The Design of Multimedia Object Support in DEC Rdb

1 Abstract

Storing multimedia objects in a relational database offers advantages over file system storage. Digital's relational database software product DEC Rdb supports the storing and indexing of multimedia objects-text, still frame images, compound documents, audio, video, and any binary large object. After evaluating the existing DEC Rdb version 3.1 for its ability to insert, fetch, and process multimedia data, software designers decided to modify many parts of Rdb and to use write-once optical disks configured in standalone drive or jukebox configurations. Enhancements were made to the buffer manager and page allocation algorithms, thus reducing wasted disk space. Performance and capacity field tests indicate that DEC Rdb can sustain a 200-kilobyte-per-second SQL fetch throughput and a 57.7-kilobyte-per-second SQL/Services fetch throughput, insert and fetch a 2-gigabyte object, and build a 50-gigabyte database.

2 Introduction

To accommodate the increasing demand for computer storage and indexing of multimedia objects, Digital supports multimedia objects in its DEC Rdb relational database software product. This paper discusses the improvements over version 3.1 and presents details of the new features and algorithms that were developed for version 4.1 and are used in version 5.1. This advanced technology makes the DEC Rdb commercial database product a precursor of sophisticated database management systems.

Multimedia objects, such as large amounts of text, still frame images, compound documents, and digitized audio and video, are becoming standard data types in computer applications. Devices that scan paper, i.e., facsimile machines, are inexpensive and ubiquitous. Devices that capture and play back full-motion video and audio are just beginning to reach the mass market. Capturing these objects for use within a computer results in many large data files. For example, one minute of digitized and compressed standard TV-quality video requires approximately 50 megabytes (MB) of storage!

To date, relational databases have been used successfully in storing, indexing, and retrieving coded numbers and characters. Relational algebra is an effective tool for reorganizing queries to reduce the number of records, e.g., from 1 million to 70 records, that an application program must search to obtain the desired information. Other database features, such as transaction processing, locking, recovery, and concurrent and consistent access, are essential to the successful operation of numerous businesses. Electronic banking, credit card, airline reservation, and

hospital information systems all rely on these features to query, maintain, and sustain business records.

However, although a business might have its numbers and characters organized, controlled, and managed in a computer database, maintaining the paper and film storage media associated with database records can be costly, both in dollars and in human resources. Some estimates place the worldwide data storage business at $40 billion, and as much as 95 percent of the information is stored on either paper or film. Currently, businesses such as insurance, banking, engineering, and medicine depend on human beings to manage the filing and retrieval of these extensive paper and film archives. Human error can result in the loss of paper and film. Clearly, scanning the paper, storing the information in a computer, and making this information available over computer networks is a better way to manage paper records. This scheme allows (1) multiple copies to be distributed at once; (2) a customer file to be electronically located and retrieved in seconds, whereas to materialize a paper folder can take days; and (3) properly programmed computers to maintain these types of information more efficiently and accurately than humans can.

The idea of eliminating paper-based storage of business records in favor of computer storage is long-standing. However, only recently have technical developments made it practical to consider capturing, storing, and indexing large quantities of multimedia objects. Storage robots based on magnetic tape or optical disk can be configured in the range of multiple terabytes (TB) at the low cost of 45 cents per MB. Central processors based on reduced instruction sets are getting fast enough to process multimedia objects without having to rely on digital signal coprocessors. Processor main memory can be configured in gigabytes (GB). Document management systems, which have thrived over the past few years, deliver computer scanning, indexing, storage, and retrieval across local area networks.

Until now, most multimedia objects have been stored in files. Document management systems generally use commercial relational database technology to store the documents' index and attribute information, where one attribute is the physical location of the file. This approach has several disadvantages: considerable custom software must be written and maintained to make the system appear logically as one database; application programs must be written against these proprietary software interfaces; a system based on both files and a relational database is difficult to manage; two backup-and-restore procedures must be learned and applied; and complications in the recovery process can occur, if the database and file system backups are executed independently.

Notwithstanding these disadvantages, storing multimedia objects in a relational database offers several advantages over file system storage.

- Coding an application against one standard interface structured query language (SQL) to store object attribute data as well as multimedia objects is easier than coding against both SQL to manage attribute data

and a file system to store the multimedia object.

- The database requires only one tool to back up and monitor data storage rather than two to maintain the database and the file system.

- The database guarantees that concurrent users see a consistent view of stored information. In contrast to a file system, a database provides a locking mechanism to prevent writers and readers from interfering with one another in a general transaction scheme. However, a file system does offer locks to prevent readers and writers from simultaneous file access.

- The database guarantees, assuming that proper backup and maintenance procedures are followed, that no information is lost as a result of media or machine failure. All transactions committed by the database are guaranteed. A file system can be restored only up to the last backup, and any files created between the last backup and the system failure are lost.

In the sections that follow, we present (1) the results of an evaluation of DEC Rdb version 3.1 for its ability to insert, fetch, and process multimedia objects; (2) a discussion of the impact of optical storage technology on multimedia object storage; and (3) design considerations for optical disk support, transaction recovery, journaling, the physical database, language, and large object data storage and transfer. The paper concludes with the results of DEC Rdb performance tests.

3 Evaluation of DEC Rdb as a Multimedia Object Storage System

Given the premise that production systems need to store multimedia objects, as well as numbers and characters, in databases, the SQL Multimedia engineering team members evaluated the following DEC Rdb features to determine if the product could support the storage and retrieval of multimedia objects:

- External interface support of inserting and fetching large objects

- Large object read and write performance

- Maximum large object size

- Maximum physical capacity available for storing large multimedia objects

The DEC Rdb product has always supported a large object data type called segmented strings, also known as binary large objects (BLOBs). The evolution from support for BLOBs to a multimedia database capability was logical and straightforward. In fact, the DEC Rdb version 1.0 developers envisioned the use of the segmented string data type for storing text and images in the database.

In evaluating DEC Rdb version 3.1, we came to a variety of conclusions about the existing support for storing and retrieving multimedia objects.

