RJ 10232 (A0202-020) February 20, 2002
Computer Science

IBM Research Report

Data Consistency in a Loosely Coupled Transaction Model

Suparna Bhattacharya
IBM Software Lab
Golden Enclave
Airport Road
Bangalore 560017
India

Karen W. Brannon, Hui-I Hsiao, Inderpal S. Narang
IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

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

Data Consistency in a Loosely Coupled Transaction Model

Suparna Bhattacharya‡, Karen W. Brannon†, Hui-I Hsiao†, Inderpal Narang†

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

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

Abstract

Managing a combination of database data and file data in a consistent manner is an interesting challenge in content management technology. Typically, meta-data referencing/indexing external files is created and stored in a database for efficient search and retrieval of file data. Tight coupling of meta-data and file-updates is unsuitable as it makes the meta-data inaccessible during the potentially long process of editing and refining content. We propose an efficient solution to the problem of maintaining consistency between the content of the file and the associated meta-data 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 filesystem 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. A temporally unique version indicator associated with the last committed update transaction is encoded in the object reference associated with a given meta-data state. The current last modification timestamp of the file is available from the filesystem. A thin interceptor layer 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. The solution is simple as it utilizes the internal mechanisms for timestamping and cache coherency employed by the independent stores linked, i.e. the database and the filesystem. The approach prevents an inconsistent view of contents, even when the interceptor system becomes out of sync with the DBMS. This aspect is utilized for enabling reliable client initiated caching schemes for optimizing performance in a distributed filesystem setup with authoritative caching, by triggering sync-ups only when a potential inconsistency is detected.

1.0 Introduction

RDBMSs have now been used to manage almost all of traditional business data in a robust fashion. Neverthless, a large fraction of world's data is still stored outside of databases, e.g.,in filesystems. Content management is a technology that intends to manage both database data and file data in a consistent way.

Content Management (CM) Systems, especially those designed for managing distributed content may 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.

A much improved approach, based on the loosely coupled transaction idea, is implemented in IBM Content Manager [1]. 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 consist 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 filesystem 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 filesystem 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 filesystem or an object store which is external to the database. The external file reference is maintained within the database using the DATALINK SQL datatype which is described as part of the SQL/MED ISO standard [2]. The DataLinks technology which provides referential integrity, coordinated backup and recovery, and data value based access control for content stored in files but with a reference to the file stored in a database table is available in the IBM DB2 DBMS product [3] [4] [5] [6]. Support for meta-data and file content consistency in a loosely coupled transaction model as described in this paper is not yet available in DB2-DataLinks.

2.0 The Loosely Coupled Transaction Model

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.

2.1 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 filesystem 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 filesystem 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 passing in the results of a scalar function that builds the object reference datatype based on the parameters describing the location of the object and some control attributes. This is also referred to as “linking” the file to the DB. Once a file is “linked” to the DB, it is under DBMS control and is managed by the DBMS. To access (read or write) the object, a user obtains a handle to reference the object from the results of an SQL SELECT on the table based on the identification criteria. The user can then use the handle to access the object directly from the external store, by supplying the handle as the name of file to the native filesystem 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.[7]. The object can be disassociated from the database by issuing an SQL DELETE operation, referred to as “unlinking” the file from the DB, at the end of which the file is no longer under DBMS control but completely back under filesystem control.

A interceptor/filter module services direct content access using the handles described above and is illustrated in Figure 1. The filter module transparently intercepts accesses to the external object store and is used to enforce referential integrity requirements and to validate the handle which contains an embedded access token used for referencing the object. The validation is required primarily for access control purposes in order to support meta-data value based access control determined by the DBMS[8] . This module communicates with a file/object management mediator daemon that tracks and manages objects on the filesystem which are linked to the database. It has a local repository to store the 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, this filter layer 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.

