平成２１年度産学連携ソフトウェア工学実践事業
（高信頼クラウド実現用ソフトウェア開発
（分散制御処理技術等に係るデータセンターの
高信頼化に向けた実証事業））
事業成果報告書

2010 年 3 月 31 日

株式会社エヌ・ティ・ティ・データ

目次

第１編　技術開発及び実証実験

1　はじめに

本事業は、今後加速が見込まれるクラウドコンピューティングの利用において、利便性のみならずビジネス利用で求められる高い信頼性を備えた次世代のデータセンターの構築・運用に必要な技術の開発・実証を行うものである。クラウドコンピューティングの活用と、それに伴う情報の集中・集積が相乗的に進むことにより、今後のデータセンターサービスでは、大量のデータを柔軟に処理する需要が高まると考えられる。そこで本事業では、大量のデータを大量サーバで処理を行う分散処理基盤をデータセンターサービスとして活用するために必要な技術開発及び実証事業を行う。

1.1 技術開発及び実証実験の目的

近年、クラウドコンピューティングは利用者がネットワークの向こう側にあるサービスやリソースを必要なときに必要なだけ利用できるという利便性が支持され、多くの利用者を集めている。利用者の増加につれて大量かつ様々なデータがクラウド上に集中・集積しつつあり、その結果、集積されたデータそのものや更にはデータの利用ログなどの２次情報まで含めた多様なデータを活用し、ビジネスに生かす機運・要望が高まっている。このような背景を鑑み、データセンターにおけるクラウド型サービスの一環として、大量なデータを蓄積するとともに多様な用途に合わせて柔軟な処理をスケーラブルに扱うことができる基盤の整備が求められている。また、その基盤はビジネス目的に必要なサービス品質を確保できるだけの信頼性、データセンターの運用において柔軟・効率的に扱える運用性を有することが必要である。

本事業では、ビジネス用途で大量なデータを柔軟に処理できるクラウド型サービスへの適用を前提として、大量のサーバで分散処理を行う技術(以下、クラウド型分散処理基盤と表記する)に注目し、クラウド型サービスの実用化に必要な技術開発を行う。また、クラウド型分散処理基盤の有用性を確認するために、具体性のある大規模データの処理をケーススタディとした実証実験を実施する。

実証実験では大規模データの例として典型的なプローブ情報としてタクシー及び携帯電話の位置情報を取り上げる。近年の通信インフラの高度化および携帯電話に代表される端末機器の爆発的な普及に伴い、位置情報やそれに付随する様々な属性情報は活用手段が着目され様々な試みがなされている典型的な大規模データである。また、データ処理については、プローブ情報から渋滞状況を検出する・渋滞傾向の統計情報を生成するといった処理を題材とする。これらによりクラウド型分散処理基盤の適用性、信頼性、運用性を確認する。

また、本実証実験では積極的なコモディティ製品とオープンソースソフトウェアの採用を行うこととする。コモディティ製品はコスト削減に貢献できる反面、調達でき

る機器のモデル変更が激しい。長期的なクラウド環境の運用においてもサービス品質や経済性を最適化できるよう、クラウド内に異なるハードウェアが混在することを前提とした適用性評価、運用技術の確立を図る。

クラウド型分散処理基盤の仕組みとしてはオープンソースソフトウェアの Hadoop を採用する。分散処理の実装方式では MPI(Message Passing Interface)に基づく MPICH、Open MPI などあるが、本技術開発及び実証実験では大量データの蓄積と処理を行う機構を有していること、つまり分散ファイルシステムである HDFS と分散処理のフレームワーク MapReduce の両者を実装していることから Hadoop を選定している。Hadoop のデータ処理方式は Google が考案し実用化した MapReduce である。

1.2 クラウド型分散処理基盤の特徴と課題

本章では、クラウド型分散処理基盤の特徴と、ビジネス用途にクラウド型分散処理基盤の適用を考えた場合の課題を説明する。

1.2.1 クラウド型分散処理基盤の概要

クラウド型分散処理基盤は大量のサーバで構成されており、計算処理とデータ格納の役目を担う「処理ノード」と、個々の処理ノードでの進捗状況やデータの格納場所を管理する「管理ノード」の 2 種類から構成される。図 1-1 にクラウド型分散処理基盤の概要を示す。

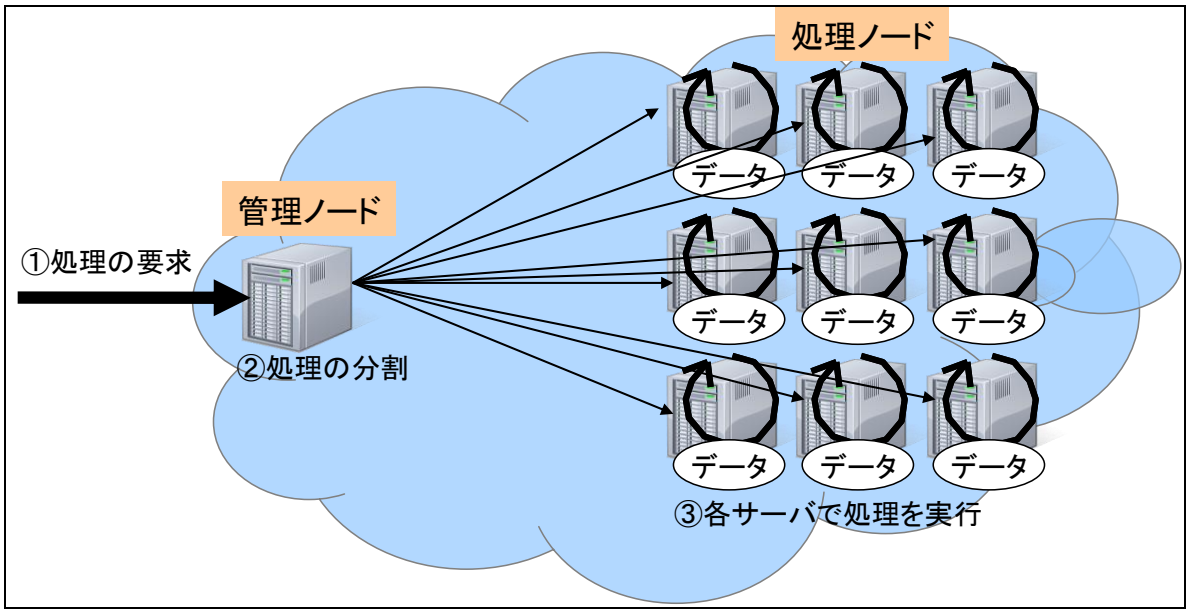


図 1-1 クラウド型分散処理基盤の構成

クラウド型分散処理基盤は、大量の処理を細かく分割して大量のサーバに分散して処理を行う分散処理フレームワークと、サイズの非常に大きいファイルを多数のサーバに分散して格納する分散ファイルシステムの機能の 2 種類から構成される。分散処理フレームワークと分散ファイルシステムについて、Hadoop での実現方法を例に説明する。

(1) 分散処理フレームワーク

分散処理を実現するためのプログラムモデルとして、Google が提唱した MapReduce が最も有名である。Hadoop の分散処理フレームワークは、MapReduce プログラムモデルに基づいている。大量データの処理を Map と Reduce の 2 つのフェーズに分けて行う仕組みである。Map フェーズでは、処理対象の入力データを細かいブロックに分割し、多数のサーバに分散して処理を実施する。Reduce フェーズでは、Map フェーズでの処理結果に対して、同じキーを持つデータを同じサーバにて集計を行う。MapReduce の処理全体を MapReduce ジョブと呼び、MapReduce ジョブは多数の Map タスクと Reduce タスクに分割して実行される。

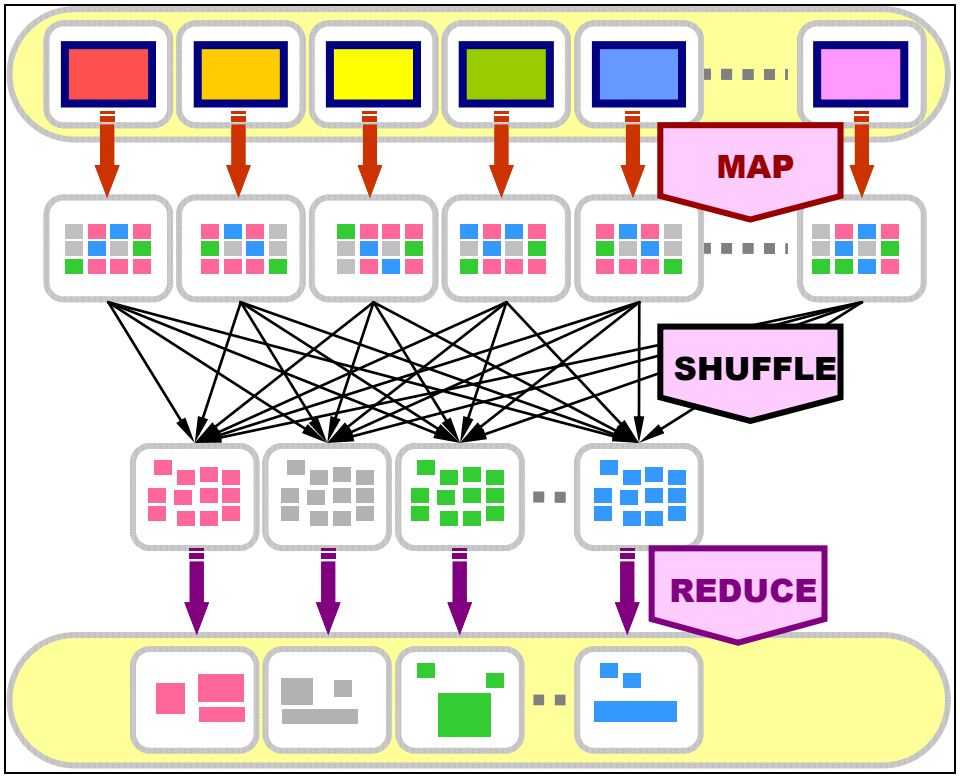


図 1-2 MapReduce の概要

(2) 分散ファイルシステム

Hadoop での分散ファイルシステムは HDFS(Hadoop Distributed File System)である。サイズが非常に大きいファイルを 64MB 程度のブロックに分割して多数のサーバに分散して格納する仕組みである。1 つのファイルを多数のサーバに格納しているため、高い読み込み性能を持っている。また、1 個のブロックを複数のサーバにコピーして格納する仕組みとなっているため、サーバに故障が発生した場合でもデータが失われることなく動作を続けることが可能である。HDFS はファイルのメタ情報を管理する NameNode と、データブロックを格納する多数のサーバである DataNode の 2 種類で構成される。

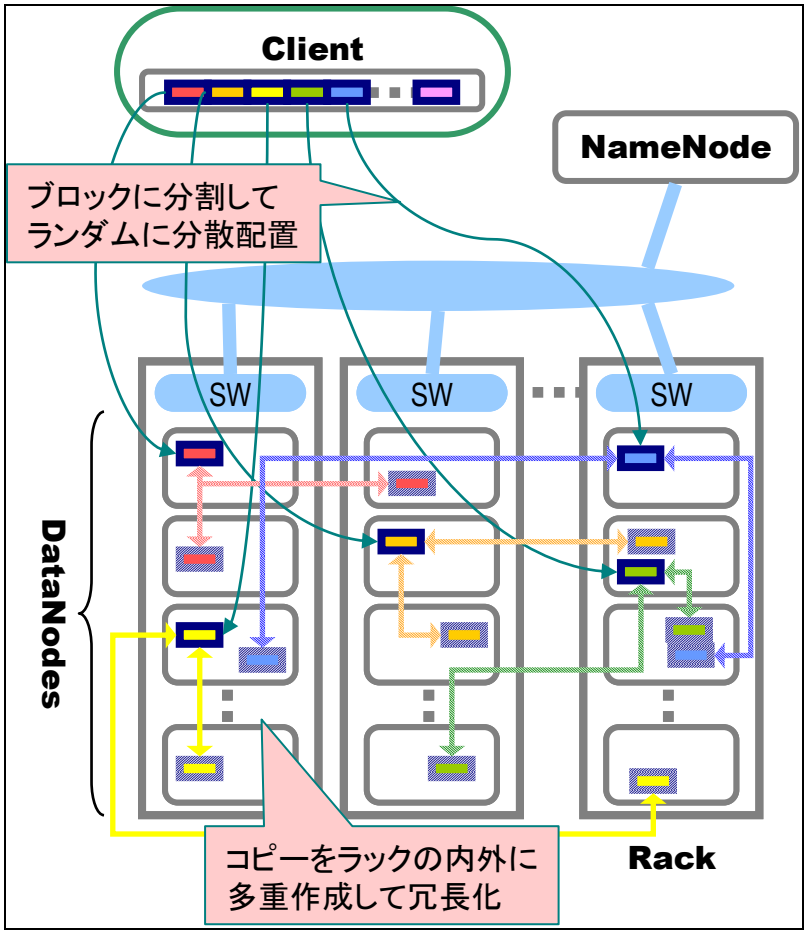


図 1-3 HDFS の概要

1.2.2 クラウド型分散処理基盤の課題

クラウド型分散処理基盤は、従来では扱うことの難しかった大量データの処理を多数のサーバに分散して行うことが可能となる非常に優れた性質を持つが、商用の業務システムでの採用を考えた場合、分散処理基盤の業務への適用性、信頼性、運用性等

に関して十分な検討が必要となる。本節では、クラウド型分散処理基盤の課題と、本技術検証及び実証実験での取り組み方針を説明する。

1.2.2.1適用性

クラウド型分散処理基盤の対象となるアプリケーションは、これまで処理対象としてきたアプリケーションと比べて膨大なデータの処理を行う。そのため、スケーラブルな基盤を十分に利用できるようにアプリケーションの処理分割方法について検討を行う必要があるなど、アプリケーションの作成方法に関して新たな知見が必要となる。

また、クラウド型分散処理基盤の利用形態である大量データの処理では、ユーザの求める処理時間・処理精度等に対する要件を満たせるかが重要なテーマである。クラウド型分散処理基盤において、ハードウェアの処理能力・分散処理の分割度合い・リソース配分等の特性を評価・検証した上で、開発したプロトタイプを用いたプローブ情報の処理で要件を満たすための制御手法について検討する必要がある。

本技術開発及び実証実験では、Hadoop 基盤上で動作する、プローブ情報を処理するアプリケーションのプロトタイプ開発を行うことで、具体的な処理へのクラウド型分散処理基盤の適用性の確認を行う。また、実際に 100 台規模の Hadoop 基盤上で動作させて、ハードウェアリソースを有効活用するためのチューニングを行い、処理時間・処理精度の制御を行うことで処理時間・処理精度に対する要件への適用性の確認を行う。

1.2.2.2信頼性

クラウド型分散処理基盤は、データを多数のサーバに複数コピーして保持する仕組みを有しており、また計算処理は多数のサーバで分散して実施する仕組みとなっているため、サーバに故障が発生した場合でも他のサーバで動作を続けることができる。しかし、動作全体を管理するサーバなどいくつかの構成要素は必ずしも冗長化の仕組みを有しておらず、ビジネス用途でクラウド型分散処理基盤を使用する場合に求められる可用性を満たせない懸念がある。

本技術開発及び実証実験では、Hadoop 基盤全体の信頼性調査を実施して信頼性確保の方針の検討を行う。また、単一故障点となる箇所に対しては冗長化の仕組みを導入し、信頼性確保が行えることを確認する。

1.2.2.3運用性

クラウド型分散処理基盤ではサーバの大量導入・増設が行われるため、1 台 1 台個別に構築を行うことは現実的ではない。そこで、大量のサーバを効率的に構築、追加

する方法を確立する必要がある。また、長期的なクラウド型分散処理基盤の運用において、サーバの導入・増設時期によってサーバのスペックが異なることが想定される。サーバスペックの違いによりサーバ個々の設定内容が異なるため、全サーバに同一内容をコピーするだけではなくサーバの種類に応じた設定変更が必要になる。

また、クラウド型分散処理基盤ではサーバが大量に存在するため、個々のサーバの稼働状況のみではなく、クラウド型分散処理基盤全体としての稼働状況を把握できる必要がある。また、大量のサーバからの稼働状況取得自体の負荷が高くなるため、効率的かつスケーラブルな状況収集の仕組みが必要になる。

本技術開発及び実証実験では、Hadoop 基盤の運用項目を洗い出した上で、大量サーバに対して行う運用項目に対して運用効率化の手法を検討する。具体的には、ハードウェアスペックが混在した Hadoop 基盤の自動構築手法、構成管理の一元化、構築・増設・故障回復手順の統一、Hadoop 基盤の動作状況の可視化の検討を行う。また、実際に 100 台規模の Hadoop 基盤上で手法を適用して運用が効率化できることを確認する。

1.3 報告書の構成

本報告書の構成を説明する。大きく 2 編に分かれており、第 1 編の 1 章～7 章では技術開発及び実証実験の実施内容と結果について報告している。第 2 編の 8 章～13 章では技術開発における特筆すべき技術トピックについて詳述している。報告書の章構成を「図 1-4 報告書の構成」に示す。また、第 1 編の 2～6 章の概要とそれぞれの章が第 2 編の各章とどのように関連しているかを以下に示す。

2 章では、クラウド型分散処理基盤のアプリケーション開発事例として、タクシー及び携帯電話のプローブ情報を用いた渋滞解析アプリケーションの作成について述べる。開発手法は、8 章の MapReduce アプリケーション開発手法に基づく。

3 章では、実証実験で使用する Hadoop 基盤とアプリケーションのパラメータを変化させて分散処理としての性能を向上させるための方策について記載する。9 章に記載している性能評価手法に基づいて設定と測定を行う。測定対象のアプリケーションとしては、2 章に記載した渋滞解析アプリケーションを用いる。

4 章では、Hadoop 基盤における可用性を担保する技術について、基盤を構成する要素ごとに方式を検討し、実証実験において効果を測定した結果を報告する。Hadoop マスタサーバの可用性向上の手法については、10 章に詳細を記載している。

5 章では、Hadoop 基盤の運用上の特徴をスケーラビリティ、クラスタ構成要素の変更への対応、基盤の混在性という 3 つの観点で評価する。11 章に記載した Hadoop 基盤の運用手法、12 章に記載した Hadoop 基盤の自動構築手法、13 章に記載した Hadoop 基盤の動作状況の可視化手法が有効であることを確認する。

6 章では、2 章～5 章に記載したクラウド型分散処理基盤と渋滞情報生成アプリケーションを、より実運用に近いシナリオに基づいて動作させ、クラウド型分散処理基盤の有用性を検証する。

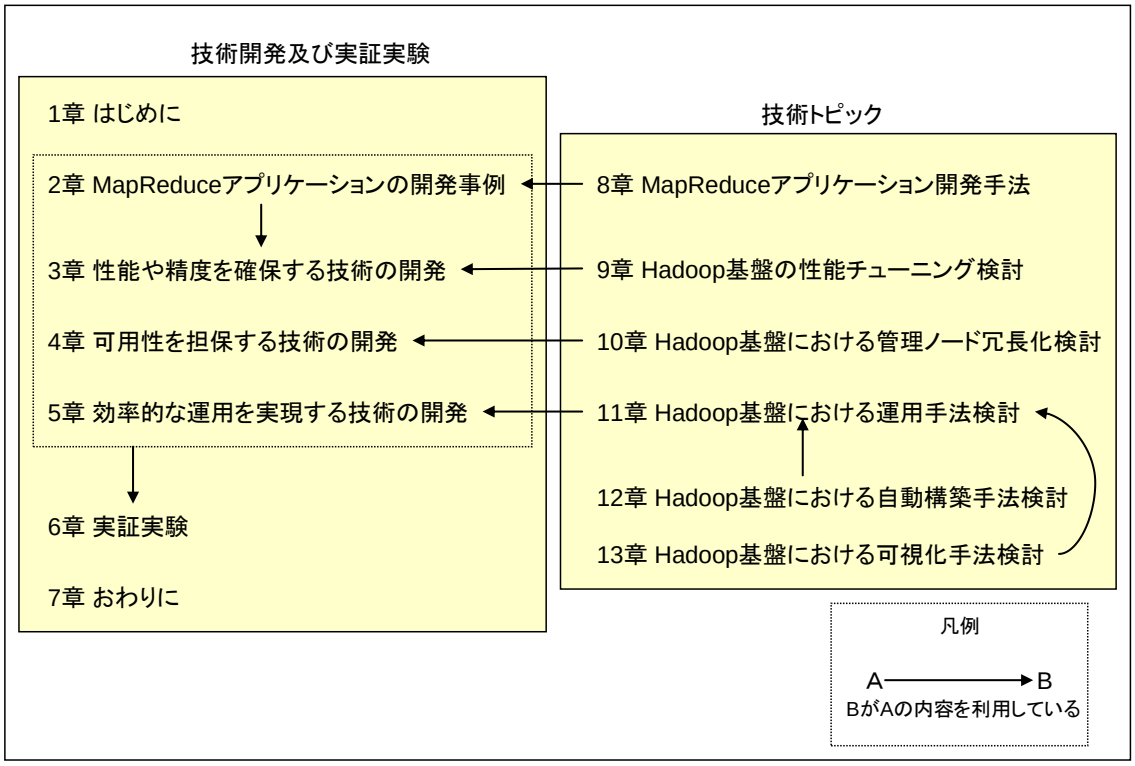


図 1-4 報告書の構成

2　MapReduce アプリケーションの開発事例

本章では、MapReduce を使用したアプリケーション開発について、具体的且つ多様な方法により収集されたプローブデータを利用した「渋滞解析アプリケーション」を事例として説明する。

本章で取り上げる渋滞解析アプリケーションでは、数秒から数分程度の頻度で取得するプローブデータを利用し、渋滞情報を生成する「短時間渋滞情報生成処理」と、長期間に渡り蓄積したプローブデータを利用し、渋滞統計を生成する「渋滞統計生成処理」の２つの処理を実装する。渋滞解析アプリケーションには、処理時間と処理精度の適切なバランスを実現するための機能として、「道路種別を指定する機能」と「道路区間距離を指定する機能」を実装する。その上で、実際に取得したプローブデータを用いて解析を行う。

これらのプロセスを通して、MapReduce アルゴリズムを適用したアプリケーションの設計から実装に至る具体的な開発事例を示す。なお、MapReduce を使用したアプリケーションの開発手法については、8 章に記載されているため参照されたい。

2.1 プローブデータによる渋滞解析

プローブデータとは、走行している多数の車両をセンサとみなし、車載機器から発信される位置情報を含んだセンサーデータを指す。近年は携帯電話の普及、GPS 機能の実装などの背景を受け、固定型の車載機器以外からもプローブ情報の取得が可能になり、プローブデータ量は増加する傾向にある。また、プローブデータはオンライン等を使用し、数秒～数分といった短い間隔で常時収集され、長期的に渡って蓄積、増加していくデータである。

プローブデータの解析事例とする「渋滞解析」では、大量に蓄積されたプローブデータを使用し、日本全国の各道路における渋滞情報を生成するため、膨大な計算が必要となる。また、プローブデータの量は年々増加していく傾向があるため、データ量の増加に柔軟に対応できるシステムが求められている。

そこで本章では、大量のデータ・膨大な計算量に対して、処理時間を台数でコントロールすることができる Hadoop を使用し、渋滞解析を実現する。

2.2 渋滞解析アプリケーションの概要

Hadoop 基盤上で動作する MapReduce アプリケーションの開発事例として、「渋滞解析アプリケーション」を実装する。渋滞解析アプリケーションの要件として、「短時間渋滞情報生成処理」、「渋滞統計生成処理」、「処理精度を制御する機能」及び渋滞解析で使用する「入出力データの仕様」を以下に示す。

2.2.1 短時間渋滞情報生成処理

本処理は、数分程度の短期間に収集されたプローブデータと日本全国の道路データを使用した渋滞情報の生成を目的とする。プローブデータはタクシープローブデータと携帯電話プローブデータの 2 種類のデータを用い、道路を一定の間隔で区切った区間ごとに渋滞情報を生成する。短時間渋滞情報生成処理では、過去 5 分のプローブデータを 5 分間隔で定期的に処理する。処理の対象とするプローブデータは、時系列順に並べて HDFS 上に格納する。本処理結果として、「順調・混雑・渋滞」を持つ直近数分前の渋滞情報を出力する。

2.2.2 渋滞統計生成処理

長期間に渡って蓄積された大規模プローブデータを一括で読み込み、道路区間ごとに生成した渋滞情報を集計キー単位で集約し、渋滞統計を生成する。集計キーは「季節、月、曜日、特異日」の組み合わせで指定する。出力した渋滞統計は、特定条件下における渋滞の傾向と見ることが出来る。

渋滞統計生成処理は、月～年単位の中長期的な間隔で定期的に実行するバッチ処理として、季節別、月別、曜日別、特異日別に集計できることを要件とする。本処理結果は、ある特定期間内に生成された渋滞統計として「順調・混雑・渋滞」で出力する。

2.2.3 処理精度を制御する機能

短時間渋滞情報生成処理では、5 分以内で処理を完了することが求められいてる。そのため、定めらた時間内に処理が完了できるように「処理時間と処理精度のバランス」を制御する必要がある。

そこで、渋滞解析アプリケーションでは、「処理時間と処理精度のバランス」をコントールする機能として、「道路種別を指定する機能」と「道路区間距離を指定する機能」を実装する。各機能で扱うパラメータを変更することで、処理時間と処理精度のバランスを制御することが出来る。各機能について以下に説明する。

道路種別を指定する機能

道路種別を指定し、解析対象道路の設定を行う機能である。道路種別は「主要道路」、

「一般道路」の 2 つから選択する。「主要道路」では、解析対象道路を比較的大きな「高速道路と国道」などとすることで処理量の軽減を実現し、「一般道路」では、「都道府県道や地方道」などといった比較的小さな道路まで対象を広げ、渋滞情報の精度の向上を実現する。

道路区間距離を指定する機能

渋滞情報の生成は、道路を一定間隔で区切った「道路区間」単位に行う。道路区間の距離を指定する単位として、「標準」と「詳細」の 2 つから選択する。「標準」では、数百メートルの道路区間で渋滞情報を生成することで処理量の軽減を実現し、「詳細」では、数十メートル間隔の高精度な渋滞情報の生成を実現する。

2.2.4 渋滞解析アプリケーションの入出力データ

渋滞解析アプリケーションで使用する入力データと出力データについて説明する。

2.2.4.1 入力データの仕様

本事例で入力データとして利用する「タクシープローブデータ」、「携帯電話プローブデータ」と、処理内で使用する「道路データ」について説明する。

タクシープローブデータ

タクシープローブデータは、プローブの発信頻度が数分～10 分間隔であるため、連続した点間の距離は長い。タクシー車両は高速道路や国道といった主要道路以外にも都道府県道や地方道といった細かな道路も走行するため、プローブデータは多様な地点から取得される。タクシープローブデータの仕様を表 2-1 に示す。

表 2-1 タクシープローブデータ仕様

No.	項目名	説明
1	対象エリア	東京都近郊
2	プローブ取得間隔	1～10 分
3	測位日時	プローブデータの測位日時を示す
4	プローブ ID	時系列に連続して取得されたプローブデータを一連のデータとして結びつけるための識別子
5	速度	測位日時における車両移動速度を示す
6	測位進行方向	測位日時における車両進行方向を示す
7	測位位置	測位日時における緯度経度を示す

携帯電話プローブデータ

携帯電話プローブデータは、数秒間隔という高頻度で発信されるため、連続した点間の距離は短く、高密度なプローブデータとして扱うことが出来る。

携帯電話プローブデータの仕様を表 2-2 に示す。

表 2-2 携帯電話プローブデータ仕様

No.	項目名	説明
1	対象エリア	全国
2	プローブ取得間隔	1～2 秒
3	測位日時	プローブデータの測位日時を示す
4	プローブ ID	時系列に連続して取得されたプローブデータを一連のデータとして結びつけるための識別子
5	測位位置	測位日時における緯度経度を示す

道路データ

本アプリケーションで使用する道路データには、日本全国の高速道路、国道、都道府県道、地方道などが含まれている。高速道路と国道などを併せたものを「主要道路」、主要道路に都道府県道と地方道などを併せたものを「一般道路」と定義する。渋滞解析アプリケーション実行時には、「主要道路」か「一般道路」のどちらか 1 つを解析対象道路として選択する。各道路は交差点間で区切られた形状となっており、道路種別によって総道路本数は異なる。道路数の違いを表 2-3 に示す。

表 2-3 道路種別による道路数の差

No.	道路種別	道路数
1	主要道路	約 105 万件
2	一般道路	約 388 万件

2.2.4.2 出力データの仕様

出力データは、5 分間分のプローブデータを使用し解析した「短時間渋滞情報生成結果」と渋滞情報をユーザが指定する集計キーで集約した「渋滞統計生成結果」とする。出力データはタクシープローブデータと携帯電話プローブデータを併せて解析した結果であり、道路区間単位の渋滞情報、渋滞統計として「順調・混雑・渋滞」の 3 段階で表す。出力ファイルはテキスト型式で出力する。出力データの仕様を表 2-4 に示し、渋滞解析結果の表示例を図 2-1 に示す。

表 2-4 渋滞解析アプリケーション出力データ仕様

No.	項目名	説明
1	集計キー	渋滞情報の集計単位を示す
2	道路 ID	道路特定となる ID
3	進行方向	上り方向、下り方向を示す
4	渋滞度	渋滞度を「順調・混雑・渋滞」で示す

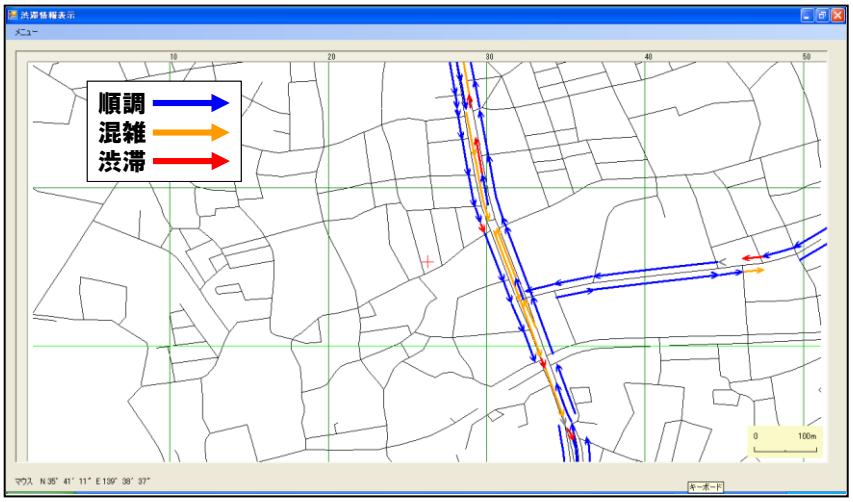


図 2-1 渋滞解析結果の表示例

2.3 MapReduce アプリケーションの設計技法

本節では、MapReduce アプリケーション設計について、渋滞解析アプリケーションを例に解説する。

MapReduce には、次の特徴がある。

- 処理を Map 処理、Reduce 処理に分割して行う。
- 各処理の入出力はキーと値に分類したデータである。

したがって、MapReduce を使ったアプリケーションでは Map 処理、Reduce 処理で行う処理と、その処理で扱うデータ項目の決定が必要となる。

そこで MapReduce 設計では、MapReduce で行う処理と、処理間を流れるデータを次のとおり分類する。

- Map 処理
- Reduce 処理
- 中間データ
- 入力データ
- 出力データ

これらの項目に分類するには、作成するアプリケーションの処理を明確にした処理フローが必要となる。

そこで本節では、まず渋滞解析アプリケーションの処理フローを解説し、次にその処理フローを例に MapReduce 設計について解説する。

2.3.1 渋滞解析アプリケーションの処理フロー

本項では、渋滞解析アプリケーションの処理フローについて解説する。

渋滞解析の構成

渋滞解析は、図 2-2 のとおり 3 つのプロセスで構成される。タクシープローブ解析、携帯電話プローブ解析では、タクシープローブ、携帯電話プローブを使ってある一定の距離ごとの速度情報を作成する。解析結果集計では各速度情報を集計し、短時間渋滞情報と渋滞統計の出力を行う。

