

# Database Managed External File Update

Neeraj Mittal  
The University of Texas at Austin  
Austin, TX 78712, USA  
neerajm@cs.utexas.edu

Hui-I Hsiao  
IBM Almaden Research Center  
San Jose, CA 95120, USA  
hhsiao@almaden.ibm.com

## Abstract

*RDBMS's have evolved to an extent that they are used to manage almost all of traditional business data in a robust fashion. Nevertheless, a large fraction of unstructured and semi-structured data continues to be managed by file systems. As companies increasingly depend on non-traditional data such as web pages and images for their daily business operations, it becomes more and more important to provide higher degree of integrity, security, and reliability to the data stored in file systems. DataLinks technology developed at IBM Almaden Research center achieves this by providing a vital integration between RDBMS and file system. It enables DBMS to manage files residing in file systems as though they are logically within the database.*

*Current DataLinks technology supports only read access to external files that are being managed by DBMS. This severely restricts the applicability of DataLinks technology in transaction oriented and/or e-business applications. Traditional database systems enforce ACID properties for database update. Extending these properties to cover both external files (such as web pages) stored outside of a DBMS and metadata stored in the DBMS is a hard problem. This is because files are updated through standard file system API while metadata, which reference the files, are updated through database API.*

*This paper describes our experiences in the design and prototyping of an advanced DataLinks technology that supports database managed external file update. This enhanced capability makes DataLinks technology an even more attractive solution for managing world's data.*

## 1. Introduction

In the past decade, RDBMS's have evolved and matured to an extent that they are used to manage almost all of traditional business data in a robust fashion providing integrity, security and reliability. Nevertheless, a large fraction of unstructured and semi-structured data continues to be managed by file systems for a number of reasons. Firstly, inherent overhead of data access in RDBMS leads to substantial

application performance degradation. Secondly, data that has to be "streamed" to the client and delivered within a specified time period in order to be meaningful, such as audio and video clips, is best managed by specialized servers optimized for such data. RDBMS simply does not know how to handle it. Finally, physically storing files in the form of LOBs/BLOBs inside the database entails increasing access overhead and/or changing access API which is undesirable due to a large number of legacy applications.

There are scenarios where a file residing in a file system bears close relationship to the traditional data stored in an RDBMS. For example, a video merchant stores attributes associated with movies, such as cast, category, inventory and price, in an RDBMS that could be used for search and analysis. In addition, (s)he stores clips of the same movies as files in the file system for preview purposes. Later, if the merchant stops selling a movie, both the clip, stored in the file system, and the metadata, stored in the RDBMS, for the movie should be deleted or archived. With conventional database and file system technologies, it is difficult to maintain this level of coordination **automatically** because file system and RDBMS function independently. What is needed is an *extended database (or web content-management)* system that allows RDBMS to manage data in a file system without physically importing the files into the database. This enables applications to continue accessing the files using existing file system API with minimal changes and/or performance penalty. RDBMS extends referential integrity, access control, and backup and recovery to the *external* files while allowing them to be stored in *close proximity* to the application to reduce network traffic and maximize application performance.

IBM Almaden Research center has developed DataLinks technology [7] that provides the vital integration between RDBMS and file system. It enables DBMS to manage files residing outside the database as though they are logically within the database. Data in files does not need to be physically imported into the database. An external file is put under database control by *linking* it to the database. Control to a file can be removed by *unlinking* it. DataLinks guarantees referential integrity to external files

that are linked to the database and provides access control, automatic backup and restore capability within transactional environment that are crucial for data management. Enterprises can manage files on multiple distinct file servers within a DataLinks database, allowing robust centralized control over distributed resources across intranets. Applications can continue to access data in files directly through file system API. DataLinks technology is available commercially since 1998 (DB2 UDB V5.2). Several corporations and institutes have deployed DataLinks technology to provide database management of distributed business and engineering data stored in operating system files [7, 10].