3.0 The Consistency Detection Scheme

3.1 Design Considerations

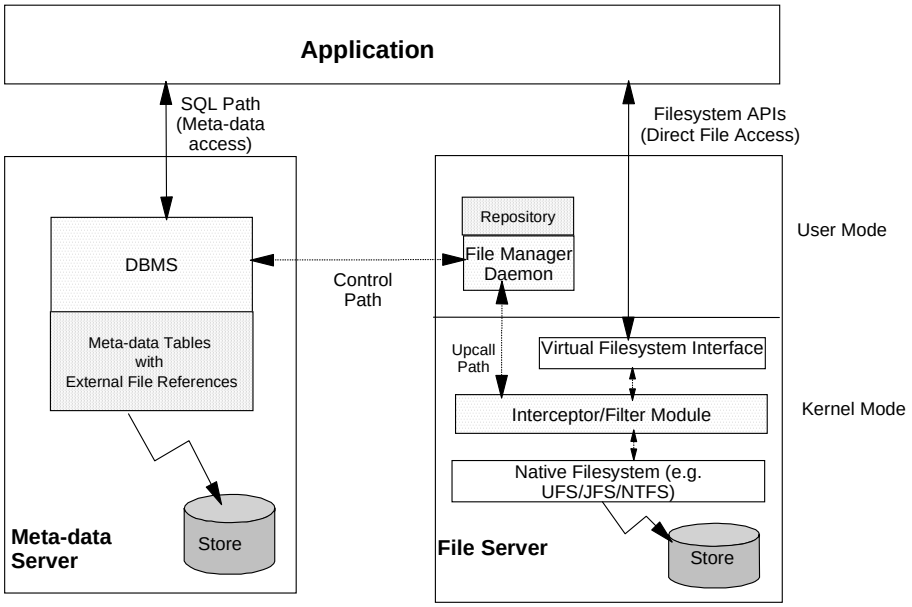


Fig 1: System Architecture

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 filesystem 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 a 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 filesystem. The idea here, however, is to avoid imposing such clock synchronization requirements on the database and filesystem servers for supporting consistency detection. This keeps the administrative overhead and complexity low.

Another requirement is the suitability of the scheme for a distributed filesystem (or object store) implementation with authoritative caching at filesystem clients. The problem that arises in the presence of “authoritative caching” where the filesystem client system 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 filesystem clients. Figure 2 illustrates extensions to the architecture as shown in Figure 1 for support of distributed file system clients. Since the DBMS communicates information about “linked” objects to the file server, the client side interceptor may need to consult the server in order to check for consistency. Thus the ability to make efficient use of caching and reduce communication overheads without losing correctness is required in order to achieve acceptable levels of performance. This is more relevant in situations when the filesystem client is able to service the file access request from its

cache without interaction with the file server. The nature of information used for detecting consistency needs to be chosen in a manner that makes this possible while deterministically guarding against ever exposing readers to inconsistent data. For example, one of the alternative schemes under consideration involved making a ticky mark against the file in the interceptor's (or rather file management daemon's) data structures to keep track of and check for in-progress-updates. In a distributed file system setup, however sharing this information across all filesystem clients becomes a little complicated. Such external state is not sufficient in itself for a reliable client initiated caching scheme that can assure consistency without consulting the server even if the file hasn't actually changed since the last time the client checked with the server. The modification timestamp of the file, on the other hand provides such an indicator, as existing internal cache coherency schemes (stateful ones in the case of DCE-DFS[9]) employed by the distributed filesystem already ensure that it gets reflected at every client.

An underlying design principle is simplicity and reuse of existing internal mechanisms of the two storage engines, that is the DBMS and the filesystem, 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.

3.1 The Source of Inconsistencies

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:

- If uncommitted updates are made to the file any time before the token is used to access the file
- If earlier updates get committed any time after the token was generated.
- If the database gets restored to a prior time before the token is used to access the file.

The file updates above may have happened after the token is generated, or may have been pending from before the token generation. 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. Updates 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), or 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.

