Vol.2019-HPC-170 No.29
2019/7/26

オブジェクトストレージに対するクライアント協調型の重複排除方式適応への試み

七海 龍平[1,a)]　伊藤 恵[1]

概要：近年，動画ファイルや画像データを始めとする非構造化データの増大に伴い，従来のファイルストレージの I/O にボトルネックが生じるようになった．こうした，急激に増大するデータに対する課題を解決する手段としてオブジェクトストレージと呼ばれるストレージシステムが注目を集めている．オブジェクトストレージでは，データをオブジェクトという単位でフラットな空間に保存することで安価に大容量なストレージを構築することが可能となった．一方でストレージの容量効率を高める重複排除技術も注目を集めているが，重複排除の処理には大量のメモリと処理のための時間が必要となる．そのため，低コストかつ大容量を特徴とするオブジェクトストレージにおいて通常の重複排除を適応させることは難しい．本研究では，クライアント協調型の重複排除方式と提案することで重複排除にかかる処理負荷の一部をクライアント側へオフロードし，オブジェクトストレージに対する重複排除の適応を試みる．本稿では，提案方式実現へ向けインライン方式での重複排除機能を搭載した簡易オブジェクトストレージを実装し．重複排除の負荷実験を行った．結果として実験環境下にてデータを固定長サイズ読み込み，そのハッシュを計算する処理が，データをストレージに保存する処理と比較した時に，最大で約 130 倍メモリを消費していることが分かった．

An Attempt to Adapt the Client Cooperation Type Deduplication Method to Object Storage

1. はじめに

近年，世界のデータ量は急激に増加しており，調査会社である IDC 社の調べによると 2018 年の世界のデータ量は 33 Zettabytes であったが，2025 年には 175 Zettabytes まで増加すると言われている [9]．データには構造化データと非構造化データと呼ばれるものが存在する．世界のデータ量の 90 %は非構造化データあるとされており，そのデータ量は今後ますます増加すると予測されている [4]．

構造化データのような小さなデータに対する Read / Write は，従来のディレクトリ構造を用いたファイルストレージが非常に有効であった [2]．しかし，動画ファイルや画像データなどの非構造化データの増大に伴い，ファイルストレージの I/O にボトルネックを生じさせるようになった．こうした，急激に増大するデータに対する課題を解決するストレージとしてオブジェクトストレージと呼ばれるストレージシステムが注目を集めている．オブジェクトストレージはデータを，ファイル単位やブロック単位ではなくオブジェクトという単位で扱う．このオブジェクトは，対象データの本体およびメタデータによって構成されている．このデータ構造によって，オブジェクトストレージではディレクトリのような階層構造ではなくフラットな空間にデータを保存することを可能としている．そのため，ファイルストレージに比べてデータの移動が容易でありストレージのスケールアウトが非常に柔軟に行えるため，大容量なストレージを安価に構成することが可能である．

一方で，増大するデータへの対処法として重複排除と呼ばれる技術も注目を集めている．重複排除はデータのサイズを圧縮する技術である．データを保存するためのストレージの容量単価を削減することにとても有効であり，多

[1] 公立はこだて未来大学大学院
Graduate School of Systems Information Science Future University Hakodate, Hakodate, Hokkaido 041–8655, Japan
[†1] 現在，公立はこだて未来大学大学院 システム情報科学研究科 高度 ICT 領域
Presently with Graduate School of Systems Information Science Future University Hakodate
[a)] g2118028@fun.ac.jp

くのファイルストレージで採用されている．この重複排除では，ストレージ内に保存されるデータのバイナリに同じ文字列を持った箇所を特定し，存在した際には，片方へオリジナルデータへのアクセス情報を付与し，バイナリ自体は削除することでデータの圧縮を実現する．しかし，重複排除に必要な重複位置の特定には大容量のメモリと処理時間が必要となるため，システムのハードウェアコストを引き上げる．このため，安価かつ大容量を特徴とするオブジェクトストレージシステムで重複排除を適用させることは難しいとされている [1]．

