Subversion

Compressed Pristines Design

History

Rev.	Notes	Who	When
0.1	Initial public version. Contains reqs, design and plan.	Ashod Nakashian	21-Mar-2012
0.2	Updated based on comments from Greg Stein.	Ashod Nakashian / Greg Stein	22-Mar-2012

Contents

Discussions on dev-list

Overview

Subversion stores the original files (called Pristine) checked out to a working copy in a local store. Pristine files therefore double the storage requirement of the working copy. Since pristine files are only used when executing certain Subversion operations, there is a large potential for storage savings by compressing pristine files.

This feature will add built-in support to Subversion to compress pristine copies of files in a working copy transparently and efficiently.

Goals

1. Reduce disk-space overhead due to pristine copies.
2. Abstract implementation details via public API.
3. Maintain current core WC commands.
4. Maintain performance where possible.

Non-Goals

- Reduce disk space of repository files.
- Improve network/remote access protocol.
- Remove pristine store or render optional.
- Improve SVN commands' performance in WC.
- Fix outstanding bugs and/or limitations of SVN client commands.

History

This features has been reported as a bug #908[1] and has been referred to as "Compressed text-base" in the past. It was the subject of numerous discussions on the SVN dev-list at least since late 2002. There has been a few patches to compress pristine stores. While there was notable support for reducing the overhead of pristine files, the preference has been towards simplicity at the expense of affordable storage.
TBD: Add some of the discussion and design decision background.

Keywords

SVN - Subversion tools and services.
WC - Working Copy - A directory tree checked out from an SVN repository.
PS - Pristine Store - Pristine copies of files in a working copy.
CP - Compressed Pristines.

[1] http://subversion.tigris.org/issues/show_bug.cgi?id=908

WCCP - Working Copy with Compressed Pristines.
WCUP - Working Copy with Uncompressed (legacy) Pristines.
EC - Entropy Coder / Compressor - A lossless compression transformer.
OS - Operating System - The software managing user applications and hardware.
FP - Fingerprint - A cryptographically-secure hash value.
FS - File System - Either OS-level or virtual.
VCS - Version Control System.

Requirements

1. Efficient and (near) real-time compression of pristine files (compression).
2. Mimetype-aware compression (efficiency).
3. Full support for SVN client commands (transparency).
4. Detection of corrupted CP files (robust).
5. Recovery from interrupted operations where possible (self-healing).
6. Provision to change the CP file format (versioning).
7. Provision to use different compression algorithms in the future (typing).

Concerns

Here are some identified high-risk areas.

1. Large file support in terms of compression library limits and performance.
2. Tools that assume direct access to the pristine files.
3. CP corruption and recovery.
4. Performance when updating pristine files.
5. Locking.

Design Overview

The main goals of this feature is to reduce disk storage overhead of pristine files while maintaining an abstraction via the SVN WC API[2].

There are two categorical approaches to realizing CP support. One is to compress individual files and the other to combine all PS files into an archive and compress the resulting file.

While compressing individual pristine files will yield some disk-space saving, in addition to being a good starting-point for implementing this feature, due to block granularity of file-systems and the tendency of source file to be small in size, there will be limited savings. Source files that are smaller or equal to a single FS block will yield no space savings when compressed as the FS will still allocate at least one block for them and round the allocated size towards the next block size. In addition, compressing such small files will result in pure overhead due to the compression and decompression processing and there won't be much reduction in disk read

[2] http://svn.apache.org/repos/asf/subversion/trunk/subversion/libsvn_wc/

duration precisely because an FS reads in complete blocks. The wasted space due to this rounding-up is called internal fragmentation. A typical FS block size is 4096 bytes.

Combining the PS files without compressing them may yield significant storage savings by avoiding internal fragmentation. This becomes more significant as the number of files within a PS grows and the average file size drops. There are patches[3] to compress individual pristine files in SVN. As such, this feature is designed to be implemented as a combined archive, called a Pack.