Besides IBM, database vendors such as Oracle and Informix have also been developing technologies for integration of database and file system. Microsoft is also working on a similar technology. However, its approach has not been made public. Unlike DataLinks's approach, both Oracle's iFS [9] and Informix's IXFS [2] store file data in database tables as LOB or BLOB. Oracle's iFS technology provides a file system API enabling applications to access LOB/BLOB data as if it were stored on a mapped network disk drive. Informix's IXFS technology takes a slightly different approach. It provides a middleware that intercepts file system calls. Accesses to file stored in the database are converted into SQL requests. The results returned from the database are first formatted into file system objects and then returned to applications. A detailed comparison among different approaches is beyond the scope of this paper. At a very high level, both Oracle's and Informix's approaches incur extra overhead in read/write accesses as they require database processing to read/write files from/to LOB/BLOB column. In contrast, DataLinks imposes far less overhead as it is only involved in open and close of the file and does not interfere in read/write accesses to it. Further, DataLinks is far more flexible and scalable since the files (or web pages) under its control can be distributed across multiple systems (or web servers) in the network.

With DataLinks technology, permission to access files can be managed by either file system or DBMS. Currently, when write access to an external file is controlled by DBMS, the file becomes read-only and any update to the file by an application is rejected. To update such a file, an application has to first *unlink* the file, update it and finally *link* it again. Clearly, this approach is quite inefficient and a more efficient and simpler solution is desirable that allows an external file to be modified while a reference to it continues to exist in the database. In the previous example, if the merchant wishes to modify the movie clip (s)he has to temporarily break off the association between the movie clip and the movie's attributes in the database which is undesirable. Besides, update is an essential operation in the traditional transaction processing systems and the new e-business applications and its absence severely restricts the applicability of DataLinks technology in those areas.

Traditional transaction systems enforce a set of properties for database update referred to as ACID (atomicity, consistency, isolation, and durability) properties [4]. Extending them to file data stored outside of a DBMS is a hard problem. This is especially true when files can be updated through standard file system API. One major problem is defining the transaction boundary for file update as it needs to avoid long transaction lock yet be able to guarantee data consistency. Further, DataLinks only controls file access privileges but does not control read/write operations to the file itself. Since DBMS has no control over application's file access behavior, enforcing ACID properties for external file update efficiently has turned out to be a non-trivial task. In this paper, we present our experiences in the design and prototyping of an advanced DataLinks technology that supports database managed *in-place* file update. Since most static web pages are stored as files in traditional file systems, the technology can be applied to maintain the consistency and referential integrity between a web page and its metadata, referencing the page, stored in a DBMS.

The remainder of this paper is organized as follows. Section 2 gives an overview of the current DataLinks technology. A detailed description of the technology can be found in [5]. Section 3 discusses various approaches for providing update support. We present our approach for update support, termed update *in-place*, in Section 4. Finally, Section 5 summarizes our work and describes the current status of the prototype. In the paper, we omit the word external when it is clear from the context that the file being referred to is not physically stored in a database table but resides in a file system.

## 2. DataLinks technology

DataLinks technology consists of the following major components: a new data type termed as DATALINK which has been proposed to be a part of the SQL Management of External Data standard [8], a new *DataLinker* component and an extension to the RDBMS engine known as *DataLinks engine*. *DataLinker* is further divided into two sub-components: DataLinks File Manager (DLFM) and DataLinks File System (DLFS) [1]. Figure 1 shows the architecture of the DataLinks technology. DLFM and DLFS components reside at file servers where the database managed external data/files are stored. DLFM consists of one main daemon and several child processes running in user space. DLFS, sitting between logical file system and native/physical file system, is built as a virtual file system (VFS) [11, 3] layer on UNIX and a device driver on NT.

### 2.1. DATALINK data type

A DATALINK value contains a pointer to the external file in the format of a URL (uniform resource locator)-*protocol://server-name/pathname/filename*. As shown in

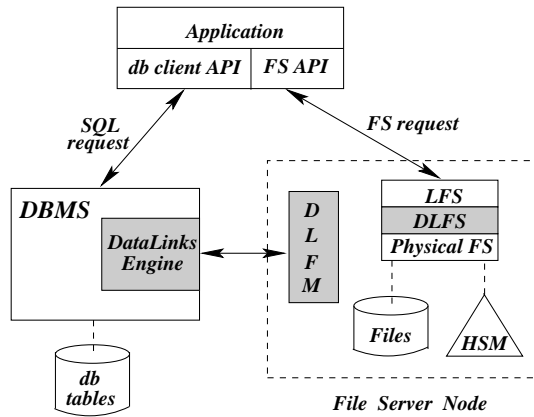


Figure 1. The architecture of DataLinks.

