

# Version Management and Recoverability for Large Object Data

Randal Burns <sup>†</sup>

IBM Almaden Research Center  
650 Harry Rd., San Jose, CA 95120  
randal@almaden.ibm.com

Inderpal Narang

IBM Almaden Research Center  
650 Harry Rd., San Jose, CA 95120  
narang@almaden.ibm.com

## Abstract

*Most applications that access large data objects do so through file systems, but file systems provide an incomplete solution, as they maintain insufficient metadata and do not provide general purpose query engine. Storing large objects in a database addresses these problems, but, for applications that need to update object data, databases are inefficient as they do not provide direct access to data. Additionally, databases often relax the integrity and consistency constraints for large objects, as it the case with objects stored through the Binary Large Object (BLOB) data type. These shortcomings are exacerbated by multiple users or applications that wish to access large objects concurrently.*

*We describe an architecture, based on the Datalink data type, in which large objects in a database are continuously available for read access and can be read and written through a file system interface. Additionally, this system does not relax version management, consistency and recoverability guarantees, as with the BLOB data type.*

## 1. Introduction

The Datalinks system and its associated Datalink data type have been developed at the IBM Almaden Research Center and implemented in IBM's DB/2 database system [4]. This system integrates the desirable elements from file systems and databases. Large object data can be "stored" in a database, *i.e.* a database system manages metadata, access control, consistency and integrity. At the same time, the data is stored on a cooperating file system. Having accessed an object through the database interface, an application may read, write and otherwise manipulate the object using file system function calls. In this way, applications realize the benefits of enhanced metadata and query capabilities without changing either the efficiency or manner in which data is accessed.

---

<sup>†</sup> This author is also a graduate student at the Department of Computer Science, University of California, Santa Cruz, CA 95064.

To date, most data resides in file systems and most applications access this data through the file system interface. However, file systems are unmanageable in that they provide neither sufficient metadata nor a general purpose query engine. Multimedia applications, digital libraries and computer aided design and manufacturing (*CAD/CAM*) applications illustrate this problem. For example, tools for manipulating image data expect to access a file in a standard image format. Yet often we wish to associate the image with complex metadata, such as pricing, development and inventory information in a product database. The most complex part of managing such a database is to keep associated file data consistent with its metadata when the tools for manipulating file data and the database do not cooperate.

Alternatively, a DBMS manages data well, but may not be appropriate for the storage of large data objects. Many databases store object data using the Binary Large Object (*BLOB*) data type [6]. *BLOBs* stores arbitrarily large objects in a database and allow an object to have complex metadata which may be interrogated using a general query interface. However, *BLOBs* do not provide an adequately efficient interface for accessing the contents of an object. To access the contents of a *BLOB*, the data must be exported to a file system.

Let us consider a parts database for a complex product being designed in a *CAD/CAM* system. Such a parts database stores hundreds to thousands of parts, indexed by part number. Furthermore, each part may have many design files associated with it. Obviously this application requires a database to manage the relationship between parts, part metadata and design files. However, this design application requires the frequent update of large bodies of data. As such, the efficiency of a file system interface also seems required. The demands of this application are met neither by a DBMS nor a file system and we require a new technology to manage this application.

The Datalinks system marries file systems and databases so that files appear to be stored in a database, but in actuality are stored and accessible through a cooperating file system, called a Datalinker. For this reason, Datalinks allows object

data to be associated with complex metadata while continuing to permit the access of object contents through the file system interface.

Even with this dual interface, the task of providing consistency, integrity and recoverability to large object data is formidable. Yet, applications that wish to share object data require these guarantees. The potential enormity of object data would seem to prohibit a database system from managing the multiple versions of objects required for database tasks such as point in time restore. However, in Datalinks, version management can be off loaded from the database to a cooperating file system. By leveraging file system technology for the efficient backup and restore of file system data, Datalinks manages multiple versions of object data at the file system and guarantees control and management of object data as if were stored in the database.

Section 2 describes related work in the field. In §3 we summarize the Datalinks system. In §4 we compare the semantics and performance of the Datalink data type with the BLOB data type. We continue with a description of the Datalinks file update model in §5. In §6 we look at coordinated schemes for backup and recovery for a Datalinks database and its cooperating file system. We conclude in §7.

