Lock Management in a Main-Memory DBMS

Sang-Wook Kim
Division of Computer, Information, and Communications Engineering
Kangwon National University
192-1 Hyoja 2 Dong, Chunchon, Kangwon, Korea
Tel: +82-33-250-6392, Fax: +82-33-252-6390
email: wook@kangwon.ac.kr

Abstract: The *locking* is the most widely-used concurrency control mechanism for guaranteeing logical consistency of a database where a number of transactions perform concurrently. In this paper, we propose a new method for lock management appropriate in main-memory databases. Our method chooses the *partition,* a fixed-sized container for records, as a unit of locking, and directly keeps lock information within the partition itself. These make our method enjoy the following advantages: (1) it has freedom in controlling of the trade-off between the system concurrency and the lock processing overhead by considering the characteristics of given target applications, (2) it enhances the overall system performance by eliminating the hashing overhead, a serious problem occurred in the traditional method.

1. Introduction

Recently, as the capability of computing devices gets dramatically higher, the coverage of real-time applications is becoming much wider. A typical way to manage the databases effectively in such real-time applications is to replace disk with main-memory for storage[1]. The main-memory DBMS(MMDBMS), which employs main-memory for primary storage, resolves the radical problem of the delayed processing time due to disk accesses occurred in the disk-resident DBMS[2][3][4].

Real-Time DBMS Team at Electronics and Telecommunications Research Institute(ETRI) and Data & Knowledge Engineering Lab. at Kangwon National University have been working together to develop the *Tachyon,* a high-performance MMDBMS. The Tachyon supports the deadline concept since it considers real-time applications as its major target[5]. Also, it hires the object-oriented data model to accomodate diverse applications easily.

This paper discusses concurrency control in the Tachyon. The concurrency control manager is a sub-component of a DBMS that controls the execution order of concurrent transactions in order to prevent them from destroying the consistency of a database[6]. Concurrency control methods currently-proposed are classified into the two-phase locking(2PL) protocol[7][8], the time-stamp ordering scheme[9], the optimistic method[10], and the multiple version method[11]. The 2PL protocol is to control the concurrency by making a transaction acquire the lock before accessing the corresponding data item. Most of commercial DBMSs employ it due to its practicality[12][13].

Basically, our concurrency control manager employs the *2PL protocol,* and maintains its locks with the following two features. First, it employs the *partition,* an allocation unit of main-memory, as a locking granule, and thus effectively adjusts the trade-off between the system concurrency and the lock processing cost through the analysis of applications. Second, it reduces the lock processing cost significantly by maintaining the lock information directly in the partition itself without hashing.

This paper is organized as follows. As related work, Section 2 briefly describes the method for managing the lock information in the disk-resident DBMSs, then points out the problems of applying it to MMDBMSs. Section 3 presents the locking granularity chosen in the Tachyon, and justifies the reason for this choice. Section 4 presents our approach to manage the lock information in the Tachyon, and discusses its advantages. Section 5 details the main data structures necessary for implementing our approach. Finally, Section 6 summarizes and concludes the paper.

2. Related work: lock management in disk-resident DBMSs

Most of disk-resident DBMSs employ the *hash structure* for managing the lock information[14]. Figure 1 shows the basic structure for lock management using hashing. We can find the corresponding entry in the *hash table* by applying the hashing function to the data item to be locked. Each hash entry points to the list of lock headers, and subsequently each lock header points to the list of lock requests. The list of lock headers called the *hash chain* implies the occurrence of the hash conflicts. The list of lock requests means that multiple transactions try to acquire a lock on the same data item.

The lock header blocks and the lock request blocks are dynamically maintained in two separate pools. They are allocated/deallocated from/to the pools. Also, each transaction points to a list of its own lock request blocks corresponding to the locks requested by that transaction.

The most important thing in this hash-based management is that it maintains only the information of such locks that are acquired or requested by active transactions. Thus, this method minimizes the usage of the storage space in main-memory. However, the following problems occur when we apply this method directly to the MMDBMS.

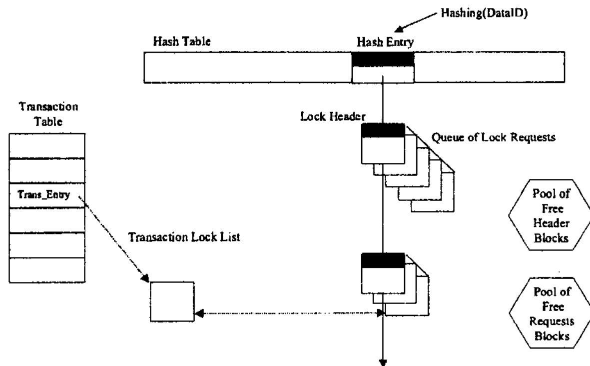

Figure 1. Lock management based on hashing in disk-resident DBMSs.

