情報処理学会論文誌 Vol.63 No.2 504–514 (Feb. 2022)

検索クエリに配慮した配置による分散ログ検索の高速化

小山 智之[1,a)] 串田 高幸[1,b)]

受付日 2021年5月11日, 採録日 2021年11月2日

概要：現在，Web サービスは広く利用されており，利用者の増加とともにアクセス数が増加している．アクセス数が増えるにつれ，サーバから生成されるアクセスログの件数は大規模になる．大規模なログの管理手法の 1 つである分散型は，ログを複数のノードに分散配置することで検索時のディスク I/O にともなう負荷を分散している．分散型ログ管理の課題は，検索時のログへのアクセス傾向が十分に考慮されないためディスク I/O が一部のノードへ偏り，検索の応答時間が高速でないことである．本研究では，Web アクセスログを対象とした検索クエリを想定した分散ログ配置により，検索の応答時間を削減する手法を提案する．具体的には，検索で発行されるクエリを想定し，それをもとにログに含まれる属性（Path や Method，ノード名，日付）から特徴量を取り出し，ログの分割と再配置を行った．これにより，検索クエリにより偏るディスク I/O を分散させ，検索の応答時間を削減した．評価では，86,400,000 件のログを 13 台のノード上に 3 種類の手法（提案配置，ノード配置，時系列配置）により配置し，発行する検索クエリを変えながら検索の応答時間を比較した．その結果，提案配置は他の配置に比べ最大 32 秒の短縮を行った．

キーワード：ログ管理，情報検索，分散システム

Fast Distributed Log Search by Query Aware Placement

Tomoyuki Koyama[1,a)] Takayuki Kushida[1,b)]

Received: May 11, 2021, Accepted: November 2, 2021

Abstract: Web services are widely used these days. The total number of visits is also increasing with the increase in the number of users of network services. The number of logs can be large scale with increasing the number of accesses. Distributed placement is a large-scale log management method that can reduce Disk I/O by putting logs into distributed nodes. However, the placement has a problem that it could not respond to search queries fastly because of unbalanced Disk I/O per node. This research proposes the distributed log placement that reduces search response time in the web access log by query prediction. The method partitions and reallocates logs based on the search query parameters (Path, Method, NodeName, Date) included in the log entries. The system enables to reduce the response time in log search by balancing the Disk I/O per node. The evaluation compares search response time among the proposed method, node-based method and time-based method. The result shows the proposed method is 32 seconds faster than other methods.

Keywords: log management, information search, distributed systems

1. はじめに

1.1 背景

ログは，ソフトウェアが動作する過程で出力されるメッセージである．ログには，ソフトウェアの動作にともなうイベントが詳細に記録される [1]．そのため，システムの信頼性を確保するために開発者やシステム管理者が広く利用する [2]．従来からログは，システムのトラブルシューティングに使用されている．たとえばシステムで障害が発生すると，システム管理者はログをもとに過去にシステムで起きたイベントを解析し，障害の原因を特定する．システム管理者はログをシステムの動作の把握に役立てる．

ログの保存期間とシステムを構成する機器の台数が増

[1] 東京工科大学大学院バイオ・情報メディア研究科コンピュータサイエンス専攻
Graduate School of Computer Science, Tokyo University of Technology, Hachiouji, Tokyo 192–0914, Japan
a) g21210247f@edu.teu.ac.jp
b) kushida@acm.org

えるにつれ，システム全体でのログの件数は増加する．たとえば，ネットワーク機器の展示会である Interop Tokyo 2017 における ShowNet では，3 日間で 600 台の機器から 1 分あたり 50,000 件のログが出力された [3]．ログの件数が増えるにつれ検索にともなうログの走査によりディスク I/O が増加する．ディスク速度の増加は，CPU 速度やメモリ容量，メモリ速度の増加に比べ遅い [4]．そのため，ディスク I/O が性能向上のボトルネックになる．

増加するログの管理手法の 1 つに集中型がある．集中型はログを 1 台のサーバまたは複数台によるクラスタサーバに集約する手法である．Spark や Elasticsearch に代表されるビッグデータ基盤を構築するミドルウェアは，複数台のマシンを組み合わせたクラスタを作成することで 1 台のマシンに集中する負荷を分散させデータを管理する．

Spark は syslog-ng（RAM: 256 [MB]）に比べ，クラスタを構成するマシン 1 台あたりに高性能なハードウェア（RAM: 8 [GB]）を必要とする[*1,*2]．同様に Elasticsearch も高性能なハードウェア（RAM: 16 [GB]）[*3]を必要とする．こうしたミドルウェアは，ログを出力するソフトウェアが動作するノードとは別に，ログの管理を目的とした専用の高性能なノードを必要とする．頻繁に発生しない障害への対処のために，ログを管理する専用ノードを運用することは，作業工数に比べ得られる効果が小さい．

Google Cloud の Cloud Logging をはじめとする Logging as Service（LaaS）は，クラウド上にログを管理するプラットフォームを提供する[*4]．LaaS は，サーバの構築や運用をサービス提供者が行うため，利用者のログサーバの構築や運用の作業を削減する．LaaS はログ件数が増えるほど，その利用料金は高くなる．これはログ件数が増えるほど，高性能なハードウェア資源（例：大容量のディスク）を必要とするためである．LaaS では数万件に及ぶログの管理をサポートするため，サービス提供者は高性能なノードを用意する．LaaS は，Spark や Elasticsearch における高性能なノードの運用工数が利用料金に変わっただけである．頻繁に発生しない障害への対応のために利用料金を支払うことは，費用に対して得られる効果が小さい．

分散型のログ管理は，ログを複数ノードに配置する．これによりログへのアクセスにともなうディスク I/O を分散させ，検索の応答時間を短縮している．分散型のアーキテクチャの 1 つに scatter-gather パターンがある [5]．このパターンではデータを複数のノードに分散させることにより，データへのアクセスにともなうディスク I/O が分散される．これにより検索の応答時間が高速になる．この手法では，すべてのノードの検索にかかる時間のうち最大値が全体の検索時間になる．そのため，検索にともなうディスク I/O が均等になるようデータを配置する必要がある．

1.2 ユースケース

ここでは，ログ検索のユースケースとして Web サービス事業者（EC サイト）を想定する．この企業では Web サービスを通じた商品の販売により収益をあげている．仮にサービスで障害が発生すると，商品の販売ができず損失が発生する．以降では，Web サービスにおいて発生した障害の原因調査を想定する．

障害発生から調査までの過程を**図 1** に示す．まず，利用者が Web サービスへアクセスするとエラーが表示される．利用者は，カスタマーサポートにサービスの動作について問い合わせる．次にカスタマーサポートは，システム管理者へ詳細な調査を依頼する．依頼を受けたシステム管理者は，原因を調査するため Web サイトのアクセスログを検索する．インターネットサービス事業者である IIJ では，自社のシステムでの障害の検知から利用者への通知までを，SLA により 30 分以内と定めている[*5]．ここで障害報告の作成やインシデントのエスカレーションに 10 分かかると想定する．このとき，ログ検索による調査にかかる時間の要件は 20 分以内である．1 回の検索クエリでかかる時間は，システムにおける検索処理とシステム管理者のログ閲覧とする．システム管理者は障害の発生した正確な時刻を調べるため，検索クエリにおいて時間帯の条件を指定した検索をする．これにより検索結果に含まれるログ件数を削減し，システム管理者が閲覧するログ件数を削減する．検索クエリでの時間帯の絞り方は，二分探索により狭める場合を想定する．直近 7 日間のログから障害の発生した 1 時

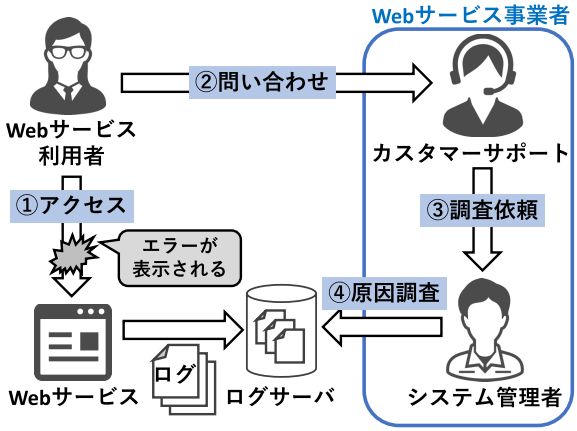

図 1 ユースケース：シナリオ
Fig. 1 Usecase: scenario.