## 2. Related Work

The desire to access multimedia data with version control and recoverability guarantees created a need for large object storage in existing databases. The simplest solution to this problem creates a new binary large object (BLOB) type embedded in relational tables [7]. However, BLOBs present two significant problems: first, applications have neither semantic knowledge nor an interface to the contents of a BLOB, and, the management of large object data for import/export, update or logging prohibitively taxes system resources.

Exporting BLOB data to a file system merely to determine that your application has no interest in the data is clearly unacceptable. To address this problem, researchers and developers embed methods associated with the object in the database. BLOBs provide their own functions for interrogation and applications may query this interface. In relational databases this is called type-extensibility [10, 14], also called the object-relational model. This type-extensibility or self-describing data is a feature inherent to object databases management systems (ODBMS) [13, 8]. In either case, this model for object interrogation is limited to published methods and therefore not general.

Storing large object data greatly taxes the I/O and storage capabilities of existing databases. For this reason, the development of BLOB storage has lead to new research describing enhancements to buffer management and paging

[11], improved database storage systems [5] and the need for fast networks [12]. All of this work points in one direction, to DBMSs that permit interrogation of object data without overutilizing database system resources. Datalinks provides these capabilities by storing object data externally in file systems and linking object data into databases.

## 3. Description of Datalinks

The Datalinks system centers around a new data type called a Datalink [9] which associates external object data with a database (Figure 1). The Datalink makes an external reference to an object on a remote and cooperating file system, called a Datalinker. At its most basic level, a Datalink is a pointer to a file stored on an external file system and in this way is very similar to a uniform resource locator (URL). The contents of this data type are the pair  $\langle server :: path \rangle$  that specify a Datalinker file server, which stores the object, and the local path of that object on the Datalinker.

As a Datalink data type is compact, a database can easily manage entries of this type. Changes to this entry may be recorded in the database log. Insertions, deletions and updates to Datalink columns may be performed transactionally. For this reason, object data referenced through a Datalink are continuously available.

In this system, the tasks of managing the data and managing the metadata have been split between the database and the file system or Datalinker. The Datalinker is essentially a distributed file system. However, when a Datalinker controls a “linked” object, one referenced by a Datalink, it must cooperate with the database to give the appearance that the file is actually stored in the database and under database control. This cooperation includes:

- Limiting file access to database clients and enforcing database security and permissions.

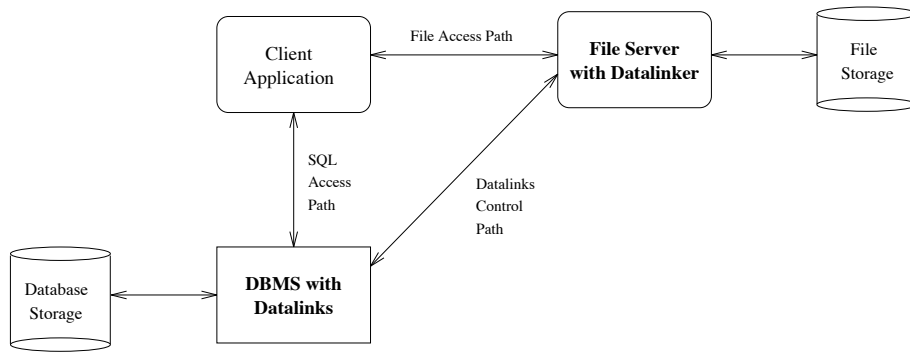
If a file system were to permit unsanctioned deletion or write, the object contents would be inconsistent with the database link.

- Safely maintaining multiple versions of object data.

As many versions of the Datalink itself must be recoverable, the associated object data each Datalink referenced must also be recoverable.

- Return object data metadata to its previous state when a database releases control, *i.e.* unlinks an object reference.

Applications query Datalink data to locate data through SQL calls to a DBMS (Figure 1). Having queried the



**Figure 1. Paths of communication in the Datalinks system.**

database for permissions and metadata, the application attains a “pointer” to the object on the cooperating file system and may then access this data directly. All file system calls are filtered by the Datalinker to enforce database constraints.

The two components of Datalinks, DBMS and Datalinker, communicate with each other through an independent control path to ensure the consistency and referential integrity of file versions and metadata. In addition, the Datalinker utilizes a backup server to guarantee recoverability by backing up each linked file.

