DLFM: A Transactional Resource Manager

Hui-I Hsiao

Inderpal Narang

IBM Almaden Research Center

San Jose, CA 95120

Email: hhsiao@almaden.ibm.com; narang@almaden.ibm.com

Abstract

The DataLinks technology developed at IBM Almaden Research Center and now available in DB2 UDB 5.2 introduces a new data type called **DATALINK** for a database to reference and manage files stored external to the database. An external file is put under a database control by "linking" the file to the database. Control to a file can also be removed by "unlinking" it. The technology provides transactional semantics with respect to linking or unlinking the file when DATALINK value is stored or updated. Further more, it provides the following set of properties: (1) managing access control to linked files, (2) enforcing referential integrity, such as referenced file cannot be deleted or renamed as long as it is referenced from the RDBMS, and (3) providing coordinated backup and recovery of RDBMS data with the file data.

DataLinks File Manager (DLFM) is a key component of the DataLinks technology. DLFM is a sophisticated SQL application with a set of daemon processes residing at a file server node that work cooperatively with the host database server(s) to manage external files. To reduce the number of messages between database server and DLFM, DLFM maintains a set of meta data on the file system and the files that are under database control. One of the major decisions we made was to build DLFM on top of an existing database manager, such as DB2, instead of implementing a proprietary persistent data store. We have mixed feelings about using the RDBMS to build such a resource manager. One of the major challenges is to support transactional semantics for DLFM operations. To do this, we implemented the two-phase commit protocol in DLFM and designed an innovative scheme to enable *rolling* back transaction update after local database commit. Also a major gotchas is that the RDBMS' cost based optimizer generates the access plan, which does not take into account the locking costs of a concurrent workload. Using the RDBMS as a black box can cause "havoc" in terms of causing the lock timeouts and deadlocks and reducing the throughput of a concurrent workload. To solve the problem, we came up with a simple but effective way of influencing the optimizer to generate access plans matching the needs of DLFM implementation. Also several precautions had to be taken to ensure that lock escalation did not take place; next key locking was disabled to avoid deadlocks on heavily used indexes and SQL tables; and timeout mechanism was applied to break global deadlocks.

We were able to run 100-client workload for 24 hours without much deadlock/timeout problem in system test. This paper describes the motivation for building the DLFM and the lessons that we have learned from this experience.

Permission to make digital or hard copies of part or all of this work or personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
MOD 2000, Dallas, TX USA
© ACM 2000 1-58113-218-2/00/05 . . .$5.00

1. Introduction

IBM has focused on extensible database research for more than a decade. The results of many of these efforts have already appeared in IBM's DB2 family of relational database management systems (RDBMSs) [1]. To extend the reach of RDBMS functions even farther, IBM Almaden Research has now developed a new technology called DataLinks [2], which enables DBMS to manage data stored in external operating system files. DataLinks gives DBMS comprehensive control over external data and provides the following properties: referential integrity, access control, coordinated backup and recovery, and transaction consistency. DataLinks technology has now been deployed by several corporations and institutes, such as Boeing, Dassault, and automotive manufacturers, to provide database management of distributed scientific and engineering data stored in operating system files [2, 3].

The DataLinks technology comprises of the following major components: the DLFM, DLFF (DataLinks File System Filter), and an extension to the RDBMS engine (termed datalink engine hereafter). It also introduces a new DATALINK data type [4] to facilitate RDBMS to reference and manage externally stored data. The datalink engine is responsible for processing DDL requests to create datalink column(s) and for processing DML requests against the datalink column(s). On the other hand, the DLFM and DLFF components reside at file servers where the database managed external data are stored. The value of the datalink column in an SQL table is URL, which may reference files on the same or

remote file server. A datalink column can be populated by an SQL insert statement or by a database load utility. Similarly, a record with datalink attribute can be deleted or updated through a standard SQL statement. Whenever a datalink value is selected or updated (or inserted or deleted), the datalink engine is invoked by the database engine to process the part of request specific to datalink. As part of the processing, a request to the DLFM residing on the file server as specified by the URL is sent to apply certain constraints in order to start (or stop) managing the file on that file server. DLFF helps to enforce these constraints when file system commands are executed against the linked files.

Database systems, in general, provide transactional semantics and ACID [5] property. To maintain the transactional semantics for SQL requests, operations performed at DLFM would have to be in the same transaction context as the one in the host database system (where the SQL requests of the application are processed). To satisfy this requirement, operations performed at the DLFM are treated as a sub-transaction[1] [6] of the transaction in host RDBMS and the two-phase commit protocol [7, 8] is used to atomically commit or rollback the operations done at both sides. Also to recover from a system failure, changes to DLFM data and state have to be both persistent and recoverable. One approach is to implement a proprietary persistent store as part of the DLFM. While this is not difficult to do technically, it is less portable and unnecessarily reinvents the technology available in all commercial database systems. As such, our design relies on a database server (DB2) for providing persistency and recoverability for DLFM data/state.

Figure 1 shows an example of the storage model of the DataLinks technology. In Figure 1, a DB2 database is used as a host RDBMS to store user's data and references to external objects via an URL in the datalink column. A DLFM residing with each file server is responsible for managing files stored in that server. System generated metadata for managing files and enforcing access control and integrity is stored in each DLFM. As previously mentioned, DLFM uses DB2 as a persistent store for all of its data. DLFM treats the DB2 as a black box and all requests to retrieve, insert, or update DLFM data/state are via standard SQL.