Descriptions of the major findings follow.

The DEC Rdb SQL, which is compliant with the standards of the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO), and SQL/Services, which is client-server software that enables desktop computers to access DEC Rdb databases across the

network, did not support the segmented string data type. Note that the most recent SQL92 standard does not support any standard large object mechanisms.[1] Object-oriented relational database extensions are expected to be part of the emerging SQL3 standard.[2]

The total physical capacity for storing large objects and for mapping tabular data to physical storage devices is insufficient. All segmented string objects have to be stored in only one storage area in the database. This specification severely restricts the maximum size of a multimedia database and thus impacts performance. One cannot store a large number of X-rays or one-hour videos on a 2- to 3-GB disk or storage area. Contention for the disk would come from any attempt to access multimedia objects, regardless of the table in which they are stored. Although multiple discrete disks can be bound into an OpenVMS volume set, thereby increasing the maximum capacity, data integrity would be uncertain. Losing any disk of the volume would result in the loss of the entire volume set.

The maximum size of the database that DEC Rdb can support is 65,535 storage areas, where each area can span 2[32] - 1 pages. That translates to 256 terapages (i.e., 256 x 10[12] pages) or 128 petabytes (PB) (i.e., 128 x 10[15] bytes). At a penny per megabyte, a 128-petabyte storage system would cost 1.28 billion dollars!

The largest BLOB that DEC Rdb can maintain is 275 TB (i.e., 275 x 10[12] bytes). A data storage rate of 1 megabyte per second (MB/s) for motion video and audio translates into 8.7 years of video. However, as mentioned previously, the maximum size and the total number of objects that can be stored are limited. As part of system testing, we successfully stored and retrieved a 2-GB object in a DEC Rdb data field.

DEC Rdb uses a database key to reference individual segments stored in database pages. A BLOB belongs to only one column of one row of a relation. The database key value that locates the first segment is stored in the column of a table defined to represent the BLOB data type. DEC Rdb implements segmented strings as singly linked lists of segments. Therefore, version 3.1 must read a segment in order to find the next segment. This process has two disadvantages: (1) random positioning with a BLOB data stream is extremely slow, and (2) BLOB pages cannot be prefetched asynchronously. Figure 1 illustrates a DEC Rdb version 3.1 singly linked list segmented string implementation.

BLOB data transfer performance of DEC Rdb version 3.1 was promising. We were able to code a load test that sustained 65 kilobytes per second (kB/s); a fetch test sustained 125 kB/s. To put these measurements in perspective, DEC Rdb is capable of inserting more than one A4-size (210 millimeters [mm] by 297 mm, i.e., approximately 8.25 by 11.75 inches) scanned piece of paper per second and capable of fetching more than two

A4-size pieces of paper per second. The test was conducted by writing and reading 50-kB memory data buffers to and from magnetic storage areas defined by the DEC Rdb software. This experiment ignores the overhead of network delays and compression.

DEC Rdb version 3.1 can write multiple copies of BLOBs, one to the target database storage area and one to each of the database journal files. The journal files provide for transaction recovery and system failures, such as disk drive failures. Database journal files tend to be bottlenecks, because every data transaction is recorded in the journal. Therefore, writing large objects to journal files dramatically impacts both the size of the journal file and the I/O to the journal file.

The volume of storage required for most modest multimedia applications can be measured in terabytes. A magnetic disk storage system 1 TB in size is expensive to purchase and maintain. An alternative storage device that provided the capacity at a much lower cost was required. We investigated the possibility of using Digital's RV20 write-once optical disk drive and the RV64 optical library ("jukebox") system based on the RV20 drives. We quickly rejected this solution because the optical disk drives were interfaced to the Q-bus and UNIBUS hardware as tape devices. Since relational databases use tape devices for backup purposes only and not for direct storage of user data, these devices were not suitable. Note that physically realizing and maintaining a large data store is a problem for both file systems and relational databases.

DEC Rdb version 3.1 does not support large capacity write once, read many (WORM) devices, which are suitable for storing large multimedia objects. Version 3.1 has no optical jukebox support either.

4 Storage Technology Impact

When we evaluated DEC Rdb version 3.1, a 1-TB magnetic disk farm was orders of magnitude more expensive than optical storage. Large format 12- or 14-inch (i.e., 30.5- or 35.6-centimeter) WORM optical disks have a capacity of 6 to 10 GB. The WORM drives support removable media. These drives can be configured in a jukebox, where a robot transfers platters between storage slots and drives. A fully loaded optical jukebox, which includes optical disk drives and a full set of optical disk platters, of approximately 1-TB capacity costs about $400,000, i.e., $0.40 per MB. By comparison, Digital's RA81 magnetic disk drive, for example, has a capacity of 500 MB and costs $20,000. Thus, to store 1 TB of data would require 2,000 RA81 disk drives at a total cost of $40 million, i.e., $40.00 per MB!

How big is one terabyte? Assume, conservatively, that a standard business letter scanned and compressed results in an object that is 50 kB in size. Therefore, 1 TB can store 20 million business letters, i.e., 40,000 reams of paper at 500 sheets per ream. A ream is approximately 2 inches (51 mm) high, so 1 TB is equivalent to a stack of paper 80,000 inches or 6,667 feet or 1.25 miles (2 kilometers) high! The total volume of paper is 160 cubic yards (122 cubic meters). A 1-TB optical disk jukebox is about 3 to 4 cubic yards (2.3 to 3 cubic meters). Assuming TV-quality video, 1 TB

can store 308 hours or approximately 12 days of video. Full-motion video archives suitable for use in the broadcast industry require petabytes of mass storage.

The Design of Multimedia Object Support in DEC Rdb

The gap between affordable and practical configurations of optical disk jukeboxes and magnetic disk farms has closed considerably since late 1992. Juxtaposing equal amounts (700 GB) of magnetic and optical storage, including storage device interconnects, installation, and interface software, reveals that magnetic disk storage is about five times more expensive than optical storage. The major disadvantage of optical jukebox storage is data retrieval latency related to platter exchanges. This latency, which is approximately 15 seconds, varies with the jukebox load and how data is mapped to different platters.

