消失訂正と重複排除を併用するオブジェクトストレージシステムの設計と実装

石山 政浩[2]　田所 秀和[1]　松崎 秀則[1]

概要：
本稿では，消失訂正と重複排除を併用した高い容量効率を実現するオブジェクトストレージシステムの設計と実装について提案する．容量効率を高める方法である重複排除技術は，大容量のメモリと処理時間が必要となるため，低価格で大容量をターゲットとする代表的なオブジェクトストレージシステムで重複排除を採用することは困難であった．本稿では，Kinetic に代表される IP ネットワークに接続される HDD を活用し，重複位置探索をこれらに行わせることで，コストを維持したまま，容量に対するスケール性を維持した重複排除を行うことを可能とした．また提案方式を実装し，重複排除が実現できることを示した．加えて，提案方式による重複排除効率をシミュレーションを用いて解析し，提案法式によって実効容量比で約 30%の向上が得られることを示した．

1. はじめに

近年の非構造データの爆発的な増大を受け，その保存先としての大規模ストレージとしてオブジェクトストレージが注目を集めている．オブジェクトストレージは抽象化されたオブジェクトの単位で管理されているため，容易にスケールアウト可能といった高い拡張性を持つため大規模なストレージシステムを安価に構築できるという性質を持つ．特にアーカイブ用途として選択されることが多いオブジェクトストレージシステムにおいては，1 バイトあたりの単価が特に重要となっており，容量効率向上のため Reed Solom 符号 [16] に代表される消失訂正符号 (Erasure Coding) を用するオブジェクトストレージが増加している [12][1][10][9][17] .

さらに容量単価を削減する方法として重複排除技術があるが，オブジェクトストレージのような高分散で大容量なストレージシステムへの摘要は難しいとされていた．重複位置の探索には大容量のメモリと処理時間が必要となるため，システムのコストを引き上げる．このため，低価格大容量をターゲットとする代表的なオブジェクトストレージシステムで重複排除を採用しているシステムは現時点では知られていない．

また，ストレージシステム全体のコストを引き下げる取り組みとして，Seagate 社が中心となって提案が行われている Kinetic[18] の存在がある．Kinetic は，HDD が直接 IP ネットワークに接続され，それ自身が Key-Value Store 型のネットワーク API を持つ．近年使用されているスケールアウト型のストレージでは，ネットワークを介してストレージを束ねる方式であるため，これまでは HDD の記憶領域を単にネットワークを介してアクセスするためにサーバや SAN のようなストレージのためのネットワークが必要とされていた．Kinetic のような IP 接続型 HDD では，これらを取り払うことができるようになるため，機器コスト及び管理コストを引き下げることができると期待されている．

これらの背景を踏まえ，本稿では，新しい，信頼性のある低価格で大容量なオブジェクトストレージの実装方式を提案する．提案方式では，IP 接続型の HDD を利用し，さらに消失訂正と重複排除を併用することで高い容量効率を実現する．

2. 要求条件

本章では，信頼性のある低価格で大容量なオブジェクトストレージを実現するために，満たすべきと考える要求条件について述べる．

2.1 重複排除を併用しても消失符号の信頼性を維持すること

ストレージにおける消失符号の典型的な利用方式として

[1] （株）東芝 研究開発センター コンピュータアーキテクチャ・セキュリティラボラトリー
[2] （株）東芝 ストレージ＆デバイスソリューション社 メモリ事業部

は，保存すべきデータを，データ部分の分割数 (情報シンボル部)k 個に分割し，これに冗長数 (パリティシンボル部)m 個を計算し，これら k+m 個をすべて異なるディスクに保存するというものである．この例では，任意の m 個のデータ消失に耐えるため，m 台の disk 故障まで許容されることとなる．

一方，重複排除は同じデータを纏める処理となる．このとき，消失訂正のために異なるディスクに保存したものを同じディスクに纏めるような処理を行った場合，上記のデータ消失への耐性を維持できない可能性がある．このため，消失訂正のパリティを共有する集合に対して重複排除を適用する際には，同じディスクに保存されないという条件を必ず満たす必要がある．

2.2 自由位置での重複排除を実現すること

オブジェクトストレージにおける重複排除にはいくつかの方式が考えられる．一つはファイル単位での重複排除である．しかしこの方式は，ある 2 つのファイルにおいて，1 ビットの差異があっただけで重複排除を行うことができなくなり，重複排除の効率は高くないと考えられる．図 1 に例を示す．図中 A, B, C, X, Y, Z はそれぞれが同じデータとなっている部分を示している．O1 と O2 は，先頭の A, B は同一であるものの，末尾が異なっている場合．O3 は O2 に対して，先頭に Y が挿入され，また A と B の間に X が挿入されている．図 1(1) は，ファイル単位での重複排除を示している．このように，ファイル中に重複部分が含まれているが，ファイル単位での重複排除はこれを検出することはできない．

そのため，ファイルをある長さに分割し，その分割単位で重複排除を行う方式が考えられる (固定位置による重複排除)．この方式では，前述の 1 ビットの差異がある分割部分は重複排除が適用されないが，他の分割部分においては重複排除が適用されるため，ファイル単位での重複排除に対して，より多くの重複排除が行えることが期待できる．しかしこの方式は，例えばあるファイルの先頭に 1 バイト挿入された場合に，まったく重複排除が行うことができなくなるという問題がある．図 1(2) は，固定位置での重複排除を示している．このように，同一位置に現れる重複部分は検出できるが，挿入等により位置が異なる場合にはこれを検出することはできない．

そのため，重複部分の検出位置を固定せず，自由位置で重複排除を行う方式が考えられる (自由位置による重複排除)．この方式では，前述の 1 バイト挿入が行われても，その 1 バイトのみが重複排除の対象にならず，残りの部分は重複排除の対象となり，高い重複排除の効果が期待できる (図 1(3))．Meyer らの報告 [13] によれば，分割するサイズにも依存するが，ファイル全体に対する重複排除と比較して 15%から 20%程度の容量に対する優位性があるとされている．よって，提案方式では自由位置による重複排除が行えることが望ましい．

