Managing Databases with Binary Large Objects

Michael Shapiro
University of Maryland at Baltimore County, Boeing IS at NASA GSFC

Ethan Miller
University of Maryland at Baltimore County

Abstract

We present recommendations on Performance Management for databases supporting Binary Large Objects (BLOB) that, under a wide range of conditions, save both storage space and database transactions processing time. The research shows that for database applications where ad hoc retrieval queries prevail, storing the actual values of BLOBs in the database may be the best choice to achieve better performance, whereas storing BLOBs externally is the best approach where multiple Delete/Insert/Update operations on BLOBs dominate. Performance measurements are used to discover System Performance Bottlenecks and their resolution. We propose a strategy of archiving large data collections in order to reduce data management overhead in the Relational Database and maintain acceptable response time.

1 Introduction

Most Relational Database Management Systems (RDBMS) experience performance problems when working with Binary Large Data Objects. Although decisions made at the physical database design phase can improve database performance in the long run, tuning of database systems is an essential process in achieving satisfactory performance. This paper describes different approaches to organizing both database storage and physical schema that, in many situations, favorably impact database query processing time.

An enormous amount of work has been done by file system developers and Relational Database Management System (RDBMS) designers to provide better support for large data object management. Nevertheless, no solution has been found as yet in the database world to guarantee high performance with respect to reading and writing large objects.

Storage space and database transaction processing time are the most important primary performance characteristics for optimizing RDBMS performance. Selecting the right storage option offered by the database vendor may be a very intricate process. In most cases, documentation does not provide information on the best choice for good performance, and the best guess often leads to poor performance. Performance measurements can be used to tune databases that have BLOB objects. Database prototyping can be used where databases with real data are unavailable.

We examine two ways for managing large objects, and determine under which access patterns a given storage management scheme is preferred.

The next section of this paper discusses related work on storage organization and management to support BLOBs. The database prototype that was used to perform benchmarking and performance measurements is presented in Section 3. Section 4 describes the measurement planning, implementation and database performance tuning.

2 Problem Background

Historically Relational Database Management Systems (RDBMS) were developed to support transaction processing. The key design

considerations were driven by the necessity to provide rapid access to tabular data. This affected the choice of tuning parameters and resulted in the logical database block size to being set only 4K.

These days multimedia applications are gaining in popularity. The necessity to store and manipulate very large data objects (LOB) provides a significant challenge for relational databases. Improving database performance for large object manipulation requires development of new storage strategies.

2.1 Related Work

Most of the past database systems were designed to manage simple facts - values that could be represented in fields of 255 bytes or less [1]. Larger fields, such as those containing thousands or millions of bytes, presented problems to the database record manager. The large fields were typically implemented via a separate long field mechanism. For some experimental RDBMSs the long field manager uses the buddy system for managing disk space. The buddy system allocates a range of small to very large disk extents (buddy segments) quickly and efficiently. When possible, a small number of large buddy segments are used to store long fields, which allows the long field manager to perform fewer disk seeks when transferring large objects to and from disks.

Most of the research that has been done on development of RDBMS mechanisms to manage databases with LOBs, concentrates on the following design goals:

1) Storage allocation and deallocation must be efficient. It is important that the allocation mechanism minimizes the time spent allocating and deallocating space to hold a LOB field.
2) The LOB field manager must have excellent I/O performance; LOB field read and write operations should achieve I/O rates near disk transfer speeds.
3) LOB field must be recoverable, but the recovery mechanism must not substantially slow down the LOB field operations.

The latest version of IBM DB2 - Universal Database (UDB) stores tablespaces within a high performance storage system (HPSS), supports caching of tablespaces on local disk, and supports manipulation of LOBs within the tablespace. This system effectively is able to manipulate petabytes of data.

A Massive Data Archive System (MDAS) testbed is being built at the San Diego Supercomputing Center. The system is a prototype to demonstrate the feasibility of managing and manipulating terabyte data sets derived from petabyte-sized archives [2]. The key software components of the MDAS testbed are a metadata library, a parallel database system (IBM DB2) [3] and HPSS.