Mass storage technology, including device interconnects, combines different classes of storage devices into storage hierarchies. Storage management software continues to be a challenging aspect of large multimedia databases.

To provide 1 TB of mass storage capacity for relational database multimedia objects at reasonable cost, we conducted a review of third-party optical disk subsystems, hardware, and device drivers for VAX computers running the OpenVMS operating system. A characterization of the available optical disk subsystems revealed three basic technical alternatives.

1. Low-level device drivers provided by the drive and jukebox manufacturers.

2. Hardware and software that model the entire capacity of an optical disk jukebox as one large virtual address space.

3. Write-once optical disk drives interfaced as standard updatable magnetic disks. The overwrite capability is provided at either the driver or the file-system level, where overwritten blocks are revectored to new blocks on the disk. For example, consider a file of 100 blocks created as a single extent on a WORM device. When requested to rewrite blocks 50 and 51, the WORM file system writes the new blocks onto the end of all blocks written. The system also writes a new file header that contains three file extents: blocks 0 to 49 stored in the original extent; blocks 50 to 51 stored in the new extent; and blocks 52 to 100 stored as the third extent. Obviously, files that are updated frequently are not candidates for WORM storage. However, immutable objects, such as digitized X-rays, bank checks, and health-benefit authorization forms, are ideal candidates for WORM storage devices.

As a result of this investigation, we decided that using write-once optical devices, interfaced as standard disk devices, was the best solution to provide optical storage for multimedia object storage. This functionality is being met with commercially available optical disk file and device drivers.

In the future, WORM devices may be superseded by erasable optical or magnetic disks. However, experts expect that WORM devices, like microfilm, will continue to be useful for legal purposes.

5 Design Considerations

The tamperproof nature of WORM devices is an asset but causes special problems in database system design. The evaluation of DEC Rdb version 3.1 indicated that several features needed to be added to the DEC Rdb product to make it a viable multimedia repository. This section describes the design of the new multimedia features included in DEC Rdb versions 4.1 through 5.1.

Mass Storage

DEC Rdb version 4.1 supports WORM optical disks configured in standalone drive or jukebox configurations. DEC Rdb permits database columns that contain multimedia objects to be stored or mapped to either erasable (magnetic or optical disk) or write-once (optical disk) areas. The write-once characteristic can be set and reset to permit the migration of the data to erasable devices. No changes to application programs are required to use write-once optical disks, including jukeboxes.

The main design goals for WORM area support were to

- Reduce wasted optical disk space by taking into account the write-once nature of WORM devices
- Not introduce DEC Rdb application programming changes for WORM areas
- Maintain the atomicity, consistency, isolation, and durability (ACID) properties of transactions for WORM devices
- Maintain comparable performance, allowing for hardware differences between optical and magnetic devices

DEC Rdb uses the optical disk file system to create, extend, delete, and close database storage files on WORM devices. Although this approach uses the block revectoring logic in the optical disk file system, minimal space is wasted. When writing blocks to WORM devices, DEC Rdb explicitly knows that blocks can be written only once and bypasses the revectoring logic in the optical disk file system.

Nonetheless, DEC Rdb software could waste space in two major ways. First, when DEC Rdb creates a storage area on an erasable medium (e.g., a magnetic or erasable optical disk), the database pages are initialized to contain a standard page format, with page numbers, area IDs, checksums, etc. Preinitialized database pages help to determine corrupted database pages. However, preinitializing database pages on write-once media makes little sense. The second way in which DEC Rdb could waste write-once optical disk pages is to use storage allocation bit maps for space management (SPAM).

SPAM pages are used to keep track of free and used pages. As records are added to and deleted from the database, the SPAM bit maps are constantly updated. SPAM pages are maintained within each database file. With write-once devices, a page can be used only once. Again, it makes no sense to update SPAM pages for write-once media.

To eliminate needlessly wasting space on write-once media, DEC Rdb does not preinitialize WORM pages. As a general rule, WORM areas should not contain any updatable data structures. DEC Rdb maintains WORM storage space allocation in the database root file. The database root file should always reside on a magnetic disk, because the root file is frequently updated and magnetic disks yield higher performance. The clusterwide object manager mechanism ensures that the pointer to the end of the written area is consistent across a cluster.

SPAM pages, although disabled for write-once areas, are in fact allocated anyway. The reason for allocating SPAM pages in a write-once area is to provide the ability to migrate the contents of the storage area to an erasable device. The SPAM pages simply need to be rebuilt to reflect the space utilization at the point of conversion.

This write-once characteristic was the basis for several enhancements we made to the buffer manager and page allocation algorithms. Given that a free WORM page has never been written to, the buffer manager simply materializes an initialized buffer in main memory for write operations without having to first read the page from disk. In the case of page allocation for magnetic disks, DEC Rdb must scan SPAM pages in search of enough free storage space to satisfy a write operation. The scanning algorithm is much simpler for write-once areas; to store new records, DEC Rdb allocates one more page at the end of the written portion of the area to a process. DEC Rdb maintains such allocated pages in a queue called the marked WORM page queue on a per-process basis. Whenever a WORM page is written to disk, that page is taken off the marked WORM page queue. An attempt to store a record checks the queue before allocating new WORM pages to storage. Facilities exist to allocate many WORM pages in one operation, thus minimizing the number of writes to the root file.

By explicitly taking into account the write-once characteristic of the device, DEC Rdb greatly reduces wasted space, keeping optical disk read and write performance high.

Transaction Recovery

To understand the discussion of transaction recovery, the concepts of first- and second-class records must be understood. Both alphanumeric records and BLOB segments are stored in database pages. Alphanumeric records are first-class records and thus have identities in tables; these records are the rows. First-class records are required to be on a medium that permits update (either magnetic disk or erasable optical disk). All relation tuples are first-class records. Second-class records, such as BLOBs, have no identities of their own. BLOBs can exist only within the domain of an alphanumeric record and are pointed to by first-class records. Second-class records may be located in WORM areas.