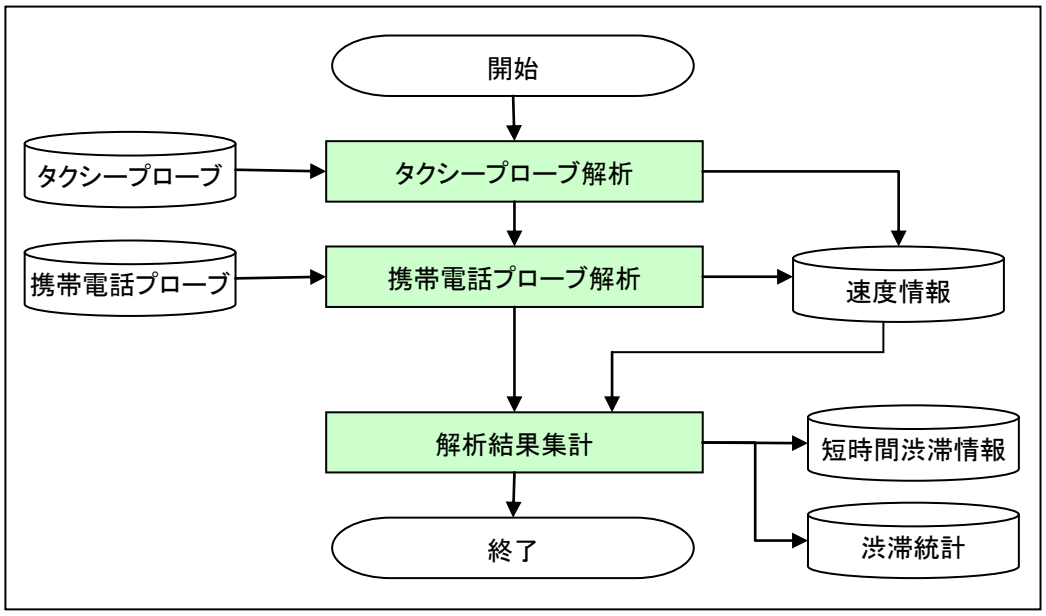


図 2-2 渋滞解析の構成

2.3.1.1 タクシープローブ解析

ここでは、図 2-2 のタクシープローブ解析の処理について解説する。タクシープローブ解析の処理フローを図 2-3 に示す。

タクシープローブ解析では、タクシープローブから道路ごとの速度の計算を行い、計算結果を一定の期間、道路、プローブ ID ごとにまとめて速度情報を作成する。

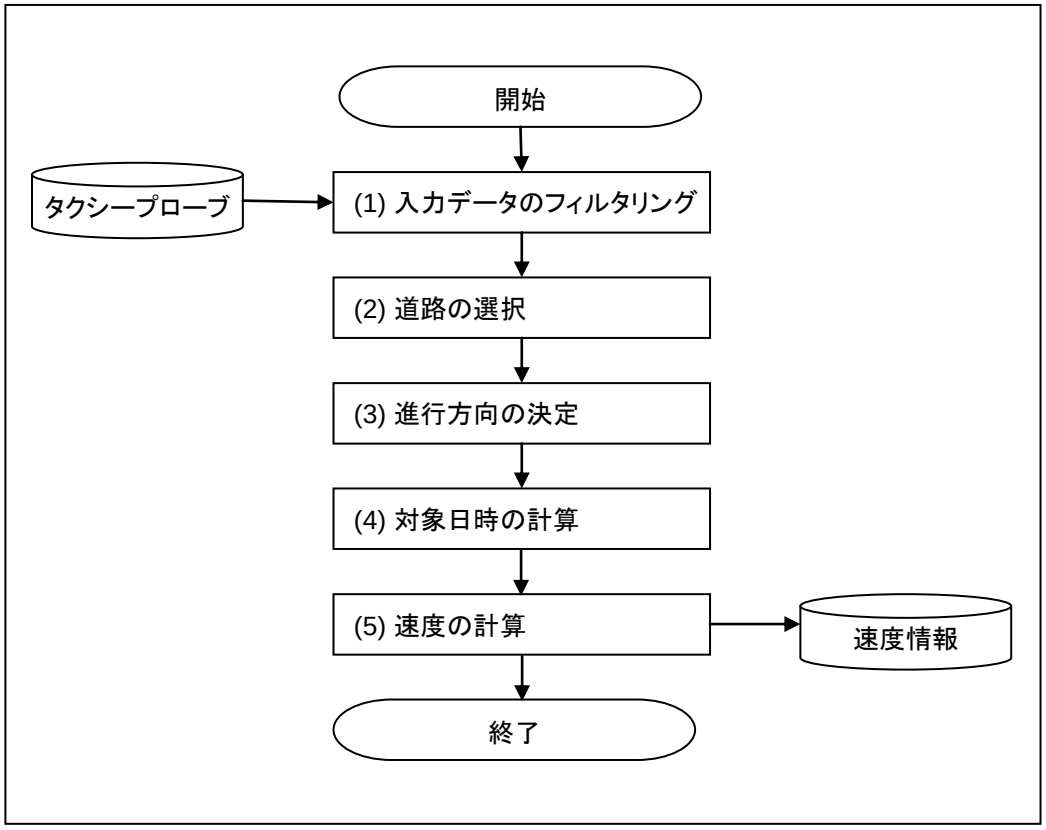


図 2-3 タクシープローブ解析の処理フロー

(1) 入力データのフィルタリング

タクシープローブデータをファイルから読み込み、渋滞解析に利用できないデータをフィルタリングする。フィルタリングのルールを以下に示す。

- フォーマットが不正なデータ
  タクシープローブデータのフォーマットが不正な場合にデータを破棄する。
- 緯度経度が不正なデータ
  タクシープローブデータの緯度経度が処理対象の範囲を外れている場合に、データを破棄する。

(2) 道路の選択

タクシープローブデータの測位位置から道路を選択し、道路 ID と道路総距離、道路の始点からの距離、道路種別を取得する。

道路を選択する際は、タクシープローブデータの測位位置と比較し、最も近い位置にある道路を選択する。なお、一定の範囲内に道路がない場合は、そのプローブデータを破棄する。選択に利用する道路は、設定に応じて主要道路と一般道路を切り替え可能にする。道路の始点からの距離は、選択された道路の始点から、タクシープローブの測位位置までの距離より計算する。図 2-4 に「道路の選択」の処理イメージを示す。

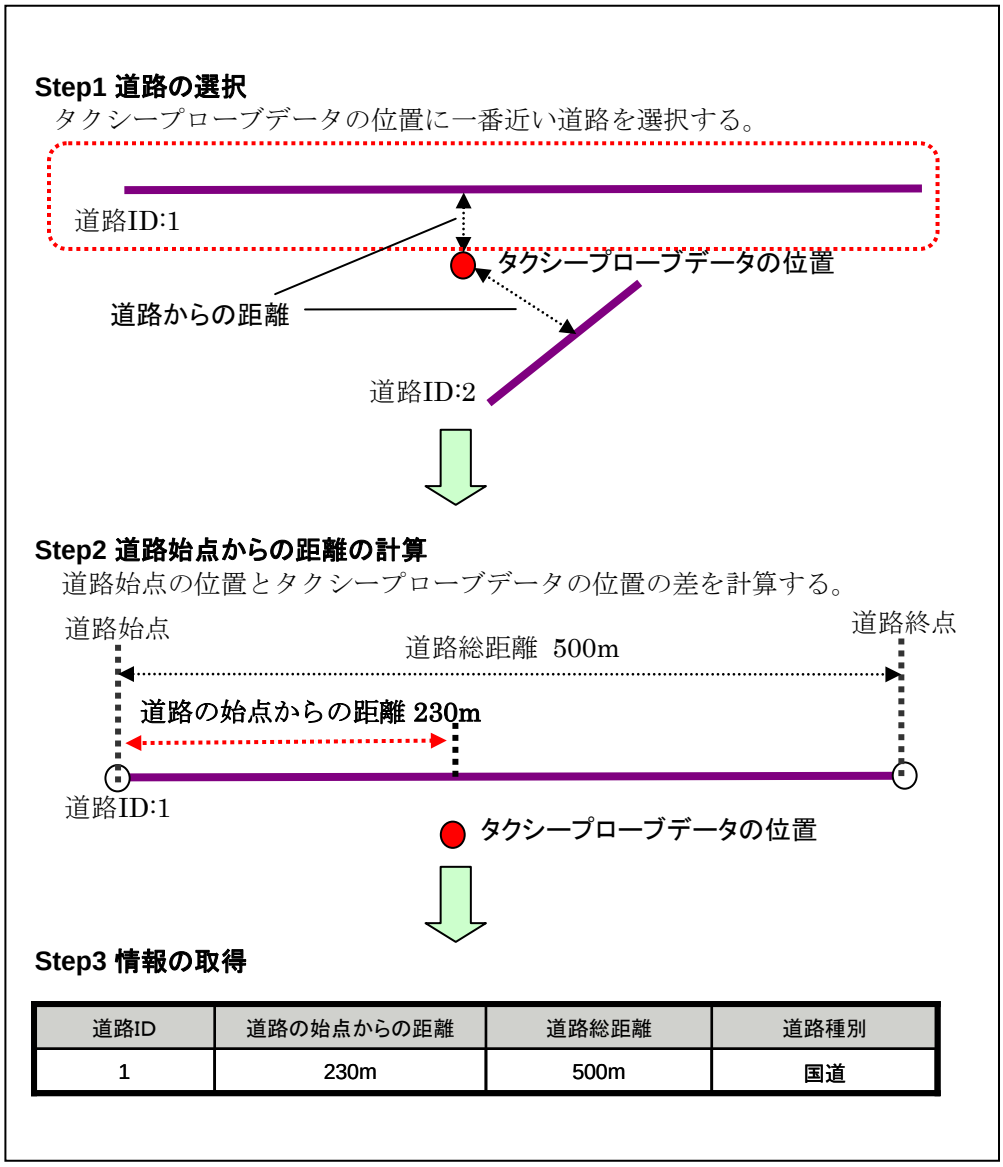


図 2-4「道路の選択」の処理イメージ

(3) 進行方向の決定

入力されたタクシープローブの進行方向が、道路の上り・下りのどちらの情報か判断する。

タクシープローブデータの測位進行方位と(2)で取得した道路の向きを比較し、タクシープローブデータが道路の上り・下りのどちらに対しての情報か判定する。上り・下りの判断は、タクシープローブデータの測位進行方向が道路始点から道路終点へ向かっている場合は上りとし、道路終点から道路始点に向かっている場合は下りとする。

(4) 対象日時の計算

速度情報がどの時間帯の情報かを識別するために対象日時を計算する。1 日を一定間隔で区切り、タクシープローブデータの測位日時が 1 日の区切りのどこに該当するか判定し、その区間に該当する時間を対象日時とする。図 2-5 に「対象日時の計算」の処理イメージを示す。

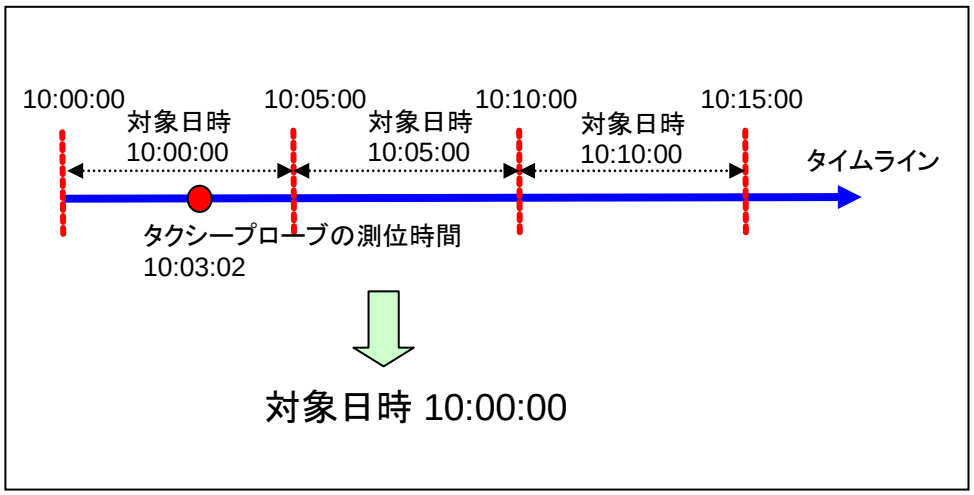


図 2-5「対象日時の計算」の処理イメージ

(5) 速度の計算

タクシープローブデータを道路 ID、対象日時、プローブ ID、進行方向ごとにまとめ速度を計算し、速度情報を作成する。

速度の計算は道路を道路の始点から一定間隔ごとに区切り、(2)で取得した道路始点からの距離より、タクシープローブデータがどの区間に属するか判定を行う。また、その区間での速度をタクシープローブデータの速度(高速、中速、低速、停止)と(2)で取得した道路種別の組み合わせより計算する。計算結果は道路 ID、対象日時、プローブ ID、道路の進行方向ごとにまとめたものを 1 レコードとし、速度情報ファイルに出力する。このとき、同

区間に複数のタクシープローブがある場合はその平均値を設定する。なお、道路を区切る間隔は、設定に応じて切り替えを行う。図 2-6 に「速度の計算」の処理イメージを示す。

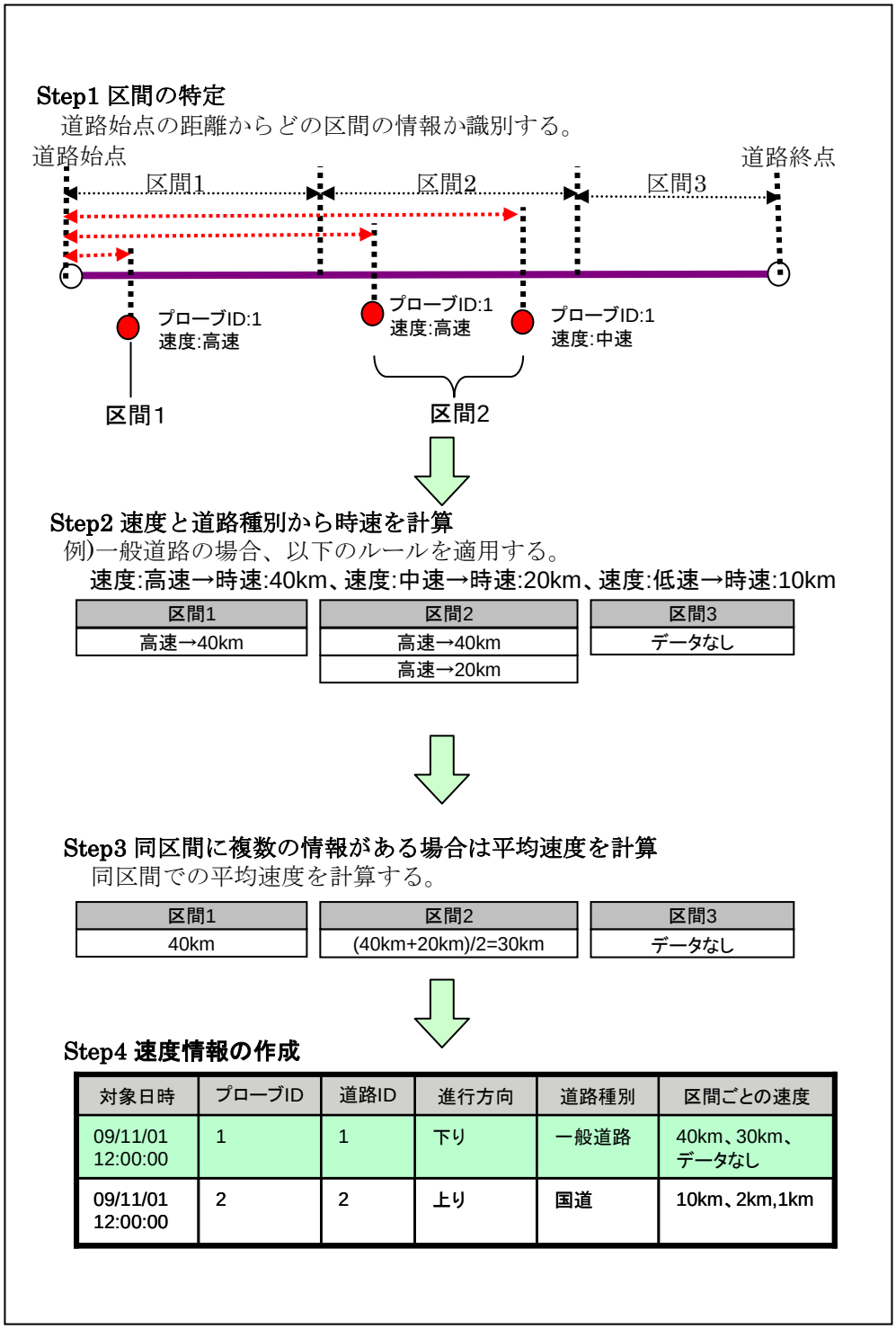


図 2-6「速度の計算」の処理イメージ

2.3.1.2 携帯電話プローブ解析

ここでは、図 2-2 の携帯電話プローブ解析の処理について解説する。携帯電話プローブ解析の処理フローを図 2-7 に示す。

携帯電話プローブ解析では、携帯電話プローブから道路ごとの速度の計算を行い、計算結果を一定の期間、道路、プローブ ID ごとにまとめて速度情報を作成する。

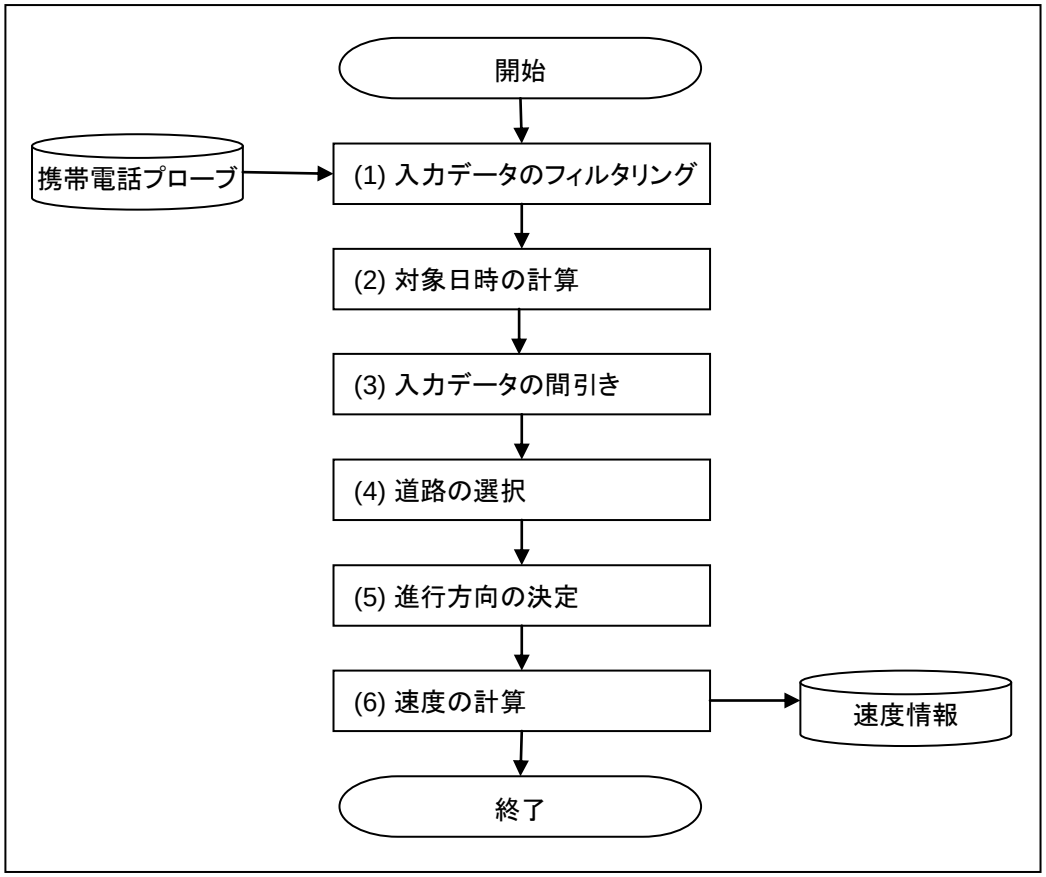


図 2-7 携帯電話プローブ解析の処理フロー

(1) 入力データのフィルタリング

携帯電話プローブデータをファイルから読み込み、渋滞解析に不向きな携帯電話プローブデータをフィルタリングする。フィルタリングのルールを以下に示す。

- フォーマットが不正なデータ
  携帯電話プローブデータのフォーマットが不正な場合は、データを破棄する。
- 緯度経度が不正なデータ
  携帯電話プローブデータの緯度経度が処理対象の範囲を外れている場合は、データを破棄する。

(2) 対象日時の計算

速度情報がどの時間帯の情報かを識別するために、対象日時を計算する。1 日を一定間隔で区切り、携帯電話プローブデータの測位時間が 1 日の区切りのどこに該当するか判定し、その区間に該当する時間を対象日時とする。図 2-8 に「対象日時の計算」の処理イメージを示す。

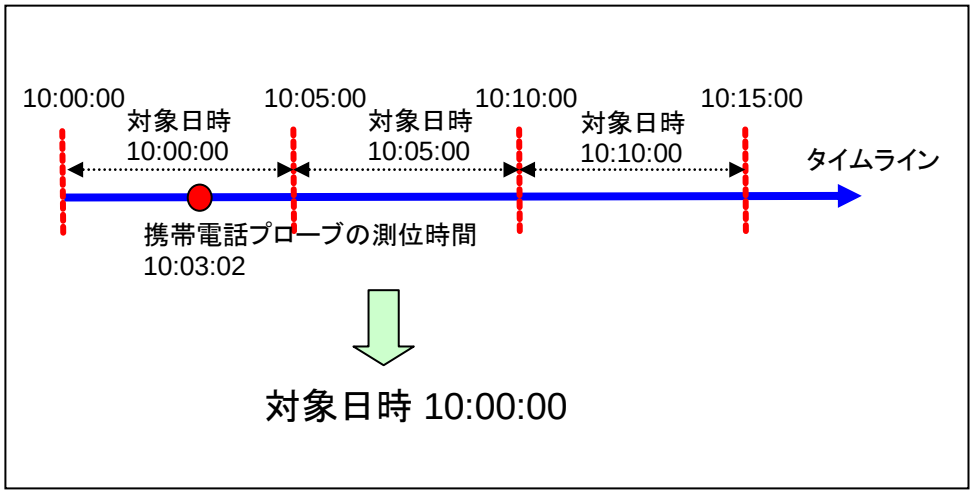


図 2-8「対象日時の計算」の処理イメージ

(3) 入力データの間引き

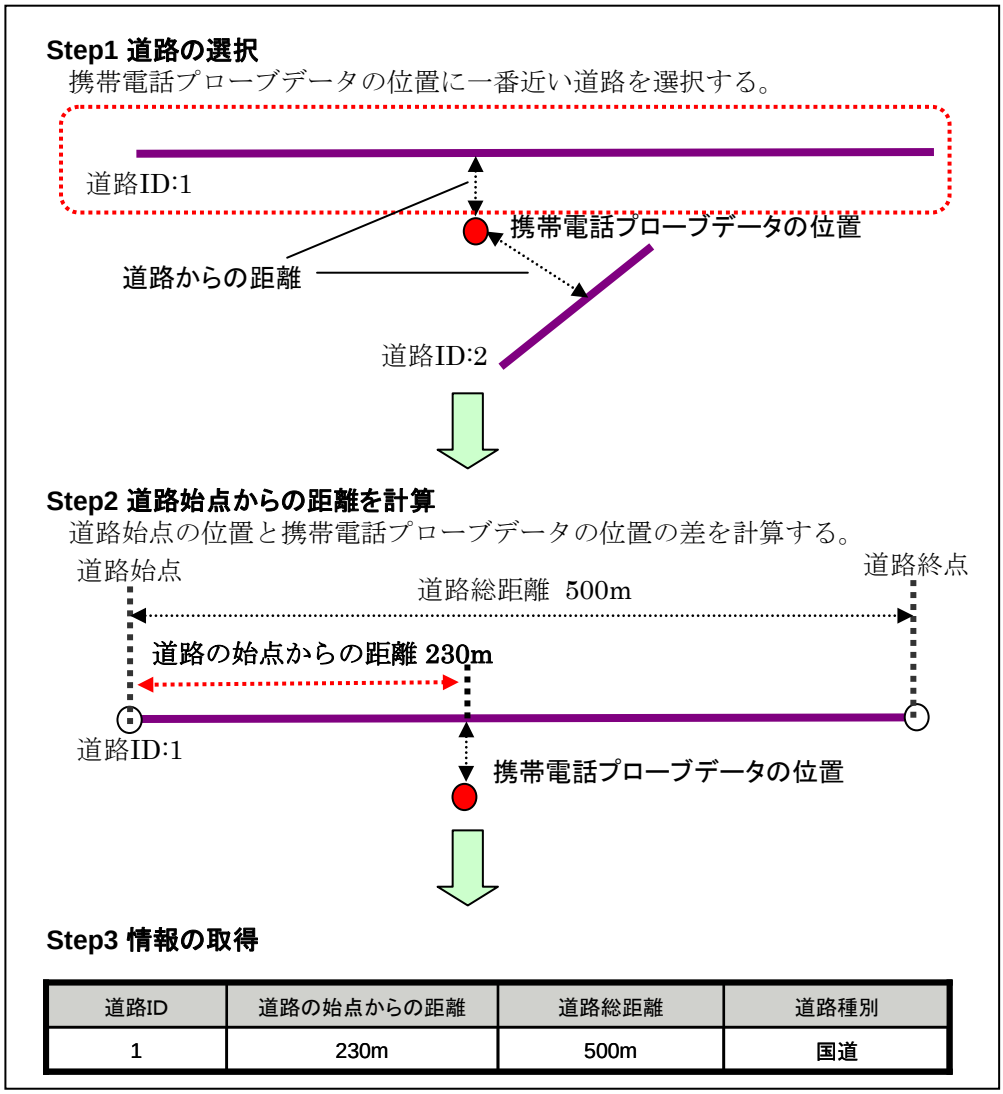
携帯電話プローブデータは、各端末から数秒おき程度の頻度で送信されてくるため、出力する速度情報の「区間ごとの速度」の区間の間隔によっては情報量が過剰となる。そのため、入力された携帯電話プローブデータを一定間隔おきに間引く処理を行う。

携帯電話プローブファイルには、到着時刻順に携帯電話プローブデータが格納されているため、本処理の入力では、さまざまなプローブ ID のデータが混在する。そこで、間引き処理では、プローブ ID と対象日時ごとにプローブデータをまとめ、プローブデータの連続性を保ちながら間引く処理を行う。なお、間引きの間隔については「区間ごとの速度」の区間の間隔に応じて適切な値を設定する。

(4) 道路の選択

携帯電話プローブデータの測位位置から道路を選択し、道路 ID と道路総距離、道路の始点からの距離、道路種別を取得する。

道路を選択する際は、携帯電話プローブデータの測位位置と比較し、最も近い位置にある道路を選択する。選択に利用する道路は、設定に応じて一般道路と主要道路を切り替える。道路の始点からの距離は、選択された道路の始点から、携帯電話プローブデータの測位位置までの距離より計算する。図 2-9 に「道路の選択」の処理イメージを示す。



道路ID	道路の始点からの距離	道路総距離	道路種別
1	230m	500m	国道

図 2-9「道路の選択」の処理イメージ

(5) 進行方向の決定

同じプローブ ID の携帯電話プローブデータを道路ごとに道路の始点からの距離順に並べ、携帯電話プローブデータを送信してきた端末が道路のどち

らの方向に進んでいるかを判定する。図 2-10 に「進行方向の決定」の処理イメージを示す。

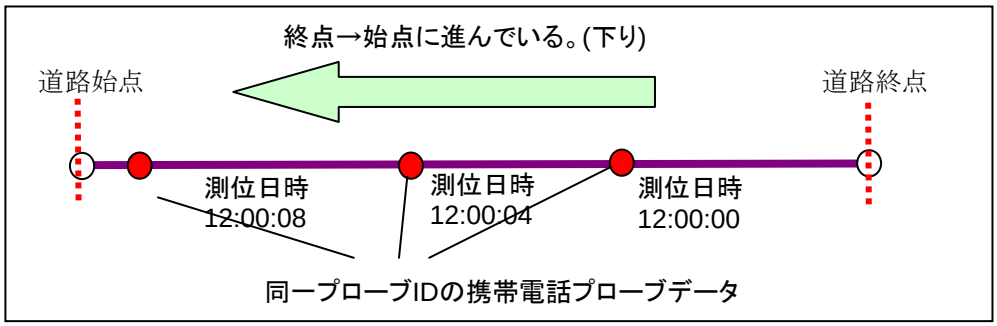
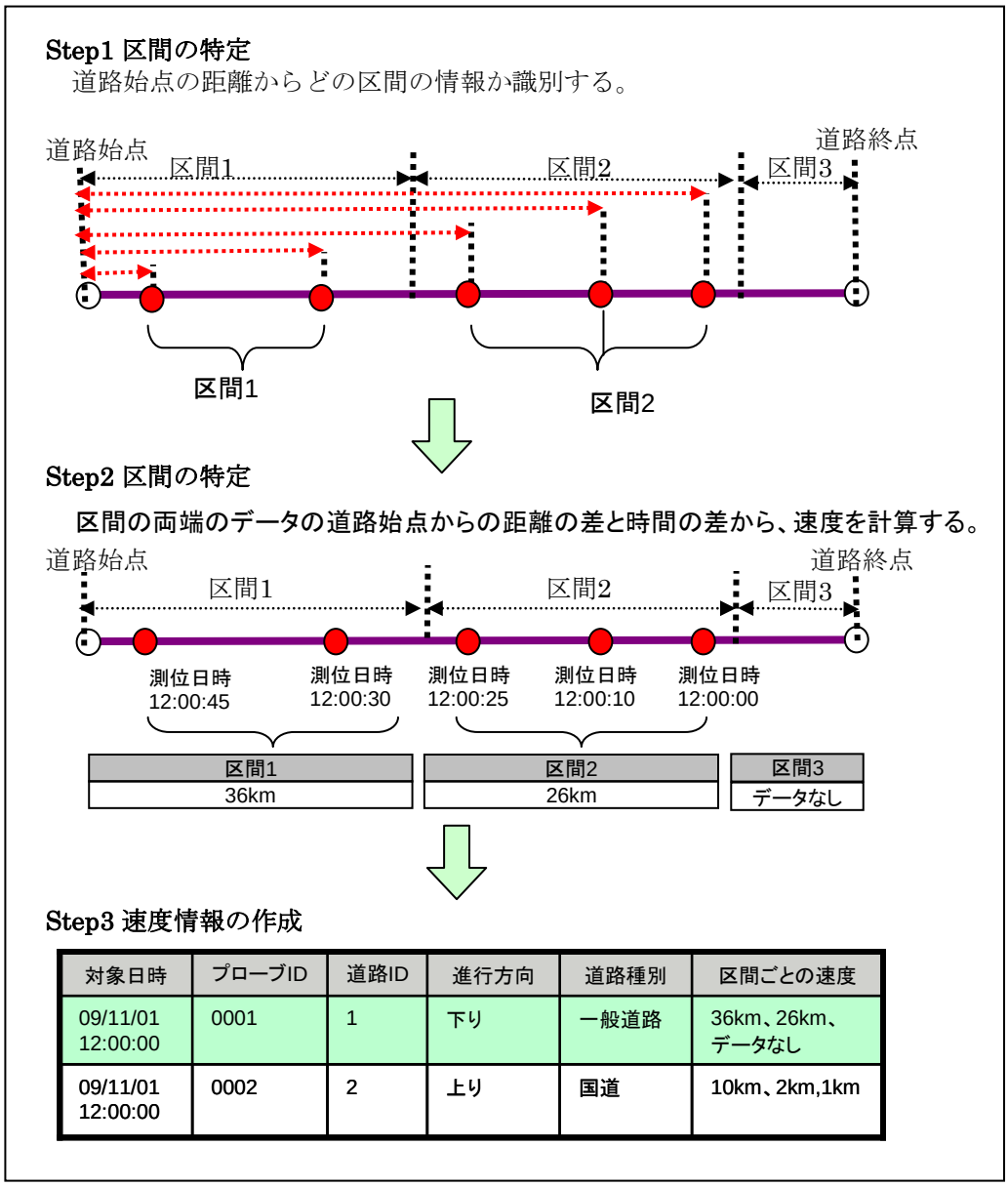


図 2-10「進行方向の決定」の処理イメージ

(6) 速度の計算

道路を道路の始点から一定間隔に区切り、各区間にある携帯電話プローブデータに紐づく道路の始点からの距離と、携帯電話プローブデータの測位時間より、「区間ごとの速度」を計算する。また、計算結果を一定時間、ユーザ、道路の進行方向ごとにまとめて速度情報を作成する。なお、道路を区切る間隔は、設定に応じて切り替えを行う。図 2-11 に「速度の計算」の処理イメージを示す。



対象日時	プローブID	道路ID	進行方向	道路種別	区間ごとの速度
09/11/01 12:00:00	0001	1	下り	一般道路	36km、26km、データなし
09/11/01 12:00:00	0002	2	上り	国道	10km、2km,1km

図 2-11「速度の計算」の処理イメージ

2.3.1.3 解析結果集計

ここでは図 2-2 の解析結果集計の処理について解説する。解析結果集計で行う処理のフローを図 2-12 に示す。

解析結果集計では、タクシープローブ解析と携帯電話プローブ解析で作成した速度情報を特定の単位でまとめ、各道路での混雑度を計算し、短時間渋滞情報、渋滞統計を作成する。

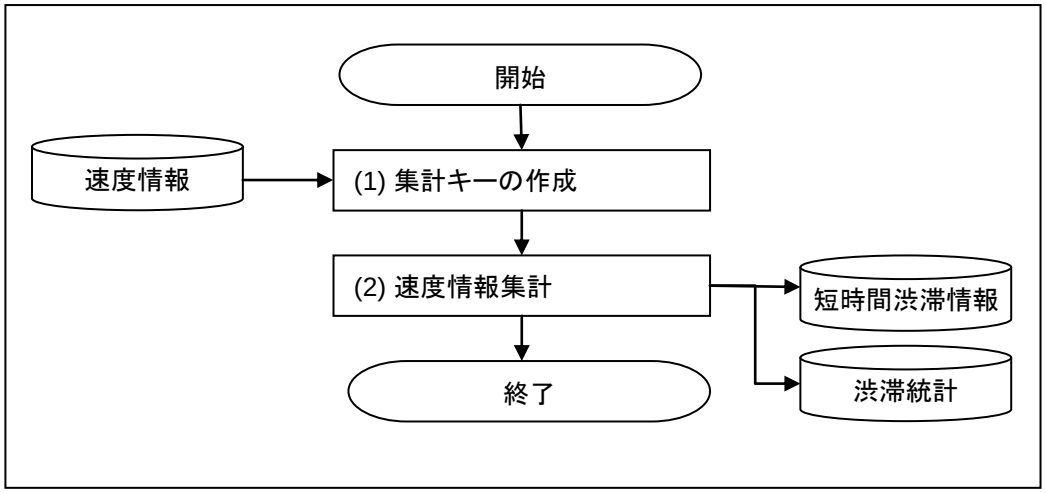


図 2-12 解析結果集計の処理フロー

(1) 集計キーの作成

タクシープローブ解析、携帯電話プローブ解析で作成した速度情報を読込み、次処理でまとめるための集計キーを作成し、速度情報に付与する。集計キーは短時間渋滞情報出力用の短時間集計キーと、渋滞統計出力用の特異日集計キー、季節集計キーの 3 種類とし、アプリケーション実行前に作成する情報に合わせた集計キーを指定する。

