RJ 10241 (A0204-013) April 11, 2002
Computer Science

IBM Research Report

Coordinating Backup/Recovery and Data Consistency between Database and File Systems

Suparna Bhattacharya †, C. Mohan ‡, Karen W. Brannon ‡
Inderpal Narang ‡, Hui-I Hsiao ‡, Mahadevan Subramanian ‡

† IBM Software Lab, India
Golden Enclave, Airport Road
Bangalore 560017 India

‡ IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120 USA

Research Division
Almaden - Austin - Beijing - Delhi - Haifa - India - T. J. Watson - Tokyo - Zurich

Coordinating Backup/Recovery and Data Consistency Between Database and File Systems

Suparna Bhattacharya† C. Mohan‡

† IBM Software Lab, India
Golden Enclave, Airport Road
Bangalore 560017 India
(bsuparna@in.ibm.com)

Karen W. Brannon ‡ Inderpal Narang‡

‡ IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120 USA

Hui-I Hsiao‡ Mahadevan Subramanian‡

(kbrannon, hhsiao, mohan, narang, maha@almaden.ibm.com)

ABSTRACT

Managing a combined store consisting of database data and file data in a robust and consistent manner is a challenge for database systems and content management systems. In such a hybrid system, images, videos, engineering drawings, etc. are stored as files on a file server while meta-data referencing/indexing such files is created and stored in a relational database to take advantage of efficient search. In this paper we describe solutions for two potentially problematic aspects of such a data management system: backup/recovery and data consistency. We present algorithms for performing backup and recovery of the DBMS data in a coordinated fashion with the files on the file servers. Our algorithms for coordinated backup and recovery have been implemented in the IBM DB2/DataLinks product [1]. We also propose an efficient solution to the problem of maintaining consistency between the content of a file and the associated meta-data stored in the DBMS from a reader's point of view without holding long duration locks on meta-data tables. In the model, an object is directly accessed and edited in-place through normal file system APIs using a reference obtained via an SQL Query on the database. To relate file modifications to meta-data updates, the user issues an update through the DBMS, and commits both file and meta-data updates together.

Keywords

DB2, DataLinks, Database backup, Database recovery, Content Management.

1. INTRODUCTION

The motivation for introducing the DataLinks feature of DB2 stems from a number of observations. Much of the world's data lives in files and most data would continue to live there and the volumes will grow. Also, most of the world's applications work on files. Corba and the OLE DB framework have been proposed for providing uniform access to database and nondatabase data [2] [3]. An example of extending database capabilities to external repositories such as file systems can be found in [4] which focuses on extending database indexing, partitioning, replication and query processing to data stored external to the database. While much of the world's data resides in files, file systems do not provide the rich data management characteristics found in a DBMS. File systems do not provide sufficient metadata, nor do they provide a general-purpose query engine to manage access to and integrity of files. The DBMS has evolved to become a superb manager of data, i.e., it provides referential integrity, access control, and backup and recovery for its data. It allows arbitrarily complex data models and arbitrary queries.

A DBMS may not always be appropriate for the storage of very large objects. Many RDBMSs support the LOB data type for storing such data, however LOBs require the use of an SQL API to operate on them. This presents a problem for preexisting applications written to operate on data in files. The Oracle Internet File System (iFS) [5] addresses this concern by supporting the file system interface. SHORE [6] is another system that merges database and file system technologies. SHORE objects can be accessed via the normal UNIX file system interface. In both systems, however the file/object data is actually stored in the DBMS as LOBs. LOBs typically do not have support for hierarchical storage management as provided by products like Tivoli TSM [7]. Such support is crucial for cost-effective management of data with varying access patterns (frequent to infrequent usage). In addition, there is a substantial read/write performance penalty to access LOB data compared to accessing the data as normal files in a networked file system [5]. Also, legacy data maintained in files would have to be loaded into such systems. With DataLinks, legacy data can remain in files as is. DataLinks attempts to marry file systems and databases in such a way to extend the DBMS data management capabilities of data-valued based access control, referential integrity and backup/recovery to file systems, while retaining the powerful data management properties of file systems. DB2/DataLinks is an optional feature available for IBM's DB2 on open systems [1].

The importance of integrating files containing unstructured and semi-structured data with business applications is growing. It is possible to maintain pointers to external files as VARCHAR data, for example in a DBMS, to address this integration problem. While supporting applications written to use the file paradigm, this solution presents a daunting administration task: keeping the file data and DBMS data synchronized, especially over DB backup/recovery and file system backup/recovery. A system such as Oracle iFS or SHORE could be used to store the file data. Since this data is actually stored in the DBMS, there are no issues with coordinating DBMS backup/recovery and file system backup/ recovery. However, there can be a significant

ACM SIGMOD'2002, June 4-6, 2002, Madison, Wisconsin, USA.

performance degradation for reading the LOB data compared with reading directly from a file system. Also, the time to do a DB backup can become excessive when backups include a large amount of data in LOBs.

DataLinks extends the DBMS functionality of management of data to files stored in file systems while providing referential integrity, access control, and coordinated backup and recovery AS IF the files are stored in the DBMS [8]. A key feature of DataLinks is that the data in linked files can be accessed directly from the file system with no need to flow the data through the DBMS. This provides a performance advantage as well as the ability to support existing/new applications based on the file paradigm, but with improved DBMS-style management of the data in files. DataLinks can also be used for the management of a large number of files in a network of computers. The storage model is the usage of a DB as the metadata repository with URLs (Uniform Resource Locators) as pointers to the files residing in multiple file servers, which may belong to heterogeneous computer nodes and operating systems. This is shown below in Figure 1. The semantics of the field containing the URL is that the reference to the object stored in the external file system is consistent with the metadata (relating to that object), which is stored in the database. For example, the object cannot be deleted from the external store as long as there is a reference to it in the database.

Normal database administration requires that the database be backed up periodically, for example, once a week. This is so that in case of database corruption or device failure the database can be restored from one of the backup copies, typically the most recent one [9]. Depending on whether or not updates to the database are allowed to take place while the backup operation is in progress, the backup copy may or may not be self-consistent. The restoration of the database contents will typically require processing of the log records to bring the database to its most recent state. The latter is typically the desired state (e.g., when a device failure occurs). However, at times, it is desired that the database not be brought to its most recent state, because an application may have corrupted the database. When the database is brought to a state other than the most recent state, it is called point-in-time recovery. In this paper, we present algorithms for performing backup and recovery of the DBMS data in a coordinated fashion with the file servers, which have files, referenced from the database. From a backup and recovery standpoint, the files are considered to be part of the database but they are actually stored external to the DBMS. In anticipation of a DBMS backup operation, which would occur sometime in the near future, the backup of a referenced file is initiated when the file is associated (*linked*) with a record in the DBMS. The file backup is performed asynchronously to the linking process so that the linking transaction is not delayed

In a typical scenario, when a database backup occurs, all unfinished file backups are ensured to be completed before the database backup is declared successful. When a database is restored to a state, which includes references to files in the file system, the DBMS ensures that the referenced files in the file servers are also restored to their correct state. Note that the files could have been deleted or newer versions may have been created as a result of operations subsequent to the backup from which the database is restored. Our algorithms take into account these possibilities. It should be noted here that the referenced external files themselves are not kept in the same backup file where the database metadata is backed up. Rather, these external files may be backed up individually in a backup server, such as, Tivoli Storage Manager (TSM) [7].