本研究では，データ保存量や処理負荷の面から見た際に優れているインライン方式での重複排除をオブジェクトストレージに適応させるために，クライアント協調型の重複排除を提案する．ストレージのみでインライン方式による重複排除操作を処理した場合，本来のストレージの目的であるデータの保存のボトルネックになる可能性が高い．そのためストレージで処理する操作の一部をクライアントと協調して行い負荷を分散させることで，この問題の解決を目指す．本提案では，インラインでの重複排除適応を目的とすることから，重複位置の特定コストを抑えることに注力するため，固定ブロックレベルでの重複排除方式を採用する．本稿では，重複排除技術について述べた後に，提案の詳細とその実現へ向けて行った重複排除の実験結果について述べる．

2. 重複排除技術

ファイルデータの保存における重複排除では，複数のファイルにてバイナリ全体または一定のウィンドウサイズでデータを読み込み，それらを比較することで重複している箇所が存在しているかを判別する．バイナリの重複を検出した際は一方をオリジナルデータとし，もう片方にはオリジナルデータへのアクセス情報のみを保存することでデータ量を削減する．重複排除の方式には，一般的に 3 種類の重複箇所検出方式が存在する．本章では，重複排除の方式およびその適応箇所について述べる．

2.1 重複排除方式

重複排除の方式には 3 種類があり，各方式は一長一短の特徴を持っている．それぞれの方式の仕組みおよび特徴を本節にて説明する．

- ファイルレベル方式
  ファイルレベルの重複排除とは，対象のファイル同士のバイナリが完全一致した場合にのみ重複排除を適応する方式である．この方式の比較回数は，保存されているファイルの数（オブジェクトストレージの場合はオブジェクトの数）になるため，比較にかかる処理のコストは最も低く，処理速度も最も速いことが期待できる．しかし，その一方でファイル間に 1 ビットの違いが生じただけで重複排除を行うことができなくなるため，重複排除を検出することは難しい．図 1(1) ではファイルレベルの重複排除を表している．

- 固定長ブロックレベル方式
  ファイルレベルではファイル間に 1 ビットの違いが生じただけで重複排除を行うことができないことを説明した．この問題を解決した重複排除方法が固定長ブロックレベルでの重複排除である．この方式では，ファイルをある長さに分割し，その分割した単位で重複排除を行う方式である．図 1(2) は固定長ブロックレベルの重複排除を表している．固定長のブロックに分割することで，図のように重複が生じている部分と重複していない部分を分けて見ることが可能となる．したがってファイルレベルの重複排除よりも多くの重複箇所を検出することが期待できる．しかし，この方式にはファイルの先頭にブロックサイズとは異なるサイズのデータが挿入された時に重複排除が検出できなくなってしまうという大きな問題が存在する．先頭にブロックサイズと異なるサイズのデータが挿入された場合，ブロックに分割する箇所が大きく変わるため実際には重複している箇所であっても上手く検出できるように分割することができなくなるためである．

- 可変長ブロックレベル方式
  固定長ブロックレベルの問題を解決したものがこの方式である．可変長ブロックレベルでの重複排除では，ファイルを分割する大きさを柔軟に変更することができるため先頭にどのようなデータが挿入されたとしても重複箇所を検出することが可能となる．しかし，可変長ブロックレベルの重複排除にも問題は存在する．この方式では，ファイル内の自由な位置での重複箇所を検出するために，対象となるファイルを固定のウィンドウサイズで読み込み，重複しているかどうかをウィンドウを 1 バイトずつずらしながら検証する必要がある．そのため上記 2 つの方式に比べ莫大な計算量が必要となる．