[1] By sub-transaction it is implied that the host DB2 always resolves the outcome of the transaction on the DLFM side. Standard 2-phase commit protocol is used between the host DB2 and the DLFM. Note that the host DB2 may or may not be the coordinator of the user initiated transaction. For example, the host DB2 can be a participant in an XA transaction that is initiated by a TP monitor, such as, CICS.

While this provides great flexibility and portability[2], it also poses significant challenge in enforcing transactional semantics and providing good system performance.

The rest of the paper is organized as follows. Section 2 gives an overview of the DataLinks technology and DataLinks application flow. Section 3 describes the functions and services provided by the DLFM component. Section 4 presents the experience and lessons we learned in building the DLFM and finally section 5 summarizes the paper.

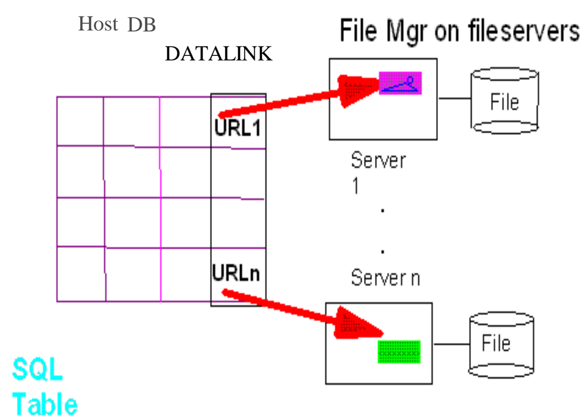


Figure 1: Datalink storage model

2. DataLinks Technology

DataLinks is a software technology that enables DBMS to manage data stored in external operating system files as if the data were stored directly in the database. By extending the reach of the DBMS to operating system files, DataLinks gives users flexibility to store data inside or outside the database as appropriate. To store and reference data outside of DBMS, a database application developer declares a column of DATALINK data type when creating an SQL table. The value stored in the datalink column is then used to represent and reference data in an external file. Figure 2 illustrates the architecture of the DataLinks technology. As shown in the figure, DataLinks comprises two components: datalink engine and Data Link Manager (ref. Figure 2). Datalink engine resides

[2] The DLFM can easily be ported to any RDBMS systems and/or any operating systems.

in the host database server and is implemented as a part of the database (DB2) engine code. It is responsible for processing SQL requests involving datalink column(s) such as table creation and select, insert, delete, and update of records with datalink column. Data Link Manager consists of 2 components, DataLinks File Manager (DLFM) and DataLinks File System Filter (DLFF).

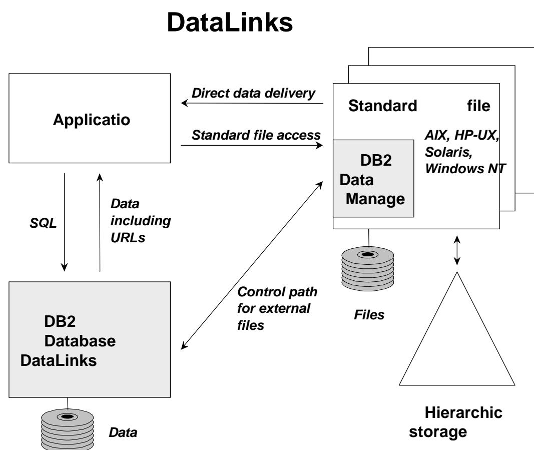


Figure 2: Datalink architecture

At high level, DLFM applies constraints on the files that are referenced by the host database and DLFF enforces the constraints when file system commands or operations affect these files. For example, a file rename or delete would be rejected if that file were referenced by the database. DLFF is a topic on its own and will not be described further in this paper. DLFM resides with the file server that can be local or remote to the host database server. DLFM provides a set of API's that the datalink engine uses to make requests for linking a file, unlinking a file, carrying out two-phase commit protocol, etc. Invoking the API's is through remote procedure call mechanism.

2.1 DataLinks Application

A datalink application is an application that manipulates datalink attribute (column) in an SQL tables and it is no different from a regular database application as far as the database is concerned. Before a datalink application can issue any requests, it must establish a database connection first. As part of the database connect request, a DB2 agent (process or thread) is identified to serve the application. After connection has been established, the application can start submitting SQL requests to the database engine. If an SQL request requires manipulating a datalink column, the datalink engine is invoked to process part of the request specific to the datalink column. In turn, the datalink engine sends one or more requests to the DLFM to manipulate files and metadata stored at file server, if necessary.

Figure 3 illustrates how DataLinks works from an application perspective. In a DataLinks environment, a host database (e.g., DB2 UDB) provides the metadata repository for external data. Attributes and subsets of the data stored in external files are maintained in the host database tables along with the logical references to the location of the files (e.g., a server name and a file name). The application searches the host database via the SQL API to identify external files of interest. Examples would be finding the following: 50-day-moving-average chart of stocks that are tripled in price during the last 12 months; a video clip used in TV commercials within the last year that contains images of Michael Jordan; all of the email attachments received within the last six months that concern customer profiles; or employee who is older than 40 and has blue eyes and red hair. DB2 processes the request and returns the references (URL's) for selected files to the application. The application then accesses the file data directly using standard file-system API calls (file-open, etc.) Using standard file API's is very important for supporting existing applications without having to modify either the applications or the file systems.