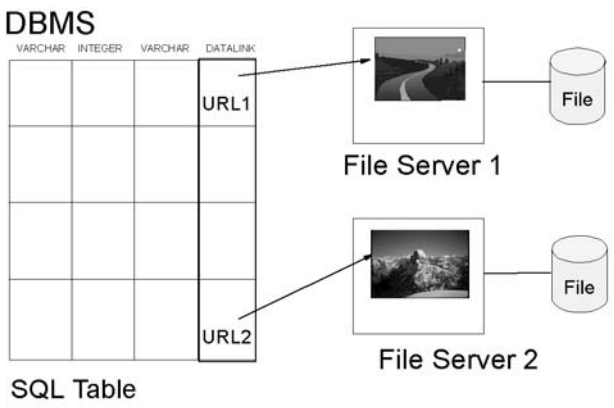

Figure 1. DataLinks Storage Model

Content Management is a technology that aims to manage both database data and file data in a consistent manner. An issue in such systems how to ensure that a reader sees a coherent view of the file data and the meta-data stored in the DBMS when updates are not tightly coupled. The DataLinks technology can be used to address this problem by generating a temporally unique version indicator associated with the last committed update transaction, and encoding this version indicator in the object reference associated with a given meta-data state. The current last modification timestamp of the file is available from the file system. A thin interceptor layer (DLFF) at the object's native store compares these with the latest information it has about the last committed version of the file and the corresponding last modification timestamp for that version. A mismatch of the versions or the timestamps indicates that the meta-data may not correspond to the current contents of the file and access to the file may be denied. Thus consistency is maintained.

The remainder of this paper is organized as follows. The normal operations of DataLinks which are relevant to backup and recovery are presented in Section 2. Section 3 presents the actions that take place as part of a database backup operation while restoration of a database is covered in Section 4. Section 5 discusses a solution to the problem of maintaining consistency between the file content and the meta-data stored in the DBMS.

2. DATALINKS CONFIGURATION

Figure 2. illustrates a typical DataLinks configuration. DataLinks functionality is supported by introducing **DATALINK** as an SQL data type in the DBMS, and by having a piece of cooperating software, called *DataLinks Manager, DLM* on a file server. The DATALINK SQL data type has been introduced as an ISO standard [10]. The DataLinks Manager (DLM) has two distinct software components, DLFM and DLFF, which are described below. The DATALINK column(s) in an SQL table contain the "pointer" to the file stored in a file server. The data structure for

the DATALINK data type includes the name of a file server and the name of a file, as in URLs. An application uses an SQL call to query the DBMS to search on the business data and perhaps, the features of unstructured (and/or semi-structured) data stored in the DBMS. The query specifies DATALINK column(s) in the select list and the DBMS returns the URL information to the client application. The client application can then access the file(s) using the normal file system protocols. The DBMS is not in the data path for delivery of file data, although file data can optionally flow through the DBMS in case an application just wants to maintain a single connection to the DB (rather than using connectivity with the file server). Note that we have introduced the new data type in the DBMS, but not any new API for database access or file access. We believe that this is significant for the following reasons.

- Current DBMS applications can be easily extended to incorporate new types of data such as image, video, text etc. using the SQL API to obtain the file server name(s) and file name(s) of interest.
- Applications to capture, edit and deliver the new types of data work on the file paradigm and would not require any change. Typically the file server name and file name obtained from the database can be fed into the application which provides the file access, e.g., web browser, or viewer (or helper) applications on the desktop for image, video, etc.
- Since the file data does not flow through the DBMS, there is negligible impact on the performance of file access compared to native file system access.

The components of the DataLinks Manager are described in the following sections

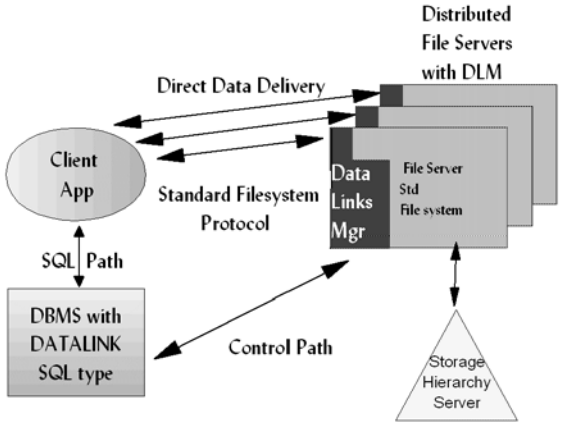

Figure 2. DataLinks System Configuration

.

2.1 DLFM (DataLinks File Manager)

As part of the data manipulative processing for storing the DATALINK value in the database, the DBMS communicates with the appropriate DLFM in the network, based on the server name provided in the DATALINK data value. This communication is the basis for the DBMS coordination for management of files on the corresponding file server. DLFM supports transactional properties to the database data and references to the files, and coordinated backup and recovery between database data and files. The DBMS interacts with DLFM using a two-phase commit protocol as described by Hsiao and Narang [11] to provide transaction atomicity. For this purpose, the DBMS acts as the coordinator and DLFM as the subordinate. DLFM needs some recoverable (transactional) storage for keeping persistent information about the operations (like LinkFile, UnlinkFile, etc.) that are initiated at DLFM by the DBMS. This functionality may come from a recoverable file system or a DBMS. To support coordinated backup and recovery with the DBMS, DLFM interfaces with a backup/archive server, such as TSM [7] to backup files and if required, restore files.

2.2 DLFF (DataLinks File System Filter)

DLFF is an interceptor module that filters certain file system calls, such as file open, rename and delete. For example, the filter logic in the file open call can be used to provide database-centric access control that is rule based [12]. The filter logic for operations like file rename and delete can be used to prevent such actions, if the file is referenced by a database in order to preserve referential integrity. That is to avoid the problem of dangling pointers from the database to the file system. DLFF does *not* come in the normal file read/write path of the underlying file system and does not impact performance. The functionality of DLFF is supported without requiring changes to existing file systems in which the files pointed to by the database are stored. This is very important for the case for supporting existing file data without any migration since such data may be terabytes in size. Also, the underlying file systems may be specialized such as a streaming media server. The added functions of DLFF are implemented in a layered approach using the facilities, like installable file systems, of modern file systems. If database centric access control option has been specified, then when an application interacting with the DBMS retrieves a DATALINK column value, the DBMS returns not only the server name and file name of the referenced file but also a *token or handle* which may be embedded in the file path name. The application passes all of this information during the file system open call. The token signifies that the DBMS has allowed access to the file temporarily and DLFF validates it before allowing the user to access the file. The token has a configurable expiration time. This is the mechanism by which DataLinks supports DBMS-managed access control for files. The token is discussed in more detail with regards to data consistency in Section 5.

2.3 LinkFile API

The LinkFile API *links* a file and places it under the control of the DBMS. The DBMS would issue this API to DLFM when a record is inserted into a table with a DATALINK (type) column or when a record update involving a DATALINK column occurs. The file server name is determined from the server name part of the DATALINK data structure. When DLFM executes the LinkFile API, it applies certain constraints to the file referred to in the DATALINK column. Such constraints can include the following and can be combined as options that are specified on the DATALINK column when the SQL table is defined.

- **DB-Owner Constraint:** The DBMS becomes the owner of the linked file. This constraint implies that the DBMS access permissions determine access to the file. Such an access would be based on a DBMS assigned token, as explained above. The user can choose to use the file system permissions rather than the DBMS permissions to support legacy applications if more appropriate. When the file system

permissions are to be used, the DBMS would not generate a token when it returns the DATALINK column value.