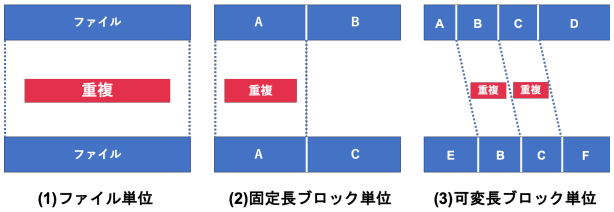

図 1 重複排除の方式

2.2 重複排除の適応タイミング

本節ではストレージへのデータ保存処理のフロー内の，どの場面で重複排除を適応させるかについて 3 つのタイミング方式を説明する．

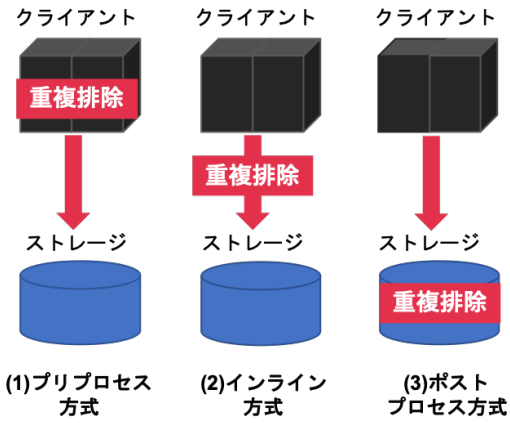

図 2 重複排除の適応箇所

- プリプロセス方式
  プリプロセス方式は，ストレージへデータを送るクライアント側で重複排除を行う方式である．図 2(1) はプリプロセス方式の処理を表している．この方式のメリットとしては，あらかじめクライアント側で重複排除を行うことでデータ転送量を削減することが可能となる点である．デメリットとしては，重複排除のような重い処理を考慮していないクライアント側に対して大きな負荷がかかる可能性がある点が挙げられる．また，一般的にこの方式での重複排除では，新たに保存するデータとストレージに保存されている全てのデータに対して重複排除を行うグローバルな重複排除ではなく，クライアントが保持している一部のデータに対して重複排除を行うため検出効率が低くなる可能性は高い．

- インライン方式
  インライン方式の重複排除は，ストレージへデータを送る処理中にストレージ側で重複排除を行う方式である．図 2(2) はインライン方式の処理を表している．この方式のメリットとしては，プリプロセスとは違いストレージ側で処理を行うためクライアントに負荷がかからないという点が挙げられる．また，全てのデータが保存されているストレージ側で重複排除を行うため，グローバルな重複排除を行うことができる．デメリットとしては，データの転送中に重複排除を行うため，重複排除処理がボトルネックとなりデータの転送性能が低くなる可能性が高い．

- ポストプロセス方式
  ポストプロセス方式の重複排除は，ストレージへデータを一旦保存した後に任意のタイミングで重複排除を行う方式である．図 2(3) はポストプロセス方式の処理を表している．この方式のメリットは，インライン方式にてデメリットであったデータ転送中に重複排除処理がボトルネックとなってしまう問題を解決していることである．データ転送性能を維持した状態でのグローバルな重複排除を実現している．しかし，上記 2 つと比べた時に一旦データを重複排除しない状態で保存するため，その分のデータ保存領域が必要となる．

3. 提案方式

前章で述べたように重複排除の各方式には一長一短の特徴があるためどれかの方式が最も優れているとは一概に言えないが，データ保存量の効率や処理負荷のバランスを考慮した場合にはインラインでの重複排除が優れていると考えることができる．しかし，インライン方式での重複排除では重複排除処理がデータ保存のボトルネックとなる可能性が高い．そこで本研究では安価に大容量なストレージを構築できるオブジェクトストレージに対してインライン方式での重複排除を適応させることを目的とし，クライアント側へ重複排除処理の一部をオフロードすることを目指す．提案の詳細は次節以降にて説明する．

