SharpMedia Physical Database Design

© 2007 Žiga Osolin – SharpMedia project, document version: DRAFT

Contents

[About 2](#_Toc167333678)

[Goals 2](#_Toc167333679)

[Deployment Structure 2](#_Toc167333680)

[Namespace Structure 3](#_Toc167333681)

[*Namespace* Physical 4](#_Toc167333682)

[Physical Storage 4](#_Toc167333683)

[*Namespace* Physical.Journalling 6](#_Toc167333684)

[Operation execution overview 6](#_Toc167333685)

[Simple operation execution 7](#_Toc167333686)

[*Class* Journal 7](#_Toc167333687)

[*Class* Allocator 7](#_Toc167333688)

[*Interface* IReadJournalServices 8](#_Toc167333689)

[*Interface* IJournalServices 8](#_Toc167333690)

[*Namespace* Physical.Operations 9](#_Toc167333691)

[Stream Helpers 9](#_Toc167333692)

[Balanced Tree 9](#_Toc167333693)

[Operations 10](#_Toc167333694)

[*Namespace* Physical.Caching 13](#_Toc167333695)

[Multicaching scenarios 13](#_Toc167333696)

[*Namespace* Physical.Provider 14](#_Toc167333697)

[Implementation Notes 15](#_Toc167333698)

# About

SharpMedia physical database is persistant storage implementation that is used by SharpMedia as primary data storage medium. It is based on database design interfaces design and implements journallled (recovering from invalid states) persistant storage.

# Goals

The physical database is used to hold objects. The primary goals are performance and atomicy of operations. Whenever the operation is executed on the physical database, it must be either executed fully or not executed at all. The cost of such safety must be minimized.

Database must behave well when a lot of small objects are placed to typed stream. Indexing of objects is especially important in this context. On the other hand, writing only one (big) object is also common and such case should also be optimized for speed and storage. In the later case, having a indexing structure would be a waste of space.

Parallel access to resources must be available. This implies that safety must be garantied at all stages of operation. Parallel execution may benefit from cached blocks that do not need synhronization when reading and some short operations may be executed meanwhile a long running operation is being executed.

# Deployment Structure

Physical database is deployed as SharpMedia Library because of it's size and complexity. It is not necessarily needed in all cases. It can be considered as a driver for SharpMedia Database.

# Unresolved Issues

* If (and when) typed are stored for object, a feasible solution would be: having type only for AllowDerivedTypes typed streams **and** not equal to default type.

# Namespace Structure

Library is organised into multiple layer, each layer abstracting out certain aspect of physical storage. From top down, the following layers are available, each in it's own namespace:

1. Database driver, including implementation of node, typed stream and database (**SharpMedia.Database.Physical**);
2. Journalling layer, operating with operations (**SharpMedia.Database.Physical.Journalling**)
3. Operations layer, consisting of many helper stream types and operation set (**SharpMedia.Database.Physical.Operations**);
4. Caching layer, with read/write blocks (**SharpMedia.Database.Physical.Caching**);
5. Block device providing (**SharpMedia.Database.Physical.Provider**);

Apart from that, block provider drivers are placed into seperate assemblies. Block providers must allow direct device access (e.g. no system caching).

## *Namespace* Physical

Implements driver classes, as specified by database design. Implementation should not use any caching strategies because they are handled by managed implementation. This implies that every call is translated to either operation (if changing contents) or is executed through journal that provides information.

All operations must be atomic. There is no need for operation synhronization, synhronization is implemented by managed layer and journalling.