Figure 2, Employee table has a picture column whose data type is DATALINK. The values of the picture column are references to the image files stored in local or remote file servers. A DATALINK column can reference files in multiple file servers and a file server can be referenced by multiple DATALINK columns in one or more databases.

DataLinks allows DBMS to exercise various degrees of control over the external files. While defining a DATALINK column, a range of options can be specified for managing the files referenced in the column such as integrity option, read permission, write permission and recovery option. Table 1 summarizes the various control modes. We represent a control mode using three letters corresponding to three attributes - *referential integrity*, *read access control* and *write access control*. The first attribute indicates whether DBMS guarantees the referential integrity of the reference to the file in the table. It could take values n (referential integrity not guaranteed) and r (referential integrity guaranteed). The second and third attributes specify who controls the particular access (read or write) to a file. Their possible values are: f - file system, b - blocked and d - DBMS. We assume that read access to a file is never blocked. We use the letter x if we do not care about the value of the corresponding attribute. For convenience, we say that a file is under *full control* of the database (or DBMS) when neither read nor write access is under the control of the file system, otherwise it is under *partial control* of the database. More details regarding various options available while defining a DATALINK column can be found in SQL reference manual [6].

## 2.2. DataLinks File Manager (DLFM)

DataLinks File Manager resides on each file server and manages files stored on that server. Whenever a reference to a file is inserted or deleted from a DATALINK column, DataLinks engine contacts the appropriate DLFM directing it to start (corresponds to link operation) or stop (corresponds to unlink operation) managing the file. It is the re-

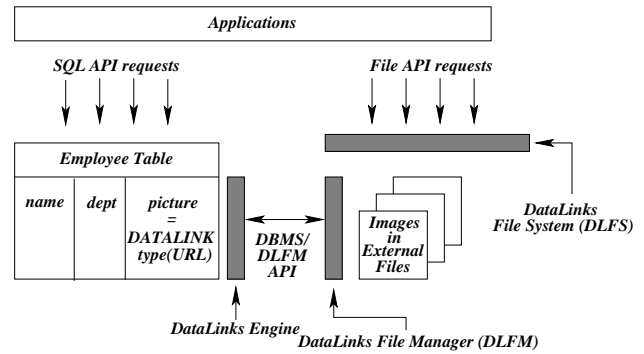


Figure 2. The application's view of DataLinks.

Table 1. A table summarizing the control modes in DataLinks.

Control Mode	Referential Integrity	Read Access Control	Write Access Control
nff	No	FS	FS
rff	Yes	FS	FS
rfb	Yes	FS	Blocked
rdb	Yes	DBMS	Blocked

n - referential integrity not guaranteed  
 r - referential integrity guaranteed f - file system  
 b - blocked d - DBMS x - don't care

sponsibility of the DLFM to ensure that the link/unlink operations are executed in the same transaction context as the one in the host database. To do this, the operations done in DLFM are treated as a sub-transaction of the host database transaction and the DLFM maintains its own repository about the transaction state and about files that are linked to the database.

When a file is linked to the database, DLFM applies the constraints for referential integrity, access control, and backup and recovery as specified in the DATALINK column definition. For example, in rdb mode, when DBMS controls the read access to a file, DLFM takes-over the file by changing its ownership and marks it "read-only", thereby blocking any update to the file. In rfb mode, the file ownership is not changed but the write permission to the file is disabled, effectively making it read-only. All these changes to the DLFM repository and file system are applied as part of the same DBMS transaction as the initiating SQL statement. Later, if the SQL transaction is rolled back, the changes made by the DLFM are undone as well.

DLFM is implemented as a main daemon with several child daemons and child agent processes coordinating with

each other to provide the required functionality. Detailed description of the functionality of each daemon is beyond the scope of this paper and is provided elsewhere [5]. We only describe the child agent and upcall daemon here as they are necessary to understand the paper. When a connect request from a database agent is received, the main daemon spawns a child agent which then establishes a connection with the requesting database agent. All subsequent requests (link/unlink operations) from the same connection are served by this child agent. The upcall daemon, on the other hand, services requests from DLFS to check the control mode and verify access permissions of linked files.

