RC24631 (W0809-020) September 4, 2008
Computer Science

IBM Research Report

Analyzing and Improving Table Space Allocation

Shanchan Wu
Department of Computer Science
University of Maryland
College Park, MD

Yefim Shuf, Hong Min, Hubertus Franke
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Bala Iyer, Frances H. Villafuerte, Julie Watts
IBM Silicon Valley Lab

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Analyzing and Improving Table Space Allocation

Shanchan Wu* Yefim Shuf[+] Hong Min[+]
Hubertus Franke[+] Bala Iyer[#] Frances H Villafuerte[#] Julie Watts[#]
* Department of Computer Science, University of Maryland, College Park
[+] IBM T.J. Watson Research Center [#] IBM Silicon Valley Lab
* wsc@cs.umd.edu [+, #] {yefim, hongmin, frankeh, 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 [1] and especially in large DBMS [2]. An efficient space allocation strategy that minimizes I/O frequency is essential. A common practice for improving read I/O performance is to cluster records with similar key values. For some access patterns, data clustering can significantly reduce or eliminate disk seek time -- a major contributing factor to I/O time. Maintaining data clustering is a desirable property for inserts. 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) [21, 22] 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 a broad class of 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 evaluates a typical table space allocation algorithm of a commercial DBMS and identifies conditions under which the algorithm performs the best. As far as we know, it addresses issues not previously investigated in the literature: it studies the quality of the cluster ratio achieved and the scaling of the algorithm when facing a large number of concurrent threads.
- 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 their benefits. We show one enhancement for reducing lock contentions during space search, another for reducing the number of data pages fetched before a page with enough free space is found, yet another enhancement to reduce the space search path length for variable size records.

Quantifying relative benefits of proposed enhancements helps decide which of them should be integrated into the implemented algorithm of a DBMS.

The rest of this 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 the design space exploration of table space allocation algorithms, while being important to the database community, has seldom been presented in the academic literature. Although there has been work on space management in 1996 [20], 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 adequate attention since the publication of [20] and must be revisited.

McAuliffe et al. [20] studied object placement algorithms from the standpoint of storage utilization and allocation performance without regard to clustering. Their work is concerned with 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 [20], we focus on multi-threaded workloads. We also use a more comprehensive set of performance parameters for analyzing performance costs from two orthogonal aspects: one that is dictated by the underlying hardware and DBMS, and one that is driven by the algorithm.

Our goal is to improve DBMS performance by reducing the CPU cost of allocating space for new records and creating a better data layout that will ultimately lead to fewer I/O delays. There has been work on improving I/O performance from different angles: disk access optimizations, prefetching and architectures for storing large volumes of data. Work has been done on arm scheduling [5], bandwidth boosting [8], cache optimizations [9], layout optimizations [6,7]. Performance modeling of disk drives has been studied [10] and I/O simulation tools have been developed. Sorting of RIDs was used to reduce I/O for bulk deletes in [4]. We are first to build a research tool for understanding the effects of various input patterns on the insertion algorithm behavior.

Data prefetching was shown to reduce synchronous I/O operations. Soloviev [12] studied prefetching in disk caches. Hsu et al. [2] examine the logical I/O reference behavior of the peak production database workloads from ten of the world's largest corporations. Their focus is on analyzing factors that affect how these workloads respond to different techniques for caching, prefetching, and write buffering. Wilson et al. [23] discussed the design and evaluation of conventional dynamic memory allocators. Chen et al. [13] applied data mining technique to discover block correlations in storage systems, and show that correlation-directed prefetching and data layout can reduce average I/O response times. Due to the popularity of high-density Flash memory as data storage medium, some began designing Flash-based DBMS [11]. 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. Most contentions that were dealt with are at the transaction level. To ensure data integrity, various locking schemes were proposed such as two-phase locking [14] and tree locking [15]. Several concurrency control techniques were investigated [13,14,16,17,18]. The contentions addressed in the literature are largely contentions on accesses to records and indexes of databases rather than contentions during table space allocation, which is a focus of our paper. Mohan et al. in [19] designed methods to improve concurrency and space utilization by space reservation and space tracking.

3. TABLE SPACE AND TABLE SPACE ALLOCATION ALGORITHM

Figure 3-1 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.

