http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 6 (6): 872-878, 2007
ISSN 1812-5638

SQLmmdb: An Embedded Main Memory Database Management System[cc]

YanLiang Cui and Dechang Pi
College of Information Science and Technology,
Nanjing University of Aeronautics and Astronautics, Nanjing, Jiangsu, 210016, China

Abstract: The embedded database system is generally used in the equipments provided with self-control ability and always deals with real time transaction. It is particularly urgent to establish embedded Real Time Database System (RTDBS) with the technique of Main Memory Database (MMDB). In this study, the design strategy of an embedded main memory database SQLmmdb is presented. We propose an improved index structure on the base of T* tree (called T** tree) and design a non-log recovery method which is fit for MMDB. And we also compared SQLmmdb on performance with the embedded DBMS SQLite which is now widely used in many applications. The experiments show that the speed of SQLmmdb is accelerated a lot, at the same time the space occupied by the database files reduced remarkably. SQLmmdb is a high performance main memory DBMS model which is suitable for embedded system.

Key words: Main memory database, embedded system, T tree, index

INTRODUCTION

In the recent years, as the development of the embedded system applications, the problem of data management in embedded system came to be the concern. The growing amount of data requires a safe and useful DBMS to control. Speak of the embedded database system, there is already some open source products abroad such as Berkeley DB (http://www.sleepycat.com) and SQLite (http://www.sqlite.com). Embedded database system often deals with real time transactions. Though the present products have an outstanding speed, they haven't met the real time demand. So the problem of how to establish an embedded real time database system becomes more and more important for the embedded system.

Researches on MMDB began from 1980, as a purpose to satisfy the increasing demand of real time data management. Now, a series of theory and products appear, such as FastDB (http://www.garret.ru/~knizhnik/fastdb/FastDB.htm), Dali (Jagadish *et al*., 1994; Philip *et al*., 1997) from AT&T Bell lab, TimesTen from Oracle (http://www.oracle.com/ timesten/index.htm), which is in widely used in many applications such as HP intellect web flat already, Cisco VoIP call Proxy, the telecom system of Alca2tel and Ericsson and so on.

Embedded real time database system is indeed a Main Memory Database (MMDB) with the purpose of high reliability, high real time, high quantity of information throughput (Jagadish *et al*., 1994). SQLite is widely used in the embedded domain for its non-copyright, zero configuration and good performance. As the official website introduces (http://www.sqlite.org/speed.html), it has more performance advantages in many aspects than MySQL and PostgreSQL.

SQLmmdb

System structure of SQLmmdb: Layered structure that SQLmmdb uses (Fig. 1) is not only good for parallel developing of the project, but also convenient for debugging and testing. We provide SQLmmdb in library form to client while client providing SQL statements to the

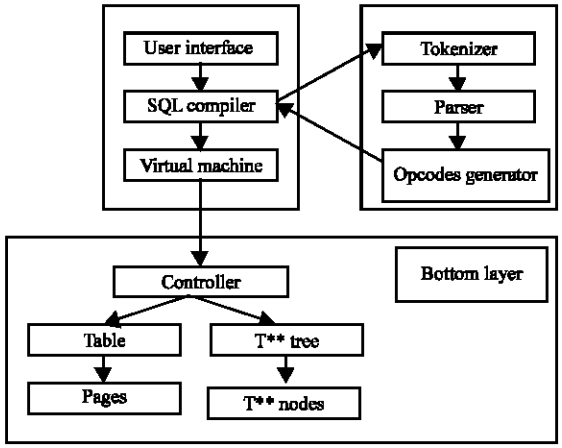

Fig. 1: System Structure of SQLmmdb

Corresponding Author: Dechang Pi, College of Information Science and Technology,
Nanjing University of Aeronautics and Astronautics, Nanjing, Jiangsu, 210016, China

interface. Then the SQL statements will be compiled to opcode sequences which can be executed by SQLmmdb virtual machine. SQLmmdb virtual machine executes the opcodes made by compiler module, saves temporary results by stack and operates the database by invoking the bottom layer interfaces. The controller of bottom layer interface manipulates the tables and indices and provides interfaces to the virtual machine to operate database.