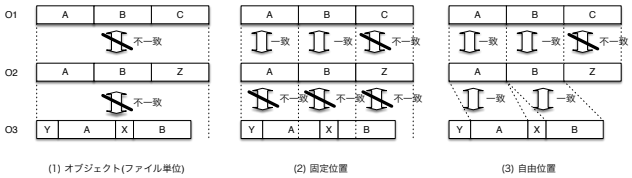

図 1 ファイル単位，固定位置，自由位置による重複排除の比較: 自由位置での重複排除は他の方式に対してより高い重複排除の効果が期待できる．

2.3 容量の増加に対しての重複排除性能維持に対するコスト増が十分に小さいこと

ストレージに保存されるデータ量の増加に対して，重複排除の性能を担保するために必要なコストが大きく増加するシステムでは，大容量化に対して低コストを維持することが難しい．そのため，データ量の増加に対して，可能な限り低いコストで対応できるシステムが望ましい．

3. 提案方式

本章では，まず提案方式のオブジェクトストレージシステムとしての概要を述べ，その上で自由位置の重複位置を実現する手法について述べる．

3.1 提案方式の概要

提案方式のアーキテクチャの概要を図 2 に示す．

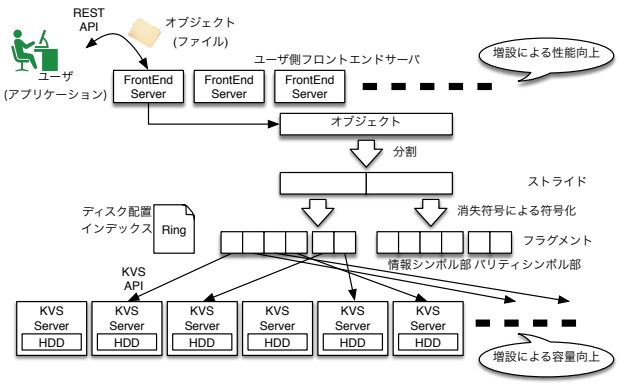

図 2 提案方式の概要: FrontEnd Server がオブジェクトを消失訂正を使用して分割し，KVS Server へと保存する．それぞれの台数に依存関係は無いため，ユーザはアグリゲート性能が必要な際は FrontEnd Server を，容量の増加が必要な際は KVS Server を，それぞれ任意の時点で独立に増設可能であり，精緻なプロビジョニングを可能とする．

提案方式では，ユーザ (アプリケーション) は REST API[7] を使用してオブジェクト (ファイル) を保存/取得

することを想定している．この API を担当するのは FrontEnd Server である．FrontEnd Server は，オブジェクトの保存要求をユーザから受け取ると，オブジェクトを適切な数の断片 (以下ストライドと呼ぶ) へと分解し，それぞれに消失符号を使用して符号化を行う．

符号化によって得られた情報シンボル部及びパリティシンボル部分はフラグメントと呼ばれる．FrontEnd Server は，これらのフラグメントをそれぞれを異なる KVS Server へと Key-Value Store (KVS) API を使用して保存する．保存する KVS Server の選択には Swift[2] などで使用されている Ring と呼ばれるディスク配置インデックスを使用する．KVS Server は基本的には Key-Value Store であり，提案方式では，Seagate 社の Kinetic[18] のような HDD が直接ネットワークに接続され KVS API を提供するようなデバイスを想定している [*1]．

提案方式では，FrontEnd Server の台数と KVS Server の台数の間に依存関係はない．そのため，ユーザはストレージの総容量を増加させたい時には，単に KVS Server を増加させるだけでよい．また， FrontEnd Server にはユーザデータは保存されず，オブジェクトの保存処理中以外はステートレスなノードとして扱うことが可能である．そのため，ストレージのアグリゲート性能を一時的に引き上げたい，たとえば期末処理などでストレージへのアクセスが一時的に増加するような期間には，IaaS のような環境を活用して一時的に FrontEnd Server を増設し，性能増加に対応することができる．このため，ユーザニーズに対して精緻なプロビジョニングを実現することができ，無駄な投資を回避することができる．

3.2 提案方式におけるオブジェクト保存処理

提案方式におけるオブジェクト保存処理の手順について述べる．

FrontEnd Server は，ユーザからオブジェクトの保存要求を受けると，以下の処理を行う．以下消失符号のパラメータは k=4, m=2 であるとする．

3.2.1 オブジェクトのフラグメントへの分割

FrontEnd Server は，オブジェクトをストライドの整数倍と等しい長さまでパディングを行う．その後，各ストライドに対して消失符号による符号化を行い，情報シンボルとパリティシンボルそれぞれのフラグメントを得る (図 3)．

3.2.2 各ストライドの fingerprint を計算

各ストライドの fingerprint を Rabin Fingerprint [15] に代表される rolling hash を用いて求める．図 3 では，ストライド A, B, C の fingerprint はそれぞれ FP_A, FP_B, FP_C となっている．

[*1] 但し，重複位置探査処理などは別途内部で処理できる機能が付与されていることが望ましい

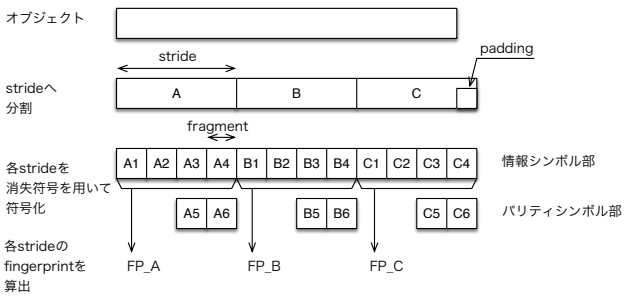

図 3 FrontEnd Server における消失符号を用いたオブジェクト分割

3.2.3 各ストライドの fingerprint で Ring を引き，得られた結果に従って保存

Ring は，OpenStack Swift[2] などが採用しているディスク選択のためのインデックスである．Ring では，あるキーに対して，データの保存先 (本提案方式では KVS Server) の集合が得られる表の様に振る舞うものである． この集合を disk_set と呼ぶ．Ring では， 任意の disk_set において，同じデータの保存先が現れることはないように構成される．