Packing

An obvious solution to the internal fragmentation issue is to combine the pristine files together into an archive called a pack. Such a pack isn't unlike general purpose archives such as *tar* or *zip*. Packing PS files will resolve the disk-space waste due to the elimination of internal fragmentation and will potentially allow for both improved compression and read performances. However, this will come at the cost of complexity.

Due to the nature of VCS operations, files are updated and modified as a matter of routine. Unlike general purpose compression archives where files are compressed and archived once and either distributed or shelved, SVN needs to update the files in a PS on-demand and without -significant- loss of performance. For this reason general purpose archiving libraries aren't a viable solution and SVN needs to tailor a custom packing format and scheme to match its requirements.

1. High-performance archive with add/modify/delete support.
2. Low wasted space due to modifications. (Low external fragmentation).
3. Highly portable. (APR is the ideal reference here.)
4. Supports compression by design.
5. SVN-compatible license.

There doesn't seem to be a ready library that can satisfy these requirements. Therefore, such a library must be created as part of this feature. Looking through VCS solutions that expectedly should have similar requirements, one VCS stands out: Git[4]. The Git Pack file format[5] has a remarkable simplicity and demonstrable feasibility as it's been deployed on huge projects. Unfortunately, the Git Pack file has but only a subset of the requirements of SVN. Most notably it lacks the very first requirement. All files within a Git Pack are cumulative and no file is modified after becoming part of it. This is due to the fact that Git stores the complete repository with all history[6], which is unlike SVN which only stores the checked-out pristines. Due to lack of modification support, the second requirement is also unavailable and as a result both design and implementation of the Git Pack files are remarkably simple. These limitations and differences aside, an effort is made to leverage any opportunities for learning from this working

[3] http://subversion.tigris.org/nonav/issues/showattachment.cgi/1167/compressed-pristines.diff

[4] http://git-scm.com/

[5] http://book.git-scm.com/7_the_packfile.html

[6] Git supports shallow checkout where the repository history is truncated at a user-defined point in the past. However , such a WC doesn't allow committing and lacks some other functionality. It's for browsing purposes.

example. Beyond that, no attempt is made to be compatible or similar to Git Pack files.

The major issue with packing pristines as required by SVN is the complexity of updating a WC. Modified files will have to be removed from the pack and replaced with their newer version. If compression is done across multiple files in a pack (or worse if across the complete pack,) then the whole pack must be re-compressed, resulting in huge latency in the update operation. On the other hand, if compression is performed on individual file level then no compression improvements can be exploited by capitalizing on similarities between files in a given WC. Another complication has to do with updating files that grew in size such that the new version can't fit within the space occupied by their former version (which is now vacant).

The design choices, therefore, are such that SVN WC operations are complete swiftly.

Compression

There are many lossless compression libraries that are candidates for this feature. However, for simplicity, this feature will first use Zlib and after completing the feature an experimental stage is planned to evaluate potential replacements to the Zlib that may result in significantly better results.

De-duplication

Many repositories duplicate some of their files verbatim. This is typically done out of necessity. If no files are duplicated directly, branching and/or tagging will ensure a high similarity between the files (assuming the user has checked-out tagged or branched folders). It is relatively easy to remove identical files thanks to the file FP hashes. Since SVN WC-NG already stores pristine files using SHA-1 hashes, the CP version will also work in the same way.

Solidifying

Small files when compressed don't yield much space reduction due to the lack of context and history for the EC to work from. By combining multiple files into a single stream, called solid data block, the EC is fed much more data and therefore the compression ratio is improved. See *Sorting* for more details.

One drawback to this approach is that extracting any single file forces the extraction of the complete solid block. This can result in inflated decompression times even for the smallest of files. A remedy is to limit the solid block size to fixed-size chunks. See *Chunking* for more details.