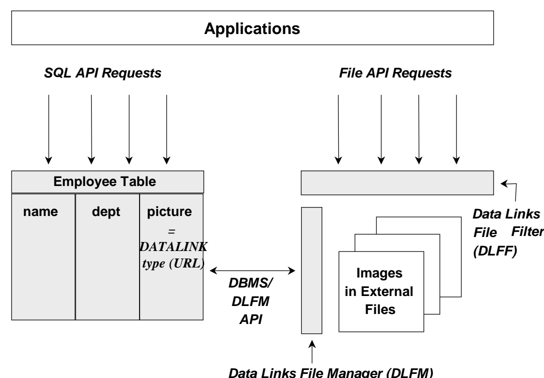


Figure 3: Managing external data with Datalinks

3. Datalink File Manager

The Data Links File Manager (DLFM) component of DB2 Data Links Manager plays a key role in managing external files. It is responsible for executing the link/unlink operations in the same transaction are linked to (referenced in) the database. When a file is initially linked to the database, the DLFM applies the constraints for referential integrity, access control, and

backup and recovery as specified in the DATALINK column definition. If the DBMS controls read access, for example, the DLFM changes the owner of the file to the DBMS and marks the file "read only". All of these changes to the DLFM repository and to the file system are applied as part of the same DBMS transaction as the initiating SQL statement. If the SQL transaction is rolled back, the changes made by the DLFM are undone as well.

In order to support certain SQL operations, such as Drop SQL table, the concept of **File Group** was introduced. A File Group corresponds to all files that are referenced by a particular datalink column of an SQL table. This is so that it is possible to efficiently unlink all files associated with a column of an SQL table when it is dropped. The DLFM is also responsible for coordinating backup and recovery of external files with the database. When the DBMS transaction that includes a *Link File* operation commits, the DLFM initiates a backup of the newly linked file if DLFM is responsible for recovery of the file. This file backup is done *asynchronously* and is not part of the database transaction for performance reasons. In addition, by doing it this way, the database backup itself is not slowed down because the referenced file would typically have been backed up. This is particularly important in the case of very large files. Coordinated backup and recovery of external files with DB2 data can be done directly to disk or to an archive server supported by DLFM, such as IBM's ADSTAR Distributed Storage Manager (ADSM).

The DLFM tracks different versions of a referenced file and maintains the backup status of each in order to support point-in-time recovery. The DBMS also provides the DLFM with a "Recovery id" for a file whenever it is linked or unlinked to help synchronize recovery of files with data. This is important because a file with the same name but different content may be linked and unlinked several times. Without a separate "Recovery id" for each link operation, DLFM would not be able to restore the file to match the database state.

When the DBMS does a backup of its database, it communicates with the DLFM's to ensure that all of the necessary asynchronous copy operations for referenced files have completed before declaring that the database backup has been successfully completed. The DBMS backup utility has been extended to handle this level of communication and to keep additional information in the backup image about which file servers and file groups are involved in the backup. Backup copies of unlinked files may be kept for a specific number of database backup cycles, in case the database is restored to a point in the past in which the file was still linked to the database. The DLFM is also responsible for "garbage collection" of backup copies of unlinked files that are no longer required by the DBMS.

3.1 Persistent Data Structure

The DLFM uses a local database to keep its metadata and state information. This information is stored in the following SQL tables.

1. Dbid Table: This table consists of registered entries for each host database that can connect to this DLFM. The *dbid* field in this table represents the unique combination of the host database name, instance name, and host machine name.

2. Group Table: This table consists of file group entries. Each group entry corresponds to a datalink column in an SQL table on the host database side.

3. File Table: This is the most accessed table that consists of the information of linked and unlinked files on the file server. Whenever a file is linked, a new entry is inserted into the file table. During the unlink operation existing file entry in linked state is marked as unlinked. This table retains the unlinked file entries if files need to be restored in the future via the host database restore utility. The columns of interest defined in this table are *dbid*, *filename, transaction_id, Recovery_id, file_status, entry_state*. Their usage will be described with the functional processing later.

4. Transaction Table: This table keeps track the transaction state of all the active DLFM transactions. Transaction state is maintained for each transaction as long as it is active. The transaction state information is first kept in an in-memory table when the transaction starts. The entry is inserted into the SQL table when the transaction begins the first phase of the commit processing. Once the transaction is completed, its entry is removed from the transaction table.

5. Archive Table: This table contains file and group entries that need to be archived to the archive server. When the load utility is used to insert a large number of files into a datalink column on the host database side, instead of replicating each file entry in the Archive table, only a group entry is inserted into the Archive table. The entry from the Archive table is processed to make copy of a set of files or just one file. After copy has completed, corresponding entry is removed from the Archive table.

3.2 Link and Unlink Operations

LinkFile and UnlinkFile are two most frequent operations that corresponds to insert and delete of the datalink value respectively from the host database. Whenever an application inserts a file entry into a datalink column the corresponding file on the server is

linked by the DLFM. Linking involves applying certain constraints on the file such that subsequent rename and deletion of the referenced file, via normal file system API's (or commands), are prevented to preserve referential integrity from the host database. Furthermore, the access control mode of the datalink column determines the partial or full takeover of the file. In full access control file ownership is changed to "DB" (to the DLFM admin user) and the file is marked read-only. Also an access token assigned by the host database is needed to access such a file. All the files linked to the host database are guarded against unauthorized move / delete / rename operations by the DLFM and DLFF. During the LinkFile operation DLFM puts a new entry in the File table. This entry consists of dbid, transaction id, filename, and Recovery id among other things. Recovery id generated at the host database consists of dbid and a timestamp. It is guaranteed to be globally unique and monotonically increasing. For every LinkFile operation the DLFM makes the following two checks,

1. If a link entry already exists for the same file in the DLFM metadata table then it rejects the LinkFile operation as the file is already in the linked state.