- 短時間集計キー
  短時間集計キーは、対象日時、道路 ID、進行方向ごとに速度情報をまとめ、短時間渋滞情報を作成するためのキーである。
- 特異日集計キー
  特異日集計キーは、速度情報の対象日時が特異日に設定されている日である場合に、特異日、対象日時、道路 ID、進行方向ごとに速度情報をまとめ、渋滞統計を作成するためのキーである。
- 季節集計キー
  季節集計キーは、速度情報を春、夏、秋、冬に分類し、季節、対象日時に対応する曜日、対象日時、道路 ID、進行方向ごとに速度情報をまとめ、渋滞統計を作成するためのキーである。

(2) 速度情報集計

(1)の集計キーごとの速度情報を集約し、平均速度を計算する。また、対象道路の道路種別と計算した平均速度から混雑度を判定し、短時間渋滞情報、渋滞統計を作成する。短時間渋滞情報、渋滞統計の区分けは(1)の集計キーにより分類する。図 2-13 に速度情報集計の処理イメージを示す。

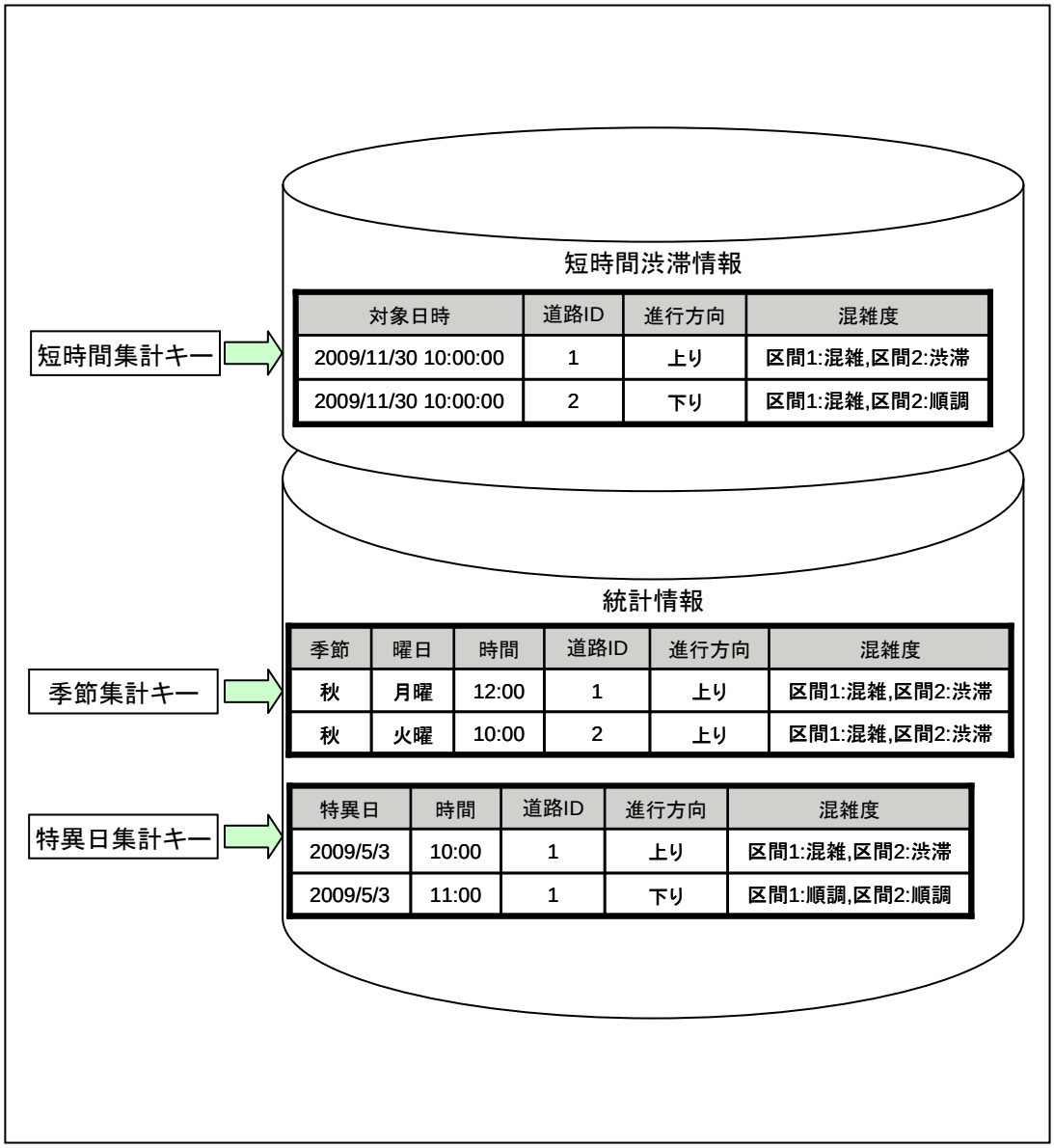


図 2-13「速度情報集計」の処理イメージ

2.3.2 MapReduce 設計

本項では、2.3.1 で解説した渋滞解析の処理フローを使い、MapReduce 設計の方法について解説する。

Map 処理、Reduce 処理で行う処理と入出力データの決定方法には、開発するアプリケーションに応じていろいろな方法が考えられるが、ここでは渋滞解析アプリケーションの開発を事例として説明する。

MapReduce では並列実行時の処理間のデータの依存関係により、処理を Map 処理と Reduce 処理に分割できる。そこで渋滞解析アプリケーションでは、データの依存関係を示すのに適しているデータフロー図(Gane-Sarson 表記法)を利用し図 2-14 の順に MapReduce で必要な項目に分類する手法を採用した。

ここでは本手法を使って、2.3.1.1 で解説したタクシープローブ解析の処理を例に各項目を決定していく。

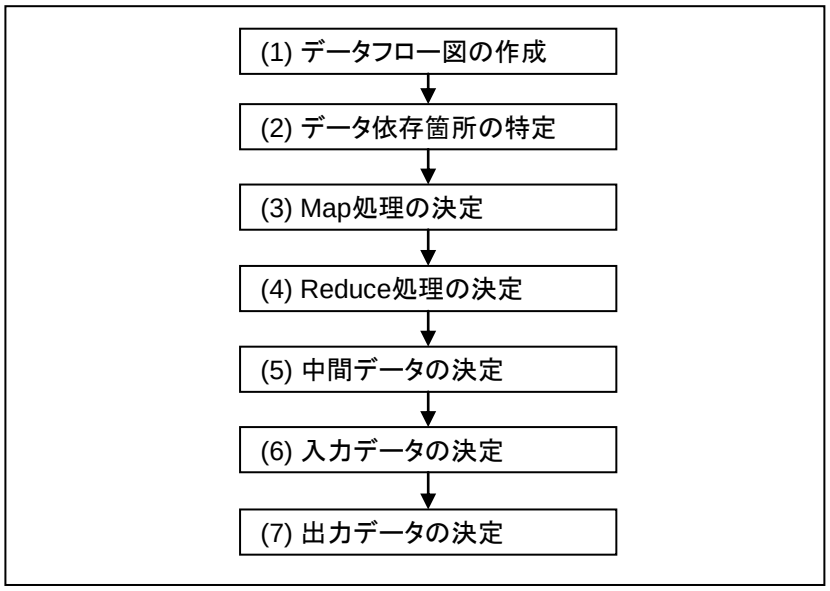


図 2-14 MapReduce 適用の流れ

なお、本項でのデータフロー図は、基本的には Gane-Sarson 表記法を利用し記載しているが、MapReduce アルゴリズム適用の解説のため一部の表記を一般の表記内容と変更している。本項におけるデータフロー図の表記を図 2-15 に示す。

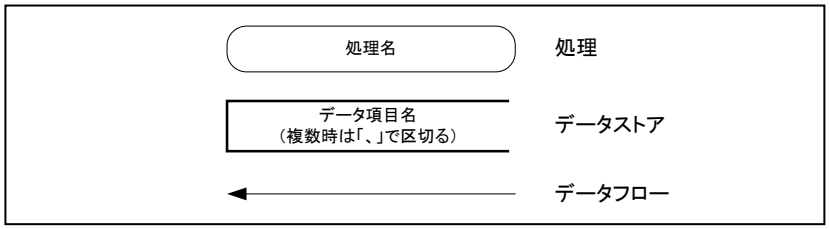


図 2-15 データフロー図の表記

(1) データフロー図の作成

処理フローを基に各処理の入出力を明確にするために、各処理の入出力をデータストアとしたデータフロー図を作成する。データストアには各処理の入出力となる項目を具体化し、その項目を記述する。

以降では、タクシープローブ解析を例に、処理フローからデータフロー図を作成する方法について解説する。図 2-16 は 2.3.1.1 で解説したタクシープローブ解析の処理フローの再掲である。

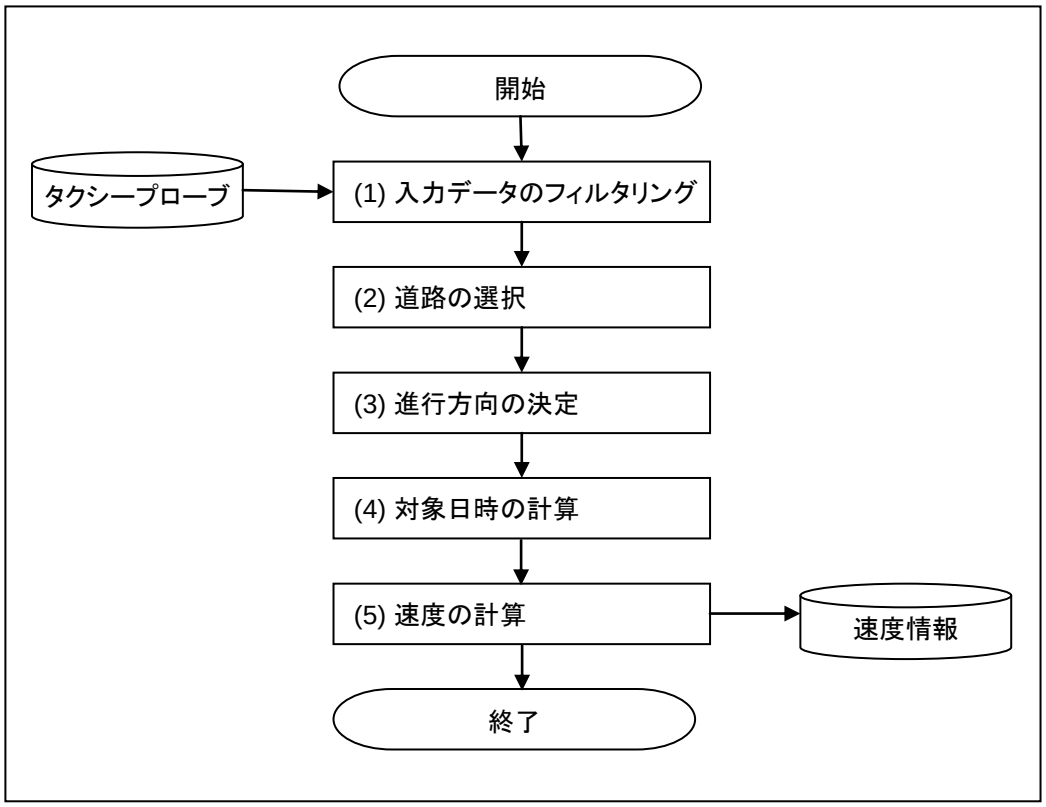


図 2-16 タクシープローブ解析の処理フロー（再掲）

本処理フローからデータフロー図を作成する。ここでは「道路の選択」の入出力を例に解説する。まず、「道路の選択」の入力について考える。「道路の選択」の入力は、「入力データのフィルタリング」の結果となる。「入力データのフィルタリング」の結果は、2.3.1.1 の解説より、フィルタリングされなかったデータがそのまま出力となることがわかる。つまり、「入力データのフィルタリング」の入力となるタクシープローブデータの各項目が、そのまま「道路の選択」の入力となる。そのためここでは、タクシープローブデータの各項目を「道路の選択」の入力のデータストアに記述する。

次に「道路の選択」の出力について考える。「道路の選択」の出力は、2.3.1.1

の解説よりタクシープローブデータに、「道路の選択」で新たに取得した情報(道路 ID、道路総距離、道路始点からの距離、道路種別)を追加したものとわかる。そのため、タクシープローブデータの各項目と、道路 ID、道路総距離、道路始点からの距離、道路種別を「道路の選択」のデータストアに記述する。本作業を全処理の入出力について行いデータフロー図を作成する。作成したデータフロー図を図 2-17 に示す。

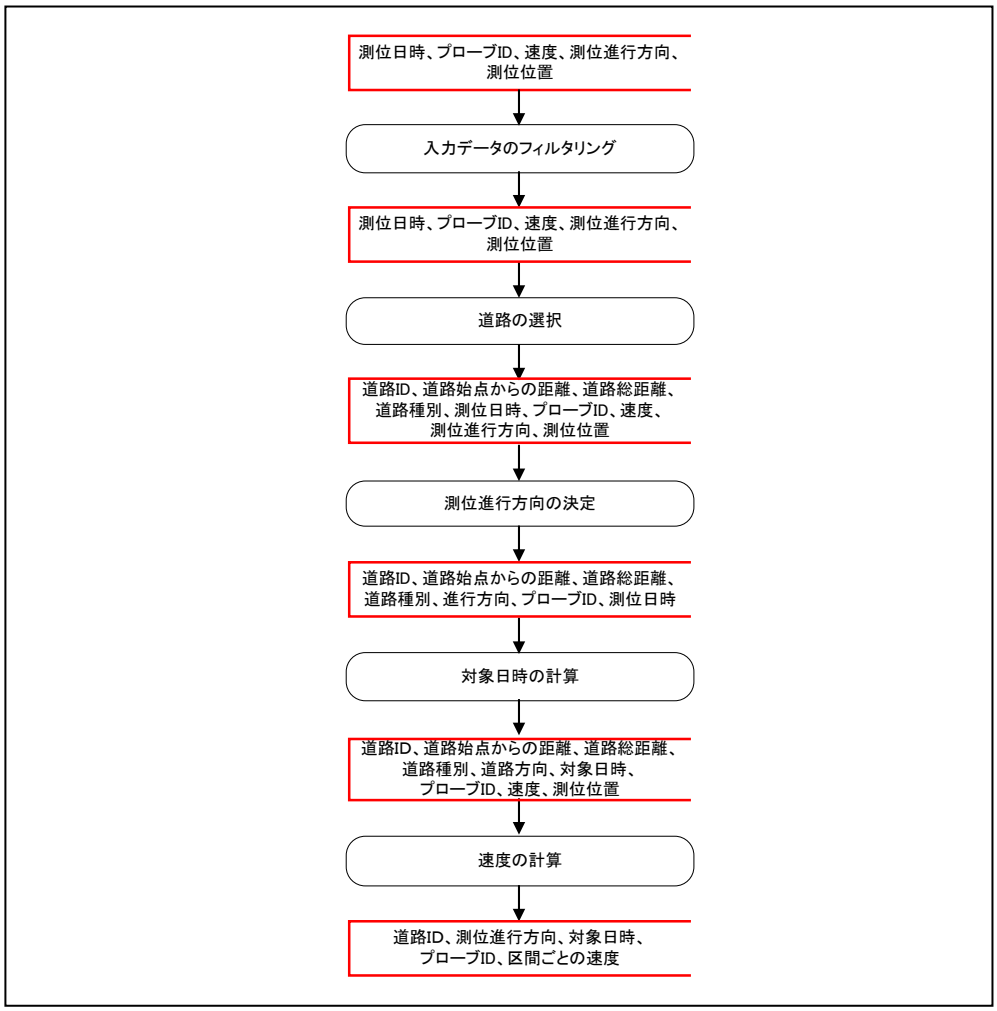


図 2-17 タクシープローブ解析のデータフロー図

(2) データ依存箇所の特定

図 2-17 のデータフロー図を 2 つ並べ、分散処理を行った場合のデータの流れを考える。各データフロー図の入力データにデータストアの 1 レコードがランダムに投入された場合に、上から順番に他方のデータが必要となるポイントがないか考えていき、他方のデータが必要となるポイントがあった場合は、そこに新たなデータフローを追加する。ここで追加した新たなデータフローがデータ依存箇所となる。

では、実際にタクシープローブ解析で各フローにデータが投入された場合を考える。「入力データのフィルタリング」は、入力データを 1 レコードずつ処理するので他方への依存はない。「道路の選択」では、入力データの位置情報のみで選択するので他方への依存はない。以上のように上から順に考えていく。すると「速度の計算」で道路 ID、進行方向、対象日時、プローブ ID ごとにまとめる必要があり、双方のデータが必要となることがわかる。つまり、ここがデータ依存箇所である。ここまでの流れを図 2-17 のデータフロー図に追加すると図 2-18 となる。

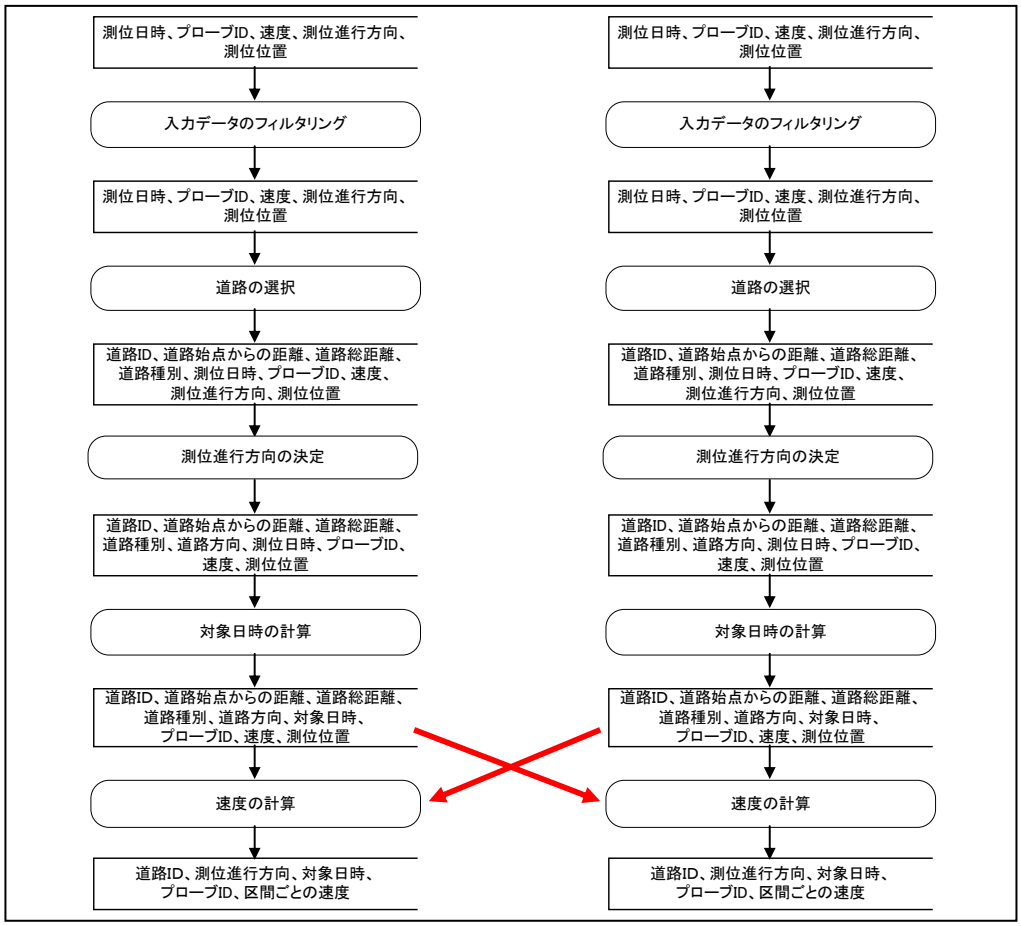


図 2-18 タクシープローブ解析のデータ依存箇所

(3) Map 処理の決定

図 2-18 のデータフロー図を基に Map 処理で行う処理を決定する。Map 処理となる部位は他の処理に影響しない独立した処理とする必要がある。そこで、図 2-18 のデータフロー図よりその処理を考える。

図 2-18 のデータフロー図では、上から順に各処理の他の処理に影響する箇所がないか確認して、データ依存箇所を定義している。そのため、データ依存箇所より上の処理は、他の処理に影響しない独立した処理と考えることができる。つまり、データ依存箇所より上の処理がすべて Map 処理となる。Map 処理となる部分を示したものを図 2-19 に示す。

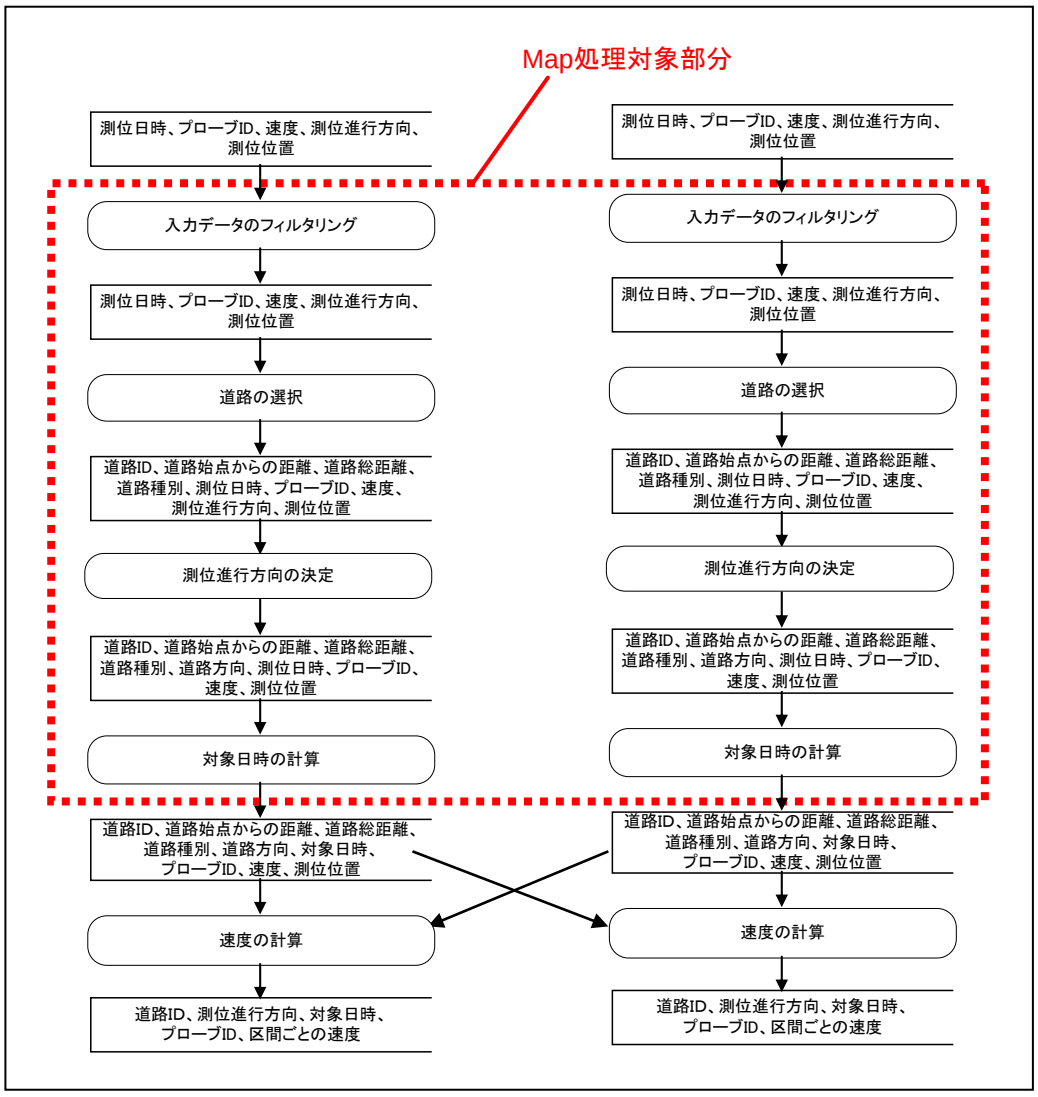


図 2-19 タクシープローブ解析の Map 処理

(4) Reduce 処理の決定

図 2-18 のデータフロー図を基に Reduce 処理で行う処理を決定する。Reduce 処理となる部位は並列に処理を実行した際に双方の処理の出力をまとめる処理とする必要がある。そこで、図 2-18 のデータフロー図よりその処理を考える。

図 2-18 のデータフロー図は、並列に処理を実行した際のデータフローを示しているため、双方の処理の出力をまとめる箇所は、データフローが他方の処理へ伸びている箇所となる。以上より、Reduce 処理の対象部分はデータ依存箇所の後の処理が該当することが分かる。Reduce 処理となる部分を示したものを図 2-20 に示す。

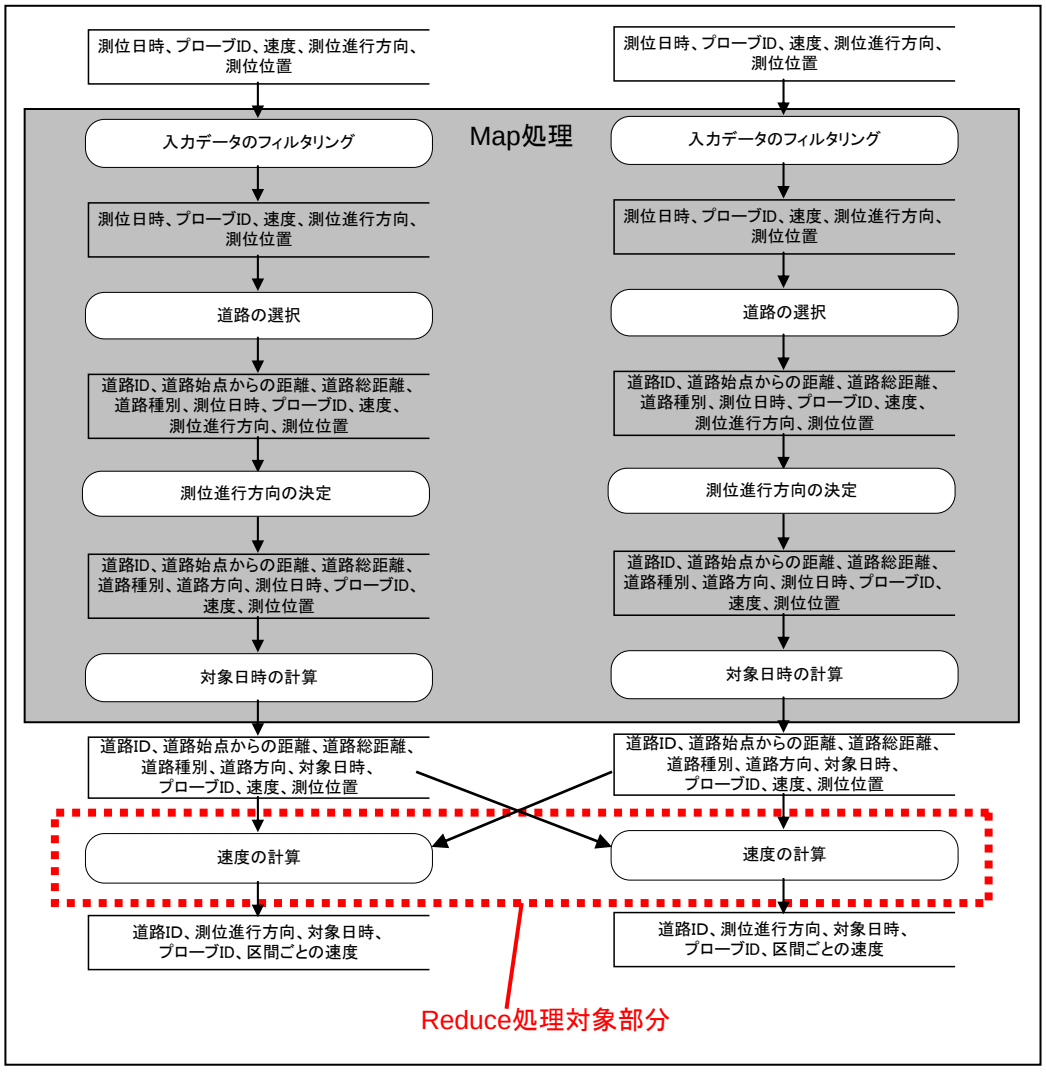


図 2-20 タクシープローブ解析の Reduce 処理

(5) 中間データの決定

図 2-18 のデータフロー図を基に中間データの Key・Value を決定する。まず Key・Value となる候補を決定するために、データフロー図からデータストアの特定を行う。中間データは Map 処理の出力であり、Reduce 処理の入力でもあるため Map 処理、Reduce 処理の間のデータストアが候補となる。

次に候補としたデータストアを Key と Value に分類する。Key は、Reduce 処理で中間データを Key 単位に集約し、処理を行う際に利用される。そのため、ここでは Reduce 処理において、どのような単位に集約して処理を行う必要があるかを考える。そこで Reduce 処理となっている「速度の計算」について、2.3.1.1 の(5)を振り返る。「速度の計算」では道路 ID、進行方向、対象日時、プローブ ID でまとめて処理するとある。つまり、タクシープローブ解析における Key は道路 ID、進行方向、対象日時、プローブ ID であることがわかる。Map 処理の出力となる Key・Value を示したものを図 2-21 に示す。なお、Key についてはアンダーラインを引いて表している。

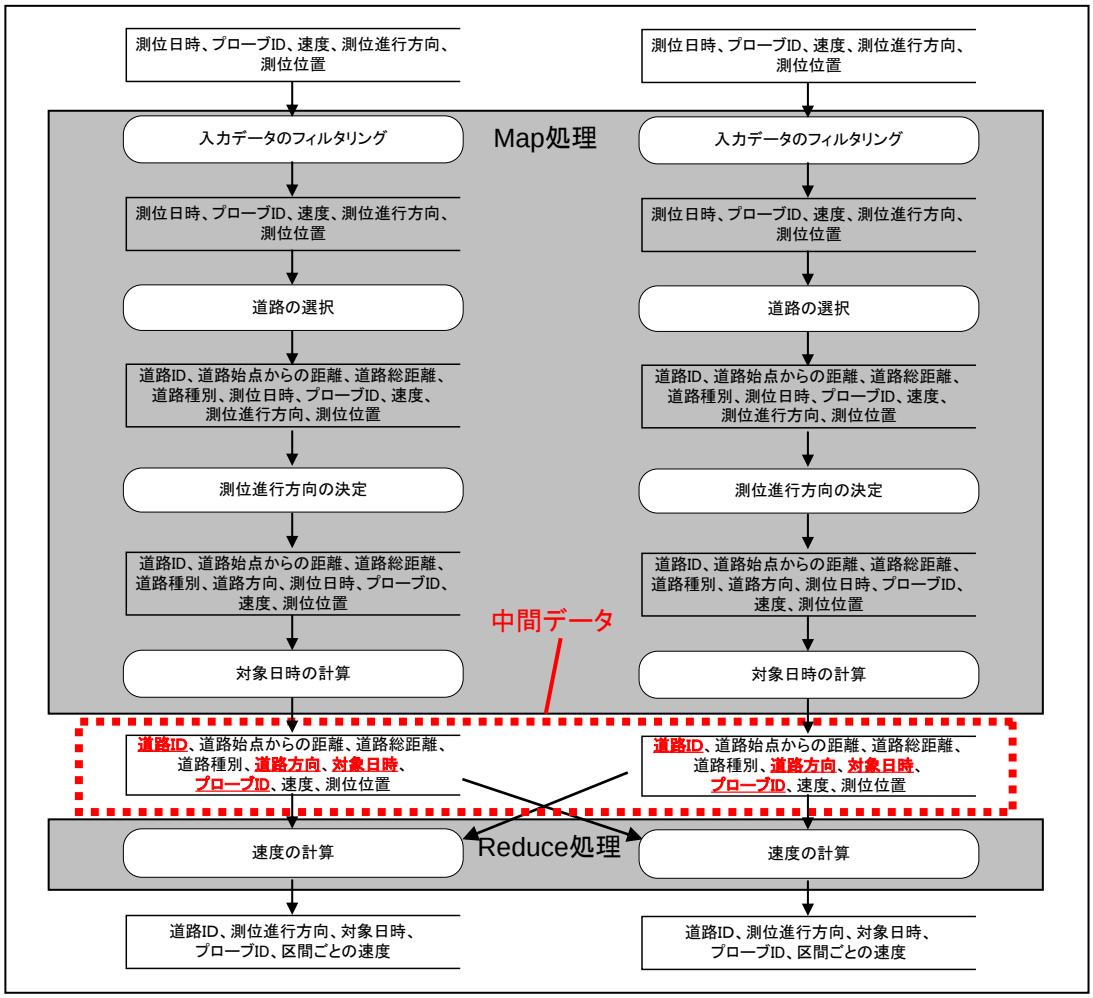


図 2-21 タクシープローブ解析の中間データ

(6) 入力データの決定

図 2-18 のデータフロー図を基に入力データの Key・Value を決定する。まず Key・Value となる候補を決定するために、データフロー図からデータストアの特定を行う。入力データは Map 処理の入力となるため、Map 処理の入力となるデータストアが、入力データの候補となる。