- **Read-only Constraint:** The linked file is marked *read-only*. Marking the file read-only guarantees the integrity of indexes that may be created on the contents of the file and stored in the DBMS for search purposes. DataLinks supports coordinated backup and recovery only if the linked file is marked read-only. Then, in order to update the file, either the file has to be in the unlinked state or a file versioning mechanism needs to be supported. If only the file reference in the DATALINK column is kept in the DBMS, i.e. no indexes on the contents of the file, and if coordinated backup and recovery is not required, then the user may not want to use the DATALINK column option which results in linked files being marked read-only. Due to space limitations, support for update-in-place of a file will not be discussed here.
- **Referential Integrity Constraint:** Metadata is maintained to indicate that the file is referenced by the DBMS. This constraint would prevent renaming or deletion of the file by a file system user once the file is referred to by the DBMS. It is applied to maintain referential integrity of the DBMS reference to the file.

In order to support the trivial option of no referential integrity, we allow an option called "No_control" which can be specified for the DATALINK column. For No_control, the URL syntax is validated, however, the DBMS does not issue the LinkFile API. Following is a table listing the possible combinations of the constraints that can be specified as options for a DATALINK column during CREATE or ALTER TABLE.

Table 1. DATALINK Column Constraints

Option	DB-owner Constraint	Read-only Constraint	Referential Integrity Constraint
DBMS control	Yes	Yes	Yes
File_control	No	No	Yes
File_control read_only	No	Yes	Yes
No_control	No	No	No

As part of the LinkFile API, the DBMS passes the constraints that DLFM should apply to the file that is being linked. For DBMS_control, DLFM may also save the original owner and access permissions (or access lists) of the file, so that it is possible to restore them when the file is unlinked from the DBMS. For File_control_read_only, DLFM may save the file's write permission information since it may need to be restored during unlink. Whether to save the original owner and permissions is determined by the Unlink option that can be specified as Unlink(Restore) or Unlink(Delete) for the DATALINK column. Unlink(Restore) implies that the original owner/permissions should be restored when the file is unlinked from the DBMS; Unlink(Delete) implies that the file should be deleted when it is unlinked from the DBMS. For the latter, it is not necessary to save the original owner/permissions at the time of link.

The linking operation is done in a transactional manner. This means that if the transaction performing the link were to fail, then the link operation is undone during transaction rollback. On the other hand, if the transaction were to commit, then the effects of the LinkFile API will persist, even if there are failures. DLFM persistently records the fact that the file is now pointed to by the database. For brevity, we do not describe here the details of change of ownership and permission of a linked file, and their persistence.

As mentioned above, coordinated backup and recovery can be supported for those files that belong to a DATALINK column with options DBMS_control or File_control_read_only, since the file is marked read-only during the LinkFile operation. We provide an option of specifying Recovery(Yes) or Recovery(No) to declare whether the files pointed to by an DATALINK column are important enough to require backup since backing up of files costs storage, CPU and, perhaps, network resources.

An example of a CREATE TABLE for T1 with 2 columns, C1 an integer value and C2 a DATALINK value, is given below. For the DATALINK column, we have selected the options DBMS_control, Recovery(Yes) and Unlink(Delete).

```
CREATE Table T1 (C1 integer,
   C2 DATALINK(DBMS_control,
   Recovery(Yes), Unlink(Delete)));
```

An example of an SQL INSERT into table T1, defined above is as follows:

```
hvc1 = 5;
hvurl = "http://server1/foo";
EXEC SQL INSERT INTO T1 (C1, C2)
VALUES (:hvc1, DLVALUE(:hvurl));
```

DataLinks Manager software should be installed in server1. The DBMS would communicate with DLFM on server1 and request it to link file foo transactionally. The DBMS would accept the data value in the DATALINK column of the row to be inserted only if the file foo exists and the transaction commits. In the above example, when the transaction commits, DLFM would make the DBMS the owner of the file and the file would be marked read-only. DLFM would not save the original owner and permissions of the file since Unlink(Delete) is specified. DLFM would make a persistent record that the file is linked to a DBMS. In this persistent record, DLFM saves a DBMS provided time-based recovery identifier called *RecoveryID_at_link*. This identifier is also stored in the DATALINK column by the DBMS. The recovery identifier also finds use in the consistency detection scheme discussed later in Section 5.

Values stored in the DATALINK column in the DBMS for the above SQL INSERT are as follows:

URL scheme	Server name	File name	RecoveryID at link
http	Server1	foo	t1

Values stored in server1's DLFM for the file record for file foo are as follows:

File name	RecoveryID at link	Linked State	RecoveryID at unlink	Backup Number
foo	t1	L	null	0

RecoveryID_at_unlink and Backup Number are discussed later in the context of the UnlinkFile API.

Since Recovery(Yes) is specified for the DATALINK column in our example, DLFM would make a persistent entry in a *copy-table* so that a backup copy of the file would be made soon after this transaction commits. Such a copy may be made in a batch mode by a copy-daemon in DLFM that processes many files. It is not efficient to make a copy of the file in the DBMS transaction scope since the size of the file may be large (several hundred megabytes to gigabytes). However, the persistent entry in the copy-table is made within the DBMS transaction's scope. It is possible to make the copy asynchronously because the file is marked *read-only* at link time. We have to handle the case where the file is unlinked before copying is completed. It is crucial that the backup operations of the copy-daemon operate efficiently and finish the file backups in a timely manner. The database backup will not succeed unless the file backups are completed. If the copy-daemon gets far behind in making the backup copies of the files, then there is an increased window during which backups could fail. These concerns can be addressed by parallelizing the functions of the copy-daemon. Since metadata is available in DLFM about the files, possibly including on which disk a file is stored, multiple copy-daemons could operate efficiently to ensure that file backups are completed as quickly as possible.

In case DLFM fails prior to completion of making a copy of the file, DLFM would start the copy-daemon during its restart recovery so that the copy would be made ultimately. The record entry (of relevance to this paper) in DLFM's copy-table would be as follows:

File Name	RecoveryID at link
foo	t1

2.4 UnlinkFile API

The UnlinkFile API lets DLFM know that the database no longer references the named file. This API will be issued when a record containing a DATALINK column is deleted or when a record update involving a DATALINK column occurs. If the options specified on the DATALINK column are DBMS_control and Unlink(Restore), then DLFM will restore the ownership of the file to the original user and also restore the access permissions that were recorded earlier when the LinkFile API was processed. If Unlink(Delete) option were specified, then DLFM would delete the file from the file system. Furthermore, if Recovery(No) were specified, then DLFM would also delete the persistent information it had stored earlier about the file. However, if Recovery(Yes) is specified, then in order to support coordinated, point-in-time recovery with the DBMS, DLFM continues to retain that information persistently and notes that the file is in the unlinked state. Note that the backup server still has a copy of the file in case the file needs to be restored as part of a coordinated restore with the DBMS. If the DATALINK column options are File_control_read_only and Unlink(Restore), then the write permissions to the file are restored. The Unlink(Delete), Recovery(Yes) and Recovery(No) discussions are the same as those for DBMS_control as described above.

The UnlinkFile API also includes a parameter called RecoveryID. It is a time-based recovery identifier, assigned by the DBMS, known as RecoveryID_at_unlink. DLFM records this ID in the record for the file when it is updated to note that the file is in the unlinked state. Note that both RecoveryID_at_link and RecoveryID_at_unlink are maintained. These can be used subsequently to support coordinated recovery with the DBMS.

