ISSN 1009－3095 *Journal of Zhejiang University*（*SCIENCE*）V.2，No.4，P.416－420，Oct.－Dec.，2001
http://www.periodicals.com.cn；http://www.zju.edu.cn；
http://lib.zju.edu.cn/English；http://www.zjupress.com

A PREDICTABLE MULTI-THREADED MAIN-MEMORY STORAGE MANAGER*

SONG Guang-hua（宋广华），YANG Chang-sheng（杨长生），SHI Jiao-ying（石教英）
（*Department of Computer Science and Engineering*，*Zhejiang University*，*Hangzhou 310027*，*China*）

Received Dec.25，2000；revision accepted Apr.18，2001

Abstract: This paper introduces the design and implementation of a predictable multi-threaded main-memory storage manager（CS20），and emphasizes the database service mediator（DSM），an operation prediction model using exponential averaging. The memory manager，indexing，as well as lock manager in CS20 are also presented briefly. CS20 has been embedded in a mobile telecommunication service system. Practice showed，DSM effectively controls system load and hence improves the real-time characteristics of data accessing.

Key words: main-memory database（MMDB），database service mediator（DSM），hash indexing，lock
Document code: A **CLC number:** TP311

INTRODUCTION

Many database applications，such as telecommunications industry，factory automation，require high performance and real-time access to data. DRDBs（Disk Resident Database），however，are inadequate to meet such requirements because of high disk access overheads for data retrieval and update. MMDBs（Main-memory Database），which are now feasible with the increasing availability of large，cheap memory，can better support these time critical applications（Son，1988；Lehman et al.，1992；Ying et al.，2000）.

Many researches have been made on MMDB. For example，Son et al.（1993）introduced "RTDB"，an MMDB supported by "ARTS"，a real-time operating system kernel. Bohannon et al.（1997）presented "Dali"，former version of "DataBitz"，a main-memory storage manager based on multi-process architecture where database files are shared among processes with file mapping. In the author's another work（Song et al.，1999），"ZEDB"，a main-memory storage manger designed for telecommunication applications，with multi-process architecture is presented in detail. Despite the fact that data access may be much faster when data（or most of the data）are fit in memory，MMDBs still seem inadequate to meet all access requirements when many applications are requesting for service，especially in a client/server environment. Therefore access control is beneficial for maximal compliance with time constraints. However，few researches have been made on access control in MMDBs.

This paper introduces a predictable main-memory storage manager，CS20，which is based on multi-threaded architecture and can limit data access to meet overall data access requirements. The DSM（an operation prediction facility using exponential averaging）is presented and will be discussed in detail. The memory management，data indexing，as well as lock manager in CS20 are also presented in the following sections.

ARCHITECTURE OF CS20

There are three possible ways for building an application program using CS20. One way is to develop a main-memory relational DBMS by providing a relational（or object-oriented）data model and implementing a query processing layer on top of CS20. The second is to let application processes make use of the functionality of CS20 by communicating with the CS20 server process. The third is to provide the functionality of CS20 as a library and to make application programs

* Project（60073026）supported by NSFC.

linked with the library.

Taking the first way, it is easy to make application programs or queries on data because the system provides a general data model that can be understood easily by the users. The second alternative is effective when an application requires simple but fast access to database rather than complex manipulations.

CS20 aims to support all the alternatives. Currently, the second alternative is implemented first because many applications require simple but fast access to the database. Furthermore, it is feasible to develop a main-memory real-time DBMS on top of the CS20 when the second way is running stably.

Fig.1 describes the architecture of the CS20 server process. The server is multi-threaded to execute multiple actions concurrently and to take advantage of multi-processor architecture. When the server process starts up, it creates a pool of threads. A thread is tied to a particular message type.

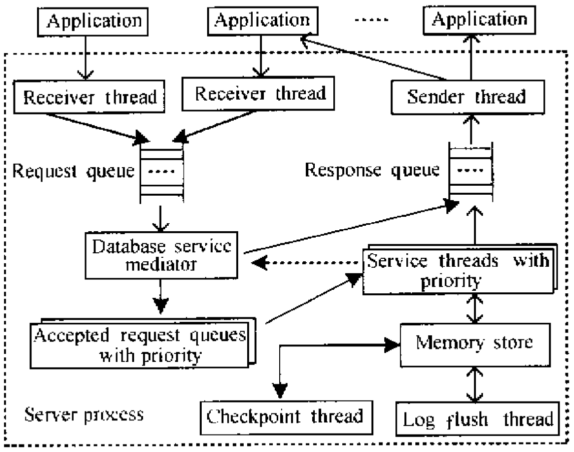

Fig.1 Architecture of CS20

If application processes run on the same machine with the CS20 server process, IPC is employed, otherwise network IPC(TCP/IP sockets) is used.