FrontEnd Server は，各ストライドの fingerprint をキーとして Ring を引き，disk_set を得て，この内容にしたがってフラグメントを保存する．

仮に，FP_A, FP_B, FP_C それぞれで Ring を引いた結果が，

```
FP_A = [K2, K6, K0, K5, K1, K8]
FP_B = [K1, K9, K6, K0, K3, K2]
FP_C = [K8, K7, K4, K2, K9, K6]
```

であった場合，各 フラグメント は 図 4 のように KVS Server へと保存される．ここで，Kn は KVS Server の識別子を表すものとする．

各ストライドの fingerprint で disk_set が定まるため，同一の内容を持つストライドは必ず同一の disk_set に保存される．

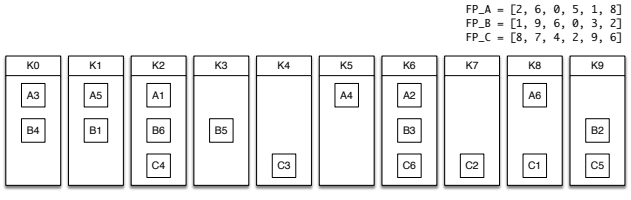

図 4 KVS Server への保存例

また，各フラグメントには，そのフラグメントを先頭として，1 ストライド長の fingerprint を計算した際の fingerprint の内部状態 (rollsum_state) を保存する．

3.2.4 重複位置探索のための情報を保存

提案方式では，KVS Server などの計算資源を活用し，重複位置探索の分散処理を行う．このため，他ノードの探査のための情報を保存する必要がある．処理の詳細について

は 3.3.2 章で述べ，ここでは保存する情報についてのみ説明する．

FrontEnd Server は，ストライドの最初のフラグメントを半分に分割し，それぞれの fingerprint を計算する．これらを 1/2 fragment fingerprint First Part (FFF1) および Second Part (FFF2) と呼ぶ．そして FrontEnd Server は，FFF1，FFF2 および ストライドの fingerprint (SF) を記憶する． この集合を Fingerprint Set と呼ぶ (図 5)．

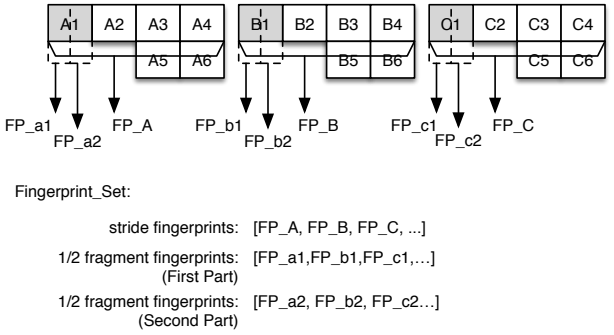

図 5 FrontEnd Server における Fingerprint Set の作成: 一つのストライドにつき，先頭フラグメントを 2 分割した部分の fingerprint とストライド全体の fingerprint の 3 つが保存される

Fingerprint Set は一定量に達した時点で，事前に定められたルールにしたがって探索可能な形で KVS Server へと保存される．

最終的にオブジェクトは，オブジェクトの管理情報 (object_metadata), ストライドの管理情報 (stride_info)，フラグメントから構成され，図 6 のようなリスト構造を構成するように保存される．なお，メタデータはそのサイズが小さいため，消失訂正ではなく複製 (replication) の形で保存される．

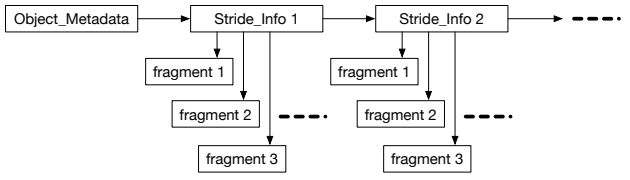

図 6 オブジェクトの保存時のメタデータ及び実データの構成: 管理情報がリスト構造を持ち，各ストライドの管理情報 (stride_info) が実データの位置を持つ

3.3 提案方式における重複排除処理

3.3.1 ストライド毎の固定位置及び自由位置での重複排除

前述のように，提案方式では同じ fingerprint を持つストライドは同じ disk_set に保存される．これは同じ内容である可能性の高いストライドは，同じ disk_set に保存されることを意味しており，同一内容のストライドであれば，同じ内容を持つフラグメントが同じ KVS Server に保存されることを示している．この特性を利用して，各 KVS Server は空き時間に同じ fingerprint を持つストライドに含まれていたフラグメントの内容を比較し，同一の内容であれば重複排除処理を行う．この処理によって，ストライドの重複排除が実現できる．

図 7 に固定位置での重複排除の問題の例を示す．すでにストライド Sa, Sb が保存されていたと仮定する．これに対しオブジェクト O1 を保存することを考える．O1 は，Sa, Sb と同じ内容を含んでいるが，ストライド長で順にオブジェクトを分割した場合 (固定位置による分割)，Sb はストライド境界に一致しているのに対し，Sa は一致していない．すなわち，単純に固定位置で分割した場合には，提案方式は 2.2 章で述べた固定位置の重複排除は実現できるが，自由位置での重複排除は実現できないこととなる．

一方オブジェクトを固定位置で分割するのではなく，オブジェクトからストライドと同じ長さで同一内容の部分を探索 (重複位置探索) し，その位置でストライドを分割すれば，前述の重複排除処理であっても，V2, V4 が一致し，自由位置での重複排除が実現できることがわかる．このため，自由位置での重複排除を行うため，オブジェクト内に含まれるストライドの重複位置の探索を行う必要がある．