A file foo can be linked at time t1, then unlinked at time t2 and deleted, then subsequently created again at time t3, and unlinked at time t4 and get deleted. Assuming that there is no garbage collection of unlinked files in between, DLFM would have the following records with their Recovery_IDs.

File Name	RecoveryID at link	Linked State	RecoveryID at unlink	Backup Number
foo	t1	U	t2	3
foo	t3	U	t4	5

Even though file foo does not exist in the file system because the user chose Unlink(Delete) option, the backup server would have 2 distinct files for foo: foo as it existed at t1 and foo as it existed at t3. Having these versions of file foo in the backup server allows coordinated recovery to be possible between DBMS and DLFM. It is possible for the DBMS to reconcile with DLFM based on the RecoveryID_at_link and/or RecoveryID_at_unlink as described in Section 4.

It is clear that garbage collection of the unlinked files is necessary at some point in time. At the time of unlink, a backup number is assigned to the file record in the Backup Number field.

3. BACKUP OPERATION

Traditionally, database backups have been supported with two flavors [9]. If a backup is made while there are no concurrent updates, then it is referred to as an offline backup. If the backup is made while concurrent updates are allowed, then it is referred to as an online backup. DataLinks supports both types of backups. For coordinated backup between database metadata and external files, a straightforward approach could be to copy all the files that are referenced by the DATALINK fields at the time the database is backed up. However, the performance of this approach may be unacceptable for the following reasons:

- A database backup typically copies a database page at a time rather than a record at a time [9]. Reading each record at backup time to determine the DATALINK field values would cause significant performance degradation to the backup procedure. Uncommitted updates may also cause additional complications.
- Many files may be large in size. Copying them may take more time than the time it takes to back up the database.

Hence, we need an alternate way of identifying and backing up the newly linked files. One of the objectives of our design is that performance of the database backup should not suffer in the presence of the DATALINK columns referencing files. This objective is realized by initiating the backup of a file when the transaction referencing that file in a LinkFile operation commits. This is described in section 2.3, “LinkFile API”. The key point is that the backup of the file is performed asynchronous to the transaction, and it is guaranteed that the backup copy would be made.

Extensions to these algorithms to ensure that the database backup operations succeed even if one or more DLM file servers are unavailable are presented in [13].

3.1 Coordinated Backup

Initiating the asynchronous copying of files poses some challenges with respect to the backup of the database and some mainline functions. At the time the DBMS backs up the database, it needs to ensure that any asynchronous copy operations of the files referenced by this database, which were started since the immediately preceding backup, are complete. It is sufficient to check completion of only the copy operations initiated since the immediately preceding DBMS backup because of the following reasons:

- A file is marked read-only when it is linked to the database and cannot be updated as long as it is linked from the database. To update a file in place, one has to unlink the file from the DBMS, update it and then link it again. From DLFM's standpoint, the RecoveryID_at_link makes these two versions of the file temporally unique.
- Completion of asynchronous copy operation is checked at every DBMS backup, so by induction it is sufficient to check completion of the copy operations, which were initiated since the immediately preceding backup.

To accomplish the above, the DBMS issues the BackupVerify and BackupEnd APIs to the DLFM. Next we describe the parameters associated with these APIs and then their use.

3.1.1 BackupVerify API

The purpose of the BackupVerify API is to verify whether the asynchronous copying of the files, which gets initiated as a result of LinkFile, is complete. The information passed via this API of interest here is as follows:

sincetime: Timestamp that represents the lower bound for the files whose LinkFile operations were performed at or after this timestamp and resulted in asynchronous copying of the files to be made to the backup server.

curtime:Timestamp that represents the upper bound for the files whose LinkFile operations were performed at or before this timestamp and resulted in asynchronous copying of the files to be made to the backup server.

The BackupVerify API is issued by the DBMS to DLFM at the time the DBMS starts to backup the database. In order to verify that all the asynchronous copy operations, which are set in motion as a result of LinkFile calls, which were initiated since the immediately preceding backup are complete, the DLFM uses the backupendtime of the previous backup registered in the backup table as the SinceTime and the start-timestamp of the current backup or the end-of-log LSN at the time of the start of the current backup as CurTime. The set of files that satisfy the following condition would be checked to see whether they have been backed up or not:

CurTime >= RecoveryID_at_Link >= SinceTime

If all the files in this set are already backed up, then the BackupVerify call returns *success.* If not, it returns *copying_in_progress*. In the latter case, the DBMS can continue to do its processing to copy the database but when the DBMS completes copying the database, it must issue the BackupVerify API to check whether the files in the above set and any newly linked files have been copied. If it is an offline backup of the database, then no new LinkFile calls would be issued while the backup is in progress and so if BackupVerify returns copying_in_progress the first time, most likely DLFM would return success on the second call. If success is not returned, then the DBMS decides on a time interval frequency at which it would continue to check whether the files have been copied or not. In the extreme case, if DLFM were to fail or some large interval of time were to expire with files in the above set still not copied, then the DBMS would fail the backup operation.

For an online database backup, backup operations could be initiated as a result of new LinkFile operations while the database backup operation was in progress. The DBMS must ensure that those linked files are copied as well before declaring that the coordinated backup is complete. The DBMS must issue BackupVerify on completing the backup on the database side. This is in contrast to the offline backup where BackupVerify needs to be issued a second time during a given backup only if the first BackupVerify (at the start of the backup) returned copying_in_progress.

The SinceTime and CurTime values for this second call for online backup are set as follows:

- SinceTime: For the first BackupVerify, it is set to end-timestamp or end-LSN of the previous backup. For the second BackupVerify, that value is moved forward to the timestamp/LSN taken just *before* the first BackupVerify was issued ONLY if that first BackupVerify returned success.
- CurTime: Timestamp or LSN taken at the end of database backup.

3.1.2 BackupEnd API

The purpose of the BackupEnd API is to declare that the coordinated backup has completed successfully. This API is transactional and is issued only after verifying that asynchronous copy operations as requested by BackupVerify APIs are complete on all the DLFMs. The information passed via this API of interest here is as follows:

txnId: The transaction within which this API is invoked.

backupendtime: Timestamp that represents the end of the coordinated backup. All subsequent LinkFile operations should have a timestamp that is greater than backupendtime.

BackupEnd tells DLFM that the coordinated backup succeeded. DLFM records the backupendtime and other information related to this coordinated backup within the Backup table. BackupEnd is issued by the DBMS to the various DLFMs in a transactional fashion, using the two-phase commit protocol. After the coordinated backup is successful, garbage collection is initiated.

3.2 Efficient Garbage Collection After Backup

A garbage collection procedure on the DBMS side monitors the number of database backups that are kept valid as well as dictates when to garbage collect files that were unlinked on the various DLFMs based on the database configuration parameter, NUM_DB_BACKUPS. When database backup files are to be garbage collected, the garbage collection procedure will mark the utility history tracking, UHT, file entry for the database backup as expired. The procedure will also notify all DLFMs to garbage collect the associated files that were unlinked before this expired backup. The backup number field associated with the unlinked file entry is used to identify such files. The unlinked files would be deleted from the backup server and the corresponding DLFM metadata would be discarded. Determining which unlinked files should be deleted is performed efficiently, as described in [13].