Oracle demonstrates excellent performance for very large databases (VLDB) and constantly improves mechanisms to maintain BLOB objects. Extension of the storage capacity of a very large database cost-effectively within an Oracle 7 data warehouse system by integrating a long term archival storage sub-system with traditional magnetic media was presented in [4]. Based on these results, we have focused on Oracle performance for large BLOBs.

Oracle, version 7, offers two data types [5] that can accommodate LOBs. The data type LONG is used for storing strings with a size of up to 2GB. The data type LONG RAW is used for storing any BLOBs that are up to 2GB in size. This paper will concentrate on performance tuning for a database with BLOBs.

The latest release of Oracle8 [6] provides support for defining and manipulating LOBs. Oracle8 extends the SQL Data Definition Language (DDL) and Data Manipulation Language (DML) commands to create and update LOB columns in a table or LOB attributes of an object type. Further, Oracle8 provides an Oracle Call Interface (OCI) and a PL/SQL package Application Programming Interface (API) to support random, piecewise operations on LOBs. In addition release 8 of Oracle offers four extra data types to accommodate LOBs, to support both internal and external data representation.

There are three SQL datatypes for defining instances of internal LOBs:

1) BLOB, a LOB whose value is composed of unstructured binary ("raw") data;
2) CLOB, a LOB whose value is composed of single-byte fixed-width character data that corresponds to the database character set defined for the Oracle8 database;
3) NCLOB, a LOB whose value is composed of fixed-width multi-byte character data that corresponds to the

national character set defined for the Oracle8 database.

There is one external LOB datatype: BFILE, a LOB whose value is composed of binary ("raw") data, and is stored outside of the database tablespace in a server-side operating system file.

The idea of improving database performance by storing data outside of the database in a server-side operating system file is not new. For example, a tokenization mechanism has been proposed to store separately object values and pointers to these objects [7].

Tokenization is an encoding mechanism that replaces data values in a database by short, fixed-length rank-preserving tokens and stores the actual values separately. The actual values are stored nonredundantly, either in a separate database table or in a file, depending on the type of database application. Since tokens are, in general, much smaller than the values they represent, a database that exhibits some degree of value duplication will require less total storage than would be needed in the traditional approach.

If tokens are assigned to values in such a way that the rank property is preserved, comparison operations can be performed on the tokens themselves without referring to the actual values. Such operations are an essential component of retrieval, update and deletion transactions.

Performance measurements on the Oracle 7.3 database demonstrates the effectiveness of tokenization. Nevertheless for certain average size and number of LOBs stored in the database, usage of internal LOB types versus external files may give better performance especially for transactions that result in data retrieval.

2.2 Storage Systems

Research on database systems and file systems has resulted in fast and efficient storage mechanisms. To see how we might tune databases with LOBs, we look at existing storage systems.

2.2.1 Storing Large Data Objects in Database Systems

The SQL relational database system, System R [8], supported long fields with lengths up to 32,767 bytes. The System R long field manager divided long fields into a linked list of small manageable segments, each 255 bytes in length. Later, an extension to SQL was proposed that provided operators for long field manipulation. Along with this language extension, a storage mechanism was proposed that stored long fields as a sequence of 4 KB data pages. The maximum length of a long field in extended SQL was about 2 GB[9].

The database system EXODUS [10] stored all data objects through a general-purpose storage mechanism that could handle objects of any size - the limit being the amount of physical storage available. EXODUS uses a data structure that was inspired by the ordered relation data structure proposed for use in INGRES [11]. The data structure is basically a B+ Tree indexed on byte position within the object, with the B+ Tree leaves as the data blocks. Although the EXODUS data structures were designed more for random access than for sequential access, scan performance was improved if the data blocks comprised several sequential disk pages.

Oracle allocates logical database space for all data in a database. The units of database space allocation are data blocks, extents, and segments. The relationships among these data structures are shown in Figure 1.

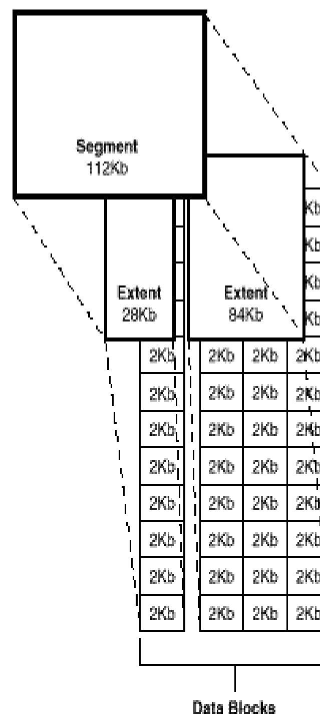

Figure 1. The Relationships Among Segments, Extents, and Data Blocks .