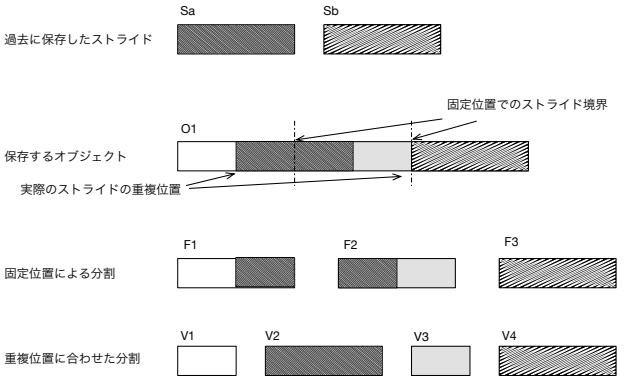

図 7 重複位置に合わせたオブジェクトの分割: 単純に固定位置でオブジェクトを分割した場合，ストライド単位での重複排除は F3 だけが対象となるが，オブジェクトの中のストライド長を持つ重複部分を探索し，その位置でストライドを分割することで，同じオブジェクトでも V2, V4 の 2 つのストライドを重複排除の対象とすることができる

3.3.2 KVS Server による重複位置探索

本節では，自由位置での重複排除を実現するための，KVS Server による重複位置探索の方式について述べる．

3.2.4 章で述べたとおり，FrontEnd Server は Fingerprint Set を保存する．KVS Server は，定期的に任意の Fingerprint Set を事前に定められたルールにしたがって探索し，得られた Fingerprint Set を一時的に記憶する．そして，自

身がもつ各フラグメントに対して，フラグメント長の 1/2 の長さに対して，順に rolling hash を適用し，Fingerprint Set の 1/2 fragment fingerprint と一致していないかを調べる．

一致していた場合，そのフラグメントの 1 ストライド先のフラグメントを他の KVS Server から読出し，ストライドの fingerprint (SF) を計算する．ただし，一致した位置によっては，そのフラグメントは想定されるストライドの先頭のフラグメントのデータをすべて含まない場合がある．この場合，一つ前のフラグメントを読出し，そのフラグメントから 1 ストライド先のフラグメントを読出す必要がある．

なお，SF の計算には同時に保存されている fingerprint の内部状態 (rollsum_state) を利用する．このため，ストライド全体の fingerprint を計算する際にも，対象となるストライドの先頭と最後を含む 2 つのフラグメントのみで計算可能であり，その間のフラグメントは計算には不要である．これは，重複位置探索時のネットワーク負荷の削減に大きく寄与する．

SF が一致した場合，その位置が重複位置である可能性がある．この位置を pivot と呼ぶ．KVS Server は，得られた pivot を事前に定められたルールにしたがって探索可能な形で KVS Server に保存する．

図 8 に具体例を示す． KVS Server K0 で， フラグメント A3 が探査対象となっていると仮定する．K0 は，A3 に対して先頭から 1/2 フラグメント長の rolling hash を順に計算し，保持している Fingerprint Set の 1/2 fragment fingerprints と比較する．n byte 進んだところで 同一の fingerprint を発見した場合 (図 8(1))， 1 ストライド後のフラグメントである B3 を他の KVS Server(K6) から読み込む (図 8(2))．この B3 と，A3 の rollsum_state を使用して，A3 の先頭から n byte 進んだ地点の SF を計算し， Fingerprint Set の stride fingerprints と比較する (図 8(3))．stride fingerprints に一致するものがあった場合，この pivot(例では 2*フラグメント長+n) を記録する．

また，pivot の記録時に，あるオブジェクトについてそのオブジェクトに関する pivot が一定量になっているものについては，適宜 KVS Server が 任意の FrontEnd Server へと通知する．

3.3.3 FrontEnd Server による オブジェクトの再配置

FrontEnd Server は，KVS Server からの通知を受けると，対象となる object を読み出す．FrontEnd Server は，読出した object を各 pivot に合わせてストライドへと分割する．この時作られるストライドは inner stride と呼ばれ，実際の保存単位のストライドは outer stride と呼ばれる．outer stride の長さは必ずストライド長，すなわちフラグメント長の k 倍に等しいが，inner stride はそれよりも短くて良い．また，inner stride にも順に番号が与えられる．

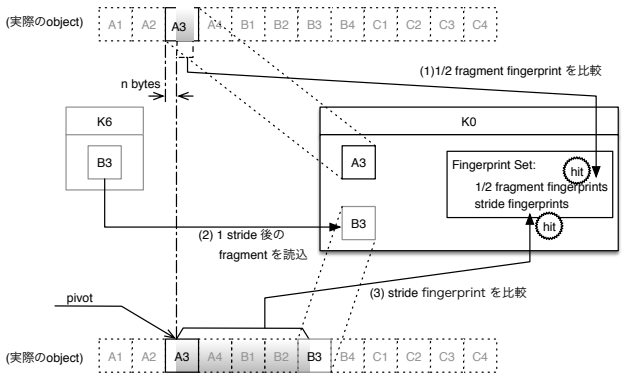

図 8 KVS Server による 重複位置の検出: 次ノードが持つフラグメントについて，FFF1/FFF2 で探索を行い，ヒットした時点で 1 ストライド長先のフラグメントを読み込み，ストライドの fingerprint を探索する．SF が一致した位置 (pivot) は記録され，一定の量に達した時点で KVS Server へと通知される

これを inner stride index と呼ぶ．ただし，すでに重複排除されているフラグメントについては移動させないように inner stride は作られる．

図 9 に具体例を示す．A3 の途中に pivot があると仮定する．また，すべてのフラグメントがまだ重複排除されてないものとする．(図 9(1-1))．この場合，A3 の後半から B3 の後半までが重複が存在する可能性のあるストライドとなるため，pivot の位置にストライドの境界が作られるように inner stride が生成される．(図 9(1-2))．