The following classes are defined:

* **PhysicalNode : IDriverNode**
* **PhysicalTypedStream : DriverITypedStream**
* **PhysicalDatabase : IDriverDatabase**

### Physical Storage

Because physical storage is complex, we use unmanaged code to handle writing to and reading from block locations. A special set of classes is introduced to be mapped to those blocks as:

struct X

{

public ulong Property1;

public ulong Property2;

public char Property3;

}

class Y

{

unsafe void DoWork(byte[] data)

{

fixed (byte\* b = data)

{

X\* x = (X\*)b;

x->Property1 = 0;

x->Property2 = 2;

x->Property3 = 'a';

}

}

}

We define many such classes in **Physical.StorageStructs** namespace. Those structs are mapped directly to blocks, as shown in example. The following storage structures exist:

* DatabaseHeader, always at block index 0. It containts all necessary information of database, such as block size, journal location …
* NodeCommonHeader, always at start of the block, includes per-version information such as children, parent[[1]](#footnote-2), current version. Children are placed into special typed stream (automatic indexing) and are inserted in alphabetical order (insert is O(logN), remove is O(logN), find is O(logN)).
* NodeVersionHeader, always at start of the block, includes per-version information, such as name (fixed size) and typed streams headers.
* TypedStreamHeader, resides in versioning. Typed stream header contains name of type, flags and B+ tree link. There is a special case with SingleObject typed streams where no B+ tree exists, only object's full beginning address.[[2]](#footnote-3)
* BTreeNode contains information that are used for indexing. We use balanced B trees, with data residing in leaves only. If a lot of objects are placed to stream (common scenario for SharpMedia), only a few blocks are read to get the object's address. BTreeNode blocks are also given higher caching level since they may be needed many times.
* JournalLog – a log into a journal, followed by information and next log links.

## *Namespace* Physical.Journalling

This namespace implements thread-safe access to resources that reside on some physical storage. It provides means to rollback or rollforward operations that did not finish execution **at any time**. When database is mounted, all such operations are executed safely.

### Operation execution overview

Operation is created when we need write access to database, read access if free through journal class (using **IReadJournalServices**). Each operation has following properties:

* Operation type for reconstruction
* Immutable state part, the one that does not change during operation (the root node address to delete etc. – null for 1 stage operations)
* Link to first journal log.

Once operation is passed to journal to execute, following parameters must be set:

* Maximum allocation size (or blocks) needed[[3]](#footnote-4);
* Allocation near block hint (so allocations are as close as possible), use 0 if no need for that;
* Is dynamic allocation supported – this means that operation can request more memory than stated as maximum allocations.

Using these information, journal allocates storage for operation and serializes the operation's immutable part to journal specific storage (it searches this storage at startup for unfinished operations). Besides that, special mutable state blocks are allocated near operation execution's blocks.[[4]](#footnote-5) These blocks are used to write state checkins.

Than operation executes in multithreaded way. All accesses are done through **IService** interface that is specifically created for operation by journal. Operation can write to tagged or allocated blocks, read from any block or commit the mutable state (a checkin). From each state, operation must be able to at least rollback itself to state when operation was not executed (rollforward operations are also desirable).

When operation completes it's execution, journal frees data in journal section. After that, memory cleanup is also made, but if something happens, the operation is considered finished.

### *Class* Journal

A journal is created upon a certain block provider. It first finishes all pending operations (that is, reverts or fully executed them). The default action is always to finish operation if possibly (that is, enough information is available), otherwise operation is rollbacked. The default action can be changed using a **RecoveryContext** class that can be passed as one of arguments to journal (or is passed through PhysicalDevice). This allows consumization of recovery that is sometimes required.

Journal is responsible for managing operations. It restricts their usage only to blocks that are locked by them and also tracks their operations. It can also provide performance and allocation counters for operations so we can further optimize them.

### *Class* Allocator

Allocator can cache available blocks for allocation and is used as a service to journal. Whenever operations is created and applied to journal, allocator is used to find free blocks or use cached ones. Allocator only serves blocks and operations can return allocated blocks (because **actual** allocation happens when you commit the allocation).

All allocation or deallocations commits happen when calling checkout. Note that when you deallocate something, no other resource can gain such memory **until** operation finishes so when rollbacking, we can reacquire(that is re-allocate) the blocks and they will be preserved.

#### Allocation Sectors

The physical system tracks the free blocks using 2 level allocation strategy. At the beginning of database, at address 1, a superblock exists. A superblock is block managed by Allocator and can never be changed directly (it can be updated through allocator). Each bit in superblock tells if the allocation block is fully used or not.