3.1 提案方式概要

本提案では可能な限り重複排除処理の負荷を軽減させるために，固定長ブロックレベル方式の重複排除を採用する．また，具体的にクライアント側へオフロードさせる処理としては，

- データの分割
- ブロックのハッシュ計算
- 重複箇所データの書換え

の 3 つの処理を想定している．図 3 を元に処理の流れについて説明する．従来のインライン重複排除方式では重複排除にかかる全ての処理をストレージ側で行なっていた．本提案の処理では，まずクライアント側で転送予定のデータを固定のウィンドウサイズに分割する．その後，各ブロックのバイナリのハッシュ値を計算し，計算したハッシュ値をストレージ側へ送信する．ストレージ側では受け取ったハッシュ値をブルームフィルタ [3] にかける．ブルームフィルタはデータのまとまりを表現するデータ構造であり，ある要素がそのまとまりの中に存在するかどうかを少ない計算量で効率よく判別することができる．ブルームフィルタにかけて重複しているデータを検出できた場合には，その

重複箇所に関する情報をクライアント側へ返す．また，重複箇所が存在しなかった場合には存在しないという情報をクライアントへ返す．本提案が実現した際には，ストレージ側からデータの分割やブロックのハッシュ計算をオフロードすることによる処理の軽減と，データ転送前に重複箇所を特定することができることからデータ転送量の削減を期待することができる．

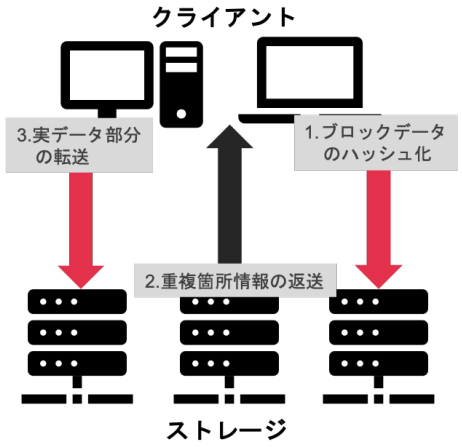

図 3 提案方式概要

3.1.1 ブルームフィルタ

ブルームフィルタとは，1970 年に Burton H. Bloom が発表した空間効率の良いデータ構造であり，ある要素が要素のまとまりの中に含まれているかを判定するために用いられている [3]．ブルームフィルタでは偽陽性があるが，偽陰性は存在しない．そのため，一定の確率でまとまりの中に入っていない要素を入っているものと判別してしまう可能性がある．しかし，この偽陽性が発生する確率は調整することが可能である．ブルームフィルタは m[bit] の配列と，フィルタにかける予定の n 個のデータ，k 個のハッシュ関数から成り立っている．この m と n，k を用いた式 (1) から偽陽性の発生確率 p を導き出すことができる．提案方式で採用した固定長ブロック方式の重複排除では，このブルームフィルタによりデータの重複を検出する．

$$p = \left(1 - e^{-\frac{kn}{m}}\right)^k \tag{1}$$

3.2 実装準備

提案方式を実装する前に，簡易的なインラインでの重複排除を実現した簡易的なオブジェクトストレージを仮実装する必要があると考える．オブジェクトストレージを仮実装し，そのパフォーマンスやメモリ・CPU といったリソースの使用量を確認することで，提案方式の実装時における課題を洗い出すことや，クライアント側へオフロードする処理の負荷を事前に調べることができると考えたためである．オブジェクトストレージへ重複排除を適応させた例は少なく，小さな処理の単位でのパフォーマンスに関する情報を得ることが難しい．このような背景から，一旦簡易ストレージを仮実装したシステムを用いて実験を行い，その後に提案方式の詳細を検討していく．

