Frequently Asked Questions

Table of contents

Questions

1. General

1.1. How to use these FAQs?

FAQs will be piled up here. Some of the answers here have been (manually) lifted directly from javadoc comments in the code. So there's some duplication here, and hopefully the 2 wont fall terribly out of synch.

These FAQs are not all that structured: use your browser's "Find in this page" facility to search for keywords.

1.2. What's the idea behind Skwish?

The project was born of a need to efficiently store lots of files, big and small, that were to be indexed using the popular Lucene library. These files have to be stored *somewhere*. Usually that "somewhere" ends up being a deep, heavily populated directory structure on the file system. This directory structure approach, however, doesn't usually scale well, and it becomes especially annoying when the blobs (files) you're storing have no inherent hierarchical structure. Worse, when those blobs (files) are small and many, and access is heavy, all that file I/O begins to take a toll.

A well-known, simple solution to this, is to throw all the blobs in a same file and maintain their offset boundaries in another file. That way, you keep only a few files open, and you let the file system and the underlying device controller do their work, e.g. paging data in and out of a block device. And if you insert related blobs in sequence, you get better locality of reference and hence less paging.

So this blob storage thing is a recurring problem. And the Skwish project began as an attempt at a clean, reusable implementation of the above described solution.

Skwish is premised on the idea that the problem of blob storage is orthorgonal to the problem of indexing.

1.3. What usage scenarios is Skwish designed to address?

Here are some examples:

The application manages a large number of records but does not use a full blown (e.g. relational) database implementation.

The application uses a database implementation, but needs to store BLOBs separately. This requirement may arise for performance reasons, or where the contents of the database is determined from a set of *source* entries (records) but which are to be maintained separately.

The application uses an indexing library (such as Lucene) that does not directly support storing the contents of what is being indexed.

The application needs all-or-nothing transactional semantics. For example, an application may need to commit a set of entries either *en bloc* or not at all.

More generally, if you find your application needs to maintain both offset and content files, where the offset file delineates the boundaries of entries within the content file, then this library hopefully provides a ready-made solution for that part of the problem.

2. Implementation Details

2.1. Why doesn't Skwish support updating an already inserted entry?

Short answer: the unmanaged version `BaseSegment` already supports this feature, but it doesn't allow changing the entry's size. Updates within transactions are trickier to implement. This will be addressed in a future version.

Longer answer: though the machinery for a bounded update model (where you can update the contents of an entry, but not its size) is already in place, this feature was put on hold until the code implementing transactions was worked out. (The transaction design is premised on append-only property of entry insertion, and the idempotence of entry deletion.)

10-25-07 Addendum: As of version 0.1.2, the transaction segment supports updating uncommitted, new entries. Eventually, we'll get to updating *existing* entries. It'll probably involve more thought than code as we consider and mitigate the attendant race conditions this feature introduces.

2.2. How big can a Skwish segment grow?

Skwish segments are characterized by their index's offset word width property, which ranges from 1 to 8 bytes. The maximum total byte size of the entry contents of any segment is governed by that word width. (As such, the 1 and 2 byte word options are purely pedagogical: they serve only the test cases.) The high bit of an offset word is used as a marker for deletion. So a typical 4 byte word segment, thus, can hold about 2GB of content; an 8 byte word segment can hold almost 1018 bytes.

2.3. What's the file format?

The file format is specified in the API documentation. It's available here.