At the finest level of granularity, Oracle stores data in data blocks (also called logical blocks, Oracle blocks, or pages). One data block corresponds to a specific number of bytes of physical database space on disk.

The next level of logical database space is called an extent. An extent is a specific number of contiguous data blocks allocated for storing a specific type of information.

The level of logical database storage above an extent is called a segment. A segment is a set of extents that have been allocated for a specific type of data structure. For example, each table's data is stored in its own data segment, while each index's data is stored in its own index segment. If the table or index is partitioned, each partition is stored in its own segment.

Oracle allocates space for segments in units of one extent. When the existing extents of a segment are full, Oracle allocates another extent for that segment. Because extents are allocated as needed, the extents of a segment may or may not be contiguous on disk.

A segment (and all its extents) is stored in one tablespace. Within a tablespace, a segment can span datafiles (have extents with data from more than one file). However, each extent can contain data from only one datafile.

Oracle manages the storage space for the data files of a database in units called data blocks. A data block is the smallest unit of I/O used by a database. In contrast, at the physical, operating system level, all data are stored in blocks. Oracle requests data in multiples of Oracle data blocks, not operating system blocks.

In two circumstances, the data for a row in a table may be too large to fit into a single data block. In the first case, the row is too large to fit into one data block when it is first inserted. In this case, Oracle stores the data for the row in a chain of data blocks (one or more) reserved for that segment. Row chaining most often occurs with large rows, such as rows that contain a column of data type LONG or LONG RAW. Row chaining in these cases is unavoidable.

However, in the second case, a row that originally fit into one data block is updated so that the overall row length increases, and exceeds the block's free space. In this case, Oracle migrates the data for the entire row to a new data block, assuming the entire row can fit in a new block. Oracle preserves the original row piece of a migrated row to point to the new block containing the migrated row; the ROWID of a migrated row does not change.

When a row is chained or migrated, I/O performance associated with this row decreases because Oracle must scan more than one data block to retrieve the information for the row.

We have examined the Oracle 7.3.4 database system with respect to reading and writing large objects and conclude that it can exhibit satisfactory performance if the size of the objects and the number of objects in the database are right. The database performance improved when the majority of the transactions were Update/Insert/Delete and the combination of external LOB with tokenization mechanism was used.

2.2.2 File Systems

File systems store data contiguously or clustered in physical extents. Files with sequentially allocated disk blocks have high I/O performance, but sequential allocation causes disk space fragmentation.

Early IBM operating systems such as DOS and OS/MVS had file systems that left disk allocation up to the user; the user specified the number and location (for DOS) of disk extents required to hold the file. IBM's CMS file system running under VM/SP provided both automatic sizing and placement. Using the reasoning that blocks recently written will be read soon, the CMS file system allocated extents that were close to the current position of the virtual disk arm.

The original UNIX file system [12] used 512 byte data pages that were allocated from random disk locations. As virtually every disk page fetch operation resulted in a disk seek, the UNIX random page allocation scheme provided only a medium-level performance at best.

The UNIX Fast File System (FFS) is an improvement over the original UNIX File System because it uses the idea of physical clustering of disk pages. The FFS uses larger data pages that are allocated from cylinder groups, thus sequential scan performance is improved significantly.

The file system used by the Dartmuth Time Sharing System (DTSS) [13] uses the binary buddy system [14] for space allocation. A comparison of DTSS against the FFS shows that DTSS is much more efficient in allocating disk space. Through the use of large disk extents supplied by the buddy system, DTSS is able to sustain a high disk throughput with much less CPU usage. The average number of extents per file is a system parameter, and is chosen to minimize the number of disk seeks per file scan, while providing a maximum of 3 percent disk space loss to fragmentation. File systems designer came to the important conclusion that for the large files such as 100 megabytes, both larger extents and a larger number of them would be required.