T tree index structure:** T* tree (Kong-Rim and Kyung-Chang, 1996) is an improved T tree (Tobin and Carey, 1986) and avoids the tree traversal when scanning successor nodes. But T* tree still need tree traversal to get predecessor nodes. In SQLmmdb, we propose a new index structure called T** tree in order to solve the problems in the T* tree for better performance. A predecessor pointer which directly points to the predecessor node is added to every node (Fig. 2). A tail node pointer is added to the whole tree (Fig. 3). Its advantages are as follows:

- With the predecessor pointer and the successor pointer of every node, T** tree has the ability of bidirectional scan. Ascending and descending ordering traversal can be done in the same index. Index is always stored in ascending ordering mode whatever you using ascending or descending ordering index. After the designating index value is found when doing range queries on the index which has several same values, we can discretionarily locate to the first or the last one of the designating value. In the T* tree, if several nodes contain a same index value, getting predecessor node must use tree traversal (means traversing to the predecessor or successor node in a tree, which is different from the ordinal traversal of index) when locating the first one of the same index value. T** tree can save much time by getting predecessor node through the predecessor pointer.
- T** tree has the ability of bidirectional sorting for it contains a head pointer and a tail pointer. So it is convenient to do ascending or descending sorting of tuples. The ascending sorting starts from the head node, using the ascending ordering traversal to get each index item. The descending sorting starts from the tail node and uses the descending ordering traversal.

In the range query whose lower bound is $+\infty$ in ascending ordering index or upper bound is $+\infty$ in descending ordering index, T** tree is more efficient than T* tree for its bidirectional scan ability. Take the search of items with the column value less than 60 in an ascending order index as an example, for T* tree, one way is to start from the head node, then do ascending ordering traversal and check each index item whether it is less than 60 until getting a index item whose value is not less than 60. Another way is to locate the last one of the index items whose value is less than 60 (also need tree traversal), then do descending ordering traversal until getting the least index item, which needs tree traversal when you want to get the predecessor node. But without tree traversal, T** tree can get predecessor node directly by using predecessor pointer. Therefore it can fast locate the last one of the index items whose value is less than 60 and do descending ordering traversal till the least item. Comparing with the first way of T* tree, it avoids a large number of comparison. Comparing with the second way

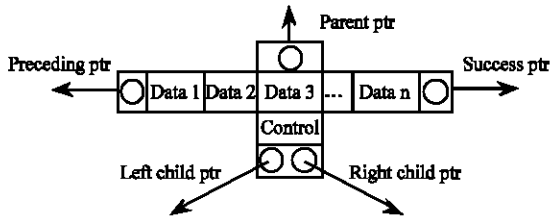

Fig. 2: T** tree node

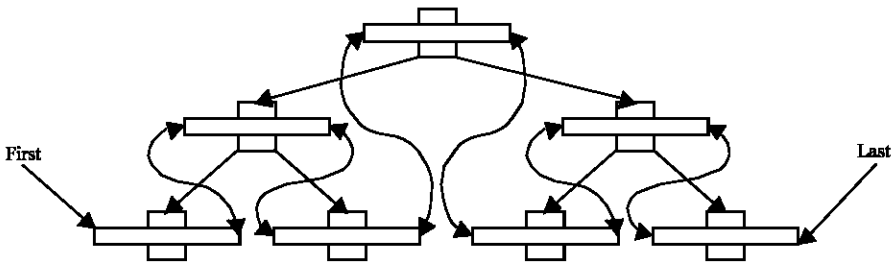

Fig. 3: T** tree structure

of T* tree, it avoids the tree traversal. As a result, SQLmmdb really saves much time by using T** tree.

Besides, as an index structure compared with T* tree, T** tree can create indices always in the same mode. It simplifies the opcodes of virtual machine that deal with the index. And without the tree traversal, the indexing task for the code generator can be reduced. In a word, using T** tree as the index structure simplifies the design of several module and increases the efficiency.

Compact table storage mechanism: The requirements for a main memory data storage model such as DBGraph (Philippe *et al*., 1990) are both compactness and efficient processing (Philippe and Thevenin, 1992) for all database operations. In SQLmmdb, the compact table storage mechanism also achieves these goals.