3.2 Basic Solution for the Local Filesystem Case

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 filesystem or external store. Since updates are committed with the knowledge of the mediator DBMS in communication with its file/object management daemon component on the file server, the latest version number for the object reference associated with the meta-data store can be maintained both in the DBMS and in the local repository on the file server. Note, that such a generation number is also required for point in time recovery purposes. In the DB2 implementation [2-4] for support of the SQL DATALINK datatype, the recovery id associated with the file, derived from the Log Sequence Number, LSN of the transaction in which the update was committed, may be used as the version indicator. 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. Note that 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 mediator 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 filesystem to open the file, the filter layer which intercepts these filesystem 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 filesystem.
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 updator 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. This is to address the case depicted in Section 4.3.7.

Application level file locking must be used by readers and updaters 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.

3.2.1 Encoding the Version Number in the Token

The access token contains a message authentication code, MAC which is a one-way hash value with the addition of a secret key[10] . 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 filesystem 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 name, servername, token length, 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 name, server name, token length, 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.

3.3 Extension to a Distributed Filesystem Environment

The following scheme could be used to extend the above solution to a distributed environment for accesses to the file from distributed filesystem clients while minimizing performance overheads in communication with the central file server where the file manager daemon's repository is located, especially in situations where the filesystem client is likely to service accesses from its cache without connecting to the server. It is assumed that the distributed

filesystem implements its own cache coherency mechanisms. For example DCE-DFS has a token manager which is used by the cache manager for these purposes.

A proxy daemon illustrated in Figure 2 may be set up at each client to service upcalls from a filter module which intercepts file accesses on the client. This proxy daemon could contact the central server / repository when required. As described earlier, the requirement is to minimize these contacts, especially for read performance in the most typical cases, i.e. repeated accesses to the same file should be servicable from the cache without contacting the server as far as possible.

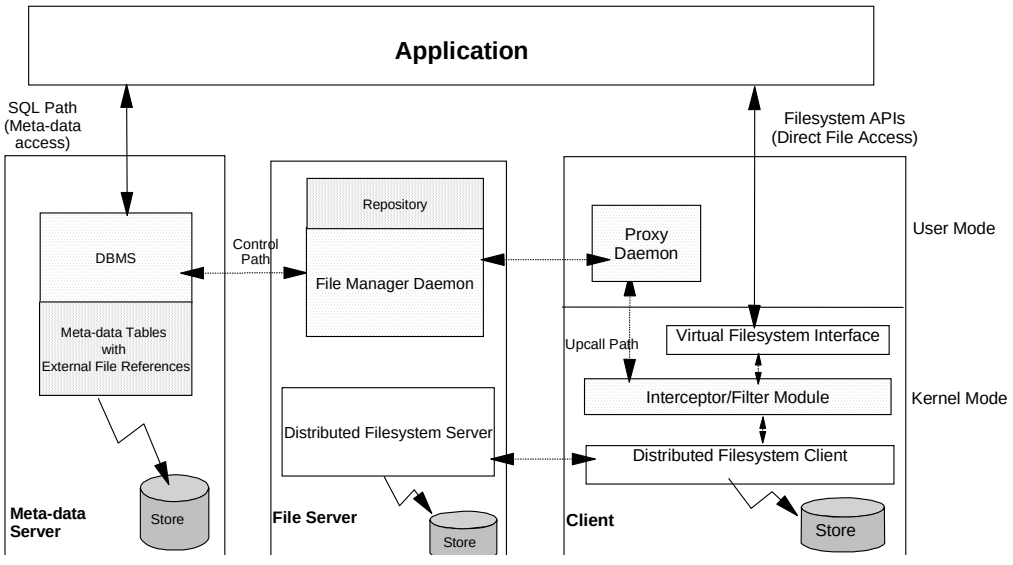
3.3.1 First Time Access

The first time a file is accessed with a token at the client, the client proxy daemon would contact the remote central repository and make an entry in a local (possibly in-memory) table with the latest committed version number, V(commit) of the file, the value of the corresponding file identifier, Fid(commit) of the latest version of the file and the modification timestamp of the file when updates to it were last committed, Tm(commit).

The check for stale data would be:

$$is\_stale = (V(token) \neq V(commit)) \ OR \ (Tm(access) \neq Tm(commit))$$

3.3.2 Cached Case (Repeated Accesses):



The next time the same file is accessed (either with the same or a different token), the proxy daemon can simply look up its local table entry and if Fid(access) == Fid(commit), then it can use the in-memory entry to perform the above check for *is_stale*.