3 The Prototype

Our objective is the development of relational database technology that provides better I/O performance than all of the systems described above. Although the existing systems show the benefits of using external large objects versus internal LOBs, we could not find any specific case description on where and when to use a particular technique. Lack of practical recommendations on when to use external versus internal LOB type objects provided the motivation to develop a prototype and perform measurements for both configurations on a variety request scales and types.

3.1 The Environment

We decided to chose Oracle 7.3.4, as the most popular commercial RDBMS, to carry out the performance measurements and evaluate the database performance tuning recommendations.

We used UNIX OSF/1 on a DEC Alpha with 256 MB RAM and a 5/300 CPU as the testbed for determining when each of the methods may improve database performance. OSF/1 was configured with POLYCENTER Advanced File System (AdvFS), a journaled local file system that provides higher availability and greater flexibility than traditional UNIX file systems (UFS). AdvFS provides greater flexibility by allowing filesets to share a single storage pool, enabling hard and soft fileset quotas in addition to user and group quotas.

Although, AdvFS supports a maximum file size of 128 GB, only a 3GB RAID 5 partition was available to develop the prototype.

In order to query the prototype database, two major test programs were used: one based on “SELECT-dominant” accesses and one based on “INSERT/UPDATE-dominant accesses. Programs were implemented using the ProcC capabilities.

3.2 Database Prototype Schemes

We call the database scheme for storing external large objects a “Scheme with Reference Tokens” and the database scheme for storing internal large objects a “Scheme with Key Tokens”. The “Scheme with Reference Tokens” (SRT) and “ Scheme with Key Tokens” (SKT) are shown in Fig. 2 and Fig. 3 respectively.

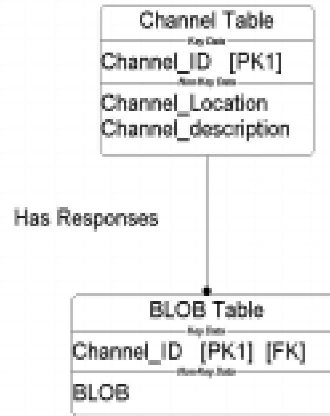

Figure 2. The Scheme with Key Tokens .

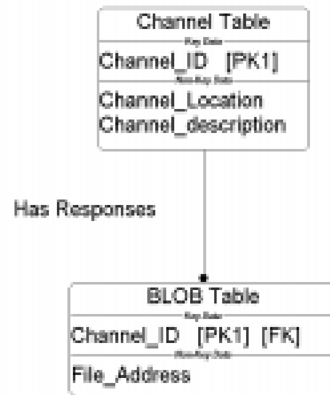

Figure 3. The Scheme with Reference Tokens.

IDEF1X notation was used to depict the cardinality of the relationship “Has Responses”, indicating that for each “Channel_ID” of the entity “Channel Table”, there exists zero, one or multiple instances of the entity “BLOB Table”. [PK1] stands for primary key while [FK] stands for foreign key.

We attempted to store exactly the same amount of data in both database schemes. The only difference is the location of the large object; in the case with reference tokens, the large objects were managed by the file system and in the case with key tokens the large objects were managed by the database system. The reference token was used to provide a pointer to a file, whereas the key token just uniquely identified the object itself.

3.3 Application Classes

Before we started planning the performance measurements, the typical workload for a database

supporting large objects was analyzed [15]. The following classes of applications appeared to characterize the workload and affect the system performance the most:

By Query Type:
- BLOB ad-hoc select - the application class where BLOB retrieval operations dominate.
- BLOB modifications - the application class where BLOB modification operations dominate.

By Size:
- Average Number of Objects stored.
- Average Size of an Object

4 The measurements

4.1 Performance characteristics

We divided the performance characteristics into primary attributes that we measured and secondary attributes that implicitly affected the primary attributes.

Primary:
- Transaction Processing Time.
- Storage Demands.

Correlated Performance Characteristics:
- Numbers Of Chaining Rows.
- Numbers of Extents:
  - Unix file system
  - Oracle database.

4.2 Implementation

We performed two sets of measurements: one sending the BLOB to the client, the other sending the BLOB to the server. Each set included two subsets: one for a schema with external LOB objects and one for a schema with internal LOB objects. In addition each subset included measurements for two LOB object sizes (0.5 MB and 5MB) and 3 database sizes (0.5 GB, 1,5GB and 3GB).