### 2.3. DataLinks file system (DLFS)

DataLinks File System is implemented as a virtual file system (VFS) layer between the logical file system (LFS) and the underlying physical file system (JFS, UFS etc.). DLFS, in coordination with the upcall daemon (part of DLFM), ensures that all accesses to the linked file are authenticated by DBMS and they do not violate any integrity constraints. DLFS will, for example, reject a user-level request to rename or delete a file referenced by database. This avoids “dangling pointers” in which a file is referenced in the database but the actual file does not exist.

DLFS intercepts calls such as `fs_open()`, `fs_close()`, `fs_remove()`, `fs_rename()` and `fs_lookup()` made by LFS to the underlying file system. Upon intercepting a call, it does additional processing before passing down the request to the underlying file system. For example, when an application issues an `open()` system call, the call is handled by LFS which first calls `fs_lookup()` to determine if the file exists. DLFS intercepts the call, verifies the token embedded in the file name (if it exists) and then calls the underlying file system’s lookup routine. If the lookup operation is successful, LFS gets back a pointer to the vnode of the file. It then allocates a file descriptor and a file structure for the file being opened, and inserts them into the system open file table. Finally, it calls `fs_open()` which is again intercepted by DLFS. DLFS does its own processing (described later) before calling the underlying file system’s `fs_open()` routine.

### 2.4. Limitations in current DataLinks technology

The major limitation in the current DataLinks design is the lack of support for database managed update to an external file which is essential for achieving complete integration between RDBMS and file system. To that end, there is a need to extend ACID properties to the update of external files as well as enforce database managed access control. The focus of this paper is to add two additional control modes to DataLinks, namely `rfd` and `rdd`. Our design philosophy is to minimize changes to DLFS component and add any new code/logic for update support in DLFM component whenever possible. This is because different file sys-

tems and platforms require different DLFS implementation. Restricting changes to DLFM will enhance portability and reduce code maintenance headache.

## 3. Managing file update with database

The most desirable approach for database managed file update is to allow applications to continue updating the files through the traditional file system API while providing transaction semantics for the update operation. Maintaining traditional file access paradigm will provide an easy migration path for current file system based application and help its acceptance by the users. With this approach, an application first gets update permission from the controlling database and then accesses the file using the traditional file system API. The application need not make a separate copy of the file and the update operation can be done in-place. We term this approach *update in-place* (UIP).

The second approach is to adopt the traditional *check-in/check-out* (CICO) mechanism and again delegate the responsibility of file access serialization to DBMS. With this approach, DBMS controls who can checkout what file and in which mode. An application first checks-out the file it wishes to update. This, in turn, places a lock on the file in the database. Before the lock is removed explicitly, no other application is allowed to check-out the same file. When the application has completed its update, it checks-in the file to the DBMS which then releases the file lock. This approach does not work well if applications are not well-behaved. For example, an application could potentially check-out many files in advance, unnecessarily blocking out applications that immediately need to access those files. This will result in significant performance degradation. Unlike in the first approach (UIP) where applications acquire an implicit lock on file open (corresponds to `begin_transaction`) and release the lock on file close (corresponds to `end_transaction`), in this approach, the lock is acquired and held for longer time, thereby curtailing concurrency. Further, the DBMS needs to keep track of who has checked out what files, which requires an extra database update operation for both check-out and check-in requests.

Alternatively, applications can first make a private copy of a file before updating it. This is similar to the check-out/check-in mechanism, except that making a private copy does not lock the file. Multiple applications are allowed to make their own copies of the same file. We term this approach *copy and update* or CAU. With this approach, the onus for maintaining consistency can be shifted to applications, thereby allowing applications to define their own consistency criterion. An application first makes a *copy* of the file it wishes to access. Multiple applications then coordinate to access the file in accordance with the agreed consistency criterion; DBMS simply performs access control. An application finally “checks-in” the file to the DBMS which

then performs necessary actions to provide backup and recovery. Unlike as in the former two approaches, transaction semantics is not enforced by DBMS and applications themselves need to worry about update atomicity. Believe it or not, CAU is a mechanism used by many development labs. When deadline for product release is near, many developers may work on the same file concurrently. The first one who completes his/her work will lock the file and integrate the work. A subsequent “checks-in” by another developer will have to “merge” his/her changes with all changes integrated after his/her private copy was made. As readers may point out that a lost update can occur with this approach, if not done carefully, and it does occur.

