Damage-Resistant Component Files

Using journaling and other techniques to provide data safety

Richard Smith

Dyalog Ltd

Minchens Court, Minchens Lane, Bramley, RG26 5BH, United Kingdom

richard@dyalog.com

Main topic: Theory and practice in array processing language design and implementation

Abstract

Dyalog **component files** are maintained by the interpreter using index trees and other high-level structures within the file. Each component file update (e.g. a single ⎕FAPPEND or ⎕FREPLACE) results in a number of updates to different parts of the file; during this process the file can be in an inconsistent state and abnormal termination may leave the file damaged.

This paper presents techniques such as journaling and checksumming which Dyalog has used to protect component files from damage caused by events such as termination of an application during an update, network failures, and loss of file caches following a catastrophic event such as a power failure or operating system crash.

Overview of the traditional Dyalog component file system

A component file is presented to the user as a contiguous sequence of components, with each containing a single APL array. Because a component can be replaced with another of a different size, updates would be extremely inefficient if data was actually stored in this way.

The Dyalog interpreter maintains component files in a way which has proved itself to perform well in terms of both performance and space allocation. Each component file is stored in a single binary file on the host system but the components are not stored contiguously within it. Each component itself *is* a single contiguous block of arbitrary length, but the blocks are placed wherever they can be best fit, and component index blocks are used so that they can be located. These component index blocks also occupy space in the file and are also placed where they can be best fit. When components are dropped and replaced, gaps are created within the data in the file which are subsequently reused (in full or sections) for new data. The gaps are themselves linked so that they may be quickly located. ⎕FRESIZE performs a file compaction by removing all the gaps. Other than on ⎕FRESIZE, components in the file are never moved, and stay in the same location until they are dropped or replaced.

Within the file there are four different object types:

1. Global file information block (Root)

   There is always exactly one such block, and it is located at the start of the file. This contains information about the file such as whether it is 32- or 64-bit. It also contains three lock bytes used to control file access (for the exclusive tie lock, update/access lock, and

⎕FHOLD lock) which are essential to allow collaborative access to the file in a multi-user environment.

2. Components

   The APL array within the component is written to the file along with a header which contains information such as the timestamp. An explicitly-created access matrix is stored as a component.

3. Component index blocks

   Components, with the exception of the access matrix, are linked using a tree; the components themselves are the leaves of the tree and the non-leaf nodes - the component index blocks - occupy space of their own on the disk. The tree allows components to be located by component number.

   The component addresses are maintained in a form of B-tree with 16 pointers per component index block. The tree is kept balanced and adding a single component can result in updates to more than one index block as the tree is rebalanced.

4. Free space

   Gaps within the file are free space. These spaces are themselves linked together using two AVL (balanced, binary) trees which allow free spaces to be located on size and location. However, there are no free space index blocks as there are for components - the free tree nodes occupy the start of the free space itself (which means that free spaces have a minimum size). As the AVL tree is kept balanced, changes to the tree can often result in updates to multiple free tree nodes.

   Free spaces never exist adjacent to other free spaces. When space in the file is freed, existing free space which is located before and/or after it is amalgamated to form one block. There is always a free block at the end of the file, which defines the remaining free space up to the file size limit set by ⎕FRESIZE. The actual file size, however, is rarely this large but grows towards this limit as needed.

   Space within the file is allocated by searching for the best-sized available block. If space of exactly the right length exists it is used, otherwise a longer space is split and the residue returned to free space. Because the residue has a minimum allowable length a free space which is split must be at least as long as the required space plus this minimum; the shortest available space which meets this requirement is chosen.

Each APL file update involves updates to multiple parts of the native file and, for efficiency, component index blocks and free tree nodes are cached in memory. With share-tied files the dirty cache entries must be flushed and all entries discarded after each file function completes (because other users may access and amend the file between updates). With exclusively tied files the dirty cache entries do not need to be flushed and all entries are retained between updates. Dirty cache entries are always flushed when the interpreter is waiting for user input. When the cache capacity is reached, entries are flushed to make room, oldest first.

File layout examples

These examples illustrate how data may be laid out in a file following various file updates. Actual file offsets are largely unimportant for the examples, and anyway differ for different file configurations.

Example component file layout following initial □FCREATE:

File offset	Content
0	Root
72	Free space

File component layout following □FAPPEND of □AV twice:

File offset	Content
0	Root
72	Component 1
620	Component index block
748	Component 2
1296	Free space

File component layout following □FREPLACE of component 1 with the value 123:

File offset	Content
0	Root
72	Free space, previously occupied by component 1
620	Component index block
748	Component 2
1296	Component 1
1344	Free space

File component layout following □FREPLACE of component 2 with 123:

File offset	Content
0	Root
72	Component 2, occupying some of the previously free space at this location
120	Free space, the remainder of the previously free space
620	Component index block
748	Free space, previously occupied by component 2
1296	Component 1
1344	Free space

File damage

Whilst the update process documented so far is efficient, it is susceptible to allowing file damage. If, for any reason, an APL file update (which appears atomic to the user but which results in multiple native file updates) is not completed in its entirety then the file can be left in an inconsistent and damaged state. This kind of damage is more likely with exclusively tied files, because updates remain in cache for longer. Broken index blocks in the file render it totally inaccessible, even if the data itself is fully intact.