Sorting

Due to the similarity of the data entropy of same-type files a potential for improved compression rises. Before solidifying the files sorting may be performed such that similar files are grouped together. Since file have mimetype information in SVN, first they may be sorted by their mimetype, then by their extensions. This will group text-based files together, which binary files will also get grouped based on their group similarities.

A more thorough approach would be to examine each file and group the files based on their histograms. The overhead of this scheme is unfortunately prohibitively high for the purposes of

this feature.

Chunking

To avoid penalizing all files within a PS by solidifying files, the solid data blocks may be limited in size. The block size is chosen such that the extraction of at most two chunks is sufficiently fast to be near real-time while still benefiting the compression by solidifying small files. A reasonable range could be 64KB to 1024KB depending on the performance of the EC.

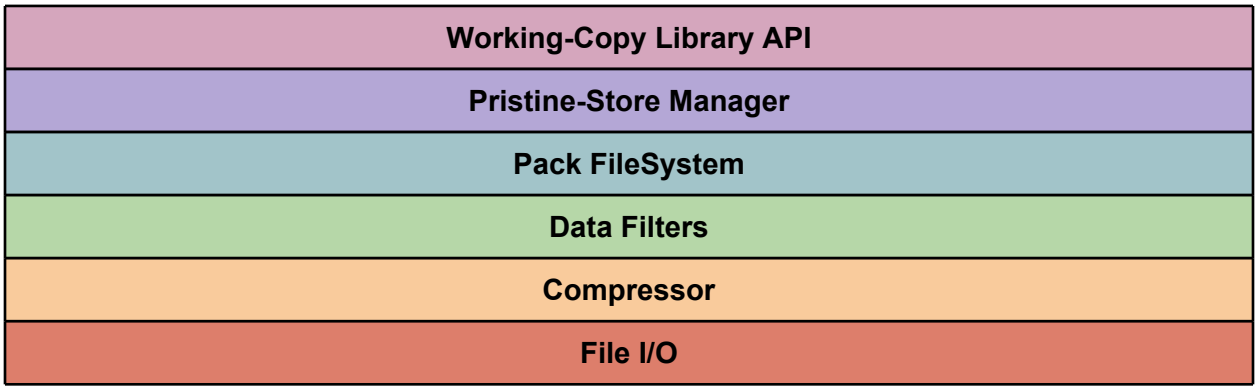
Preprocessing Data Filters

To further improve the compression of the EC, the data may massaged by preprocessor filters that transform the data bytes, in a reversible fashion, into a form readily more compressible. This stage is typically very simple and fast and yields decent compression gains.

Summary

The proposed design will combine pristine files into an archive and compress them using an EC. Each archive file will be subdivided into chunks of compressed blocks (CB) that may include one or more pristine files. These solidified chunks of pristine files will then be compressed. The files combined into CBs will be chosen based on their types which is used to sort them. Each CB is then passed via filters before finally being passed through an EC and written on disk.

Note: The methods employed in this feature as well as the code may be used to support compressed repository files on an SVN server. However, such a feature is outside the scope of this document.



The layers of the WC library after adding this feature.

The Pack File System

Since a pack file will store multiple files in it, supporting additions, modifications and deletions, it must by definition be some form of a file system not unlike a real FS atop a physical disk. File systems need to solve two interrelated problems. First, the layout of the storage must be designed, called Pack Store Files, and second a file allocation table (FAT) must be tailored to support the particular layout and operations such that it'd fulfill the requirements of the FS, called Pack Index Files.

Pack Store Files

To pack files efficiently into a single file they can be simply appended one after another. This method, while simple, isn't reasonable considering file modification and removal operations. To efficiently modify files, which may grow or shrink, and to remove them, which is equivalent to modification to occupy zero bytes, a structured layout is necessary. Two main approaches are considered and a proposed solution follows:

Solution 1: Fixed-Size Format (Block-based)

Pack files may be internally divided and managed as fixed-size blocks. This isn't unlike most real FS formats where the disk is pre-formatted to a fixed-sized blocks. Files are allocated based on blocks and blocks assigned to a given file are maintained by a linked-list where each block points to the next block index in the chain.