SQLmmdb provides three systematic tables, named table_master, column_master and index_master. They store the information about tables, columns and indices respectively. Each table is stored in the form of pages which are connected by pointer. The page size is defined by a macro PAGE_SIZE. There is a page head (containing control information) on each page which contains the predecessor and successor page number, amount of record in this page, offset of the first free place and the serial number of this page in the table. Each record has a RowID started from zero which means the serial number of this record in the table. It is easy and fast to locate the record, for the length of every record is changeless. System can easily get the record address using ROWID.

```
nPage = nRowID/nMaxRecPerPage;
nOff = nRowID MOD nMaxRecPerPage;
pRecord=tbAddr[nPage]->aData+PAGE_HEADER_SIZE
+nOff * nRowSize;
```

nMaxRecPerPage is the max capacity of a record on every page. nPage shows which page current record is on. nOff means the record number on this page. tbAddr is a map from the serial number of page to a data page. aData is the base address of data on a page. PAGE_HEADER_SIZE means the size of a page header. nRowsize notes how many bytes of each record.

First, from nPage by page map tbAddr, get the data address of the page which contains the record that we want to locate. Second, adding the data address to PAGE_HEADER_SIZE, we will get the base address of data on this page, together with the record offset which is calculated by nOff * nRowSize we will get the record address accurately.

Figure 4 describes the storage structure of a table in SQLmmdb. There is a bitmap in the head part of a record. The first flag bit of the bitmap shows whether there is a record behind or not. If it is 1, every bit behind shows whether the corresponding column is null. It makes the search with is null or not null clause very convenient. If first bit is 0, there is no record behind in this place.

All the empty places on a page make up a free list. nEmptyoffset in a page header shows the offset of the first empty place, namely the head of the free list. PREV_EMPTY is the offset of the previous empty place

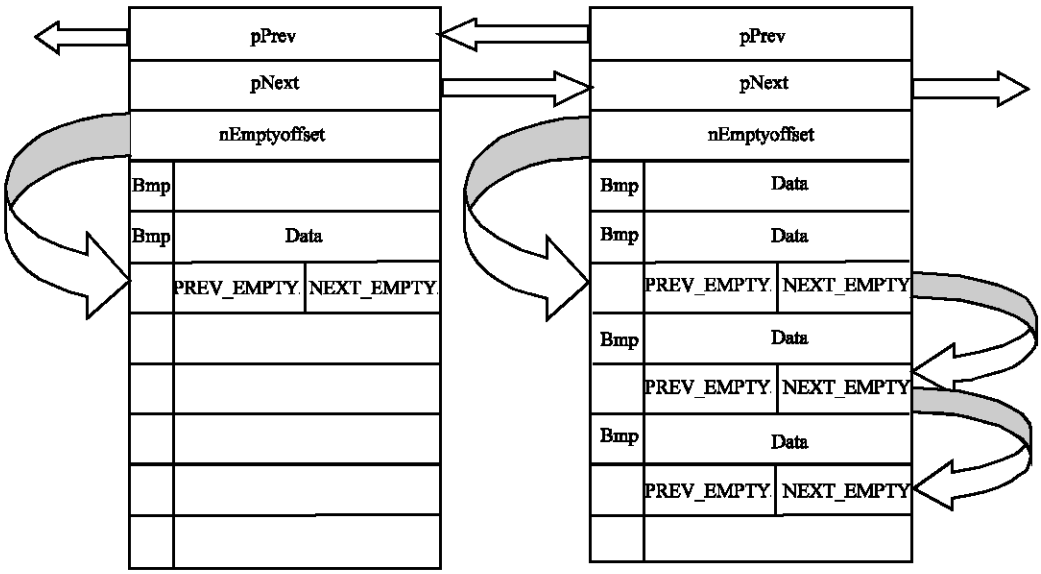

Fig. 4: The structure of data table in SQLmmdb

and NEXT_EMPTY means the next. When inserting a record, get the offset of the first empty place from nEmptyOffset and make it equal to NEXT_EMPTY, namely the next empty place offset.

