RFC-029: Compression and Deduplication

Status: Draft **Date:** January 2026 **Author:** Derrell Piper ddp@eludom.net
Implementation: Proposed

Abstract

This RFC specifies compression and deduplication for the Library of Cyberspace: how vaults reduce storage requirements while maintaining content-addressability, integrity verification, and efficient retrieval. Compression is transparent to the content-addressing layer.

Motivation

Storage efficiency matters for preservation:

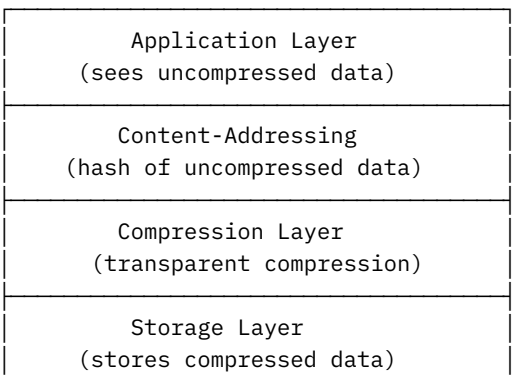
- **Cost** - Less storage means more preservation per dollar
- **Bandwidth** - Compressed transfers are faster
- **Redundancy** - More copies fit in same space
- **Longevity** - Smaller archives survive longer

But compression must not compromise:

- **Integrity** - Hashes must remain valid
- **Addressability** - Content addressing still works
- **Deduplication** - Identical content stored once
- **Accessibility** - Data remains retrievable

Compression Model

Layered Architecture



└──────────────────────────────────────┘

Compression Metadata

```
(define (compress-object data algorithm)
  (let* ((compressed (compress algorithm data))
         (ratio (/ (bytevector-length compressed)
                   (bytevector-length data))))
    `(compressed-object
      (algorithm ,algorithm)
      (original-size ,(bytevector-length data))
      (compressed-size ,(bytevector-length compressed))
      (ratio ,ratio)
      (data ,compressed))))

;; Stored in soup
(soup-object
  (hash "sha256:original-content-hash...")
  (compression
    (algorithm zstd)
    (level 3)
    (original-size 1048576)
    (compressed-size 524288)
    (ratio 0.5)))
```

Compression Algorithms

Supported Algorithms

```
(define compression-algorithms
  `((zstd . ((extension . "zst")
             (levels . (1 3 9 19))
             (default-level . 3)
             (dictionary . #t)))
    (lz4 . ((extension . "lz4")
            (levels . (1 9))
            (default-level . 1)
            (dictionary . #f)))
    (gzip . ((extension . "gz")
             (levels . (1 6 9))
             (default-level . 6)
             (dictionary . #f)))
    (none . ((extension . #f)
             (levels . ())
             (default-level . #f)
```

```
            (dictionary . #f)))))
```

Algorithm Selection

```
(define (select-compression-algorithm content-type size)
  "Select best algorithm for content"
  (cond
    ;; Already compressed formats
    ((member content-type '("image/jpeg" "image/png" "video/mp4"
                            "application/zip" "application/gzip"))
     'none)
    ;; Small objects - overhead not worth it
    ((< size 1024)
     'none)
    ;; Text and code - zstd with dictionary
    ((member content-type '("text/plain" "text/html" "application/json"
                            "application/javascript" "text/x-scheme"))
     'zstd)
    ;; Binary data - lz4 for speed
    ((string-prefix? "application/" content-type)
     'lz4)
    ;; Default
    (else 'zstd)))
```

Zstd Dictionaries

```
;; Train dictionary on sample data
(define (train-dictionary samples dict-size)
  "Train zstd dictionary from sample data"
  (zstd-train-dictionary samples dict-size))

;; Content-type specific dictionaries
(define dictionaries
  `((scheme . ,(load-dictionary "scheme.dict"))
    (json . ,(load-dictionary "json.dict"))
    (markdown . ,(load-dictionary "markdown.dict"))))

(define (compress-with-dictionary data content-type)
  (let ((dict (assoc-ref dictionaries content-type)))
    (if dict
        (zstd-compress-with-dict data dict)
        (zstd-compress data))))
```