When a request message arrives, it is submitted to the DSM(the database service mediator, discussed in Section 3). DSM decides whether the required message can be accepted. If it is accepted, DSM delivers the request to an appropriate action queue. A service thread corresponding to that action queue picks the action from the queue and executes the action. When it finishes the action, it places the result on the response queue. The service thread runs until is has to wait for a resource such as lock or semaphore, or is preempted by thread scheduling.

The checkpoint thread and log flush thread deal with I/O and run asynchronously with action service threads. The checkpoint thread moves dirty pages in the primary database to the backup, and the log flush thread flushes the hybrid log (Song et al., 2001) in the system's log buffer in main memory to a nonvolatile log volume.

DSM: THE DATABASE SERVICE MEDIATOR

1. Purpose of DSM

DSM is an action service mediator. When a request message arrives, DSM validates it, extracts its arrival time, its message priority according its message type, its constrained deadline, and then predicts its processing time. If the message is valid and is predicted capable of being executed within the required deadline, it is put into the corresponding action queue, otherwise it is rejected.

2. Prediction of message executing time

When a request message arrives at DSM, its executing time is predicted. A message is supposed to be executable if its allowed execution time is within the predicted execution time. The executing time of the next message with priority i can be predicted with the observation of real executing time of previous messages with the same priority. Here, exponential averaging is exploited:

$$PT_{i(n+1)} = \alpha \cdot RT_{in} + (1-\alpha) \cdot PT_{i(n-1)} \qquad (1)$$

where PT_{ij} is the predicted executing time of the jth message with priority i. RT_{ij} is the real executing time of the jth message with priority i. α is a constant weighting factor $(0 < \alpha < 1)$ that determines the relative weight given to more recent and less recent observations. PT_1 is initialized as 0 as no previous message has been executed.

The expansion of equation (1) is:

$$PT_{i(n+1)} = \alpha \cdot RT_{in} + (1-\alpha) \cdot RT_{i(n-1)} + \cdots + (1-\alpha) \cdot T_{i(n-1)} + \cdots + (1-\alpha)^{n-1} \cdot$$

$$\alpha \cdot T_{i1} + (1-\alpha)^n \cdot PT_1 \qquad (2)$$

Because both α and (1-α) are less than 1, each successive term in Eq.(2) is smaller. The older the observation, the less it contributes to the average. Larger value of α results in a more rapid reaction to the change in the observed value.

When a message arrives, its start time, TS, is obtained. Let TC be the current time, then

$$TL = TC - TS$$

is supposed to be the latency of message transportation. Assume that the same latency is required to transport the reply message back to the application. This implies that the maximum (allowed) execution time is:

$$MT = DL - 2 * TL = DL - 2 * (TC - TS),$$

where DL is the deadline of the message. Description of prediction is as follows:

```
    /* $MT_i$ is the allowed maximum execution time of next message with priority $i$, $PT_i$ is the predicted execution time of next message with priority $i$ */.
When a message is received:
{
    if ($MT_i < PT_i$) {
        Abandon the message and acknowledge the application;
        Set $PT_i = 0$;
    }
    else {
        Accept the message;
        Set $TC$ = current time;
        Convert the message into action of the database.
        Put the action as well as $TC$ to the corresponding action queue.
    }
}
When execution of the message is finished, $PT_i$ is adjusted:
{
    Set $FT$ = current time, $FC$ is supposed to be the finish time of the message;
    Set $RT_i = FT - TC$, $RT_i$ is the execution time of the message;
    $PT_i$ is adjusted:
        $PT_i = \alpha \cdot RT_i + (1-\alpha) \cdot PT_i$
}
```

In DSM, after a message is rejected, PT_i is reset, making the successive message accepted. Hence it avoids abandoning a sequence of messages after a series of time-consuming actions are executed.