### 3.1. Supporting file update with DataLinks

Our design for DBMS managed file update is based on the update in-place approach, where DBMS is responsible for enforcing file access serialization and update consistency. With this approach, there are various ways to define transaction boundary. The simplest one is to make every file access (through `fs_readwrite()` entry point) a separate transaction. Thus every update access to the file creates a new recoverable version. However, there are several problems with this definition. Firstly, it is not useful in practice. A file may be written multiple times between file open and close. An intermediate file state may give inconsistent information and thus is not very useful. In practice, if failure occurs before the file has been updated and closed, users want to roll back the partial update to ensure consistency. Consequently, treating each file write operation as a separate transaction does not provide any real benefit to users. Secondly, allowing multiple users to open and update a file concurrently will result in chaos and lost updates. Serializing file access at read and write boundaries does not and cannot prevent lost updates. Finally, treating each read and write operation a separate transaction requires DLFS to intercept every call to the `fs_readwrite()` entry point which, in turn, may require communication with DLFM (upcall daemon), thereby degrading the file system’s performance substantially. A more practical way to define transaction boundary for external file access is to treat all file accesses by an application between a file’s open and its close as part of the same transaction. The open and close of a file then correspond to `begin_transaction` and `end_transaction`, respectively. If an application wants to update multiple files within a user transaction, the nested transaction concept can be applied.

### 3.2. Performance consideration

Like in all software design projects, we want to minimize the overhead in the most frequently accessed path and put the burden on the rarely accessed path. Since our main focus is to support workload consisting of mostly read and

occasional update, our design tries to minimize the overhead in the read access path. Accessing static web pages in a web server is a real world example of such a workload. Without update support, prior performance study (done on a 200MHz PowerPC 604 machine) has shown that DataLinks technology incurs less than 3ms overhead for retrieving a DATALINK column, including access token generation, at the host database. At the file system side, the DLFS layer and the token validation processing add about 1ms overhead to open, read, and close a file. This translates into less than 1% overhead, counting both CPU and I/O time, for reading a 1MB file with DataLinks and about 3% overhead, counting only CPU time.

## 4. Update in-place algorithm

When external files are managed by DBMS, it is desirable to treat them as if they are part of database internal data, that is, extend the enforcement of ACID properties to external file accesses. Besides ACID properties, enforcing file access control is also important because files that are stored outside of a database can, unlike database tables, be accessed directly. Specifically, file update support mechanism should have the following desirable properties:

- The *access control* is enforced for access to the file that is under the control of DBMS. Only those applications that access the file using a valid token, obtained from the database, are granted the permission. Since applications will continue to access files through standard file system API, the access token would have to be embedded in the URL or file name. Also, multiple types of access tokens are provided for different types of file access such as read, write and execute.
- *Transactional* semantics is provided for all accesses to files that are linked in `rfd` or `rdd` mode. In particular, file update *atomicity* and access serializability (*isolation*) are enforced.
- Database *consistency* between file data and its meta-data is preserved. When a file has been updated, its meta-data, if it exists, must be updated as well. To enforce database consistency, the two update operations must be within the same database transaction context.
- Committed update to a linked file and its meta-data is hardened (*durability*) via database logging and file archiving. Moreover, *backup* and *restore* of the file is executed in synchronization with the associated database.
- Applications’ accesses to a file are synchronized or *serialized* with the link and unlink processing by DLFM.

One of the major challenges was that it is not easy to store DataLinks specific information on a per file basis at

DLFS. Doing so would require making changes to the file system specific data structure which would result in portability issues and maintenance nightmare. To alleviate this problem, our design stores that information at DLFM which is platform independent. The drawback is that accessing that information by DLFS requires an upcall (IPC) from DLFS to DLFM. As an optimization, whenever a file is under full control of DBMS, it takes-over the file by changing its ownership. The following sections discuss each property and the solution adopted for its support in detail.