Fid(commit) and Fid(access) refer to the file-id value at the time of commit and time of access respectively, i.e. Fid(commit) = value of Fid (Filename, Tm(commit)) where Fid(f, t) = value of unique file id of the file named "f" at time "t".

If *is_stale* is FALSE, access can continue without a need to contact the server. Otherwise, it would need to refresh the entry from remote central repository, and perform the check again.

The reason for explicitly introducing the unique file id comparison in the distributed case, is to correctly handle cases where the file may have been unlinked, renamed and relinked under the same name (as in Scenario 2 in Sec 4.3.2). By indexing V(commit) and Tm(commit) against Fid(commit) in the local in-memory table, and looking up the cached entry against Fid(access) we are sure that the version number and Tm values refer to the same file[1], so that a comparison with Tm(access) makes sense. This is important in the distributed case because the cached value of V(commit) may be stale, making Tm(access) the real determinant of whether the entry should be considered stale or not. Note that the filesystem client already has a reliable way (using its internal cache coherency mechanisms) of providing the correct latest value of Tm(access) in collaboration with the filesystem server if necessary in case the state of the file has changed since it was last accessed.

Just as in the local case, if Tm(access) ≠ Tm(commit) and the token is a write token, then an optional step of checking if the file is update-in-progress with the same token/user may be involved (e.g by checking state information in the repository).

With this approach, there is a need to contact the central repository on the server only if the file changed or got relinked since the last time the entry was looked up. In both these situations the

[1] The unique file id maintained by a distributed filesystem usually includes some kind of per inode generation number of its own in order to account for reuse of inode numbers after files are deleted. This generation number is typically incremented every time the inode number is reassigned to a newly created file. This ensures the temporal uniqueness of a file identifier.

file system client would also have had to contact the server in order to retreive the latest state of the file, so the relative incremental overhead of the consistency checking scheme is of the same order. In fact, in both these situations as with a first time access, the client would most likely also have to perform network transfers of file data in order to provide up-to-date data to the application, which makes the relative cost of the consistency checking scheme even less significant.

The in-memory table entries can be flushed/reused based on the expiration time and resource limits.In the cached access case if the file hasn't changed since the last access, a need to contact the server could arise if the in-memory table is completely used up.

4.0 Scenarios Analysed

This section discusses various key scenarios, and how the proposed solution handles each of these situations to prevent potential inconsistencies from arising.

4.1 Assumption

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 a normal filesystem 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.

4.2 Definitions

A set of definitions will be useful to describe various scenarios where inconsistencies may arise. A scenario is a list of events, where each event is taken from one of the following definitions.

Qr(f,t)	User does the SQL query and the read access token for file f is assigned at time t
Qw(f, t)	User does the SQL query for write token and the write token for file f is assigned at time t.
C(f,t)	Actual contents of the file f at time t

Qc(f, t)	Contents of the file f that would be consistent with the access token generated at t, i.e the file data corresponding to the results of the query at time t
Rs(f,t)	Reader opens the file f with access token generated at t
Re(f,t)	Reader closes the file f (finished reading it) corresponding to Rs(f,t)
Us(f,t)	Writer opens the file f for update using the write token generated at t
Ue(f,t)	Writer closes the file f (finishes writing it) corresponding to Us(f, t) (but not yet committed to DB using SQL UPDATE).
Uc(f,t)	Metadata in DB is updated and linked together with the file-data updates for file f. The version indicator changes (set to a new value guaranteed to be higher than the last).
I(f)	File f is linked (via SQL INSERT) to the database. The version indicator changes (set to a new value guaranteed to be higher than the last).
D(f)	File f is unlinked (via SQL DELETE) from the database (restore on unlink)
Us(f)	Writer opens the file for update without a token (when the file is not linked to the DB)
Ue(f)	Writer closes the file (finishes writing it) corresponding to Us(f) (the file is not linked to the DB at this time)
M(F,f)	File F is moved to f (could be rm f, mv F f) results in f getting replaced by F - both data + attributes (including timestamp) C(f, ta) <= C(F, tb), where ta < M(F, f) < tb
RB(f,t)	Database point-in time rollback results in reverting the file and meta-data to the state corresponding to time t. The version indicator gets reverted as well.