4. 仮実装と実験

本章では提案方式実装に向けた仮実装と，そのシステムを用いた実験の詳細について述べる．実験の結果と考察については次章にて述べる．

4.1 概要

実験を行うために実装したオブジェクトストレージシステムでは，インラインによる重複排除とデータの保存機能を実装している．この実装はインライン重複排除にかかる個々の処理負荷の調査を行うためのオブジェクトストレージの実装を目的としていることから，単一のノードでのアーキテクチャ設計となっている．次節以降に，実装と実験について詳細に述べる．

4.2 実験環境

実験環境の詳細について述べる．プロセッサには，Intel Core i7-8700 CPU@3.20GHz，6 Core，12 Thread を用いた．また，16GB RAM および 512GB SSD を搭載したデスクトップ型 PC にて実験を行った．PC では OS として，Ubuntu 18.04.2 LTS が動作している．

4.3 仮実装

オブジェクトストレージの仮実装には，OSS で公開されている MINIO[6] というオブジェクトストレージの実装にも使われている Go 言語を用いた．処理の流れとしては，保存するデータを選択し，そのデータが固定長ブロックに分割される．分割されたブロックはそれぞれ MD5[8] と呼ばれるハッシュ関数を用いてハッシュ値を計算した後に，そのハッシュ値を用いてストレージ内のブロックに重複しているものが存在していないかの調査を行う．重複したブロックの調査にはブルームフィルタを用いる．ブルームフィルタによって重複したブロックが検出された場合には，データをレストアするためのオリジナルのデータのブロックレシピ情報を書換える．データのブロックレシピについて仮実装では，別途 KVS サーバーを設け保存した．このオブジェクトストレージではファイルをオブジェクトとして受け付けるが，最終的にはブロックレベルに細かく分けられた状態で既存のファイルシステムへ保存される．その時に利用されるファイル名がブロックのハッシュ値となっているためデータを保存する際に自動的に重複排除が行われる仕組みとなっている．

4.4 実験詳細

実装したストレージシステムを用いて行った実験の詳細について述べる．本実験ではストレージシステムにランダムに生成された 1MB，10MB，100MB，1000MB のバイナリデータを保存し，その過程で行われた重複排除の 3 つの処理について，3 つの項目を監視し，それぞれのデータについて 20 回保存処理を行い各値の平均値をそれぞれの結果とした．本実験における固定長ブロックのサイズは 4KB とし，ブルームフィルタにおける各パラメータに関しては 500GB のデータを保存した時の偽陽性の確率が 0.0001% となるようにフィルタの大きさ (m) を調整した．監視項目については下記の，

- メモリ使用量 MB)
- CPU 時間 (秒)
- 処理時間 (秒)

の 3 つの項目を対象として実験を行った．監視対象とした処理についても下記にまとめる．

4.4.1 監視対象とする処理

- ハッシュ計算
  保存するデータを固定長ブロックに分けて MD5 によるハッシュ計算処理であり，提案方式においてクライアント側へオフロードする想定の処理である．
- ブルームフィルタ
  ブルームフィルタへの要素追加および要素の存在判定の処理である．処理の順番としては先に要素の存在確認を行い，もしも要素がまとまりの中に存在していなかった場合には要素を新たに追加する．
- ブロックデータ保存
  保存するデータを固定長ブロックごとに MD5 で計算したハッシュ値をファイル名としてファイル保存する処理である．

5. 実験結果と考察

本章では実験結果と考察について述べる．先ずは監視項目ごとの結果を表を元に説明する．計測結果は小数第三位までを表示する．結果を説明した後にその考察について述べる．

5.1 実験結果

表 1 は，各処理のメモリ使用量の平均値を表した表である．表から保存するデータサイズに比例して処理内部で使用しているメモリの量が増えていることがわかる．データサイズが 10 倍になるごとに消費メモリ量も約 10 倍ずつ増加している．また，最もメモリを消費する処理が固定長ブロックのハッシュ計算であることがわかる．

表 1 メモリ使用量 (MB)

	1MB	10MB	100MB	1000MB
ハッシュ計算	5.369	59.598	601.708	5982.402
ブルームフィルタ	2.489	24.944	249.381	2487.421
ブロックデータ保存	0.051	0.472	4.691	48.003