By separating the metadata and data management tasks associated with large object data, Datalinks exhibits the best features of both file systems and databases – ease of access and a method to associate and interrogate complex metadata.

#### **4. Comparing the Datalink and BLOB Data Types**

BLOBs [6] differ from the linked files of Datalinks in that they are wholly contained within a database. It follows that the ability to associate and query complex metadata with BLOBs is equally powerful to that of Datalinks. However, the contents of data objects stored in BLOBs are much less accessible, as access to this data relies on an import/export model.

Applications wishing to read the contents of BLOB data must export the content of the object to a locally accessible file system. The local copy of this object can be accessed through the file system and is totally dissociated from the database.

For write access, the BLOB must be exported. Once in the file system, an application modifies the data and re-imports the BLOB to the database. As a consequence, there is no reliable way, built into the data type, to have a BLOB always available and consistent for concurrent applications and multiple user environments. While applications may

agree to serialize access to a BLOB, this is not a general solution that is enforced and is susceptible to poorly behaved applications breaking the agreement. This problem arises as a database has no knowledge of the contents or access methods of an object. An object can only be modified after it is exported and the database has relinquished control. For comparison, in Datalinks, the database and file system cooperate to manage write access to the object and no data copying is required as the database and file system share a single copy of an object.

When updating a BLOB, replacing the contents of the data type in a table, there are several options for how to manage recoverability. The most common of which is to not log BLOBs, as logging implies copying the whole object to the database log. BLOBs may be arbitrarily large and may overflow the database log. Most applications, unwilling to incur the expense of copying large objects to the log, turn off logging for BLOBs. In this case, BLOBs can neither be versioned nor recovered in case of failure, as there is no copy of this object outside the database. Consider that the database fails and that a BLOB has been updated since the last backup. After restoring the database, the updates to the BLOB are gone as they are neither present in the backed up database nor in the database log.

Alternatively, applications needing recoverability may elect to pay the cost of copying large objects in exchange for the ability to recover BLOB data to the current state. For applications that wish to frequently update BLOBs, many versions of the same large object would need to be logged and the log may grow without bound.

With either logging on or off, the BLOB update model is less powerful than the Datalinks model which supports multiple versions of large objects which are all recoverable. Furthermore, the Datalinks update model provides these guarantees without additionally taxing the DBMS or the database log with object export, import and copying.

## 5. Modifying Linked Files

Having compared the BLOB and Datalink object type, we describe the details of the Datalinks implementation and the role of the cooperating file system in maintaining consistency and referential integrity. We confirm that Datalinks offers a significant advantage in its ability to efficiently modify object data in a database system.

### 5.1. Database Control over Linked Data

Databases store the contents of a Datalink data type in its columns. Application clients query the contents of the database to attain the location of object data in the form of a server name and path to the object on the local server,  $\langle server :: path \rangle$ .

When a remote object is linked, an entry in a Datalink type column is created, the database makes a call back to the cooperating file system on the Datalinks control path. This call informs the cooperating file system that the file has been linked and is now under database control. For a link event, the Datalinker stores the current permissions of the object and modifies the permission of the object to be read only. In this way, the file cannot be deleted or changed without the databases knowledge. By placing the file under database control, referential integrity and consistency will be enforced.

Also on linking a new object, the cooperating file system makes a backup copy of this file and stores the backup copy remotely on a backup server. To give the appearance that data is stored in a database, the file system must ensure that any linked versions of data are recoverable. Backing up this file will guarantee the future recovery of the databases current state. To see that this is necessary, consider performing a point in time restore to a previous version of a database with linked object data. The linked object data may have been modified since that previous database. But, the object data was backed up at link time and may be recalled from a backup server.

By backing up data on the cooperating file system, Datalinks again takes advantage of its dual interface to object data. The Datalink column is logged at the database for recoverability, this is appropriate as the logging can be done efficiently and allows transactions to complete rapidly. The object data is backed up at the file system, which already has methods to deal with backup and restore and solves the problem BLOBs data has with recoverability.