To show this place has no record anymore, the first bit of current record bitmap should be reset to zero when deleting a record. After that, the place of this deleted record will be put into the head of the free list. This means the PREV_EMPTY of the place where the nEmptyoffset pointed to is assigned the offset of the deleted record. In the deleted record, PREV_EMPTY is assigned the value of zero (means it is the head of the free list) and NEXT_EMPTY is assigned the former value of nEmptyoffset. nEmptyoffset is assigned the offset of the deleted record. The page will be deleted from the table if the amount of record is zero after record deleting. For this reason, all the operations just deal with the link-list. So the insert and delete operations are very efficiency which is main memory database most need.

The length-alterable field such as string in SQLmmdb is stored by T** tree with FieldID which is the serial number of the field value in T** tree as its index item, namely the key. The data item is made up of the field value, the length of value and the reference count. The pointer to the length-alterable field in records actually stores the FieldID of the corresponding item of T** tree. The same value from different records will be stored in the same copy. By using the Copy-On-Write technology, when a record is modified, the new value will be inserted into the T** tree and the reference count of the old value will be subtracted from 1. The old value will be deleted only if the reference count is equal to zero. The advantages that only the corresponding FieldID in the T** tree is stored for the length-alterable field in the actual records are as follows.

- Keep a changeless length of a record will lead to locate a record quickly and efficiently.
- For each value with Copy-On-Write technology contains a reference count and T** tree data is compact, just the minimum needed space is allocated for the length-alterable data.
- Because of the high scan efficiency of T** tree, we can locate the actual data rapidly by FieldID stored in records.

Non-log recovery method: In the condition of no stable memory and no special recovery processor for the embedded system, SQLmmdb use a non-log recovery method. This method comes from the Ping-Pong check-point (Philip *et al*., 1997). SQLmmdb stores two database copies on disk and renew them alternately. The system also records which one is the latest. If the system crashes, there will always be a database copy in the state of consistency.

The data table in SQLmmdb is memory resident while database on disk is just a copy. It only needs to read the old value when roll-back and write the new value when committing. So it is unnecessary to write log in transaction processing. Every page in memory has a corresponding one in the database file on disk. It has two important flags, bDirty and bsecDrity. If a page is modified, the bDirty will be true. If transaction fails and rolls back, the corresponding page in the disk copy will be read and overwrite the pages where bDirty is true in memory. If transaction is committed then it writes the pages where bDirty or bsecDirty is true to a disk copy. And copy all the bDirty in these pages to bsecDirty. Write dirty pages to another disk copy when committing next time.

The disadvantages of this non-log recovery method is obvious, it enlarges the space which the database file occupied for two database copies on disk. It also affects the speed for it has to read disk when roll-back. But compare with the disadvantages, the advantages of this method is more outstanding.

- The work of dealing with log and checkpoint is avoided. Compared with the log-based recovery method (Tobin and Carey, 1987; Yonggui and Juwei, 2003; Yingyuan *et al*., 2005), the disk writing operation is not decreased (Two times every page, the first time is to write the current data (redo log) or original data (undo log) of dirty page to log. The second time is to write pages modified by committed transactions to the database file). The efficiency of transaction execution rises sharply, because it avoids log reading and analyzing when doing checkpoint. Especially, in the application of the transactions with high successful rate, SQLmmdb reduces the I/O operation to minimum (Write disk only when transaction is committed).
- Avoid memory allocation and data copy which the shadow-memory method has to do and so the speed accelerates.
- Whenever restarting system for crash, a database copy is always in the state of consistency on disk. SQLmmdb only need to read this copy into memory to accomplish the recovery and does not need log-analyzing and the recovery efficiency rises.

EXPERIMENTS

Test on T** Tree

Test contents: The test platform was a personal computer with a 1.833GHz Athlon CPU, 512M Memory and Windows XP operating system.

For T** tree uses the same algorithms in insert, delete and search operation, we did not test on these operations. The purpose we improved T* tree is to accelerate scanning predecessor nodes so as to accelerate the range queries whose lower bound is ∞ in ascending ordering index or upper bound is ∞ in descending ordering index. In order to prove the performance improvement of T** tree, we compared T** tree with T* tree on the performance of range queries whose lower bound is ∞ in ascending ordering index.