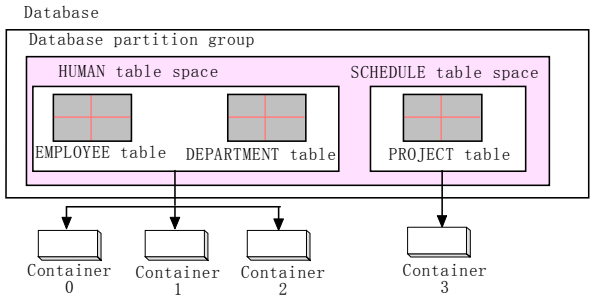

Figure 3-1. Table spaces and tables in a database

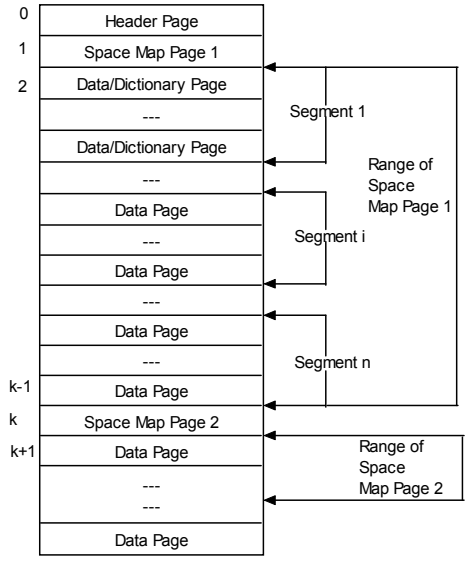

Figure 3-2. The structure of a table space

Figure 3-2 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 in a page set covers 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 tree is updated after a record is inserted into a table. The number to the left of each page in Figure 3-2 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. In some DBMS, the quality of data clustering is measured by a cluster ratio. A cluster ratio gives an indication of how closely the order of index entries on index leaf pages matches the actual ordering of rows on data pages. The higher the cluster ratio, the lower the cost of referencing data pages during an index scan. To maintain a high cluster ratio, the algorithm first tries to insert a record into a candidate page referred by the index tree. Then it tries to insert records with the same or similar keys into locations near each other. It considers pages in the vicinity of a candidate page, i.e. the same segment where the candidate data page is located, and in the next step, considers pages in segments 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 tries to access a data page or a SMAP that already resides 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 allow for more 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 that are 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 contended 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 the memory footprint, the tool does not store the content of records, but only keys and records' 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 different attributes. The attributes are configurable.

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

Figure 4-1 describes the structure of the simulation framework.

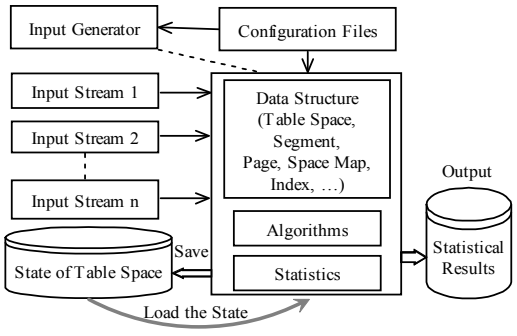

Figure 4-1. The structure of the simulation framework

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 prologs a space search process for those records. The quality of data clustering degrades.

We simulate this workload and investigate potential 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 percentage of free space. After that, several concurrent streams periodically insert records into the table space. When there is sufficient free space in a table space, a record can be inserted quickly. When the available space decreases, it takes more effort 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 (i.e. 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 account activities. We create input sequences representing this skewed account activity pattern. Each sequence consists of many 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.

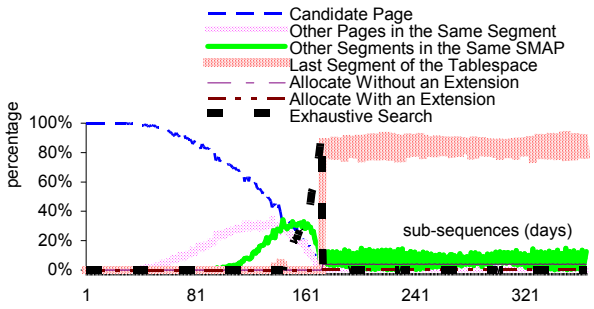

Figure 5-1. The trend of where space for a record is found for the "Banking application example"

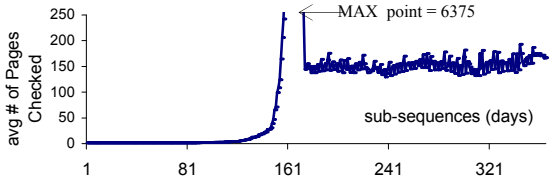

Figure 5-2. The trend of number of Pages Checked in SMAP for the "Banking application example"

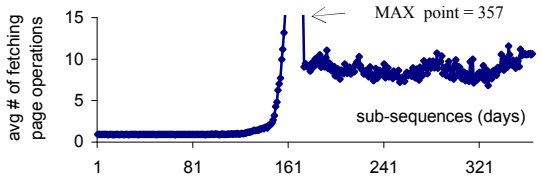

Figure 5-3. The trend of fetching page operations for the "Banking application example"