The main advantage of this approach is simplicity. First, the allocation algorithm becomes very simple and seeking within a pack becomes trivial. Files that don't fit within a free run of blocks are split and new blocks are allocated where available.

The disadvantages however are that this approach suffers internal fragmentation as well a as suffering file fragmentation. One solution to the problem of internal fragmentation is to use smaller block size, which would increase overhead of allocation management and only minimize the internal fragmentation but not resolve it completely. A solution to the file fragmentation problem, however, is more costly and involves defragmenting the pack store files.

Solution 2: Variable-Size Format (File-based)

To avoid the disadvantages of the fixed-size format a variable-size approach may be utilized. In this scheme each file is allocated a contiguous range of bytes within a pack. This solution avoids both internal and file fragmentation issues found in the fixed-size solution. However to find free regions within a pack that fits the given file free-regions must be tracked and managed by the pack file-format. In addition, unused free-regions that don't fit any files may accumulate resulting in external fragmentation which is a waste of space. The solution to external fragmentation is to defragment the packs which is costly.

Hybrid Solution: Variable-Size Format with Splitting

The purpose of packing files together is to avoid wasted space due to FS block granularity which results in internal fragmentation, which is especially significant for small files (after compression). This resulted in the major drawback of updating files within the pack, which is a matter of routine for SVN. The resulting overhead in potential internal and external fragmentation, defragmentation and housekeeping are significant indeed. So much so that with the most complex of algorithms it might still defeat the purpose in many cases.

The proposed solution is to split pack files based on a configurable minimum size. When a pack file reaches a certain size newer files are not added to it. Instead, another pack file is used and new files are stored there. This scheme avoids the overhead of the fixed-size allocation format and solves the external fragmentation issue of the variable-size format by reducing the pack store file's size that may develop external fragmentation and need repacking or defragmentation.

This scheme has other advantages: files that are rarely modified will not be touched or moved while files that get much activity will tend to be in a separate pack that is modified and maintained often. Another advantage is that the pack files will not potentially get too much fragmented on the real FS and decrease SVN's performance. Combining splitting with variable-size allocation should give a balanced performance.

Note: A large split-size may be used to practically disable pack file splitting.

Pristine files will never be broken across pack files even if the pack file exceeds the split limit. This is to keep the Pack FS simple and avoid excessive overhead. Also, files that are larger than the minimum pack size will reside in separate pack files.

Pack Store File Structure

Each Pack Store file will have a fixed-size header and one or more Compressed Block (CB) entries. Each CB entry will have the following structure.

<table>
<tr><td>Signature (0x7034714e)</td><td>4-bytes</td></tr>
<tr><td>Format Version (0x01)</td><td>2-bytes</td></tr>
<tr><td>Generator Version
((SVN_VER_MAJOR << 8) | SVN_VER_MINOR)</td><td>2-bytes</td></tr>
<tr><td>Reserved-1 (0x00)</td><td>4-bytes</td></tr>
<tr><td>Reserved-2 (0x00)</td><td>4-bytes</td></tr>
<tr><td>Compressed Block 1
<table><tr><td>Data Start Marker (0x75555557)</td><td>4-bytes</td></tr><tr><td>Data</td><td>variable</td></tr><tr><td>Data Fingerprint (MD5)</td><td>16-bytes</td></tr></table></td><td>variable</td></tr>
<tr><td>Compressed Block 2</td><td></td></tr>
<tr><td>...</td><td></td></tr>
<tr><td>Compressed Block N</td><td></td></tr>
</table>

Note that a CB block doesn't imply a single pristine file's data. It's an opaque container of data.

TBD: Should we include original file-size and compressed data-size here?

Compressed Blocks

Pack files may be split based on a minimum size and at Compression Block boundaries. Each pack file will grow to at least the PACK_FILE_MIN_SIZE_BYTES before it's closed and a new pack file created. As a corollary, pristine files larger than this minimum value will never share a pack file, nor a CB for that matter, with any other pristine files.