The first one is the overhead caused by using the hash structure. The hashing incurs a high cost because it has to perform the hashing function and to manage the hash chains. This cost is negligible in disk-resident DBMSs since the cost for disk accesses is dominant in all database operations. However, this is not the case in MMDBMS environment since there are no disk accesses. Therefore, the cost of hashing could get larger than that of searching and updating data items in MMDBMSs.

The second one is the overhead for mutual exclusion of hash chains. The hash chain is a shared data structure read/written concurrently by multiple transactions in a DBMS. Thus, we need a mechanism that guarantees the physical consistency of the hash chains. While the cost for this mutual exclusion is negligible in disk-resident DBMSs, this is also not the case in MMDBMSs.

The third one is the overhead for maintaining the lock header blocks. As mentioned earlier, they are maintained dynamically within its pool in hash-based lock management. Thus, a lock header block is allocated/deallocated from/into the pool whenever a new block is requested or an allocated block is returned. In addition, mutual exclusion of the pool has to be guaranteed since the pool is a shared data. These two overheads cause the performance of the MMDBMSs to deteriorate.

3. Our choice of lock granularity

The lock granularity determines the size of the individual data item for locking. The lock granularity determines the lock processing cost and also the system concurrency[8]. Typical locking granules are the database, segment, page, and record. As the lock granularity becomes larger, the lock processing cost gets smaller while the system concurrency gets lower. On the contrary, as the lock granularity becomes smaller, the system concurrency gets higher while the lock processing cost gets larger[3].

In disk-resident DBMSs, the obtaining and releasing of locks incur only main-memory accesses. Their costs are quite small compared with those of whole searching and updating of data items that incur disk accesses. Thus, the disk-resident DBMSs mainly focus on maximizing the system concurrency rather than the lock processing cost. As a result, commercial DBMSs employ the *record* as the locking granule.

On the contrary, since the searching and updating of data items perform within main-memory in MMDBMSs, thus their entire cost is so small. Thus, the costs for obtaining and releasing of locks become fairly important. Also, the duration of holding locks tends to be short in MMDBMSs since transactions perform very fast, and therefore, lock conflicts among transactions are rare. Thus, the performance gains coming from the small locking granule diminish in MMDBMSs[3].

This fact causes the MMDBMSs to hire smaller lock granules. Reference [2] allocates just one lock for the entire database. This strategy contributes to minimize the lock processing cost. Furthermore, it completely eliminates additional overhead for handling deadlock detection and resolution since the deadlock does not occur in this situation with only one lock. However, the transactions that read and write the database are not able to run simultaneously, and thus the system concurrency degrades too much.

In this research, we observed that the lock granules of extremely large or small sizes such as the database and the record are not appropriate for the MMDBMS. Thus, we considered the *segment* and the *partition* whose sizes are in the range of the two extremes as the candidates for the lock granule in the Tachyon. The segment is a *logical* unit for storing the records having the same schema, and corresponds to the relation in relational databases. The segment consists of a number of partitions. The partition is the *physical* allocation unit of main-memory with a fixed size, and stores multiple records.

The MMDBMS Starburst[4] employs the segment as a lock granule. However, in case multiple transactions simultaneously access the records in a segment, the system concurrency deteriorates seriously as in case of the granule of the database. Also, it is not easy to predict the degree of the system concurrency since the size of a segment varies as the number of records changes. Currently, the Starburst dynamically changes its lock granule into the record or the segment according to the features of applications. However, this strategy requires an additional overhead for checking the status, and thus causes the performance of MMDBMS to degrade.

Through these considerations, we select the partition as the lock granularity for avoiding two extremes of the low system concurrency and the high lock

processing cost. Also, we can easily adjust the size of the partition when installing the DBMS. If users require the high system concurrency, we make the partition smaller. On the other hand, if users do not want to pay the high lock processing cost, we make the partition larger. Therefore, we can effectively adjust the trade-off between the system concurrency and the lock processing cost through the analysis of target applications.