また，inner stirde の長さが規定のストライド長に満たない場合には，ストライド長まで padding を行い，消失訂正符号を計算する．一方で，もし短い inner stride が大量にある場合，これらに対するパリティシンボルが容量のオーバヘッドとなる可能性がある．そのため，もし複数の inner stride が，ストライド長より短い場合 [*2] には，これらを纏めて一つの outer stride とすることができる．このような stride を packed stride と呼ぶ．図 9(1-3) はその例であるが，これは C4 が padding されているためにこのような構成をとることができる．図では outer stride 1 の padding は割愛しているが，もし inner stride 1 と inner stride 4 の長さと packed stride header の長さの和よりもストライド長が短かった場合には，適宜 padding が行われる．これにより，短い inner stride に対するパリティシンボルを一つに纏めることができるため，容量に対するオーバヘッドを回避することができる．

FrontEnd Server は， outer stride に対して消失符号を計算し，outer stride をフラグメント長に分割して保存する．オブジェクトを読出す際には，outer stride を順に読出し， 含まれている inner stride を元の順に並べていくことで再構成する．

[*2] 厳密には，packed stride のヘッダが存在するため，そのヘッダ長を加算したものよりも短い場合

次に，C1 から C4 のフラグメントから構成されるストライドに対してすでに重複排除が適用されている場合について述べる (図 9(2-1))．この場合，pivot の位置にストライドの境界が作られるように適宜 padding を行うが，C1-C4 は移動することができないため，B4 の後に padding を行う (図 9(2-2))．

FrontEnd Server は，この再配置されたストライドを保存する．この保存処理は最初に object を保存する場合と基本的には変わらないため，前述 (3.3.1 章) の各 KVS Server による重複排除処理がいずれ適用される．これにより，自由位置の重複排除が実現できる．

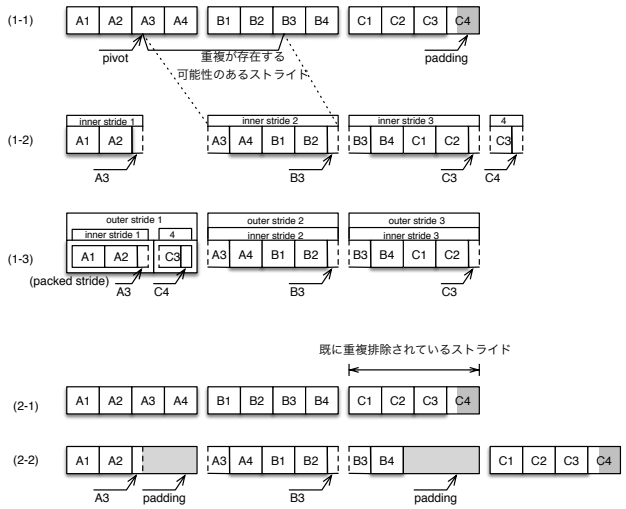

図 9 FrontEnd Server によるストライドの再配置: オブジェクトを pivot の位置で分割し保存する．規定のストライド長に満たないものは padding されるか，packed stride として纏められる

4. 試作

我々は提案方式を実装し，その動作検証を行った．

試作の動作検証の環境は， KVS Server が 8 台， FrontEnd Server が 1 台である．消失符号訂正のパラメータとして k=4, m=2 とし， フラグメント長は 4K バイトとした．よってストライド長は 16K バイトとなっている．

4.1 試作による重複排除処理の動作検証

重複排除の動作確認には v1.pptx ，v2.pptx という 2 つの 2 つの PowerPoint で作成されたファイルを使用した．v2.pptx は，v1.pptx を元に修正を行ったものであり，部分的に重複が含まれていることが期待できるものである．

重複排除を行う前に，この 2 つのファイルにどのような重複部分があるかを調査した．結果を図 10 に示す．この図は，v2.pptx が，v1.ppt に対してどのような重複部分を持っているかを示したものである．緑色の部分が重複しているストライドを示しており，薄い緑の枠は，それらのストライドが連続していることを示している．この調査の結果，v2.pptx 内には v1.pptx と同一のストライドが 18 個存在していることがわかる．

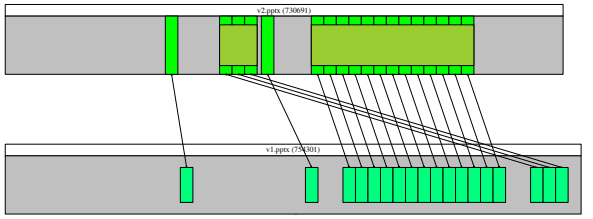

図 10 v2.pptx の v1.ppt に対する重複部分: 緑色の部分が重複しているストライドを示しており，薄い緑の枠は，それらのストライドが連続していることを示している

この 2 つのファイルをオブジェクトストレージに保存し，ファイル v2.pptx に対して重複排除処理を行った．結果を図 11 に示す．

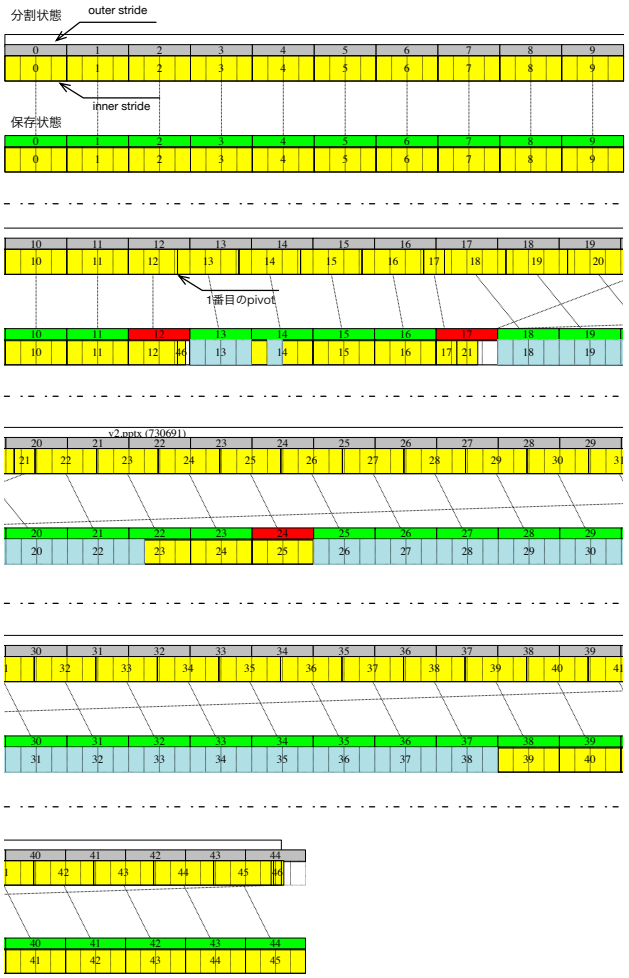

図 11 v2.pptx に対して重複排除を行った結果: 上段がストライドをどのように分割したかを示しており，下段がどのように保存されたかが示されている．細い部分は outer stride を示しており，その下が inner stride を示している．水色になっている部分が重複排除が行われた部分を示している．また，赤い outer stride は，packed stride であることを示している