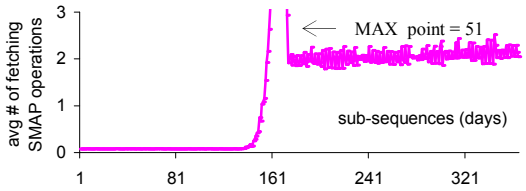

Figure 5-4. The trend of fetching SMAP operations for the "Banking application example"

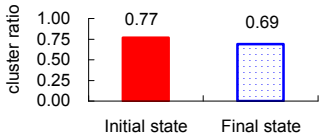

Figure 5-5. Cluster ratios

The performance metrics of the experiment are shown in Figure 5-1 through 5-5. In Figure 5-1 through 5-4, x-axis value corresponds to a particular sub-sequence (a particular day). For example, x=50 means the 50th subsequence or the 50th day. Figure 5-1 shows where the records are inserted into (candidate pages, other pages in the same segment and so on, corresponding to algorithm steps described in Section 3) over time, represented by a percentage of records in a sub-sequence. 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 5-2 shows the average number of pages checked in SMAP per record insertion. Figure 5-3 and Figure 5-4 show the average number of fetching operations per record insertion. Figure 5-5 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.

5.1.2.1 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 5-6 through Figure 5-10.

A comparison of Figure 5-1 to 5-5 with Figure 5-6 to 5-10 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.

5.1.2.2 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

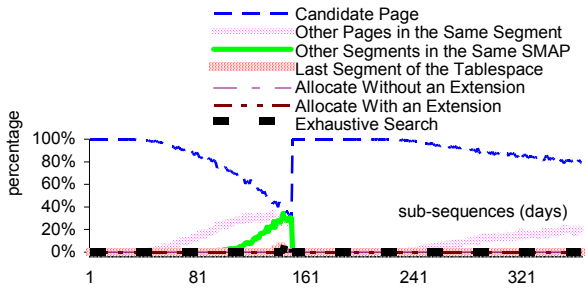

Figure 5-6. The trend of where space for a record is found for "Banking example" with reorganization at day 150

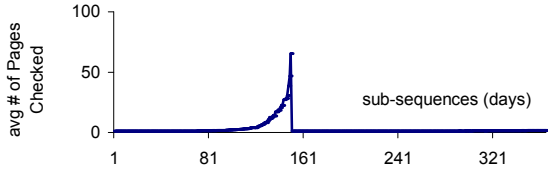

Figure 5-7. The trend of number of Pages Checked in SMAP for "Banking example" with reorganization at day 150

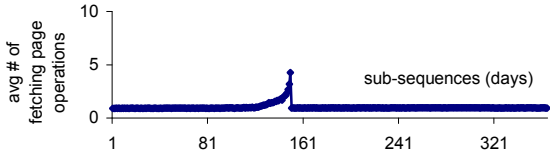

Figure 5-8. The trend of fetching page operations for "Banking workload example" with reorganization at day 150

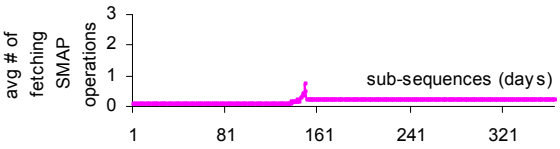

Figure 5-9. The trend of fetching SMAP operations for "Banking example" with reorganization at day 150

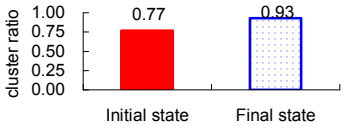

Figure 5-10. Cluster ratios with reorganization at day 150

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 5-11 through Figure 5-15.

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 5-12 through 5-14. In contrast, in the original algorithm, the pulses appear in Figures 5-2 through 5-4. 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 exhaustive 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 space allocation 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 a 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.

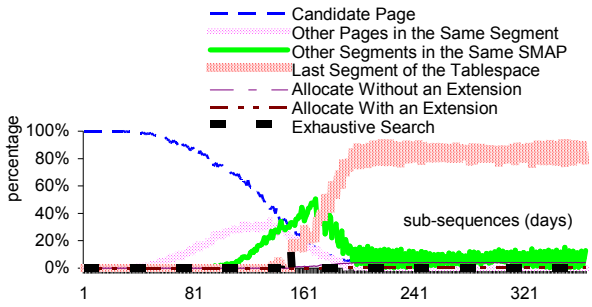

Figure 5-11. The Trend of where space for a record is found for a "Banking workload example" with a detection scheme