Multimedia objects can be stored as second-class records in either write-once or erasable areas. However, due to transaction recovery constraints, the rows of relations must be stored in magnetic disks as first-class records.

If an update transaction against the database is aborted, then the database must restore the state of all database areas to pretransaction state. Regardless of the transaction recovery scheme employed, e.g., hybrid undo-redo, the effects of an uncommitted transaction to write-once media may have to be undone.

By definition, a write transaction on write-once media, once complete, can never be undone. In cases where a transaction fails and the transaction has written data to a write-once area, DEC Rdb employs a logical undo operation. This operation de-references the database key that points to the BLOB data written as part of the failed transaction. An example helps to illustrate how the logical undo operation works.

1. Consider row R of table T, which contains a column defined as data type BLOB.

2. The BLOB storage map indicates that the large objects are stored in a write-once area.

3. A process starts a transaction and updates the row storing a BLOB in the write-once area.

4. For some reason the transaction aborts.

5. Recovery nullifies the value of the database key that locates the first page of the BLOB.

The write-once pages can never be reused and will never again be allocated. Nothing points to or references data written as part of an aborted transaction.

This transaction recovery scheme introduces the interesting phenomenon of WORM holes. Consider the following scenario:

- A write-once area has the first 106 pages written and allocated.

- Process X starts a transaction that writes a BLOB segment to the write-once area.

- Page 107 is allocated for process X.

- Later in time, process Y starts a transaction to store a BLOB in the same write-once area.

- Process Y causes pages 108 to 120 to be allocated, data is written, the transaction commits, and process Y disconnects from the database.

- At this point, process X decides to roll back its transaction.

- Page 107 remains in a preinitialized state.

Page 107 can never be allocated to store BLOB data. Recall that DEC Rdb manages space on write-once devices by maintaining an end-of-area pointer to keep track of pages that have been written. Zero-filled pages that will never be allocated are called WORM holes. WORM holes are interesting because DEC Rdb utilities, such as verify, expect to find all allocated

pages in a standard format. The utilities have been modified to ignore empty pages on write-once areas.

Journaling Design Considerations

An effective database management system guarantees the recovery of a database to a consistent state in the event of a major system failure, such as media failure. Hence, full and incremental backups must be performed at regular intervals, and the database must record or keep a journal file of transactions that occur between backups. In DEC Rdb, the after image journal (AIJ) file records all transactions against the database since the last backup. Also, to recover from a system failure, the database must keep track of all outstanding or pending transactions. The recovery unit journal (RUJ) file records the state and data associated with all pending transactions.

Journal files are heavily utilized in a database management system. Contention for the journal files comes from every process that is updating the database. To be completely recoverable, the database management system must record BLOB data, as well as alphanumeric data, to both the AIJ and the RUJ files. Because multimedia objects are large, eliminating the need to write these objects to the journal files is desirable. The double-write transaction negatively impacts the performance of the application storing the object and taxes the journal file, one of the most burdened resources in the database.

As discussed in the Transaction Recovery section, DEC Rdb uses logical undo operations to undo aborted transactions. In addition to the minimal processing required to de-reference a database key pointing to the WORM area pages, DEC Rdb automatically disables RUJ log writes for WORM area records. This is another advantage of using WORM devices for multimedia objects.

Recording multimedia objects in the AIJ file is not so straightforward. DEC Rdb uses the AIJ file for media recovery, as well as for transaction recovery. By definition, keeping a media recovery journal forces twice the number of I/O operations, each to a separate device. DEC Rdb must write the multimedia object to the storage area designated for the multimedia object and write a copy of the object to the AIJ file. If the primary storage device that contains the object fails, the database administrator can apply the last full backup of the storage area, followed by any subsequent incremental backups, and roll forward through the AIJ journal file to recover the data. If a multimedia database is to be completely recoverable and consistent, then multimedia objects must be recorded in the AIJ file. Since they can never be erased, WORM optical disks might be the best devices to write an object (or a journal file) to. Even though a jukebox can misfeed and permanently damage the media, disks in a jukebox

can be disk shadowed. The trade-off is doubling the I/O versus risking data integrity. Rather than legislate a policy, DEC Rdb permits applications to disable AIJ logging for BLOBs, thus transferring the risk to individual applications.

Database Physical Design Considerations

The original design of segmented strings specified a singly linked list, where the segments were written one at a time, as shown in Figure 1. When writing a new segment, the previous segment had to be updated with a pointer value that identified the location of the new segment. For example, to store a BLOB with two segments R1 and R2, the old algorithm stored R1, stored R2, and then modified R1 to point to R2. Although this algorithm does not waste space on a magnetic disk, it does waste space on write-once optical disk. Segment R1 must be rewritten to disk with a pointer to segment R2.

If we impose the dependency between the two stores that R2 must be stored before R1, the store dependency for BLOBs becomes a reverse order of segments. Storing segments in reverse order requires buffering all segments of a multimedia object. Whereas buffering the entire object in main memory may be feasible for small multimedia objects, main memory is not large enough to buffer audio and video data objects. The singly linked list method that DEC Rdb used prior to version 4.1 is not well suited for WORM devices. Therefore, we redesigned the format of BLOBs in WORM areas to eliminate the need to buffer large amounts of data.

The new design replaces the singly linked list with BLOB segment pointer arrays and BLOB data segments. The segment pointer array maintains a list of database keys that locate each segment, in order, for a BLOB, as illustrated in Figure 2. Because segment pointer arrays are stored as a singly linked list, the pointer arrays can become large. Application data is stored in BLOB data segments. The new method buffers and writes the BLOB segment pointers to disk after assigning the segmented string to a record.

Besides eliminating the waste problem for write-once devices, the segment pointer array has other advantages. DEC Rdb reads the pointer array into memory when an application accesses a BLOB. DEC Rdb can, therefore, quickly and randomly address any segment in the BLOB. Also, DEC Rdb can begin to load segments into main memory before the application requests them. This feature benefits applications that sequentially access an object, such as playing a video game.