3.3 Information Tracked for DBMS Backups

Recall the topology illustrated in Figure 1., where a DBMS can reference files on multiple file servers. Additional information that is tracked with a database backup includes the list of DLFMs, and the list of DATALINK columns with data referencing this DLFM. The reason this information needs to be tracked is that during the restore operation we need to validate that those DLFMs still have the meta information about the DATALINK columns of the database. The case in point is that a database may have been deleted but it is necessary to restore it from an earlier version. When a database is deleted, DLFMs do not delete DATALINK meta information immediately but keep it for some period of time that is user configurable. When a database is restored to an earlier version and a DLFM that is in the list of DLFMs that are tracked with that backup no longer has the DATALINK column defined, then the restore procedure has to resort to the reconciliation procedure. The reconcile utility is described later.

Now, the question arises as to how does the DBMS determine which DLFMs are involved since, as mentioned earlier, the database backup does not read any database record as it makes the copy of the database. This is discovered by contacting all known DLFMs as per the configuration file and then recording the ones that respond positively that they have the DATALINK metadata defined. The DATALINK metadata is defined in a DLFM when the first LinkFile (ever) is issued to that DLFM.

3.4 Impact on LinkFile and UnlinkFile Operations

As mentioned in Section 3.1, DLFM needs to verify the completion of copying of only those files that were linked since the last backup. This requires validation of RecoveryID_at_link (a timestamp and/or LSN) so that it is greater than the backupendtime. Otherwise, the verification of whether the asynchronous copy of the file is complete or not, may not be performed correctly. This validation is performed by the DLFM since it tracks the BackupEndTime of the last backup. If the above condition is not met, then DLFM returns an error condition. The DBMS can then reassign a new RecoveryID and reissue the LinkFile operation.

In Section 2.3, “LinkFile API”, we mentioned the case where a file is unlinked before it is copied to the backup server. Below, we describe how we handle this. The UnlinkFile operation has to pay attention to whether asynchronous copying of the referenced file, which would have been initiated when the corresponding LinkFile call was issued, is complete or not. In the rare case where a database update of the DATALINK field causes the file to be unlinked and the copy of the file is not yet made, then the unlink operation is serialized by the DLFM with the backing up of the file. This is necessary since the file could be deleted from the file system as a result of the unlink operation committing before the copy is made. Once the deletion is performed, copying cannot be done and a subsequent database restore operation might require that version of the file be available.

4. RESTORE OPERATION

A backup copy of the database can be used to recover the database to one of many possible consistent states:

- The consistent state could be the state when an offline backup was taken. Such a state is referred to as offline backup state, OBS.
- The consistent state could be the state at some point in the log when no update was being allowed to the database. Such a state is referred to as a quiesce point state, QPS.
- The consistent state could be to some arbitrary point-in-time state, PTS.
- The consistent state could be the state at the time of crash (i.e., the database is current) known as the current time state, CTS.

If the database does not have a DATALINK field, i.e., there are no references to external objects, e.g., files, then the restore of a database from an offline backup (OBS recovery) restores the database to a consistent state. For QPS, the restore processing applies log records to the restored version of the database from when the appropriate online copy was started to the point when the update transactions were quiesced. The database is brought to a consistent state up to when the quiesce point was established. However, when DATALINK fields are defined, references to files would be involved. Additional processing is required to synchronize references to files with respect to the restored version of the database. Later, we describe the details of synchronizing files referenced by the database for the cases OBS and QPS.

For recovery up to the current state or CTS, the DBMS applies the log to the last database backup that was restored up to the point when the database was closed after a problem like media failure. For point-in-time recovery or PTS, the user dictates that the DBMS may apply the log only up to some arbitrary point in the log. In the case of CTS, applying the log up to the point of closing the database after the media failure implies that the database was not in use after that point, hence the database would be brought to a consistent state. The references to the files would already be synchronized in this case (CTS). However, in the case of PTS, when the log is applied up to some arbitrary log record that is at a point earlier in the log than at the end of the log, the database may not be restored to a consistent state with respect to the DATALINK data that is referenced by the database. The administrator may have to run the reconciliation procedure described below, to bring the database to a consistent point.

Extensions to these algorithms to ensure that the database recovery operation succeeds even if one or more DLM file servers are unavailable are presented in [13].

4.1 Restore of a Database for OBS

Below, we describe the additional processing which is required to synchronize references to the files with respect to the restored version of the database. Efficient synchronization is possible because the DBMS provides DLFM a RecoveryID, at the time of establishing the link of the reference, RecoveryID_at_Link, and another Recovery_ID at the time of unlinking the reference, RecoveryID_at_Unlink. As mentioned before, these Recovery_IDs could be in terms of LSNs or timestamps and DLFM tracks them in its persistent storage. Recall that an unlinked file is tracked by DLFM for some number of backups until it is purged as a result of garbage collection. Therefore, when the database is restored to a consistent state, before declaring that the database has been restored, the DBMS would do the following

additional processing. The DBMS would provide a Restore_Recovery_ID, which is the timestamp or the LSN of when the backup used for restore happened, to DLFMs which were involved in the backup. A DLFM would do the following processing for the files that it tracks:

- Unlink all files whose RecoveryID_at_link is greater than or equal to the restore_recovery_ID. This would unlink all files that were linked after the backup.
- Link all files whose RecoveryID_at_link is less than Restore_recovery_ID AND RecoveryID_at_unlink is greater than or equal to the restore_recovery_ID. This would relink all files that were in the linked state prior to the backup but were unlinked after the backup.

The above reconciliation between DBMS and its external references is referred to as *reconciliation with respect to RecoveryID*. This is in contrast to the more elaborate reconciliation process that is performed by accessing each database record, determining the file names, and then reconciling with the appropriate DLFMs. Such a detailed reconciliation procedure may be needed when point-in-time recovery (PTS, QPS) is performed for the database, and is described later.

It should be noted that relinking a file after it was unlinked implies restoring the version of the file when it was linked. If required, DLFM would interact with the backup server, e.g., TSM to retrieve that version of the file. If the correct version of the file exists in the file system, then that version can be used for relinking. There are possibilities of various errors while restoring the file, such as, duplicate file name in the file system. If there are any such errors, then the database table is put in the datalink-reconcile pending state since detailed analysis of which row/column is linked to the file needs to be reported. This reconciliation process is described later. Though reconciliation with respect to RecoveryID is more efficient, the following point should be noted.

If the database backup image used for restore is older than the limit of "NUM_DB_BACKUPS" backups, then the files that were unlinked "NUM_DB_BACKUPS" backups ago may have been garbage collected. Such a check should occur before Reconciliation with respect to RecoveryID procedure is invoked. If the above conditions are not met, then the database tables should be put in the datalink-reconcile pending state and the more detailed reconciliation process to be described next should be performed.

4.2 Restore of a Database for PTS and QPS

After restoring a database from a backup file that was made while concurrent updates were going on, log records have to be applied to the restored copy. This process is called *roll forward* [14]. The following possibilities exist with respect to this application of the log:

- Rollforward is performed to the end of the logs. Processing to the end of the logs ensures transactional consistency between the database references and the files in the file servers. So there is no need for any additional processing when DATALINK columns are defined in the database. This is the CTS case mentioned earlier.
- Rollforward is performed to an arbitrary point in the log, PTS or to some point in the log when no update was being allowed to the database, QPS. The database tables with DATALINK columns are put in the datalink-reconcile pending state at the end of database rollforward processing. Such tables are identified through the DBMS catalogs, which are consistent at the end of rollforward processing. In this case, the Reconcile utility, which we describe next, should be run. For QPS, reconcile *with respect to recoveryID* can be performed when the log sequence numbers, LSNs are unambiguous [13].