We divided the 1000 times range queries test into two types, small data volume that each tree had 0.1M integer index items and large data volume that each one had 1M. All the upper bounds of range queries distributed evenly throughout the index data space. We tested the T** tree algorithm and two algorithms of T* tree with the 128b, 1 and 16 kb tree node, respectively.

Test results: Every test result is an average of ten same tests. Operation time measured by second (s) is given in every test.

From Table 1 and 2, we can see that T** tree was more effective than T* tree in the range queries whose lower bound is ∞ in ascending ordering index or upper bound is 8 in descending ordering index.

Test on SQLmmdb performance

Test contents: We compared SQLmmdb with the famous embedded DBMS SQLite.

The tests consisted of three major tasks: data insertion, search and index creation. We divided the transaction test into two types, small data volume that 50,000 records and large data volume that 500,000 records were inserted at one time, respectively. Index was created in ascending order. Insert operations were divided into three types:

- Insert into a table without transaction.
- Insert into a non-indexed table in a transaction.
- Insert into an indexed table in a transaction.

Search operations were made up of short range query and the range query whose lower bound is ∞. A short range query spanned two index values. Take integer index as an example, one of the spans was. All the spans of short range queries and the upper bounds of range queries whose lower bound is ∞ distributed evenly throughout the table data space. The tests about search were divided into four types:

- Execute 100 short range queries on a non-indexed table in a transaction.
- Execute 1000 short range queries on an indexed table in a transaction.
- Execute 100 range queries whose lower bound is ∞ on a non-indexed table in a transaction.
- Execute 100 range queries whose lower bound is ∞ on an indexed table in a transaction.

The following three tables have been tested, respectively:

- Create table TestINT (one int). All tests have been done on this table.
- Create table TestFLOAT (one float). Insert and search tests have been done on this non-indexed table.
- Create table TestSTRING (one char (10)). Insertion and search tests have been done on this non-indexed table. Because SQLmmdb doesn't support string-part-matching operation, we only did the string equivalent search.

Test results: Like the test on T** tree, every test result is also an average of ten same tests and operation time is measured by second (s). If the size of database changed after operation, then the increment of database file measured by MB (short to M) will be given.

It took SQLmmdb 0.015 second to do 1000 non-transaction insert operations on an integer table. The increment of database was 0.028M. While it took SQLite 61.125 second and the increment was 0.029M.

The following shortenings are used for describing clearly Table 3-6.

Table 1: Range query test with small data volume

Node size	T** tree algorithm	1st T* tree algorithm	2nd T* tree algorithm
128b	0.849s	1.182s	0.932s
1kb	0.343s	0.578s	0.401s
16kb	0.338s	0.525s	0.370s

Table 2: Range query test with large data volume

Node size	T** tree algorithm	1st T* tree algorithm	2nd T* tree algorithm
128b	8.727s	11.890s	9.890s
1kb	4.041s	5.911s	4.245s
16kb	3.305s	5.609s	3.586s

Table 3: Small data volume test on an integer table in a transaction

Test items	SQLmmdb	SQLite
Execute 50,000 insert operations on a non-indexed table	0.550 s/0.26M	1.029 s/1.16M
Execute 100 short range queries on Table A	2.104 s	2.614 s
Execute 100 range queries whose lower bound is ∞ on Table A	1.931 s	2.568 s
Create an index on the table which contains 50,000 records	0.048 s/0.396M	0.388 s/1.17M
Execute 100 short range queries on Table C	0.046 s	0.064 s
Execute 100 range queries whose lower bound is ∞ on Table C	1.420 s	4.482 s
Execute 50,000 insert operations on an indexed table	0.645 s/0.604M	2.335 s/2.33M
Execute 1000 short range queries on an indexed table which contains 100,000 records	0.043 s	0.078 s