Storage Map Enhancements for BLOBs

Designers addressed several issues related to storage mapping. The major problems solved involved capacity and system management, jukebox performance, and the failover of full volumes.

Capacity and System Management. DEC Rdb can map user data, represented logically as tables, rows, and columns, into multiple files or storage areas. Besides increasing the amount of data that can be stored in the

database, spreading data across multiple devices reduces contention for disks and improves performance. However, as mentioned in the section Evaluation of DEC Rdb as a Multimedia Data Storage System, prior to DEC Rdb version 4.1, only one storage area could be used for storing BLOB data. All BLOB columns in the database were implicitly mapped into the single

area, which severely limited the maximum amount of multimedia data that could be stored in DEC Rdb.

Prior to new multimedia support for BLOBs, DEC Rdb restricted the direct storage of a particular table column to one DEC Rdb storage area (i.e., file). This partitioning control is accomplished by means of the DEC Rdb storage map mechanism, as shown in the following code example:

This code directs the BLOB data from the table PLACEMENT_HISTORY and the column RESUME of the table CANDIDATES to be stored in the area RESUME_AREA and the BLOB column PICTURE of the table CANDIDATES to be stored in the area PHOTO_AREA. The remaining BLOB data in the database is stored in the default RDB$SYSTEM area.

Restricting the storage of all BLOBs across the entire database schema to a single file or database area was clearly undesirable. The size of the area would be limited to the largest file that could be created by the OpenVMS operating system and the mass storage devices available. The limited mapping of one BLOB area mapped to one disk can be circumvented by using the OpenVMS system's Bound Volume Set mechanism. This mechanism allows n discrete disks to be bound into one logical disk. DEC Rdb can then create a single storage area on the logical disk that spans the bound set of disks.

However, although the volume set mechanism solves the problem of limited area mapping, serious limitations exist in the database system administration and recovery processes. All database-related facilities operate at the granularity of a database storage area. Thus, if one disk in a 10-disk volume set is defective, DEC Rdb would have to restore all 10 disks. Not only does restoring data on functioning disks waste processing time, but during the restore operation, applications are stalled for access at the area level. This situation introduces concurrency problems for on-line system operations.

DEC Rdb version 4.1 and successive versions solve the capacity problem by (1) permitting the definition of multiple BLOB storage areas, (2) binding discrete storage areas into storage area sets, and (3) providing the ability to map or to vertically partition individual BLOB columns to areas or area sets. Applications can set aside a disk or a set of disks for storing employee photographs, X-rays, video, etc. The alphanumeric data and indexes can be stored in separate areas as well. Figure 3 depicts the employee photograph column being mapped to the EMP_PHOTO_1, EMP_PHOTO_2, and EMP_PHOTO_3 storage area set. All alphanumeric data in the table EMPLOYEES is assumed to be mapped to storage area A.

Coding this example results in

This code directs the BLOB data, i.e., the column PHOTOGRAPH from the table EMPLOYEES, to be stored in the three specified areas EMP_PHOTO_1, EMP_ PHOTO_2, and EMP_PHOTO_3.

The ability to define multiple BLOB storage areas and to bind discrete areas into a storage set eliminates the BLOB storage capacity limitation in DEC Rdb. Consider the storage problem of storing 1 MB of medical X-rays as part of a patient record. Prior to DEC Rdb version 4.1, the limited one-BLOB storage area could store approximately 2,000 X-rays on a 2-GB disk device. The features included in version 4.1 allow the creation of a DEC Rdb storage area set that spans multiple disk devices. Also, adding storage areas or disks to a storage area set can expand the capacity initially defined for the column.

Jukebox Performance Problems. When a storage area set is defined using the SQL storage map statement, DEC Rdb implements a random algorithm to select a discrete area or disk from the set to store the next object. Since multiple processes access multimedia objects across the entire set, a random algorithm that evenly distributes data across the disks in the area set reduces contention for any one disk.

Using a random algorithm to select from a set of platters in a jukebox is extremely inefficient. A jukebox comprises one to five disk drives with 50 to 150 shelf slots where optical disk media is stored. A storage robot exchanges optical disk platters between drives and storage slots. As described earlier, a full platter exchange-spin down the platter currently in the drive, eject the platter, insert a new platter, spin up the new platter-takes approximately 15 seconds. Each optical disk surface, i.e., side of a platter, is modeled as a discrete disk to the OpenVMS operating system. Consider, for example, ten storage areas defined on optical disks in the jukebox and mapped into a storage area set. All patient X-rays from a single table in the database are to be stored in this area set. Each new X-ray inserted in the database causes DEC Rdb to randomly select a disk surface in the jukebox, which probably results in a platter exchange. Consequently, each X-ray insertion takes 15 seconds!

The solution to the jukebox performance problem was not to eliminate random storage area selection, which works successfully with fixed-spindle devices. Rather, the solution was to accommodate an alternate algorithm that sequentially filled the disks in an area set. Using DEC Rdb, applications can specify random or sequential loading of storage area sets as part of the storage map statement. Contention for a single optical disk in a jukebox is a far more desirable situation, with respect to latency, than causing one platter exchange per object stored.

When multiple users simultaneously issue requests to read multimedia objects stored in a jukebox, long delays occur, whether the storage area is loaded sequentially or randomly. Using a transaction monitor to serialize access to the database helps eliminate jukebox thrashing and improve the aggregate performance of the database engine.

Failover of Full Volumes. The introduction of storage area sets gave rise to another problem: What happens when one area in the set becomes full? Normally, within the DEC Rdb environment, disk errors that result from trying to exceed the allocated disk space are signaled to the application

so that the transaction can be rolled back (discarded). When related to storage area sets, however, the error is just an indication that a portion of the disk space allocated to the column has been exhausted and that processing should continue. Also, since multimedia objects tend to be exceedingly large, great amounts of data may have already exhausted cache memory and been written back to the WORM media, even though the database transaction has not committed. Handling such an error by signaling to the application and expecting the application to roll back and retry the transaction would result in the waste of a large number of device blocks that have already been burned. Thus, DEC Rdb had to implement a new scheme.