Within each pack file data is written in Compression Blocks (CB). Each CB grows to at least COMPRESSED_BLOCK_MIN_SIZE_BYTES rounded up to the next file boundary.

TBD: MORE ON CB.

Pack Store File Generation

Each pack file will have a unique ID. The ID of a pack file will be coded in its name in hexadecimal form. The ID encoded in hexadecimal shall be in the name with the extension of the file as PRISTINE_STORAGE_EXT. This is in line with the current approach of naming the files with their SHA-1 hash in hexadecimal and the extension PRISTINE_STORAGE_EXT. For simplicity, all pack files will have an equal number of characters in their names.

This unique ID of a pack file will be used in the Pack Index to reference pack store files. For simplicity, pack indexes will be assumed to be 1 to 65535. With each pack file limited to 16MB, this allows for a total of 1TB of compressed data. For future expansion, larger pack files may be used instead of increasing the upper bound of the pack indexes to avoid creating a very large number of files in the PS and potentially burdening the OS. Since in the first version of this specification a maximum pack file index of 65535 (0xFFFF) is provisioned, exactly four characters will be used.

Note: The average pack store file size and the maximum number of pack store files are subject to tuning based on actual benchmark data. However, in practice, SVN will not, must not, fail if the PS size exceeds the assumed limit or size. This is a point subject to extensive testing, especially to make sure that besides not breaking, the performance shouldn't noticeably degrade.

Sample pack file-name: **00A3.svn-base**

Note: We may utilize a scheme similar to the current one used on SHA-1 names where the first two characters of the filename is used to name folders where the files are distributed. This will allow for a larger number of pack files without cluttering the PS folder and maintaining a relatively small pack file sizes. However this will probably be unnecessary as even an 8MB limit per pack file will allow for a PS of 500GB, which is gives reasonable headroom. Working with larger pack files shouldn't prove expensive either, especially in due course when hardware will invariably be more powerful.

0-byte files are not stored in pack files, rather their indexes reference the nonexistent pack file with ID 0.

Pack Index Files

To swiftly find files within a pack, a file-system structure must be maintained. This structure will comprise of a number of tables and relevant information about each file with the pack file(s). To

simplify working with this structure a separate file will be used. This file shall be call Pack Index.

The Pack Index will have an entry for each packed file. The entries will contain the following information:

- The SHA-1 of the original uncompressed file.
- The original uncompressed size of the file.
- The index of the pack file where it resides.
- The offset within the pack file where the compressed block starts.
- The offset within the uncompressed block where the file starts.

Pack Index Format

<table>
<tr><td>Signature (0x70493878)</td><td>4-bytes</td></tr>
<tr><td>Format Version (0x01)</td><td>2-bytes</td></tr>
<tr><td>Generator Version
((SVN_VER_MAJOR << 8) | SVN_VER_MINOR)</td><td>2-bytes</td></tr>
<tr><td>Reserved-1</td><td>4-bytes</td></tr>
<tr><td>Reserved-2</td><td>4-bytes</td></tr>
<tr><td>Entry 1
<table><tr><td>SHA-1 hash of the original/uncompressed file.
(A.K.A. Filename)</td><td>20-bytes</td></tr><tr><td>Original/uncompressed file size.</td><td>5-bytes</td></tr><tr><td>Pack file index.</td><td>2-bytes</td></tr><tr><td>CB offset within Pack file.</td><td>4-bytes</td></tr><tr><td>File offset within CB.</td><td>4-bytes</td></tr></table></td><td>35-bytes</td></tr>
<tr><td>Entry 2</td><td>35-bytes</td></tr>
<tr><td>...</td><td>...</td></tr>
<tr><td>Entry N</td><td>35-bytes</td></tr>
</table>

The Pack Index Entries are made constant-size, as opposed to more compact variable-size, to make updates easy. Since each entry represents a single pristine file, updating a pristine file will involve seeking and writing exactly a single entry into the index file. This is both simple and fast and doesn't require potential expensive re-writes of index files.

