Advanced Engineering Forum *Online: 2012-09-26*
ISSN: 2234-991X, Vols. 6-7, pp 58-63
doi:10.4028/www.scientific.net/AEF.6-7.58
© 2012 The Author(s). Published by Trans Tech Publications Ltd, Switzerland.

An Optimized Method for Access of LOBs in Database Management Systems

Liang Junjie[1, a], Zhan Wei[2,b]

[1]Hubei UniversityFaculty of Mathematics & Computer Science Wuhan 430062, China

[2]Information and Communication Branch of Hubei Electric Power Company, Wuhan 430077, China

[a]Ljjhubu@163.com, [b]carloszhan@163.com

Keywords: large object, log, database management system

Abstract. This paper researches methods used by relational database management systems to manage LOBs. It attempts to use LOB slave table methods to optimize and advance the storage structures of large objects as well as optimize the redo and undo log mechanisms to increase LOB access efficiency. For changes in slave table data, it effectively prevents the redo log from being written into the LOB database, while diminishing the amount of changes being recording within the undo journal. This improves the writing speed of the log, thus improving the access efficiency and speed of the slave table record. At the same time, this optimized log mechanism also ensures the correct recovery of data after a system crash. Tests have shown that the optimized LOB access efficiency is slightly better than Oracle's.

Introduction

In database systems, large objects (hereafter referred to as LOBs) often refer to large-volume data which has a byte usage in the KB, MB, or even GB range. For DBMS, this data is unstructured. They are just simple bitflows, which are temporarily stored and extracted. The pictures, videos, audio, and other large data texts and files that we encounter in our daily lives can be categorized as LOBs.

The research found in this paper is based off of the author's design and development of the JBASE original database management system. Comparative analysis has been performed between the Oracle9i database management system and the optimized system efficiency in order to provide more advanced optimization methods for the management capabilities the JBASE provides for LOB data.

The Original LOB Access Method

LOB Storage Organization Methods. LOB data file types supported by Oracle include BLOB, CLOB, NCLOB, and BFILE, which provides users with storage and operation of large unstructured data in binary or character format (texts, images, videos and audio wave files). Oracle has relatively high access efficiency for LOBs, and also supports random and piece-wise data queries. Each LOB in Oracle is made up of two parts: a pointer (locator) and data content. Based on the difference in the size of content in a LOB, Oracle uses different storage methods. If the content in a LOB is less than 4000 bytes, it will store the data in a data table, placing the LOB's data and data table in the same table space. In such a case, the LOB is the same as a VARCHAR2, and the data in the LOB column can also be stored in the buffer. If the content in a LOB exceeds 4000 bytes, it will move the data to alobsegment.

Unlike Oracle's clustered storage, JBASE uses a LOB storage structure which is identical to a B-tree structure used for ordinary data. For every table with a LOB field (known as a master table), the system will set a slave table, which is used solely to store the LOB data of this table. The naming scheme of the slave table is taken from adding a JBLOB suffix onto the name of the master table. Regular users have access to the slave table, but it is suggested that they not make direct changes to it. The record format of the slave table is defined below:

Table 1 LOB Slave Table Structure in JBASE

The row ID for the LOB record in the master table.	The column ID for the LOB tuple in the master table.	The fragment ID of a LOB after it has been fragmented.	LOB data
ROWID	COLID	FRAGID	DATA

When a record containing a LOB is inserted into a master table, JBASE determines whether or not the record's size exceeds 950 bytes. If it is less than 950 bytes, the system will directly insert the record into the master table. Otherwise, the LOB data contained in the record will be directly inserted into a slave table, and it is simultaneously established by the system that the maximum value of a DATA field within each record of the slave table cannot exceed 950 bytes. If a LOB data exceeds 950 bytes, then it must be divided into several fragments. The extra three fields within each data fragment table 1 are transformed into a slave table record and inserted in the slave table.

This method used by JBASE is relatively simple. It adds fragment modules and re-assembled modules onto the basic memory module, which can slows for LOB access. However, this method has two flaws:(1)The maximum value (950 bytes) of data within a slave table record is too small. (2) The small size of the maximum value of a data fragment also leads to a space usage efficiency which is far too low.

Setting the maximum value of each data fragment to 950 bytes is unnecessary. If this maximum value is suitably increased, it could diminish the amount of time consumed in creating LOB fragments and re-assembling them, which would also increase space usage efficiency.

The LOB Log-Writing Mechanism . Commonly-used database management systems typically use change redo and undo logs to record the data change process when data is being edited. Similarly, JBASE also uses redo and undo logs to record the data change process when inserting LOB data. The following is a brief introduction of these two mechanisms.