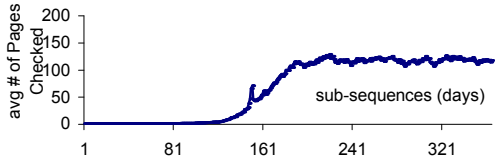

Figure 5-12. The trend of number of pages checked in SMAP for "Banking example" with a detection scheme

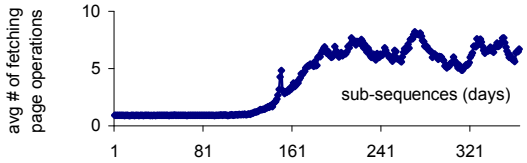

Figure 5-13. The trend of fetching page operations for "Banking example" with a detection scheme

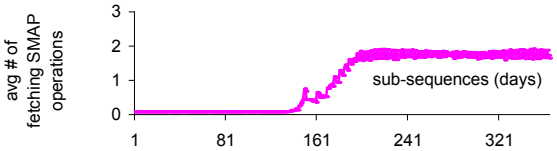

Figure 5-14. The trend of fetching SMAP operations for "Banking example" with a detection scheme

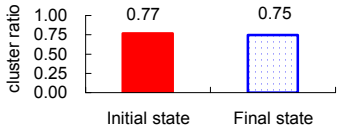

Figure 5-15. Cluster ratios with a detection scheme

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 5-16 and 5-17 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 5-18 through 5-21 show the comparison of the average number of page latch hits, the average number of pages checked in SMAP, 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 5-18 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 5-19 through 5-21 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 5-18 through 5-21 further suggests that, to minimize the time to insert records into a table space, it is advisable to order the records before inserting. Furthermore, Figure 5-22 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 related to heavy contentions that are often present in highly multi-threaded environments and the frequency of I/O operations. Our goal is reducing contentions and I/O operations while at the same time maintaining or improving a cluster ratio. In this section, we propose three techniques that improve the space search algorithm targeting the areas of reducing contentions and reducing I/O operations. 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.

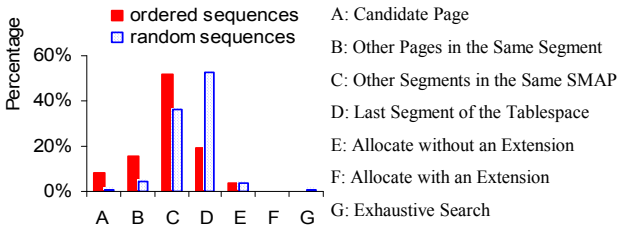

Figure 5-16. Summary of where free space for a record is found for 10 threads (Ordered vs. Random)

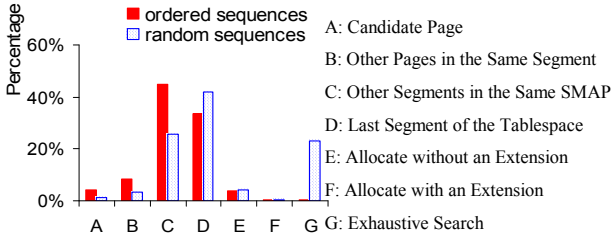

Figure 5-17. Summary of where free space for a record is found for 25 threads (Ordered vs. Random)

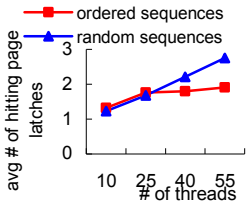

Figure 5-18. The average numbers of page latch hits

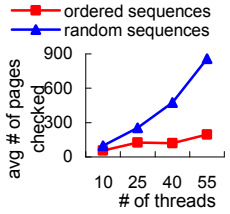

Figure 5-19. The average numbers of pages checked in SMAP

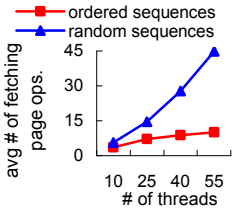

Figure 5-20. The average numbers of fetching data page operations

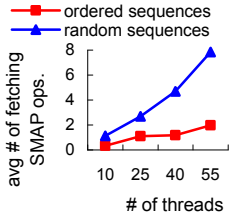

Figure 5-21. The average numbers of fetching SMAP operations

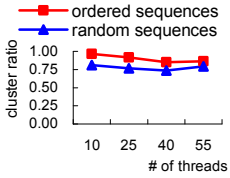

Figure 5-22. Cluster ratios

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 about the same time. The first

thread that gets the page will lock it, perform space checking, and insert into it. All subsequent 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. This random offset number identifies the start searching page in the last segment of the current table space. As newly allocated space becomes the last segment of the growing table space, the offset number for each thread to start a search within the last segment remains unchanged until a thread terminates. For a period of time during the insertion, the most recently visited data page and the most recently visited SMAP for each thread are likely maintained in memory, and this helps reduce the number of page fetching operations.