次に候補としたデータストアの項目を Key と Value に分類する。入力の Key は、入力ファイルからデータを読込む際に特別な処理をしないのであれば不要である。タクシープローブ解析では、入力ファイルからデータを読込む際に特別な操作はしていないため、Key 設計は行わない。Value についてはすべての入力データの候補すべてとなる。入力データを示したものを図 2-22 に示す。

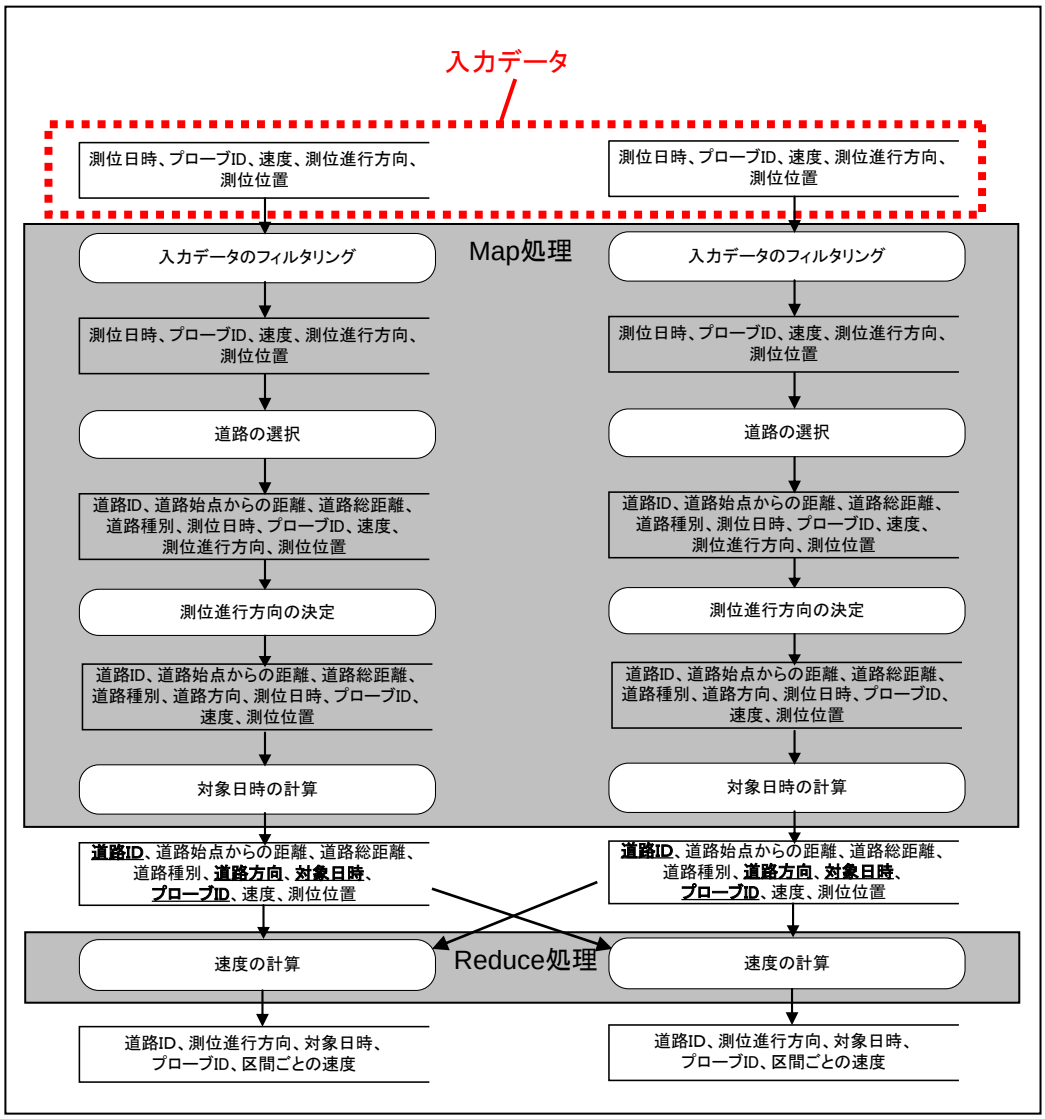


図 2-22 タクシープローブ解析の入力データ

(7) 出力データの決定

図 2-18 のデータフロー図を基に出力データの Key・Value を決定する。まず Key・Value となる候補を決定するために、データフロー図からデータストアの特定を行う。出力データは Reduce 処理の出力となるため、Reduce 処理の出力となるデータストアが、出力データの候補となる。

次に候補としたデータストアの項目を Key と Value に分類する。出力データの Key は、出力時に特別な処理をしないのであれば不要である。タクシープローブ解析でも Reduce 処理の出力時に特別な操作はしていないため、Key 設計は行わない。Value については出力データの候補すべてとなる。出力データを示した図を図 2-23 に示す。

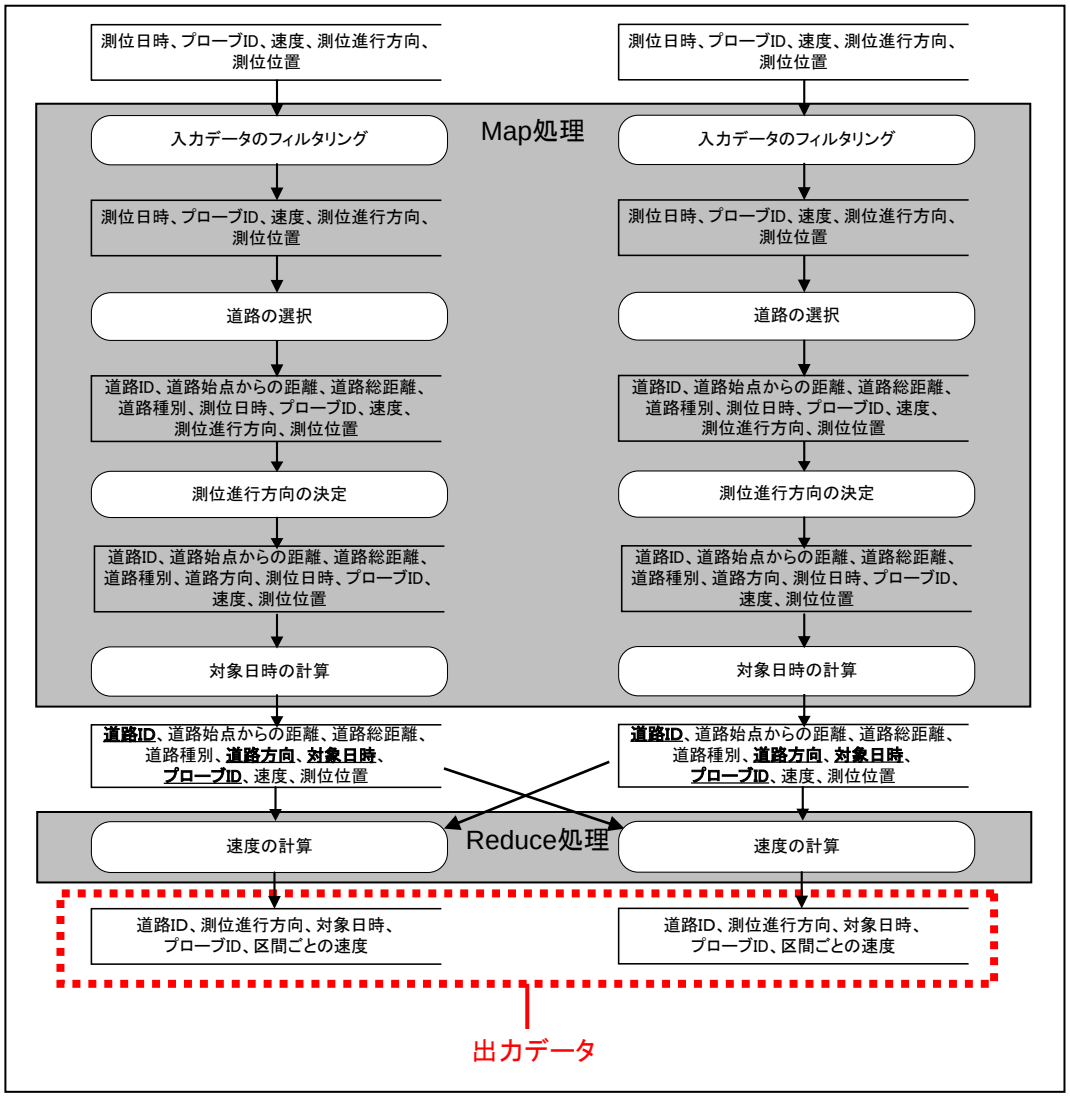


図 2-23 タクシープローブ解析の出力データ

以上で MapReduce の要素への分類は完了である。ここまでの結果をまとめると、図 2-24 のとおりとなる。MapReduce ジョブの Map 処理、Redcue 処理で行う処理と、入力データ、中間データ、出力データのデータ項目が定まっている。

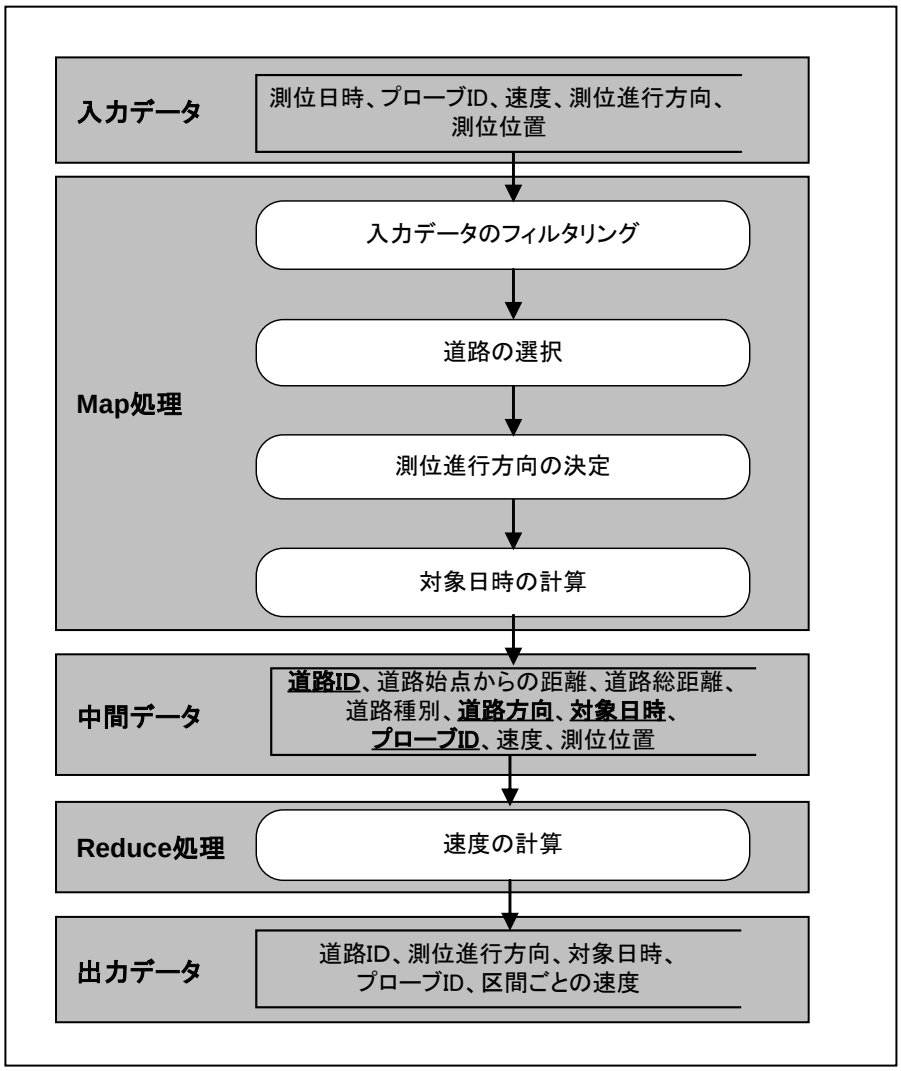


図 2-24 タクシープローブ解析の MapReduce の構成

ここではタクシープローブ解析について解説してきたが、携帯電話プローブ解析、解析結果集計についても、同様の方法で MapRedcue 処理に分類することができる。参考に MapReduce に分類したものを図 2-25、図 2-26 に掲載する。

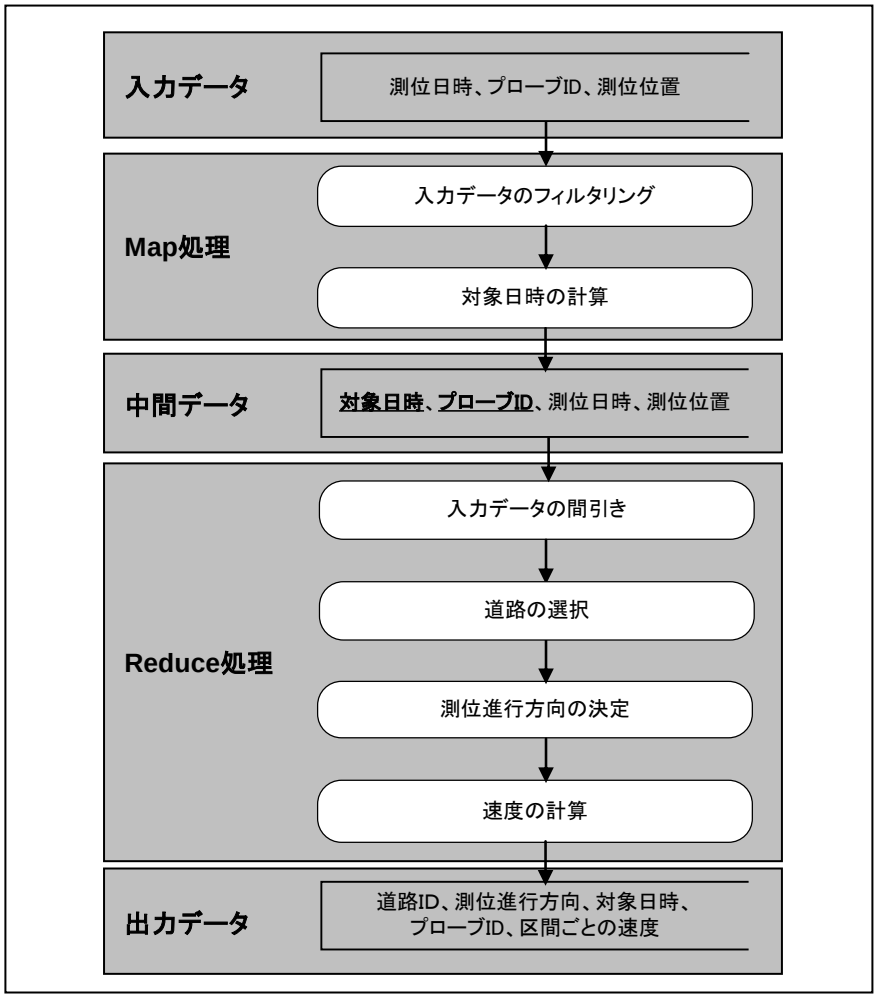


図 2-25 携帯電話プローブ解析の MapReduce の構成

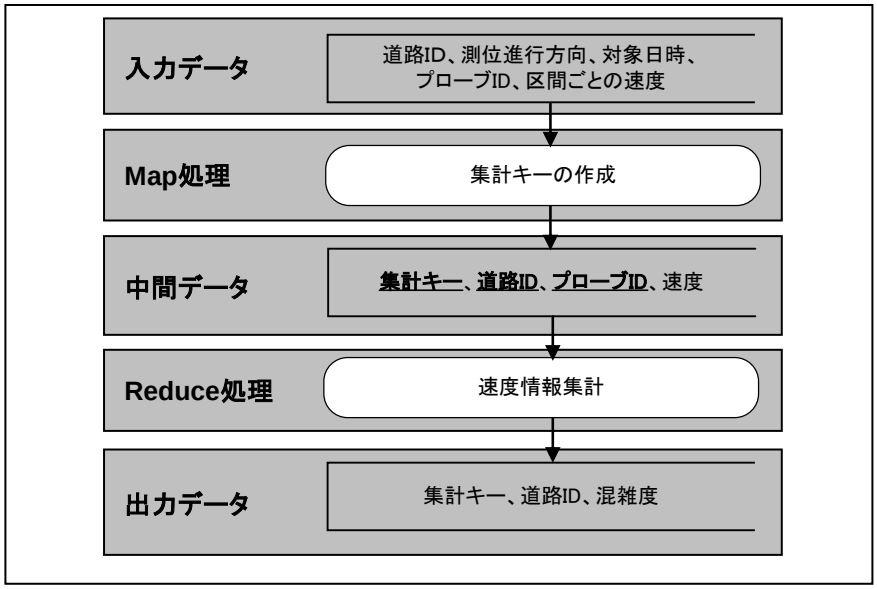


図 2-26 解析結果集計の MapReduce の構成

2.4 MapReduce アプリケーションの実装

本節では、タクシープローブ解析の実装を例に MapReduce の実装方法について解説する。

MapReduce の実装を行うためには、MapReduce で必要な各クラスを定義する必要がある。MapReduce で必要となる各クラスの定義は、2.3.2 の MapReduce の構成を基に作成する。タクシープローブ解析の MapReduce の構成から作成したクラス構成を図 2-27 に示す。本節では、ここで定義したクラスを使って実装の解説をしていく。

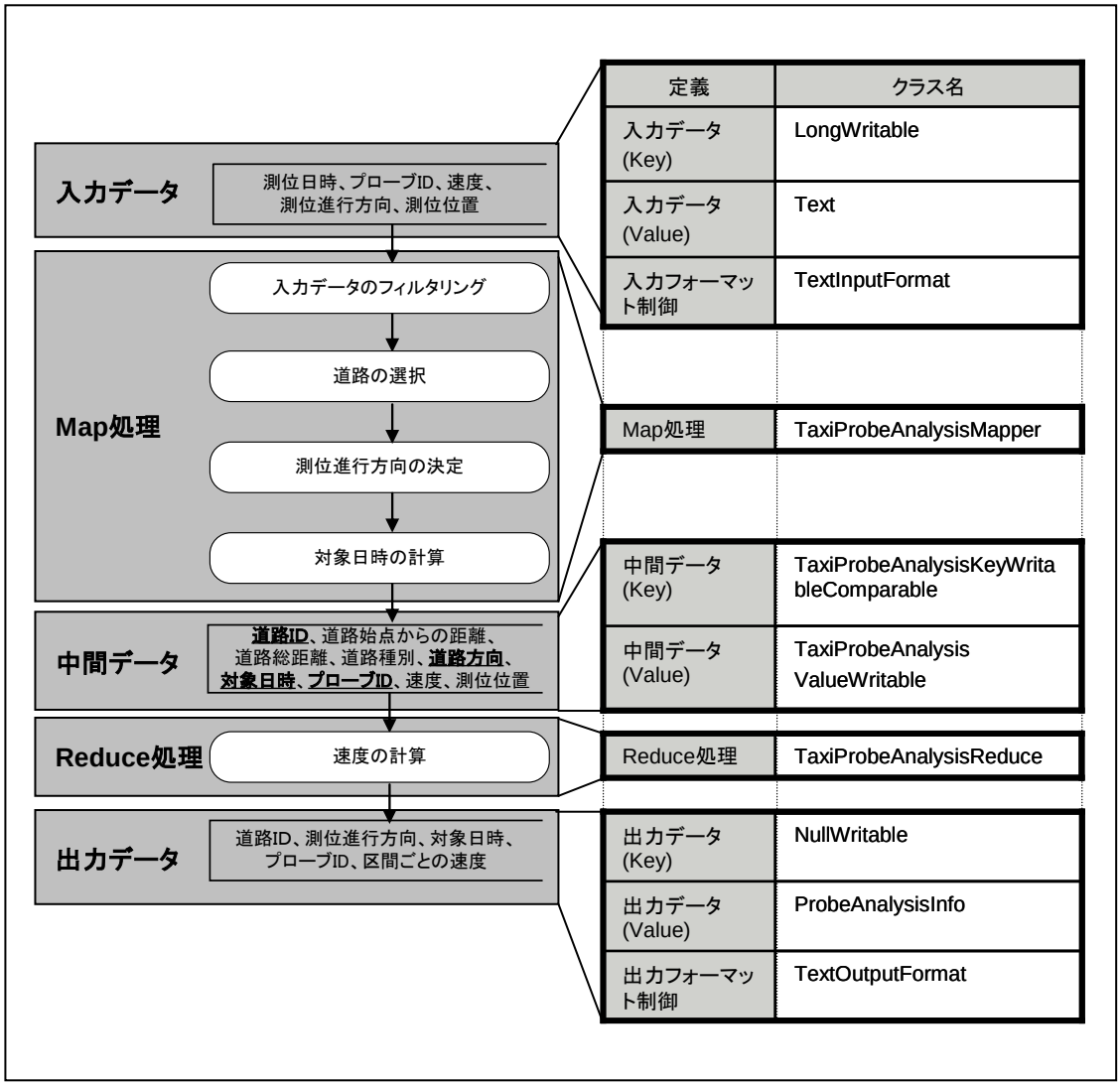


図 2-27 タクシープローブ解析のクラス構成

2.4.1 Map 処理の実装

本項では、タクシープローブ解析の Map 処理と、Map 処理の入出力に関わるクラスの実装について解説する。

Map 処理

タクシープローブ解析の Map 処理を実装した結果を図 2-28、図 2-29 に示す。

```
24. public class TaxiProbeAnalysisMapper
25.         extends
26.         Mapper<LongWritable, Text, TaxiProbeAnalysisKeyWritableComparable,
27.                                         TaxiProbeAnalysisValueWritable> {
28.
29.     /**
30.      * 道路検索クラス
31.      */
32.     private AbstractNeigbhorhoodRoadFinder finder;
33.
34.     /**
35.      * ログ出力クラス
36.      */
37.     private Log log = LogFactory
38.             .getLog(TaxiProbeAnalysisMapper.class);
39.
40.     /**
41.      *設定値取得KEY
42.      */
43.     public final static String KEY_ROAD_SEARCH_CLASS = "road-finder-type";
44.
45.     @Override
46.     protected void setup(Context context)
47.             throws IOException, InterruptedException {
48.         try {
49.
50.             Configuration conf = context.getConfiguration();
51.
52.             // 道路検索クラスの設定(主要道路、一般道路の切り替えを行う)
53.             NeigbhorhoodRoadFinderFactory finderFactory =
54.                                     new NeigbhorhoodRoadFinderFactory();
55.             // Configurationクラスから設定値を取得
56.             String roadSearchClassName = conf
57.                     .get(TaxiProbeAnalysisMapper.KEY_ROAD_SEARCH_CLASS);
58.             finderFactory.setClassName(roadSearchClassName);
59.             finder = finderFactory.createInstance();
60.         } catch (Exception e) {
61.             log.error("初期化に失敗しました。", e);
62.             throw new IOException("初期化に失敗しました。");
63.
64.         }
65.     }
66.
```

① ②

図 2-28 Map クラスの実装イメージ(1/2)

```
67.     @Override
68.     protected void map(LongWritable key, Text value,
69.             Context context) throws IOException,
70.             InterruptedException {
71.         try {
72.             TaxiProbeCreater probeCreater = new TaxiProbeCreater();
73.             TaxiRoadMatchMapLogic taxiRoadMatchMapLogic =
74.                                     new TaxiRoadMatchMapLogic(finder);
75.             TaxiProbeData probe = probeCreater.parse(
76.                     value.toString());
77.
78.             boolean isSearch = taxiRoadMatchMapLogic
79.                     .searchRoadMap(probe);
80.             if (!isSearch) {
81.                 context.getCounter(
82.                         TaxiProbeAnalysisMapper.class
83.                                 .getSimpleName(),
84.                         "道路との一致なし").increment(1);
85.                 return;
86.             }
87.
88.             TaxiProbeAnalysisKeyWritableComparable hadoopKey =
89.                                         taxiRoadMatchMapLogic.getKey();
90.
91.             TaxiProbeAnalysisValueWritable hadoopValue =
92.                                       taxiRoadMatchMapLogic.getValue();
93.             context.write(hadoopKey, hadoopValue);
94.         } catch (FormatException e) {
95.             // タクシープローブのフォーマット不正の場合
96.             context.getCounter(
97.                     TaxiProbeAnalysisMapper.class
98.                             .getSimpleName(), "フォーマットエラー")
99.                     .increment(1);
100.        } catch (FilterException e) {
101.            // 緯度、経度が日本国内でない場合
102.            context.getCounter(
103.                    TaxiProbeAnalysisMapper.class
104.                            .getSimpleName(),
105.                    "国外データ").increment(1);
106.        }
107.    }
108. }
```

③

図 2-29 Map クラスの実装イメージ(2/2)

①: Map 処理は、Mapper を継承する必要がある。Mapper の仮型引数には、入力データの Key の型、Value の型と、中間データの Key の型、Value の型を順に指定する。入力データの Key の型、Value の型は、入力フォーマット制御クラスを TextInputFormat としたため、TextInputFormat の仕様に従い LongWritable、Text としている。中間データの Key の型、Value の型は、後で解説する中間データの TaxiProbeAnalysisKeyWritableComparable と TaxiProbeAnalysisValueWritable を指定している。

②: setup()メソッドでは、Map 処理で必要な初期化処理を記述する。
Map 処理で行う「道路の選択」処理で、設定に応じて対象とする道路を切り替える必要がある。ここでは変更のための値を次の順で取得する。

A) Context の getConfiguration()メソッドより Configuration を取得する。
B) Configuration の get()メソッドよりキーを指定し、該当する値(道路種別)を取得する。

なお、Configuration の情報は後述する MapReduce ジョブでキーと値のセットを設定している。

③: map()メソッドでは、2.3.2 の設計で Map 処理の対象とした処理を記述する。
map()メソッドの引数の Key は、2.3.2 の設計で入力データでは不要としためここでは利用しない。Value には、TextInputFormat の setPaths()メソッドで指定したパスのテキストファイルの 1 行が、文字列で入力される。ここでは、これを使い次の順で処理を行う。

A) map()メソッドの引数の value 値を使い「入力データのフィルタリング」「道路の選択」、「進行方向の決定」、「対象日時の計算」の処理を行う。
B) 処理の結果を中間データの Key(TaxiProbeAnalysisWritableComparable)と、Value(TaxiProbeAnalysisValueWritable)に格納する。
C) Key と Value を Context の write()メソッドに指定し、出力する。

中間データの実装

中間データの Key・Value は、Hadoop に用意されているデータ型の Text を利用しても作成できる。しかし、データ型を新たに定義すると、中間データで保持する属性に合うデータ型でシリアライズする処理を記述できるので、Text を利用する場合に比べて、データ転送量の削減による処理速度の向上が期待できる。そのため、ここではデータ型を新たに定義する。

Key の実装

2.3.2 で設計したキーのデータ型を新たに定義した結果を図 2-30、図 2-31 に示す。

```
21.  public class TaxiProbeAnalysisKeyWritableComparable
22.          implements
23.          WritableComparable<TaxiProbeAnalysisKeyWritableComparable> {
24.
25.      /**
26.       * 道路ID
27.       */
28.      private int roadID;
45.      /**
46.       * 道路IDを設定する。
47.       *
48.       * @param roadID
49.       */
50.      public void setRoadID(int roadID) {
51.          this.roadID = roadID;
52.      }
53.
54.      /**
55.       * 道路IDを取得する。
56.       *
57.       * @return
58.       */
59.      public int getRoadID() {
60.          return roadID;
61.      }
117.     @Override
118.     public void write(DataOutput output) throws IOException {
119.
120.         // 道路IDのシリアライズ
121.         IntWritable roadIDInt = new IntWritable(this.roadID);
122.         roadIDInt.write(output);
123.
124.         // 進行方向のシリアライズ
125.         BooleanWritable roadDirectionBoolean = new BooleanWritable(
126.                 this.isRoadDirection);
127.         roadDirectionBoolean.write(output);
128.
129.         // プローブID
130.         Text probeIDText = new Text(this.probeID.toString());
131.         probeIDText.write(output);
132.
133.         // 対象日時のデシリアライズ
134.         Text timeNumberText = new Text(this.timeNumber
135.                 .toString());
136.         timeNumberText.write(output);
137.
138.     }
139.
```

①
②
③

図 2-30 中間データの Key クラスの実装イメージ(1/2)

```
140.     @Override
141.     public void readFields(DataInput input)
142.             throws IOException {
143.
144.         // 道路IDのデシリアライズ
145.         IntWritable roadIDInt = new IntWritable();
146.         roadIDInt.readFields(input);
147.         this.roadID = roadIDInt.get();
148.
149.         // 進行方向のデシリアライズ
150.         BooleanWritable roadDirectionBoolean = new BooleanWritable();
151.         this.isRoadDirection = roadDirectionBoolean.get();
152.
153.         // プローブIDのデシリアライズ
154.         Text probeIDText = new Text();
155.         probeIDText.readFields(input);
156.         this.probeID.setValue(probeIDText.toString());
157.
158.         // 対象日時のデシリアライズ
159.         Text timeNumberText = new Text();
160.         timeNumberText.readFields(input);
161.         this.timeNumber.setTimeNumber(timeNumberText
162.                 .toString());
163.
164.     }
```

④

```
170.     @Override
171.     public int compareTo(
172.             TaxiProbeAnalysisKeyWritableComparable dest) {
173.
174.         //道路IDの比較
175.         int roadIDResult = this.roadID-dest.roadID;
176.         if (roadIDResult != 0) {
177.             return roadIDResult;
178.         }
179.
180.         //進行方向の比較
181.         if (dest.isRoadDirection != this.isRoadDirection) {
182.             return 1;
183.         }
184.
185.         //プローブIDの比較
186.         int probeIDResult = probeID.compareTo(dest.probeID);
187.         if (probeIDResult != 0) {
188.             return probeIDResult;
189.         }
190.
191.         //対象日時の比較
192.         return timeNumber.compareTo(dest.timeNumber);
193.     }
194. }
```

⑤

図 2-31 中間データの Key クラスの実装イメージ(2/2)

①: 中間データの Key は、WritableComparable インターフェイスを実装する必要がある。WritableComparable インターフェイスの仮型引数には比較対象を指定する必要があるので、本クラスの型を指定している。

②: 属性には、2.3.2 で設計した中間データの Value の項目(道路 ID、進行方向、対象日時、プローブ ID)を定義している。また、各属性の値を設定、取得するために get()、set()メソッドを定義している。

③: write()メソッドでは、属性をシリアライズするための処理を記述する。各属性の型とシリアライズ方法を表 2-5 に示す。

表 2-5 write()メソッドでのシリアライズ方法一覧

順番	属性名	型	シリアライズ方法
1	道路 ID	int	IntWritable の write()メソッド
2	進行方向	boolean	BooleanWritable の write()メソッド
3	対象日時	String	Text の write()メソッド
4	プローブ ID	String	Text の wirte()メソッド

int(道路 ID)、boolean(進行方向)、String(対象日時、プローブ ID)は、それぞれの型に合った Hadoop の標準のデータ型クラスを利用し、シリアライズしている。この方法をとることで、Comparator の比較の際に標準のデータ型クラスの compare()メソッドが利用でき、比較が容易に実装できる。

④: read()メソッドでは、属性をデシリアライズするための処理を記述する。read()メソッドは write()メソッドでシリアライズした順、方法に則りデシリアライズさせる必要がある。各属性の型とデシリアライズの方法を表 2-6 に示す。

表 2-6 read()メソッドでのデシリアライズ方法一覧

順番	属性名	型	デシリアライズ方法
1	道路 ID	int	IntWritable の readFields()メソッド
2	進行方向	boolean	BooleanWritable の readFields ()メソッド
3	対象日時	String	Text の readFields ()メソッド
4	プローブ ID	String	Text の readFields ()メソッド

⑤: comparatorTo()メソッドでは、Key の属性値の比較を記述する。道路 ID、進行方向、対象日時、プローブ ID の比較を行い、比較結果を返却している。Hadoop では、Key 専用の Comparator が指定されていない場合は、本メソッドを利用して中間データの集約が行われる。

Comparator の実装

Comparator は、シリアライズされたバイナリデータの状態で Key の比較を行うクラスである。

Hadoop のデフォルトの Comparator では、Key に対応する Comarator が登録されていない場合は、デシリアライズを行ってから Key の comparatorTo()メソッドを呼び出し、比較を行う仕様となっている。そのため、Comparator を実装せずとも動作上は問題とならない。

しかし、デフォルトの Comparator では比較のたびにデシリアライズしているため、バイナリデータの状態で比較を行う Comparator と比較すると、処理速度が遅くなる。そのため Key を新たに定義した場合は Comparator を作成することが望ましい。

中間データの Key に該当する TaxiProbeAnalysisKeyWritableComparable の Comparator を実装した結果を図 2-32、図 2-33 に示す。

```
21.  public class TaxiProbeAnalysisKeyWritableComparable
22.          implements
23.          WritableComparable<TaxiProbeAnalysisKeyWritableComparable> {
24.
25.      static {
26.          WritableComparator
27.                  .define(
28.                          TaxiProbeAnalysisKeyWritableComparable.class,
29.                          new TaxiProbeAnalysisKeyComparator());
30.      }
31.
```

①

```
203.     public static class TaxiProbeAnalysisKeyComparator
204.             extends WritableComparator {
205.
206.         /**
207.          * Text型比較用のComparator
208.          */
209.         private static final Text.Comparator TEXT_COMPARATOR
210.                                                 = new Text.Comparator();
211.
212.         /**
213.          * Int型比較用のComparator
214.          */
215.         private static final IntWritable.Comparator INT_COMPARATOR
216.                                             = new IntWritable.Comparator();
217.
218.         /**
219.          * コンストラクタ
220.          */
221.         public TaxiProbeAnalysisKeyComparator() {
222.             super(
223.                     TaxiProbeAnalysisKeyWritableComparable.class);
224.         }
225.
```