4. Our approach: lock management in the Tachyon

In the Tachyon, we keep the lock information directly within the data item unlike the hash-based method in disk-resident DBMSs. Since we hire the partition as the locking granule in the Tachyon, the lock header containing the lock information resides in each partition. Figure 2 shows a basic data structure for managing lock information. We see that there are no hash table and pool for lock header blocks since the lock header block exists within its corresponding partition. As a result, the proposed method clearly resolves the three overheads caused by the hash-based method.

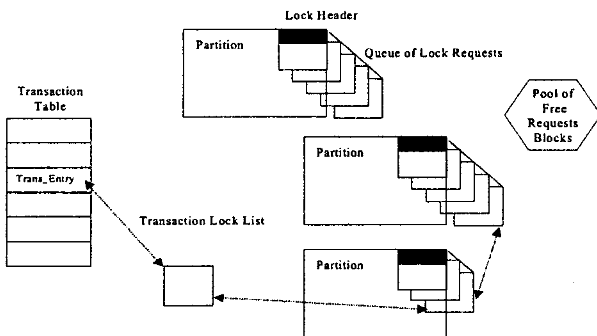

Figure 2. Lock management in the proposed method.

While the hash-based method dynamically maintains lock header blocks that correspond to the locks acquired or requested, the proposed method maintains them statically within every partition. Thus, main-memory space for the lock header block wastes in each partition when the partition is not locked. However, this space is not that serious in case of using the partition as a locking granule. In the current implementation, the lock header occupies 28 Bytes and the partition 8 KBytes. Therefore, the proposed method requires only 0.33% additional space for this static maintenance. This small space overhead is sufficiently compensated by the rapid response time of the MMDBMSs.

The question naturally arising at this point is the effect of applying the proposed method to disk-resident DBMSs. For example, we can include a lock header in each record in the same way. However, the lock header blocks reside on disk in disk-resident DBMSs as the records do. Thus, disk accesses occur for lock processing whenever a transaction issues a lock request. If the lock is immediately acquired, the transaction obtains that record directly from the buffer rather than disk. If the transaction has to wait for other transactions to finish due to the lock conflict, however, the page containing the record is likely to be swapped out in the meantime. This makes the system performance deteriorate seriously. In contrast, this problem does not occur in MMDBMSs, since the whole database is resident on main-memory.

5. Data structures for lock management

This section details the main data structures shown in Figure 2 necessary for implementing the proposed method. The Tachyon manages all the data structures in the *shared memory* of the Unix[15].

LockHeader corresponds to the lock header in each partition and represents whether the partition is locked. *LockHeader* consists of four sub-elements of *Latch, RequestQueue, GrantedMode,* and *Waiting*. *Latch* is used as a means for *latching*[16] that guarantees the physical consistency of LockHeader. *RequestQueue* is a pointer to the list of *LockRequests* described below. *GrantedMode* represents the lock mode held on the partition, and has one of two values of the shared or exclusive mode[14]. The shared mode allows multiple transactions to read the corresponding partition concurrently, but does not allow any writings. The exclusive mode allows only one transaction to write the corresponding partition. Also, *Waiting* indicates whether other transactions are waiting for the lock in this partition to be released.

LockRequest corresponds to the lock request block and manages the information of the locks obtained or requested by transactions. *LockRequest* is allocated from its pool responding to the lock request and is returned into the pool at the time of lock releasing. The newly allocated *LockRequest* is added to the list pointed by *RequestQueue* of *LockHeader*.

LockRequest is composed of the seven sub-elements of *RequestQueueNext, RequestQueuePrev, Status, GrantedMode, TransEntry, Next,* and *Prev*. *RequestQueueNext* and *RequestQueuePrev* are the pointers for keeping the lock request blocks as a doubly-linked list. *Status* represents the status of the transaction requesting the lock, and has one of three values of *granted, waiting,* and *converting*. The value of *converting* appears when a transaction has acquired a shared lock on a partition together with other transactions, and then requests an additional exclusive lock on the same partition. This case has to be handled in a special way, and thus represented as a different status.

GrantedMode represents the lock mode, and has one of two modes: shared or exclusive one[14]. *TransEntry* points to the entry in the transaction table that has all the information of the transaction issuing the lock request. Finally, *Next* and *Prev* are used to construct the *transaction lock list* as shown in Figure 2. This list is useful for releasing all the locks requested by the transaction when it ends.