*1 https://www.syslog-ng.com/technical-documents/doc/syslog-ng-store-box/5.0.2/installation-guide/5
*2 https://spark.apache.org/docs/latest/hardware-provisioning.html
*3 https://docs.paloaltonetworks.com/cortex/cortex-xsoar/5-5/cortex-xsoar-threat-intel-management-guide/migrate-indicators-to-elasticsearch/elasticsearch-sizing-requirements.html
*4 https://cloud.google.com/logging
*5 https://www.iij.ad.jp/svcsol/sla/

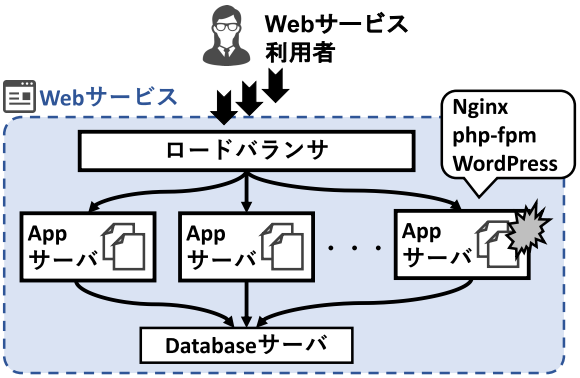

図 2 ユースケース：Web サービスアーキテクチャ
Fig. 2 Usecase: Web service architecture.

ソースコード 1 ログの例

```
23.92.36.1 - - [27/Oct/2020:20:23:11 +0900] "GET /
    mypage HTTP/1.1" 200 612 "-" "Java/1.8.0_265"
```

間のログに到達するまでに 8 回の検索クエリ（7 日，3.5 日，1.75 日，21 時間，10.5 時間，5.25 時間，2.63 時間，1.31 時間）が発生する．調査にかかる時間は 20 分以内であるため，1 回の検索（検索クエリ発行，検索結果の参照）あたり 20/8 = 2.5 分で終える必要がある．したがって，1 回の検索クエリあたり 1 分以内に検索結果を返す必要がある．

ユースケースであげた Web サービスのアーキテクチャを**図 2** に示す．このアーキテクチャは，ロードバランサと App サーバ，Database サーバの 3 種類から構成される 3 層型を採用している．インターネットを経由したユーザのトラフィックは，ロードバランサで受信される．ロードバランサは，複数台ある App サーバへトラフィックを分散させる．ログの例を**ソースコード 1** に示す．このログ形式は Nginx をはじめとするミドルウェアで採用されている．ログには，アクセス元の IP アドレス（例：23.92.36.1）やアクセス日時（例：27/Oct/2020:20:23:11 +0900），アクセス先のパス（例：/mypage）が含まれる．以降では，**ソースコード 1** に示すログを対象としたログの検索を想定する．このログは App サーバへのアクセスが記録されている．Web サーバへのリクエスト数は，想定した利用者数と近い規模のログ（インディアナ大学が公開）をもとに平均 1,000 [request/sec] とした [6]．図 2 のアーキテクチャにおける App サーバの障害を想定する．このとき，カスタマーサポートから報告を受けたシステム管理者は，App サーバで生成されたログを検索する．ログは，App サーバごとにマシン内に保存される．ユースケースにおいてシステム管理者はログを検索するために次の 2 種類のクエリを発行する．

クエリ 1. 障害の発生した時刻の絞り込み：すべてのログの詳細な分析は時間がかかる．そのため，システム管理者は最初に広い期間のログから障害の発生した時刻の見当をつける．たとえば，**ソースコード 2** の検索クエリでは，絞り込む時間帯を変更している．これにより，障害の発生した時刻の見当をつけることで，必要のない期間のログへの検索を削減する．

ソースコード 2 障害発生の時刻を絞り込むクエリ

```
DateRange="2020/03/09 - 2020/03/16" & StatusCode>=400
    & Path=/mypage & Method=GET
DateRange="2020/03/16 - 2020/03/23" & StatusCode>=400
    & Path=/mypage & Method=GET
```

ソースコード 3 障害発生のノードを絞り込むクエリ

```
DateRange="2020/03/29 - 2020/03/28" & StatusCode>=400
    & Path=/mypage & Method=GET & Node=ap3
```

クエリ 2. 障害の発生したノードの絞り込み：クエリ 1 により障害の発生時刻の見当をつけた後，詳細に障害の発生ノードを特定する．たとえば，ap3 ノードのログに絞ってログを検索する場合，システム管理者は**ソースコード 3** のクエリを発行する．

1.3 課題

scatter-gather パターンによるログの分散配置は，集中配置に比べログ検索時の App サーバへのディスク I/O が分散される．そのため，ログ件数や期間が増加した場合も検索時の応答時間が集中配置に比べ高速である．しかし，分散配置ではログデータへのアクセスとログデータの配置が偏ると一部のノードへディスク I/O が集中する課題がある．これは，ログデータのノードへの配置と実際のログデータへのアクセス傾向に乖離があるためである．つまり，ログの分散配置における検索の応答時間の高速化には，ログデータへのアクセス傾向に着目したログデータの配置を決定する方法の提案が必要である．ログを分散配置する手法には，時系列で分割し配置する方法やノードごとに配置する手法がある．これらの手法では，期間を絞った検索やノードを絞った検索において応答時間を削減している．しかし，これら手法はログ検索で発行されるクエリの特徴への配慮が十分ではない．たとえば，ノードごとに配置する手法では，ノードを絞り込んだ検索クエリが低速である．一方，それ以外の特定のノードに絞った検索では，ノードに負荷が集中する．そのため，検索クエリごとに応答時間にばらつきが生じる．検索の応答時間の削減には，ログデータを検索により取り出す際の特徴に配慮した配置の決定が必要不可欠である．

1.4 各章の概要

2 章では，本研究と関連した既存研究を取り上げる．本研究の提案を 3 章で述べる．4 章では，実装と実験環境について述べる．評価とそれに基づく分析を 5 章で述べる．

6 章では，評価と分析をもとに議論をする．最後に，本論文の取り組みと貢献を簡潔に述べる．

2. 関連研究

ネットワークやディスク I/O に着目してデータ配置を決定する既存の取り組みがある．グリッド環境を対象とした研究では，ネットワークスループットとファイルアクセス方式に着目したファイルクラスタリングを行い，ディスク I/O のオーバヘッドを削減した [7]．ネットワークレイテンシに配慮したレプリカ配置では，ノード間のネットワークレイテンシに基づくノードのグルーピングにより，データの移動コストを削減した [8]．Fog アプリケーションの配置に関する研究では，QoS 要件を満たすためにネットワークレイテンシへ着目したアプリケーションの配置を行った [9]．キーワード検索におけるオーバヘッドを削減する研究では，ディスクからの読み出しを削減するため，エントリとサイズに基づく分割を行った [10]．これにより，インデックスの読み出しの削減と検索結果の再現度の向上を達成した．ログは，検索における取り出し方がログの内容や使い方に依存するため，ネットワーク帯域へ配慮した手法では，ログ検索の高速化は十分でない．