2. If an unlink entry exists for the same file in the DLFM table whose unlink transaction has not committed (i.e. in in-flight or in-doubt state) then it rejects the LinkFile operation as the outcome of the unlink transaction is still unknown.

During an Unlinkfile operation, the table entry for the file is marked as unlinked. It also updates the unlink transaction id and unlink timestamp in the entry. At any given time the DLFM File table can have at most one linked entry for a given file while there can be multiple unlinked entries for a file because many successive link and unlink operations can take place for the same file. The unlinked entry is used in the coordinated backup-and-restore operation to identify the correct version of the file from the archive server, if needed. In this case, the unlinked file entry is later removed by the Garbage Collector daemon (described in Section 3.5) when it is no longer needed. If file recovery is not needed, the unlinked entry is deleted in the second phase of the commit processing. Note that we could not delete the entry earlier than the second phase of commit since we would not be able to undo the action if the transaction's outcome is abort after phase 1 (see Section 3.3).

During the link file operation, file entry checking and insertion **must** be an atomic operation (otherwise there is a small window where two DLFM agents can both check for and not find the linked entry for a file and then insert the two linked entries for the same file). To close the window for the race condition, a unique index on the filename column and a new check-flag is defined.[3] During link file operation, the check-flag attribute is set to zero and during unlink file operation, the check-flag is set to Recovery id provided by the host database. This unique index prevents two linked entries but allows multiple unlinked entries for the same file.

During the forward progress of a transaction DLFM manipulates the entries in the File table as per link/unlink file operations. If the transaction needs to rollback, DLFM uses the recovery mechanism provided by the local database to undo the actions. The file server, on the other hand, does not support transactional semantics in general. Thus, actual takeover or release of the file from the file system is done during the second phase of the commit processing and is done by Chown daemon (described in Section 3.5). DLFM also supports unlinking of a file from one datalink column and re-linking of the same file to another datalink column within the same transaction. This is an important customer requirement where current and old versions of the file are maintained in separate SQL tables.

When an error occurs during regular link or unlink processing, DLFM reports the error status to the host database that will result in either statement level (savepoint) or transaction level rollback at the host database. If a link or unlink file request is initiated by a savepoint rollback at the host database, then any error reported by the DLFM local database will result in rolling back the full transaction at the host database. This is because DLFM treats local database as a black box and it is not possible to rollback a rollback. In addition, if a severe error such as deadlock occurs in the local database, the host database will rollback the full transaction. This is because the current transaction has already been rolled back in the local database. Also since DLFM does not write recovery log records for its own link and unlink file operations, it is not possible to do a database-style rollback. In our design, undoing link (or unlink) file operation is done by sending DLFM another link (or unlink) file request but with a special *in_backout* flag set to *true*. For a link file request with *in_backout* set, DLFM deletes the linked file entry that was inserted by current transaction. For an unlink request with the flag set, the unlinked file entry is restored back to linked state.

3.2.1 Performance Consideration

The File table has at least one entry for each file under database control. In a production environment, it would

[3] Note that for a given file name, there can be multiple entries in the File table and yet DLFM has to ensure that at most one entry is in the linked state. So a unique index on the filename alone is not sufficient.

be common to have hundreds of thousand or millions of entries in the File table. Since each link or unlink file operation needs to access the File table, efficiency in finding and retrieving entries from the File table is essential to provide good overall performance. The first thing we did was to avoid table scan by building several indexes, one for each access path. A side benefit of avoiding table scan is that the probability of triggering lock escalation is also reduced. When the table size (cardinality) is small, the optimizer could still pick table scan even when an index is available. To ensure that the optimizer always picks the access plan we want, the statistics in the database catalog are manually set before DLFM's SQL programs are compiled and bound. In a multi-user test, a different problem surfaced that was partially due to use of multiple indexes. When multiple insert and/or delete entry operations are being done concurrently, different DLFM processes may use different indexes to access the File table. This results in frequent deadlocks because of the next key locking [9] feature supported in the local database server. Since repeatable read is not really needed by the DLFM processes, that feature is turned off. With these enhancements, we were able to run 100-client workload for 24 hours without much deadlock/timeout problem in the system test. Also, the system achieved rates of 300 inserts per minute and 150 updates per minute.

3.3 Transaction Support

When a new transaction is started by the application, the host database assigns a new transaction id. In the case of an XA transaction, the host database also generates a local transaction id that is different from the global XA transaction id. A transaction id is associated with a particular database so that there is no problem with transaction id being the same from different databases. The transaction id generated at a specific database is guaranteed to be monotonically increasing, which is absolutely essential.[4] This id is passed to the DLFM in each of the API invocation. The DLFM associates the transaction id with each operation that changes DLFM metadata and state. The reason is that DLFM *does not* have logging services of its own, but uses a local database for persistence and logging. By associating the transaction id along with the operation, and storing them in the database tables, it can relate the actions performed by a particular transaction. This is important because a) the actions done by a DLFM for a particular sub-transaction may need to be undone if the host transaction aborts after the sub-transaction completing the prepare phase (i.e., completed phase 1 of the 2 phase commit protocol) in the DLFM and b) certain actions on the file system have to be performed during phase 2 of the commit processing of the transaction.