"SELECT-dominant" and "INSERT/UPDATE-dominant" programs were used to generate database transactions on both the client and server sides.

Our analysis of applications using LOB objects showed that random access to the LOB object is the most common. Thus the LOB object requests are uncorrelated.

4.3 Performance observations on the Server side

We ran the measurement programs on the Server first in order to reduce the number of factors that might influence system performance.

The graphs in Fig. 4,5 depict the transaction processing time for the different database scales for each of the BLOB sizes of 0.5 MB and 5MB. The transactions were generated with the "SELECT-dominant" program. The label SKT+0.5 means that transactions were run against the Scheme with Key Tokens (SKT) with average size of an object 0.5MB.

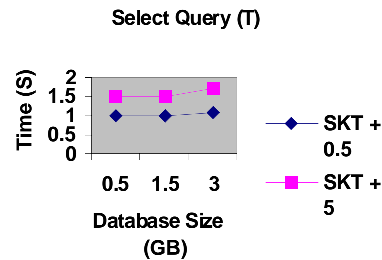

Figure 4. Transaction processing time versus database scale for "SELECT-dominant" random queries for SKT scheme.

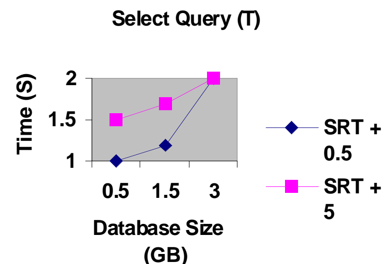

Figure 5. Transaction processing time versus database scale for "SELECT-dominant" random queries for SRT scheme.

The response time for 0.5MB objects for the Scheme with Reference Tokens increased by a factor of two as the size of the database increased. We believe that this fact is explained by the growth of the number of external files stored in the AdvFs system. For the 3 GB database this number reached 2,458 files.

Before the data were loaded the database was defragmented. This minimized the number of extents. However, multiple extents that were allocated for the new inserts may have affected system performance for the Scheme with Key Tokens.

Overall, Oracle demonstrated similar performance for "SELECT-dominant" random

queries for both internal and external types of LOB objects.

The graphs in Fig. 6,7 shows the transaction processing time for different database sizes with the average size of a large object of 0.5 MB and 5MB, when the transactions were generated with the "INSERT/UPDATE-dominant" program.

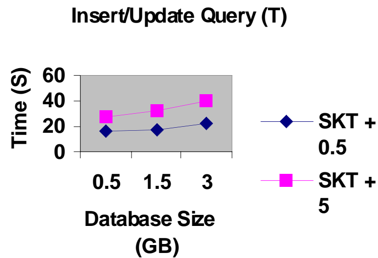

Figure 6. Transaction processing time versus database size for "INSERT/UPDATE-dominant" queries for SKT scheme.

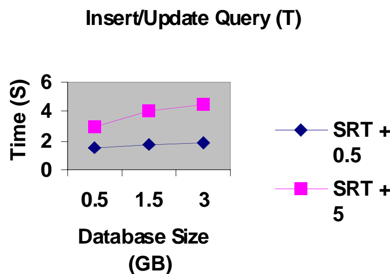

Figure 7. Transaction processing time versus database size for "INSERT/UPDATE-dominant" queries for SRT scheme.

Again, the database storage fragmentation was reduced to the minimum. Updates of objects stored internally in the database were substantially longer than for the objects stored to external files. The average response time for SRT approach was about seven times longer than for SKT approach. This is explained by the enormous number of I/O operations Oracle needs to maintain rollback and log information in addition to processing multiple migration and chaining rows. This number is substantially larger than the number of I/O operations AdvFs performs to execute the same INSERT/UPDATE request.

4.4 System Performance tuning considerations in the lights of prospective data collection growth and archiving strategy

As the database size grows, the response time gradually increases and the system must be tuned and redeployed [16]. The typical tuning process includes: design tuning, application tuning, memory and I/O tuning, and contention tuning.