表 2 は，各処理の CPU 時間の平均値を表した表である．表から保存するデータサイズが大きくなると，CPU 時間が長くなることがわかる．しかし，処理ごとの CPU 時間の差異は少ないことが見てとれる．1000MB の時のみハッシュ計算の CPU 時間が他の 2 つの処理と比べて早い．

表 2 CPU 時間 (秒)

	1MB	10MB	100MB	1000MB
ハッシュ計算	0.014	0.072	0.550	4.771
ブルームフィルタ	0.009	0.060	0.523	5.219
ブロックデータ保存	0.011	0.060	0.525	5.551

表 3 は，対象の処理全体の時間の平均値を表した表である．表から保存するデータサイズが大きくなると，処理時間もほぼ比例して長くなることがわかる．ただし，表 2 の CPU 時間同様に処理の違いによる値の差異は大きくない．表 2 と比べた時に 100MB のデータ保存時のブルームフィルタとブロックデータ保存の処理および，1000MB のデータ保存時のブロックデータ保存処理の 3 パターン以外では全体の処理時間よりも CPU 時間の方が長いことがわかる．

表 3 処理時間 (秒)

	1Mb	10MB	100MB	1000MB
ハッシュ計算	0.007	0.057	0.541	4.522
ブルームフィルタ	0.008	0.068	0.518	4.196
ブロックデータ保存	0.005	0.062	0.521	5.676

5.2 考察

本節では実験結果に対する考察を述べる．まず，メモリ使用量について考察する．ハッシュ計算とブルームフィルタでは，ハッシュの計算を内部で行う似たような処理であるにも関わらず，約 2 倍のメモリ使用量の差が出たことに関しては不要なメモリコピーがハッシュ計算にて行われていたことが考えられる．使用した Go 言語の言語仕様について調査した結果，ハッシュ計算時にファイルからバイナリを byte 型のスライスで読み取りそれを string 型にキャストする際にメモリコピーが発生していたことが分かった．型のキャストに関してはさらに調査を進め対処法を施すことで，メモリ使用量の削減が出来る可能性がある．次に CPU 時間について，オブジェクトストレージでは本実験で扱った最大データサイズの 1000MB 以上のデータが大量に投入される可能性が大いに考えられるため，データサ

イズによって実験結果のように CPU 時間が増えることは，本処理がデータ保存のボトルネックになる可能性が考えられる．CPU 時間の増加の原因としては保存データサイズが大きくなることによりハッシュ処理の計算量，ブルームフィルタへの要素の確認回数，保存ブロック数の増加などが考えられる．現状考えられる対処としては，処理の並列化による負荷分散が考えられる．最後に処理時間について考察していく．処理時間では表 3 の結果で述べているように，ほとんどの数値が表 2 の CPU 時間よりも短くなっている．これは，処理の実行に複数のプロセッサを利用しそれらの占有時間の合計が CPU 時間となっているためだと考える．今回実装に利用した Go 言語ではバージョン 1.5 以降では自動で実行環境のプロセッサ数を判別する仕様になっている．そのため，実際の時間では CPU 時間と処理時間には数値以上の差があることが考えられ，CPU 処理の他のディスク I/O などの処理が発生していたことが考えられる．

6. 関連研究

オブジェクトストレージに重複排除を適応させるための研究として，石山ら（2016）は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を行った [1]．石山らが発表を行った頃は重複排除機能を搭載したオブジェクトストレージシステムは知られていなかった．その理由としては，低コスト大容量を強みとするオブジェクトストレージシステムでは重複排除の負荷を処理するために計算リソースを拡張することが難しいからだ．そこで，石山らは IP 接続型の HDD を利用しコストを削減しつつ，従来の手法ではストレージの CPU にかかっていた処理負荷の一部を HDD に分散させることで，重複排除の実現を目指した．結果として，提案方式による自由位置での重複排除効率を，シミュレーションを用いて解析し，提案方式によって実容量比で約 30 %の向上が得られることを示した．