Redo Log Processing. The JBASE redo log records updated actions the system takes on data blocks. By using the redo log, we can re-perform each and every system action, and guarantee the consistency of the system. When the system is operating normally, each physical record which is inserted into a table will result in the generation of a redo log. Before a transaction is committed, the system will first write the redo log in the system buffer onto the disk drive, and then commit the transaction. Using this method ensures that data can be restored through the redo log in the event of a crash, and also ensures the integrity and consistency of system data.

For a physical record which is to be inserted, a redo log will be created before the record is inserted into the data page. The log record structure consists of a sextuple:
(dbid, file_id, page_no, offset, len, new_data)
(dbid, file_id, page_no, offset, len, new_data)

The first four parameters are used to identify the address and in-block migration of the data block of the physical record within a file which is associated with the log record. The new_data parameter is formal data, which is information in the record head and record body of a saved physical record. The len parameter displays the length of the new_data data.

Each time the system restarts after a crash, it uses the redo log record created in the log file during previous operation. It uses information from the four dbid, file_id, page_no, and offset fields within the record to locate the address in the datablock within the associated physical record, then uses new_data to overwrite this physical record. This ensures that the transaction committed before the crash will still be updated in the database.

Undo Log Processing. The JBASE undo log mechanism can ensure the removal of any effects uncommitted transactions have on the system when restarting after a system crash. The system distributes many rollback segments for each transaction. Each time a transaction edits data, an associated rollback segment record will be created within the rollback segment. If the transaction is successfully committed, this rollback segment will be dropped. If the transaction is rolled back, the system will use the rollback segment to return the data within the database to the state it was in before the start of the transaction. In order to ensure that rollback segment data will not be lost in the event of a power loss of system crash, the rollback segments' data record is stored in the database's file, and not in the internal memory.

Optimized LOB Access Method

Optimization of LOB Storage Organization. The maximum value limit on the value of LOBs within a LOB slave table within the JBASE system has a very large effect on the storage space usage rate and access efficiency of LOBs. It is apparent that the larger the volume of a LOB that the slave table allows to be stored is, the less time is needed to access it, and the higher the space usage rate will be. In consideration of this, one can think about increasing the maximum value for LOB volume within a slave table to a "peak". The following is a discussion of how to determine this peak.

JBASE LOB data uses a B-tree structure for storage; one B-tree node has at least two physical records, in order to prevent the B-tree from reverting to a linked list. The system places a limit on record size: the maximum value of any physical record in the system cannot exceed one-half of the block length. The JBASE system allows for setting of the block size at the beginning of installation, with values of 2k, 4k, 8k, and 16k, which allows for easy calculation of the maximum allowable LOB volume. It becomes obvious that by expanding the maximum value of LOB volume in slave table records, the space usage rate can be increased and the time used to fragment and re-assemble LOB can be diminished.

Optimization of the LOB Log Mechanism. Through analysis of the JBASE log-writing mechanism it can be seen that when LOB data is being inserted into the slave table, there is no need to generate a complete redo log record of every physical record. Thus, one type of "optimized" thinking is: (1) Only store the physical record's rowid, colid, and fraid fields in the redo log record, with no need to store the DATA field or LOB data. By not storing the LOB data, the data amount in the redo log will be diminished, which will quicken the speed at which the log is written, as well as diminish the time needed to insert LOB data. (2) While a record is being inserted, the block number of the slave table's physical record datablock is saved. When the LOB is completely inserted, and before the transaction is committed, the redo log generated by the insertion operation will be wiped from the disk, followed by the wiping of these datablocks.

From this it can be seen that this method can also ensure the correctness of data after a system crash. Such an optimization is entirely capable of being implemented. The concrete instructions for this optimization can be seen below:

1. Edit the redo log generation mechanism. For the INSERT operation of the slave table, a pre-optimized system will create a log record for every edit of the physical page, and then write them into a log file. After optimization, the redo record only records non-fragmented data and its edits.

2. When inserting a LOB into the slave table, record the page number of the edited slave table data.

3. After completely inserting a LOB into the slave table, perform a wipe of the redo log.

4. After wiping the redo log, completely wipe the slave table data page which was edited during the insertion process.

Thus the optimization of the undo log generation method used by the system when inserting LOBs is complete. For ordinary data, if a record is inserted into a table, the system will create a rollback record and save it in the rollback segment. For LOB data, when it is fragmented and inserted into a slave table, a rollback record will be created for slave table records with identical rowid and colid entries, as well as those with a fragid value of 0. When being rolled back, the slave table's rowid and colid records with identical entries will also be rolled back based off of the rowid and colid of the rollback records. It is apparent that this method can ensure the correct rollback of the insertion operation.