The Reconcile utility performs the processing necessary to ensure that files which are referenced by the database table data are in the linked state and files which are not referenced by the database table data are in the unlinked state. Reconciliation is performed in the transactional context since files may have to be linked and unlinked in this process. The DBMS Reconcile utility examines the DATALINK columns of the restored database's tables records and collects the list of referenced file names, file server names, RecoveryID_at_link recovery ids and the (DBMS) record-IDs. The record-IDs are needed to report errors as described below. Then, this list of file names, RecoveryID_at_link recovery ids and their respective record-IDs is passed to appropriate DLFMs. DLFMs have to try to restore the correct version of the corresponding files, determined by their RecoveryID_at_link and put them in the linked state. Any other files (i.e., files not in the list passed by the DBMS) that are linked from that database, as per DLFM metadata, should be unlinked. If a DLFM cannot find in its backup server some of the files needed by the DBMS or the files cannot be restored for any variety of reasons, it returns the corresponding record-IDs for exception processing to the DBMS. The DBMS can set such DATALINK column value to NULL and report them as exceptions. By removing such entries from the database tables, the DBMS can make the table available for use. After determining the fate of the files reported in the exception list, and taking appropriate actions on the file server(s), the DATALINK column of the exception rows may be updated to reference the appropriate files. Applications are allowed read access to the database tables for which the reconciliation processing is being performed and also on tables that are in datalink-reconcile pending state.

5. THE LOOSELY COUPLED TRANSACTION MODEL

Content Management (CM) Systems, especially those designed for managing distributed content typically store meta-data that describes an object or related information in a store such as in a RDBMS, that is separate from the file containing the content of the object. One challenge in such a system is that of maintaining consistency between the content of the file and the associated meta-data from a reader’s point of view. If file and meta-data updates are tightly coupled such that updates to both happen within a single unified transaction, then the transaction coordinator typically ensures a consistent view by locking out readers of meta-data as well as file data until the transaction is committed. In this way intermediate or uncommitted updates to either file data or meta-data are not visible. Given that file content edits can be relatively long running, such an approach may not be desirable. Thus, a loosely coupled transaction model for file and meta-data updates is needed, where file edit is independent of the meta-data update, to avoid holding long duration locks on meta-data tables.

An improved approach, based on the loosely coupled transaction idea, is implemented in IBM Content Manager [15]. It de-couples the action of updating a file from the database/CM transaction that makes the updated file visible (to CM users) and changes meta-data associated with the file. With this approach, a new version of a file is created when the file is updated. During file update and before an updated file is "committed" to CM, existing meta-data is not locked and continues to reference the last committed version of the file. When a user decides to commit an updated file, the associated meta-data is then updated and committed within the same database/CM transaction. Since file update and associated meta-data update are done within the same transaction, users will always have a consistent view of meta-data and file data. In addition, the last committed version of a file is maintained, thus users can continue to reference a consistent version of the file even when it is being updated. This approach, however, has its drawbacks. Firstly, multiple versions of a file are maintained which requires much more disk storage space. Secondly, either the file reference needs to be changed (or a version number increased) every time a new version is created, which is undesirable for some applications, e.g. Web servers, or new file version name needs to be changed to the existing name at the commit time, which requires additional file system access. It is often convenient and natural to allow the content of files to be accessed and edited in-place without creating multiple versions by directly accessing the file system natively. This, however, creates the possibility for leaving file content and meta-data in an inconsistent state, which is unacceptable in many applications. Thus, a novel scheme to guarantee a consistent view of file-data and meta-data to a reader without the need of holding long duration locks on meta-data stored in the database and without the need of maintaining extra version of a file is highly desirable.

The following sections address the above problem in the context of meta-data maintained in a database table associated with an external file reference to content that is stored in a file system or an object store that is external to the database. The external file reference is maintained within the database using the DATALINK SQL data type. In order to be able to unbundle file and meta-data updates without loss of basic transactional semantics on the linkage and consistency between the two types of data, the update model lets a user directly edit the file in-place independently of the meta-data without the knowledge of the meta-data store, and then effect the corresponding meta-data updates on the meta-data store explicitly relating them to the file updates and committing both file and meta-data updates together. Thus the potentially long-running content edit process is decoupled from the database transaction that updates the associated meta-data, or more precisely, the two updates are loosely coupled. The file must be accessed or edited using an object reference supplied by the meta-data store, rather than its native name (which may or may not be the same) in order to ensure a consistent view to the CM/DB application.

5.1 SQL Mediated Object Manipulation

The notion of SQL mediated object manipulation can be applied to an object that is located in an external store, but referenced in a database table. The idea is that the DBMS acts as a mediator in terms of providing update access and relating file updates to corresponding meta-data updates without actually implementing the content updates itself. Once an update reference is granted by the DBMS, actual file updates happen in-place directly on the file system using native file operations without going through the DBMS. Updates are committed to the database via the DBMS once the update is done. The permission to update the file may be mediated by the DBMS or by file system permissions. With this approach, actual file edits happen outside of the database transaction. This is significant because it means that database locks are not held during potentially long running content edits.

To associate an object with its meta-data in the database, a user issues an SQL INSERT statement involving a DATALINK entry as described in Section 2.3. To access (read or write) the object, a user obtains a handle to reference the object (Section 2.2) from the results of an SQL SELECT on the table based on the identification criteria. The object is then accessed directly from the external store, by supplying the handle as the name of file to the native file system API. Once content updates are completed, the user issues an SQL UPDATE on the database, optionally passing in the handle, then updates the corresponding meta-data fields in the table, and finally commits the update transaction. Note that this update transaction need not be part of the transaction in which the SQL SELECT was issued to obtain the handle. Coordinated backup and recovery is supported for the external data "linked" to database meta-data, so that a restore of the database to an earlier point in time also restores the content files to the corresponding state in conjunction with the associated meta-data (Sections 3 & 4). The object can be disassociated from the database by issuing an SQL DELETE operation, which "unlinks" the file from the database.

An interceptor/filter module, i.e. DLFF services direct content access using the handles described above and is illustrated in Figure 3. As described in Section 2.2, it transparently intercepts accesses to the external object store and validates the handle, which contains an embedded access token used for referencing the object. DLFF communicates with a file/object management mediator daemon, i.e. DLFM that tracks and manages objects on the file system, which are linked to the database. DLFM records certain attributes of these objects at the time of their association with the DB, and at the time of committed updates. To support file-data and meta-data consistency, DLFF returns an error for file accesses where the file contents are no longer consistent with the meta-data associated with the handle being used to reference the object. For a UNIX operating system, this error could be ESTALE.

5.2 The Consistency Detection Scheme

The following describes the design goals considered in arriving at the proposed consistency detection scheme. The central objective is to make sure that a reader application always sees consistent data in all possible scenarios including the possibility of a database restore to an earlier point in time. In order to provide the convenience of in-place update using native file system APIs, the consistency check is performed via the interceptor layer. A crucial concern is to avoid holding database locks during long running content-edits. The meta-data is thus visible while the file is being modified. As explained earlier, this is why a tight transaction model is not appropriate.