TBD: Consider adding a short list of available pack IDs to avoid enumerating all pack files every time we need to figure out the next available pack ID.

TBD: Consider adding a sorted index of SHA-1 values to quickly find the entry by SHA-1 without constructing this map every time.

Corruption and recovery

It is best that corruption and recovery are both automatically detected and recovered with minimum user intervention, where possible. Ideally the user will not be involved in such a process and only when SVN panics should it ask for the user's manual intervention. The CP design is to facilitate automatic corruption detection upon which the PS manager will attempt to recover automatically to the best of its abilities. Only where it's not sure, or where there is potential data loss that it doesn't act on behalf of the user and instead involves the user by aborting recovery attempts and informing the user of the status.

Pack files have format and generator version numbers. The first is to document the structures of the files, to be forward compatible with future SVN releases that will need to be compatible with older files and to support upgrading from older formats to newer. The generator version number is a provision against potential bugs in SVN. By documenting the SVN version used to generate the files, bugs found in the wild are easier to handle as detecting affected files is simplified.

Pack stores include the compressed data FP hash to facilitate corruption detection. Recovery in such a case is possible by checking out affected pristine files.

SVN status

To perform a quick check on files to detect modification, the *status* command first compares the file size and last-modified date between the pristine and working copies. PackFS must therefore support querying these values to support this behavior.

TBD: Add support for fast SVN status.

Entropy Compression

The compressor library will only handle fixed-size data blocks managed by the Pack FileSystem. There are many contenders that satisfy the requirements (see below) of this feature. Once the core functionality of this feature is implemented using Zlib, other candidate EC libraries and algorithms will be tested in a bid to find the best choice.

Compressor Library

The current Subversion mainline has Zlib compression support[7]. As such, the initial and primary target of implementing this feature will be to utilize the available stream compression support available. This will make the implementation simpler and get us to a major functional milestone

[7] http://svn.apache.org/repos/asf/subversion/trunk/subversion/libsvn_subr/stream.c

faster. However, due to the severe limitations of Zlib (primarily its dated algorithm and 32KB window limitation) other libraries will be explored, researched and benchmarked. Implementing this feature around Zlib, however, will make such researching much more readily accessible. This section will address potential entropy-compression algorithms libraries, including Zlib, comparing and evaluating them for the purposes of this feature.

The compression back-end must be a relatively-fast entropy compressor. The entropy coder will be the lowest level of compression. Data ready for compression may optionally pass from any number of filters to massage and prepare the data for better compression. Such filters will probably not be utilized until this feature's main goals are achieved. Of the widely-used filters are delta-coders and run-length coders.

The entropy compressor must have a number of features[8] to be a candidate for this feature.

1. Lossless compressor.
2. Relatively fast performance.
3. Relatively low memory footprint.
4. In-memory compression/decompression (shouldn't touch files).
5. Compatible license.
6. Portable and functional code on Windows/Mac/Linux.
7. Patent-free algorithm(s).
8. Security-aware and protected (how to evaluate?).
9. Configurable and flexible algorithm and implementation (plus).
10. Self-contained (plus).
11. 64-bit and Multi-core support (plus).

Performance Targets

These are some tentative initial performance targets to aim for. These numbers will be revised and ultimately decided based on benchmarking actual functional implementation. These numbers are chosen based on current hardware and compression algorithm technologies.

Category	Target
Minimum Compression Rate	20 MB/s
Minimum Decompression Rate	50 MB/s
Minimum Compression Ratio	50% (soft limit), 30% (hard limit)
Maximum Memory Requirement	8 MB

Exceptions will be considered when there is significant improvement in another category for a given library. For example if may be acceptable to allow a slightly slower compression rate for a

[8] Some of these were enumerated on the dev-list: http://svn.haxx.se/dev/archive-2004-03/0664.shtml and http://svn.haxx.se/dev/archive-2004-03/0664.shtml

significant compression ratio gain, in relative terms to other candidate libraries.