上段がストライドをどのように分割したかを示しており (分割状態)，下段がどのように保存されたかが示されている (保存状態)．細い部分は outer stride を示しており，その下が inner stride を示している．また，水色になっている inner stride は重複排除が行われたことを示しており，赤い outer stride は，そのストライドが packed stride であることを示している．なお，図が長いため，5 分割し，これを縦に並べて記載した．

初期の保存状態は，pivot が全く存在しないため，outer stride と inner stride はまったく同じ構造となる．重複排除を実行した後である図 11 においても，outer stride 11 までは pivot が存在しないため，同じ構成となっている．しかし，本来 inner stride 12 内に入る部分に pivot があるため，inner stride 12 は通常のストライドより短くなっていることがわかる．この結果，保存時には outer stride 12 には，inner stride 12 と，46 が packed stride として保存されていることがわかる．そして，inner stride 13 を保持している outer stride 13 は意図どおりに重複排除されていることがわかる．

同様に inner stride 17 に pivot があり，このストライドは短くなっている．この結果 outer stride 18 から 20 までは重複排除されていることがわかる．

最終的には，inner stride 13, 18 から 20, 22, 26 から 38 の計 18 ストライドが重複排除されていることがわかる．この結果は図 10 と合致しており，重複排除が正しく行われたことを示している．

また，一部フラグメントが部分的に重複排除されている部分がある (inner stride 14, の第 2 フラグメント及び inner stride 23 の第 1 フラグメント) が，これは KVS Server の数が少ないため，偶然同一の内容のフラグメントが同じ KVS Server に保存されたものと考えられる．例えば，inner stride 13 は重複しているストライドがあることがわかっているが，重複している部分はストライド長と完全に一致しているわけではなく，実際にはもう少し長い可能性がある．この例では，この部分は次のストライドの第 2 フラグメントを超える部分まで一致している部分が存在していたが，ストライドの最後までは一致していなかったと考えられる．その結果，inner stride 14 の第 1, 第 2 フラグメントについては，全く同じ内容のフラグメントが存在しており，この例では偶然第 2 フラグメントが同じ KVS Server へ保存されたと考えられる．

これにより，試作において設計に沿った重複排除が実現できていることがわかった．

5. シミュレーションによる重複排除効率の調査

提案方式における重複排除の効果を調査するため，提案方式の重複排除処理部分のシミュレータを試作した．本シミュレータは与えられたファイルを逐次処理していくものであり，固定位置で区切られた各ストライドの SHA-1 ハッシュ値を保存する．また，ストライド区間を 1 バイトづつ移動させ，その SHA-1 ハッシュ値が過去に (固定位置で) 保存したストライドと一致する部分 (自由位置一致) を自由位置での重複排除可能部分として記録する．また，ファイル全体が完全に一致している (ファイル一致) も同様に記録する．加えて，ファイル一致ではないが，ファイル内のストライドが固定位置で一致 (固定位置一致) するものが一部存在する場合には，それも記録する．また，ストライド長は自由に選択できる．

対象とするファイルは，オブジェクトストレージへの保存対象として考えられるオフィス系ドキュメント (Microsoft Word, Excel, Powerpoint および PDF) とした．これらのファイルを，2013 年 4 月から 2015 年 2 月末までの一人のユーザの電子メールから抽出し，シミュレータで処理を行った．

ストライド長は 8192 バイト, 32768 バイト, 65536 バイト，131072 バイト，262144 バイト， 524288 バイトの 6 種類で調査を行った．

5.1 重複排除のシミュレーション結果

今回の調査で対称となったファイルの内訳を表 1 に示す．

Total はすべてのファイルを，Others はなんらかの理由でファイルタイプが判定できなかったものを示している．ファイル数比は，総ファイル数に対して，そのファイルタイプのファイル数がどれだけの割合を占めているかを表しており，たとえばこの結果では powerpoint はファイル数では全体の 20.6%を占めているが，ファイルサイズの合計では 46.6%を占めていることがわかる．この結果より，ファイル数では powerpoint, word, excel, pdf に大きな差はないが，ファイルサイズでは powerpoint が大きな比率を占めていることが分かり，powerpoint はファイルサイズが大きい傾向にあることが予想できる．

また，これらのファイルサイズの分布を図 12 に示した．なお，横軸は対数軸となっていることに注意されたい．

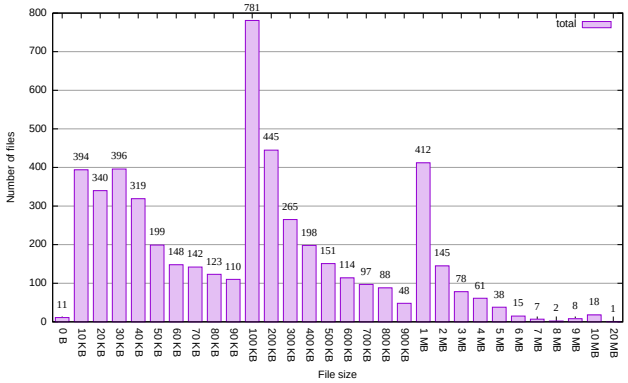

図 12 ファイルサイズの分布 (全体)