データのアクセス頻度に基づきデータを配置する取り組みがある．非集中型の動的な複製アルゴリズムでは，データへのアクセス頻度に基づきデータのレプリカを再配置することで，頻繁にアクセスされるデータの読み取り時の通信コストを削減した [11]．また，地理的に分散したデータセンタから構成される大規模な検索エンジンでは，すべてのデータセンタで分割したインデックスをアクセス頻度に基づき複製し配置した [12]．これにより，集中型に比較した場合の検索における応答時間を高速化した．これら手法では，利用頻度の高いデータへのアクセスは高速化されている．一方，Web アクセスログの検索で発行されるクエリは，ログの内容に依存しておりログのアクセス頻度への配慮では高速化が十分でない．

時系列に基づきデータを分割し，配置する取り組みがある．シンプルな全文検索エンジン Hayabusa は，高速のためにログを 1 分単位で保存先ファイルを変更することで，並列で複数のファイルへ同時に全文検索を実現している [13]．時系列に着目した文章検索の研究では，時系列によりインデックスファイルの結合によるファイル削減でディスク I/O を削減した [14]．時系列によるシャーディングを行った研究では，データを到着した順序で並べ替えることで，クエリ処理時間の削減を図った [15]．こうした時系列に着目した取り組みでは，ログの種類により出力される量や検索における取り出し方は異なる．ユースケースにおける検索の応答時間の要件は 1 分以内である．同一時刻におけるログ件数が増えるほど，時系列によるアプローチは検索対象のデータ量が削減できず，ディスク I/O が増加する．そのため，短期間の大量のログを検索する場合において検索の改善が十分でない．

3. 提案

ログ検索における応答時間を削減するためには，ログデータへのアクセス傾向に基づいたログデータの配置が必要である．本研究では，Web アクセスログを対象にログ検索の応答時間を削減する分散ログ管理手法を提案する．ログ検索の応答時間を高速にするため，Web アクセスログの内容から特徴量を取り出し，それに基づきログの配置を記述したルール（以降，ログ配置ルール）を生成する．生成されたログ配置ルールに基づきログを再配置する．これによりログデータへのアクセス時に発生するディスク I/O を複数のノードへ分散させ，ログ検索を高速化する．検索のコストモデルを式 (1) に示す．複数のノードにログを分散配置した場合の検索における応答時間 T を高速化する．関数 f は，入力としてログ配置 a と 1 件以上の検索クエリが含まれる集合 q をとり，出力としてそのクエリにより検索に費やした時間 T を返す．ログ配置は，ログとログを配置するノードの組合せの集合を表す．関数 g は，入力としてログ配置 a と検索クエリの集合 q，ネットワークのレイテンシ b とシステム全体のログ件数 l，ワーカノード台数 n とディスク I/O のレイテンシ s をとる．また g は，入力に対応するログ検索の応答時間 T を返す．関数 f は，関数 g により表される．本研究ではノード間はローカルネットワークで接続され，ネットワークレイテンシは一定とする．そのため，ネットワークのレイテンシ b を定数とおく．また，ストレージの容量に限りがあり保管できるログの件数が一定であることから l を定数とする．さらに，本研究ではホモジニアスなノードを対象とするため，ディスク I/O のレイテンシ s は一定であるとする．ノード数 n は，検索クエリとログ配置の関係に絞るため，定数とした．検索クエリ q を想定してログ配置 a を決定することにより，検索の応答時間 T の削減を図る．

$$T = f(a, q) = g(a, b, l, n, s, q) \tag{1}$$

提案の概要を**図 3** に示す．ノードは，ワーカノードとマスタノードの 2 種類から構成される．ここでは簡単のため，ワーカノードはホモジニアスなハードウェア性能を持つノードの集合とする．ワーカノードでは Web アプリが動作し，ログをローカルディスク上のファイル（以降，ログファイル）へ出力する．ログは，**ソースコード 1** に示す形式であり，クライアントからサーバ（ワーカノード）へのアクセスにともない生成される．提案では，ログを検索する場合の応答時間を高速化するため，ログ配置ルールに基づきログを再配置する．システム管理者がログを検索するには，マスタノードへ検索クエリを発行する．マスタノードは，受け取ったクエリに対応するログが配置されたノー

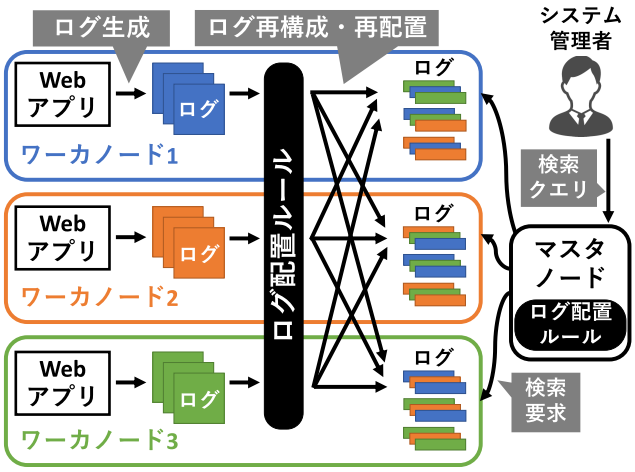

図 3 提案の概要
Fig. 3 Proposal overview.

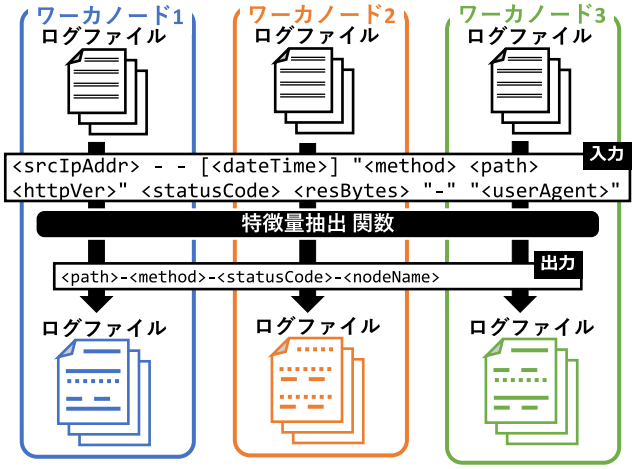

図 4 特徴量の抽出
Fig. 4 Extracting features.