DEC Rdb now implements full failover of an area within the area set. Thus, when an area becomes full, DEC Rdb traps the error, selects a new area in the set, and writes the remaining portion of the BLOB being written to the new area. This area failover works whether the storage allocation is random or sequential. In addition, the area that is now full is marked with the attribute of full, and the clusterwide object manager of DEC Rdb maintains this attribute consistently throughout the cluster. Consequently, writers to the database will consider the area unavailable for future BLOB store operations. Further, the DEC Rdb database management utilities can remove the attribute if additional space is made available to the database area (e.g., if DEC Rdb moves BLOBs from area A to another copy of area A that resides on a device with twice the capacity).

Language Design Considerations

SQL, the ISO/ANSI standard relational database structured query language, is well suited to expressing queries against alphanumeric data yet hardly begins to address the needs of multimedia objects. Putting aside the fact that sampled data (i.e., a scanned image) is more difficult to query than coded data (e.g., text coded in ASCII), SQL cannot provide data compression and rendition capabilities for multimedia objects. Multimedia object processing is better suited to a language like C or C++. Ideally, SQL would support the ability to define objects and to associate methods with those objects. SQL3 is a new version of the SQL standard that the standards organizations are just beginning to work on. SQL3 contains the mechanism to define abstract data types and to execute external procedures as part of SQL statements. However, SQL3 will not become a standard for four to five years.

As discussed previously, DEC Rdb SQL lacks support for the segmented string or BLOB data type that was available in the Rdb relational engine. A new DEC Rdb SQL data type, LIST OF BYTE VARYING, was designed based on the native Rdb segmented string data type. The data access mechanism for the LIST OF BYTE VARYING data type is a list cursor, which operates like a table cursor-open the cursor, fetch segments of a BLOB, and close the cursor. This new data type with associated access mechanism was also

added to SQL/Services. SQL/Services software enables remote clients on a network, such as personal computers, to attach to remote DEC Rdb databases. The ability to scroll or to randomly position the list cursor allows

positioning at a particular data segment within the multimedia object stream without having to physically read through the entire data stream.

Although applications can program directly to list cursors, this interface was cumbersome and did not offer any object typing or processing. The list cursor mechanism does not present the straightforward byte-stream interface that is common in most file systems. Applications want to store objects, such as images and compound documents, not BLOBs. Data compression was another important consideration. Multimedia objects should be compressed on the client side of the network; then, compressed bits are transferred through the network, servers, and disks. The objects should be decompressed when they are to be rendered for display. Finally, the enormous size of multimedia objects saturates main memory resources on personal computers, so application developers must use disk storage to buffer as well as persistently store multimedia objects.

The limitations of the LIST OF BYTE VARYING data type and the list cursor data access mechanism led to the development of multimedia object extensions. SQL Multimedia is an object library that operates against SQL and SQL/Services. SQL Multimedia allows application developers to classify or type multimedia data types (e.g., IMAGE, TEXT, and COMPOUND_DOCUMENT) and to specify the data format within a type or class. Because no widely agreed upon multimedia object encodings or formats exist, we decided not to limit the types of data encoding or formats that could be stored in the database. For example, the database can store an image in Digital Document Interchange Format (DDIF) or Tagged Image File Format (TIFF). The option of defining a canonical encoding and format for each object class was too restrictive.

In both the SQL and the SQL/Services versions, the SQL Multimedia insert and fetch calls operate within the bounds of a transaction. All multimedia objects enjoy the same rights and privileges as alphanumeric data types in the database, with respect to concurrent access, recovery, etc.

A process that attaches to a DEC Rdb database can specify that an authorization identifier or a default identifier be created and referenced by the "RDB$HANDLE" symbolic label. A transaction can be started explicitly or a default transaction begins. To operate within the bounds of the default transaction, the SQL Multimedia routines required access to the default authorization identifier RDB$HANDLE. A new SQL compile time switch, for the SQL module language and precompilers, causes this identifier to be defined in a global address space. The SQL Multimedia routines can thus access the value of the identifier. If a distributed transaction identifier is not passed to the SQL Multimedia routines, the SQL Multimedia operation is executed using the default transaction.

SQL Multimedia improves the cumbersome list cursor interface by supporting

the following object sources and destinations:

- The entire object sourced from or deposited to main memory
- The object buffered through main memory

o A file

SQL Multimedia handles file I/O operations across many different software environments, including the MS-DOS, Windows, Macintosh, ULTRIX, and OpenVMS operating systems. SQL Multimedia preserves file attributes on insert operations. For example, the Macintosh file system's resource fork, which contains the name and version of the application to be launched when the object is accessed by a user, is preserved. If another Macintosh user fetches the object to a local file, then SQL Multimedia restores the file including the resource fork. Assuming the second user has the same application, the user can now access and manipulate the multimedia object, e.g., a compound document or a QuickTime video file. Rules and default file organizations exist for the case where a user inserted a file from an OpenVMS system and another user causes the object to be fetched to a different client file system, say on a PC. Application programmers can direct SQL Multimedia to override the default file attributes.

Although SQL Multimedia handles disparate file system I/O, at present, it does not convert multimedia object formats or encodings. Images captured and stored in DEC Rdb in DDIF are delivered to each client in DDIF.

SQL Multimedia makes it easy for application programmers to insert and fetch compound documents to and from the database. The buffered I/O data stream conforms to Digital's Compound Document Architecture (CDA) stream management interface. Fetching a compound document using the buffered I/O interface, SQL Multimedia returns the address of a procedure entry mask, a data buffer pointer, and the buffer length. These returned arguments can be passed to the CDA viewer in the DECwindows environment. The viewer then repeatedly calls the SQL Multimedia buffer-fill procedure until the object has been transferred to the viewer and displayed.