#### 4.1. Access control

Our implementation performs the access control check only during the file open processing. Before we describe the issues involved in its implementation, we first explain the steps taken by a UNIX file system for processing a file open request. When an application issues an open system call, LFS intercepts the call. It then invokes the `fs_lookup()` entry point passing it the name of the file along with the credentials of the process opening the file to obtain a pointer to the vnode structure of the file. Finally, LFS calls the `fs_open()` entry point with the pointer to the file's vnode structure, the access mode, and the credentials of the process to actually open the file. This decoupling of the `open()` system call into separate calls to `fs_lookup()` and `fs_open()` entry points makes it difficult for DBMS to provide support for multiple types (e.g. read and write) of access tokens. When `fs_lookup()` is called, DLFS does not know the access mode in which the file is being opened. Similarly, when `fs_open()` is called, DLFS does not have access to the type of token that was used to access the file. DLFS does not know the token type when processing the `fs_open()` call since file name is not passed in as a parameter in the call. In short, this decoupling creates problems in enforcing the access control by DBMS when the file is linked in either `rfd` (access can be without a token or with a write token) or `rdd` (access can be with a read token or a write token) mode. Thus one of the challenges is to ensure that the type of token used to access the file is consistent with the mode in which the file is being opened. Otherwise, an application could potentially use a read token to open a file for update.

The problem has been solved as follows. On intercepting the `fs_lookup()` call for a file `foo`, DLFS contacts the upcall daemon to validate the token on behalf of the user, say *Doe*. The upcall daemon on validating the token creates a *token entry* at DLFM. The entry indicates that the user *Doe* has permission to access the file `foo` till time  $t$  for  $a_{type}$ , where  $t$  is the expiration time of the token and  $a_{type}$  is the token type. Later, when `fs_open()` is called with the requested access mode, DLFS contacts the upcall daemon which then checks for the existence of the appropriate token entry. The `userid` is used instead of the `processid` in the token entry since the `processid` can be reused and therefore other users can inadvertently gain access to the file. On the other hand,

since the `userid` is stored in the token entry, once such an entry is created for a `userid`, all applications sharing that `userid` automatically get permission to access the file.

#### 4.2. Transactional semantics for file access

From the point of view of DBMS, open and close requests to a linked file correspond to `begin_transaction` and `end_transaction (commit)`, respectively. The file access is serialized, when needed, using the `fs_lockctl()` entry point of the file system to lock the file in the desired access mode. The main problem here is that if a file is not under the full control of DBMS (which can be ascertained by examining the ownership of the file) then it is not possible to determine the control mode of the file (or if the file is linked at all) without contacting DLFM (which entails an overhead of message exchange). This has an important implication for the file that is linked in `rfd` mode where DBMS controls only write access to the file. To minimize communication overhead, only weaker consistency guarantees are provided for such a file. In particular, only write accesses to the file are serialized and no locks are acquired for read accesses. This helps to avoid any upcalls to DLFM whatsoever when a file, not under the full control of DBMS, is opened for read. However, a reader of a file linked in `rfd` mode could potentially read inconsistent data since read-write serialization is not enforced.

Further, to minimize the overhead of upcalls when none of the file accesses (read or write) is under the control of DBMS, the file is made read-only when it is linked in `rfd` mode. When an application opens such a file for write, DLFS contacts DLFM through an upcall only if the `fs_open()` entry point of the file system fails. Note that request to open an `rfd` linked file for write will fail since the file has been made read-only. DLFM, on receiving the upcall, verifies that the file is indeed linked in the `rfd` mode and takes-over the file granting it write permission, otherwise it returns an error. DLFS, on receiving the approval from DLFM, again invokes the `fs_open()` entry point to open the file. The ownership of the file is released upon its closure. Since in our design, when a write request is granted, DLFM takes-over the ownership of a file linked in `rfd` mode, subsequent read requests to the same file will be rejected by the file system's access control mechanism. Consequently, our design enforces read-write serialization even though no locks are acquired for read requests.

If system fails during file update operation or processing of file close request fails, the update operation is rolled back as part of system or transaction recovery. A copy of the file is saved to an archive device/server after update to a file has completed and committed. When a failure occurs, the last committed version of the file is restored from the archive and the in-flight version of the file is moved to a temporary directory. This ensures that either all changes to a file between open and close calls complete successfully or none of

the changes survive the failure. Thus our mechanism is able to extend the *atomicity* property to external files managed by DBMS.