表 1 対象ファイルの内訳: 総ファイル数比，総サイズ比はそれぞれ総数に対しての比を示している

ファイルタイプ	ファイル数	総ファイル数比	総バイト数	総サイズ比
Total	5155	1.000	3048909948	1.000
powerpoint	1061	0.206	1421337552	0.466
word	1295	0.251	407023914	0.133
excel	1315	0.255	292974252	0.096
pdf	1451	0.281	909636709	0.298
Others	33	0.006	17937521	0.006

5.2 重複ファイルの排除による効果の推定

次に，これらのファイルのうち，ファイル全体が完全同一のファイル (重複ファイル) について調べた，ここでの重複ファイル数には重複の対象となったファイル自体は含めていない．すなわち，例えば ファイル f1, f2, f3 と同じファイルがあった場合，総ファイル数は 3 であり，重複ファイル数は 2 となる．この結果を表 2 に示す．

ファイル数比は，重複ファイルの中における比率であり，総ファイル数比は，調査対象となったすべてのファイルに対する比率となっている．サイズ比も同様である．重複ファイルの出現率は，powerpoint がやや少ないものの，全体としてはファイルの出現率に対する比率は大きくかわらず，どのファイルタイプでも重複したファイルが一定数送られてくることがわかる．また，合計ファイルサイズ比から，単純な重複ファイルに対する重複排除によって 11%程度，実際の消費容量を削減できることがわかる．

5.3 固定位置一致ストライドの重複排除による効果の推定

次に，重複ファイルを除いた状態で，固定位置一致の探索を行った結果を表 3 に示す．サイズ比は固定位置一致内で占める比率を表している．総サイズ比は全てのファイルサイズの合計に対してどれだけの比率を占めているかを表しており，この数字が重複排除の効果を示すことになる．

この結果から，重複ファイルを除いた状態での固定位置一致部分は全てのファイルタイプを合計しても最高でも 2%しかないことがわかる．この時のストライド長は 8KB であるが，この数字は分散保存時のメタデータオーバヘッドを鑑みると現実的なものではない．なぜならば，前述の通りストライド長は消失符号の適用単位に等しいため，たとえばユーザが 8 分割 3 冗長のような構成をとった場合，各ディスクに保存されるフラグメント長は 1KB となるため，メタデータオーバヘッドが無視できないサイズとなってしまう．ユーザが求める冗長構成と容量効率を考えるとストライド長は最低でも 64KB から 128KB 程度が望ましい．

よって，ファイル単位での重複排除を行うことが可能であれば，固定位置一致の重複排除の効果はほとんどないといえる．

表 3 固定位置一致探索の結果: 完全に重複しているファイルを除いた場合，固定位置での重複排除の効果は非常に小さい

ストライド長	総サイズ比
8192	0.020
32768	0.012
65536	0.009
131072	0.007
262144	0.005
524288	0.003

5.4 自由位置一致ストライドの重複排除による効果の推定

次に，重複ファイルを除いた状態で，自由位置一致の探索を行った結果を表 4 に示す．このとき一致したストライドが固定位置一致を含む他の一致部分と重なる場合にはその一致は除外している．すなわち，ここで示されている重複排除可能領域は重複ファイル及び固定位置一致を適用後でも得られる領域となっている．

表 4 自由位置一致探索の結果: 効果はストライド長に大きく影響される

ストライド長	総サイズ比
8192	0.204
32768	0.155
65536	0.127
131072	0.099
262144	0.069
524288	0.048

この結果より，自由位置では固定位置に比べて多くの部分の重複排除が可能となることがわかる．一方で，前述のように現在想定されているストライド長の一つである 128KB では総サイズに対して 10%程度の重複排除が可能となるが，ストライド長が 8KB であれば倍の 20%程度の重複排除が可能となり，重複排除率は強くストライド長に影響されることが分かる．

5.5 提案方式の重複排除による総合実効容量効率

表 5 には，重複ファイル排除，固定位置一致重複排除，自由位置一致重複排除を行った場合の実用量に対する推定実効容量比 [*3] を各ストライド長毎に示した．

[*3] 推定実効容量比 = (総ファイルサイズ / (総ファイルサイズ - (重複ファイルによる重複排除可能サイズ + 固定位置一致による重複排除可能サイズ + 自由位置一致による重複排除可能サイズ)))

表 2 重複ファイルの内訳: ファイル数比，サイズ比はそれぞれ重複ファイルの総数に対する比を示している

ファイルタイプ	重複ファイル数	ファイル数比	総ファイル数比	総バイト数	サイズ比	総サイズ比
Total	796	1.000	0.154	356516787	1.000	0.117
powerpoint	122	0.153	0.024	141062187	0.396	0.046
word	195	0.245	0.038	60162036	0.169	0.020
excel	226	0.284	0.044	36042312	0.101	0.012
pdf	246	0.309	0.048	114316933	0.321	0.037

この結果より，現在想定されているストライド長の一つである 128KB でも，提案方式の重複排除方式で 29%程度の容量効率向上が見込めることがわかる．一方で，仮にストライド長が 8KB であれば 50%以上の容量効率向上が見込めることもわかり，実際のシステムでのストライド長が重複排除による容量効率に大きな影響を与えることが分かった．

表 5 提案方式の重複排除による総合実効容量効率

ストライド長	実効容量
8192	1.517
32768	1.397
65536	1.339
131072	1.286
262144	1.237
524288	1.201

6. 考察

本章では，提案方式がまず 2 章で述べた要求事項を満たしているかについて議論する．

6.1 重複排除を併用しても消失符号の信頼性を維持すること

提案方式では，消失符号の符号化の単位であるストライド毎に重複排除が行われ，また，重複排除時に各フラグメントが他の KVS Server へと移動することはない．よって，提案方式では，重複排除を行っても同一のストライドに属しているフラグメントは，必ず異なる KVS Server へ保存されることを保証することができる．

6.2 自由位置での重複排除を実現すること

提案方式では，保存後に rolling hash を使用することで，自由位置での重複位置を検出することができる．その後，FrontEnd Server がその位置に合わせた形で保存することで，自由位置での重複排除が実現されている．