Dyalog has applied a number of techniques to make this existing file structure both robust and repairable.

Journaling

Journaling, first introduced in release 12.0, protects the file from damage caused by abnormal APL termination. It does this by performing a file update in distinct stages:

A. Write only to the unused parts of the file. That is, all writes to the file which would overwrite any part of it which is "in use" (the root, component index blocks and free tree nodes at the start of free blocks) are deferred.

B. Write a journal to the file which describes all deferred updates. The journal is located after the free tree nodes in the unused space at the end of the file.

C. Write the location of the journal to the root, making the journal "active".

D. Complete the deferred writes.

E. Remove the journal address from the root, making the journal "inactive" and effectively removing it (because it was in unused space, it is not necessary to actually delete the journal).

In the event of abnormal termination of APL, either:

1. No file update was in progress at the time of termination, or

2. Updates had been made only to previously unused parts of the file (stages A and B, above), or

3. Updates had been made to in-use parts of the file (stages C, D and E above), and the journal allows the update to be completed.

Whenever a journaled component file is tied, it is checked to see if it contains an outstanding journal. If it does (case 3) the updates described in it are (re)performed and the journal is removed so that the interrupted update is completed and any damage in the file is repaired. If there is no journal (cases 1 and 2) then no action is taken – if an update had been in progress (case 2) then the update is effectively rolled-back, but no actual roll-back process takes place; it is not needed due to the design of the update procedure.

Journaling has a largely negligible impact on performance. There is no impact on reading files.

Note that journaling affects only the way that files are updated; there is no difference in the file structure itself (except that the root is larger to contain the journal pointer, and that the journal exists on disk for the duration of a file update). Even though the structure is largely unchanged, Versions of Dyalog APL prior to 12.0 are prevented from accessing journaled files because they do not have the ability to honour the journaling requirement or handle an outstanding journal in the file.

Checksumming

Checksumming, first introduced in release 12.1, enables components in damaged component files to be recovered. (It is also required for OS crash-proof journaling, detailed in the following section). A component file with damaged index trees is rendered unusable but checksumming allows the components to be located and validated so that the indices can be rebuilt. When a component file is checksummed, each component contains additional meta-data in a trailer attached to it containing not just the checksum but also the component number and update sequence number, magic numbers etc., and some duplication of the component header so that deleted components may be recovered from free space. This information is used by the repair tool, `⎕FCHK`.

Unlike journaling, checksumming changes the structure of a component file because of the additional data attached to each component within it. Enabling checksumming on an existing component file requires that each component be updated. Once enabled, checksumming adds a small overhead to each file update due to the additional computation requirements and writing the additional trailer

File validation and repair using checksums

`⎕FCHK` has two distinct modes of operation: it can validate a component file to check that it is in an undamaged state and it can rebuild damaged component files. When it rebuilds it always performs a check afterwards and the returned value of the function is the result of the check.

For validation, it builds a list of everything in the file – root, components, component index blocks and free space nodes – and checks that every word of the file is properly accounted for. For free spaces there are two trees to traverse and it checks that both trees cover exactly the same spaces. It checks that all components that are supposed to be present are indeed present (and none are present that should not be).

For repair, `⎕FCHK` finds all the components in the file, then builds a new free space tree in the gaps between them, then builds a new component tree which populates some or all of the free spaces.

The process of locating components is as follows:

- Scan the file looking for the potential starts of a component trailer. This is identified by a magic number.

- Read the checksum and length of the component from the presumed trailer, then validate the data in the file matches the checksum. If it does then an undeleted component has been found (the component number is in trailer), otherwise:

- Deleted components will have been overwritten at their start by the free tree nodes written when the space was freed. The trailer, however, contains a duplicate of the start of a component. The data in the file is validated again, but this time assuming the duplicate in the trailer was also at the start of the component. If the checksum matches this time then a deleted component has been found, otherwise:

- Either a damaged and unrecoverable component has been found or the magic number was just random data in the file – either way nothing usable has been found.

Once the entire file has been scanned there will be a list of all found components with zero, one or more candidate for each expected component. Where more than one candidate component exists it is necessary to select the best – whether the component had been deleted, and sequence numbers in the trailer are used to determine this – and the remainder are discarded. Any deleted components which are to be recovered are "undeleted" by fixing the part overwritten by the free tree nodes with the copy in the trailer.

Once the location of all recoverable components is known the gaps between them are recorded as free by rebuilding the free trees. As previously noted, these gaps have a minimum size and if a gap is too small a component will be copied elsewhere in the file, otherwise the recovered components are left where they were found.

