1  A Proposed Enhancement

In the original algorithm described in section 3, when inserting a new record, before making a decision to allocate
more space at the end of a table space for a new record, a potentially long search has to be made to better utilize existing space. In a workload with variable record sizes, as a result of frequent inserts and updates, the free space is largely fragmented into small empty slots (where larger records cannot be placed). For this scenario, our intuition is that (a) for large records, we should find a way to shorten a search path and (b) for small records, we can still try to use the original search path to maximize space utilization.

We propose Conditional Append Algorithm (CAA) enhancement. (a) For a large record, we first try a candidate page; if we cannot insert the record in the candidate page, we skip both the search for pages in the same segment and the search for pages covered by the same space map page. We attempt to place the record into one of the last M pages at the end of the table space. If we cannot successfully insert the record into one of the last M pages, we allocate a new page for this record. If the new allocation needs an extension, then the exhaustive search before the extension is also bypassed. (b) For a small record, we follow the original search path, except when searching in the last segment, we use the last M pages to replace the last segment. Whether a record is large or small, when searching in the last M pages, we apply the same technique described in section 6.1 and select a random start page for a search in the last M pages.

For our experiments, we define large size records as those records whose sizes are greater than the average record size. We find that CAA has better performance characteristics than the original algorithm. When properly selecting the parameter M (which indicates the number of pages at the end of the table space where large records are placed), CAA and the original algorithm consume similar amounts of table space. CAA improves the performance metrics for the following reason: (1) It directly reduces a search path for large records; (2) The availability of small slots unused by large records in the last M pages shortens the time to find space for small records; (3) Randomizing a starting lookup page for each thread helps reduce contentions.

We find experimentally that in high contention environments, when the parameter M is too small (compared to the number of threads), CAA might underutilize space. The reason is that when many threads with large records fail to find space in the last M pages at about the same time, there is a chance these threads start to allocate new pages concurrently. When the number of these new pages is significantly larger than parameter M, some new pages allocated by threads will reside outside the range of last M pages from the end of the table space. As a result, they are not available for the search step in the last M pages dictated by CAA. Consequently, these pages might be under filled in the immediate future. Other than the above observation regarding very small M values, our experiments show that the value of M does not significantly affect performance characteristics.

6.3.2 Experimental Results

We generate a number of sequences of distinct key values; each sequence having the same number of records. The sizes of the records we use are distributed between 100 and 250 bytes. The average record size is 175 and a standard deviation is 35. We conduct experiments using 10, 25, 40 and 55 concurrent threads. Parameter M is set to (# of threads)/2. We compare performance metrics of the original algorithm and CAA. The experimental results, shown in Figures 36 through 40, demonstrate that CAA has better performance metrics. To illustrate how parameter M affects space utilization, we evaluate CAA by varying M and the number of threads. Figure 41 shows space utilization over various values of M and the number of threads. To understand how the space utilization is related to the ratio of (M : number of threads), we display data presented in Figure 41 in a different format shown in Figure 42. We observe that when M exceeds (number of threads / 2) the space utilization gets greater than 90%. In conclusion, we can leverage this observation: a DBMS can make observations on the number of concurrent threads and dynamical adjust parameter M to ensure it operates in the mode to achieve the highest space utilization.

7 Performance Factors Analysis

In the two previous sections, we evaluate the table space allocation algorithm and a few enhancements with respect to several performance metrics. In this section, we further analyze how the performance metrics relate to performance. The ultimate performance measures are average response times and throughput for record inserts (while maintaining good space utilization). Some previous work (McAuliffe et al. 1996) used “objects created per second” metric to analyze the performance. This is a throughput measure and by itself does not provide enough details to explain how the throughput is affected by the cost of each step of the space search process. In addition to providing a throughput measure, our framework allows us to get a breakdown on the cost of each step of the space search process and identify bottlenecks by plugging-in hardware and DBMS dependent parameters. We explain how it can be done using the performance model below:

\[
\text{avg\_cost\_to\_insert\_a\_record} = \text{(cost\_of\_index\_look\_up)} + \text{(cost\_of\_check\_a\_page\_in\_SMAP)} + \text{(cost\_of\_check\_a\_record\_in\_SMAP)} + \text{(cost\_of\_per\_allocation\_operation)}
\]
The parameter values in parentheses, which depend on DBMS and hardware, are system parameters. The remaining parameters are algorithm parameters. With fixed system parameters, reducing the values of algorithm parameters by algorithm changes, as shown in previous sections, avg_cost_to_insert_a_record decreases and performance improves. The system parameters we use are:

- \( \cost_{\text{index\_look\_up}} \) is the cost of the primary index look up. We can make an assumption that non-leaf index pages are cached in a buffer pool. A percentage of leaf pages area accessed from disk units while the other pages are cached in a buffer pool.

- \( \cost_{\text{check\_a\_page\_in\_SMAP}} \) is the cost of checking SMAP if a page has enough space (i.e. the cost of checking a few bytes).

- \( \cost_{\text{per\_allocation\_operation}} \) is the cost to allocate a page (or several pages depending on the allocation scheme) at the end of the table space. It is an amortized cost between allocations that need extensions and those that do not.

- \( \cost_{\text{waiting\_on\_a\_latch}} \) is the cost of waiting for a latch until the waiting thread can access the resource protected by the latch. This cost is related to the speed of an operation on the contented resource and the waiting scheme. If the waiting scheme is always wait until success then it is mainly related to the operation speed on the contented resource (a page)

- \( \cost_{\text{fetching\_a\_data\_page}} \) is the cost of fetching a page from a disk or memory. Typically, some data pages are fetched from disk units while other data pages are cached in a buffer pool.

- \( \cost_{\text{fetching\_a\_SMAP\_page}} \) is the cost of fetching a SMAP page. If we assume that all SMAP pages are kept in a buffer pool, then it is a memory operation. Otherwise, its cost is similar to \( \cost_{\text{fetching\_a\_data\_page}} \), fetching a SMAP page with a percentage of pages coming from disk units and other pages cached in a buffer pool.

- \( \cost_{\text{insertion\_into\_a\_page}} \) is the cost to insert a record into a memory resident page. It includes finding the free block via an offset table on the same page and then locating the space to insert.

Three system parameters, \( \cost_{\text{fetching\_a\_data\_page}} \), \( \cost_{\text{index\_look\_up}} \), \( \cost_{\text{fetching\_a\_SMAP\_page}} \) include the cost a page read from disk units. This cost includes time of transferring data on I/O channel, reading data from the cache in disk units, and potential disk seek time if the page is not found in the cache. In a cost analysis using the performance model above, we can make reasonable assumptions on the percentage of pages read from a cache based on the sizes of a cache and a database.

8 Summary

The problem of space allocation in a DBMS remains important. In this paper, we study the performance characteristics of a table space allocation algorithm of a modern DBMS, make observations about its behavior with practical performance implications, identify opportunities for optimization, propose and evaluate algorithm enhancements, and quantify their benefits with respect to performance metrics. To conduct this research, we build an extensive simulation framework. In our study, we look at the space allocation problem from a new angle and consider factors like contentions and cluster ratios. We show that pre-sorting data leads to better insertion performance which has implications for load utilities. We believe to be the first to build a research tool for studying the effect of input patterns on space allocation algorithm behavior in a DBMS. Our framework is flexible and can be used to explore SSD optimized algorithms. This simulation framework is used by both database researchers and developers for (i) design exploration and (ii) using real-world workload patterns to identify high-value optimizations and avoid performance regressions. It allows us to enhance the algorithm in response to new requirements.

References