6.3 容量に対する重複排除性能のスケーラビリティを実現すること

提案方式では，重複排除処理の中で大きな処理能力を必要とする重複位置探索を，IP 接続型の HDD にオフロードしている．IP 接続型の HDD の処理能力は高くないが，大容量のオブジェクトストレージでは多数の HDD があることが期待されるため，総スループットでは大きな能力となる．また，保存容量が非常に大きくなった場合でも，保存容量を増やすためには必ずディスクが必要となり，これは IP 接続型の HDD が増加することを示しており，より重複排除処理能力が向上することとなる．このため，容量に対する重複排除性能のスケーラビリティを実現していると言える．

7. 関連研究

容量単価の優位性をターゲットとしたオブジェクトストレージについては，多くの製品や実装が存在する．オープンソースとして公開されている代表的なものとしては Swift[2], Ceph[19], LeoFS[11] 等が存在するが，LeoFS は消失訂正および重複排除は実装されておらず，Swift, Ceph については消失訂正は実装されているが [1][10]，重複排除は実装されていない．商用オブジェクトストレージとしては Atoms[6] や Himalaya[4] が存在するが，これらも重複排除は実装されていない．

クラウドサービスとしては Amazon S3[3], Google Cloud Storage[8], Microsoft Azure Blob Storage[14] などが代表的と考えられるが，Azure Storage には消失訂正が採用されていることは公表されている [9] が，重複排除については採用されているという記述は無い．Liquid[20] は消失訂正および重複排除を実装しているが，仮想マシンイメージに最適化されており，汎用のオブジェクトストレージとして使うことは難しいと考えられる．また，Hydrastor[5] も消失訂正および重複排除を実装しているが，Hydrastor はバックアップ等を用途ターゲットとしており，スループットが重要視されている中で，インラインでの重複排除を行う．このため，他のオブジェクトストレージと比較してコストを十分に低く押さえることが出来ていない．

8. おわりに

本稿では，新しい，信頼性のある低価格で大容量なオブジェクトストレージの実装方式を提案した．提案式では，複数の物理ディスクに分散されて保存される消失符号の集合を管理しながら重複排除を適用することにより，ディスク故障に対する信頼性を維持する．また，重複排除の判定

を保存処理中と保存後の二段階に分けることで，ユーザへの性能インパクトを抑えながらも，システム全体での精細な重複排除を可能とした．さらに，本設計では，IP 接続型 HDD のような高機能な HDD の利用を想定し，それらに重複位置の探索を分担させることにより，容量の増加に対するフロントエンドサーバの負荷を低減することで，容量に対するスケール性を維持した重複排除を行うことを可能とした．また，この提案方式の実装を行い，想定通りに重複排除が行われることを確認した．加えて，オブジェクトストレージへの保存対象として考えられるオフィス系ドキュメントに着目し，これらが提案方式で保存された場合の重複排除効率をシミュレータを用いて調査した．結果としてストライド長が 128KB の場合であっても実効容量比で 30%程度の向上が見込めることが分かった．一方，ストライド長はオフィス系ドキュメントの保存に対し，実効容量比の向上に大きな影響を与えることが分かり，ストライド長の選択には十分な注意が必要であることがわかった．一方で，今回の結果は個人のメールに添付されてきたファイルに限定されており，例えば複数人のバックアップ等は異なる結果が予想されるため，より広範な調査を行う必要がある．

今後は，本実装を実環境で使用することで，重複排除効率のより詳細な調査と，ストレージとしての性能及び信頼性を調査していく予定である．

参考文献

[1] Openstack Swift. `http://docs.openstack.org/developer/swift/overview_erasure_code.html`.

[2] Openstack Swift. `http://docs.openstack.org/developer/swift/`.

[3] Amazon. Amazon simple storage service (amazon s3). `http://aws.amazon.com/s3/`.

[4] Amplidata. Himaraya. `http://amplidata.com`.

[5] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. Hydrastor: A scalable secondary storage. In *FAST*, Vol. 9, pp. 197–210, 2009.

[6] EMC. EMC Atoms Cloud Storage. `http://www.emc.com/storage/atmos/atmos.htm`.

[7] Roy Thomas Fielding. Representational State Transfer (REST). `https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm`.

[8] Google. Google cloud storage. `https://cloud.google.com/storage/docs/json_api/v1/objects`.

[9] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, Sergey Yekhanin, et al. Erasure coding in windows azure storage. In *USENIX ATC*, Vol. 12, 2012.

[10] Inktank. Erasure Code - Ceph Documentation. `http://docs.ceph.com/docs/master/rados/operations/erasure-code/`.

[11] Leo Project. LeoFS: The Lion of Sorage Systems. `http://leo-project.net/`.

[12] Jun Li and Baochun Li. Erasure coding for cloud storage systems: a survey. *Tsinghua Science and Technology*, Vol. 18, No. 3, pp. 259–272, 2013.

[13] Dutch T Meyer and William J Bolosky. A study of practical deduplication. *ACM Transactions on Storage (TOS)*, Vol. 7, No. 4, p. 14, 2012.

[14] Microsoft. Microsoft azure blob storage. `https://azure.microsoft.com/en-us/services/storage/blobs/`.

[15] Michael O Rabin. *Fingerprinting by random polynomials*. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.

[16] Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial & Applied Mathematics*, Vol. 8, No. 2, pp. 300–304, 1960.

[17] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. Xoring elephants: Novel erasure codes for big data. In *Proceedings of the VLDB Endowment*, Vol. 6, pp. 325–336. VLDB Endowment, 2013.

[18] Seagate. Seagate Kinetic Open Storage Platform. `http://www.seagate.com/www/kinetic/`.

[19] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pp. 307–320. USENIX Association, 2006.

[20] Xun Zhao, Yang Zhang, Yongwei Wu, Kang Chen, Jinlei Jiang, and Keqin Li. Liquid: A scalable deduplication file system for virtual machine images. *Parallel and Distributed Systems, IEEE Transactions on*, Vol. 25, No. 5, pp. 1257–1266, 2014.