Datalinks does make one concession to performance at the expense of strict consistency. File backup for linked data is performed asynchronously from the transaction which links the object. This prevents a transaction from stalling while waiting for the potentially slow process of file backup. The linked data would be potentially unrecoverable were

the file system to fail before the backup is complete. Note that this would require a media failure not just a system failure. Therefore we view trading off this potential inconsistency for transaction performance highly advantageous. Using this model for linking data, a Datalink may be updated or created with no data copying overhead within the transaction. This differs greatly from BLOB data which must be imported into the database.

### 5.2. Releasing Database Control

When a Datalinks database releases control over linked object data, the database removes the Datalink, records the change in its log and informs the cooperating file system that the data has been unlinked. At this time, the cooperating file system releases database control over the object and returns the object permissions to the state they were in before link. Again this process can be performed efficiently without copying object data out of the database.

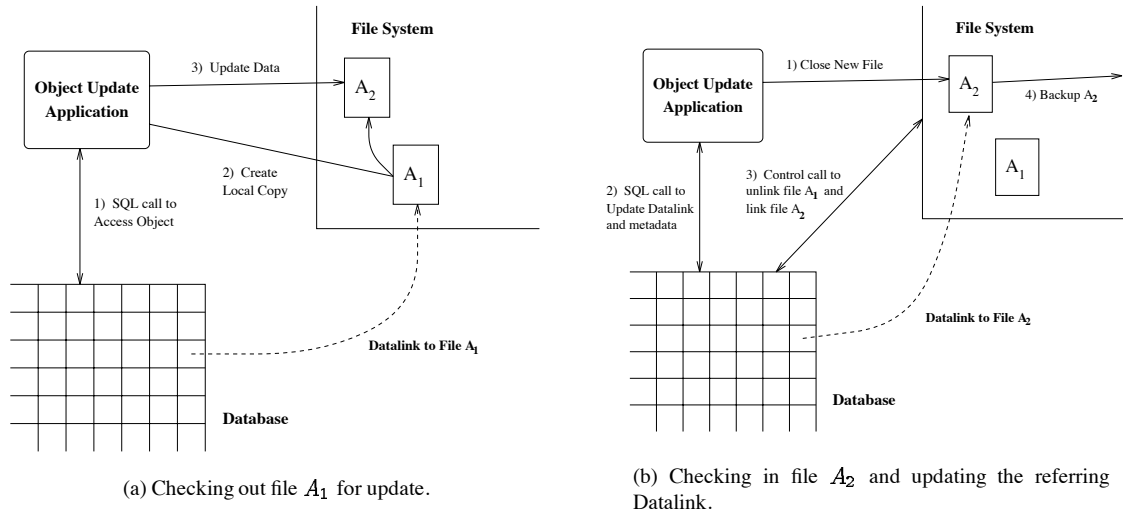
### 5.3. Modifying Object Data

With the link and unlink facility provided, the Datalink data type outperforms the BLOB data type when modifying object data. An application can unlink data, modifying it in the file system, and then relink the object data. This modifies object contents without performing any data copying. Furthermore, all database transactions complete efficiently without waiting for potentially slow file I/O to complete. While advantageous over current technology, we view this update model as inadequate for applications that wish to write shared data. By requiring applications to unlink data to modify it, we require data to be unavailable for reading whenever the data is being written.

To address the needs of concurrent applications, we expand the capabilities of the Datalinks database and cooperating file system to permit object data to be updated transactionally and always available for read. This is possible as we can control access to the data from the cooperating file system. To make data continuously available in the presence of updates, we add the capability for the Datalink data type to be update transactionally, *i.e.* **SQL Update**, and we let the applications write linked data, accessed through the database, on the cooperating file system. The cooperating file system supports two types of write in a different fashion.

#### 5.3.1 File Append

When appending a file, no changes occur to data present in the pre-appended version of the file. The subsequent file version consists of all bytes in the initial version followed by the appended bytes. This property is invariant over arbitrarily many append operations. As a consequence, the



**Figure 2. Example of an application performing file update.**

current version of the file implicitly stores all past versions of a file.

For linked files that support append, a modifying application attains write access, an append only handle, on the existing files, with the understanding that no existing bytes in the files will be modified or deleted. When the modifying application closes its append only handle, it calls **SQL Update** to inform the DBMS that a new version of the file exists.

Calling **SQL Update** in this case does not change the value of the linked file. However, on **SQL Update** of a Datalink column, an asynchronous backup initiates at the Datalinker. This helps satisfy the recoverability guarantee.