ド一覧をログ配置ルールから取得する．マスタノードは取得したノード一覧へ検索要求を発行する．ワーカノードは検索要求に基づきログファイルを検索し，その結果をマスタノードへ応答する．マスタノードは，ワーカノードの応答を統合し検索結果としてシステム管理者へ応答する．

3.1 特徴量の抽出

収集した Web アプリのアクセスログから，検索されるクエリの特徴に着目し，クエリでの使用頻度の高い属性（例：Method，Path）を抽出する．以降，この属性を特徴量とよぶ．ログに含まれる特徴量のうち実際にログの検索で利用されるものは一部である．そのため，本提案ではユースケースから作成した検索クエリをもとに，Web アクセスログの特徴量間の依存関係に着目しログから特徴量を抽出した．ログから特徴量を取り出す手順を図 4 に示す．Web アプリから出力されたログは，ログファイルへ保存される．このファイルへ保存されるログは，**ソースコード 1** と同様の形式を持ち，Web アプリがクライアントから受信

アルゴリズム 1 特徴量抽出関数

```
1: function EXTRACT_FEATURES(log)
2:     return (log.method, log.path,
3:         log.statusCode, log.nodeName)
4: end function
```

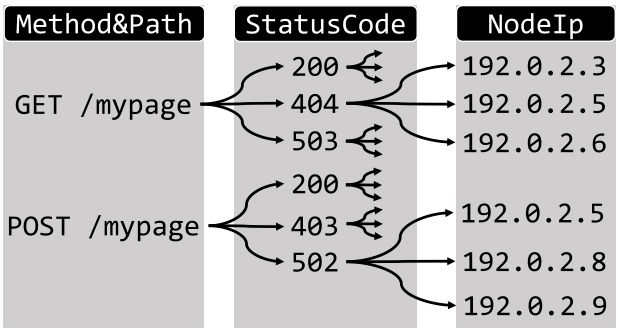

図 5 属性の依存関係
Fig. 5 Attribute dependencies.

した 1 リクエストあたり 1 件のログが記録される．このログから特徴量を抽出するため，入力されたログから特徴量を取り出し出力する特徴量抽出関数を作成した．

特徴量抽出関数（以降，抽出関数）のアルゴリズムを**アルゴリズム 1** に示す．抽出関数は，入力として 1 件のログ *log* を受け取り，ログに含まれる属性を取り出す．ログに含まれる属性である method と path はぞれぞれ HTTP リクエストのメソッド名（例：GET）とパス（例：/index.html）を表す．statusCode はリクエストに対応するレスポンスのステータスコード（例：404）を，nodeName はログの生成されたノード名を表す．次節では抽出関数の出力をキーとして，ハッシュマップを作成しログのまとまり（ブロック）を作成する．したがって，抽出関数の出力がブロックの大きさを決定する．

Web アクセスログに含まれる属性には，検索時のクエリによる属性どうしの依存関係が生じる．**ソースコード 2** のクエリは，障害の発生した時刻を絞り込むために発行される．**ソースコード 3** のクエリでは，障害の発生したノードを絞り込むためを発行される．ログに記録された属性のうち，実際にログ検索で利用される属性は一部である．そのため，本提案ではユースケースから作成した検索クエリをもとに，Web アクセスログの属性間の依存関係に着目しログから特徴量を抽出した．Web アクセスログに含まれる属性（例：StatusCode）には，ユースケースにより依存関係が発生する．これらの発行される検索クエリにおける属性の依存関係を**図 5** に示す．Web アクセスログの Method&Path 属性と StatusCode には $Method\&Path \Rightarrow StatusCode$ の関係がある．また，同様に StatusCode と NodeIp（ログが生成されたノードの IP アドレス）には $StatusCode \Rightarrow NodeIp$ の関係がある．

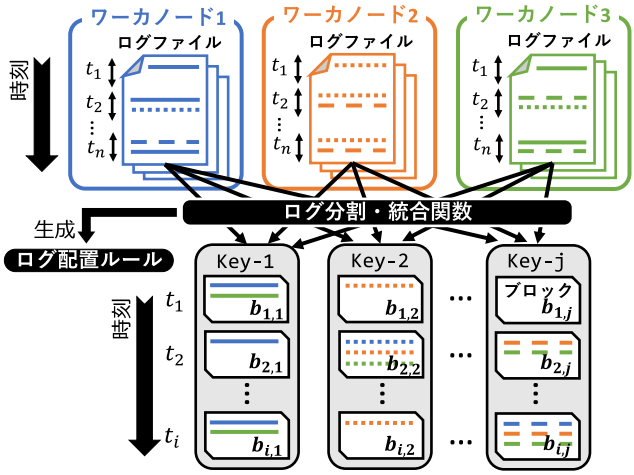

図 6 ブロックの作成
Fig. 6 Building blocks.

3.2 ブロックとログ配置ルールの作成

ワーカノードで動作する Web アプリから生成されたログは，ノードの内部ストレージにログファイルとして保存される．本節では，ログファイルをログ配置ルールに基づき，特徴量に着目したデータの集合であるブロックを作成する方法を述べる．

ブロックの作成方法を**図 6** に示す．ノードごとのログファイルには，ログが時系列で記録されている．**ログ分割・統合関数**は，ノードごとのログファイルを前節で抽出した特徴量と一定の時間間隔で分割することにより，ログファイルからブロックを作成する．また，ブロックと配置先ノードの対応を記述した**ログ配置ルール**を生成する．このルールは，ログの検索処理でディスク I/O を複数のワーカノードへ分散させるため，同一の特徴量を持つブロックを日時（年，月，日，時）ごとに異なるワーカノードへ割り当てる．さらに，ブロックはログ分割・統合関数によりノードをまたいだ特徴量のグループ $Key-1$ から $Key-y$ に分けられる．y はユニークな特徴量と日時の組合せ総数とする．Key は，同一の特徴量（path，method，statusCode，nodeName）と日時（年，月，日，時）の組合せを持つログの集合である．グループには同一の特徴量を持つブロック群が時間間隔 t_1 から t_x に分割され含まれる．x は，あるグループにおける時間間隔の総数とする．これはログを同一の特徴量ごとにクラスタリングするために作成する．たとえば，$Key-1$ は同一の特徴を持つ時刻 t_1 から t_x までの時間間隔に対応する $b_{1,1}$ から $b_{x,1}$ までのブロックを含む．時間間隔によるブロック作成は，時間帯を絞った検索クエリにともない発生するブロック走査を削減し，検索の応答時間を削減する．ユースケースよりトラフィックが平均 1,000 [request/sec] の場合，1 時間あたりのログ件数は 3,600,000 件となる．走査対象のブロック数とブロックの大きさ，トラフィックから時間間隔の長さを 1 時間とした．

提案手法のアルゴリズムを**アルゴリズム 2** に示す．ワー

アルゴリズム 2 ログ分割・統合関数

```
1: newBlocks ← HashMap()
2: newBlocksKeys ← Set()
3: for i ← {1, ..., N'} do
4:     for j ← {1, ..., F'_i} do
5:         for k ← {1, ..., E'_{i,j}} do
6:             key ← EXTRACT_FEATURES(e_{i,j,k})
7:                 ∪ floor(e_{i,j,k}.dateTime by year, month, day,
8:                     hour)
9:             newBlocksKeys ← newBlocksKeys ∪ key
10:            hkey ← Hash(Concat(key))
11:            if newBlocks[hkey] == ∅ then
12:                newBlocks[hkey] ← Set()
13:            end if
14:            newBlocks[hkey] ← newBlocks[hkey] ∪ e_{i,j,k}
15:        end for
16:    end for
17: end for
18: // ログ配置ルール生成アルゴリズム
19: sortedKeys ← sort(newBlocksKeys
20:     by path, method, statusCode, nodeName)
21: nodes ← HashMap()
22: for i ← 0; k ← sortedKeys do
23:     hk ← Hash(Concat( k ))
24:     nodes[i mod N] ← nodes[i mod N] ∪ newBlocks[hk]
25:     i ← i + 1
26: end for
```

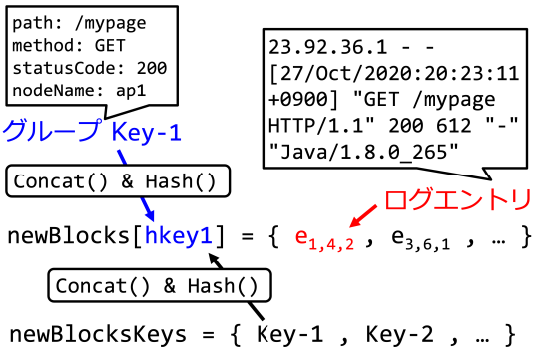

図 7 ハッシュマップの作成方法
Fig. 7 The method to make HashMap.