DLFM uses the 2-phase-commit protocol to enforce the transactional semantics. Four API's are provided by the DLFM for this purpose: BeginTransaction, Prepare, Commit, and Abort. A sub-transaction starts when the host database makes BeginTransaction API call to a DLFM.[5] The transaction id generated at the host database is passed along with the BeginTransaction call. All subsequent API calls by the host database within the same transaction for linking and unlinking files are tagged with the same transaction id and are processed within the same transaction context by the DLFM. Once all operations are done under the present transaction, as a part of the commit processing on the host database, it sends a Prepare request to the DLFM. Prepare request processing on the DLFM makes sure that all the operations on the file server are made persistent by issuing an SQL commit to the local database. A separate transaction table is used for keeping the transaction id, its state, and other related information. The transaction entry for the current transaction is not made into the transaction table until the prepare request for the transaction has arrived. After the prepare transaction request is done successfully on all DLFM's, the host database sends a Commit transaction request to the DLFM's. On the other hand, if the prepare request fails, an Abort request will be sent to the DLFM's. It is important to note that, when multiple DLFM's are involved in a transaction, if one of the DLFM's fails to prepare the transaction, the host database sends Abort request to all the remaining DLFM's, even though they may have prepared successfully. Normally, prepare and commit/abort API's are invoked by the host database as part of an application's SQL commit. If the transaction is a branch of a global (distributed) transaction, the prepare request to the DLFM is invoked as part of global prepare processing and the commit/abort request is invoked when the outcome of the global transaction is known.

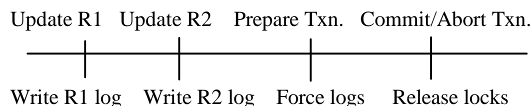
It is assumed that the commit transaction processing should not fail on the DLFM side if the prepare transaction processing has been successful. But that is not always true because there is a major difference

[4] DLFM records the transaction id as persistent information along with other information in the File table. Entries associated with a transaction are identified by this id during the commit processing.

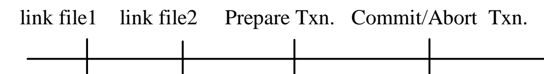
[5] It is possible that files may be linked or unlinked to multiple DLFM's in a given host database transaction. This implies that a host DB2 transaction may involve sub-transactions on multiple DLFM's. In order to improve the readability of the paper we discuss the transaction management with respect to only one DLFM.

between database's SQL commit processing and DLFM's commit processing [ref. Fig 4]. The SQL commit processing does not acquire any new locks. It, in fact, releases all the locks acquired by the present transaction. On the other hand the DLFM uses the SQL interface to update the metadata and its state stored in its local database during commit processing. For a commit request, for example, DLFM retrieves entries from the File table and deletes an entry from the Transaction table. This, in turn, requires additional locks to be acquired by the DLFM. Since deadlocks are always possible when new locks are acquired, retry logic is included in the commit processing and it keeps retrying until it succeeds. However, if a deadlock forms among committing and/or aborting transactions, retry will not solve the deadlock. In our case, deadlocks have been found to form between a committing transaction and one of the DLFM daemons but not between two or more committing and/or aborting transactions. This is because table entries inserted or updated by two concurrent transactions are always disjoint[6]. Thus, our retry logic can solve deadlocks formed in the DLFM commit/abort processing.

SQL Transaction (Txn)



DLFM Transaction



DB: Write log Write log force logs log/rel. locks

Figure 4: Commit processing

During a prepare transaction processing, DLFM inserts an entry into the transaction table and marks the transaction as prepared. If DLFM fails after the transaction has been prepared, then that transaction remains in an in-doubt state. It is the host database's responsibility for resolving the in-doubt transactions with the DLFM. Either host database restart processing does it, or if DLFM is unavailable at the restart, host database spawns a daemon whose sole purpose is to poll the DLFM periodically and resolve the in-doubts when the DLFM is up. In-doubt transactions are resolved based on the outcome of the parent transactions in the host database.

[6] This is enforced by the corresponding locking of the host database.

3.4 Coordinated Backup and Restore

The DLFM plays an important role in the coordinated backup and recovery of DBMS data along with the file data. When the transaction linking a file commits and the file group has recovery option[7] equals yes, DLFM starts archiving that file to the archive server such as ADSM.[8] The DLFM child agent (described in Section 3.5) puts an entry for the file into the Archive table and the Copy daemon picks up the entry from the Archive table and writes the file to the archive server. The main purpose behind the Archive table is to avoid contention in the main metadata table, the File table, and also to efficiently restart copying after recovering from any DLFM failure. Because multiple indexes are defined on the Archive table and size of the Archive table is small (entry gets deleted as soon as it is archived), deadlocks were encountered between child agent and the Copy Daemon while accessing the Archive table. Disabling the next key locking feature in DLFM's local database eliminated those deadlocks. Notice that phantoms may arise when the next key locking is not enforced. However, repeatable read property is not required for the DLFM to function correctly.