MEMORY MANAGEMENT WITH PARTITIONS

1. Memory partitioning

User data in CS20 are stored in partitions. A partition is a fixed-length contiguous memory block, which can store a definite number of user tuples. Let the system contain N partitions, with P_i the ith partition. The data set in P_i is presented as DP_i. Then the total data set in CS20 (DU) is:

$D_U = \bigcup_{i=1}^{N} D_{P_i}$, and $D_{P_i} = \Phi$, for all i, j such that $1 \leqslant i, j \leqslant N$ and $i \neq j$.

The partition information in CS20 are maintained in the PDT(partition directory table).

2. User data indexing

User data indexing in CS20 is implemented as2-level separated hashing.

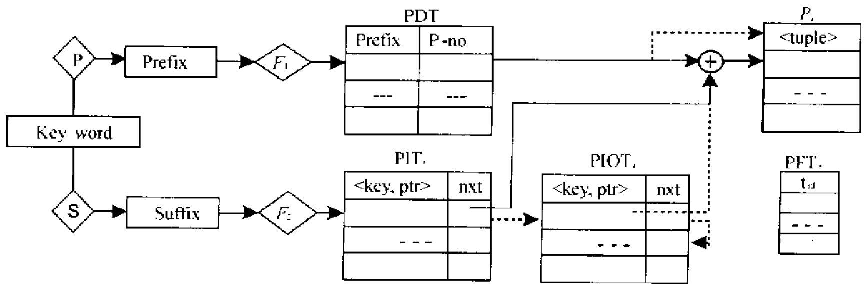

Fig.2 Partition-based 2-level hashing

Let K be the key word of a user tuple, with its length L_K. The prefix of K is the most significant of L_P bits, denoted as $P(K)$; The suffix of$K$ is the least significant of L_S bits, denoted as $S(K)$. Here $L_P = f(L_K)$, $L_S = g(L_K)$, and $L_P + L_S \leqslant L_K$, f and g are a priori. Every partition has a PFT(partition free tuple list table), a PIT(partition indexing table)and a PIOT(partition index overflow table). The first level hashing function, F_1, is used to index a partition where user tuple will be fit. While the second hashing function, F_2, is exploited to index a tuple in the partition.

LOCK MANAGER

The Lock Manager in CS20 is a thread to manager locks. CS20 exploits one granule of lock: record level lock. Hence it improves the level of parallelism. Deadlock is avoided because "priority abort" and "deadlock detection" is employed in the lock manager.

When a transaction wants to aquire a lock on a tuple that is already locked by another transaction, conflict occurs. If the former has higher priority over the latter, the latter is aborted. If the former has lower priority than the latter, it is aborted. If two transactions have equal priority, deadlock detection is made. If deadlock occurs, the former aborts, otherwise, it is blocked until the lock is released.

Deadlock detection in CS20 exploits the checking of circular wait. If circular wait exists when the lock requirement is granted, deadlock may occur, otherwise, deadlock may not occur.

Every partition includes a partition lock table (PLT) which maintains every lock in the partition. Each lock is a tuple < TID, TransId, LockCount > , where TID is the tuple-ID(or key) of the tuple, TransId is the identifier of the transaction that owns the lock. LockCount is a counter that reflects the relative elapsed time since the lock was made. Lock Manager periodically scans every PLT of the partitions, increments every LockCount by 1. If LockCount reaches to a pre-determined limit, it is supposed that the lock is too old. Unlock it, and abort the transaction.

PERFORMANCE ANALYSIS OF DSM

The performance of CS20 with and without DSM has been tested in HLR subsystem, most important in the mobile telecommunication service system where frequent access to subscriber information is required. The test was carried out on a HP LH3 server, with 1 Pentium-Ⅲ processor of 500 MHZ, and 256 MB main memory running Windows NT4.0. In the test, 30 thousand pieces of mobile subscriber profile information were stored, each occupying 256 bytes of memory space. The experiment was implemented this way: each test contained 100 000 lookup-and-modify operations, the most frequent operation type in HLR. Each test time value was the mean of the 100 000 operation time measurements. Eight groups of test results were recorded. Each group corresponded to the number of application processes that invoked such operations, ranging from 1 to 8, respectively. Table 1 compares time measurements of operation, with and without DSM. Fig.3 compares the operation miss rate, i.e., the rate of the operations that miss their deadline constraints compared to the accepted operations, with and without DSM. Fig.4 is the abandon rate when using DSM, i.e., the rate of the operations that are abandoned since they are