6.1.3 Experimental Results

We generate a number of 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 algorithm and the enhanced algorithm (proposed in Section 6.1.2) that reduces contentions.

The experimental results are in Figures 6-1 through 6-4. The average number of page latch hits with the enhanced algorithm is considerably smaller than with the original algorithm. With the enhanced algorithm, the number of pages checked in SMAP and the number of fetching operations are also smaller especially when the number of threads is high (in experiments when number of threads is greater than 40). There is no significant difference in the cluster ratios. The performance metrics collectively demonstrate that the proposed technique reduces contentions and improves algorithm performance.

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 the process of searching for free space is to keep track of 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

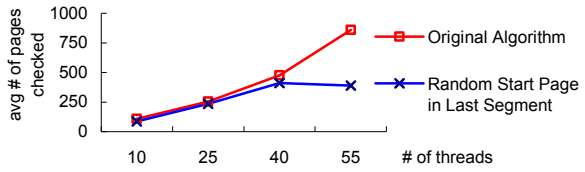

Figure 6-1. The average numbers of pages checked in SMAP

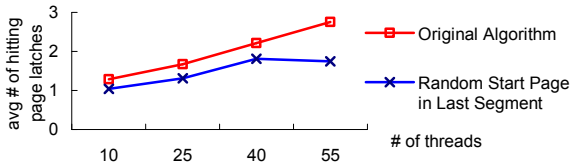

Figure 6-2. The average numbers of page latch hits

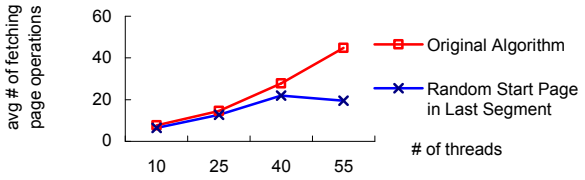

Figure 6-3. The average numbers of fetching page operations

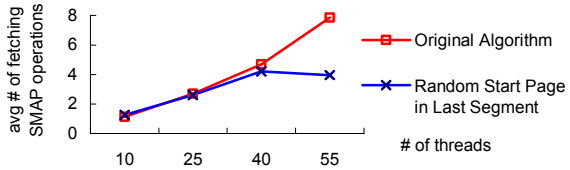

Figure 6-4. The average numbers of fetching SMAP operations

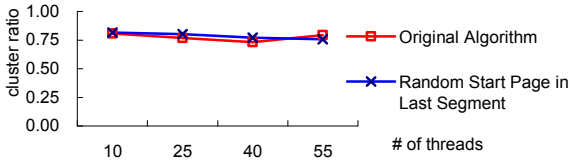

Figure 6-5. Cluster ratios

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 this item in RHL_LIST. There are three possible status states for each item: AVAILABLE, TRASH, BUSY. The AVAILABLE status of an item indicates that the page in this item is not 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.

The strategy of the table space allocation algorithm by using RHL_LIST is as following.

(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. If not successful, try to get an available page from the RHL_LIST and insert the record into the page. Update RHL_LIST when applicable (more details on updating are described later).

(2) If the placement in step 1 fails, find space within the same segment where the candidate data page is located. If not successful, try to get an available page from the RHL_LIST and insert the record into the page. Update RHL_LIST when applicable (more details on updating are described later).

(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. If not successfully, try to get an available page from the RHL_LIST and insert the record into the page. Update RHL_LIST when applicable (more details on updating are described later).

(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. If not successfully, try to get an available page from the RHL_LIST and insert the record into the page. Update RHL_LIST when applicable (more details on updating are described later).

(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.

In this strategy, we alternate between the search steps in the original space allocation algorithm and the search using *RHL_LIST* (the differences are underlined).

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. The pseudo code for these three operations is as follows:

```
update_status_of_page(item, newStatus){
      temporarily latch the item. // when leaving that item, unlatch it.
      If (item.status != BUSY) item.status = newStatus
}

get_one_available_page( ){
      for each item in the RHL_LIST by starting from a random position{
            temporarily latch the item. // when leaving that item, unlatch it.
            if  (item.status == AVAILABLE){
                  check the available size of the page in the item.
                  if the size of the page is less than the maximum size of a record, update
                  the status of this item to be TRASH, go to next item; otherwise, return
                  this item.
            }else{
                  go to next item.
            }
      }
      return null // fail to get an available page from the RHL_LIST
}
```