カノード i に含まれるログファイル j の k 件目のログを $e_{i,j,k}$ とする．このとき，ワーカノードの総数 N'，ワーカノード i に含まれるログファイル数 F'_i，ログファイル j に含まれるログ件数 $E'_{i,j}$ とする．アルゴリズムの 1 行目で定義したハッシュマップ $newBlocks$ は，キーにログエントリの持つ特徴量（特徴量抽出関数の結果を文字列結合）のハッシュ値をとり，バリューにキーへ対応する特徴量を持つログエントリの一覧（セット）をとる．これにより各ファイルに含まれるログエントリを特徴量ごとに集計する．2 行目のセット $newBlocksKeys$ は，$newBlocks$ のキーについてハッシュ化する前の値（特徴量）を一覧で持つ．3–16 行目では，ワーカノードに含まれるログファイルごとに 1 行ずつログエントリを取り出し，$newBlocks$ と $newBlocksKeys$ へ集計する．**図 7** は，ハッシュマップ $newBlocks$ とセッ

ト *newBlocksKeys* の関係を表す．グループ $Key-1$ は，特徴量（path，method，statusCode，nodeName）を持つ．グループごとの特徴量によりログエントリを集計するため，ハッシュマップ *newBlocks* によりログを特徴量ごとに集計する．ハッシュマップにおけるキーは，グループごとに持つ特徴量を文字列結合（Concat）し，ハッシュ関数（Hash）にかけることで求める．バリューは，セットにより $Key-1$ に対応するログエントリの一覧を含む．*newBlocksKeys* は，セットによりグループの一覧を持つ．18–25 行目では，ブロックとその配置先ノードを表すログ配置ルールを生成する．18–19 行目では，集計した *newBlocksKeys* を特徴量 *path*，*method*，*statusCode*，*nodeName* の順で優先順位をつけ，昇順でソートする．20–23 行目は，モジュロ演算によりソートされたブロックをノードに割り当てる．これにより，同一の特徴量を持つログが時系列で複数ノードに分散配置される．これは時間帯を絞った検索にともなうディスク I/O を複数ノードに分散させ，検索の応答時間を削減する．なお，ブロックの配置先となるワーカノードは，クエリに含まれる特徴量と日時（年，月，日，時）により一意に定まる．これは，古いログを含むブロックはサイズの変化が発生せず再配置の必要がないためである．ログは時系列で古いものほど先に届き，新しいものほど後に届く性質を持つ．日時の新しいログは，古いログを含むブロックへ追加されず，新しいログを含むブロックへ追加される．

3.3 ブロック配置

前節でログから作成したブロックをノードへ配置する方法を述べる．ブロックの配置を図 8 に示す．特徴量と時間間隔で分割されたブロックは，シャーディング関数によりワーカノードに配置される．この関数は，ログ配置ルール（ログ分割・統合関数により生成）をもとに配置先のワーカノードを決定する．これによりログ配置ルールに基づいたワーカノードへのブロック配置を実現している．

ログ配置ルールには，ブロックごとに配置先のノード名が記述されている．シャーディング関数は入力されたブロックから対応するノード名を求め，そのノードへログを配置する．ワーカノードのブロックは，ログ配置ルールに基づきブロックの持つ特徴量と時間間隔の昇順で配置する．

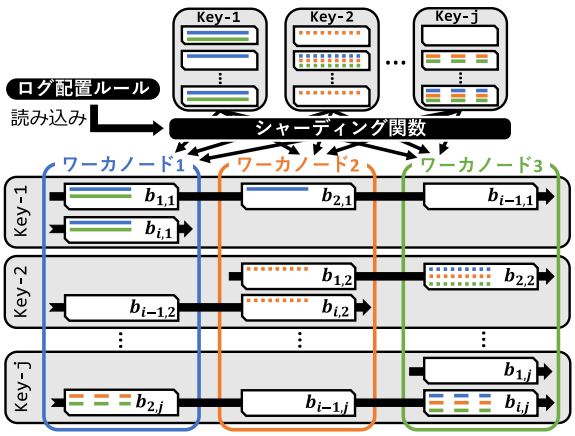

図 8 ブロックの配置
Fig. 8 Allocating blocks.

4. 実装と実験環境

ソフトウェアアーキテクチャを図 9 に示す．ルールジェネレータは，ログジェネレータから出力されたログを読み込み，ログ配置ルールを生成する．ログフォワーダは，生成されたログをログ配置ルールに基づきワーカノードへ転送する．検索マネージャは，システム管理者から発行された検索クエリへ応答するため，ログ配置ルールをもとにワーカノードに検索要求を発行する．検索要求の応答を受け取った検索マネージャは，統合した結果をシステム管理者へ検索応答として返す．マスタノードへ配置する新たに実装したソフトウェアを次に示す．

- **ログジェネレータ**：Web アプリをエミュレートしたログを生成する．秒間リクエスト数，期間の指定により生成されるログ件数が変化する．ログは公開済み WordPress から収集したログを属性ごとに分割し，ランダムに組み合わせ生成する．
- **ルールジェネレータ**：ログの配置ルールを自動生成する．配置手法ごとにジェネレータを実装している．
- **ログフォワーダ**：マスタノードからワーカノードへ転送したログを転送する．rsync とシェルスクリプトにより実装した．
- **検索マネージャ**：システム管理者の発行したクエリからログ配置ルールを使い，ワーカノードへ検索要求を発行し，結果を取りまとめる．評価では，ワーカノードへの検索の応答時間を multitime により計測する．

ワーカノードとマスタノード用に仮想マシンをそれぞれ作成した．それぞれの仮想マシンは，図 10 に示す VMware ESXi 6.7 上に配置され，ストレージはネットワーク上の

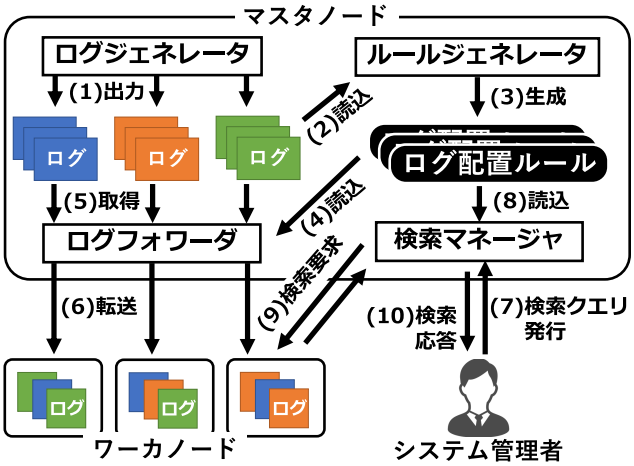

図 9 ソフトウェアアーキテクチャ
Fig. 9 Software architecture.