The solution has to be suitable for a distributed model of file and meta-data storage and support fast content access times in such an environment. This means that the scheme should avoid having to

check back with the DBMS or the meta-data store to ensure consistency during file access. At the same time it should avoid introducing additional overheads in the update-path or the normal transaction path as that could lead to performance degradation. Certain time based decision schemes avoid communication between multiple entities in a distributed environment, but require that the clocks on these systems be kept in synchronization through some means, for example, through a combination of administrative settings and distributed time protocols. One of the initial schemes that was explored had such an element as it involved comparing the timestamp of the last committed update embedded in the handle by the database with the modification timestamp of the file which is set by the file system. The idea here, however, is to avoid imposing such clock synchronization requirements on the database and file system servers for supporting consistency detection. This keeps the administrative overhead and complexity low.

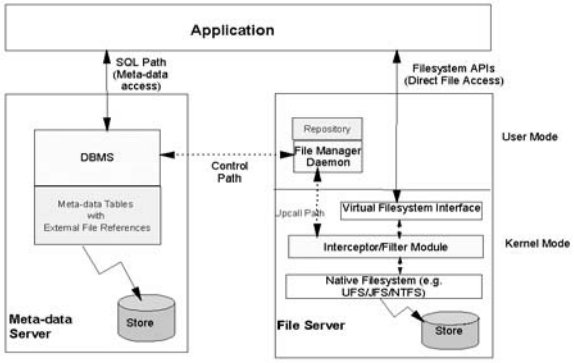

Figure 3. DataLinks System Architecture Detailing the Interceptor/Filter Module

Another requirement is the suitability of the scheme for a distributed file system (or object store) implementation with authoritative caching at file system clients. The problem that arises in the presence of "authoritative caching" where the file system client takes decisions on its own based on prior information from the file server cached at the client, is that an interceptor layer on the file server may not always be able to intercept file accesses and validate consistency. This gives rise to a need for implementing an interceptor layer at the file system clients. In such a situation it is desirable to minimize the need for communication between the client side interceptor and the file management daemon on the server. This must be achieved without requiring the server to maintain any state about its clients. The design for a distributed file system with authoritative caching is described fully in [17]. It involves extending the consistency detection logic with a reliable client initiated caching scheme for the relevant attributes stored by the file management daemon.

An underlying design principle is simplicity and reuse of existing internal mechanisms of the two storage engines, i.e. the DBMS and the file system, avoiding any duplication or introduction of complex logic. The scheme should require only the absolute minimal additional state to be maintained outside of what is already available.

5.2.1 The Source of Inconsistencies

As described, the handle required to access the object from the object store contains an access control token generated by the DBMS. From a reader's perspective, inconsistencies between the meta-data accessed from the database and the corresponding contents referenced by the handle could occur in the following situations:

- File updates happen after the token is generated.
- Uncommitted updates were pending at the time the token was generated. In this case, the meta-data in the table at the time the token was generated corresponds to the last committed file data, and hence does not reflect any uncommitted file updates that happened before the token was generated.
- The database gets restored to a prior time before the token is used to access the file.

An update can happen via a re-linking of the same file with different contents (after an SQL DELETE or "unlinking" of the file and a change in contents via a rename or a write to the file, followed by an SQL INSERT or "linking "of the file. It can also happen through an SQL UPDATE following file edits in case of an update-in-place. There are two cases to be considered:

- Updates committed (or reverted to an earlier point in time in case of a database restore) before the token is used (results in a change in last committed version of the object)
- Updates not yet committed before the token is used (update-in-progress state)

This leads us to the following solution.

5.2.2 Basic Solution for Local File systems

The proposed solution is based on two constituents: a version indicator or generation number associated with the file, which is temporally unique for every committed update to the file, and use of the last modification timestamp attribute of the file maintained natively by the file system or external store. Since updates are committed with the knowledge of the DBMS in communication with DLFM, the time based *RecoveryID* (discussed in Section 2.3) for the update, can be used as the version indicator. Also, whenever updates are completed, the last modification timestamp of the file at the time of update completion is noted in the local repository as the last modified time of the latest committed version of the object. It is more efficient if the last modification timestamp is recorded when an SQL UPDATE is issued for the file, rather than at the time when the transaction finally commits. This is possible because in this model, no further updates to the file are allowed until the transaction commits.

The approach works as follows. The DBMS encodes the latest version indicator (or generation number) as part of the embedded token in the handle it provides for referencing the file. Then when the user supplies the above handle directly to the file system to open the file, the filter layer which intercepts these file system operations would make a call to the file management daemon to perform the following checks to ensure consistency at the point of open:

I. Lookup the latest committed version number V(commit) of the file, and the corresponding modification timestamp Tm(commit) recorded in the local repository.

II. Extract the version number encoded in the embedded token in the handle, V(token)

III. Get the current value of the last modification timestamp of the file Tm(access) from the file system.

IV. Return an error (indicating stale data), if either of the following conditions are met:

- If V(token) ≠ V(commit), the token is inconsistent with the current file contents. This covers the case where the reader had obtained the access token/handle any time prior to the start of the update which was last committed and also the situation where the reader had obtained the token and the metadata, and then in the meantime the database and the file got rolled back to an earlier state.
- If Tm(access) ≠ Tm(commit), there are uncommitted updates to the file's contents. This covers the case where an updater has modified the file and then closed it after releasing any file locks, but hasn't updated the corresponding meta-data. Then, optionally, if the supplied handle has write-access, check the repository state to rule out the case where the updates have been made using the same access token.

Readers and updaters must use application level file locking to cover concurrent file read and updates; in particular, to ensure that there is no open file descriptor when an updater starts updating the file.

When the consistency detection logic is performed at a distributed file system client rather than at the file server [17], then there is an added step of loading/refreshing V(commit) and Tm(commit) values from the server. If these values are cached at the client, then this is required only if the file has not already been accessed at the client, or if the consistency check using the cached values indicates stale data.

5.2.3 Encoding the Version Number in the Token

The access token contains a message authentication code, MAC that is a one-way hash value with the addition of a secret key [18]. The encryption scheme exists because the access token is also used to provide database-mediated data value based access control for access to the referenced file system objects. Note that the hash is not a mandatory for the consistency detection scheme as such and the version indicator could be embedded into the object reference in any alternative manner that seems appropriate, if access control is not required.

The MAC, h is computed as follows:

h = hash(K, Tx, file, server, token len, flags),

where K is a secret symmetric key, Tx is the expiration time and the flags contain information about the token including which type of access is granted. Tx, token length and flags are also passed as part of the token. For cases where consistency is to be maintained between the DBMS meta-data and the object, the token can be set up to contain the version number and a different MAC, H which would be computed as:

H = hash(K, Tx, file, server, token len, flags, Vc),

where Vc is the latest committed version at which the token is created and flags can contain information indicating that consistency is requested.

For token validation, the file system filter passes the token to the file manager daemon. If the flags indicate that consistency is requested, the daemon validates the token by calculating H where Vc is obtained from the last committed version number stored in the daemon's repository tables. The daemon checks that the token has not expired and compares H with the MAC contained within the access token.

5.3 Scenarios Analyzed

A set of key scenarios has been analyzed in detail in [17] showing how the proposed solution prevents potential inconsistencies from arising in these scenarios. It is assumed that it is sufficient to guarantee the consistency at the point of the file/object open. The user is expected to use some kind of file/application-level locking as in normal file system applications to ensure serialization between concurrent readers and updaters. Specifically, an updater must not start its updates if there is any reader, which has opened the file, otherwise the reader could see inconsistent data.