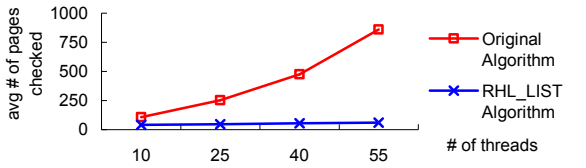

Figure 6-6. The average numbers of pages checked in SMAP

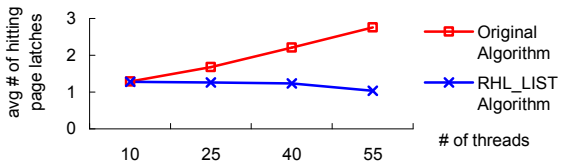

Figure 6-7. The average numbers of page latch hits

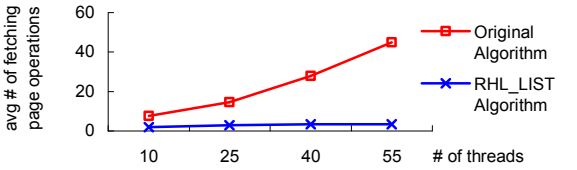

Figure 6-8. The average numbers of fetching page operations

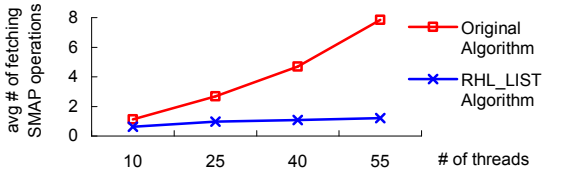

Figure 6-9. The average numbers of fetching SMAP operations

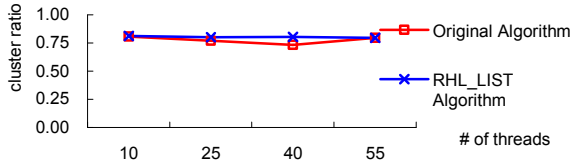

Figure 6-10. Cluster ratios

```
insert_one_new_item(newPage, newSMAP ){
    for each item in the RHL_LIST by starting from a random position {
        temporarily latch the item. // when leaving that item, unlatch it.
        if  (item.status == TRASH){
            item.page = newPage; item.status = AVAILABLE; item.smap = newSmap
            return
        }else{
          go to next item.
        }
    }
}
```

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 be 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 if 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.

6.2.2 Experimental Results

We generate a number of 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 algorithm and the enhanced algorithm using the *RHL_LIST*. The experimental results are shown in Figures 6-6 through 6-10. Although cluster ratios do not change, other performance metrics (e.g., the number of pages checked and the number of pages fetched during a space search) improve significantly. In Figures 6-6 through 6-9, 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 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 visited data pages that have free space. Attempting to insert records directly into these pages reduces time spent on searching for free space.

As an alternative to tracking data pages that have 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 would first check the candidate page, then try to search in some SMAP pages in that structure, then try to search in the last segment, and then proceed as in the original algorithm. The structure which keeps a set of SMAP pages will need 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 the following enhancement. (a) For a large record, we first try a candidate page; if we cannot insert the record in the candidate page, then we skip both the search for pages in the