A few more definitions to help with some of the analysis or explanations:

C(f,t)	Actual contents of the file f at time t
Cq(f, t)	Contents of the file f that would be consistent with the access token generated at t, i.e the file data corresponding to the results of the query at time t
T(e)	Time of occurance of event e (i.e just at the point of completion of the event, except for pure read type events like Rs(f, t) which do not cause a change in meta-data/file reference/content state)

4.3 Addressing Potential Inconsistency Scenarios

It should be the responsibility of the application to ensure serialization so that Rs-Re essentially happens under a shared file lock and Us-Ue happens under an exclusive file lock, but note that multiple Us-Ue iterations can happen before Uc to finally commit the update, and an Rs-Re can happen between two such Us-Ue sequences.

4.3.1 Scenario 1: Query - Unlink - Update - Relink - Access

A file is unlinked and linked back again after modifying contents (not in-place update), and then accessed using a token generated prior to the unlink.

I1(f); Qr(f, t); D1(f); Us (f); Ue (f); I2(f); Rs(f,t)

$Cq(f, t) = C(f, T(I1(f))) = C(f, t) \neq C(f, T(Rs(f,t)))$

The file data consistent with Qr(f, t) is C(f, T(I1(f))) or C(f, t). C(f, T(Rs(f,t))) differs from C(f, t) as the file has been written to in between while it was "unlinked".

With the solution proposed, Rs (f,t) would fail since V(token) < V(commit), i.e., the version-id embedded in the read-token is earlier than the one after I2(f) (the insert operation I2 causes the change in version number). If I2(f) had not happened, then Rs(f,t) would fail since the file wouldn't be linked to the database and the file filter would detect this and ignore the read token embedded in the file pathname.

4.3.2 Scenario 2: Query - Unlink - MoveTo - Relink - Access

A file is unlinked and linked back again after renaming another file to this file name , and then accessed using a token generated prior to the unlink (that token was meant to represent the original file, not the just renamed one)

I1(f); Qr(f, t); D1(f); M(f', f); I2(f); Rs(f,t)

$Cq(f, t) = C(f, T(I1(f))) = C(f, t) \neq C(f', t) = C(f, T(Rs(f, t)))$

The file data consistent with Qr(f, t) is C(f, T(I1(f))) or C(f, t). C(f, T(Rs(f,t))) == C(f', t < T(M(f', f))) which differs from C(f, t) (the file has changed to a different one in between).

As in Scenario 1, above, with the solution proposed, Rs (f,t) would fail since

V(token) < V(commit),

i.e., the version-id embedded in the read-token is earlier than the one after I2(f). In the distributed filesystem case, where the client's in memory table has a stale value of V(commit)

and Tm(commit), the condition [Tm(access) for f' > Tm(commit) for f] may not suffice to detect stale information. However the check Fid(access) = Fid(commit) fails, and triggers a refresh of the values of V(commit) and Tm(commit) from the server.

4.3.3 Scenario 3: Query - Get write token - Update file in-place - SQL update commit - Access

The file is updated in place and committed and then accessed using a token generated prior to starting the update.

I(f); Qr(f, t1); Qw(f, t2); Us(f, t2); Ue(f, t2); Uc(f, t2); Rs(f, t1)

$Cq(f, t1) = C(f, T(I(f))) = C(f, t1) \neq C(f, T(Rs(f, t1)))$

The file data consistent with Qr(f, t1) is C(f, T(I(f))) or C(f, t1). C(f, T(Rs(f,t1))) differs from C(f, t1) as the file contents have been updated.

With the solution proposed, Rs (f,t1) would fail since V(token) < V(commit).

4.3.4 Scenario 4: Update-file in place - Query - SQL Update commit - Access

The file is updated in place and committed, and then accessed using a token generated after the file modifications were completed, but before the SQL Update was issued to complete the corresponding meta-data updates.