While an application appends a linked object on the cooperating file system, read requests can still be serviced. These read requests are directed to the old version and the Datalink metadata correctly encodes the length of the linked version, omitting appended data. The Datalink metadata will not refer to appended data until the append handle is closed and the Datalink updated.

### 5.3.2 File Update

While many applications have append only semantics, others need a more general model for update that allows random access for writing. Applications that use general update on object data must perform a check in and check out. To modify a linked file, an application creates a local copy of the file data and serializes write access through a check-out facility (Figure 2a). This local copy is the checked out version of the file. Since there is a private copy, the linked file that was copied from is still available in an unaltered

state through the database's Datalink data field. The application modifies its local file and on completion checks it back in (Figure 2b). At check in, the application performs an **SQL Update** to the referring Datalink data field. The DBMS modifies the Datalink column to reference the new file atomically. At all times during file update, at least one version of this file is externally linked and therefore continuously available. As with file append, the **SQL update** triggers a backup of the newly linked file on the cooperating file system.

After the update, both file versions reside on the local system. Applications that have open handles to the old version may continue to use those handles. The file system guarantees to retain a file as long as there are applications with open file handles. However, subsequent accesses to the Datalink data field return a handle to the more recent version.

## 6. Backup/Restore for Datalinks

For a variety of reasons, including media failure and data corruption, a database may need to be restored. Datalinks provides flexible restore capabilities such that a database may be recovered to an arbitrary prior point in time, including the time of failure. Restoring to an arbitrary point in time consists of restoring the database to the backed up version that most closely preceded that point in time, and then replaying a log to roll that database version forward to the desired state.

Having restored a database to a point in time, the file contents on cooperating file systems must be made consis-

tent with the Datalink columns in the database. This may mean replacing current files with the previous versions of the same files. A cooperating file system does not have a log, equivalent to a database, and cannot be rolled forward. Each unavailable file must be explicitly recalled from a backup server. For each Datalink field, the DBMS requests that the appropriate Datalinker recall the version that it references. Having recalled all referenced versions, the database and the local file systems are in a consistent state.

The Datalinker must have the capability to recover all files that have been linked by all restorable databases. To guarantee that the backup server has all these files, every time a file is linked, an automatic call requests that the Datalinker initiate a backup. For consistency, it would be ideal to perform file backup as part of the database transaction that creates or updates a file link. However, file backup requires an unpredictably long amount of time, depending upon file size, destination and network parameters, and would prevent a database transaction from committing in a timely fashion. Consequently, Datalinks elects to perform file backup asynchronously from the corresponding database transaction.

Until the Datalinker completes this asynchronous backup, there is potential for file system failure to create inconsistencies and referential integrity problems. Datalinks limits this potential for inconsistency by barring an unlinking or modification of a file until backup on the file completes. The window of inconsistency arises solely as a result of asynchronous backup, were the backup to be performed within the transaction, there could be no potential for inconsistency. We choose asynchronous backup as a design decision and trade off latency in the transaction for a small window where the Datalinker might lose a single version of a file.

A more efficient backup system allows us to limit the window of inconsistency and also have files more highly available for write modifications. With standard file backup, Datalinks would have to backup the entire shared file every time an individual user made a modification. For larger files, whole file backup would render the document unavailable for the period of time needed to transmit the whole file to a backup server for each update. For fine grained consistency and write sharing, backing up the file each time turns out to be prohibitively expensive. With a delta backup mechanism [2] that transmits only the modified bytes, a small portion of the entire file, backup time decreases and file availability increases.

## 6.1. Backup Semantics in Datalinks

The Datalinker performs file backup when any link operation occurs. When linking a new file, *i.e.* not previously backed up, the system merely performs a conventional file

backup. However, when linking a file as a result of an SQL Update, we reduce the amount of data transferred to the server by sending a delta file with respect to the previously linked file in this update. Recently, efficient algorithms have been developed for delta encoding binary inputs at fine granularity [3, 1] and have been applied to file system backup and restore [2]. The semantics of updating a Datalink imply that two subsequent versions of the same file exist. Datalinks takes advantage of the presence of an older file version to compactly encode the backup of the new version.