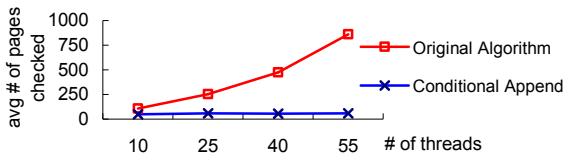

Figure 6-11. The average numbers of pages checked in the SMAP

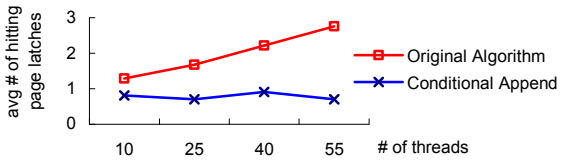

Figure 6-12. The average numbers of page latch hits

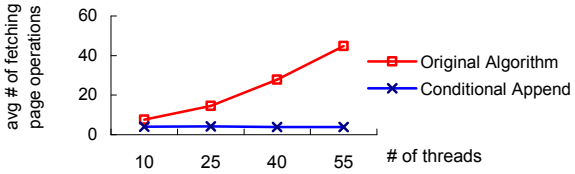

Figure 6-13. The average numbers of fetching page operations

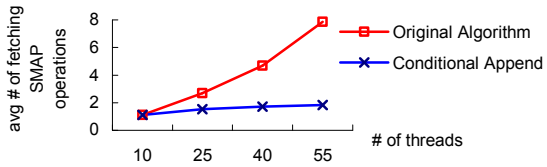

Figure 6-14. The average numbers of fetching SMAP operations

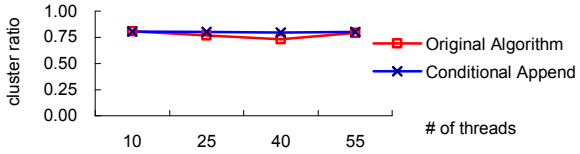

Figure 6-15. Cluster ratios

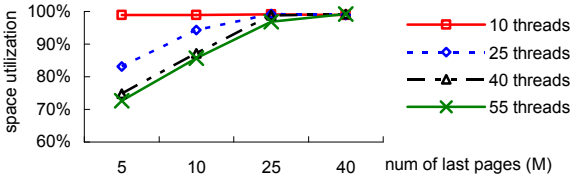

Figure 6-16. Space utilization

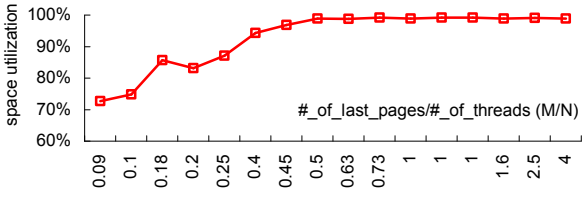

Figure 6-17. Space utilization with respect to #_of_last_pages/#_of_threads (M/N)

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 that 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. We call this enhanced algorithm *Conditional Append Algorithm* or CAA.

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 at 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 6-11 through 6-15, 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 6-16 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 6-16 in a different format shown in Figure 6-17. 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 [20] 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:

avg_cost_to_insert_a_record = (cost_of_index_look_up)
*+ (cost_of_check_a_page_in_SMAP) * avg_numPagesCheckedInSMAP*
*+ (cost_of_per_allocation_operation) * avg_numAllocationOperation*
*+ (cost_of_waiting_on_a_latch) * avg_numHittingPageLatches*
*+ (cost_of_fetching_a_data_page) * avg_numFetchingDataPages*
*+ (cost_of_fetching_a_SMAP_page) * avg_numFetchingSMAPs*
+ (cost_of_insertion_into_a_page)

We call the parameter values in the parentheses, which are dependent on DBMS and hardware, *system parameters*. We call the remaining parameters *algorithm parameters*. For fixed values of system parameters, when we reduce the values of algorithm parameters by algorithm changes, as we show in previous sections, the *avg_cost_to_insert_a_record* will decrease and the performance will improve. We use the following system parameters:

cost_of_index_look_up is the time 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_of_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_of_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_of_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_of_fetching_a_data_page is the cost of fetching a page from the disk or the memory. In most cases, a percentage of data pages will be fetched from disk units while the other data pages are cached in a buffer pool.

cost_of_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_of_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_of_insertion_into_a_page is the cost to insert a record into a page which is already in memory. It includes the cost of finding the free block via an offset table on the same page and then locating the space to insert.

Three system parameters, *cost_of_fetching_a_data_page cost_of_index_look_up, cost_of_fetching_a_SMAP_page* include a cost of reading a page from disk units. This cost includes time of transferring data on I/O channel, reading data from the cache in the 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 page that are read from a cache based on the sizes of a cache and a database.

7.1 Implications of SSD Technology

Recent technological changes help make SSDs [21, 22] more affordable and available. SSDs present different performance characteristics compared to HDs. Random read time is significantly lower in SSDs than in HDs since there is no seek time in SSDs. Sequential read access times in SSDs do not show a similar scale of improvement over HDs as most pages are read from a disk cache in both cases. The read and write speed of a SSD is asymmetric as it takes longer to write. SSDs provide interfaces (compatible with HDs) to applications such as DBMS. When databases are migrated to SSDs, we can still apply the same framework contributed by this paper to study space search performance, using different system parameters learned from SSDs. For example, we can make the same assumption that a percentage of index, SMAP and data pages are cached in a database buffer pool, another percentage is read from a disk cache, and the rest of pages are read from a disk (which takes much less time to read when using SSDs).

Other characteristics in SSDs that are different from those in HDs include a rather limited number of erase (write) cycles due to a wear out effect and an internal disk *garbage collection* to mitigate this effect, a block level erase operation, possibly a different physical and logical page mapping than in HDs, and etc. In addition, the fact that enterprise class SSD storage remains considerably more expensive than HDs suggests that better space management, achieving lower internal in-page fragmentation, and higher space utilization will be even more critical for an enterprise DBMS using SSDs. These additional or different characteristics provide new research and design opportunities to further optimize DBMS overall performance and maximize space utilizations when SSDs are used for backend stores. The optimizations in disk units and DBMSs might further lead to different SSD database design practices than HD. We plan to investigate these issues using our simulation framework in the future work and further extend the framework to accommodate advances in database design practices.