I(f); Qw(f, t1); Us(f, t1); Ue(f, t1); Qr(f, t2); Uc(f, t1); Rs(f, t2)

$Qc(f, t2) = C(f, T(I(f))) = C(f, t1) \neq C(f, T(Rs(f, t2)))$

The file data consistent with Qr(f, t2) is C(f, T(I(f))) or C(f, t1) [not C(f, t2), since the meta-data has not changed yet at t2, but only after that]. C(f, T(Rs(f,t2))) differs from C(f, t1) as the file contents have been updated between t1 and t2.

With the solution proposed, Rs (f,t2) would fail since V(token) < V(commit).

Another case which can be considered equivalent to the above for our purposes:

I(f); Qw(f, t1); Us(f, t1); Qr(f, t2); Ue(f, t2); Uc(f, t1); Rs(f, t2)

With the solution proposed, Rs (f,t2) would fail since V(token) < V(commit).

4.3.5 Scenario 5: Get Write tokens 1 and 2 - Update-file in place with token 1 - SQL Update Commit - Access with token 2

The file is updated in place and committed using one write token and then accessed with another write token, that was generated prior to the update.

I(f); Qw(f, t1); Qw(f, t2); Us(f, t1); Ue(f, t1); Uc(f, t1); Rs(f, t2) or Us(f, t2)

$Cq(f, t2) = C(f, T(I(f))) = C(f, t2) \neq C(f, T(Rs(f, t2)))$

The file data consistent with Qw(f, t2) is C(f, T(I(f))) or C(f, t2). C(f, T(Rs(f,t2))) differs from C(f, t2) as the file contents have been updated. This case is very similar to Scenario 3 except that this happens with a write token instead of a read token.

With the solution proposed, Rs (f,t2) would fail since V(token) < V(commit).

4.3.6 Scenario 6: Get Write token 1 and read/write token 2 - Update-file in place with token 1 - Access with token 2 - SQL Update Commit

The file is updated in place using one write token but not yet commited and then accessed with another write token, that was generated prior to the update.

I(f); Qw(f, t1); Qr(f, t2)/Qw(f,t2); Us(f, t1); Ue(f, t1); Rs(f, t2) /Us(f, t2); Uc(f, t1);

$Cq(f, t2) = C(f, T(I(f))) = C(f, t2) \neq C(f, T(Rs(f, t2)))$

The file data consistent with Qr(f, t2)/Qw(f, t2) is C(f, T(I(f))) or C(f, t2). C(f, T(Rs(f,t2))) differs from C(f, t2) as the file contents have been updated (by the same user, though).

With the solution proposed, Rs (f,t2) would fail since Tm(access) > Tm(commit)
[Note: However, for the same user-id, it may make sense for the read to be allowed, and the updates to be visible to the reader. This is accomplished by changing the ownership of the file to the updater's userid for the duarion of file update until commit. This makes it possible to

perform the above timestamp check only if the reader is not the same userid as the current updater]

4.3.7 Scenario 7: Get Write token - Update-file in place with token - Access again with token - SQL Update Commit

We need to be able to differentiate Scenario 6 from the following situation, which we shouldn't consider as an inconsistency: The file is updated in place using a write token and then accessed again with the same write token without committing

I(f); Qw(f, t1); Us(f, t1); Ue(f, t1); Rs(f, t1)/Us(f, t1); Uc(f, t1);

Here, as the same token that was used for the first update is used again, it is more appropriate that it return the latest state of the file data as written using earlier with this token. Note: There are two options to handle the case of a second write-token for the same user: deny write access to even to the same user with different token (than what is already tracked) or be liberal to allow update based on the same user-id. If we decide to make uncomitted updates visible and modifiable on accesses by the same user, then this situation would be taken care of using a similar logic as described under the previous point. If we decide on the other option, since the write-token is tracked until the update is committed, it should be possible to detect when a different token is used. (In the distributed filesystem extension, this check should happen at the end after all other conditions are covered, as it would require interaction with the file server) There is an implicit assumption here that other users wouldn't be allowed to use the write-token as long as an update is in progress.