②

③

図 2-32 中間データの Comparator クラスの実装イメージ(1/2)

```
226.         @Override
227.         public int compare(byte[] b1, int s1, int l1,
228.                 byte[] b2, int s2, int l2) {
229.
230.             try {
231.                 // 道路 IDの比較
232.                 int keyLength1 = Integer.SIZE / 8;
233.                 int keyLength2 = Integer.SIZE / 8;
234.                 int ret = INT_COMPARATOR.compare(b1, s1,
235.                         keyLength1, b2, s2,
236.                         keyLength2);
237.                 if (ret != 0) {
238.                     return ret;
239.                 }
240.
241.                 // 進行方向の比較
242.                 int startPos1 = s1 + keyLength1;
243.                 int startPos2 = s2 + keyLength2;
244.                 keyLength1 = 1;
245.                 keyLength2 = 1;
246.                 if (b1[startPos1] != b2[startPos2]) {
247.                     return 1;
248.                 }
249.
250.                 //プローブ IDの比較
251.                 startPos1 = startPos1 + keyLength1;
252.                 startPos2 = startPos2 + keyLength2;
253.                 keyLength1 = WritableUtils
254.                         .decodeVIntSize(b1[startPos1])
255.                         + readVInt(b1, startPos1);
256.                 keyLength2 = WritableUtils
257.                         .decodeVIntSize(b2[startPos2])
258.                         + readVInt(b2, startPos2);
259.                 ret = TEXT_COMPARATOR.compare(b1,
260.                         startPos1, keyLength1, b2,
261.                         startPos2, keyLength2);
262.                 if (ret != 0) {
263.                     return ret;
264.                 }
265.
266.                 //対象時間
267.                 startPos1 = startPos1 + keyLength1;
268.                 startPos2 = startPos2 + keyLength2;
269.                 keyLength1 = WritableUtils
270.                         .decodeVIntSize(b1[startPos1])
271.                         + readVInt(b1, startPos1);
272.                 keyLength2 = WritableUtils
273.                         .decodeVIntSize(b2[startPos2])
274.                         + readVInt(b2, startPos2);
275.
276.                 ret = TEXT_COMPARATOR.compare(b1,
277.                         startPos1, keyLength1, b2,
278.                         startPos2, keyLength2);
279.                 return ret;
280.             } catch (IOException e) {
281.                 throw new IllegalArgumentException(e);
282.             }
283.         }
284.     }
285. }
```

④

図 2-33 中間データの Comparator クラスの実装イメージ(2/2)

①: 中間データの Key の比較で TaxiProbeAnalysisComparator が利用されるように、Comparator のデフォルトに TaxiProbeAnalysisComparator を登録する。

②: Comparator は、WritableComparator を継承する必要がある。

③: コンストラクタでは、本クラスの比較で対象とするクラスを指定する必要があるため、TaxiProbeAnalysisKeyWritableComparable を指定している。

④: compare()メソッドでは、シリアライズされた結果のバイト配列を使って Key を比較する処理を記述する。Compare()メソッドの引数は、配列 b1、配列 b2 は比較元、比較先の Key をシリアライズしたバイナリデータが格納されている。s1、s2 は、比較元、比較先の Key をシリアライズしたバイナリデータを格納している位置が格納されている。ここでは配列 b1、配列 b2 の s1、s2 地点から各属性を比較していく。比較の際は、シリアライズしたバイナリ配列は、Key の write()メソッドでシリアライズした順、データサイズで格納されているため、順序とサイズを意識する必要がある。次に各属性の比較について解説していく。各属性の比較順と比較対象の格納位置、データサイズを表 2-7 に示す。

表 2-7 compare()メソッドの比較で必要な情報一覧

比較順	属性名	型	格納位置	データサイズ(バイト)
1	道路 ID	int	s1,s2	Integer.SIZE/8
2	進行方向	boolean	道路 ID の格納位置+道路 ID のデータサイズ	1
3	プローブ ID	string	進行方向の格納位置+進行方向のデータサイズ	WritableUtils.decodeVIntSize()+WritableUtils.readVInt()
4	対象日時	string	プローブ ID の格納位置+プローブ ID のデータサイズ	WritableUtils.decodeVIntSize()+WritableUtils.readVInt()

比較のデータの格納順は、シリアライズした順となるので、中間データの Key(TaxiProbeAnalysisKeyWritableComparable)の write()メソッドでのシリアライズ順となる。そのため、道路 ID、進行方向、プローブ ID、対象日時の順となっている。比較では、各項目のデータが可能されている位置から比較するために、バイナリデータの格納位置を各項目のデータサイズずつ移動させている。

int(道路 ID)の比較では、IntWritable の Comparator.compare()メソッドを利用し比較している。これは、シリアライズする際に IntWritable の write()メソッドでシリアライズしているため、本メソッドで比較している。

文字列(プローブ ID、対象日時)の比較では、文字列のサイズが可変長のため、int の比較のときのようにデータサイズを固定できない。そこで

WritableUtils.decodeVIntSize()+WritableUtils.readVInt()の組み合わせでデータサイズを取得している。Text の write()メソッドを利用してシリアライズしたデータに限り、この方法でデータサイズの取得ができる。

Partition の実装

Key の実装でデータ型を新たに定義する場合は、Partition を作成する必要がある。

Hadoop のデフォルトの Partititon は Key の hashCode()メソッドの値を利用し、データの振り分けを行っている。しかし、独自に作成した Key クラスでは、hashCode()メソッドの値が Key クラスの保持する値と同じであっても、インスタンスごとに別々の値となってしまう。そのため Key の持つ値が同一であっても振り分け先がばらばらとなり、Reduce 処理で Key の値ごとに集約されない問題が発生する。

TaxiProbeAnalysisKeyWritableComparable の Partition クラスを実装した結果を図 2-34 に示す。

```
11. public class TaxiProbeAnalysisPartitioner                          ①
12.         extends
13.         Partitioner<TaxiProbeAnalysisKeyWritableComparable,
14.                                 TaxiProbeAnalysisValueWritable> {
15.
16.     @Override
17.     public int getPartition(                                       ②
18.             TaxiProbeAnalysisKeyWritableComparable key,
19.             TaxiProbeAnalysisValueWritable value,
20.             int totalTask) {
21.         StringBuilder partitionerKey = new StringBuilder();
22.         partitionerKey.append(key.getRoadID());
23.         partitionerKey.append("\t");
24.         partitionerKey.append(key.isRoadDirection());
25.         partitionerKey.append("\t");
26.         partitionerKey.append(key.getProbeID().toString());
27.         partitionerKey.append("\t");
28.         partitionerKey.append(key.getTimeNumber()
29.                 .toString());
30.         return (partitionerKey.toString().hashCode()
31.                                 & Integer.MAX_VALUE) % totalTask;
32.     }
33. }
```

図 2-34 Partition クラスの実装

①: Partition は Partitioner を継承する必要がある。Partitioner では、仮型引数に対象とする Key の型を指定する必要があるので、TaxiProbeAnalysisKeyWritableComparable を指定している。

②: getPartition()メソッドでは、TaxiProbeAnalysisKeyWritableComparable の値により振り分けを行う処理を記述する。

TaxiProbeAnalysisKeyWritableComparable の各値を文字列に結合し、結合した文字列の hashCode()を使い、振り分け先の決定を行っている。

Value の実装

2.3.2 で設計した Value のデータ型を、新たに定義した場合の実装結果を図 2-35 に示す。

```
11.  public class TaxiProbeAnalysisValueWritable implements
12.          Writable {                                                    ①
13.
14.      /**
15.       * 道路始点からの距離
16.       */
17.      private double totalRoadLength;
18.
34.      /**
35.       * 道路始点からの距離を設定する。
36.       *
37.       * @param roadLength
38.       */
39.      public void setRoadLength(double roadLength) {
40.          this.roadLength = roadLength;
41.      }
42.
43.      /**
44.       * 道路始点からの距離を取得する。
45.       *
46.       * @return
47.       */
48.      public double getRoadLength() {                                   ②
49.          return roadLength;
50.      }
51.
134.     @Override
135.     public void readFields(DataInput in) throws IOException {
136.
137.         this.roadLength = in.readDouble();
138.         this.totalRoadLength = in.readDouble();
139.         this.roadType = RoadType.findRoadType(in.readInt());          ③
140.         this.speed = in.readDouble();
141.
142.     }
143.
144.     @Override
145.     public void write(DataOutput out) throws IOException {
146.
147.         out.writeDouble(this.roadLength);
148.         out.writeDouble(this.totalRoadLength);
149.         out.writeInt(this.roadType.getCode());                        ④
150.         out.writeDouble(this.speed);
151.
152.     }
153.
154. }
```

図 2-35 中間データの Value クラスの実装

①: 中間データの Value は、Writable インターフェイスを実装する必要がある。

②: 属性には、2.3.2 で設計した中間データの Value の項目(道路総距離、道路始点か

らの距離、道路種別、速度)を定義している。また、各属性の値を取得するために、各属性の get()、set()メソッドを定義している。

③: write()メソッドでは、属性をシリアライズするための処理を記述する。
各属性の型とシリアライズ方法を表 2-8 に示す。

表 2-8 write()メソッドのシリアライズ方法一覧

順番	属性名	型	シリアライズ方法
1	道路始点からの距離	double	DataOutput の writeDouble()メソッド
2	総距離	double	DataOutput の writeDouble()メソッド
3	道路種別	int	DataOutput の writeInt()メソッド
4	速度	double	DataOutput の wirteDouble()メソッド

④: read()メソッドでは、属性をデシリアライズするための処理を記述する。本項目は write()メソッドでシリアライズした順、方法に則りデシリアライズさせる必要がある。各属性の型とデシリアライズ方法を表 2-9 に示す。

表 2-9 read()メソッドのデシリアライズ方法一覧

順番	属性名	型	シリアライズ方法
1	道路始点からの距離	double	DataInput の readDouble()メソッド
2	総距離	double	DataInput の readDouble()メソッド
3	道路種別	int	DataInput の readInt()メソッド
4	速度	double	DataInput の readDouble()メソッド

2.4.2 Reduce 処理の実装

本項では、タクシープローブ解析の Reduce 処理と、Reduce 処理の入出力に関わるクラスの実装について解説する。

Reduce 処理

2.3.2 で設計した Reduce 処理を実装した結果を図 2-36 に示す。

```
17.  public class TaxiProbeAnalysisRedcue                                    ①
18.          extends
19.          Reducer<TaxiProbeAnalysisKeyWritableComparable,
20.            TaxiProbeAnalysisValueWritable, NullWritable,
21.                                              ProbeAnalysisInfo> {
22.
23.      /**
24.       * 道路区切り時の区間距離
25.       */
26.      private double roadPartitionLength;
27.      /**
28.       * ログ出力
29.       */
30.      private Log log = LogFactory
31.              .getLog(TaxiProbeRoadInfoReduce.class);
32.
33.      /**
34.       *設定値取得KEY
35.       */
36.      public final static String KEY_ROAD_PARTITION_LENGTH =
37.                                          "road-partition-length";
38.
39.      @Override
40.      protected void setup(Context context)                              ②
41.              throws IOException, InterruptedException {
42.          try {
43.              Configuration conf = context.getConfiguration();
44.              // 道路を区切る際の区間距離の取得
45.              String strRoadPartitionLength=
46.                      TaxiProbeAnalysisRedcue.KEY_ROAD_PARTITION_LENGTH
47.              roadPartitionLength =
48.               Double.parseDouble(strRoadPartitionLength);
49.
50.          } catch (Exception e) {
51.              log.error(e);
52.          }
53.      }
54.
55.      @Override
56.      protected void reduce(                                             ③
57.              TaxiProbeAnalysisKeyWritableComparable key,
58.              Iterable<TaxiProbeAnalysisValueWritable> values,
59.              Context context) throws IOException,
60.              InterruptedException {
61.          TaxiRoadInfoReduceLogic logic = new TaxiRoadInfoReduceLogic(
62.                  key, roadPartitionLength);
63.          try {
64.              for (TaxiProbeAnalysisValueWritable value : values) {
65.                  logic.addValue(value);
66.              }
67.              ProbeAnalysisInfo taxiProbeAnalysis = logic
68.                      .getTemporaryCongestionInfo();
69.              context.write(NullWritable.get(),
70.                      taxiProbeAnalysis);
71.          } catch (Exception e) {
72.              log.error(e);
73.          }
74.      }
75.  }
```

図 2-36 Reduce クラスの実装イメージ

①: Reduce 処理は、Reducer を継承する必要がある。Reducer の仮型引数は中間データの Key の型、Value の型と、出力データの Key の型、Value の型の順に指定する。中間データの Key の型、Value の型は、中間データで作成した TaxiProbeAnalysisKeyWritableComparable と TaxiProbeAnalysisValueWritable を指定している。出力データの Key の型は NullWritable を指定している。これは出力フォーマット制御クラスの TextOutputFormat の仕様で、出力データの形式を Key と Value の対とする必要がない場合に指定することになっているためである。出力データの Value の型は出力データで実装する ProbeAnalysisInfo を指定している。

②: setup()メソッドでは、Reduce 処理で必要な初期化処理を記述する。Reduce 処理で行う「速度の計算」処理では、2.3.1.1 の記述のとおり、設定に応じて道路を区切る際の区間の距離を変更する必要がある。ここでは変更のための値を次の順で取得する。

A) Context の getConfiguration()メソッドより Configuration を取得する。
B) Configuration の get()メソッドより、キーを指定し、該当する値(区間の距離)を取得する。

なお、Configuration の情報は後述する MapReduce ジョブでキーと値のセットを設定している。

③: reduce()メソッドでは、2.3.2 の設計で Reduce 処理の対象とした処理を記述する。reduce()メソッドの引数の Key・Value は Map 処理の map()メソッドの Context.write()メソッドで出力した中間データが Key 単位に集約され、入力される。ここでは、集約されたデータを使い次の処理を行う。

A) reduce()メソッドの引数で指定された Key・Value の全レコードを取得し、2.3.2 で Reduce 処理の対象とした「速度の計算」処理を行う。
B) 計算の結果を出力データの Value(ProbeAnalysisInfo)に格納する。
C) Key・Value を Context クラスの write()メソッドに指定し出力する。このとき Key は NullWritable のインスタンスを指定する。これは出力フォーマット制御クラスに指定した TextOutputFormat の仕様で、出力の形式が Key と Value の対になっている必要がない場合の Key には NullWritable を指定することになっているためである。

出力データの実装

ここでは、タクシープローブ解析の出力データの Key・Value の実装について解説する。タクシープローブ解析で指定する出力フォーマット制御クラスの TextOutputFormat では、出力の形式で Key が不要な場合に NullWritable を指定できるようになっている。Key については、2.3.2 で出力データには不要としたので、ここでは本仕様に従い Key に NullWritable を指定する。

Value については、2.3.2 の設計で出力するデータ項目を決めているので、これに従い実装する。出力データの Value の実装結果を図 2-37 に示す。

```
 7. public class ProbeAnalysisInfo {
 8.
 9.     /**
10.      * 道路ID
11.      */
12.     private int roadID;
13.
39.     /**
40.      * 道路IDを設定する。
41.      *
42.      * @param roadID
43.      */
44.     public void setRoadID(int roadID) {
45.         this.roadID = roadID;
46.     }
47.
48.     /**
49.      * 道路IDを取得する。
50.      *
51.      * @return
52.      */
53.     public int getRoadID() {
54.         return roadID;
55.     }
146.    @Override
147.    public String toString() {
148.        StringBuilder output = new StringBuilder();
149.        SimpleDateFormat format =
150.            new SimpleDateFormat("yyyyMMddHHmmss");
151.        output.append(format.format(this.probeDate
152.                .getTime()));
153.        output.append("\t");
154.        output.append(roadID);
155.        output.append("\t");
156.        if (isDirectionReverse) {
157.            output.append("0");
158.        } else {
159.            output.append("1");
160.        }
161.        output.append("\t");
162.        output.append(roadType.getCode());
163.        output.append("\t");
164.        output.append(probeID.toString());
165.        output.append("\t");
166.        output.append(intervalSpeed.toString());
167.        return output.toString();
168.    }
169. }
```

① ②

図 2-37 出力データの Value クラスの実装イメージ

①: 属性には、2.3.2 で設計した出力データの Value の項目(道路 ID、進行方向、対象日時、プローブ ID、速度)を定義している。また、各属性の値の設定や取得をするために get()、set()メソッドを定義している。

②: toString()メソッドでは、出力データの形式に整えた文字列を作成する。出力フォーマット制御クラスの TextOutputFormat では、出力の際に Value に指定されたクラスの toString()メソッドを呼び出し、出力するテキストを取得する仕様となっている。そのため、ここでは各属性を文字列に変換してタブ区切りで結合し、出力ファイルのフォーマットに沿った文字列を作成している。

2.4.3 MapReduce ジョブの実装

2.3.2 で設計した MapReduce ジョブの実装結果を図 2-38 に示す。

```
21.  public class TaxiProbeAnalysisJob extends Configured                    ①
22.          implements Tool {
23.
24.      /**
25.       * 入力パス
26.       */
27.      private String inputPath;
28.
54.      /**
55.       * 入力パスを指定する。
56.       *
57.       * @param inputPath
58.       */
59.      public void setInputPath(String inputPath) {                        ②
60.          this.inputPath = inputPath;
61.      }
109.     @Override
110.     public int run(String[] arg0) throws Exception {
111.
112.         Configuration conf = this.getConf();
113.         conf.set(TaxiProbeAnalysisMapper.KEY_ROAD_SEARCH_CLASS,
114.                                         this.roadSearchClassName);
115.         conf.set(TaxiProbeAnalysisRedcue.KEY_ROAD_PARTITION_LENGTH,
116.                         String.valueOf(this.roadPartitionLength));
117.
118.         Job job = new Job(conf, "タクシープローブ解析");
119.         job.setJarByClass(TaxiProbeAnalysisJob.class);
120.         TextInputFormat.setMaxInputSplitSize(job,
121.                 this.maxInputSplit);
122.         TextInputFormat.setInputPaths(job, new Path(
123.                 this.inputPath));
124.         job.setMapperClass(TaxiProbeAnalysisMapper.class);
125.         job.setMapOutputKeyClass(                                       ③
126.                     TaxiProbeAnalysisKeyWritableComparable.class);
127.         job.setMapOutputValueClass(
128.                             TaxiProbeAnalysisValueWritable.class);
129.         job.setPartitionerClass(
130.                              TaxiProbeAnalysisPartitioner.class);
131.         job.setReducerClass(TaxiProbeAnalysisRedcue.class);
132.         job.setOutputKeyClass(NullWritable.class);
133.         job.setOutputValueClass(ProbeAnalysisInfo.class);
134.         TextOutputFormat.setOutputPath(job, new Path(
135.                 this.outputPath));
136.         job.setNumReduceTasks(this.numReduceTask);
137.
138.         job.waitForCompletion(true);
139.         return 0;
140.     }
141. }
```

図 2-38 Reduce クラスの実装イメージ

①: MapReduce ジョブは Configured クラスを継承し、Tool インターフェイスを実装する必要がある。

②: 属性には、タクシープローブ解析の処理で必要な各種パラメータを設定するための属性している。また、各属性に他クラスから値を設定できるように、set()メソッドを定義している。Hadoop では、実行時に設定ファイルを指定することで、

自動的に設定ファイルの内容を Configuration に格納することができるが、本アプリケーションでは、set()メソッドを作成し、本 MapReduce ジョブを呼び出すクラスで各種パラメータを設定させる方法を取った。

渋滞解析アプリケーションは、複数の MapReduce ジョブで構成されている。そのため、設定ファイルを直接 Configuration に格納する機能を使うと設定値のチェックができず、設定ミスにより複数の MapReduce ジョブの一部だけが失敗する可能性が出てくる。こうした問題を防ぐために、本クラスを呼び出すクラスで値をチェックしてから、set()メソッドで値を設定する方針で実装している。

③: run()メソッドでは、MapReduce ジョブで必要な各種項目を設定する。各項目の設定を表 2-10 に示す。

表 2-10 MapReduce ジョブの設定項目一覧

No.	設定項目	対象クラス	対象メソッド	設定内容
1	Job 名	Job	コンストラクタ	“タクシープローブ解析”を指定
2	InputSplit サイズ	TextInputFormat	setMaxInputSplitSize	入力データ分割サイズを指定
3	入力パス	TextInputFormat	setInputPaths	入力ファイルのパスを指定
4	Map 処理	Job	setMapperClass	TaxiProbeAnalysisMapper
5	中間データの Key	Job	setMapperOutputKeyClass	TaxiProbeAnalysisKeyWritableComparator
6	中間データの Value	Job	setMapperOutputValueClass	TaxiProbeAnalysisValueWritable
7	Shuffle 処理	Job	setPartitionerClass	TaxiProbeAnalysisPartitioner
8	Reduce 処理	Job	setReducerClass	TaxiProbeAnalysisReducer
9	出力データの Key	Job	setOutputKeyClass	NullWritable
10	出力データの Value	Job	setOutputValueClass	ProbeAnalysisInfo
11	出力パス	TextOutputFormat	setOutputPath	出力ファイルのパスを指定
12	Reduce タスク数	Job	setNumReduceTasks	Reduce 処理のタスク数を指定

Map 処理、Reduce 処理で必要となる設定値は、Configuration の set()メソッドを使って、Key と Value の組み合わせで設定している。Map 処理、Reduce 処理ではここで指定する Key を使い値の取得をするので、Map 処理、Reduce 処理で取得する際の Key と合わせる必要がある。本プログラムでは、Map クラス、Reduce クラスに Key となる文字列を定義し、それぞれで必要となる Key を間違いなく設定できるよう工夫している。

MapReduce ジョブの実行は Job クラスの waitForCompletion()メソッドで行っている。渋滞解析アプリケーションでは本 MapReduce ジョブの実行結果をうけて次のジョブを実行する処理があるため、waitForCompletion()メソッドの引

数に true を設定し、MapReduce ジョブの完了を待たせている

2.5 まとめ

本章では、プローブデータによる渋滞解析アプリケーションを事例とし、Hadoop を使った MapReduce の設計・実装方法を中心に解説を行ってきた。

Hadoop を用いた開発で重要となるのは Map 処理、Reduce 処理で行う処理と、その入出力データの決定である。

本章では、これらを決定するプロセスを MapReduce 設計と定義づけ、その項目の決定方法を検討した。本方法は、一般的な設計プロセスの成果物である処理フローを基としたため、開発プロセスへの取り込みが容易に行え、短期間で Map 処理、Reduce 処理で行う処理と、その入出力データの決定が行えた。

また実装については、Hadoop で定められているコンポーネントに MapReduce 設計で決定した項目を割り当て、プローブデータによる渋滞解析処理の実装を行った。Hadoop が提供するコンポーネントに対する理解があれば、容易に分散処理を実装できることを示した。

ここで事例として取り上げたような「大容量データを使った分散処理アプリケーション」の開発を行う場合、規模に応じた分散処理のための仕組みと、大容量データを扱うための仕組みの開発が必要となる。従来の方法で大規模な分散処理を実装するためには、高度な技術が必要となるほか、その方法によってはハードウェアへの依存が必要となる場合もある。しかし、Hadoop では分散処理を実装するためのコンポーネント群が提供されており、開発者は、分散処理アプリケーションの開発において、MapReduce 設計と Hadoop で定められたコンポーネントによる処理の実装に注力することができる。結果として、システムの核となる業務処理の開発が主となることから、MapReduce アルゴリズムに対するある程度の理解があれば、高度な分散処理技術を持たない技術者でも、クラウド環境への適用が可能なスケーラブルなアプリケーションを開発することができる。

3　性能や精度を確保する技術の開発

本章では、2 章で開発した Hadoop 基盤上で動作する渋滞解析アプリケーションを利用して、アプリケーションの処理性能や精度に関する技術について報告する。本章では、以下の内容について報告する。

- Hadoop 基盤設定によるチューニング
  アプリケーション実行基盤として、Hadoop 基盤の設定をチューニングする。
- MapReduce ジョブでの処理分割検討
  アプリケーションを分散処理させるための分割方法を検討する。
- MapReduce ジョブ実行時間の見積もり
  大量のデータを実行する場合の処理時間を事前に把握するため、少量データ実行により大量データでの MapReduce ジョブ実行時間を見積もる。
- アプリケーション評価
  アプリケーションで使用する情報や処理量などの条件を変更することで、性能や精度にどのような影響を与えるか把握する。また、アプリケーションがスケーラビリティを実現していることを確認する。

3.1 Hadoop 基盤としてのチューニングポイント

本節では、2 章で開発した渋滞解析アプリケーションを実行する Hadoop 基盤のチューニングポイントについて述べる。

3.1.1　Hadoop 基盤環境構成

MapReduce ジョブを処理ノード数が異なる 3 つの環境で実行する。表 3-1 に 3 つの環境構成を示す。また、処理ノードのハードウェアスペックについて表 3-2 に示す。表 3-1、表 3-2 に示すように、24 台分散処理環境と 48 台分散処理環境は、同一のハードウェアスペックで Hadoop 基盤を構成する。混在分散処理環境は、ハードウェアスペックが異なる。

表 3-1 Hadoop 基盤環境構成

No.	環境名	ノード数	備考
1	24 台分散処理環境	24	
2	48 台分散処理環境	48	
3	混在分散処理環境	93	ハードウェアスペックが混在するノードで構成

表 3-2 処理ノードのハードウェアスペックと環境構成関係

No.	名称	CPU	メモリ	ディスク	台数	環境構成での利用
1	S1	Core 2 Duo T9400 2.53GHz 2 コア	2GB	SATA 250GB×2	48	24 台, 48 台,混在分散処理環境
2	S2	Xeon E5504 2GHz 4 コア	6GB	SAS 300GB×2	17	混在分散処理環境
3	S3	Xeon 5148 2.33GHz 2 コア	2GB	SAS 72GB×2	16	混在分散処理環境
4	S4	Xeon X5460 3.16GHz 4 コア	6GB	SAS 146GB×2	8	混在分散処理環境
5	S5	Xeon E5345 2.33GHz 4 コア×2	8GB	SAS 146GB×2	4	混在分散処理環境

3.1.2 MapReduce 処理に関する Hadoop 基盤の環境制約

本項では、MapReduce を実行するときの Hadoop 基盤の環境制約について述べる。

Hadoop 上での MapReduce は、図 3-1 に示す 2 つのステージから構成される。

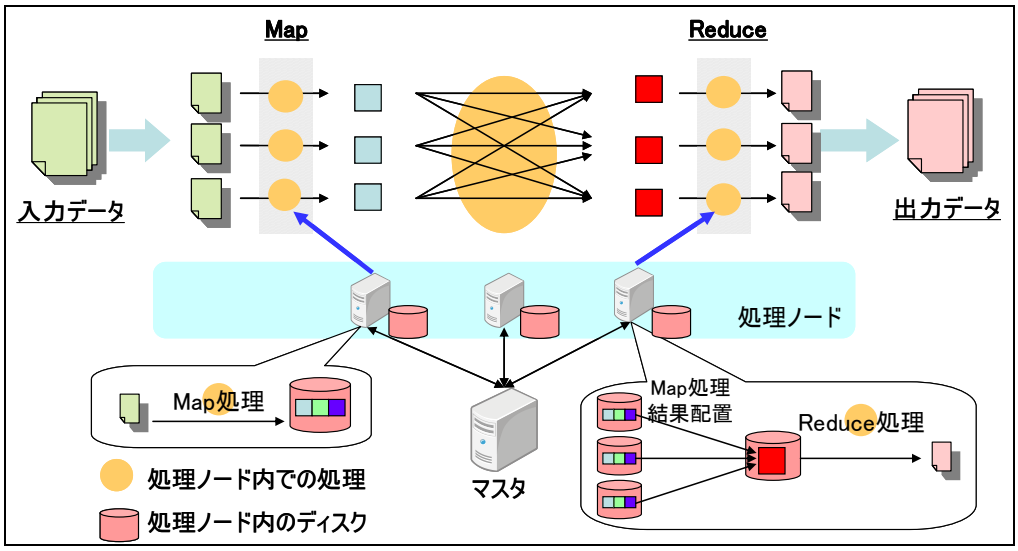


図 3-1 Hadoop 上での MapReduce 処理の流れ

(1) Map 処理 ： 入力データを各自で定義した Map 処理に適用する。
(2) Reduce 処理 ：Key 単位で配置されたデータに対して Reduce 処理を適用する。

処理ノードのハードウェアスペックの違いが MapReduce 処理に与える影響について検討する。Map 処理、Reduce 処理のスロット数や Hadoop で扱うディスク台数は 9 章で説明する設定方針で決定できる。しかし、ハードウェアスペックによっては制約により設定を見直さなければならないこともある。特に以下の 2 点を制約として考える。

- 実メモリ容量制約による影響

  Hadoop の MapReduce ジョブは、ジョブ単位で、処理に関する JavaVM ヒープメモリサイズを設定できる。ヒープメモリサイズ設定は、処理ノード全体に適用されるため、ハードウェアスペックが混在する各サーバの実メモリ容量を考慮していない。そのため、大量のヒープメモリが割り当てられた場合、処理ノードでスワップへのアクセスが発生して処理時間が大幅に伸びる可能性がある。

- ディスク容量制約による影響

  MapReduce ジョブ実行時に、ディスク容量不足により処理がエラーとなり失敗する可能性がある。特にディスク容量が少ない場合は、MapReduce ジョブ実行時に処理が失敗となりやすい。そのため、ディスク容量が少ない処理ノードは、RAID-0 のように複数のディスクを 1 台のディスクにまとめるように設定する。

3.1.3 環境制約を考慮した Hadoop 基盤設定

本項では、環境制約を考慮した Hadoop 基盤の設定方法について述べる。

Map 処理、Reduce 処理の同時処理数

Map 処理、Reduce 処理の同時処理数(スロット数)を調整することで処理時間の短縮を図る。スロット数は、処理ノードの CPU コア数に着目する。

9 章の Map 処理、Reduce 処理のスロット数設定方針は以下の通りである。

- Map スロット数は、コア数×1～1.5 程度で良い。
- Reduce スロット数は、コア数×1～コア数＋1 程度で良い。

Map 処理、Reduce 処理で利用するディスク台数

Hadoop は、HDFS、MapReduce ともに処理ノードのディスクアクセスが頻発する。そのため、データ読み書きのためのディスクを分散させることで、ディスクボトルネ

ックを軽減する。
9 章のディスク台数に設定方針は、以下の通りである。

・ HDFS で使用するディスク
　処理ノードが複数のディスクを持っている場合、ディスクアクセスの負荷分散や容量を拡張できるので、全てのディスクを HDFS として利用できるように設定する。
・ MapReduce で使用するディスク
　HDFS の時と同様に、MapReduce 実行によるディスクアクセスを負荷分散させるために、全てのディスクを MapReduce で利用できるように設定する。

ハードウェアスペック混在による Hadoop 基盤設定

ここまでで説明した設定方針と 3.1.2 で述べた環境制約を考慮して、Hadoop 基盤を設定する。Hadoop 基盤を設定するときに、処理ノードの環境制約に対する影響を考慮しなければならない。環境制約を考慮した設定について述べる。

・ 実メモリ容量制約による影響を考慮した設定
　Map スロット数や Reduce スロット数の設定や JavaVM ヒープメモリの割り当てサイズによっては、処理ノードの実メモリが不足するケースがある。
　実メモリ容量不足となる場合は、以下の 2 パターンで対処する。

　➢ JavaVM ヒープメモリサイズの変更
　➢ Map スロット数、Reduce スロット数の変更

　JavaVM ヒープメモリサイズ変更は、アプリケーション単位で設定可能である。Map スロット数、Reduce スロット数は、処理ノードの TaskTracker 起動タイミングで変更する。
　どの MapReduce ジョブを実行させてもスワップへのアクセスを抑えるため、Map スロット数と Reduce スロット数を変更する。また、処理に割り当てる JavaVM ヒープメモリサイズを設定する。
　9 章 に記載されている見積もり式を表 3-2 で示した環境に適用した場合、コア数の考え方に応じて 200MB～450MB を JavaVM のヒープメモリとして割り当てられる。

・ ディスク容量制約による影響を考慮した設定

Hadoop 基盤は、表 3-2 に示すとおり全て 2 台のディスクから構成される。本環境では、OS やソフトウェアに関する領域は 2 台のうち 1 台のディスクに格納する。そのため、ディスク容量が小さいサーバでは、1 台のディスクで利用できる Hadoop 領域が非常に小さくなる。本環境では、OS やソフトウェアの領域として 60GB を 1 台のディスクに割り当てている。このため、表 3-2 の S3 サーバの片方のディスクは数 GB しか Hadoop に割り当てられない。そして、MapReduce ジョブでディスク容量不足により Map 処理、Reduce 処理が失敗する可能性がある。よって、S3 サーバのディスクは RAID-0 構成にして、2 台のディスクを 1 つに束ねて、Hadoop として利用するディスク容量を多くする。

以上より、Hadoop 基盤としての設定を表 3-3 に示す。

表 3-3 Hadoop 基盤設定

No.	名称	Map スロット数	Reduce スロット数	ディスク台数
1	S1	2 スロット	2 スロット	2 台利用
2	S2	4 スロット	4 スロット	2 台利用
3	S3	2 スロット	2 スロット	RAID-0 として 2 台を 1 台として利用
4	S4	4 スロット	4 スロット	2 台利用
5	S5	8 スロット	8 スロット	2 台利用