また，Oh ら（2019）は分散ストレージのための新たなグローバル重複排除方式の提案を行った [7]．実際の提案手法を Ceph と呼ばれる OSS に実装しその効果を検証した．Oh らの提案手法では，オブジェクトストレージのフラットなデータプールを，クライアントから送られてきたばかりのデータを一時的に保存するメタデータプールと，その後重複排除を行う Chunk のみを保存するチャンクプールに分けることでポストプロセスでの重複排除を実現している．また，クライアントから呼び出される回数が多いホットデータはメタデータプールにオブジェクトの本体ごと保存し，プールを 2 つに分けることによってデータレストア時に発生する I/O の遅延の減少を図っている．

7. おわりに

本論文では，オブジェクトストレージへインラインでの固定長ブロック方式の重複排除を適応を目的として，クライアント協調型の重複排除方式の提案を行った．提案方式の実現へ向けた簡易オブジェクトストレージの実装を行い，実装した仮システムを用いたインラインでの重複排除の負荷実験とその結果と考察について述べた．実験の結果としては重複排除に関わる 3 つの処理において保存するデータサイズが大きくなると，ほぼ比例してメモリ使用量，CPU 時間，処理時間が増加することが分かった．また，重複排除のような大量にハッシュ計算や型のキャストを実行するプログラムでは，あらかじめ言語で用意されているハッシュ関数や型のキャストを使用するとメモリの無駄が発生する可能性が示唆された．提案方式におけるクライアント側へハッシュ計算および重複データの操作をオフロードする方針は有用性が見られそうだが，実装方法に関しては今回の実験で得られた結果を元に再考していく必要があると考える．

参考文献

[1] 石山, 政浩 and 田所, 秀和 and 松崎, 秀則：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, マルチメディア，分散協調とモバイルシンポジウム 2016 論文集, Vol. 2016, pp555-564, 2016.
[2] 富士通株式会社：オブジェクトストレージとは (online)，入手先 ⟨https://www.fujitsu.com/jp/products/computing/storage/lib-f/tech/beginner/object-storage/⟩(2019.06.23).
[3] Bloom, Burton H.：Space/Time Trade-offs in Hash Coding with Allowable Errors，1970-ACM, Commun. ACM，Vol.13, No.7, pp.422-426 (1970).
[4] David R, John G, John R：The Digitization of the World From Edge to Core(online)，入手先 ⟨https://www.seagate.com/www-content/our-story/trends/files/idc-seagate-dataage-whiwhitepa.pdf⟩ (2019.06.23).
[5] Li, Yan-Kit and Xu, Min and Ng, Chun-Ho and Lee, Patrick P. C. : Efficient Hybrid Inline and Out-of-Line Deduplication for Backup Storage，ACM Transactions on Storage, Vol. 2014-ACM, Vol.11, No.1, pp. 1-21, 2014.
[6] MinIO, Inc.：MINIO (online)，入手先 ⟨https://min.io/⟩ (2019.06.23).
[7] Oh, M., Park, S., Yoon, J., Kim, S., Lee, K. W., Weil, S., ... Jung, M. : Design of global data deduplication for a scale-out distributed storage system，ICDCS 2018, Vol. 2018-July, pp. 1063-1073, 2018.
[8] Rivest, R.：The MD5 Message-Digest Algorithm，1992-RFC, IETF Network Working Group, (1992).
[9] Veritas：DATA GENOMICS INDEX 非構造化データの専門家による，ストレージ環境の真の構成内容に関する調査レポート 2017(online)，入手先 ⟨https://www.veritas.com/content/dam/Veritas/docs/reports/V0479_Data-Genomics-Index-Report-JA.pdf⟩ (2019.06.23).