Design Tuning - we examined system performance for SKT and SRT schemes in chapter 4.3 using Oracle database prototyping. We recommend design system having in mind possible future transition from a Scheme with Reference Tokens to a Scheme with Key Tokens and vice versa. This allows cost effective transition from one scheme to another in case such transition results in performance gain.

Application Tuning - we investigated the implications of application tuning on system performance. The application code can be analyzed to determine its effectiveness.

For Oracle it can be achieved by running the ANALYZE command on the tables with LOBs. This command collects statistics about the tables and stores them in the system tables. Switching the SQL TACE option on tells Oracle to collect statistics about queries that were executed against tables with LOBs. The TKPROF utility provides information about different phases of SQL statement execution. After analyzing these statistics we modified the SQL statements to substitute constants with bind variables. In addition, in order to keep SQL statements parsing to a minimum we pinned all SQL statements that operate on LOBs. Pinning SQL statement prevents it from being removed from the Library Cache.

The response time for both "SELECT-dominant" and "INSERT/UPDATE-dominant" programs improved by 3%. The observed difference in performance is understandable in terms of the decrease of number of I/O operations due to a Library Cache Hits increase.

Due to the limitations of Data Manipulation Language (DML) statements performing operations on LOBs, application code tuning can only insignificantly improve the system performance.

Memory and I/O tuning - we considered the implications of I/O and memory optimization on system performance.

For Oracle this is affected through use of the multiple kernel parameters. Tuning the following parameters affected the system performance: DB_BLOCK_SIZE , DB_BLOCK_BUFFERS and DB_FILE_MULTIBLOCK_READ_COUNT . These parameters can be considered for each phase of system deployment.

The first kernel parameter we tested was DB_BLOCK_SIZE. This parameter represents the database logical data block size and must be set up prior to allocating database space and loading data. The initial testing was performed using a 4K setting

(see performance observations on the Server side in chapter 4.3).

We reset this parameter to 16K and loaded into a newly installed database the same data using the same physical schemes. The response time for the "SELECT-dominant" program improved by 12% whereas the response time for the "INSERT/UPDATE-dominant" program improved by 30%.The observed difference in performance can be explained in terms of the decrease of physical data blocks read and written that reduced the total number of I/O operations needed to execute a transaction. The statistics were obtained by querying joined V$FILESTAT and V$DATAFILE views. The database logical data block size and size of a LOB correlate. Performance improves significantly when the LOB can be stored in one logical database block. Unfortunately this parameter is limited by a database.

The buffer cache is one of the most important structures which needs to be sized correctly when tuning memory. Its size in bytes is equal to DB_BLOCK_BUFFERS multiplied by DB_BLOCK_SIZE. The kernel parameter DB_BLOCK_BUFFERS represents the number of logical database blocks loaded into buffer cache. The effectiveness of this parameter can be determined by the cache hit ratio statistics, which show the proportion of data found in cache when requested. These statistics can be obtained by using the TLBSTAT/UTLESTAT utility. When object requests are uncorrelated. the DB_BLOCK_BUFFERS parameter can be tuned to accommodate the query result set. For example, having a buffer cache size of four megabytes degraded the response time by 20% when the table with 5 MB BLOBs was queried. At the same time, a buffer cache size of 10 megabytes improved the response time for "INSERT/UPDATE-dominant" programs by 5%. We found that the size of the buffer cache should be at least equal to 150% of the size of the largest LOB in the database to preclude substantial performance degrading. The observed difference in performance is understandable in terms of the decrease in the number of database accesses when the size of buffer cache is big enough to accommodate the query result set.

The parameter DB_FILE_MULTIBLOCK_READ_COUNT defines multiple block reads and sets the batch size. This parameter should be large enough to allow reading the entire LOB in one batch operation. Tuning this parameter can substantially improve database performance. After this parameter was reset from 50 to 400 we observed a 10% performance improvement for "SELECT-dominant" queries. This is explained by the decrease in latency of reading entire LOB object into buffer cache.

One or more rollback segments are allocated for a database to temporarily store "undo" information. Using deletes for LOB tables creates an extra overhead on the database system due to the large volume of data that must be temporarily (until commit is issued) stored in the rollback segment. In addition to a substantial performance degrading, the overflowed rollback segment crashes the system and the application terminates. The SQL TRUNCATE statement deletes without storing "undo" information. We observed 90% performance improvement by using TRUNCATE statement for deleting LOBs. The observed difference in performance can be explained in terms of the decrease in the number of database accesses. The TRUNCATE statement can be used where undoing delete operations is not application critical.