In summary, DataLinks provides full transaction semantics for accessing files linked under *rdd* mode. Read and write accesses to a linked file are serialized at file open time. For files linked in *rfd* mode, multiple write accesses to the same file are serialized. However, read-write synchronization is not guaranteed. In addition, atomicity of a file update is enforced by restoring the last committed version when a failure occurs.

### 4.3. Update consistency

When an external file is controlled by a database, meta-data (size, modification time, content specific attributes, etc.) of the file is normally stored in the database along with the reference (URL) to the file for reference and search purposes. After the linked file has been updated, its associated meta-data must also be updated in order to preserve database consistency. Our design automatically updates the size and modification time of a file after it has been updated. Moreover, the two updates are done within the same transaction context. As described above, DLFS sends a request to the upcall daemon in DLFM when an updated file is closed. The new file size and file modification time are passed to DLFM as parameters of the upcall. This information is then updated as part of file close processing in DLFM. However, we have not come up with a systematic way for automatically updating user meta-data (content specific attributes) associated with the file. It is a topic for our future research. For now, updating content specific attributes, if needed, is the responsibility of the application developers.

### 4.4. Coordinated backup and recovery

While it is not done regularly, from time to time, a database may be restored to a specific time in the past for auditing purposes or for fixing a serious problem. When external files are referenced and managed by a database, backup and restore of the files and database would need to be done synchronously. Supporting coordinated database and file system backup and restore has always been a hard problem even without database managed file update. With file update support, the problem has become even more complex. Fortunately, current DataLinks technology has a well defined architecture for supporting coordinated backup and restore. That has made the work of extending the current technology to cover file update support easier.

With our in-place update mechanism, the coordinated backup and restore of database and files is supported as follows. Whenever a file linked in *rxd* mode is opened for write, DLFS makes an upcall to DLFM to validate the application's access token. On receiving the upcall, besides

validating access permission, DLFM creates an entry indicating that the file is being updated, and a new version may be created. If the file update transaction is rolled back, this entry is used to identify and restore the last committed version of the file. Later, when the application closes the file, DLFS again makes an upcall to DLFM informing it of the closure. DLFM then determines whether the file has been modified using the last modification time of the file. If the file has been modified, it triggers an asynchronous archiving of the file. Any new update request to the file is blocked until the archiving completes. Each new version is associated with a database state identifier (for example tail LSN). When database is restored to a previous point in time, the corresponding files, according to the restored database state identifier, are also restored from the archive.

### 4.5. Synchronization of file access with (un)link operation

For files linked in *rxd* mode, synchronization between the write access and the transaction that unlinks the same file can be achieved quite easily and efficiently. Whenever such a file is opened for write, DLFS makes an upcall to DLFM which then creates an entry in a DLFM table, called *Sync* table, to indicate the same. When some transaction tries to unlink the file, DLFM simply needs to check for the presence of such an entry and reject the unlink request if an entry exists. The synchronization of read accesses with unlink operation can be done in a similar fashion. When an application opens a linked file for read, DLFS makes an upcall to inform DLFM of the same (for access validation if linked in full control mode). DLFM then makes an entry in its *Sync* table to indicate that the file is currently being read. Later, when the application closes the file DLFS again makes an upcall to inform DLFM to purge the read entry. Similar to the write entry, when a read entry exists in the DLFM *Sync* table, any unlink operation by other applications will be rejected. Notice that, synchronizing read access with unlink operation will add two extra database update operations and one extra upcall for every request that opens file for read, which is undesirable for performance reasons. This is the very reason that we have decided to synchronize read accesses with unlink operation for files linked in full control mode only.

Notice that, for files linked in full control mode, the read/write entry created in the *Sync* table for a file open request obviates the need for an explicit file locking by DLFS. Had we taken the same approach for files linked in *rfb* or *rfd* mode, no explicit file locking would have been needed by DLFS. However, doing so would incur additional overhead (message sending and database access) for every open call to files linked in these mode. Since our design goal is to minimize the overhead in file access, our prototype does not rely on the read/write entry in the *Sync* table to obtain read-write serialization for *rfb* or *rfd* mode.

Once a file is linked, our scheme provides the correct semantics for file access. However, a link transaction can succeed even when the file is currently open by other applications, thus creating a window of inconsistency. This problem can be rectified by making an upcall to DLFM for every file open call, even for files that are not linked (not under database control). But this is undesirable for performance reasons. We leave the elimination of this window of inconsistency as a future exercise.