Compression Operations

Transparent Compression

```
(define (cas-put-compressed data)
  "Store data with transparent compression"
  (let* ((hash (content-hash data))  ; Hash uncompressed
         (algorithm (select-compression-algorithm
                     (detect-content-type data)
                     (bytevector-length data)))
         (stored (if (eq? algorithm 'none)
                     data
                     (compress algorithm data))))
    ;; Store compressed, index by uncompressed hash
    (storage-put hash stored)
    ;; Record compression metadata
    (soup-put hash
      compression: `((algorithm . ,algorithm)
                     (original-size . ,(bytevector-length data))
                     (compressed-size . ,(bytevector-length stored))))
    hash))

(define (cas-get-decompressed hash)
  "Retrieve and decompress data"
  (let* ((stored (storage-get hash))
         (meta (soup-get hash))
         (algorithm (assoc-ref (assoc-ref meta 'compression) 'algorithm)))
    (if (or (not algorithm) (eq? algorithm 'none))
        stored
        (decompress algorithm stored))))
```

Batch Compression

```
(define (compress-batch objects)
  "Compress multiple objects efficiently"
  ;; Group by content type for dictionary efficiency
  (let ((groups (group-by detect-content-type objects)))
    (append-map
      (lambda (group)
        (let ((content-type (car group))
              (items (cdr group)))
          (map (lambda (obj)
                 (compress-with-dictionary obj content-type))
               items)))
      groups)))
```

Recompression

```
(define (recompress-object hash new-algorithm new-level)
  "Recompress object with different settings"
  (let* ((data (cas-get-decompressed hash))
         (new-compressed (compress new-algorithm data new-level)))
    (storage-put hash new-compressed)
    (soup-update hash
      compression: `((algorithm . ,new-algorithm)
                     (level . ,new-level)
                     (original-size . ,(bytevector-length data))
                     (compressed-size . ,(bytevector-length new-compressed))))
    (audit-append action: 'recompressed
                  hash: hash
                  algorithm: new-algorithm)))
```

Deduplication

Content-Based Deduplication

```
;; Content addressing provides automatic deduplication
(define (store-deduplicated data)
  (let ((hash (content-hash data)))
    (if (cas-exists? hash)
        (begin
          ;; Increment reference count
          (soup-update hash ref-count: (+ 1 (soup-ref-count hash)))
          (audit-append action: 'deduplicated hash: hash)
          hash)
        (cas-put data))))
```

Block-Level Deduplication

```
;; Split large objects into blocks
(define *block-size* 65536)  ; 64KB

(define (chunk-object data)
  "Split object into content-defined chunks"
  (let ((chunks '())
        (pos 0))
    (while (< pos (bytevector-length data))
      (let ((boundary (find-chunk-boundary data pos)))
        (set! chunks (cons (subbytevector data pos boundary) chunks))
        (set! pos boundary)))
    (reverse chunks)))
```

```scheme
;; Content-defined chunking (Rabin fingerprint)
(define (find-chunk-boundary data start)
  "Find chunk boundary using rolling hash"
  (let ((min-size 4096)
        (max-size 131072)
        (mask #x1FFF))  ; Average 8KB chunks
    (let loop ((pos (+ start min-size)))
      (cond
        ((>= pos (min (+ start max-size) (bytevector-length data)))
         pos)
        ((= (bitwise-and (rabin-hash data pos) mask) 0)
         pos)
        (else (loop (+ pos 1)))))))
```

Chunk Storage

```scheme
(define (store-chunked data)
  "Store large object as deduplicated chunks"
  (let* ((chunks (chunk-object data))
         (chunk-hashes (map (lambda (chunk)
                              (store-deduplicated chunk))
                            chunks)))
    ;; Store manifest
    (let ((manifest `(chunked-object
                      (chunks ,chunk-hashes)
                      (total-size ,(bytevector-length data))
                      (chunk-count ,(length chunks)))))
      (cas-put (serialize manifest)))))

(define (retrieve-chunked manifest-hash)
  "Reassemble chunked object"
  (let* ((manifest (deserialize (cas-get manifest-hash)))
         (chunks (map cas-get (assoc-ref manifest 'chunks))))
    (bytevector-concatenate chunks)))
```

Deduplication Statistics

```scheme
(define (deduplication-stats)
  "Calculate deduplication effectiveness"
  (let* ((objects (soup-query type: 'any))
         (logical-size (sum (map soup-original-size objects)))
         (physical-size (sum (map soup-compressed-size
                                  (delete-duplicates objects hash=?)))))
    `((logical-size . ,logical-size)
      (physical-size . ,physical-size)
```

```
      (dedup-ratio . ,(/ logical-size physical-size))
      (space-saved . ,(- logical-size physical-size)))))
```