スロットに割り当てる JavaVM ヒープメモリを 200MB～450MB の範囲で設定する。

3.2 MapReduce 処理分割検討

本節では、MapReduce ジョブを実行するときの Map 処理、Reduce 処理の分割方針について述べる。

3.2.1 アプリケーションボトルネック調査

2 章で開発した渋滞解析アプリケーションの処理特性を把握する。

まず、アプリケーション内の MapReduce ジョブでどのような処理を実行しているか机上で確認したものを表 3-4 に示す。

表 3-4　渋滞解析アプリケーション内の MapReduce ジョブ

No.	MapReduce ジョブ	処理概要	Map 処理	Reduce 処理
1	タクシープローブ解析	連続したタクシープローブ情報からプローブに紐づく速度を求める	入力データのフィルタリング、道路の選択、進行方向の決定、対象日時の計算	速度の計算
2	携帯電話プローブ解析	連続した携帯電話プローブ情報からプローブに紐づく速度を求める	入力データのフィルタリング、対象日時の計算	入力データの間引き、道路の選択、進行方向の決定、速度の計算
3	解析結果集計	No.1,No.2 で求めた速度から渋滞情報や渋滞統計情報を求める	集計キーの作成	速度情報集計

次に、MapReduce ジョブ内のどの部分で処理時間が長いか確認するため、分散処理しない設定で渋滞解析アプリケーションを実行する。以下の条件で処理特性を確認する。

- タクシープローブ、携帯電話プローブともに 100,000 件で実行
- 各 MapReduce ジョブの Map 処理数、Reduce 処理数を 1 に設定

処理特性把握のための処理時間測定結果を表 3-5 に示す。

表 3-5 処理特性把握のための測定結果

No.	タクシープローブ解析		携帯電話プローブ解析		解析結果集計	
	Map	Reduce	Map	Reduce	Map	Reduce
1	670 秒	25 秒	9 秒	141 秒	9 秒	19 秒

この測定結果より確認できたことは以下の通りである。

- タクシープローブ解析 MapReduce ジョブは、Map 処理に処理時間を要する
- 携帯電話プローブ解析 MapReduce ジョブは、Reduce 処理に処理時間を要する

以上より、Map 処理、Reduce 処理の分割を以下のように検討する。

- タクシープローブ解析 MapReduce ジョブは Map 処理時間が長い。そのため、Map 処理を分散させることで、処理時間を短縮する
- 携帯電話プローブ解析 MapReduce ジョブは、Reduce 処理時間が長い。そのため、Reduce 処理を分散させることで、処理時間を短縮する

次項で、各処理の分割方針を述べる。

3.2.2 Map 処理分割のポイント

Map 処理の分割では、特にタクシープローブ解析ジョブのチューニングについて説明する。

タクシープローブ解析ジョブでは、Map 処理に時間を要する。そのため、Map 処理を分割することで処理時間を短縮させる。

Map 処理の分割は、9 章で説明する Map 処理分割方針に沿って検討する。机上での調査と処理特性把握のための測定結果より CPU ボトルネックであることを確認した。そこで、CPU ボトルネックに着目して Map 処理を分割する。9 章より、Map スロット数の 30 倍以内の範囲で Map 処理を分割する。携帯電話プローブ解析ジョブは、表 3-5 より処理時間が短いため、HDFS で管理するブロックサイズに沿って Map 処理を分割する。

解析結果集計ジョブは、タクシープローブ解析結果と携帯電話プローブ解析結果が Map 処理の入力となるため、Map 処理分割として特別な設定はしない。

3.2.3 Reduce 処理分割のポイント

Reduce 処理分割では、特に携帯電話プローブデータ解析の MapReduce ジョブについて説明する。

携帯電話プローブデータ解析の MapReduce ジョブは、Reduce 処理でプローブデータを主に処理する。このため、9 章で示した Reduce 処理の分割方針に従い、1 つの Reduce 処理に割り当てられる JavaVM ヒープメモリサイズを超えないように、Reduce 処理の分割数を調整する。

タクシープローブ解析 MapReduce ジョブ、解析結果集計 MapReduce ジョブも同様に Reduce 処理に割り当てられる JavaVM ヒープメモリサイズを超えないように Reduce 処理の分割数を調整する。

3.3 MapReduce 処理時間見積もり方法

本節は、MapReduce を実行するアプリケーションの処理時間を見積もる方法について説明する。処理時間を見積もる目的は、数十 GB を超える大量のデータをいきな

り実行しても、処理がいつ完了するか判断できないためである。そのため、事前に少量のデータを実行して基礎情報を確定し、大規模データを実行する場合の処理時間を見積もる。

3.3.1 少量データ実行により確定するパラメータ

まず、少量(数 10MB～数 GB 程度)の入力データに対して、MapReduce ジョブを実行する。その結果、以下のパラメータを確定できる。

- Map 処理時間 ：少量データのジョブより Map 処理時間を設定する。
- Reduce 処理時間 ：少量データのジョブより Reduce 処理時間を設定する。
- Map 入力データ量 ：少量データを利用する。
- Reduce 入力データ量 ：少量データのジョブより Reduce 入力データ量を設定する。
- Map 分割数 ：Map 処理の分割数を設定する。
- Reduce 分割数 ：Reduce 処理の分割数を設定する。
- Map スロット数 ：Map 処理の多重度を設定する。
- Reduce スロット数 ：Reduce 処理の多重度を設定する。

3.3.2 MapReduce 処理時間見積もり式

前項までで確定したパラメータを元に、MapReduce 処理時間見積もり式を検討する。MapReduce の特性で、スケーラビリティを踏まえて処理時間を見積ることができる。少量データ実行によるパラメータより、処理時間を算出する。

大量データ実行前に、以下の必要な情報が分かる。

- 大量データの Map 入力データ量
- Map 分割数 ： 大量データとともに Map 分割数を設定
- Reduce 分割数 ： 大量データでの Reduce 分割数を設定
- Map スロット数 ： 大量データ実行環境の Map スロット数
- Reduce スロット数 ： 大量データ実行環境の Reduce スロット数

Map 処理時間見積もり式

Map 処理時間は、Map への入力データ量を元に決定する。

[少量]1Map あたりの処理時間 ＝ [少量]Map 処理時間 ÷ ([少量]Map 分割数÷[少量]Map スロット数)

[大規模] Map 処理時間概算 ＝ [少量]1Map あたりの処理時間 × [大規模]Map 分

割数 ÷ [大規模] Map スロット数

[少量]と[大規模]1Map 処理のデータ量変化率 =([少量]Map 入力データ量÷[少量]Map 分割数) ÷ ([大規模]Map 入力データ量÷[大規模]Map 分割数)

Map 処理時間 =[大規模] Map 処理時間概算 ÷ 1Map 処理のデータ量変化率

Reduce 処理時間見積もり式

Reduce 処理では、最初に少量データでの処理を元に以下の情報を決定する。

[大規模]Reduce 入力データ量 = [大規模]入力データ量÷([少量]入力データ量÷[少量]Reduce 入力データ量)

上で算出した大規模での Reduce 入力データ量を利用して、以下のように算出する。

[少量]1Reduce あたりの処理時間 = [少量]Reduce 処理時間 ÷ ([少量]Reduce 分割数÷[少量]Reduce スロット数)

[少量]1Reduce あたりの処理量= [少量]Reduce データ入力量 ÷ [少量]Reduce 分割数

[大規模]1Reduce あたりの処理量 = [大規模]Reduce 入力データ量 ÷ [大規模]Reduce 分割数

[大規模]1Reduce あたりの処理時間 = [少量]1Reduce あたりの処理時間 × [大規模]1Reduce あたりの処理量 ÷ [少量]1Reduce あたりの処理量

Reduce 処理時間 = [大規模]1Reduce あたりの処理時間 × [大規模]Reduce 分割数÷[大規模]Reduce スロット数

MapReduce 処理時間

Hadoop の MapReduce ジョブで Reduce 処理は、Map 処理の途中から開始される。Reduce 処理開始時の Map 処理完了率(デフォルト 5%)を踏まえて、MapReduce 処理時間を見積もる。

MapReduce 処理時間 = Map 処理時間 × Reduce 処理開始時の Map 処理完了率

+ Reduce 処理時間

3.3.3 MapReduce 処理時間の評価

本項では、前項までで定義した MapReduce 処理時間見積もり式を渋滞統計生成処理の携帯電話プローブ解析 MapReduce ジョブに適用して、見積もり式の妥当性を評価する。

少量データとして 1 日分の携帯電話プローブ量で MapReduce ジョブを実行した時のパラメータを使い、1 ヶ月分の携帯電話プローブ情報を処理する場合の MapReduce 処理時間を見積もる。

少量データでの実行により確定するパラメータ

1 日分(約 5GB)のデータを処理することで、少量データ試行によるパラメータを確認する。パラメータは以下の通りである。なお、少量データ実行は、24 台分散処理環境を使用する。

- Map 処理時間 ：74 秒
- Reduce 処理時間 ：896 秒
- Map 入力データ量（1 日分のデータ量） ：5.82×10^9 Byte
- Reduce 入力データ量 ：7.27×10^9 Byte
- Map 分割数 ：87
- Reduce 分割数 ：260
- Map スロット数(24 台分散処理環境)：48
- Reduce スロット数(24 台分散処理環境)：48

大規模データ実行に関するパラメータ

- Map 入力データ量(1 ヶ月分のデータ量)：1.86×10^11 Byte
- Map 分割数 ：2782
- Reduce 分割数 ：1300
- Map スロット数(混在分散処理環境)：260
- Reduce スロット数(混在分散処理環境)：260

Map 処理時間見積もり

少量データ試行によるパラメータより、Map 処理時間を算出する。

[少量]1Map あたりの処理時間 = [少量]Map 処理時間 ÷ ([少量]Map 分割数÷[少量]Map スロット数)

= 74 ÷ (87 ÷ 48)
= 40.828 (秒)

[大規模] Map 処理時間概算 = [少量]1Map あたりの処理時間 × [大規模]Map 分割数 ÷ [大規模] Map スロット数
= 40.828 × 2782 ÷ 260
= 436.86 (秒)

[少量]と[大規模]1Map 処理のデータ量変化率 = ([少量]Map 入力データ量÷[少量]Map 分割数) ÷ ([大規模]Map 入力データ量÷[大規模]Map 分割数)
= (5.82×10^9 ÷ 87) ÷ (1.86×10^11 ÷ 2782)
= 1.000026

Map 処理時間 =[大規模]環境での Map 処理時間概算 ÷ 1Map 処理のデータ量変化率
= 436.86 ÷ 1.000026
= 436.84 (秒)

Reduce 処理時間見積もり

少量データ試行によるパラメータから、まず大規模環境での Reduce 入力量を決定する。

[大規模]Reduce 入力データ量 = [大規模]入力データ量 ÷ ([少量]入力データ量 ÷ [少量]Reduce 入力データ量)
= 1.86×10^11 ÷ (5.82×10^9 ÷ 7.27×10^9)
= 2.32 × 10^11 (Byte)

次に、大規模環境での Reduce 入力データ量を利用して、Reduce 処理時間を見積もる。

[少量]1Reduce あたりの処理時間 = [少量]Reduce 処理時間 ÷ ([少量]Reduce 分割数÷[少量]Reduce スロット数)
= 896 ÷ (260 ÷ 48)
= 165.41 (秒)

[少量]1Reduce あたりの処理量= [少量]Reduce データ入力量 ÷ [少量]Reduce 分割数

= 7.27×10^11 ÷ 260
= 2.79×10^7 (Byte)

[大規模]1Reduce あたりの処理量 = [大規模]Reduce 入力データ量 ÷ [大規模]Reduce 分割数
= 2.32×10^11÷ 1300
= 1.79×10^8 (Byte)

[大規模]1Reduce あたりの処理時間 = [少量]1Reduce あたりの処理時間 × [大規模]1Reduce あたりの処理量 ÷ [少量]1Reduce あたりの処理量
= 165.41 × 1.79×10^8 ÷ 2.79×10^7
= 1057.87(秒)

Reduce 処理時間 = [大規模]1Reduce あたりの処理時間 × [大規模]Reduce 分割数 ÷[大規模]Reduce スロット数
= 1057.87 × 1300 ÷ 260
= 5289.35 (秒)

MapReduce 処理見積もり時間

Reduce 処理は、Map 処理進捗率 5%より開始される。

MapReduce 処理時間 ＝ Map 処理時間 × Reduce 処理開始時の Map 処理完了率 + Reduce 処理時間
＝ 436.84×0.05 + 5289.35
≒ 5311.19 (秒)

実際の測定結果

実際に 1 ヶ月分のデータを利用して渋滞統計情報生成処理を実行した。実行した結果は、以下の通りになった。

- Map 処理時間 ：492 秒（見積もり時間と約 13%の誤差)
- Reduce 処理時間 ：5812 秒（見積もり時間と約 10%の誤差)
- MapReduce 処理時間 ：5841 秒（見積もり時間と約 10%の誤差)

以上より、MapReduce 処理時間の見積もり式は、実測値より 10%程度の誤差となった。

3.4 アプリケーション評価

本節では、2 章で開発した渋滞解析アプリケーションについて、表 3-1 に示す環境による影響について評価する。特に処理時間に着目して評価する。

渋滞統計情報生成処理で、以下の項目を評価する。

- 処理データ量を一定として、処理ノード数を変化させた場合の影響
- 処理ノード数を一定として、処理データ量を変化させた場合の影響
- 渋滞解析アプリケーションの処理精度を制御する設定内容を変更した場合の影響

なお、渋滞解析アプリケーションで処理精度を制御するために設定できる内容は、以下の通りである。

- 道路種別指定

  プローブデータ解析ジョブで、全国の高速道路、国道、都道府県道などからなる道路データを使用する。道路データは、高速道路と国道などを併せた「主要道路」、主要道路に都道府県道と地方道などを併せた「一般道路」の 2 種類の道路種別のどちらかを指定する。道路数は、「主要道路」が約 105 万、「一般道路」が約 388 万である。

  「一般道路」を指定することで、「主要道路」よりも高精度な渋滞情報解析を実現できる。ただし、道路数が多いためプローブデータから道路を特定する処理に時間を要することになる。

- 道路区間距離指定

  渋滞情報を生成する距離単位となる「道路区間距離」を指定する。道路区間距離は「標準（数百メートル）」と「詳細（数十メートル）」から選択する。

  「詳細」では、より高精度な渋滞情報を生成することが出来る。

以上に示すアプリケーションの設定変更とその影響に対して、特に処理時間に影響を与える「道路種別指定」を変更した場合の影響について、渋滞統計情報生成処理により評価する。

3.4.1 処理ノード数の変化による性能への影響について

Hadoop 基盤の処理ノード数を表 3-6 のように 3 段階に変化させた場合の影響を調査する。以下に測定条件を示す。

表 3-6 Hadoop 基盤環境構成

No.	環境名	ノード数	備考
1	24 台分散処理環境	24	
2	48 台分散処理環境	48	
3	混在分散処理環境	93	ハードウェアスペックが混在するノードで構成

- タクシープローブデータは 1 ヶ月分のデータを使用
- 携帯電話プローブデータは 9 日分のデータを使用
- 道路種別は、「一般道路」のデータを使用
- 処理ノード数は 93 台。Map スロット数、Reduce スロット数ともに 260
- タクシープローブデータは、1Map 処理あたり 640KB で設定
- 携帯電話プローブ Reduce 分割数は、1300 で設定
- 各処理に使用する JavaVM ヒープメモリサイズは、最大 400MB で設定

タクシープローブデータ解析ジョブ、携帯電話プローブデータ解析ジョブでの処理時間の変化について、図 3-2 に示す。

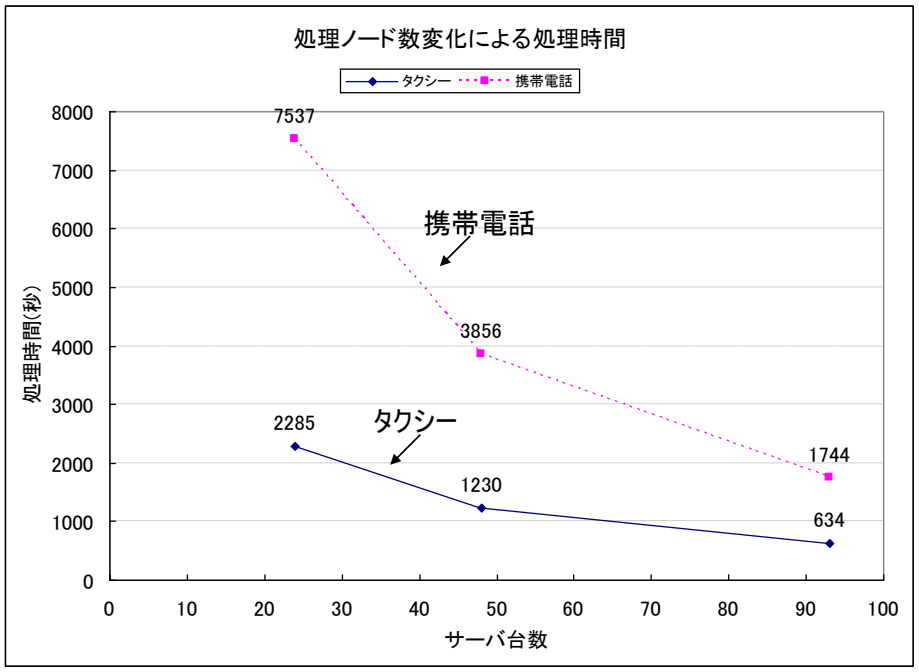


図 3-2 処理ノード数による処理時間

処理ノード数を変化させた結果より、サーバ台数の増減率と処理時間の増減率は、

ほぼ一致していることを確認できた。以上から、開発したアプリケーションは、処理ノード数によるスケーラビリティを達成していると言える。

3.4.2 処理データ量変化による性能への影響について

処理データ量を変化させることで、スケーラビリティを実現できているかどうか確認する。本測定では、携帯電話のプローブデータの処理量を変化させる。

- 携帯電話プローブデータは、9 日分、30 日分(1 ヶ月)、90 日分(3 ヶ月)、365 日分(1 年)を使用
- 道路種別は、「一般道路」のデータを使用
- 処理ノード数は 93 台。Map スロット数、Reduce スロット数ともに 260
- 各処理に使用する JavaVM ヒープサイズは、最大 400MB を設定

携帯電話プローブデータの処理量を変化させた場合の処理時間について測定結果を図 3-3 に示す。

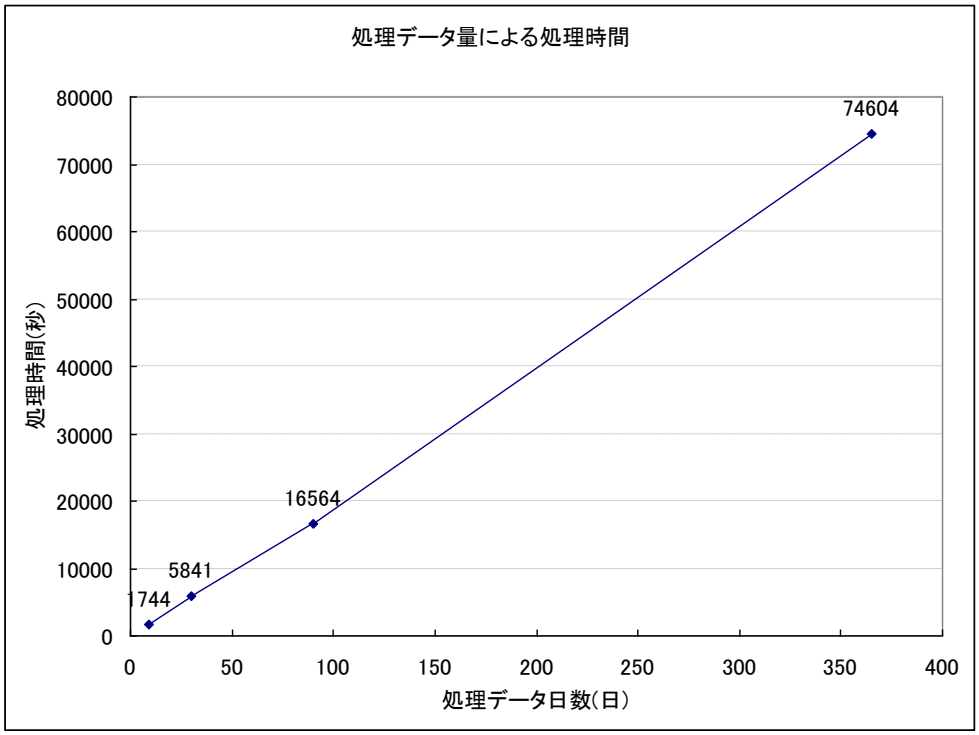


図 3-3 処理データ量変化による処理時間

以上より、処理データ量を変化させた場合には、処理時間が指数関数的に変化することなく、スケーラビリティを実現していることを確認した。

3.4.3 道路種別変化による性能への影響について

渋滞解析アプリケーションでは、プローブデータの位置情報から道路上の位置を特定させる。この、特定する道路データを変更することで処理時間にどのような影響を与えるか確認する。測定条件は以下の通りである。

- タクシープローブデータは、1ヶ月分のデータを使用
- 携帯電話プローブデータは、9日分のデータを使用
- 道路種別を「主要道路」と「一般道路」の2パターンを使用
- 処理ノード数は93台。Mapスロット数、Reduceスロット数ともに260
- タクシープローブデータは、1Map処理あたり640KBで設定
- 携帯電話プローブ処理のReduce分割数は、1300に設定
- 各処理に使用するJavaVMヒープメモリサイズは、最大400MBで設定

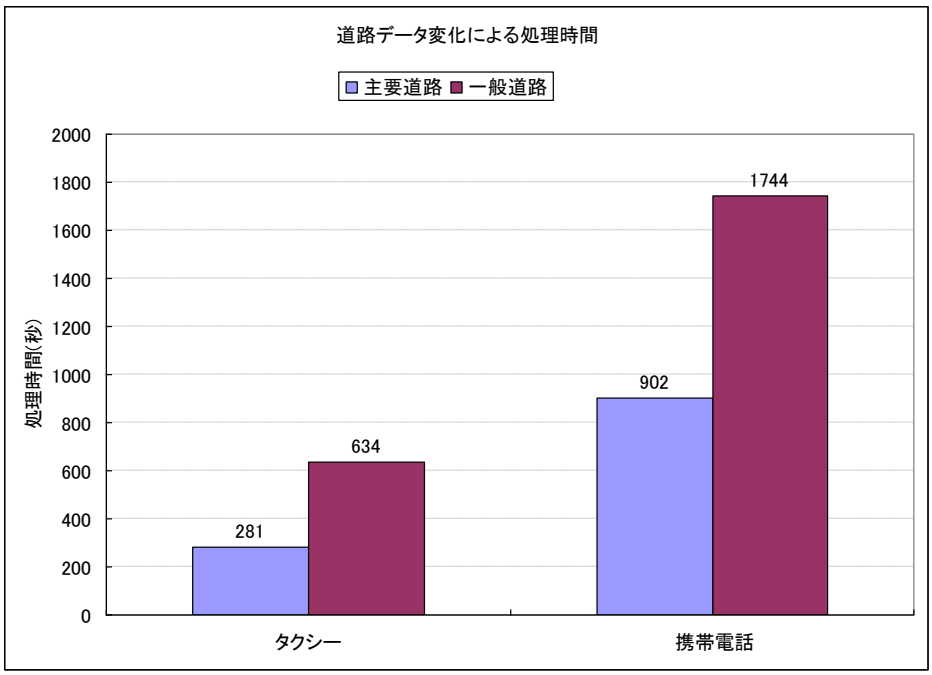


図 3-4 道路種別変化による処理時間の変化

「主要道路」と「一般道路」の道路データ量は、「主要道路」約105万件、「一般道路」約388万件である。この道路データ量の差が処理時間の比率に影響を与えている結果となった。

3.5 まとめ

本章では、2 章で取り上げた渋滞解析アプリケーションで、MapReduce ジョブとしての性能やアプリケーション実行で得られる結果の精度に関して説明した。

大量のデータを処理する「渋滞統計生成処理」に着目して、アプリケーションの処理データ量や処理サーバ台数でのスケーラビリティと道路種別による処理精度について検証した。「渋滞統計生成処理」の MapReduce ジョブを実行した結果、スケーラビリティについては以下を確認できた。

- 処理データ量を一定、処理ノード数を変化させたときの処理時間のスケーラビリティ
- 処理ノード数を一定、処理データ量を変化させたときの処理時間のスケーラビリティ

また、処理時間と処理性能の影響を確認するため、渋滞解析アプリケーションで使用する道路データを「主要道路」、「一般道路」と変化させた MapReduce ジョブを実行した。その結果、携帯電話プローブ解析 MapReduce ジョブでは、道路データ量を「主要道路」から「一般道路」に変化させたとき、処理時間は約 1.9 倍増加したことを確認した。

さらに、処理時間見積もりとして 24 台環境での少量データ実行結果を用いて、大規模環境での処理時間を見積もり、実測値との誤差が 10%程度であることを確認した。

4 可用性を担保する技術の開発

本章では、Hadoop 基盤における可用性を担保する技術について、基盤を構成する要素ごとに方式を検討し、実証実験において効果を測定した結果を報告する。

Hadoop管理ノードの冗長化の検討については10章に詳細に記載しているため参照されたい。

4.1 実証実験における可用性目標と課題

本節では、実証実験環境における可用性の目標を定義し、Hadoop 基盤の構成要素ごとに可用性確保の方針を決定する。

4.1.1 実証実験における可用性目標

本実証実験は 2 章で作成したアプリケーションを用い、収集したプローブ情報を 5 分間隔で定期的に解析を行う。利用者へは解析結果を 5 分間隔で提供するため、可用性の目標を以下に定義する。

- システム構成要素の単一故障発生時は、1 回の出力結果の欠損を許容する
  故障発生時は 5 分以内に復旧し、次回の解析処理が正常に行える状態となること
- 単一故障発生時にデータが失われないこと
  Hadoop 基盤内に保持するプローブ情報および解析結果は構成要素の単一故障時にも消失せず継続して使用できること
- 単一故障発生時にも想定時間内に解析が終了すること
  故障発生時の処理能力の低下を 2 割とする

4.1.2 Hadoop に備わる可用性確保の仕組み

クラウド型分散処理基盤は、1.2.2.2 で記載したとおり、一部のサーバが故障した場合も残りのサーバで動作が継続できるという特徴をもっている。Hadoop においても、あるスレーブサーバが故障した場合は、自動的に他のスレーブサーバで処理を継続する仕組みをもっている。

Hadoop 基盤の持つ次の 2 つの機能について、それぞれが持つ可用性確保の仕組みを以降で説明する。

- HDFS が複数のサーバで分散してデータを保持する
- MapReduce を用いて複数のサーバで並列処理を行う

4.1.2.1 HDFS レイヤでの可用性確保の仕組み

HDFS でのファイルレプリケーションに関する処理を図 4-1 に示す。HDFS 上に格納されたデータは DataNode に実データを格納し、NameNode に DataNode と実デ

ータのマッピング情報をメタデータとして格納する。また実データは複数の DataNode でレプリケーションして保持し、DataNode の可用性を確保している。またレプリケーションの際には RackAwareness という仕組みにより、ネットワークトポロジを意識したレプリケーションを行うことができ、ネットワーク構成の可用性も確保している。

これに対してメタデータは NameNode 上にのみ保持するため、故障時には HDFS へのアクセスが不可能となる。

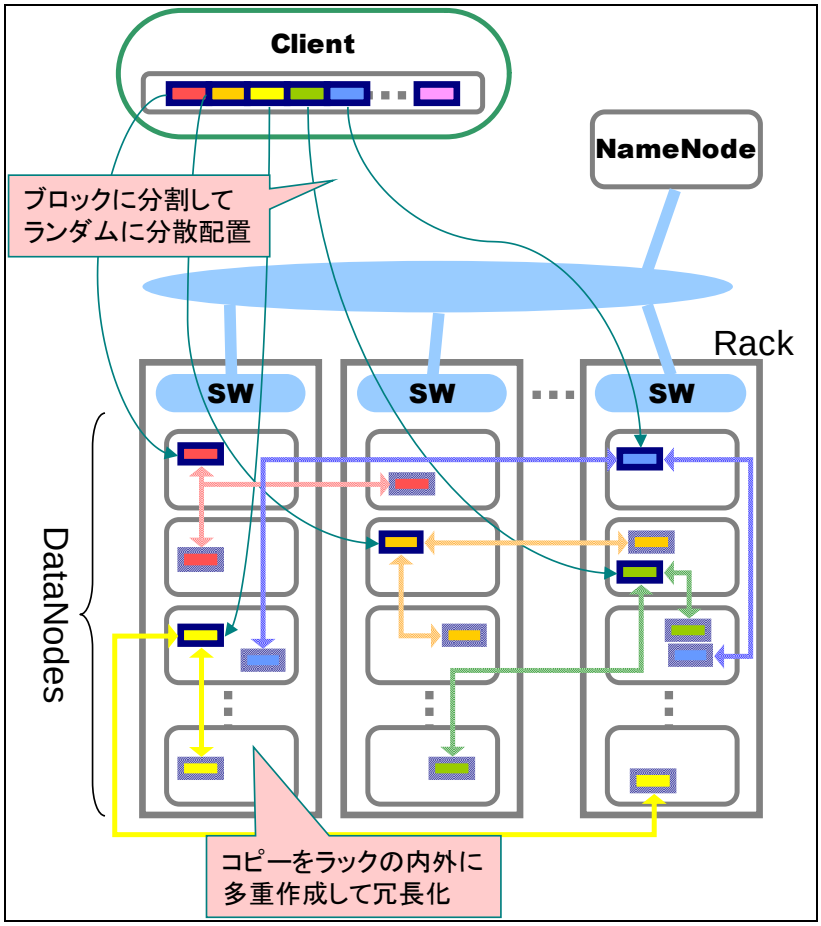


図 4-1 HDFS のレプリケーションの仕組み

メタデータの可用性確保のために、HDFS は次の 2 つの仕組みを持っており、データ消失持にも復旧することが可能である。

- メタデータを複数のファイルシステムに保存する
- SecondaryNameNode に定期的にバックアップデータを保存する

図 4-2 に SecondaryNameNode による定期的なバックアップの仕組みを示す。SecondaryNameNode はチェックポイント処理として、NameNode のメタデータの

情報を保存する。また NameNode ではチェックポイント後のメタデータのみ保持するため、メタデータのデータ量を削減する効果がある。

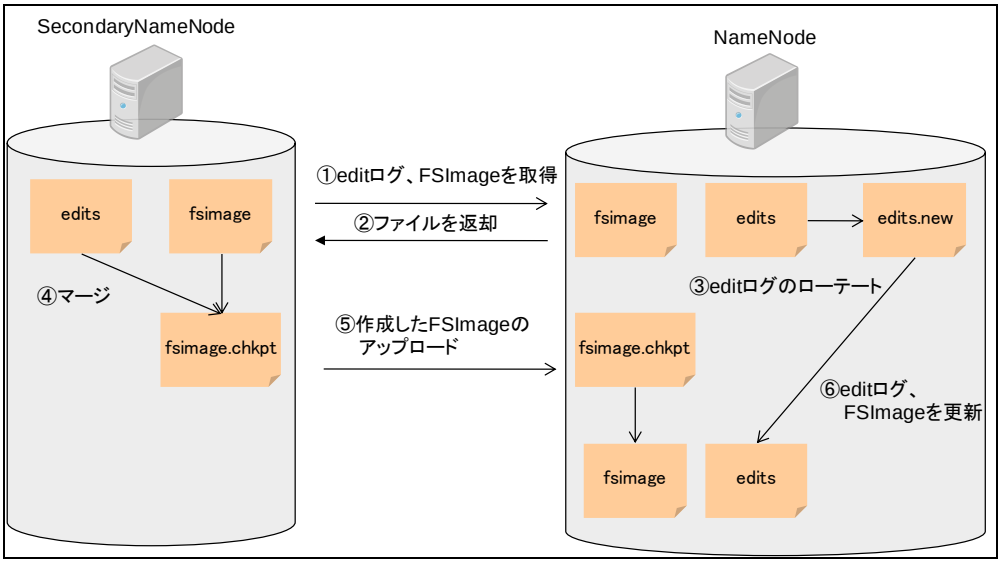


図 4-2 SecondaryNameNode の仕組み

Hadoop に存在するメタデータの可用性確保方式はどちらも故障時に自動的なデータの引継ぎや切り替えを実施するものではないため、メタデータのバックアップ用途として使用し、可用性確保のために別途方式を検討する。

4.1.2.2 Hadoop ジョブ管理での可用性確保の仕組み

MapReduce のジョブフローを図 4-3 に示す。MapReduce ジョブは各 TaskTracker でタスクとして実行され、実行状況や結果を JobTracker に通知する。タスク自体が失敗した場合は TaskTracker 内で再実行を行い、TaskTracker に故障が発生した場合は JobTracker が他の TaskTracker にタスクを割り当てる。