Note that the archiving of files is asynchronous when a transaction commits. DLFM does not hold any database locks while backup copy is being made. The asynchronous backup is possible because DLFM takes away the "write" permission of the file during commit operation. The Backup utility on the host database side makes sure that all the files linked since the last backup are archived to the archive server before declaring that backup is successful. In case archiving of some files is pending then it asks the Copy daemon to archive this set of files with high priority.

Restore utility restores the database from a backup image on the host database side. Whenever the host database is restored, DLFM may need to retrieve files from the archive server to match the database state if the linked files are not present in the file system. The database Recovery id at the time of backup is preserved in the backup image that is sent to the DLFM during restore to reconcile its metadata. Based on this Recovery id, all the files that are linked before the backup and unlinked after the backup are restored to the linked state. Similarly, files that are linked after the backup are removed from the linked state. All these actions (entry manipulation) are done via SQL calls to the local database in the DLFM side and we did not find it to be an issue.

[7] Recovery option is one of the properties of the datalink column.

[8] The DLFM also supports the option of backing up the files to a local disk.

The Reconcile utility is a new database utility introduced by DataLinks for synchronizing the host database state with the DLFM metadata information. After a database is restored to a point in the past, database state and DLFM state may be out of synchronization.[9] To bring the two sides back to a consistent state, the reconcile utility is invoked. When invoked, this utility goes through each datalink column, scans all entries for the column on the host database side, and then compares the information with the corresponding file status and metadata information on the DLFM side. It updates the information on either or both sides if necessary to bring the system back to a consistent state. Since the number of entries/records processed could potentially be very large, they are first stored in a temp table in the local database to reduce the number of file scans and the number of messages between the host database and DLFM. The processing on the DLFM side involves complex joins, sub-queries, and EXCEPT (difference) operation between the temp table and the File table, thus picking the right access plans is absolutely essential. To further optimize the performance, we handcrafted the table statistics to ensure that the database optimizer generates the best access plans.

3.5 DLFM Process Model

The DLFM is a concurrent server, i.e., it has a main daemon which spawns a child agent (or a process) when a connect request from a DB2 agent is received. The child agent then establishes a connection with the requesting DB2 agent. This child agent will serve all subsequent requests from the same connection. DLFM's main daemon then waits for another connect request from same or different host DB2. Applications on the host DB2 side will establish separate connections with DLFM, thus they are served by separate child agents on the DLFM side. Besides the child agent, DLFM provides several other services implemented as daemons and they are also spawned by the main DLFM daemon [ref. Figure 5]. This section describes the functionality and service provided by each of the daemons.

Delete Group Daemon

Whenever an SQL table is dropped on the host DB2 side then the corresponding file groups on the DLFM side, if any, will also need to be deleted. There can be lots of files referenced by the datalink column(s) in the dropped table and all those files need to be unlinked. So during the forward progress of the transaction, the file groups are marked deleted by the current transaction in the Group table. During prepare processing the child agent notes the number of groups deleted by this transaction and records it with the transaction entry in the transaction table. The commit processing checks if any group is deleted, by checking the deleted group count in the transaction entry, in the current transaction and if it is, it sends the transaction id to the Delete Group daemon. Using the transaction id the Delete Group daemon finds all the groups deleted in this transaction and then unlinks all the files in each group.

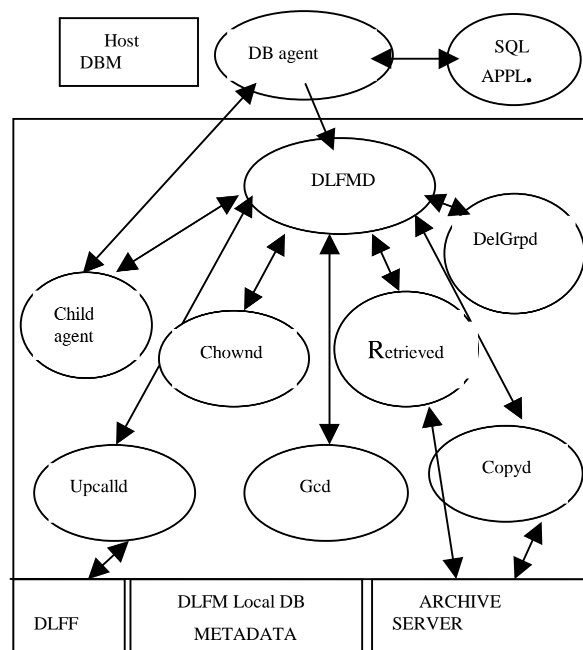


Figure 5: DLFM process model

The unlinking of the files by this daemon is asynchronous and the commit processing for drop table does not wait for it to complete. Note that the group entry is not deleted until all the files in that group have been unlinked. And as long as this transaction does not commit, the same file name is not allowed to be re-linked. Thus if DLFM fails before the Delete group daemon has completed unlinking all files from the deleted groups, then after DLFM restarts the Delete group daemon can still pickup all committed transaction entries from the transaction table and resume its work.

Garbage Collector Daemon

The Garbage Collector daemon is another asynchronous process, which does the cleanup of DLFM metadata. There are two types of cleanups; one is triggered by

[9] Restoring a database to the end-of-log (i.e. current state) does not require any reconciliation.

database backup while the other is to cleanup the deleted group whose lifetime has expired. The one by backup consists of cleaning up old backup entries according to the policy of keeping last N backups. So the last N+1 onwards backup entries and the corresponding unlink file entries from the File table are removed by the garbage collector daemon. It also removes the copies of those files from the archive server. The other one to cleanup deleted groups is based on their lifetime expiry. Each deleted file group is assigned a life span. Once the lifetime expires, the Garbage Collector daemon removes those deleted file group entries as well as associated unlink file entries from the DLFM metadata tables. If archive copies associated with the unlinked file entries exist, they are also deleted from the archive server.