TransEntryTable corresponds to the transaction table

that stores all the information about the currently running transactions. *TransEntryTable* allocates an empty entry when a new transaction starts to run, and returns it when the transaction ends. *TransEntryTable* consists of many sub-elements, but here we present some of them related to concurrency control such as *LockRequestList, LatchWaitingList, LockWaitForList, SemID,* and *LatchMode*. *LockRequestList* points to the list of the lock request blocks requested by the transaction that corresponds to this entry. *LatchWaitingList* is a pointer to the list of the entries, each of which corresponds to the transaction waiting for the same latch.

We use the Unix semaphore[15] to wake up the transactions in the sleep state into the ready state or to make the running transactions be in the sleep state. The *SemID* is an semaphore identifier allocated to each transaction for this purpose. Finally, *LatchMode* represents the mode of the latch which the transaction likes to acquire, and has a value of shared or exclusive ones.

6. Conclusions

In this paper, we have discussed lock management in the Tachyon, an MMDBMS. The cost of lock management occupies a large portion of the cost of searching or updating a data item in MMDBMSs since there are no costly disk accesses. Thus, efficient management of locks is fairly important to enhance the performance of the entire MMDBMS.

The lock management in the Tachyon has the characteristics as follows: First, it employs the *partition*, an allocation unit of main-memory, as a locking granule, and thus effectively adjusts the trade-off between the system concurrency and the lock processing cost through the analysis of applications. Second, it reduces the lock processing cost significantly by maintaining the lock information directly in the partition itself without hashing.

The Tachyon has been served as a high-performance storage engine for a variety of real-time applications especially in telecommunications domain. Currently, we are performing extensive experiments in real domains in order to show the effectiveness of our approaches adopted in our concurrency control manager compared with the previous ones.

Acknowledgment

This research was supported by the 2000-2001 Research Project(Grant KRF-2000-041-E00258) of Korea Research Foundation(KRF). Sang-Wook Kim would like to thank Jung-Hee Seo, Suk-Yeon Hwang, Grace(Joo-Young) Kim, and Joo-Sung Kim for their encouragement and support.

References

[1] S. H. Son(Editor), Special Issue on Real-Time Database Systems, *ACM SIGMOD Record*, Vol. 17, No. 1, Mar. 1988.

[2] H. Garchia-Molina and K. Salem, "High Performance Transaction Processing with Memory Resident Data," In *Proc. Intl. Workshop on High Performance Transaction Systems*, Dec. 1987.

[3] H. Garcia-Molina and K. Salem, "Main-Memory Database Systems: An Overview," *IEEE Trans. on Knowledge and Data Engineering*, Vol. 4, No. 6, 1992.

[4] T. J. Lehman et al., "An Evaluation of Starburst's Memory Resident Storage Component," *IEEE Trans. on Knowledge and Data Engineering*, Vol. 4, No. 6, Dec. 1992.

[5] J. H. Kim, S. W. Kim, D. Y. Kim, W. Choi, "Implementing a Real-Time Scheduling Daemon in General Purpose Operating System Unix," In *Proc. IEEE Intl. Real-Time Computing Systems and Applications (IEEE RTCSA 2000)*, 2000.

[6] P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[7] K. P. Eswaran et al., "The Notion of Consistency and Predicate Locks in a Database System," *Comm. of the ACM*, Vol. 19, No. 11, Nov. 1976.

[8] J. Gray, R. Lorie, and G. Putzolu, "Granularity of Locks in a Shared Data Base," In *Proc. Intl. Conf. on Very Large Data Bases*, Sept. 1975.

[9] P. Bernstein and N. Goodman, "Timestamp-Based Algorithms for Concurrency Control in Distributed Systems," In *Proc. Intl. Conf. on Very Large Data Bases*, VLDB, 1980.

[10] H. Kung and J. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Trans. on Database Systems*, Vol. 6, No. 2, 1981.

[11] C. Papadimitriou and P. Kanellakis, "On Concurrency Control by Multiple Versions," *ACM Trans. on Database Systems*, Vol. 9, No. 1, 1984

[12] M. Carey and M. Stonebraker, "The Performance of Concurrency Control Algorithms for Database Management Systems," In *Proc. Intl. Conf. on Very Large Data Bases*, VLDB, 1984.

[13] R. Agrawal, M. Carey, and M. Livny, "Models for Studying Concurrency Control Performance: Alternatives and Implications," In *Proc. Intl. Conf. on Management of Data*, ACM SIGMOD, May 1985.

[14] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufman Publishers, 1993.

[15] M. J. Bach, *The Design of the Unix Operating System*, Prentice-Hall, 1980.

[16] C. Mohan et al., "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," *ACM Trans. on Database Systems*, Vol. 17, No. 1, Mar. 1992.