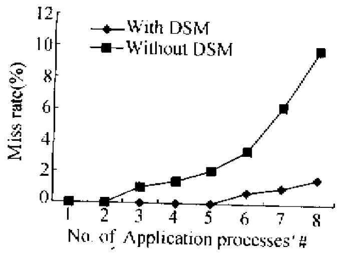

Fig.3 Miss rate with and without DSM

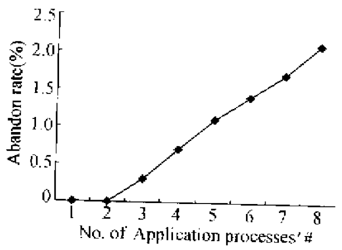

Fig.4 Abandon rate with DSM

predicted to be possible of missing their deadline constraints. In the experiment, the constant α was 0.5, and messages had same time constraints in each case.

Table 1 Time measurements of 100 000 Lookup-and-Modify operations

Number of application processes	Without DSM					With DSM						
	Access time (μs) of an operation			Missed operations		Access time (μs) of an operation			Missed operations		Abandoned operations	
	Min.	Max.	Avg.	Total	%	Min.	Max.	Avg.	Total	%	Total	%
1	807	4641	2722	0	0	918	4652	2766	0	0	0	0
2	926	4731	2796	0	0	940	4869	2890	0	0	0	0
3	1139	6210	3210	995	1	1025	5420	2930	0	0	305	0.3
4	1320	6630	3652	1528	1.5	1102	5530	2988	0	0	690	0.7
5	1526	7012	4020	2216	2.2	1325	5730	3100	0	0	1100	1.1
6	1728	8320	4520	3630	3.6	1520	6110	3410	812	0.8	1362	1.4
7	2130	12020	5420	6140	6.4	1722	6214	3632	1140	1.1	1732	1.7
8	2530	15232	6210	9849	9.8	2122	6320	3787	1627	1.6	2184	2.2

The experiment showed that, with the increasing of application processes, the miss rate with DSM is much lower than that without DSM, while the abandon rate was acceptable. It is valuable sacrificing a small percentage of operations to gain overall performance as a real-time application database supporter.

CONCLUSIONS

This paper presents CS20, focusing on its ability to predict message execution time using exponential averaging. The advantage of prediction is distinct when many applications access the database.

CS20 has been embedded in a telecommunication application system and has proved its good access performance over conventional DBMSs. However, CS20 can only support simple data operations. Many sophisticated database operations, such as projection and join, have not been achieved. It is our major work currently to implement a query processing layer on top of CS20 to support those complicated operations and provide better user interface.

References

Bohannon, P., Lieuwen, D., Rastogi, R., 1997. The Architecture of the Dali Main-Memory Storage Manager. *Bell Labs Tech. J.*, **2**(1):36 – 47.

Lehman, T., Shekita, E.J., Cabrera, E.J., 1992. An evaluation of Starburst's Memory-Resident Storage Component. *IEEE Trans. On Knowledge and Engineering*, **4**(6):555 – 566.

Song, G. H., Yang, C. S., 2001. Recovery subsystem in main-memory database based on hybrid logging. *Journal of Zhejiang University*(science edition), **28**(2):164 – 168(in Chinese).

Song, G.H., Yang, C.S., Shi, J.Y., 1999. ZEDB: a main-memory database system for real-time message processing applications. Proc. of the 6th intl. conf. on CAD&CG(CAD/CG'99), Shanghai, P. R. China, p. 238 – 242.

Son, S.H., 1988. ACM SIGMOD Record 17, 1. Special Issue on Real-Time Database Systems.

Son, S.H., George, D.W., Kim, Y.K., 1993. Developing a Database System for Time-Critical Applications. IEEE Intl. Conf. on Database and Expert Systems Applications(DEXA'93), Prague, Czech Republic, Lecture Notes in Computer Science #720, Springer-Verlag, p. 313 – 324.

Ying, J., He, Z. J., Wu, M. H., 2000. Evolutionbased software developing environment. *Journal of Zhejiang University*(*SCIENCE*), **1**(4):381 – 383.