8. SUMMARY

Due to the constant technological changes, emergence of new applications, and demands for rapid loading of high volumes of data, the problem of space allocation in a DBMS becomes even more important today than a decade ago. In this paper, we present an in-depth study of the performance characteristics of a table space allocation algorithm of a modern DBMS, make several observations about its behavior with practical performance implications, identify opportunities for optimization, propose and evaluate several algorithm enhancements, and quantify their benefits with respect to relevant performance metrics. To conduct this research, we build an extensible simulation framework. In our study, we look at the space allocation problem from a new angle and consider factors like contentions in multi-threaded environments and cluster ratios. We show that pre-sorting data can lead 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 various input patterns on the space allocation algorithm behavior for a DBMS. Our framework is flexible and can be used to explore SSD optimized algorithms. This simulation framework is being used by both database researchers and by a development product team 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.

9. REFERENCES

[1] J. L. Hennessy, D. A. Patterson. Computer architecture: a quantitative approach, Morgan Kaufmann Inc., 1996.

[2] W. W. Hsu, A. J. Smith, H. C. Young. I/O reference behavior of production database workloads and the TPC benchmarks – an analysis at the logical level. ACM Tran. on Database Systems (TODS), v.26 n.1, p.96-143, March 2001.

[3] P. A. Franaszek, J. T. Robinson, A. Thomasian. Concurrency control for high contention environments. ACM Tran. on Database Systems (TODS), v.17 n.2, p.304-345, June 1992.

[4] A. Gartner, A. Kemper, D. Kossmann, B. Zeller. Efficient Bulk Deletes in Relational Databases. In Proc. of the IEEE ICDE 2001, p.183-192.

[5] B. L. Worthington, G. R. Ganger, Y. N. Patt, J. Wilkes. On-line extraction of SCSI disk drive parameters. In Proc. of ACM SIGMETRICS 1995, p.146-156.

[6] C. Ruemmler, J. Wilkes. Disk Shuffling. Technical Report HPL-91-156, HP Laboratories, Oct 1991.

[7] W. W. Hsu, A. J. Smith, H. C. Young. The automatic improvement of locality in storage systems. ACM Tran. on Computer Systems (TOCS), v.23 n.4, p.424-473, Nov. 2005.

[8] D. A. Patterson, G. Gibson, R. H. Katz. A case for redundant arrays of inexpensive disks. SIGMOD 1988, p.109-116.

[9] Y. Hu, Q. Yang. DCD – disk caching disk: a new approach for boosting I/O performance. ISCA 1996, p.169-178.

[10] C. Ruemmler, J. Wilkes. An introduction to disk drive modeling. Computer, v.27 n.3, p.17-28, March 1994.

[11] S. Lee, B. Moon. Design of Flash-Based DBMS: An in-page logging approach. In Proc. of ACM SIGMOD 2007, p.55-66.

[12] V. Soloviev. Prefetching in segmented disk cache for multi-disk systems. In Proc. of the 4th workshop on I/O in parallel and distributed systems: part of the FCRC, p.69-82, 1996.

[13] Z. Li, Z. Chen, Y. Zhou. Mining block correlations to improve storage performance. ACM Tran. on Storage (TOS), v.1 n.2, p.213-245, May 2005.

[14] P. Felber and M. K. Reiter. Advanced Concurrency Control in Java. Concurrency and Computation: Practice and Experience, 14(4):261-285, 2002.

[15] A. Silberschatz and Z. Kedem. Consistency in Hierarchical Database Systems. J. of ACM, 27(1), Jan. 1980.

[16] A. Reuter. A Concurrency on high-traffic data elements. In Proc. of ACM PODS 1982, p.83-92.

[17] D. Gawlick, D. Kinkade. Varieties of concurrency control in IMS/VS fast path. IEEE Database Eng., 4 (1985), 63-70.

[18] M. Hsu, B. Zhang. Performance evaluation of cautious waiting. ACM TODS v.17 n.3, p.477-512, Sept. 1992.

[19] C. Mohan, D. Haderle. Algorithms for flexible space management in transaction systems supporting fine-granularity locking. In Proc. of EDBT 1994, p.131-144.

[20] M. L. McAuliffe, M. J. Carey, M. H. Solomon. Towards effective and efficient free space management. In Proc. of the ACM SIGMOD 1996, p.389-400.

[21] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, R. Panigrahy. Design Tradeoffs for SSD Performance. USENIX 2008, p.57-70.

[22] V. Prabhakaran, T. L. Rodeheffer, L. Zhou. Transactional Flash. In proc. of ACM OSDI 2008, p.147-160.

[23] P. R. Wilson, M. S. Johnstone, M. Neely, D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. In proc. of IWMM 1995, p.1-116.