5. Conclusion.

Two ways of storing LOB objects in a database were presented. The database performance was measured for each case and compared to determine the preferred mechanism. The measurements on the server side proved:

1) For "Select-dominant" applications, using the Scheme with Key Tokens leads to better performance. The overhead for UNIX AdvFs was bigger than the overhead for database access. The difference in overhead is understandable in terms of the smaller number of I/O operations required to execute the respective transactions.
2) For "Insert/Update-dominant" applications, using the Scheme with Reference Tokens can significantly decrease the response time. The database overhead was bigger than the UNIX AdvFS overhead. Again, the difference in overhead can be explained by the large number of I/O operations that Oracle needs to maintain rollback and log information in addition to processing multiple migration and chaining rows. The larger the size of a LOB, the bigger the performance gain. Nevertheless, after database tuning we observed a

60% performance improvement. This resulted in similar response times for a LOB size of 0.5 MB for both the SKT and SRT approaches.

3) Tuning a database with LOBs can be productive in terms of system performance improvement. On Oracle example we achieved the better performance by tuning the following kernel parameters: DB_BLOCK_SIZE , DB_BLOCK_BUFFERS and DB_FILE_MULTIBLOCK_READ_COUNT. Application tuning provided us with only insignificant performance gain.
4) Increasing the database size and average object size adversely affects system performance. The bigger the size of the database the worse the response time. This trend held constant no matter how we stored the data and which physical scheme was used.

Acknowledgements

The authors thank the anonymous referees for their valuable comments and suggestions. Part of the material contained in this paper was first presented at the Mid-Atlantic Association of Oracle Professionals Special Interest Group Conference. This article research was supported, in part, by the NASA Goddard Space Flight Center Office of Flight Assurance, Applied Engineering and Technology Directorate and Management Operations Directorate.

References

[1] T. J. Lehman and B.G. Lindsay "The Starburst Long Field Manager," Proceedings of the Fifteenth International Conference on Very Large Data Bases, pp. 375-383, 1989.
[2] R. Moore, "High Performance Data Assimilation", 1995.
[3] C.K. Baru et al, "DB2 Parallel Edition," IBM System Journal, Vol 34, No. 2, 1995.
[4] S. M. Moran and V. J. Zak., "Incorporating Oracle On-Line Space Management with Long-Term Archival Technology," Proceedings of the Fourteenth IEEE Symposium on Mass Storage Systems, March 1996.
[5] ORACLE 7 Server Concepts User's Guide.
[6] ORACLE 8 Server Concepts User's Guide.
[7] R.C. Goldstein and C. Wagner, "Database Management with Sequence Trees and Tokens", IEEE Transactions on Knowledge and Data Engineering, Vol. 9, No. 1, January-February 1997.
[8] M.M. Astrahan et al., "System R: Relational Approach to Database Management," ACM TODS, Vol.1, no. 2, June 1976.
[9] R. Lorie and J. Daudenarde, "Design System Extensions User 's Guide," April 1985.
[10] M. J. Carey et al., "Object and File Management in the EXODUS Extensible Database System," Proceedings of the Twelfth International Conference on Very Large Data Bases, August 1989.
[11] M. Stonebraker et al., "Document Processing in a Relational Database System," ACM TOFS, vol. 16, no. 4, December 1984.
[12] D.M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," CACM, vol. 24, no. 7, July 1981.
[13] P.D.L. Koch, "Disk File Allocation Based on the Buddy System", ACM TOCS, vol. 5, no. 4, November 1987.
[14] D.E. Knuth, "The Art of Computer Programming, Vol 1, Fundamental Algorithms," Addison-Wesley, Reading Mass., 1969.
[15] D.A. Menasce and V.A.F. Almeida, "Capacity Planning and Performance Modeling," Prentice Hall PTR, Englewood Cliffs New Jersey, 1994.
[16] Eal Aronoff and Kevin Loney, "Oracle8 Advance Performance Tuning and Administration", Osborne/McGraw-Hill, Berkeley California, 1998.