In addition, SQL Multimedia provides object-specific processing for image and text objects. Disk image objects formatted according to DDIF and main memory objects formatted according to Digital's image toolkit DECimage Application Services (DAS) can be processed on either fetch or insert operations. SQL Multimedia leverages the capabilities of DAS software to provide image processing, e.g., compression, decompression, scaling, and dithering. When an image is inserted or fetched, SQL Multimedia object processing arguments permit the specification of image process steps and parameters. The DAS toolkit supports Comité Consultatif Internationale de Télégraphique et Téléphonique (CCITT) compression (a ubiquitous compression standard for facsimile machines) for bitonal images and Joint Photographic Experts Group (JPEG) compression (an ISO/ANSI standard) for multispectural images.

To improve application performance, SQL Multimedia can generate multiple rendered versions of an image that are stored in a single database field.

Therefore, a user can store the original image, retaining its fidelity, and also store a miniature version of the image for fast access or browsing purposes. For example, consider a personnel application where 90 percent

of the fetches for employee photographs are to be displayed in a passport-size format on an employee information form. If the capture portion of the application stored the original employee photograph and directed SQL Multimedia to generate and store a passport-size rendered version in addition to the original, at fetch time, the I/O operations required to transmit the image to the employee form would be reduced. Storing multiple rendered versions would also eliminate using CPU time to scale the fetched image.

6 System Testing and Evaluation

After the multimedia engineering of the DEC Rdb product was complete, we conducted several testing activities to determine the performance and capacity boundaries. The performance work presented is not complete but is offered as an indication of the multimedia object access capabilities of the DEC Rdb software.

In the debit credit domain, the Transaction Processing Performance Council (TPC) tests provide a standard procedure to measure the performance of one database as compared to another. However, no standard multimedia database performance tests exist. The performance of a DEC Rdb multimedia database is influenced by many variables, including the processor, mass storage medium, database design, object sizes, and workload. The performance data presented in this paper should be used only as a guide.

Performance Testing

For performance testing we used a VAX 6360 processor (relatively slow by today's standards) configured with 128 MB of main memory, an HSC50 storage interconnect processor with 16 RA70 magnetic disks, 6 RA92 magnetic disks, and 2 ESE20 solid-state disks. The total mass storage available for building databases was 10 GB. We evaluated the SQL performance of DEC Rdb version 4.2 Field Test 1 (FT1) and SQL Multimedia version 1.0 Field Test 2 (FT2), and generated the SQL/Services remote client data fetch and insert performance data for DEC Rdb version 4.1 Field Test 4 and SQL Multimedia version 1.0 FT2.

This performance data should be used as a guideline, because the field-test software contained implementation errors that affected performance but were corrected in the released products. As presented in Table 1, using the released version of DEC Rdb, we are able to sustain a 300-kB/s throughput from a magnetic disk DEC Rdb storage area, across an Ethernet network, to a DECstation 5240 workstation. This test demonstrates fetching a software motion pictures (SMP) video clip out of the database for display on an ULTRIX-based workstation.[3] Although the video was sampled at 15 frames per second, we can play back the video clip at 20 frames per second! The performance measured for an fetch was 57.7 kB/s, as shown in Table 2. We

expect to conduct similar performance tests on a DEC 7000 AXP processor.

The performance test inserted and fetched 50-kB records. Fifty kilobytes is a conservative estimate of a compressed A4-size piece of paper, probably the most prevalent object to be stored in multimedia databases. For both the distributed SQL/Services client and the local SQL interface, 50-kB main memory buffers were the sources and destinations for the inserts and fetches.

We built several 50-MB databases, varying database design parameters such as page and buffer sizes, to determine the fastest set of parameters for the large object performance test. Using the largest page and buffer sizes yielded the best performance. The database table was organized into three columns: two key columns and a BLOB column. The BLOB column was mapped to a storage area set consisting of multiple magnetic storage disks.

After we established the best database organization, we built many 3- to 10-GB databases by

- Varying the number of processes executing insert and fetch operations
- Varying the number of tables in the database
- Varying the number of inserts and fetches per transaction
- Enabling and disabling AIJ journaling
- Inserting and fetching from an SQL/Services client or using SQL for local database access

When we conducted the performance tests, the computer was dedicated to our task; no other activity was taking place. A simple contention test, where multiple readers simultaneously fetch objects from a single table, and a more complicated update test, where multiple writers are simultaneously updating one table, have yet to be fabricated and run.

To put some of the performance results presented in Table 1 into perspective: the tested configuration can sustain approximately 600 kB of insert bandwidth, which translates into twelve 50-kB A4-size pieces of paper per second. Even a single process scanning paper at 103.4 kB/s can keep up with some of the fastest paper scanners available.

Also, scanning both sides of a compressed bank check (scanned at 200 dots per square inch) results in an object size of about 20 kB. Therefore, the particular configuration we tested could store 30 checks per second with multiple processes, and 6 checks per second with a single process.

Capacity Testing

We conducted two capacity tests. The first stored and fetched a 2-GB object in a DEC Rdb field, and the second built a 50-GB database. A 2-GB known pattern was generated in virtual memory. DEC Rdb wrote this object, with no AIJ, to a field in an empty database. The BLOB column was mapped to three disks, totaling 2.5 GB of storage. To avoid having to sustain storage area or file extensions, the storage area set was defined to be 2.3 GB. DEC Rdb was able to successfully insert and fetch the 2-GB object.

To demonstrate the capacity that could be achieved with SQL Multimedia, DEC Rdb, and optical storage, we built a 50-GB database. The hardware configuration consisted of the following:

- A VAX 4000 Model 500, with 6 GB of magnetic disk and 128 MB of main memory
- A Kodak Automated Disk Library Model 6800, with 100 GB of storage (with a maximum capacity of 1.2 TB)
- DEC Rdb version 4.2 Field Test 0
- SQL Multimedia version 1.0 FT2
- Perceptics LaserStar optical disk software

Starting with a backup of a 2-GB manufacturing database that was used by Digital's Mass Storage Group, DEC Rdb added an SQL Multimedia column to a table that contained over 550,000 rows. DEC Rdb then mapped the column to five platters, modeled as ten 9.5-million-block (5.1-GB) magnetic disks to the OpenVMS operating system, using the sequential load algorithm. An update table cursor was devised that returned between 2,000 to 3,000 rows. Using SQL Multimedia, DEC Rdb inserted images representing the disk assembly process until the storage was full.