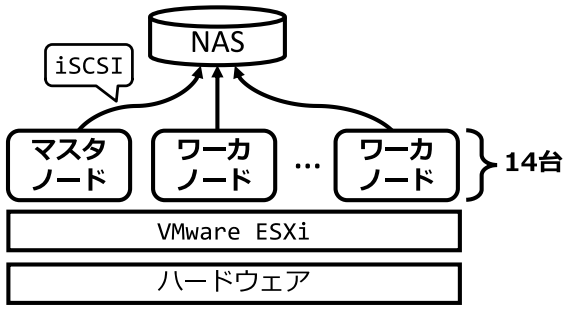

図 10 システムアーキテクチャ
Fig. 10 System architecture.

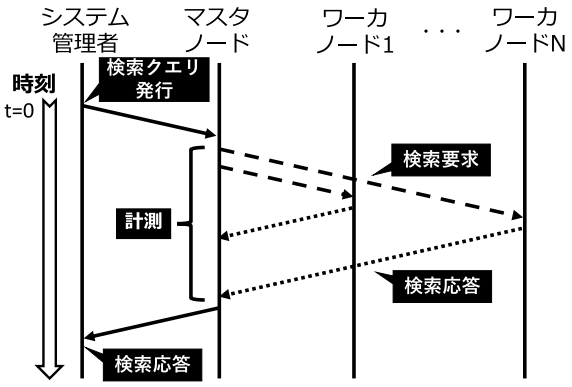

図 11 応答時間の計測方法
Fig. 11 The way of measuring response time.

NAS を VM ごとに iSCSI 経由でマウントする．まず，ワーカノードとして動作させる同一性能の仮想マシンを 13 台作成し Linux OS をインストールした．ワーカノードには，OpenSSH サーバと multitime コマンドがインストールされている．マスタノードとの検索要求や検索応答は，SSH を介してやりとりされる．ワーカノードは，マスタノードからの検索要求を受けると，ログ配置ルールで対応するファイルを対象に grep コマンドでログの検索をする．検索の結果は，SSH 経由でマスタノードへ転送される．次にマスタノード用に 1 台の仮想マシンを作成した．マスタノードには，開発したログジェネレータと配置ルールジェネレータ，ログフォワーダと検索マネージャを配置した．ログを転送するため rsync コマンドをインストールした．

5. 評価と分析

ログ検索の応答時間を測定し，検索における応答時間を評価する．比較は，提案配置（ログの特徴量に基づく手法），ノード配置（ノードごとに配置した手法），時系列配置（ノードごとに時系列配置した手法）で行う．ノード配置は，ログが生成されたノードから移動されず保存された場合を想定した．時系列配置は，時系列での検索を高速化した手法をもとに作成した．

検索の応答時間の測定方法を図 **11** に示す．ここでは，検索の応答時間をシステム管理者がマスタノードへ発行した検索クエリに対応した検索結果が得られるまでの時間とする．実験では，ワーカノードの仮想マシンがストレージを共有することから，並列でのアクセスによるストレージのディスク I/O 低下を避けるため，マスタノードからワーカノードへの検索要求を直列で発行した．直列で計測した応答時間を並列で計測した応答時間 T_p へ $T_p = \max(T_s);\ t_w \in T_s$ により変換する．t_w は，直列での計測における個々のワーカノードの応答時間である．T_s は，全ワーカノードの応答時間を含む集合とする．マスタノードとすべてのワーカノードは同一の 1 Gbps のネットワークで接続されている．ワーカノードが 13 台のとき，検索要求と検索応答によるネットワーク帯域の消費はわずかである．また，マスタノードにおいて検索応答の統合にかかる処理時間は，ワーカノードにおける検索処理にかかる時間に比べて十分に小さい．VM に iSCSI 経由でのランダム読み書きを行い，ディスク I/O の性能（10 回平均）を求めた．書き込み性能は 107.4 [MB/sec] であり，読み出し性能は 122.2 [MB/sec] である．ユースケースからログエントリは 1 件あたり 100 [B] と想定する．このとき，1 秒あたり $122.2 \cdot 10^6/100 = 12{,}220{,}000$ 件が読み出せる．したがって，実験においてスループットは十分に確保されている．ログジェネレータでユースケースをもとに平均 1,000 [request/sec] を想定してログを生成した．このログは，研究室で公開している Web サイト[*6]のアクセスログから属性を取り出しそれぞれをランダムに組み合わせることで生成した．ランダムな組合せは，実際のアクセスログの持つ不規則なリクエストパターンを再現するために行った．ユースケースより 1 日あたり生成されるログは，1,000 [request/sec] × 86,400 [sec] = 86,400,000 件である．実験ではハードウェアのキャパシティからログの総量を 3 日分，ワーカノードを 13 台とした．

5.1 ログ配置と検索クエリ

ログの配置が検索の応答時間に与える影響を調べるため，性質の異なる検索クエリを 3 種類のログ配置（提案配置，ノード配置，時系列配置）へ発行し，検索の応答時間を計測した．提案配置はログに含まれる属性と日時に基づきワーカノードへログを配置する．ノード配置は 1 台のノードから出力されたログを，それに対応する別の 1 台のノードへまとめて配置する．時系列配置は，全ノードの同じ時間帯のログを 1 台のノードへまとめて配置する．なお，1 ブロックに含まれるログの期間は 12 時間とした．実験では，次に示す 3 種類のクエリ（ノードで限定，時間帯で限定，メソッド&パスで限定）を発行した．ノードで限定したクエリを**ソースコード 4** に示す．

- **ノードで限定：**ノード 1，3，5，7，9 に配置された 3

[*6] https://ja.tak-cslab.org/

ソースコード 4 ノードを絞り込むクエリ

```
sword="(Chrome|Firefox)" smethod=".*" sstatus=".*"
     spath=".*" snode="koyama-log[13579]"
```

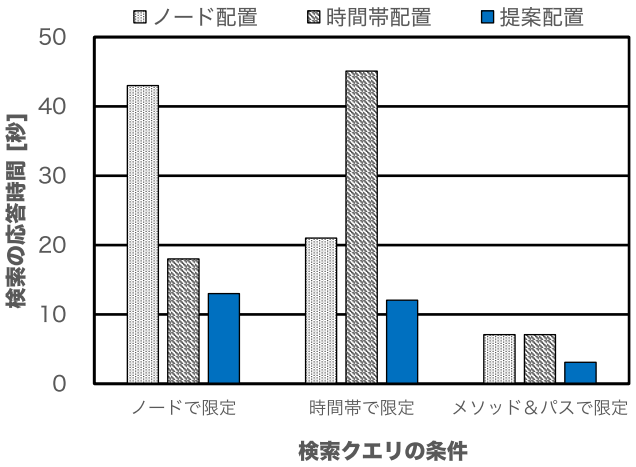

図 12 ログ配置ごとの検索の応答時間
Fig. 12 Search response time per log placement.