Table 4: Large data volume test on an integer table in a transaction

Test items	SQLmmdb	SQLite
Execute 50,000 insert operations on a non-indexed table	5.698 s/2.41M	10.912 s/13.1M
Execute 100 short range queries on Table B	20.591 s	33.654 s
Execute 100 range queries whose lower bound is ∞ on Table B	18.939 s	34.448 s
Create an index on the table which contains 500,000 records	0.493 s/3.85M	4.189 s /13.1M
Execute 1000 short range queries on Table D	0.048 s	0.071 s
Execute 100 range queries whose lower bound is ∞ on Table D	14.541 s	57.886 s
Execute 500,000 insert operations on indexed table which contains an indexed table	6.701 s/6.24M	21.631 s/26.5M
Execute 1000 short range queries onanindexed table which contains 1000,000 records	0.046 s	0.088 s

Table 5: Test on a floating point table in a transaction

Test items	SQLmmdb	SQLite
Execute 50,000 insert operations on a non-indexed table	0.726 s/0.456M	1.167 s/1.57M
Execute 100 short range queries on Table A	2.196 s	5.711 s
Execute 500,000 insert operations on a non-indexed table	7.890 s/ 4.33M	11.240 s/5.7M
Execute 100 short range queries on Table B	21.578 s	67.895 s

Table 6: Test on a string table in a transaction

Test items	SQLmmdb	SQLite
Execute 50,000 insert operations on a non-indexed table	0.531 s/1.02M	0.975 s/1.36M
Execute 100 string equivalent searches on Table A	2.824 s	2.882 s
Execute 500,000 insert operations on a non-indexed table	6.250 s/10.0M	9.414 s/13.5M
Execute 100 string equivalent searches on Table B	28.742 s	36.218 s

Table A: a non-indexed table which contains 50,000 records.

Table B: a non-indexed table which contains 500,000 records

Table C: an indexed table which contains 50,000 records

Table D: an indexed table which contains 500,000 records

It took SQLmmdb 0.016 second to do 1000 non-transaction insert operations on a floating table. The increment of database was 0.024M. While it took SQLite 63.25 second and the increment was 0.034M.

Analysis: From the experiments, we can see that SQLmmdb always result in better outcomes than SQLite.

- On non-transaction insert operations, the consuming time of SQLmmdb was less than one thousand of that of SQlite.
- Comparing the insert operation on a non-indexed table in a transaction, SQLmmdb was 40% time faster than SQLite. Meanwhile, SQLmmdb only used about one third time of SQLite for an indexed-table insertion. SQLmmdb only needs pointer adjustments and data copy on bytes without any conversion; but SQLite has to convert all the data into character string for storage. At the same time, SQLmmdb uses T** tree as its index, while SQLite uses B tree. Because T tree has a higher insertion efficient than B tree (Tobin and Carey, 1986; Garcia Molina and Salem, 1992) and T** tree inherits form T tree, data insertion of T** tree is more efficient than B tree.
- The time that SQLmmdb consumed to create an index was one ninth of SQLite's. T** tree is a compact index structure that each node only has five pointer and two child nodes. At the same time, in a B tree node, the sum of pointers is one less than the sum of index items and each node has several child nodes. So it is more complex to use B tree to create an index.
- Compared with SQLite, SQLmmdb did not accelerate much in short range queries on a non-indexed table with small data volume. But as the data volume grew, the speed advantage of SQLmmdb gradually appeared. With large data volume, the time of short range queries SQLmmdb used was two third of SQLite's. SQLmmdb consumed half the time of SQLite's on the range queries whose lower bound is ∞. That is because SQLite has to do lots of I/O operations to swap the pages between buffer and disk. However, the data of SQLmmdb is memory resident so as to avoid the I/O operations. Also, SQLmmdb uses the compact table storage mechanism in order to locate records quickly. In addition, range queries of floating point with large data volume spent only one third of SQLite's. The reason is that, SQLite has to convert all the data into character string for storage and compare the strings converted from the floating point numbers, which is more time-consuming than comparing the floating point numbers directly. Therefore, it is obvious that SQLmmdb has a large advantage in floating queries.

- SQLmmdb was a little bit faster than SQLite when doing short range queries on an indexed table. Because the time of searching a single value of T** tree and B tree are both $O(\log_2 N)$. But T** tree avoids the tree traversal which B tree has to do for range queries. And also SQLmmdb needs no I/O operations for search. For all the reasons above, SQLmmdb is better than SQLite. As a preponderant item of T** tree, in the range queries whose lower bound is ∞, SQLmmdb spent one third time of SQLite with small data volume and one forth with large.
- SQLmmdb also had great advantages in the space that the database file occupied. That is because the structures of table and index of SQLmmdb are both much more compact than that of SQLite.