## 5. Summary

IBM Almaden Research Center has developed DataLinks technology that provides vital integration between RDBMS and file system. It enables DBMS to manage files residing outside the database as though they are logically within the database. Current DataLinks technology delivered in DB2 UDB does not support database managed file update. This limitation severely restricts the applicability of DataLinks technology. To remedy this limitation, we have extended the DataLinks technology to support database managed external file update. Our scheme enables applications to update database managed files in-place through standard file system API and provides transaction semantics for the file update operation. Write access control is provided through a new update token mechanism. When read and/or write accesses to a file are managed by a database, read-write and write-write access conflicts are serialized by implicit (via Sync table) or explicit (via `fs_lockctl()`) file locking. Update atomicity is guaranteed by treating a file update between file open and close as a single transaction. After a file update has been completed and committed, an archive copy is made asynchronously. If a file update transaction aborts or a failure occurs before update has completed, in-progress version of the updated file is discarded and the last committed version of the file is restored from the archive automatically. In addition, coordinated backup and restore of a database and the database managed external files are supported by associating a database state identifier with every version of an updated file.

With the addition of database managed file update support, DataLinks has become an even more attractive technology for efficiently managing semi-structured and unstructured data in e-business and content management applications. In addition, with read and/or write accesses to external files under database control, DataLinks can be used to enhance file system and web server security. We have completed a prototype of our update support mechanism on DB2 UDB with AIX file system. Preliminary experiments indicate that the extra overhead of maintaining file update status at DLFM is insignificant. There is only minor difference in the response time between opening a DataLinks managed file and opening a file system managed file.

Our implementation of update in-place support, however, is far from perfect. The DBMS serializes read/write access to the external files that are linked in the full control mode but provides only limited serialization in the `rfd` mode. The write accesses to the files linked in `rfd` mode are serialized but the read access is not completely synchronized with the write access. In particular, an application can successfully open a file for update while another application has the file open for read. While solutions exist for this problem, they either incur extra overhead for all files not under the full control of database (making an upcall to DLFM from DLFS and adding an entry in the Sync table will eliminate the problem) or are difficult to implement in practice, and thus not recommended. Finding a better practical solution with less overhead is a future research problem.

## Acknowledgment

The authors would like to thank Ajay Sood for his tremendous help in understanding the existing implementation of DataLinks. They would also like to thank Inderpal Narang, Kiran Mehta, Karen Brannon, Parag Tijare, Atul Goel and members of content management team including David Choy, Henry Gladney, Shrish Agarwal, Sriram Raghavan, Atul Chadha, James Lin, and last but not least Robin Williams for their helpful suggestions and support.

## References

- [1] A. Agarwal, A. Goel, and A. Sood. *Software Design Description for DLFS on Solaris, HP-UX, and AIX*, 1996.
- [2] I. Balabine and R. Kandasamy. *File System Interface to a Database*, Aug. 1999. United States Patent Number 5937406 granted to Informix Software Inc.
- [3] B. Goodheart and J. Cox. *The Magic Garden Explained: The Internals of Unix System V Release 4*. Prentice Hall, 1993.
- [4] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
- [5] H. Hsiao and I. Narang. DLFM: A Transactional Resource Manager. In *Proceedings of the 19th ACM SIGMOD Conference*, pages 518–528, Dallas, Texas, May 2000.
- [6] IBM. *DB2 UDB SQL Reference*, v6.1 edition, 1999.
- [7] IBM Corporation. *DataLinks: Managing External Data with DB2 Universal Database*, Feb. 1999. White paper prepared by J. R. Davis.
- [8] N. Mattos, J. Melton, and J. Richey. *Database Language SQL - Part 9: Management of External Data (SQL/MED)*. ISO working draft, June 1997.
- [9] Oracle. *iFS: Oracle Internet File System*, 1999. <http://www.oracle.com/database/options/ifs.html>.
- [10] M. Papiani, J. Wason, A. Dunlop, and D. Nicole. A Distributed Scientific Archive Using the Web, XML and SQL/MED. *SIGMOD Record*, 28(3):56–62, Sept. 1999.
- [11] U. Vahalia. *Unix Internals: The New Frontiers*. Prentice Hall, Jan. 1996.