日分のログから，Firefox または Chrome を含むログを検索．

- **時間帯で限定**：すべてのノードに配置された 3 日分のログから，各日の半日分（0:00〜11:59）の Firefox または Chrome を含むログを検索．
- **メソッド&パスで限定**：すべてのノードに配置された 3 日分のログから，メソッドが GET かつパスが/about/about-cdsl である Firefox または Chrome を含むログを検索．

実験結果を**図 12** に示す．横軸は発行した条件ごとに検索クエリを，縦軸は検索の応答時間（単位：秒）を示す．検索の応答時間は，小さいほど検索に費やす時間が短く，高い性能を表す．配置ごとに 10 回の検索クエリを発行し，その平均応答時間を計測した．このとき，Linux のページキャッシュやバッファキャッシュによる影響を実験ごとに揃えるため，実験の条件を変更するたびにキャッシュを削除した．

ノードで限定した検索クエリでは，提案配置がノード配置に比べ検索の応答時間が 30 秒早い．また，提案配置は時間帯配置に比べ検索の応答時間が 25 秒早い．これは，ワーカノードに発生するディスク I/O が配置により異なるためである．**図 13** は，ノードで限定した検索クエリにより発生するブロックへのアクセス数をワーカノードごとに示した．横軸にワーカノード番号，縦軸に配置されたブロック数をとる．応答時間が他の配置に比べ長いノード配置は，ワーカノード 5，7，11，13 の 4 台にブロックが配置されている．これは時間帯配置や提案配置に比べノード数が少なく，1 台あたりで検索するブロック数が多い．1 ノードあたりが検索するブロック数の増加は，1 ノードあたりの

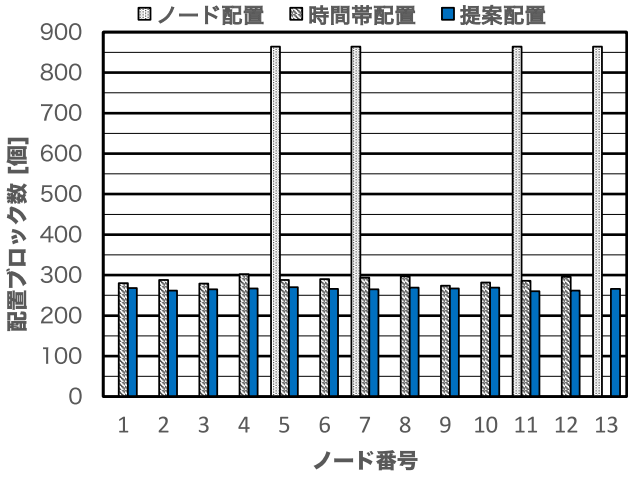

図 13 ノードで限定したクエリにおけるワーカノードごとの配置ブロック数
Fig. 13 Number of allocated blocks per worker node in node search query.

ディスク I/O を増加させる．また，ディスクの許容量を超えた I/O は待ち状態となり読み込みの遅延の原因となる．つまり，ブロック配置の偏りが局所的なディスク I/O を増加させ，検索の応答時間を消費している．同様に時間帯配置では，時間帯を限定したことによりディスク I/O が一部のワーカノードに偏り，応答時間が提案配置に比べ長くなった．

時間帯で限定した検索では，提案配置は検索の応答時間がノード配置に比べて 9 秒早く，時間帯配置に比べ 32 秒早い．ノードで限定した配置と同様に同一時間帯のログを含むブロックが一部のノードに集中して配置される．これはノードで限定した場合と同様に，ノードへブロックの配置が偏ることによるディスク I/O の増加による遅延が影響している．

メソッド&パスで限定した検索は，提案配置がノード配置と時間帯配置に比べて 4 秒早い．これはワーカノードでのディスク I/O が配置により異なるためである．ブロックへのアクセス数を**図 14** に示す．提案配置は，ノード配置や時間帯配置に比べ，1 ノードあたりに配置されたブロック数が分散されている．また，時間帯配置は配置ブロック数が均等に分配されているが，ノード 13 にブロックの配置がない．そのため，ディスク I/O の偏りにより遅延が発生し検索の応答時間が遅くなった．したがって，提案配置はワーカノードごとのディスク I/O が均等に分配され，提案配置は検索の応答時間がノード配置や時間帯配置に比べ早くなった．

6. 議論

6.1 特徴量の選定

ログの分割と統合を行うため，Web アクセスログの持つ属性と検索される傾向から作成した特徴量を利用した．ロ

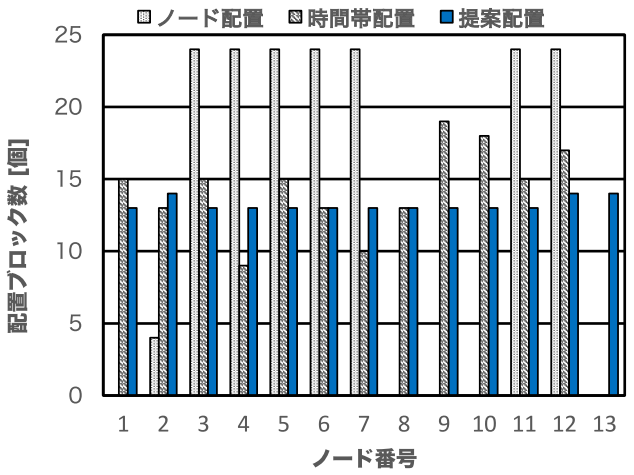

図 14 メソッド&パスで限定したクエリにおけるワーカノードごとの配置ブロック数

Fig. 14 Number of allocated blocks per worker node in method&path search query.

グの持つ属性はログの種類により異なる．また，検索される傾向はユースケースにより異なる．たとえば，セキュリティの目的で接続元 IP アドレスごとのアクセス数を集計する場合，ログ分割の特徴量に IP アドレスを含める必要がある．したがって，検索される傾向に合わせた手動での特徴量の選定が必要となる．これには検索時に発行されるクエリを収集し学習することで自動で特徴量を選定する方法がある．具体的には，クエリに含まれる属性の出現頻度が高い属性どうしを求め，特徴量を自動で選定する．また，キーワードによる全文検索ではキーワードどうしの関係性と言語モデルをもとに特徴量を求められる [16]．

6.2 ブロックのサイズ

ログのまとまりであるブロックは，そのサイズが検索の応答時間に影響を与える．ブロックのサイズを大きくするほど，OS の持つキャッシュ（例：ページキャッシュ）により検索処理にかかる時間を削減できる．一方，ブロックサイズが小さくなるほど個別ノードのストレージ空き容量に配慮した配置が必要となる．2 億件のデータを対象に Hadoop のブロックサイズと処理時間を比較した研究では，ブロックサイズが 128 MB の場合の処理時間が最速である [17]．また，Amazon Redshift [*7]では，推奨するファイルサイズの最小値が 64 MB である．したがって，ブロックサイズは 64 MB から 128 MB の範囲で設定することが一般に最適である．アクセスが毎秒 1,000 リクエストかつログ 1 行が 100 バイトのとき，ブロックサイズが 64 MB を満たすためには，分割する期間はブロック 1 つあたり約 11 分（$64 \times 10^6/100/1{,}000 = 640$ [sec]）である．一方，ログをブロックのサイズのみに考慮した分割をする場合，時間帯を指定した検索クエリにおいて不要なブロックへのアクセスと走査が発生する．たとえば，10:00 から 11:00 のログを検索クエリを発行する場合，9:50 から 10:10 までのログが記録されたブロックを走査すると 9:50 から 10:00 までのログの走査はクエリの条件を満たないため無駄なリソース消費が生じる．その結果，検索の応答時間が長くなる．これに加えて，時間帯での分割はトラフィックの変動にともなうログ件数の増減により，ブロックごとのサイズに不揃いが発生する．ブロックサイズが不揃いであることにより，ノードごとの検索にともなう走査にかかる時間が異なる．これは，ブロックサイズを揃えた場合に比べ検索の応答時間が長くなる原因となる．ブロックサイズの調整と不必要な検索にともなうブロックへの走査の削減への対処は，不必要な検索にともなうブロック走査を許容することである．scatter-gather パターンでは複数あるワーカノードの応答時間のうち最大値が全体の応答時間になる．そのため，ワーカノードにおいてブロック内の不必要なログの走査にともなう遅延は，他のワーカノードでブロック内のすべてのログを走査する時間に比べわずかである．