6. SUMMARY AND CONCLUSIONS

For an environment where a linkage is maintained, with referential integrity, between data in a DBMS and files in a file server which is external to the DBMS, we present algorithms for performing backup and restore of the DBMS data in a coordinated fashion with the file system. We have introduced a new SQL data type in the DBMS, DATALINK, to enable a merger of DBMS and file system technologies. The object referenced in a DATALINK column can be perceived as being stored in the database for access control, referential integrity, and backup and recovery, but actually it is stored outside in a file system. DataLinks does not require changes to existing file systems that manage the files pointed to by the database. For backup, our approach makes use of a backup/archive server like TSM for storing the file backups. In order to avoid delays in the database backup operation, backup of a referenced file is initiated when the file is associated (*linked*) with a record in the DBMS. The file backup is performed asynchronously to the linking process so that the linking transaction is not delayed. No database locks are held while the backup of the referenced file is in progress. During a database backup operation, we do not require the processing of the copied records to identify what files need to be backed up on the file system side. This results in improved performance. It even allows for efficient copying of the database pages (e.g., by avoiding reading in the copied pages into the buffer pool of the DBMS [9]). When database backup occurs, all unfinished file backups are completed before the database backup is declared successful. When a database is restored to a state that includes references to files in a file system, the DBMS ensures, using its cooperative software on the file server, that the referenced files are also restored to their correct state. Our algorithms have been implemented in the IBM DB2/DataLinks product [1] and the DataLinks concept has become part of the SQL standard [10]. This product is currently used in production environment, for engineering designs in large automotive and aerospace enterprises.

A loose transaction model for updates to a file and its corresponding meta-data through a mediator can be a useful notion for directly performing in-place edits of content data residing on stores external to the indexed meta-data store (the latter could be a DBMS), provided there is a way to guarantee consistency between the file contents and the associated meta-data from a reader's perspective. We observe that it is possible to achieve this without holding long duration locks on meta-data tables.

The proposed solution encodes a version indicator in the handle for referencing the object associated with a given meta-data state. A thin interceptor layer on the native store where the object's contents are stored decodes this version indicator and compares it with the version indicator of the latest committed version of the file, to determine if the handle refers to the current version. If the version matches then it checks for uncommitted updates by comparing the last modification time stamp of the file with the last modification timestamp for the latest committed version. If these match, then it allows access to proceed as usual for the file, otherwise it reports an error indicating that the handle refers to stale data.

The solution turns out to be surprisingly simple in hindsight, as it exploits the internal timestamping and cache coherency mechanisms of the two storage engines (i.e. DBMSs and file systems). And yet, as we have demonstrated, it is powerful enough to work for various potential inconsistency scenarios including the situation where a file's contents may have changed via a rename operation, which does not affect the modification timestamp of the file. It is capable of covering cases where the database has been rolled back to an earlier state, as might happen in the case of a point-in-time restore, and should also work with database replication, as long as the legitimacy of the last modified timestamp of the file is maintained during a restore and across replicas.

A major advantage of this approach is that it is suitable for a distributed model for file and meta-data storage since the file content access path checks do not require any direct communication with the meta-data server. The consistency detection scheme prevents an inconsistent view of contents even in the event that the interceptor system becomes out of sync with the DBMS. This turns out to be a very useful characteristic in terms of robustness as well as optimization opportunities. For example, we have shown how this technique extends easily to a configuration where the external store happens to be a distributed file system and the content is accessed directly from distributed file system clients [17]. The approach may be enhanced to work correctly (in terms of preventing access to potentially inconsistent data) with mobile file systems (e.g. Coda) that operate in disconnected mode.

7. ACKNOWLEDGMENTS

The authors would like to thank all the people from IBM Almaden, IBM Silicon Valley Lab, IBM Toronto and IBM India who have contributed to the DataLinks technology over the last few years. Members of the DataLinks development team are Karen Brannon, Vitthal Gogate, Inderpal Narang, Ajay Sood, Mahadevan Subramanian and Parag Tijare. Other early contributors to the DataLinks project are Suparna Bhattcharya, Ashok Chandra, Lindsay Hemms, Joshua Hui, Hui-I Hsiao, Dale McInnis, Nelson Mattos, C. Mohan, Robert Morris, Frank Pellow, Bob Rees and Stefan Steiner. Other major contributors to the design for SQL Mediated Object Manipulation support in DataLinks include Joshua Hui, Parag Tijare, Ajay Sood, Rajagopalan P. Krishnan, S. Ravindranadh Choudhary, Vitthal Gogate, Mahadevan Subramanian and Steven Elliot.

8. REFERENCES

[1] IBM DB2 DataLinks, http://www.ibm.com/software/data/db2/datalinks

[2] Object Management Group. CORBA: The Common Object Request Broker: Architecture and Specifications, July 1995. Release 2.0

[3] Blakely, J. *Data Access for the Masses through OLE DB*, Proc. ACM SIGMOD International Conference on Management of Data, Montreal, June 1996

[4] Blott, S., Relly, L., Schek, H. An Open Abstract-Object Storage System, Proc. ACM SIGMOD International Conference on Management of Data, Montreal, June 1996

[5] Oracle Corporation: Oracle Internet File System, Features Overview, Oracle Internet File System, Frequently Asked Questions: Technical Questions, Oct 2000

[6] Carey, M., DeWitt, D., Naughton, J., Solomon, M., et al. Shoring Up Persistent Applications, Proc. ACM SIGMOD Conference, Minneapolis, MN, pp. 383-394, May 1994.

[7] Cabrera, L.-F., Rees, R., Hineman, W. Applying Database Technology in the ADSM Mass Storage System, Proc. 21st International Conference on Very Large Data Bases, Zurich, September 1995

[8] Narang, I., Rees R. DataLinks - Linkage of Database and FileSystems, Proc. Sixth Int Workshop on High Performance Transaction Systems, Asilomar, September 1995

[9] Mohan, C., Narang, I. An Efficient and Flexible Method for Archiving a Data Base, Proc. ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 1993. A corrected version of this paper is available as IBM Research Report RJ9733, March 1994

[10] ISO/IEC 9075-9-2000, Information Technology – Database Languages – SQL – Part 9: Management of External Data (SQL/MED).

[11] Hsiao, H. and Narang, I., DLFM: A Transactional Resource Manager, Proc. ACM SIGMOD Conf Dallas, Texas , May 14-19, 2000.

[12] IBM, DB2 Universal Database V7, Administration Guide: Controlling Access to Database Objects, 2000.

[13] Narang, I., Mohan, C., Brannon, K. and Subramanian, M. Coordinated Backup and Recovery between Database Management Systems and File Systems, IBM Research Report, RJ10231, Feb 2002.

[14] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging, ACM Transactions on Database Systems, Vol. 17, No. 1, March 1992.

[15] IBM Content Manager, http://www.ibm.com/software/data/cm

[16] Papiani, M., Weson, J., Dunlop, A. and Nicole, D., A Distributed Scientific Archive Using the Web, XML and SQL/MED, ACM SIGMOD Record, Vol 28, No. 3, September 1999.

[17] Bhattacharya, S., Brannon, K. W., Hsiao, H. and Narang, I., Data Consistency in a Loosely Coupled Transaction Model, IBM Research Report, RJ10232, (Feb 2002). (Available from http://www.ibm.com/research/resources)

[18] Schneier, "Applied Cryptography 2nd Edition", J. Wiley & Sons, New York, 1996.