The next level are allocation blocks. In one superblock, there are so many allocation blocks as there are bits in one block[[5]](#footnote-6). Allocation blocks keep track whether data blocks are allocated or not. As superblocks, each allocation block can address so many blocks as there are bits in one block.

#### Journal Sectors

Journal sectors must have the following properties:

* Must be close to actual action (locality) to ensure optimal speeds;
* Must not affect mount too much (since all sectors have to be checked).
* Must not allocate too much memory.

The tradeoff is used – there are N journal sector blocks right after allocation block in one super block. The journal sector block only holds links to actual logs or operation data. This means one block can hold up to BlockSize/8 links.

Those links point to blocks that are free and were tagged in **Allocator** that they must not be served anyone until operation succeeds. Journal can freely keep log, block copies and operation data in those blocks. Blocks are always near the execution of the operation.

### *Interface* IReadJournalServices

Exposes block reading services and is available as a property of a journal.

### *Interface* IJournalServices

Exposes reading, writing and commiting services.

## *Namespace* Physical.Operations

Operations are built upon journal service abstraction. Because journalling does not abstract everything away, we introduce many helper objects and operations.

### BlockStream

Stream helper **BlockStream** abstract single blocks to blocks that are linked. Following links are supported:

* Beginning of block defines a ***ulong*** link to next block. Special 0 index means no further link;
* Block sequence link is available. A sequence has a header block that contains two values: ***ulong*** next chain address and ***uint***,number of sequential blocks (greater than 1 and smaller than the maimum sequence permitted).

### Balanced Tree

Balanced tree is a helper class that let you read and write indices from B+ tree. Indexing is implemented using balanced tree. This enables fast search for:

* Objects with indices;
* Children (string hashes can be used for entry indices, with a special level of indirection – because children can share the same hash key);
* Versions (the same as children, with hashes).

Balanced tree supports adding, removing and searching. All operations can also be prepares, e.g. tell how many allocation blocks are needed to add an index etc.

### Operations

All operations are implemented here:

#### Rename

Renaming happens on parent node. We atomically remove the object from children and add it to this list once more, this time at different index (computed from name):

1. We remove the child from children typed stream, remembering it's previous name and it's common node index;
2. We add the child to the same typed stream with new name and the same index.

#### AddStream

Adding a typed stream always happens at the end of the node. This may need expansion of block stream. Adding a typed stream consists of following:

1. Create a B+ tree node on newly allocated space and initialize it (write it);
2. Commit allocation (one block update, atomic, there is no need to track it because allocation is part of immutable state), is postponed until commit by journal (only cached);
3. Append the typed stream. Because we want to limit this operation to be at least 2 blocks wide, typed stream header contains value if next stream is available and there is no »count of typed streams« available. We first write to second, allocated block, if necessary (if we go beyond first block), commit allocation.
4. Issue a checkpoint, commiting all allocations (usually atomically if within same allocation block, otherwise with help of jornal). This is a point of no rolling back (only rollforward), until now both available.
5. We update the node's data. This includes setting next typed stream to true and adding a header and possibly a link to next block, if it was necessary to create it. After this is done, we have finished the operation.

#### RemoveStream

Removing a stream is swapping the last stream in link into this place. We have two possible scenarios:

1. Typed stream is single object. This means there is no B+ tree available and we proceed the following way:
   1. We commit block deallocations for entire object and issue a checkout;
   2. We remove stream from node by removing it from list if last or swapping it, overwriting location by other stream, and then deleting the last stream. This may need a potencial backup for reconstruction (e.g. one commit);
2. Typed stream is multi object and contains B+ tree.
   1. We first delete all blocks containing objects;
   2. We delete a B+ tree by deallocating blocks used by B+ tree. We issue a commit here;
   3. We remove the typed stream by the same procedure as stated by 1.b.

#### ChangeDefaultStream

Changing default stream requires two updates, one in current default typed stream and one in the typed stream that is becoming the default one. There are again two scenarios:

1. Both streams reside in the same block (at least their headers, where flags reside). This makes this operation a *simple* operation, because only one block update is issued;
2. The other type is when streams reside in different blocks. This makes the operation non-simple because we need to backup some data:
   1. We write both TypedStreamHeaders with our immutable state, including their offsets in buffer;
   2. We update both blocks accordingly;
   3. We finish operation.