4.3.8 Scenario 8: Get write token 1 - Update file in place - Get read token 2 - DB rollback to earlier version - Access with read token 2

A database rollback to an earlier version happens, and the file is accessed with a token that was generated prior to the rollback (and hence corresponding to a later version, which is no longer valid after the rollback)

I(f);Qw(f, t1);Us(f, t1); Ue(f, t1); Uc(f, t1); Qr(f, t2); RB(f, t1); Rs(f, t2)

$Cq(f, t2) = C(f, T(Uc(f, t1))) = C(f, t2) \neq C(f, t1) = C(f, T(I(f))) = C(f, T(Rs(f, t2)))$

The file data consistent with Qr(f, t2) is C(f, t2) i.e. C(f, T(Uc(f, t1))), while after the rollback, file contents are C(f, t1) which is the same as C(f, T(I(f))).

Note: This is because the coordinated backup and recovery support in DataLinks reverts the file contents along with the database state in a consistent manner [6]. This is initiated by the DBMS through the file management daemon in coordination with a file archive/restore agent. The recovery id associated with a link/update operation (i.e the version indicator) is utilized for achieving the co-relation between database state and the right instance of object state during recovery. .

With the solution proposed, Rs(f, t2) would fail since in this case V(token) > V(commit), i.e. the version-id embedded in the read-token is different from the one after I(f). In the distributed filesystem case, where the client's in memory table has a stale value of V(commit) and Tm(commit), the condition V(token) == V(commit) could succeed and thus may not suffice to detect stale information (since the client may have cached the version id corresponding to t2, i.e. Uc(f t1) and not yet be aware of the rollback to t1). However in that situation, the check Tm(access) = Tm(commit) would fail (because the cached value of Tm(commit) is still t2 so far, while the rollback would have reverted the file's actual modification timestamp back to t1), and trigger a refresh of the values of V(commit) and Tm(commit) from the server.

5.0 Summary

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 filesystems). 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 filesystem and the content is accessed directly from distributed filesystem clients. It works efficiently even in the presence of authoritative caching (as in DCE-DFS), with minimal communication overheads, and does so using an entirely client initiated caching approach, without any client specific state being required to be maintained on the servers. This is because it is possible to trigger sync ups of the

required state (i.e the latest known version and its corresponding timestamp information) lazily when potential inconsistencies are detected without any risk of ever exposing inconsistent data to a reader. The approach may be enhanced to work correctly (in terms of preventing access to potentially inconsistent data) with mobile filesystems (e.g Coda) that operate in disconnected mode.

6.0 Acknowledgement

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 Datalinks technology over the last few years. Major contributers to the design for SQL Mediated Object Manipulation support in Datalinks include Joshua Hui, Ajay Sood, Rajagopalan P. Krishnan, S. Ravindranadh Choudhary, Parag Tijare, Vitthal Gogate, Mahadevan Subramanium and Steven Elliot.

7.0 References

[1] Blaine Lucyk, “Controlling the Digital Deluge: IBM Content Manager”, DB2 Magazine, Summer 2000.

[2] ISO/IEC 9075-9-2000, “Information technology - Database Languages - SQL - Part 9: Management of External Data (SQL/MED)”

[3] IBM, http://www.software.ibm.com/data/datalinks

[4] Judith R. Davis, “Datalinks: Managing External Data with DB2 Unviversal Database”, White paper, February 1999.

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

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

[7] M. Papiani, J. Weson, A. Dunlop, and D. Nicole, “A Distributed Scientific Archive Using the

Web, XML and SQL/MED, ACM SIGMOD Record, Vol 28, No. 3, September 1999.

[8] I. Narang, C. Moha, K. Brannon, and M. Subramanian, “Coordinated Backup and Recovery between Database Management Systems and File Systems”, Internal IBM Report.

[9] Open Software Foundation, White Paper: “File Systems in a Distributed Computing Environment”, 1991.

[10]. Schneier, “Applied Cryptography 2nd Edition”, J. Wiley & Sons, New York, 1996.