Delta Compression

Version Deltas

```
(define (store-delta base-hash new-data)
  "Store object as delta from base"
  (let* ((base-data (cas-get base-hash))
         (delta (compute-delta base-data new-data))
         (new-hash (content-hash new-data)))
    (if (< (bytevector-length delta)
           (* 0.5 (bytevector-length new-data)))
        ;; Delta is worthwhile
        (begin
          (storage-put new-hash delta)
          (soup-put new-hash
            delta: `((base . ,base-hash)
                     (type . xdelta)))
          new-hash)
        ;; Store full object
        (cas-put new-data))))

(define (retrieve-delta hash)
  "Reconstruct object from delta"
  (let ((meta (soup-get hash)))
    (if (assoc-ref meta 'delta)
        (let* ((base-hash (assoc-ref (assoc-ref meta 'delta) 'base))
               (base-data (cas-get base-hash))
               (delta (storage-get hash)))
          (apply-delta base-data delta))
        (storage-get hash))))
```

Delta Chains

```
;; Limit delta chain depth
(define *max-delta-depth* 10

(define (delta-chain-depth hash)
  "Calculate depth of delta chain"
  (let ((meta (soup-get hash)))
    (if (assoc-ref meta 'delta)
        (+ 1 (delta-chain-depth (assoc-ref (assoc-ref meta 'delta) 'base)))
        0)))
```

```
(define (should-store-delta? base-hash)
  "Check if delta storage is appropriate"
  (< (delta-chain-depth base-hash) *max-delta-depth*))
```

Archive Compression

Sealed Archive Format

```
;; RFC-018 integration
(define (create-compressed-archive objects)
  "Create compressed sealed archive"
  (let* ((serialized (serialize-objects objects))
         (compressed (zstd-compress serialized 19))  ; Max compression
         (encrypted (age-encrypt compressed recipients)))
    (cas-put encrypted)))
```

Streaming Compression

```
(define (stream-compress input-port output-port algorithm)
  "Stream compression for large files"
  (let ((ctx (compression-context-create algorithm)))
    (let loop ()
      (let ((chunk (read-bytevector *chunk-size* input-port)))
        (unless (eof-object? chunk)
          (write-bytevector (compress-chunk ctx chunk) output-port)
          (loop))))
    (write-bytevector (compress-finish ctx) output-port)))
```

Performance Optimization

Compression Cache

```
(define compression-cache (make-lru-cache 1000))

(define (cached-decompress hash)
  "Cache decompressed data for frequent access"
  (or (lru-get compression-cache hash)
      (let ((data (cas-get-decompressed hash)))
        (lru-put! compression-cache hash data)
        data)))
```

Parallel Compression

```
(define (parallel-compress objects num-threads)
  "Compress multiple objects in parallel"
  (let ((work-queue (make-channel))
        (results (make-channel)))
    ;; Start workers
    (do ((i 0 (+ i 1)))
        ((= i num-threads))
      (thread-start!
        (make-thread
          (lambda ()
            (let loop ()
              (let ((obj (channel-get! work-queue)))
                (when obj
                  (channel-put! results (compress-object obj))
                  (loop))))))))
    ;; Queue work
    (for-each (lambda (obj) (channel-put! work-queue obj)) objects)
    ;; Collect results
    (map (lambda (_) (channel-get! results)) objects)))
```

Adaptive Compression

```
(define (adaptive-compress data)
  "Select compression based on content analysis"
  (let* ((sample (subbytevector data 0 (min 4096 (bytevector-length data))))
         (entropy (calculate-entropy sample)))
    (cond
      ;; High entropy - already compressed or encrypted
      ((> entropy 7.9) 'none)
      ;; Low entropy - highly compressible
      ((< entropy 4.0) (values 'zstd 19))
      ;; Medium entropy - balanced
      (else (values 'zstd 3)))))
```

Integrity

Verification

```
(define (verify-compressed-object hash)
  "Verify compressed object integrity"
  (let* ((stored (storage-get hash))
         (meta (soup-get hash))
         (decompressed (decompress (assoc-ref meta 'algorithm) stored))
```

```
        (computed-hash (content-hash decompressed)))
    (equal? hash computed-hash)))
```

Corruption Recovery

```
(define (recover-compressed hash)
  "Attempt to recover corrupted compressed object"
  ;; Try partial decompression
  (let ((partial (decompress-partial hash)))
    (when partial
      (audit-append action: 'partial-recovery hash: hash)
      partial))

  ;; Fetch from replica
  (let ((replica-data (fetch-from-replica hash)))
    (when replica-data
      (storage-put hash replica-data)
      replica-data)))
```

References

1. Zstandard - Facebook's compression algorithm
2. Content-Defined Chunking - Restic blog
3. RFC-018: Sealed Archive Format
4. RFC-020: Content-Addressed Storage

Changelog

- **2026-01-07** - Initial draft

Implementation Status: Draft **Dependencies:** cas, zstd, soup **Integration:** Storage layer, archive creation, replication