In a word, compared with SQLite, SQLmmdb improved a lot on operation speed. And the space that database file occupied decreased sharply. So SQLmmdb is a high performanced MMDB.

CONCLUSIONS

In this study, we described the system structure, index structure, table storage mechanism and recovery method of an embedded main memory database system, SQLmmdb. Base on T* tree, we proposed a new improved index structure called T** tree. It added a predecessor pointer which points directly to the predecessor node to each T* node and a tail pointer to the whole T* tree. In the range query whose lower bound is ∞ in ascending ordering index or upper bound is ∞ in descending ordering index, we can directly get the predecessor node by the predecessor pointer. Therefore, the T** tree saves the T* tree's many time on comparisons or tree traversal. It also saves the searching time and has higher searching efficiency than the T* tree.

Besides, we designed and implemented a non-log recovery method which is fit for MMDB. It avoids many log reading and analyzing works while doing checkpoint. And it's unnecessary to deal with the log while recovering system. So the efficiency of a transaction executing and system recovery is successfully rises.

At last, we compared SQLmmdb on performance with a famous embedded DBMS SQLite. The experiments show that, compared with SQLite, SQLmmdb improved a lot on operation speed and the space database file occupied decreased sharply.

At present SQLmmdb is only a single-user supporting DBMS. In future research, in order to make SQLmmdb a highly efficient and multi-user supporting main memory DBMS, we will strengthen the functions include crossing-platform, lock mechanism and concurrency control.

REFERENCES

Garcia-Molina and K. Salem, 1992. An main-memory database systems: An overview. IEEE Trans. Knowledge Data Eng., 4: 509-516.

Jagadish, H.V., D.F. Lieuwen, R. Rastogi, A. Silberschatz and S. Sudarshan, 1994. Dali: A High Performance Main Memory Storage Manager. In: Proceedings of the 20th International Conference on Very Large Data Bases. San Francisco: Morgan Kaufmann Publishers Inc., pp: 48-59.

Kong-Rim, C. and K. Kyung-Chang, 1996. T*-tree, A Main Memory Database Index Structure for Real Time Applications. Proceedings of the 3rd International Workshop on Real-Time Computing Systems Application (RTCSA '96). Washington, DC: IEEE Computer Society, pp: 81-88.

Philip, B., D. Lieuwen, R. Rastogoi, A. Silberchatz, S. Seshadri and S. Sudarshan, 1997. The Architecture of the Dalí Main-Memory Storage Manager. Multimedia Tools Appl., 4: 115-151.

Philippe, P., J. Thévenin and P. Valduriez, 1990. Efficient main memory data management using the DBGraph storage model. Proceedings of the 16th International Conference on Very Large Data Bases. San Francisco: Morgan Kaufmann Publishers Inc., pp: 683-695.

Philippe, P. and J. Thevenin, 1992. Pipelined query processing in the DBGraph storage model. Proceedings of the 3rd International Conference on Extending Database Technology. London: Springer-Verlag, pp: 516-533.

Tobin, J.L. and M.J. Carey, 1986. A study of index structures for main memory database management systems. Proceedings of the 12th International Conference on Very Large Data Bases. San Francisco: Morgan Kaufmann Publishers Inc., pp: 294-303.

Tobin, J.L. and M.J. Carey, 1987. A recovery algorithm for a high-performance memory-resident database system. Proceedings of the 1987 ACM SIGMOD International Conference on Management of data. New York: ACM Press, pp: 104-117.

Yingyuan, X., L. Yunsheng, L. Guoqiong and L. Ping, 2005. An Efficient Crash Recovery Technique for Real-Time Main Memory Database. Wuhan University J. Nat. Sci., 10: 61-64.

Yonggui, Z. and G. Junwei, 2003. The multi level recovery of main memory real time database systems with ECBH. J. China Univ. Posts Telecommun., 10: 15-24.