このため MapReduce ジョブ中に Hadoop スレーブサーバに故障が発生しても、ジョブ全体は継続して実行される。

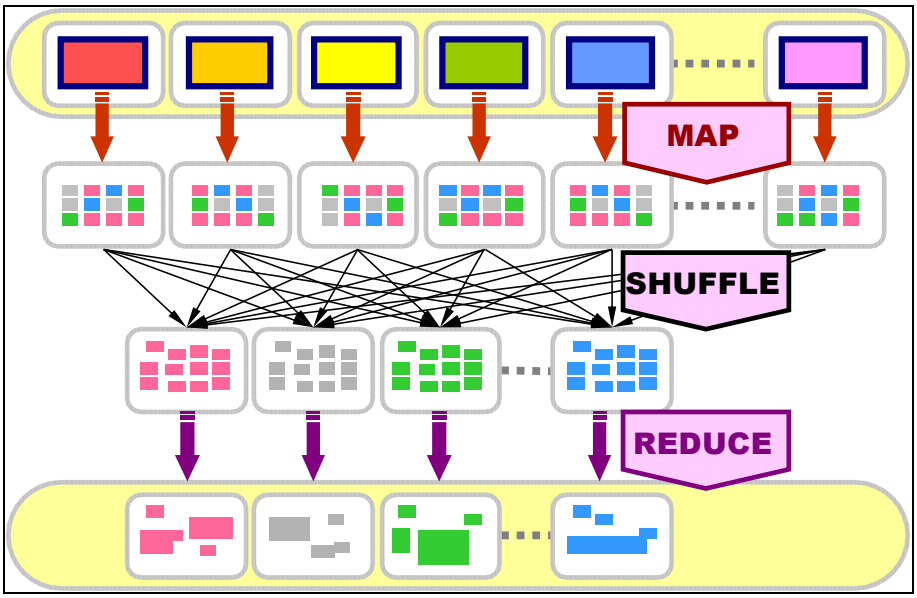


図 4-3 MapReduce ジョブの仕組み

4.1.2.3 Hadoop の可用性動作確認

Hadoop に備わる可用性確保の仕組みに対して動作確認を行った結果を示す。

(1) HDFS レイヤ

HDFS の観点での試験項目と結果を表 4-1 に示す。

表 4-1 HDFS 可用性試験

No.	試験項目	結果	備考
1	ファイル書き込み中の DataNode 故障	ファイル書き込みに成功する	デフォルト設定では停止後 10 分程度 (注)経過すると Hadoop 基盤から切り離される
2	ファイル読み込み中の DataNode 故障	ファイルの読み込みに成功する	
3	DataNode 復帰後の再同期	一時的に Hadoop 基盤から除外されていた間の変更が適用される	

(注) 2 × heartbeat.recheck.interval (300 秒)+ 10 × dfs.heartbeat.interval(3 秒)

(2) MapReduce レイヤ

MapReduce 観点での試験項目と結果を表 4-2 に示す。MapReduce アプリ

ケーションとしてはHadoop付属のTerasortを使用した。

表 4-2 MapReduce 可用性試験

No.	試験項目	結果	備考
1	Map実行中のTaskTracker故障	他のTaskTrackerでタスクが実行される	
2	Map実行中のMapタスク強制終了	同じTaskTrackerにてタスクが再実行される	
3	Reduce実行中のTaskTracker故障	他のTaskTrackerでタスクが実行される	ジョブ実行時間が遅延する
4	Reduce実行中のReduceタスク強制終了	同じTaskTrackerにてタスクが再実行される	

故障時にもジョブが正常に終了することから、Hadoop基盤はHadoopスレーブサーバの故障に対して可用性を確保していることが確認できた。

ただし、ジョブ実行時間という観点では、Reduce中に故障を発生させると、ジョブの実行時間が10分程度長くなることが分かった。図 4-4に示すように、MapReduceジョブは、Mapの実行結果をReduceの入力とするためにHadoopスレーブサーバ間でデータを転送するShuffleフェーズが存在する。Reduce実行中にサーバが停止すると、JobTrackerは他のHadoopスレーブサーバにReduceタスクを再実行させるが、停止したサーバで実行したMap結果を必要とするため、Shuffle待ちとなる。

Reduceタスクは、Shuffleにタイムアウトが発生した後、Mapタスクが再実行されることにより継続実行される。

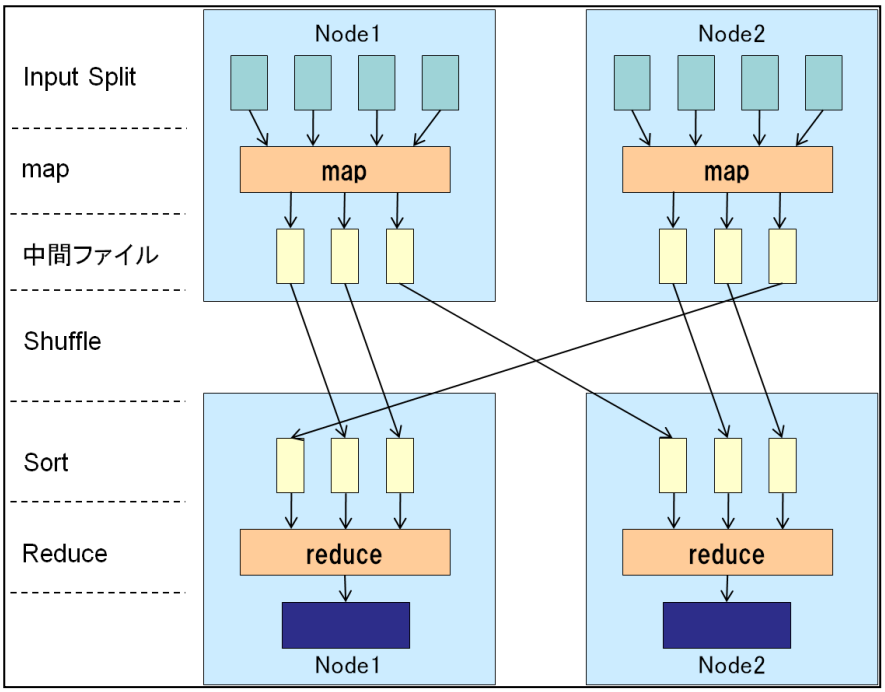


図 4-4 MapReduce における Shuffle

下記に JobTracker のログを抜粋する。Shuffle の失敗と、Map タスクを別サーバで実行していることが分かる。

shuffle 失敗

```
Failed fetch notification #1 for task attempt_201001130758_1351_m_000053_0
Failed fetch notification #2 for task attempt_201001130758_1351_m_000053_0
Failed fetch notification #3 for task attempt_201001130758_1351_m_000053_0
Too many fetch-failures for output of task: attempt_201001130758_1351_m_000053_0 ... killing it
Error from attempt_201001130758_1351_m_000053_0: Too many fetch-failures
Choosing a non-local task task_201001130758_1351_m_000053
Adding task 'attempt_201001130758_1351_m_000053_1' to tip task_201001130758_1351_m_000053, for
tracker 'tracker_r7-2-0-16.example.net:localhost.localdomain/127.0.0.1:44354'
```

map 再実行

4.1.3 Hadoop 基盤における可用性確保の方針

Hadoop に存在する機能を踏まえて、本実証実験を構成する機器について、故障時の影響を整理した。構成を図 4-5 に、影響を表 4-3 に示す。

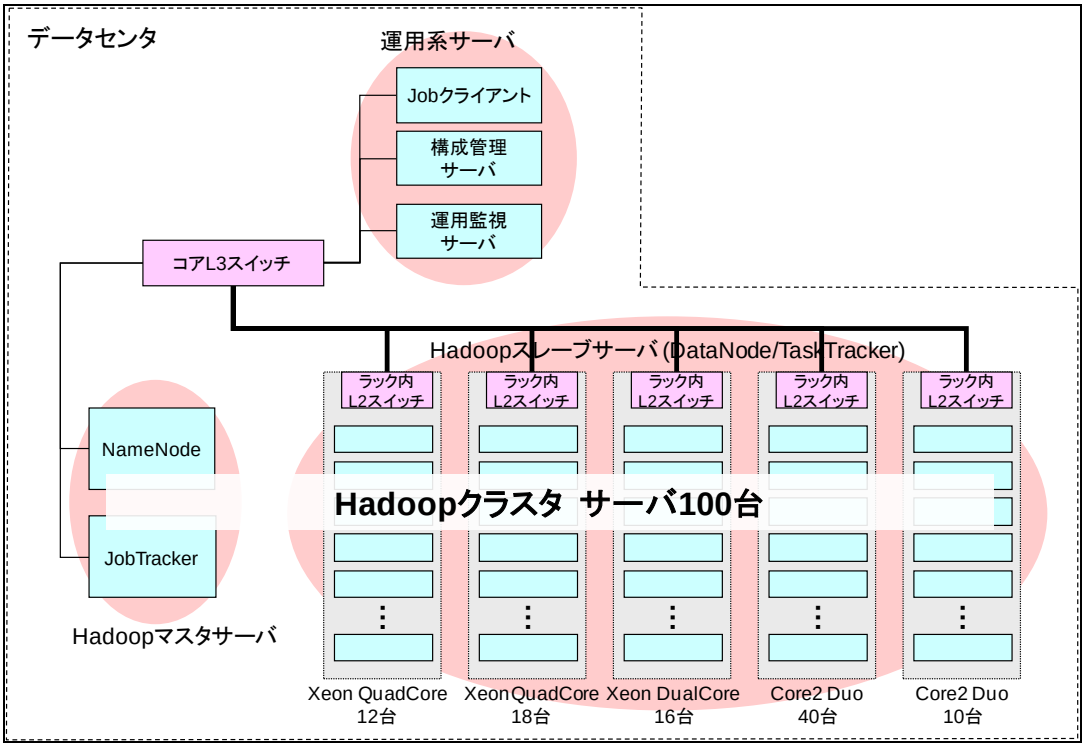


図 4-5 システム全体構成図

表 4-3 構成要素毎の故障時の影響

No.	構成要素	用途	故障時の影響
1	コア L3 スイッチ	ラック間を接続し、ルーティングを行う	停止
2	ラック内 L2 スイッチ	ラック内の Hadoop スレーブサーバを収容する	ラック単位で縮退
3	Hadoop マスタサーバ NameNode	HDFS メタ情報を格納する	停止
4	Hadoop マスタサーバ JobTracker	MapReduce ジョブの状態を管理する	停止
5	Hadoop スレーブサーバ	HDFS のデータ格納、MapReduce ジョブを実行する	機器単位で縮退
6	構成管理サーバ	DHCP/DNS 機能を提供する	特になし
7	運用管理サーバ	システム内の機器の性能情報収集と、故障検知を行う	監視が停止

各機器の故障時の影響から冗長化の方針を次のとおり分類する。

・ 処理全体が停止する機器

コア L3 スイッチ、Hadoop マスタサーバは機器冗長化の対象とする

・ 処理能力が縮退する機器

ラック内 L2 スイッチ、Hadoop スレーブサーバは冗長化しない

・ MapReduce ジョブに影響を与えない機器

運用管理サーバ、構成管理サーバは冗長化しない

4.1.4 コア L3 スイッチ可用性確保の方針

コア L3 スイッチは停止するとシステム全体が停止するため、部位および機器の冗長化構成をとる。故障対策を表 4-4 に示す。

表 4-4 コア L3 スイッチの故障対策

No.	故障部位	故障時の影響	対策
1	機器故障	Hadoop マスタサーバ- Hadoop スレーブサーバ間で通信ができず、システムが停止する	機器の冗長化
2	ポート故障	接続先の機器との通信が行えない	スイッチ間インタフェースまたはスイッチ-サーバ間インタフェースの冗長化
3	LAN ケーブル故障	接続先の機器との通信が行えない	スイッチ間インタフェースまたはスイッチ-サーバ間インタフェースの冗長化

4.1.5 Hadoop サーバ可用性確保の方針

Hadoop マスタサーバ停止時には HDFS に格納しているブロックがどの DataNode に所属するか不明となり、Hadoop として処理の継続が不可能となる。このように Hadoop マスタサーバは停止すると MapReduce ジョブが停止するため、サーバ単体として部位を冗長化するとともに、サーバ 2 台による冗長化構成をとる。

マスタサーバを構成する部位に故障が発生した場合の影響と、対策方針を表 4-5 に示す。

表 4-5 Hadoop マスタサーバの故障対策

No.	分類	故障部位	故障時の影響	対策
1	ハードウェア故障	CPU	サーバがダウンし MapReduce ジョブが停止する	待機系サーバへフェイルオーバする
2		メモリ	サーバがダウンし MapReduce ジョブが停止する	待機系サーバへフェイルオーバする

No.	分類	故障部位	故障時の影響	対策
3		HDD	サーバがダウンしMapReduceジョブが停止する	RAID1によりミラー化する
4		NIC	Hadoopスレーブサーバと通信できないためMapReduceジョブが停止する	bondingにより冗長化する
5		電源	サーバがダウンしシステムが停止する	冗長化電源を使用する
6	ソフトウェア故障	OS	サーバがダウンしMapReduceジョブが停止する	OSを再起動する
7		NameNodeプロセス	HDFSへのアクセスが不可能になりシステムが停止する	プロセス再起動を行う
8		JobTrackerプロセス	MapReduceジョブが停止する	プロセス再起動を行う
9	論理故障	所持データ	MapReduceジョブが停止する	定期的なバックアップを行う

4.1.6 Hadoop スレーブサーバ可用性確保の方針

Hadoop基盤はHadoopスレーブサーバが故障した場合に該当サーバを切り離し、縮退運転を行う仕組みを持つ。またクラウド基盤では一般的にスレーブサーバの台数が多いことが想定されるため、サーバ数台の故障の影響は基盤全体としては低いと言える。表 4-6にHadoopの持つ機能ごとに、Hadoopスレーブサーバが故障した場合の影響を示す。台数が多い環境では故障時の影響はほとんど無視できるため、可用性よりもコストパフォーマンスを重視する方針で機器を選定し、サーバ単体の部位の冗長化は行わない。

なおHadoopスレーブサーバの単一故障によりデータが失われないようにするため、HDFSのレプリケーションは3とする。

表 4-6 Hadoop スレーブサーバ故障時の影響

No.	観点	故障時の影響
1	HDFS	故障したHadoopスレーブサーバが持つデータのレプリケーションが行われ、Hadoop基盤全体のディスク容量が減少する
2	MapReduce	Hadoop基盤全体のMapReduceジョブ処理能力が低下する

4.1.7 ラック内 L2 スイッチ可用性確保の方針

ラック内 L2 スイッチが故障すると、そのスイッチに収容されている Hadoop スレーブサーバすべてが Hadoop 基盤から切り離される。ラック内 L2 スイッチに故障が発生しても Hadoop 基盤全体として正常に稼働するためには、以下の点について検討が必要である。

- 計算能力の低下が許容できる範囲か
- 余分にディスク領域を保持できるか
- 再レプリケーションのオーバーヘッドが MapReduce ジョブに影響を与えないか

本実証実験では処理能力の低下は 2 割程度を許容するため、Hadoop スレーブサーバを 6 つのラックに分散して配置することとし、ラック内 L2 スイッチの単一故障時の処理低下やオーバーヘッドを 1/6 に抑える。

4.2 課題の解決方法

本節では、実証実験において各構成要素の可用性確保の方式を検討し、実機による確認結果を記載する。検討対象は前節で可用性を確保すると方針決定したネットワーク構成、Hadoop マスタサーバである。

4.2.1 ネットワークの可用性確保方式

通常、ネットワークの可用性はネットワークの冗長化構成をとることにより実現する。Hadoop においても同じであるが、すべてのネットワーク機器を冗長化することは非常にコストが高くなる問題がある。特に、サーバにはコストパフォーマンスが高いコモディティ製品を採用するため、システム全体に占めるネットワーク機器の価格の割合が相対的に高くなる傾向がある。そのため、ネットワークの可用性とコストのバランスを考える必要がある。

4.1.4 のコア L3 スイッチの可用性確保方針から、図 4-6 に示すコア L3 スイッチの冗長化と、それにともなうラック内 L2 スイッチとコア L3 スイッチを接続する経路の冗長化が必要である。以下でそれぞれの冗長化方式を検討する。

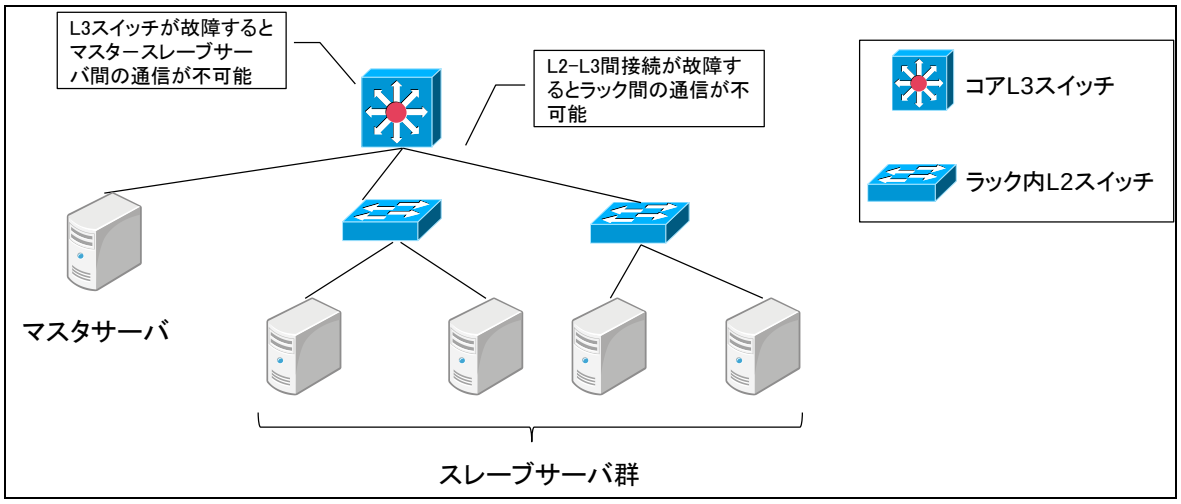


図 4-6 冗長化必要箇所

L3 スイッチの冗長化方式

L3 スイッチの冗長化方式としては表 4-7 に示すように VRRP のような冗長化プロトコルを使用してゲートウェイを冗長化することが一般的である。今回使用する L3 スイッチはスタック接続することによりポート収容率を高めつつ冗長性を保つ機能を有している。

表 4-8 にて両者を比較した結果、実証実験ではネットワークトポロジを単純化するためスタック接続による冗長化を採用する。

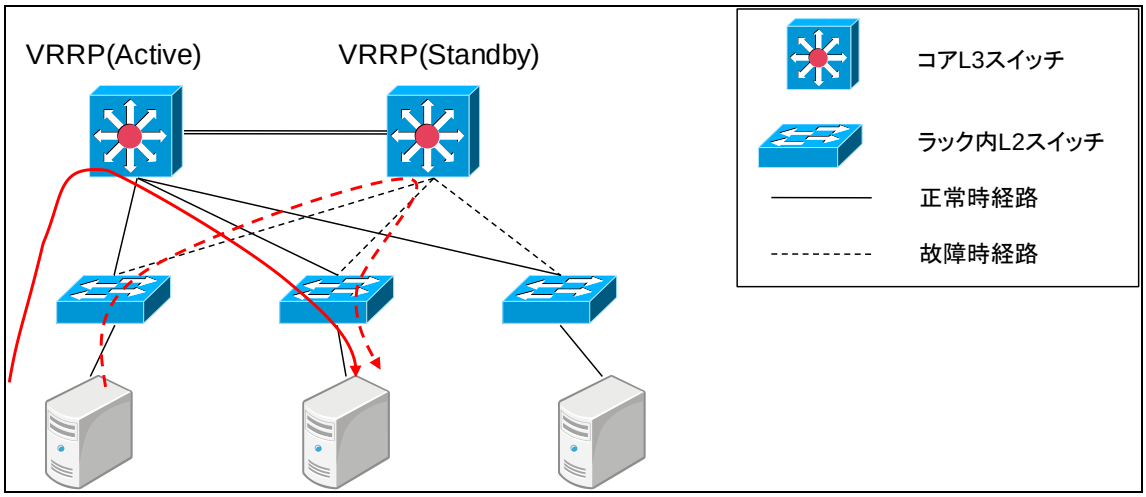


表 4-7 VRRP の例

表 4-8 L3 スイッチ冗長化方式の比較

No.	方式	想定切り替え時間	構成の制約	運用性
1	VRRP	10 秒	特になし	Active/Standby を意識した設定が必要
2	スタック接続	5 秒	スタック接続するには同一機種および同一 OS が必要	論理的に 1 台のスイッチのため構成が管理しやすい

L2 レイヤの冗長化

L3 スイッチとラック内 L2 スイッチ間の接続を冗長化する。L2 レイヤでの冗長化方式として、スパニングツリー(STP)とリンクアグリゲーションが存在する。

本実証実験ではラック間で大量の通信が発生するため、可用性を確保しつつ使用するネットワーク帯域を増強できるリンクアグリゲーションを採用する。

以上から実証実験のネットワークトポロジを図 4-7 に示す。

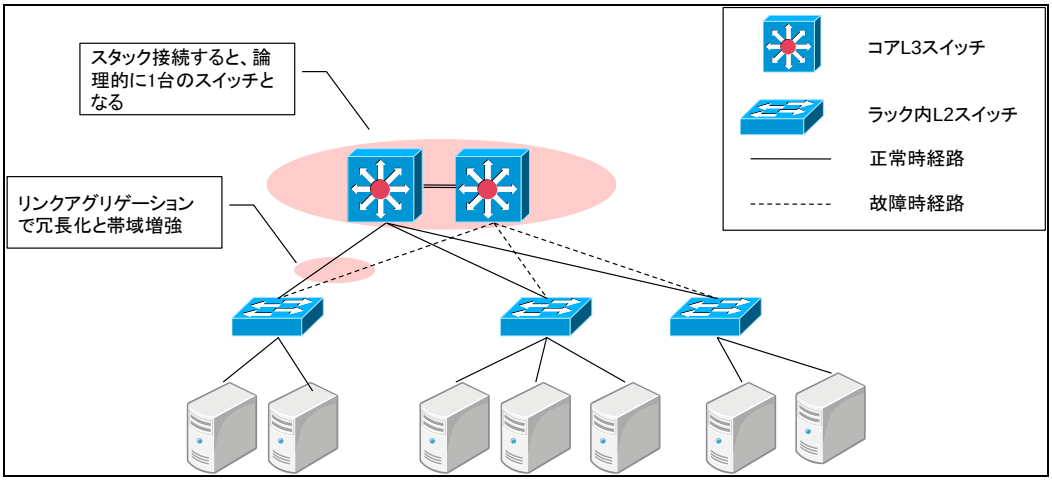


図 4-7 実証実験ネットワークトポロジ

4.2.2 ネットワークの可用性動作確認

実証実験用クラウド環境において、ネットワークの可用性について動作確認を行った。表 4-9 に試験項目と結果を示す。

表 4-9 ネットワーク試験

No.	試験項目	結果
1	コア L3 スイッチ(マスタ)の故障	スレーブスイッチ経由で処理が継続する

No.	試験項目	結果
2	コア L3 スイッチ(スレーブ)の故障	使用できるネットワーク帯域が減少する
3	ラック内 L2 スイッチの故障	ラック単位で縮退する
4	コア L3 スイッチーラック内 L2 スイッチ接続の故障	リンクアグリゲーションしている他の経路で処理が継続する

結果として、1 秒でスイッチの切り替えまたは経路の切り替えが行われ、Hadoop の動作が継続できることがわかった。

4.2.3 Hadoop マスタサーバの可用性確保方式

4.1.3 の Hadoop サーバの可用性確保の方針から、各種故障に対しての対策を以下で検討する。

ハードウェア故障対策

ハードウェア故障の対策を実現するには HA クラスタによる冗長化構成をとることが一般的である。

本実証実験ではオープンソースソフトウェアの HA クラスタとして Heartbeat を使用し、メタデータの DRBD でミラーリングした構成をとる。HA クラスタの構成イメージを図 4-8 に示す。

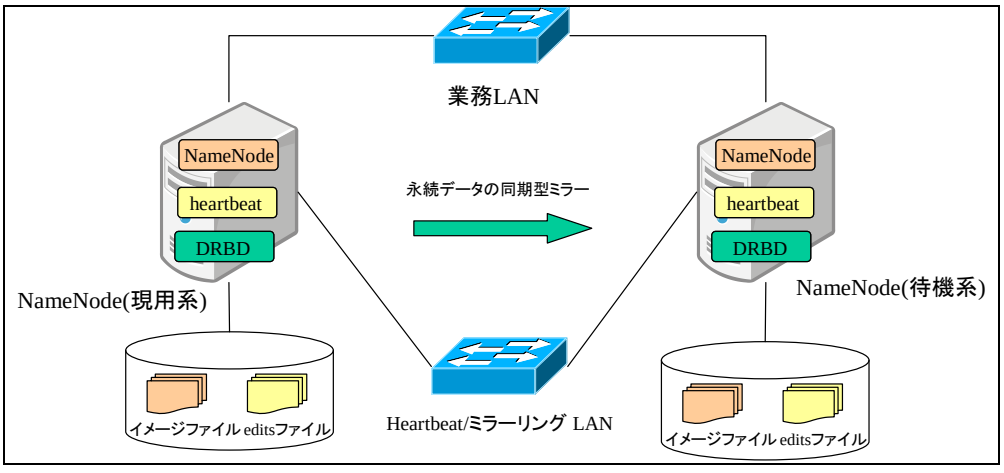


図 4-8 HA クラスタ構成

論理故障対策

NameNode の持つメタデータのバックアップ方式を表 4-10 から決定する。

SecondaryNameNode は NameNode に昇格する機能を持つため、本実証実験では SecondaryNameNode を採用し、JobTracker サーバに同居させる。

表 4-10 メタデータバックアップ方式

No.	方式	メリット	デメリット
1	複数のファイルシステムに出力	故障時に直前のデータまでの復旧が可能	NFS サーバ等の機能が追加で必要 NFS サーバ故障時の検討が必要
2	SecondaryNameNode	プロセスを起動するだけでよい JobTracker と同居させることで簡易な構成で実現可能	チェックポイントから故障発生時までのデータが失われる

4.2.4 Hadoop マスタサーバの可用性動作確認

実証実験用クラウド環境において、Hadoop マスタサーバの可用性について動作確認を行った。表 4-11 に確認項目と結果を示す。

表 4-11 Hadoop マスタサーバ可用性試験

No.	試験項目	結果
1	現用系電源断	待機系で仮想マシンの処理が再開される
2	現用系電源復帰	影響なし
3	現用系ネットワーク断(片系)	bonding により経路が切り替わる
4	現用系ネットワーク断(両系)	待機系で仮想マシンの処理が再開される
5	待機系電源断	影響なし
6	待機系電源復帰	影響なし
7	ハートビートネットワーク断(両系)	スプリットブレイン状態のため STONITH 機能により強制再起動による切り替えが発生する
8	ハートビートネットワーク断(片系)	影響なし
9	NameNode プロセス停止	NameNode プロセスがフェイルオーバ

No.	試験項目	結果
10	JobTracker プロセス停止	JobTracker プロセスがフェイルオーバ

NameNode フェイルオーバ時の影響

Heartbeat による切り替え自体は 2 分程度で行われたが、NameNode は起動後に Safemode となるため、HDFS にアクセスできない時間が存在する。

このためジョブの実行時間は 3 分弱増加した。

JobTracker フェイルオーバ時の影響

Heartbeat の切り替えにより、ジョブクライアントから実行したジョブは異常終了し、ジョブのステータスは実行中のままとなった。

Heartbeat による切り替え後、新規で実行されたジョブは正常終了した。

結果として Hadoop マスタサーバの切り替えは最大で 3 分程度必要であった。また JobTracker 停止時には切り替え後にジョブクライアントからの MapReduce ジョブの再実行が必要であり、タイミングによって複数回の解析失敗が発生する。

このことから、HA クラスタ構成では実証実験における可用性目標を達成することができないことが分かった。

4.3 Hadoop マスタサーバ故障への対策検討

4.2.4 で記載したように、Hadoop マスタサーバを HA クラスタにて構成した場合、下記の課題があることが分かった。

- NameNode 起動後に一定時間 Safemode となるため、切り替えに時間がかかる
- JobClient との通信が切断されるため、ジョブの再実行が必要である

本節ではこれらの問題を回避するためにソフトウェア FT ソリューションの適用性を評価し、実機による検証結果を示す。

4.3.1 ソフトウェア FT の適用性評価

本節ではハードウェアの冗長化方式としてソフトウェア FT に着目し、本実証実験に適用できるかを検討する。

4.3.1.1 ソフトウェア FT とは

ソフトウェア FT とは、物理的に 2 台のハードウェアを用意し、同期制御することによって、仮想マシン単位で冗長性を確保する技術である。物理サーバに故障が発生しても仮想化したサーバは別の物理サーバ上で継続して稼働する。同期制御により、メモリ上のデータやコネクションの状態も引き継がれるのが大きな特徴である。

ソフトウェア FT の動作イメージを図 4-9 に示す。

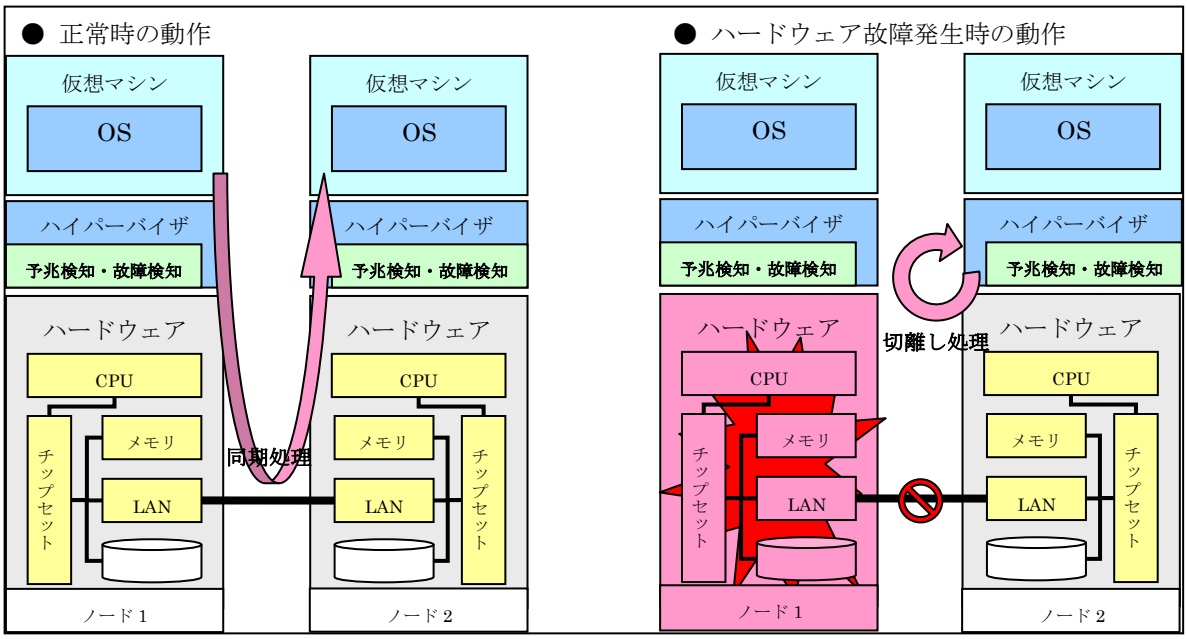


図 4-9 ソフトウェア FT 動作イメージ

本実証実験では積極的にオープンソースソフトウェアを採用するという方針から、ソフトウェア FT の中でも、Kemari を採用する。Kemari はメモリ同期方式を採用したソフトウェア FT であり、I/O イベント発生時にネットワークを経由して他系のサーバに同期を行う機能を持つ。

Kemari の同期処理のイメージを図 4-10 に、特徴を表 4-12 に示す。他のソフトウェア FT 製品は CPU をソフトウェアで同期するのに対し、メモリ同期を行う Kemari はオーバーヘッドが少ない反面、Kemari 単体では故障検知や切り替え機能を持たないことが特徴といえる。

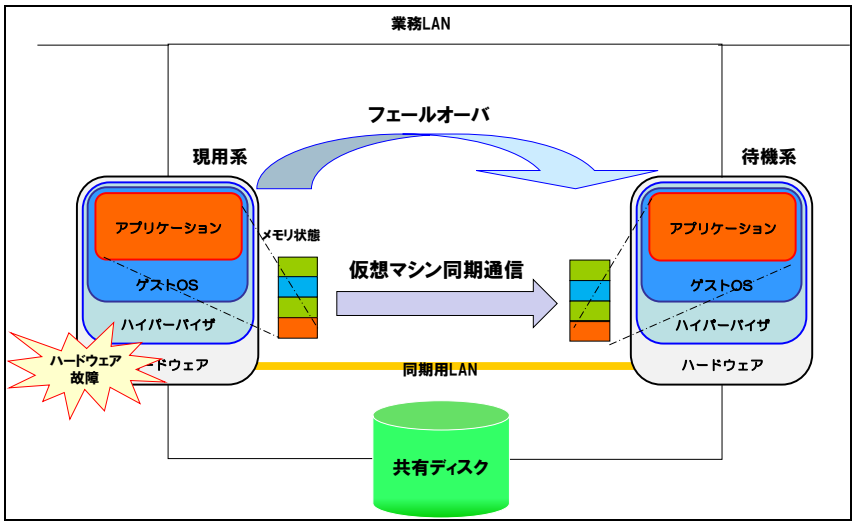


図 4-10 Kemari 同期処理

表 4-12 Kemari の特徴

No.	長所	短所
1	他のソフトウェア FT 製品に比べて、ハードウェア制約が少ない	同期用に使用するネットワークの帯域が大きい
2	ゲスト OS に複数の CPU を割り当てることが可能	ハードウェアの故障予兆や故障検知の仕組みと連動する機能がない

4.3.1.2 ソフトウェア FT の適用範囲

Kemari を採用する際の検討ポイントは次のとおりである。以降で Hadoop マスタサーバへの適用性を判断する。

- サーバ故障時の切り替えがアプリケーションに対して透過的である
- 性能のオーバーヘッドが多い

Hadoop マスタサーバ故障時の影響

Hadoop マスタサーバが一時的に停止した場合にはジョブクライアントからのジョブの再実行が必要である。このためアプリケーション側は再実行を意識した作りを行わなければならないという問題があり、切り替えがアプリケーションに対して透過的であるというのはメリットが大きい。