Once the free trees are rebuilt, the component tree is rebuilt in the now-accessible free spaces. If any expected component is not present a replacement is generated; this replacement is one which will give a COMPONENT DAMAGED error if read.

O/S crash-proof journaling

The journaling discussed so far depends on data being written to file in a particular sequence. However, the operating system will perform its own file caching and will not necessarily write data to the physical disk in the same order the APL interpreter presented it. This causes problems if the operating system terminates abnormally (e.g. due to a power failure) because the data still in cache will not be committed to disk, but data generated by APL later in its update sequence may have been.

There are four boundary points in the journaled update sequence where this can cause problems. For the five stages, A-E, described previously these occur at end of stages B, C, D and E:

1. If stage C (writing of journal address to the root) occurs but stages A and B (writing the component data and journal) are not complete on disk, either data will be missing which the journal cannot replace, or file recovery using the broken journal will fail, or both.

2. If stage D (performing the deferred writes) occurs in part, but stage C (writing the journal address to the root) is lost, the file will be damaged and file recovery using the journal will not take place.

3. If stage E (removing the journal address) occurs, but stage D (performing the deferred writes) is not complete, the file will be damaged and file recovery using the journal will not take place.

4. If stages A and B (both writing to unused parts of the file) occur in full or part, but stage E (removing the journal address) from the previous update is lost, the journal will be used to recover the file but by this point it may have been over-written by new data, causing unpredictable results.

The straightforward solution to this problem is to issue an `fsync()` operating system call at the end of these stages B, C, D and E to force the data from cache to disk. However this has a major performance impact and four such calls during a single APL file update would cause a significant slow-down. It is, however, possible to eliminate some of these cache flushes. For the four cases above, two `fsync()` calls may be eliminated when checksums are enabled:

1. By checksumming components and the journal, and validating the relevant component checksum and journal checksum before performing file recovery, it is not necessary to flush the caches after stages A and B. A journal is simply ignored if either of the checksums fail; the file is not at this point damaged and ignoring the journal "rolls back" the update.

2. Similarly, by checksumming the journal and validating, corruption to an old journal will be detected and file recovery will not be attempted - even if the journal address is still in the root - so it is not necessary to flush the caches after stage E. (If the journal has not been overwritten then file recovery may take place unnecessarily, but this will be harmless)

Fully O/S crash-proof journaling therefore requires that checksumming be enabled and uses two calls to `fsync()` per update. If an update is interrupted after stage C the journal will automatically ensure the file will be recovered and intact.

It is possible to eliminate a further `fsync()` from each update, but at the expense of automatic detection and repair of files. If either cache flush at the end of stages C or D is also omitted (cases 2 and 3) the file can get damaged with no journal with which to repair it and no indication that damage has occurred. However, the damage would be confined to, at most, the component index blocks, the free tree nodes and the new component so the damage is repairable and it is possible to fully recover the file.

O/S crash-proof journaling (both fully automatic recovery and repairable) was introduced in Dyalog APL in release 12.1. It requires that checksumming be enabled and additionally adds an overhead to each file update due to the one or two flushes of the O/S file caches.

If *both* of the `fsync()` calls at the end of stages C and D were omitted then there would be no forced disk flushes at all per update, which is the case with the original journaling. The journal only allows recovery of the single most recent update, but the disk cache could contain data spanning several updates, and in the event of operating system termination there may be unrecoverable data loss. The recovery tool may be used to attempt recovery, but only if checksums are enabled - and even then there may be unrecoverable components within it and/or components which can only be recovered to an earlier version. The components affected will be largely unpredictable and not necessarily dependent on update sequence – for example, the penultimate update before an operating system termination may not be unrecoverable, even though the final one may be.

Summary of Journaling and Checksumming options

Journaling /checksumming	Crash protection	Size / performance implications	Dyalog APL compatibility
No journaling or checksumming.	Potential for file damage causing total loss of access to file.	None.	All supported versions of Dyalog APL.
Journaling only.	Files safe in the event of APL termination. Potential for file damage causing total or partial loss of access to file in the event of O/S termination.	Small performance impact. Virtually no size impact.	12.0 and later.
Checksumming only.	Potential for file damage. Good data recoverable but full recovery not guaranteed.	Small performance impact. Each component larger on disk.	12.1 and later.
Journaling and Checksumming.	Files safe in the event of APL termination. Potential for file damage in the event of O/S termination - good data recoverable but full recovery not guaranteed.	Small performance impact. Each component larger on disk.	12.1 and later.
O/S crash-proof journaling (1 fsync) and checksumming.	Files safe in the event of APL termination. Potential for file damage in the event of O/S termination but *all* data recoverable.	Larger performance impact. Each component larger on disk.	12.1 and later.
O/S crash-proof journaling (2 fsyncs) and checksumming.	Files safe in the event of APL and O/S termination.	Largest performance impact. Each component larger on disk.	12.1 and later.