Experimental Evaluation

Finally, we compared the test results of the pre-optimized and post-optimized systems. Because the goal of this optimization was to reduce the difference in LOB access efficiency between JBASE and Oracle9i, the pre-optimized LOB access speed needed to be compared between both JBASE and Oracle91 systems, after which the pre-optimization and post-optimization access speeds of the JBASE system were compared. Through comparison of Oracle9i and the post-optimized JBASE, pre-optimization and post-optimization results for LOB access speed were given. By comparing the LOB access speeds of the Oracle9i and the optimized JBASE with the increase factor of the pre-optimized JBASE, the difference in time efficiency between the Oracle9i and optimized JBASE systems can be compared.

Comparison of LOB Access Performance

1. LOB Insertion

The test program used an ODBC linked database, which committed a transaction for each data insertion. First is the comparison of the LOB insertion speed results of the pre-optimized JBASE and Oracle9i, the second is the comparison of the LOB insertion speed results of the pre-optimized JBASE and post-optimized JBASE:

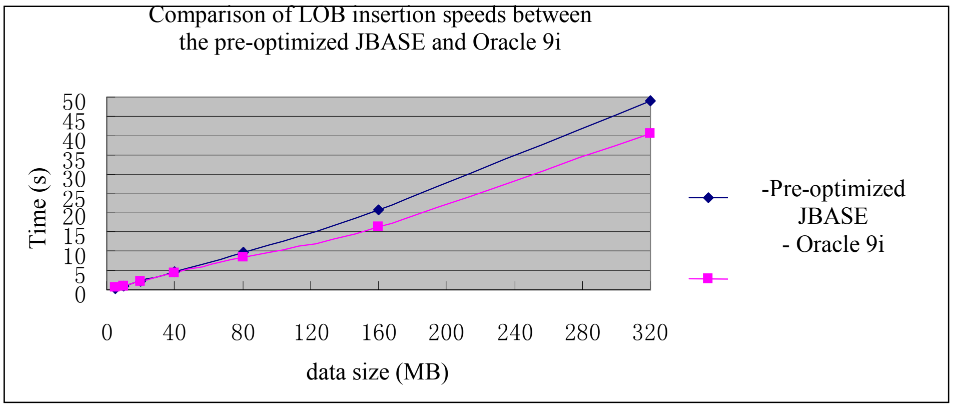

Fig. 1:Comparison of LOB insertion speeds between the pre-improved JBASE and Oracle9i

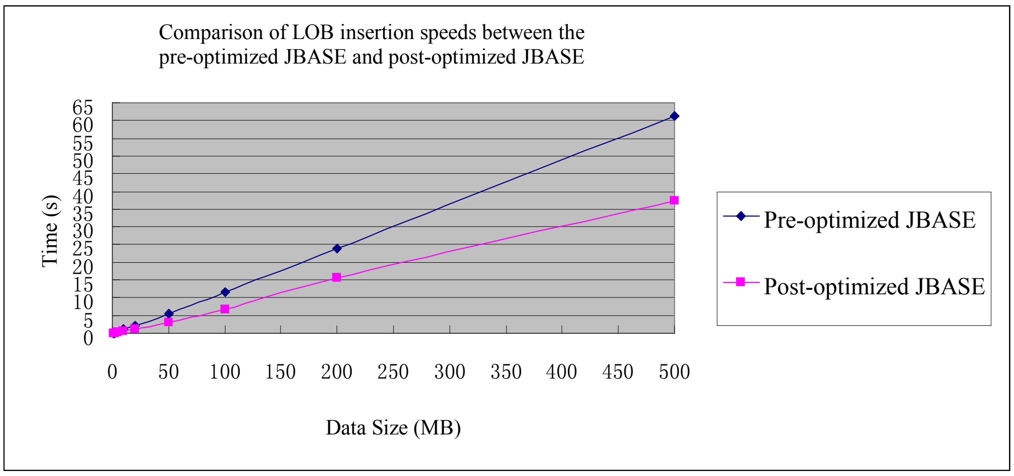

Fig.2:Comparison of LOB insertion speeds between thepre-optimized JBASE and post-optimized JBASE

2. LOB Reading

Next is the comparison of LOB reading speeds between the pre-optimized JBASE and Oracle9i, and between the pre-optimized JBASE and the post-optimized JBASE:

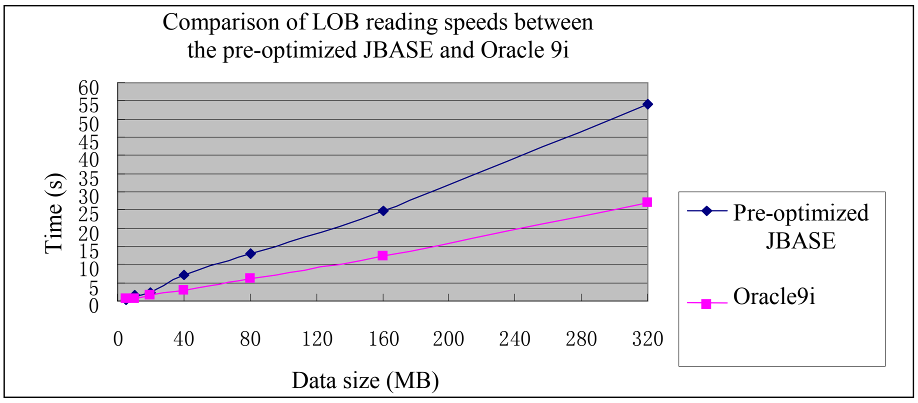

Fig.3:Comparison of LOB reading speeds between the pre-optimized JBASE and Oracle9i

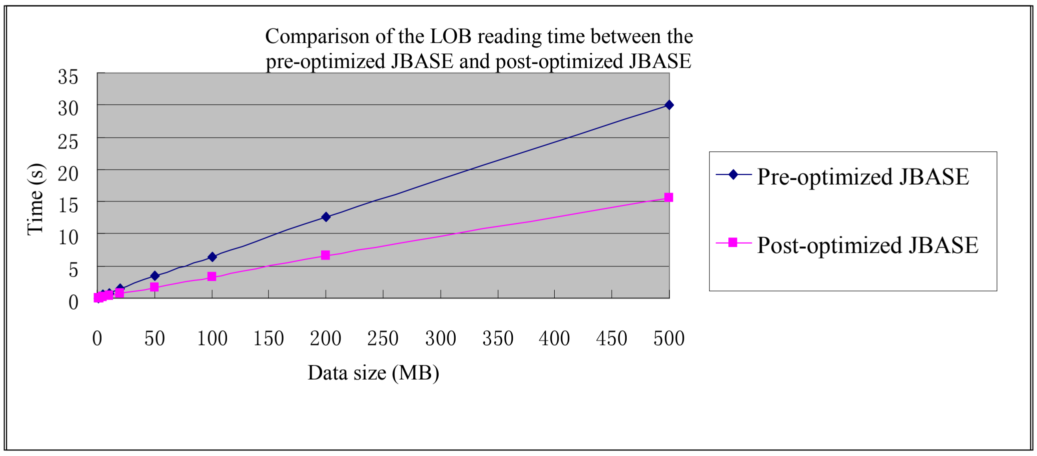

Fig.4: Comparison of the LOB reading time between thepre-optimized JBASE and post-optimized JBASE

Results Analysis. Based on the above four tests, the following conclusion can be reached: (1) The difference in LOB insertion speed between the pre-optimized JBASE and Oracle9i was not very large; on average, Oracle9i was faster by approximately 10%. However, for LOB reading speed, the Oracle9i was twice as fast as the pre-optimized JBASE.(2) The LOB insertion speed of the post-optimized JBASE was a vast improvement. As Graph 3 shows, the post-optimized JBASE was approximately 30% faster than the pre-optimized JBASE. For LOB reading speed, the post-optimized JBASE was approximately 90% faster than the pre-optimized JBASE (nearly twice as fast), and could basically be considered equal to Oracle9i in terms of reading speed.

Conclusions

This paper has researched the current methods for access of LOBs used by current database management systems. Based on this research, this paper discusses new methods for advancement and optimization, and through B tree combined slave table methods, it can reach optimized access efficiency based on organizational methods of LOB data access. Test results show that the optimized methods are slightly better than Oracle in terms of LOB access efficiency.

References

[1] Zhou AY. Data intensive computing-challenges of data management techniques. Communications of CCF, 2009,5(7):50−53 (inChinese with English abstract).

[2] Gunarathne T, Wu TL, Qiu J, Fox G. Cloud computing paradigms for pleasingly parallel biomedical applications. In: Hariri S,Keahey K, eds. Proc. of the HPDC. Chicago: ACM Press, 2010. 460−469. [doi: 10.1145/1851476.1851544]

[3] Das S, Sismanis Y, Beyer KS, Gemulla R, Haas PJ, McPherson J. Ricardo: Integrating R and Hadoop. In: ElmagarmidAK,Agrawal D, eds. Proc. of the SIGMOD. Indiana: ACM Press, 2010. 987−998. [doi: 10.1145/1807167.1807275]

[4] Liao HJ, Han JZ, Fang JY. Multi-Dimensional index on Hadoop distributed file system. In: Xu ZW, ed. Proc. of the Networking,Architecture, and Storage (NAS). Macau: IEEE Computer Society, 2010. 240−249. [doi: 10.1109/NAS.2010.44]