For example, a system updates its copy of linked file  $A_1$ . By convention we will call the  $n^{th}$  version of file  $A$ ,  $A_n$ . Consider the process of modifying a file with Datalinks update facility. The Datalinker on the local file system has previously marked  $A_1$  read only. An agent attempts to check out file  $A_1$  with the intention of modifying its contents. If the agent has write permission and no other process has checked out the file for append or update, the agent generates local copy,  $A_2$ , that is writable and private. At this point, the file system contains identical files  $A_1$  and  $A_2$ , where  $A_1$  remains linked and marked read only (see Figure 2a). After modifying the contents of  $A_2$ , the agent checks in the file. For the duration of  $A_2$  being checked out, access to unaltered file  $A_1$  continues to be available through the Datalink column in the database.

When the agent checks in  $A_2$ , it emits an SQL Update to the associated Datalink column that includes the new file name and perhaps additional metadata (see Figure 2b). The metadata and Datalink column are updated atomically to reference file  $A_2$ . The semantics of this update process cause the DBMS to instruct the Datalinker to mark file  $A_1$  for future garbage collection. This file may no longer be accessed through the database. Other agents that currently have a read-only handle on  $A_1$  can continue to use it, which necessitates not garbage collecting this file immediately.

At the same time, the Datalinker initiates an asynchronous backup of  $A_2$ . The DBMS has no knowledge of file versions on the Datalinker's file system, it need only be aware that a new file has been linked and must be backed up. With the request for file backup, the DBMS includes metadata to identify the previously linked file. When the Datalinker receives this request, it realizes  $A_2$  is a version of  $A_1$  and may choose to back up this file by sending a delta compressed encoding,  $\Delta_{A_1, A_2}$ .<sup>1</sup> The backup server stores this file and therefore has an indirect backup of  $A_2$ . This backup may later be recovered by restoring both the delta file,  $\Delta_{A_1, A_2}$ , and the additional version from which the delta file was taken,  $A_1$ , and running the reconstruction operation on these files.

If we repeat this process for each modification to a file,

---

<sup>1</sup>If the unlinked file resides on a remote file system, the Datalinker will not compute a delta file.

after creating  $n$  versions of file  $A$ , the backup server contains the following files:

$$A_1, \Delta_{A_1:A_2}, \Delta_{A_2:A_3}, \dots, \Delta_{A_{n-1}:A_n}$$

This storage sequence is a “forward delta chain” [2]. Delta chains are an optimally compact version storage method. However, this storage necessitates all dependent files to be available to restore a file. Under this system, to reconstruct an arbitrary version  $A_i$ , the algorithm must run the inverse delta compression algorithm iteratively for all intermediate versions 1 through  $i$ . The time required to restore a version is dependent upon the time to restore all of the intermediate versions. In general, restoration time grows linearly with the number of intermediate versions.

There exist version storage methods that provide faster reconstruction in exchange for degraded file compression [2], but these methods are either inappropriate for binary data or require the maintenance of additional files at the client. In Datalinks, the reconstruction of a file from its deltas occurs infrequently – only for database restore. We therefore elect delta chain storage as a design decision to optimize the backup process for time and space. To prevent poor performance for delta reconstruction, we have two methods to limit the number of intermediate versions, *i.e.* the number of delta compressed versions between un-encoded files, and therefore bound the maximum number of files than the Datalinker will ever need to restore. First, we choose a minimum compression at which we transmit a delta file; at compression worse than this low water mark we send un-encoded files. In addition, we pick a maximum number of delta files to be sent without an un-encoded file. With these two heuristics, we balance the complexity of restoring files with the benefit of delta compression. Both the low water mark and maximum number of versions parameters are adjustable and merit study to determine desirable values.

## 6.2. Version Compression

By reducing the size of the files backed up, the cooperating file system can support many update operations without having the amount of data to back up explode. In general, we can expect the amount of data that needs to be backed up to grow in proportion to the size of the modified data. This differs from a naive backup implementation which sends whole files every time. In this case, the data to be backed up would grow as the product of the file size and the number of backups performed.

Previous, experiments indicate that for weekly backups, one can expect a delta file to be 10 to 100 times smaller than the original file [3]. For our application, we expect even better compression as the time granularity of file modification is less than weekly.