Upcall Daemon

The Upcall daemon services requests from DLFF to determine if a file is in the linked state. If it is, user's request to delete, rename, or move the file via file system API's will be rejected by the DLFF. Its main purpose is to enforce referential integrity for the linked files.

Chown Daemon

The Chown daemon is a special process whose effective user id is *root*. The Chown daemon needs super user privilege as it manipulates attributes (such as ownership, permissions etc) of the files belonging to different users. A child agent communicates with the Chown daemon whenever it needs to get the file information, such as, file system id, inode, last modification time, owner, group etc. During commit processing, the child agent sends a request with a file name to the Chown daemon to take over the file, i.e. change owner and access permissions, or to release the file to the file system to restore original owner and access permissions. Since Chown daemon runs as super user, it is important to safeguard unauthorized requests. Thus, Child agent communicates with chown daemon with proper authentication.[10]

Copy Daemon

The Copy daemon is responsible for copying linked files from file system to an archive server or disk. When a file is linked, it will be copied asynchronously by the Copy daemon if DLFM is responsible for restoring the file after a database restore.

[10] The child agent encodes each message to the chown daemon with the specific signature. The chown daemon validates each message with the signature before doing any operation. The signature is a shared secret between the child agent and the chown daemon.

Retrieve Daemon

The Retrieve daemon is responsible for restoring files from archive server or disk. When the host database is restored to a point in the past, the file system state may be out of sync with the new database state. As part of re-synchronization, files are restored by the Retrieve daemon from the archive server, if necessary.

4. Lessons Learned in Building DLFM

As mentioned previously, we decided to use a DBMS (DB2) as a persistent store for storing DLFM metadata information. All changes to the DLFM metadata are written to the DB2 tables. Since standard SQL does not support two-phase commit between application and database, changes to metadata are hardened[11] during the prepare phase of the 2PC protocol. When something goes wrong in the host DB2 or in other DLFM's, the transaction will be aborted. In such cases, an abort request is sent to the DLFM in the second phase of the 2PC protocol and DLFM has to undo the changes even after they have already been committed in the local DB2. While schemes based on compensation application technique have been proposed for undoing committed transactions, it is extremely complicated to implement one in production systems. Consequently, our design takes a delayed update approach. With this approach, delete of any metadata information is marked as "deleted" while update creates a new entry in the database table with the old entry marked "deleted". When the transaction commits (second phase of commit), entries marked "deleted" in the current transaction are then deleted from the database. If the transaction aborts DLFM then changes these entries back to the normal state from the deleted state. This however, incurs a different problem. During both commit and abort processing, for example, locks will be acquired in the local database since these are normal SQL update/delete calls. This, in turn, may result in deadlocks or lock timeouts in both commit and abort processing. Since it is a sub-transaction and is not possible to change the outcome of a transaction in phase 2, DLFM will retry the commit/abort operation until it succeeds. Our experience has been that this was **not** a problem.

A set of indices is defined on the DLFM tables to improve search performance. We found that deleting a record from a table having index results in the next key locking. Since we have multiple indexes on some of the frequently accessed tables, the next key locking feature

[11] DLFM issues a commit to local DB2 to harden the changes before replying "yes" to a prepare request from the host DB2.

results in deadlocks frequently when multiple datalink applications are running concurrently. To maintain high performance and avoid such deadlocks, we turned off the next key locking in the DLFM database.

Load and Reconcile utilities tend to run for a long time and involve large number of link/unlink operations. Like any other long running transaction, there is a potential for running out of system resources such as log file or lock table entry. Since very long running transactions are always resulted from the database utilities that can be broken into pieces (i.e., undo of completed piece is not needed in case of a utility failure), we put intelligence in DLFM to recognize such transactions and to do local commit after finishing processing of each piece. A transaction entry is inserted into the transaction table in DLFM database when a local commit is issued for the first time for a given transaction but keep the entry marked as in-flight. The same mechanism is also applied to deleting entries in batch. For example, in the delete group daemon we unlink all the files under a deleted group. If a large number of files are linked under one group then unlinking them in a single DB2 transaction can cause the DB2 log full error condition. So we issue commits to local DB2 periodically after processing every N records (where N is implementation dependent).

We found that commit transaction API must be synchronous with respect to host database. Desire was to release the database locks on the host DB2 side while DLFM is doing the commit processing. However, this could lead to a distributed deadlock between the host database and DLFM as shown in the following scenario.

> Transaction T1 is going through commit processing on DLFM side asynchronously. The host DB2 agent for T1 commits and starts a new transaction T11. T11 acquires an X lock on record *x* and then makes a LinkFile request to the DLFM. T11 is blocked on message send as the DLFM child is still doing the commit processing for T1[12] (and has not issued message receive). Assume that the commit processing of T1 on DLFM side is blocked waiting for lock *y* held by transaction T2. If the host DB2 agent for T2 happens to need to access record x, it will also be blocked. Now a deadlock cycle forms and it cannot be broken unless one of the transaction aborts. Since T11 and T2 are not involved in any local deadlock in the host DB2, they will not be aborted by the host DB2. On the DLFM side, T2 is not waiting for any locks and T1's request for lock y will eventually get timeout. But since it is in the phase two of the commit processing, T1 will retry commit and later gets timeout again. This process will repeat forever as the deadlock cycle persists. By making commit request synchronous, distributed deadlock like the one above was avoided.