[*7] https://docs.aws.amazon.com/ja_jp/redshift/latest/dg/c-spectrum-external-performance.html

6.3 関連研究との比較

Hayabusa では，10 台のワーカノードにおいて 14.4 億件の syslog を対象にログの検索を行った [13]．ログデータの配置は，時系列により 1 分ごとにログデータをファイルへ分割した．その結果，検索の応答時間は 39 秒であった．提案配置では，13 台のワーカノードにおいて 2.6 億件の Web アクセスログを対象にログ検索を行った．ログデータは，ログから抽出した特徴量をもとに配置した．実験の結果，検索の応答時間が 3 秒であった．

提案手法と Hayabusa を計算により同一の条件（ログ件数，ワーカノード数）に揃え，見積りにより比較する．Hayabusa のログ件数は，提案手法の $14.4/2.6 \approx 5.5$ 倍である．また，Hayabusa におけるワーカノードの台数は，提案手法の $10/13 \approx 0.77$ 倍である．ログ件数は増加するほど，走査するデータ件数が増えるため検索にかかる時間は増加する．ワーカノードの台数は増加するほど，並列度が高まるため検索にかかる時間は減少する．したがって，Hayabusa にログ件数やワーカノード数をあわせた提案手法の応答時間は，$3\,[\text{sec}] \cdot 5.5/0.77 \approx 21.4\,[\text{sec}]$ である．提案手法は，Hayabusa に比べ $39.0 - 21.4 = 17.6$ 秒の応答時間を削減した．これは，ログへのアクセスパターンをもとにログの分割と配置を行った効果である．

7. おわりに

Web アプリのログを対象に，検索クエリの特徴に基づくログの分割と配置により，検索の応答時間を削減した．ログの分割には，検索クエリにより走査されるファイル数を削減するため，Web アプリの持つ属性から取り出した

特徴量を使用した．ログの配置は，ワーカノードのディスク I/O が分散されるよう，時系列の昇順で並べ換えワーカノードへ均等に割り当てた．評価は，13 ノード上に生成したログの分割と配置を行い，3 種類の配置（ノード配置，時間帯配置，提案配置）へそれぞれ 3 種類の検索クエリを発行し，検索の応答時間を計測した．その結果，提案配置は時間帯で配置した場合において他の配置に比べ，検索の応答時間を最大 32 秒削減した．本提案により Web アクセスログの検索にかかる時間が削減され，システム管理者の障害対処に費やす時間を削減した．これにより，従来に比べ短時間での障害対処を実現可能である．

謝辞 本研究は，JSPS 科研費 JP20K11776 の助成を受けたものです．

参考文献

[1] Kent, K. and Souppaya, M.: Guide to computer security log management, *NIST Special Publication*, Vol.92, pp.1–72 (2006).

[2] He, P., Zhu, J., He, S., Li, J. and Lyu, M.R.: Towards Automated Log Parsing for Large-Scale Log Data Analysis, *IEEE Trans. Dependable and Secure Computing*, Vol.15, No.6, pp.931–944 (2018).

[3] Abe, H., Shima, K., Miyamoto, D., Sekiya, Y., Ishihara, T., Okada, K., Nakamura, R. and Matsuura, S.: Distributed Hayabusa: Scalable Syslog Search Engine Optimized for Time-Dimensional Search, *Proc. Asian Internet Engineering Conference*, *AINTEC '19*, pp.9–16, Association for Computing Machinery (2019).

[4] Ousterhout, J. and Douglis, F.: Beating the I/O Bottleneck: A Case for Log-Structured File Systems, *ACM SIGOPS Operating Systems Review*, Vol.23, No.1, pp.11–28, DOI: 10.1145/65762.65765 (1989).

[5] Cutting, D.R., Karger, D.R., Pedersen, J.O. and Tukey, J.W.: Scatter/gather: A cluster-based approach to browsing large document collections, *ACM SIGIR Forum*, Vol.51, No.2, pp.148–159, ACM (2017).

[6] Meiss, M.R., Menczer, F., Fortunato, S., Flammini, A. and Vespignani, A.: Ranking web sites with real user traffic, *Proc. 2008 International Conference on Web Search and Data Mining*, pp.65–76 (2008).

[7] Sato, H., Matsuoka, S. and Endo, T.: File Clustering Based Replication Algorithm in a Grid Environment, *2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pp.204–211 (2009).

[8] Szymaniak, M., Pierre, G. and Van Steen, M.: Latency-driven replica placement, *IPSJ Digital Courier*, Vol.2, pp.561–572 (2006).

[9] Fahs, A.J. and Pierre, G.: Tail-latency-aware fog application replica placement, *International Conference on Service-Oriented Computing*, pp.508–524, Springer (2020).

[10] Anand, A., Bedathur, S., Berberich, K. and Schenkel, R.: Efficient temporal keyword search over versioned text, *Proc. 19th ACM International Conference on Information and Knowledge Management*, pp.699–708, DOI: 10.1145/1871437.1871528 (2010).

[11] Xu, Z., Xianliang, L., Mengshu, H. and Jin, W.: A Dynamic Distributed Replica Management Mechanism Based on Accessing Frequency Detecting, *SIGOPS Oper. Syst. Rev.*, Vol.38, No.3, pp.26–34 (2004).

[12] Cambazoglu, B.B., Plachouras, V. and Baeza-Yates, R.: Quantifying Performance and Quality Gains in Distributed Web Search Engines, *Proc. 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, *SIGIR '09*, pp.411–418, Association for Computing Machinery (2009).

[13] Abe, H., Shima, K., Sekiya, Y., Miyamoto, D., Ishihara, T. and Okada, K.: Hayabusa: Simple and Fast Full-Text Search Engine for Massive System Log Data, *Proc. 12th International Conference on Future Internet Technologies*, *CFI '17*, Association for Computing Machinery (2017).

[14] Berberich, K., Bedathur, S., Neumann, T. and Weikum, G.: A Time Machine for Text Search, *Proc. 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, *SIGIR '07*, pp.519–526, Association for Computing Machinery (2007).

[15] Anand, A., Bedathur, S., Berberich, K. and Schenkel, R.: Index Maintenance for Time-Travel Text Search, *Proc. 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*, *SIGIR '12*, pp.235–244, Association for Computing Machinery (2012).

[16] Song, F. and Croft, W.B.: A General Language Model for Information Retrieval, *Proc. 8th International Conference on Information and Knowledge Management*, pp.316–321, DOI: 10.1145/319950.320022, Association for Computing Machinery (1999).

[17] Mi, H., Wang, H., Zhou, Y., Lyu, M.R.-T. and Cai, H.: Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems, *IEEE Trans. Parallel and Distributed Systems*, Vol.24, No.6, pp.1245–1255 (2013).

小山 智之 （学生会員）

2021 年東京工科大学コンピュータサイエンス学部卒業．現在，東京工科大学大学院バイオ・情報メディア研究科コンピュータサイエンス専攻に所属．研究対象はログ管理とシステム監視．

串田 高幸 （正会員）

2003 年岩手県立大学大学院ソフトウェア情報学研究科後期博士課程修了．日本アイ・ビー・エム株式会社東京基礎研究所でクラウド，分散システムとネットワークの研究に従事．2019 年より東京工科大学コンピュータサイエンス学部教授．IEEE，ACM 各会員．2018～2020 年本会監事．本会フェロー．