For general file update, the cooperating file system runs a delta compression algorithm [3] on its local versions of the linked and unlinked data. However, many types of data, such as newsfeeds and system logs, have append only semantics. These data have obvious delta files implicit from the modifications. For append only changes in our system, the delta file consists only of the additional bytes tacked on the end of the file, and a delta compression algorithm need not be run.

In either case, a compact file encoding exists and Datalinks can transmit this to a backup server in lieu of the whole file. The time required to perform backup is now proportional to the number of changed bytes, rather than the file size. It is noteworthy that these delta files, like un-encoded files, can be further reduced with standard file compression [1].

## 7. Conclusions

We have presented a system in which large object data may be placed under the control of database where it may be associated with arbitrarily complex metadata in a consistent fashion. At the same time, the data may be read and written through a file system interface. The Datalinks database and cooperating file system provide for the continuous availability of large object data in the presence of write for concurrent applications. Additionally, we have shown this model for the management of object data to be both more efficient than the BLOB data type, and to provide more robust consistency and recoverability guarantees.

Previously, to modify data in an externally referenced or linked file, Datalinks required an application to unlink, edit, and relink that file. This resulted in externally referenced data being unavailable for the duration of a file update and was unacceptable to many of the Datalinks user community. We have extended Datalinks to permit the update of currently linked files. The Datalinker grants append only handles to applications that modify data in this way and applications that require more general update capabilities may use check-out and check-in to generate a local file copy while the original file is continuously available. However, on each update, the Datalinker must back up the newly linked file. The Datalinker can use delta compression to minimally encode modifications to files and may back these files up without transmitting all the contents a file. This allows Datalinks to efficiently support the online update of externally referenced files with confidence that the cost of backing up such data will not explode in the presence of frequent modifications.

## References

- [1] Miklos Ajtai, Randal Burns, Ronald Fagin, Darrell Long, and Larry Stockmeyer. Compactly encoding arbitrary inputs with differential compression. IBM Research: *In Preparation*, 1997.
- [2] Randal C. Burns and Darrell D. E. Long. Efficient distributed backup with delta compression. In *Proceedings of the 1997 I/O in Parallel and Distributed Systems (IOPADS'97)*, 17 November 1997, San Jose, CA, USA, November 1997.
- [3] Randal C. Burns and Darrell D. E. Long. A linear time, constant space differencing algorithm. In *Proceedings of the 1997 International Performance, Computing and Communications Conference (IPCCC'97)*, Feb. 5-7, Tempe/Phoenix, Arizona, USA, February 1997.
- [4] Judith R. Davis. Datalinks: Managing external data with DB2 universal database. <http://www.software.ibm.com/data/pubs/papers/datalink.html>, August 1997. Prepared for the IBM Corporation by InfoIT, Inc., a Database Associates Company.
- [5] B. Hwang, I. Jung, and S. Moon. Efficient storage management for large dynamic objects. In *Proceedings of the 20th EUROMICRO Conference*, September 94.
- [6] IBM. *DATABASE 2 SQL Reference - for Common Servers*. Part No. S20H-4665-00.
- [7] P. Jackson. Unleashing the database. *Management Computing*, 13(12), December 1990.
- [8] J. Martinez. The design of an extensible multimedia library for an OODBMS. In *Seventh International Workshop on Database and Expert Systems Applications (DEXA)*, pages 208–213, 1996.
- [9] Nelson Mattos, Jim Melton, and Jeff Rickey. Database language SQL – Part 9: Management of external data (SQL/MED), June 1997. Available at <ftp://jerry.ece.umassd.edu/isowg3/x3h2/1997docs/97-233.pdf>.
- [10] M. Rennhackkamp. Extending relational DBMSs. *DBMS*, 10(13):45–6,48, December 1997.
- [11] M. F. Riley, J. J. Feenan, Jr., J. L. Janosik, and T. K. Rengarajan. The design of multimedia object support in DECrdB. *Digital Technical Journal*, 5(2):50–64, 1993.
- [12] T. Sacks. BLOBs and the need for FDDI. *Telecommunications*, 25(9), 1991.
- [13] B. Sayrs. Architecting multimedia database systems. *Object Magazine*, 6(12), February 1997.
- [14] M. Ubell and M. Olson. Embedding image query operations in an object-relational database management system. In *Proceedings of the SPIE – The International Society for Optical Engineering*, pages 197–203, 1995.