#### CreateNewVersion

Creating a **empty** new version is similiar to creating a children. Operations execute in the following order:

1. We create new node header and empty default stream (B+ tree or just a header inside node's block for single object) and commit all those allocations, then issue a checkout of operation;
2. We relink the common node header to new version and finish the operation;

#### CreateChild

Creating a child is non-trivial operation. It is similiar to creating a new version. The operation executes as:

1. Create node common header, node header and default typed stream, fill the data and commit allocations, then do a checkout;
2. Insert a child in typed stream of children on precomputed location (must be ordered for fast searches) and issue a checkout.

#### DeleteChild

Deleting a child **can** result in long going operations. Deleting a child can essentially delete many thousand blocks.

Delete child is a multi-stage operation. In each stage, one child is deleted.

#### WriteObject

Writing an object is always atomic. It consists of following operations:

1. Writing an object into allocated blocks. Object can be packed with other objects into space that is not yet used. This is still perfectly safe operation since updating a block will perserve all contents of previous objects. All blocks allocations must also be commited. We issue a checkout;
2. We add the object's beginning offset in B+ tree under it's current index. If index is already used, we also free the object at that place;

#### WriteObjects

Writing several objects is always atomic. It is similiar to write object operation:

1. Write **all** objects into allocated blocks and issue a checkout;
2. Update B+ tree and remove all objects we are overwriting.

Writing several object at once can be faster than writing each object seperately because less block updates can be issued (especially in B+ tree nodes).

#### InsertObject

Inserting an object is similiar to writing. All it needs is that B+ tree's indices after object's index are incremented by one, which requires more states for operation.

## *Namespace* Physical.Caching

Caching is implemented using **SharpMedia.Caching**. Blocks are cached based on priority specified when read or written to (highest priority is always used). Blocks implement copy on write scenario.

### Multicaching scenarios

Caching is implemented on several levels. The following caches exist in top to bottom order:

1. Fast cache, includes currently used blocks, updated most often. It has at most 128 blocks (65kB on 512 bytes block size) and hopefully resides in CPU cache. This cache is LRU and does not depend on privilegies;
2. Common data cache, includes only high priority blocks and is LRU. Here, allocation blocks and super blocks are cached. It's size is up to a few MBs.
3. Medium priority cache, contain data such as node header blocks, typed stream headers and B+ trees for fast access. It's size is up to a few MBs.
4. Low priority cache, contains infrequently used data. You can remove such cache if you want. It's size is configurable and can range from 0 to many megabytes. Object data is cached here (especially small objects).

## *Namespace* Physical.Provider

Provider can read and write block as an atomic operation. Because providers are OS dependant, this is essentially a driver architecture. The only interface the driver must provide is the **IProvider**. This provider only implements reading and writing, all as atomic (immediate) operations.

# Implementation Notes

The following stages of development should be taken:

1. Create data containers;
2. Create provider;
3. Create simple (pass through) cache;
4. Create B+ tree implementation;
5. Create journal and allocator;
6. Create DB initialization;
7. Create some simple operations and fully test them;
8. Add more operations;
9. Add cache with multiple levels and good caching;
10. Optimize;
11. Implement all missing features and clear all bugs.

A lot of unsafe code will be used in this component. Try to minimize it's usage whenever applicable. Also, here are some C# features that are interesting:

* StructLayout attribute to control alignment of structures;
* Static arrays (as C arrays) available for unsafe code, but only for built-in types.

1. Just parent's common block address. [↑](#footnote-ref-2)
2. Minimum number of blocks is 0 for empty SingleObject stream (because it is packed after node header) and 1 for multi-object stream with empty B+ tree. [↑](#footnote-ref-3)
3. Could be more than actually needed, unused blocks are discarded if allocation is not confirmed. [↑](#footnote-ref-4)
4. This makes the write access faster (data is closer to each other). [↑](#footnote-ref-5)
5. Size of block is usually 512 bytes, that yields 512\*8 allocation blocks, or 4096. [↑](#footnote-ref-6)