7 Conclusion

The multimedia features that have been added to Rdb are in direct support of the increasing demand for computer data storage and indexing of multimedia object types (i.e., text, still images, compound documents, audio, and video). Relational database systems must expand mass storage device support, database physical database design, language functionality, and performance to manage the variety of today's information. The development of this advanced technology in Digital's DEC Rdb product provides desktop computer-to-optical disk jukebox integration by means of a commercial database. As multimedia technology matures, databases must address the need to store and index information beyond numbers and characters.

The work accomplished to support multimedia objects in DEC Rdb is just "the tip of the iceberg." Current multimedia capabilities are able to successfully manage the majority of document and still frame applications. However, improvement in capacity and performance are required before the database can serve multiple channels of video and audio data. As the SQL standard evolves to incorporate a more object-oriented mechanism, much of the SQL Multimedia functionality will migrate to using standard interfaces to define, operate on, and query abstract data types.

8 Acknowledgments

A large number of people from various disciplines contributed to the success of this multimedia database project, including Becky Jacobs, Michael Sawyer, John Lacey, Cheri Jones, Bruce Mills, Steve Hagan, Ian Smith, Susan Hillson, Peter Spiro, J. M. Smith, Jim Gray, Dave Lomet, Rudy Downs, Ken Cross (Perceptics), Chris Eastland, Mase Merchant, Scott Matsumoto, Paul Carmen (Eastman Kodak), Jim Lewis (Eastman Kodak), and Marilyn Gulliksen.

9 References

1. American National Standard for Information Systems-Database Language-SQL,
   ANSI X3.135-1992 (New York, NY: American National Standards Institute, 1992) and

   Information Technology-Database Language-SQL, ISO/IEC 9075:1992 (Geneva: International Organization for Standardization, 1992).

2. J. Melton, ed., Database Language SQL (SQL3), ISO/ANSI Working Draft, ANSI X3H2-93-091 and ISO/IEC JTC1/SC21/WG3/DBL YOK-003 (February 1993).

3. B. Neidecker-Lutz and R. Ulichney, "Software Motion Pictures," Digital Technical Journal, vol. 5, no. 2 (Spring 1993, this issue): 19-27.

10 General References

SQL Extensions

K. Meyer-Wegener, V. Lum, and C. Wu, "Image Management in a Multimedia Database System," Proceedings of the IFIP TC 2/WG 2.6 Working Conference on Visual Database Systems, Tokyo, Japan (1989): 497-523.

M. Stonebreaker, "The Design of the POSTGRESS Storage System," Proceedings of the 13th International Conference on Very Large Databases, Brighton, U.K. (1987): 289-300.

M. Stonebreaker and L. Rowe, The POSTGRESS Papers, Memorandum No. UCB/ERL M86/85 (Berkeley, CA: University of California, 1986).

Object Storage Management

M. Stonebreaker, "Persistent Objects in a Multi-Level Store," Proceedings of the ACM SIGMOD International Conference on Management of Data, Denver, CO (1991): 2-11.

WORM Devices

D. Maier, "Using Write-Once Memory for Database Storage," Proceedings of the ACM SIGMOD/SIGACT Conference on Principles of Database Systems (PODS) (1982).

S. Christodoulakis et al., "Optical Mass Storage Systems and Their Performance," IEEE Database Engineering (March 1988).

S. Christodoulakis and D. Ford, "Retrieval Performance Versus Disk Space Utilization on WORM Optical Disks," Proceedings of the ACM SIGMOD International Conference on Management of Data, Portland, OR (1989): 306-314.

Storage Management for Large Objects

A. Biliris, "The Performance of Three Database Storage Structures for Managing Large Objects," Proceedings of the ACM SIGMOD International Conference on Management of Data, San Diego, CA (1992): 276-285.

11 Trademarks

12 Biographies

James J. Feenan, Jr. Principal engineer Jay Feenan has been implementing application code on database systems since 1978. Presently a technical leader for the design and implementation of stored procedures in DEC Rdb version 6.0, he has contributed to various Rdb and DBMS projects. Prior to joining Digital in 1984, he implemented Manufacturing Resource Planning systems and received American Production and Inventory Control Society certification. Jay holds a B.S. from Worcester Polytechnic Institute and an M.B.A. from Anna Maria College. He is a member of the U.S. National Rowing

Team.

John L. Janosik, Jr. A principal software engineer, John Janosik was the project leader for DEC Rdb version 5.0, the Alpha AXP port version. John has been a member of the Database Systems Group since joining Digital in 1988. Prior to this, he was a senior software engineer for Wang

Laboratories Inc. and worked on PACE, Wang's relational database engine and application development environment. John received a B.S. in computer science from Worcester Polytechnic Institute in 1983.

T. K. Rengarajan T. K. Rengarajan, a member of the Database Systems Group since 1987, works on the KODA software kernel of the DEC Rdb system. He has contributed in the areas of buffer management, high availability, OLTP performance on Alpha AXP systems, and multimedia databases. Presently, he is working on high-performance logging, recoverable latches, asynchronous batch writes, and asynchronous prefetch for DEC Rdb version 6.0. Ranga holds M.S. degrees in computer-aided design and computer science from the University of Kentucky and the University of Wisconsin, respectively.

Mark F. Riley Consulting software engineer Mark Riley has been a member of the Database Systems Group since 1989 and works on multimedia data type extensions in Rdb/VMS. Prior to this, he worked for five years in the Image Systems Group and developed parts of the DECimage Application Services toolkit. Mark received a B.S.E.E. from Worcester Polytechnic Institute in 1980 and an M.S. in engineering from Dartmouth College in 1982.