As in most distributed systems, identifying and breaking distributed deadlock is an important issue. While a distributed deadlock detector can be built in theory, it will add significant complexity and overhead to the system as host DB2 and DLFM database do not communicate directly. Instead, we take a simple approach and rely on the timeout mechanism to resolve potential distributed deadlock. The problem with the timeout mechanism is that it is difficult to come up with a perfect timeout period and some transactions may get rollback unnecessarily. In our case, we set the timeout to 60 seconds and it has performed reasonably well. Another problem related to locking is lock escalation. When a DLFM process holds lots of row level locks in a metadata table then it may result in a lock escalation to table level lock. The lock escalation for a high traffic table will result in timeouts for other applications. The rollback operations as a result of timeouts in turn add additional workload to the system. We observed that lock escalation in any of the metadata tables usually brings the system to its knees. Within our daemons, we are careful that they commit frequently enough so as to avoid any lock escalation. Also, applications should issue commit frequently to avoid holding a large number of locks and lock list size should be set sufficiently large to avoid forced lock escalation.

Cost based optimizer is the most advanced database optimizer and it has been used in most commercial database systems. We observed that Cost based Optimizer does not take locking cost (concurrent accesses) into account when choosing an index for access. In certain cases it also chose an index that was not only sub-optimal but also caused table scan, instead of index scan, to evaluate predicates. To get the desired access plan, we wrote a utility to set the statistics in the database catalog to force optimizer to select the plan we want. While this works in the lab, issuing a Runstat operation by any user will overwrite the handcrafted statistics and potentially result in sub-optimal plan being generated again. To prevent this from happening, additional logic is put into DLFM to check for changes in metadata statistics and re-invoke the utility to reset statistics and rebind access plans, if necessary.

5. Summary

In summary, DataLinks meets a very challenging application requirement that has existed for many years. DataLinks enables organizations to continue storing

[12] Recall that the same DLFM child is used to serve all requests for the same application on the host database.

data (particularly large files of unstructured or semi-structured data such as documents, images, and video clips) in the file system to take advantage of file-system capabilities, while at the same time coordinating the management of these files and their contents with associated data stored in an RDBMS.

DLFM is a key component of the DataLinks technology developed at the IBM Almaden Research Center. It plays a key role in enforcing access control, providing referential integrity, and supporting coordinated backup and restore. DLFM uses a DBMS as a persistent store for storing its data (metadata) and state change information that takes the advantage of existing database technology and at the same time offers excellent portability. Doing this, however, has its drawbacks too. Because the DBMS used is treated as a block box, one of the major challenges is to support transactional semantics for DLFM operations. To do this, we implemented the two-phase commit protocol in DLFM and designed an innovative scheme to enable rolling back transaction update after a commit to the local database. Also, a major gotchas is that the RDBMS' cost based optimizer generates the access plan, which does not take into account the locking costs of a concurrent workload. Using the RDBMS as a black box can cause "havoc" in terms of causing the lock timeouts and reducing the throughput of a concurrent workload. To solve the problem, we came up with a simple but effective way of influencing the optimizer to generate access plans matching the needs of the DLFM implementation. Also, several precautions had to be taken to ensure that lock escalation did not take place, that the next key locking was disabled to avoid deadlocks on heavily used SQL tables with multiple indexes, and that the timeout mechanism was applied to break deadlocks. In the system test, we were able to run 100-client workload for 24 hours, with a reasonably heavy update activity, without much deadlock/timeout problem.

Acknowledgement

Many people contributed to building DataLinks technology over the last few years. Major contributors include Suparna Bhattacharya, Karen Brannon, Kiran Mehta, Suhas Gogate, Mahadevan Subramanian, Ajay Sood, Parag Tijare, Dale McInnis, Lindsay Hemms, Jason Gartner, Nelson Mattos, Robin Williams, and last but not least Ashok Chandra. The authors would like to thank them for their dedication and contribution that made this paper possible.

References

[1] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks", CACM 13, p 377-387, 1970.

[2] IBM, "DataLinks: Managing External Data With DB2 Universal Database", white paper prepared by Judith R. Davis, IBM corporation, February 1999.

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

[4] N. Mattos, J. Melton, and J. Richey, "Database Language SQL – Part 9: Management of External Data (SQL/MED)", ISO working draft, June 1997.

[5] T. Hearder and A. Reuter, "Principal of Transaction Oriented Database Recovery", ACM Computing Surveys. 15(4), p287-317, 1983.

[6] J. Moss, "Nested Transactions: An Approach to Reliable Computing." MIT, LCS-TR-260, 1981.

[7] B. Lindsay, et al., "Notes on Distributed Databases", IBM San Jose Research Laboratory, RJ 2571, 1979.

[8] G. Samaras, K. Britton, A. Citron, C. Mohan, "Two Phase Commit Optimization in a Commercial Distributed Environment", Distributed and Parallel Databases Journal, 3(4), 1995.

[9] C. Mohan, "ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions on B-Tree Indexes", 16th VLDB. P392-405, 1990.