Candidate Entropy-Compression Libraries

Library	Algorithm	Typical Compression Ratio	Typical Compression Time Factor	Typical Decompression Time Factor
Snappy[9]	?	1:1.3 - 1:1.5	0.1x	0.1x
Zlib[10]	Deflate	1:2 - 1:3	1x	1x
BZip2[11]	BTW	1:4 - 1:5	6-8x	10-13x
LZMA SDK[12]	LZMA	1:6 - 1:7	20-30x	2-3x
BSC[13]	PPMD	1:7 - 1:10	?	?
LZMA SDK	PPMD	1:7 - 1:10	?	?

* Data taken from http://tukaani.org/lzma/benchmarks.html based on the benchmarks done on the source code packages.

Benchmarks

A set of benchmark tests will be performed to collect raw data on performance and disk-space savings.

TBD.

Testing

Unit Tests

The current unit-tests will need to be reviewed and updated in accommodation of this feature. In addition, tests that are no longer valid (due to behavioral changes) will need to be removed. Where applicable new tests will be created to perform white-box testing, especially to test robustness and reliability.

1. Test for exceeding designed nominal PS size (for now 16MB x 65K files = 1TB).

[9] code.google.com/p/snappy/
[10] zlib.net
[11] bzip.org
[12] http://www.7-zip.org/sdk.html

[13] libbsc.com

Known Issues / Limitation

1. ~~Due to the nature of combining the PS into a single file, external tools and even SVN API can no longer directly access individual files in a PS~~.

Milestones

Feature	Status	Notes
Write to single pack file without compression.	Done.	Not used for reading yet.
Read from single pack file without compression.	Partial.	Index entries are written.
Support all operations (add/remove/ modify).	-	
Support all API (diff, path etc). Replace current PS manager.	-	
Compress individual files within the pack using Zlib (no solidifying).	-	
Compress multiple files within a pack (Compression Block).	-	
Pack file splitting.	-	
Extend unit-tests.	-	
Support upgrading WC.	-	
Benchmarking, Tuning and new compression algorithms.	-	
Improved API to leverage the new design.	-	

References

There has been discussions surrounding this feature. Here are the relevant threads and links.

Bug Report: http://subversion.tigris.org/issues/show_bug.cgi?id=908
WG-NG Doc: http://svn.apache.org/repos/asf/subversion/trunk/notes/wc-ng/

Early Patches

http://svn.haxx.se/dev/archive-2003-10/att-0506/svn-compress-tb-smoynes.diff
http://subversion.tigris.org/nonav/issues/showattachment.cgi/1167/compressed-pristines.diff

Discussions on dev-list

text-base penalty: A proposed solution: http://svn.haxx.se/dev/archive-2002-12/1050.shtml
Space wasting: http://svn.haxx.se/dev/archive-2004-03/0238.shtml
[PATCH] Compressed streams (take 2): http://svn.haxx.se/dev/archive-2003-03/1975.shtml
Compressed text-base patch: http://svn.haxx.se/dev/archive-2003-04/1511.shtml
Compressed text-base, take 2: http://svn.haxx.se/dev/archive-2003-05/1037.shtml
Optional | Compressed text-base storage: http://svn.haxx.se/users/archive-2005-06/1331.shtml
[Reminder] Subversion a mentor for Google Summer of Code: http://svn.haxx.se/dev/archive-2006-05/0149.shtml
Optional text base design discussion?: http://svn.haxx.se/dev/archive-2006-10/0419.shtml
Another working copy library: http://svn.haxx.se/dev/archive-2007-01/0522.shtml
A text-base penalty solution (without a working copy rewrite): http://svn.haxx.se/dev/archive-2007-05/0486.shtml
pristine store design: http://svn.haxx.se/dev/archive-2010-02/0354.shtml