性能オーバーヘッド

Hadoop マスタサーバの二つの機能に求められるハードウェアスペックは次のとおりである。Hadoop マスタサーバの機能として高い性能は必要ないため、オーバーヘッドの影響は少ないと考えられる。

(1) NameNode

HDFS のメタデータをメモリ上に保持するため、HDFS のファイル数分のメモリが必要である。メタデータ 1 ファイルあたり 200byte メモリを使用するので、8GB メモリあれば HDFS として 40TB のデータを保持でき、実証実験として必要十分である。

NameNode の負荷特性はメモリ使用量である。

(2) JobTracker

JobTracker に必要なハードウェアスペックは MapReduce ジョブの実行数に依存する。ジョブ自体は TaskTracker で実行するため、高い性能は不要である。

以上から Hadoop マスタサーバは Kemari を適用した場合に効果が得られると考え

る。

4.3.2 ソフトウェア FT の検証結果

Kemari を採用した構成で、可用性に関する検証、性能オーバーヘッドに関する検証を実施した。結果を示す。

可用性検証

Hadoop マスタサーバにソフトウェア FT 技術を適用し、可用性について動作確認を行った。表 4-13 に検証項目と結果を示す。

表 4-13 ソフトウェア FT 可用性試験

No.	試験項目	結果
1	現用系電源断	待機系で仮想マシンの処理が再開される
2	現用系電源復帰	影響なし
3	現用系ネットワーク断(片系)	bonding により経路が切り替わる
4	現用系ネットワーク断(両系)	待機系で仮想マシンの処理が再開される
5	待機系電源断	影響なし
6	待機系電源復帰	影響なし
7	ハートビートネットワーク断(両系)	スプリットブレイン防止の STONITH 機能により強制的に切り替わる
8	ハートビートネットワーク断(片系)	影響なし

結果として、Hadoop マスタサーバの切り替えは 1 秒程度で行われ、ジョブクライアントからは切り替えを意識せず Hadoop の動作が継続できることがわかった。

性能オーバーヘッド検証

Kemari によるオーバーヘッドの影響有無を調査するため、表 4-14 に Kemari 有環境/Kemari 無環境での性能比較を行った結果を示す。

表 4-14 Kemari 性能オーバーヘッド試験

No.	測定ツール	対象環境	測定結果	備考
1	NNBench	Kemari 無	268tps	Map 数:1、ファイル数:5000
2		Kemari 有	30tps	
3	Terasort	Kemari 無	17.5 分	データサイズ:10GB
4		Kemari 有	16 分	

結果として、NNBench により NameNode 単体では 1/10 の速度低下となったが、Terasort による MapReduce 処理全体としては速度の変化は誤差の範囲内であった。これは、Terasort は各 TaskTracker 上での処理に時間がかかること、および実施したジョブの分割数が少ないため、Hadoop マスタサーバの負荷が低いことが理由であると考えられる。

4.4 まとめと今後の課題

本章の結果をまとめ、残された課題について記載する。

4.4.1 Hadoop 基盤における可用性を確保する技術のまとめ

本章では、Hadoop 基盤の可用性を担保するための技術について検討、実測による効果の確認を行った。

Hadoop マスタサーバの可用性確保のために HA クラスタ構成を採用したところ、切り替えは正常に行えたが、切り替えに時間がかかり、切り替え後のクライアントからの再接続が必要であるといった課題があることが判明した。

この課題への対応としてソフトウェア FT 技術である Kemari を適用し、ハードウェア故障時に無停止で、かつクライアントから透過的に処理が継続できることが確認できた。

4.4.2 今後の課題

Hadoop マスタサーバの可用性確保の方式は Hadoop コミュニティでも議論されており、Hadoop0.21.0 から SecondaryNameNode に代わり StandbyNode が機能追加され、NameNode の保持するメタデータの最新状態を転送することが可能となる予定である。動作イメージを図 4-11 に示す。

このような今後の Hadoop の機能拡張に対して、より親和性の高い可用性確保の方式を継続して検討することが課題である。現状 DRBD を使用してメタデータの同期を行っている部分が不要になるといったことが考えられる。

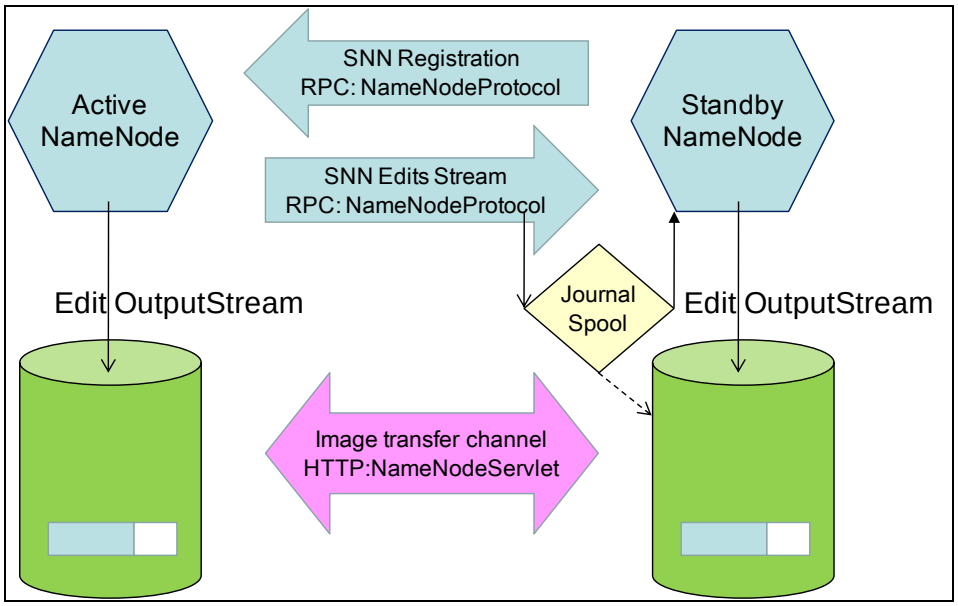


図 4-11 StandbyNameNode 動作イメージ

5　効率的な運用を実現する技術の開発

本章では、Hadoop 基盤の運用項目として「初期構築」、「監視・故障検知」、「維持管理・メンテナンス」、「回復・増設」を取り上げ、Hadoop 基盤運用上の特筆するべき三つの特徴「スケーラビリティ」、「クラスタ構成要素の変更への対応」、「基盤の混在性」を観点とした運用項目の効率性を評価する。効率的な運用のための方式に関しては、11 章、12 章、13 章に詳細を記載する。

5.1 本章における運用性評価観点

本節では、Hadoop 基盤における運用項目を明らかにするとともに、運用性向上のために、Hadoop 分散処理基盤の運用の際に評価するべき観点を明らかにする。

5.1.1　本章における運用の範囲

本章では、表 5-1 の運用項目においてクラウド型分散基盤である Hadoop 基盤の運用性の評価を行う。

表 5-1 本章で評価する運用の分類

No.	項目	運用の概要
1	初期構築	ハードウェアに OS やソフトウェアをインストールして Hadoop による分散計算ができるように構成する。
2	監視・故障検知	クラスタを構成している個々のハードウェア・ソフトウェアの死活監視を実施するとともに、クラスタ全体のサービス継続性や性能などの各種指標値を定常的に監視する。
3	維持管理･メンテナンス	ハードウェア・ソフトウェアの構成情報を管理したり、物理的な配置や故障・回復履歴を管理したりする。また、構築済みのクラスタで使用した設定ファイルを維持管理するなど、構成を管理する。 また、アプリケーションレイヤのメンテナンスを実施する。メンテナンスには、サービスの閉塞・開放や、システムバックアップや、データのバックアップ作業も含まれる。
4	回復・増設	クラスタを構成する各要素に関して故障検知や、その他アラートが挙がった際にどのような手順で復旧させるかを記載する。ハードウェアの交換や、回復後のハードウェアの再構築とクラスタへの組み込みを実施する。 増設では、計算性能をあげる為にラック単位でハードウェアを増設する。

5.1.2 クラウド型分散処理基盤における運用上の特徴

クラウド型分散処理基盤における運用の特徴的な項目として、基盤を構成するサーバが非常に多いことが挙げられる。

また、それに伴い、多数の機器の故障・回復は定常的に行われることを前提として、Hadoop 基盤が設計されていることだけでなく、効率的な機器の故障検知・回復・増設手法が確立されていること、すなわちクラスタ構成要素が動的に変更しうることを前提とした運用性が求められる。

また、本章では、上記に加え、Hadoop 基盤における構成要素が混在することを特徴として取り上げる。これは、故障検知によるサーバ機器の交換、増設時におけるコストパフォーマンスを意識した機器選定によって、長期的な運用の中 Hadoop 基盤を構成する構成要素が混在することは避けがたいためである。

上記 3 つの特徴「スケーラビリティ」、「クラスタ構成要素の変更への対応」、「基盤の混在性」、をクラウド型分散基盤の特徴として取り上げ、本章での評価軸とする。

5.1.3 運用のスケーラビリティと評価観点

本章における運用性のスケーラビリティを定義し、その評価観点を記述する。

一般に Hadoop 基盤は大量のサーバで構成される。従って、台数に比例した運用を行うのではなく、効率的な運用方式を検討することが必要である。運用コストが、システムの規模に比例しないような運用を、本報告書ではスケーラブルな運用と定義する。概念図を図 5-1 に記載する。なお、運用コストは、運用作業者の時間的なコスト、すなわち、作業者の作業時間を基準とする。

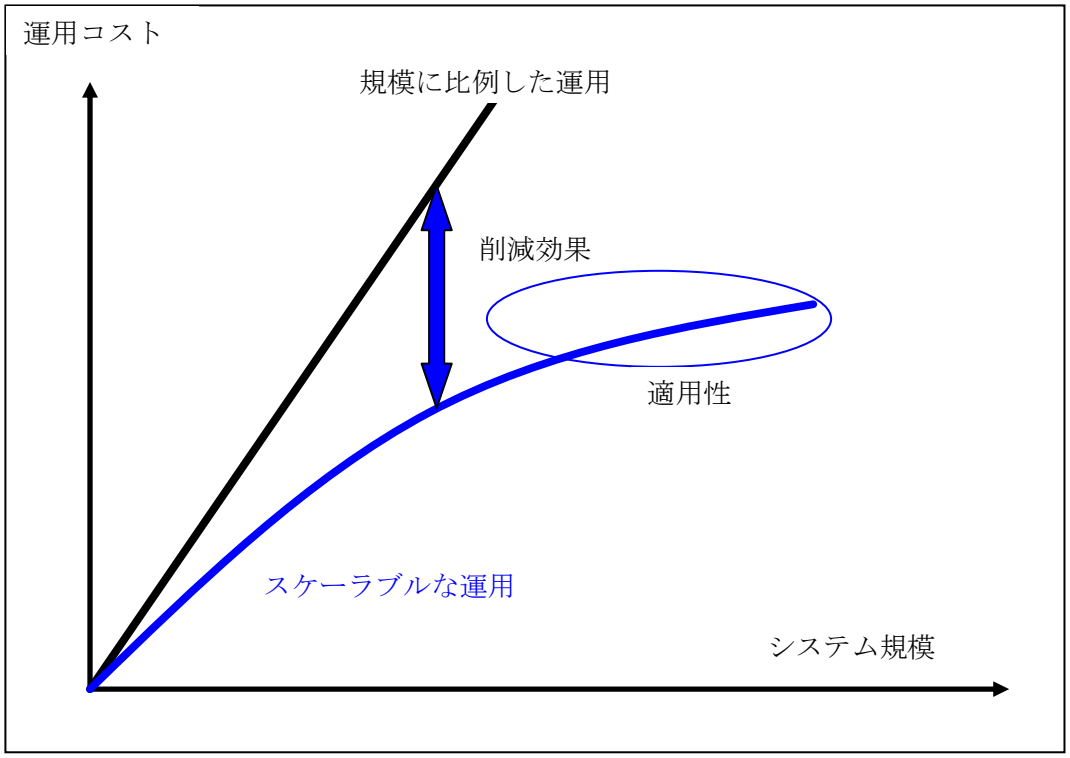


図 5-1 スケーラブルな運用方式

本章では、11 章、12 章、13 章において検討した効率的な運用方式を利用する。それぞれの運用項目にて以下の観点を本章でのスケーラビリティの評価観点とし、表 5-1 の各項目を本章で検証する。

表 5-2 スケーラビリティの評価観点

No	観点	概要
1	有効性	本章において運用コストを測定し、有効であるかを評価する。
2	効率性	規模に比例した運用と比較して効率化の効果を評価する。
3	適用性	本章の検証により、どの範囲まで適用可能かを確認する。また、計測結果に基づき、どの程度まで本方式が適用可能かを評価する。

5.1.4 クラスタ構成要素の変更に関する評価観点

Hadoop 基盤で高頻度に発生する故障からの回復においては、機器故障に伴う新規機器の Hadoop 基盤への組み込みが必要となる。また、システムの長期的な発展を意識すると、取り扱うデータ量や、処理量の増加に伴って、機器の増設を考慮する必要がある。

これらのサーバ構築は、自動化する等の効率化だけでなく、手順としても運用項目として共通化し、運用項目数自体を削減させることで効率化することを目標とする。

本章では、各運用項目に関して、可能な限り共通化された手順を採用することで、運用項目数自体を削減する。そこで、以下を評価観点とする。

・ 共通性

初期構築・故障からの回復・増設に関して共通化・簡略化された運用手順で実施できるかどうかを評価する。

5.1.5 基盤の混在性と評価観点

本章では、Hadoop 基盤は種別の異なるサーバから構成されていることを前提とする。具体的にはベンダ、CPU 周波数、搭載メモリ、内蔵ディスクの違い、等が挙げられる。本章では、各運用項目に関してこれらの違いを隠蔽し、共通化された手順で効率的に運用が可能かを評価する。

上記を鑑み、以下を基盤の混在性を考慮した運用の評価観点とする。

・ 共通性

各運用項目に関して、混在性を意識しない運用が可能であるかを評価する。また、共通化できない運用項目を明らかにする。

5.1.6 評価手法概要

本章での評価概要について記載する。

本章では、5.1.1 で定義した四つの運用項目「初期構築」、「監視・故障検知」「維持管理・メンテナンス」、「回復・増設」それぞれに対して、5.1.2 で明らかにしたクラウド型分散基盤における三つの特徴を踏まえた評価軸「スケーラビリティ」、「クラスタ構成要素の変更」、「基盤の混在性」を評価軸として 11 章、12 章、13 章で開発した運用手法が妥当であるかを確認する。それぞれの評価軸を確認するための評価観点はそれぞれ 5.1.3、5.1.4、5.1.5 に記載する。

すなわち表 5-3 に示す各項目に関して、運用手法を評価し、妥当性を確認する。

表 5-3 各運用項目の評価概要

No	運用項目	評価軸	評価観点	確認する章節
1	初期構築	スケーラビリティ	有効性	5.3.1.3
2			効率性	
3			適用性	
4		クラスタ構成要素の変更	共通性	5.3.1.4
5		基盤の混在性	共通性	5.3.1.5
6	監視・故障検知	スケーラビリティ	有効性	5.3.2.3
7			効率性	

No	運用項目	評価軸	評価観点	確認する章節
8			適用性	
9		クラスタ構成要素の変更	共通性	5.3.2.4
10		基盤の混在性	共通性	5.3.2.5
11	維持管理・メンテナンス	スケーラビリティ	有効性	5.3.3.3
12			効率性	
13			適用性	
14		クラスタ構成要素の変更	共通性	5.3.3.4
15		基盤の混在性	共通性	5.3.3.5
16	回復・増設	スケーラビリティ	有効性	5.3.4.3
17			効率性	
18			適用性	
19		クラスタ構成要素の変更	共通性	5.3.4.4
20		基盤の混在性	共通性	5.3.4.5

5.2 前提条件

本節では運用性検証の前提条件を記載する。

5.2.1 全体構成

図 5-2 に、本章で想定する構成を記載する。

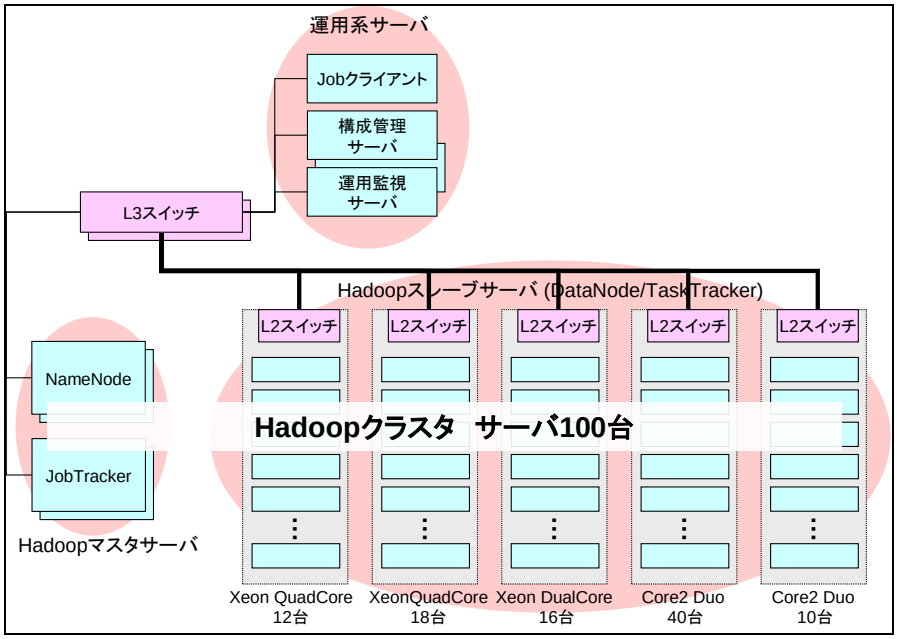


図 5-2 全体構成

5.3 運用性評価

本節では、評価の対象となる各運用項目に対して、クラウド型分散基盤の運用の三つの評価軸及びそれらの評価観点に従って Hadoop 基盤の運用性を評価した結果を記載する。評価項目に関しては、表 5-3 も参照のこと。評価対象は規模に応じて比例する Hadoop スレーブサーバの運用項目を記載する。

5.3.1 初期構築における運用性実験

Hadoop 基盤を構築する機器のうち、システム規模に比例する機器である Hadoop スレーブサーバの構築を実施する。

5.3.1.1 初期構築における実験のスコープ

システム規模に比例する構成要素である Hadoop スレーブサーバに関して、96 台の構築を実施する。Hadoop を構成する機器のラッキング、ケーブリングは済んでいる状態で、構築は 12 章で提案した完全自動化のための方式を利用する。概念図を図 5-1 に記載する。適用性を検証するため、Hadoop スレーブサーバである 96 台の同時構築を実施する。

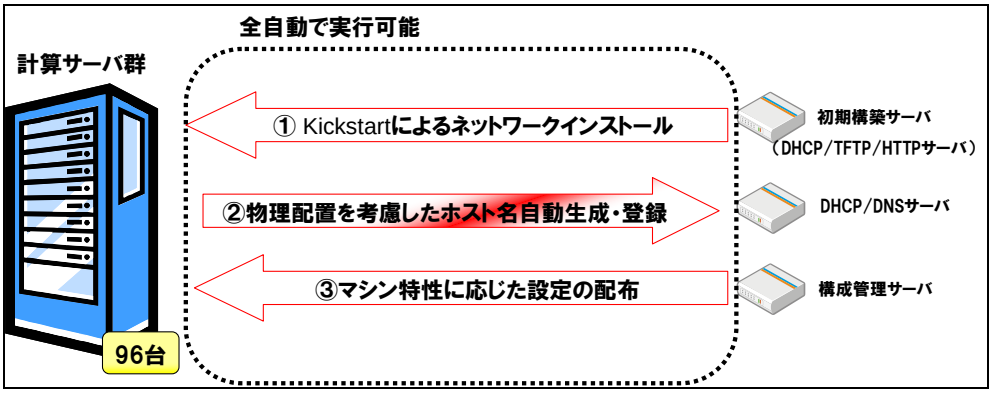


図 5-3 11 章で検討した自動構築化手法

5.3.1.2 実験結果

96 台のサーバの自動構築を実施した結果を以下に記載する。構築はおよそ 90 分で完了した。作業者の作業は対象サーバの電源を投入するだけであり、作業時間は 10 分程度であった。インストール後は、それぞれの Hadoop スレーブサーバが、処理に必要な各種アプリケーションがインストールされ、利用可能な状態になっていることを確認した。図 5-4 にインストール最中の CPU 使用率を掲載する。

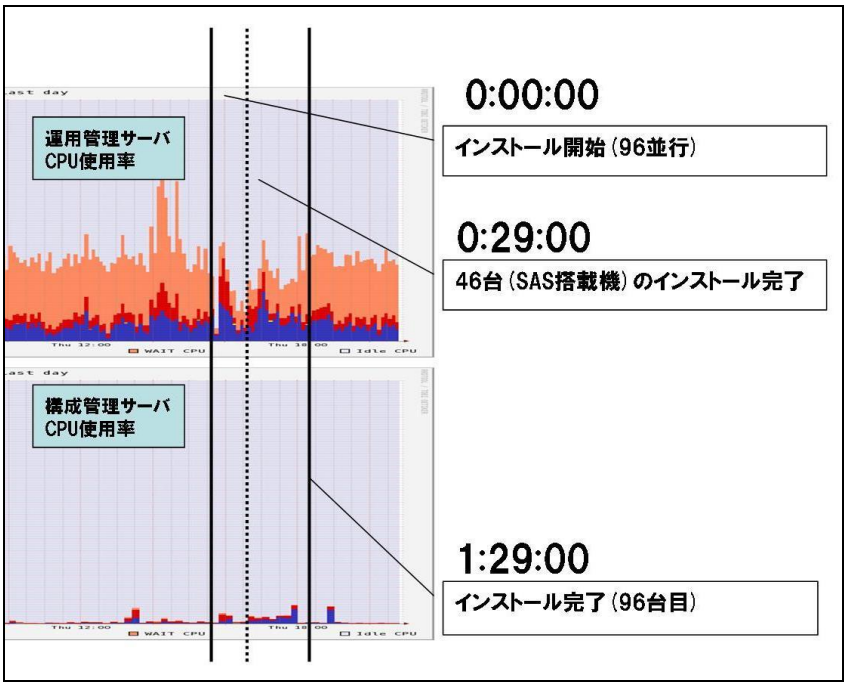


図 5-4 初期構築の自動化による CPU 使用率及びインストール時間

5.3.1.3スケーラビリティの評価

本方式によって 96 台のサーバの初期構築が行えることが実証され、有効に動作することを確認した。

また、本方式は作業者の作業時間が 10 分程度で構築が可能である。一般的に作業者が手順書に従って環境構築・インストールする時間は 4 時間程度であるため、380 時間の作業者の作業時間の効率化が可能であると概算できる。また、本方式では作業が自動化されているため、作業漏れや作業ミスに起因する作業の手戻りをしなくてもよいという利点がある。

運用監視サーバにおけるサーバの構築中は CPU 使用率が概ね 25%となっており、本方式では、現在の 4 倍程度の 400 台程度まで、構築対象の Hadoop スレーブサーバのインストール時間に影響することなく、対応が可能と考える。なお、HTTP サーバは負荷分散を実施することでスケールアウトすることも可能である。

一方、維持管理サーバにおける CPU 使用率は、構築中最大で 5%程度となっており、本方式では、現在の 20 倍程度の 2000 台程度まで、構築対象の Hadoop スレーブサーバのインストール時間に影響することなく、対応が可能と考える。維持管理サーバのスケール性に関しては、維持管理サービスを提供する Puppet はその通信の方式上、サーバクライアント間で認証鍵による暗号化通信を行うため、また、構成を一元管理するという観点から、複数台構成でスケールアウトしづらい構成である。

上記評価結果を表 5-4 にまとめる。

表 5-4 初期構築におけるスケーラビリティの評価結果

No.	観点	概要
1	有効性	10 分の作業時間で、多数の構築ができることを確認した。対象機器に対する手順も自動化により斉一に保たれ、有効な手法であることを確認した。
2	効率性	手動による構築時間を一台あたり 4 時間と想定すると、およそ 380 時間の作業工数の削減と試算する。
3	適用性	現行の方式により、90 分で 96 多重での同時構築ができることを確認した。HTTP サーバの CPU 使用率からサイジングを行うと、およそ 90 分で 400 多重程度のスループットで自動構築が可能であると考えられる。

上記検討により、本章における Hadoop 基盤を構築するために、本章で利用した構築手法が、十分なスケーラビリティを持つことを結論する。

5.3.1.4 初期構築におけるクラスタ構成要素の変更に関する評価

初期構築時においては対象機器が明確に決まっており、Hadoop スレーブサーバ数は増減しないため、評価対象としない。

5.3.1.5 初期構築における基盤の混在性に関する評価

初期構築において構築した 96 台の Hadoop スレーブサーバは表 5-5 のように機器・種別が混在している。これらの 5 種の異なる環境において、混在環境を意識せずに単一の手順で構築ができることを確認した。

表 5-5 構築対象の混合性

No.	ベンダ	CPU	メモリ	HDD
1	HP	Xeon QuadCore/2.33GHz x2	8GB	SAS 146GB x 2
2	HP	Xeon QuadCore/3.16G	6GB	SAS 146GB x 2
3	HP	Xeon DualCore/2.33G	2GB	SAS 72GB x 2
4	HP	Xeon QuadCore/2G	6GB	SAS300GB x 2
5	NEC	Core2 Duo T9400	2GB	SATA 250GB x 2

上述の結果より、ベンダ・CPU 種別・メモリ容量・ディスク本数・ディスクコントローラの異なる混在機器に対して、同一の作業手順で、機器に応じたセットアップが行われ、Hadoop 基盤の要素として構成されることを確認し、混在性に関して共通化された手法で構築することを確認した。

5.3.2 監視・故障検知における運用性実験

Hadoop 基盤を構築する機器のうち、システム規模に比例する機器である Hadoop スレーブサーバの構築の効率的な監視・故障検知・可視化を実施する。

5.3.2.1 監視・故障検知における実験のスコープ

システム規模に比例する構成要素である Hadoop スレーブサーバに関して、スケーラブルな方式で、監視・可視化・故障検知ができることを確認する。監視手法は、オープンソースソフトウェアの監視システムに加え、11 章で検討した Ganglia を監視システムの基盤としたスケーラブルな運用方式を採用する。また、システム規模に比例し、大量に存在するスレーブサーバの可視化に関しては、12 章で議論した Ganglia による可視化手法を採用する。概念図を図 5-5 に記載する。

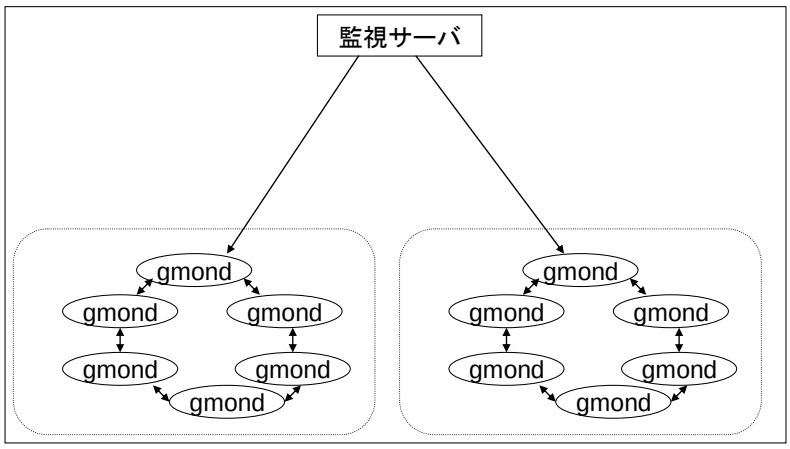


図 5-5 Ganglia・監視システム連携方式

5.3.2.2 実験結果

Ganglia により、Hadoop 基盤を特徴付ける各種パラメータ（ラック間転送量や、総 CPU 使用率など）を取得、可視化することで、Hadoop 基盤を全体として捉えるための情報を必要十分なだけ取得することができた。

図 5-6 は、本章におけるラック番号 4 の代表サーバ r4-1-0-01 に対して上記方式を適用し、ラック番号 4 に配置された全てのサーバのディスク容量監視や、プロセス監視を実現したものである。

r4-1-0-01.example.net	Group::Deadhost	OK	01-26-2010 10:53:32	0d 18h 26m 23s	1/3	OK: dead node: 0
	Group::Df::HDFSA	OK	01-26-2010 10:52:44	0d 18h 17m 11s	1/3	OK: scanned 16 hosts. r4-1-0-14.example.net diskusage: 6%
	Group::Df::Log	OK	01-26-2010 10:52:39	0d 18h 57m 16s	1/3	OK: scanned 16 hosts. r4-1-0-08.example.net diskusage: 7%
	Group::Df::Root	OK	01-26-2010 10:52:39	0d 18h 57m 16s	1/3	OK: scanned 16 hosts. r4-1-0-01.example.net diskusage: 39%
	Group::Proc::Datanode	OK	01-26-2010 10:57:21	0d 0h 52m 34s	1/3	OK: scanned 16 hosts. r4-1-0-01.example.net procnum: 1
	Group::Proc::Tasktracker	OK	01-26-2010 10:57:59	0d 0h 51m 56s	1/3	OK: scanned 16 hosts. r4-1-0-01.example.net procnum: 1
	Service::PING	OK	01-26-2010 10:50:20	3d 20h 49m 42s	1/3	PING OK - Packet loss = 0%, RTA = 0.37 ms

図 5-6 監視サーバと Ganglia 連携方式の実装

ラック番号 4 のいずれかのサーバ故障時においては、「図 5-7 監視サーバと Ganglia 連携方式による故障検知」のように監視結果が得られることを確認した。

r4-1-0-01.example.net	Group::Deadhost	WARNING	01-19-2010 11:10:50	0d 0h 10m 27s	3/3	WARN: dead node: 1
	Group::Df::HDFSA	OK	01-19-2010 11:08:51	4d 2h 17m 1s	1/3	OK: scanned 16 hosts. r4-1-0-13.example.net diskusage: 8%
	Group::Df::Log	OK	01-19-2010 11:16:36	4d 2h 11m 4s	1/3	OK: scanned 16 hosts. r4-1-0-01.example.net diskusage: 3%
	Group::Df::Root	OK	01-19-2010 11:16:36	4d 2h 11m 4s	1/3	OK: scanned 16 hosts. r4-1-0-01.example.net diskusage: 39%
	Service::PING	OK	01-19-2010 11:08:13	4d 2h 19m 4s	1/3	PING OK - Packet loss = 0%, RTA = 0.19 ms

図 5-7 監視サーバと Ganglia 連携方式による故障検知

図 5-8 では、Hadoop 基盤全体の可視化を Ganglia で実現した例を記載する。

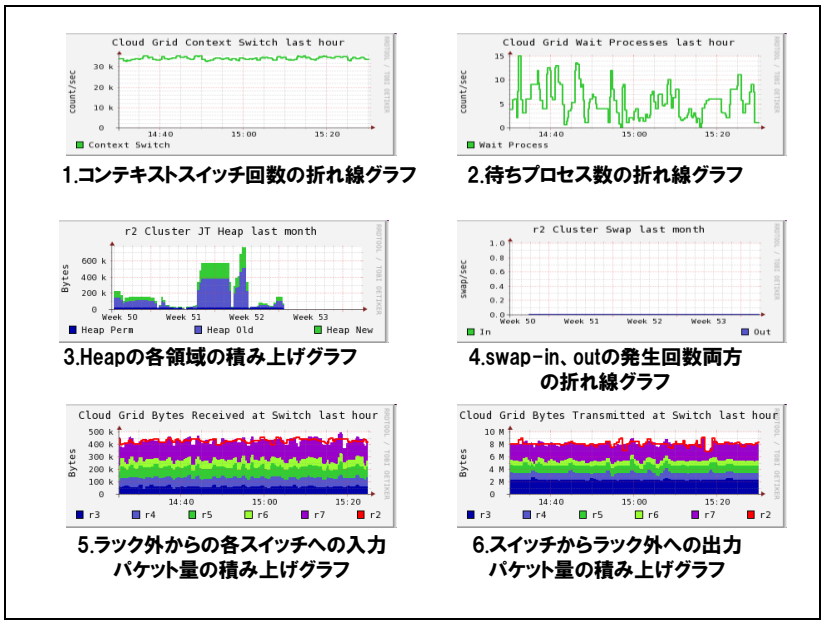


図 5-8 Ganglia による Hadoop 基盤における特徴量の可視化

5.3.2.3スケーラビリティの評価

本方式によって 96 台のサーバの故障検知・可視化・監視が効率的に行えることが実証され、有効に動作することを確認した。

また、監視・可視化に関しては、作業者は Web 画面で構成された監視コンソールを閲覧するだけで可能である。また、可視化に関しては、集計機能により、一つの画面にて全てのサーバのリソース情報を集約した形で直感的に把握することが可能であり、1 人の作業者において Hadoop 基盤の監視・可視化ができると結論づけた。

100 台の Hadoop 基盤の監視・可視化に関しては、監視・可視化サーバである運用監視サーバにおける CPU 使用率が概ね 35%となっており、外挿する方式では、おおよそ 300 台程度まで本方式において対応が可能と考える。

なお、本章では、監視サーバと可視化サーバは同居しているが、これらは容易に分離可能である。監視サーバ単体における CPU 使用率は、およそ 10%程度であり、監視サービス自体では、現在の 10 倍の 1000 台程度まで対応可能である。

一方可視化サーバに関しては、1 サーバ追加するごとに約 0.25%”WAIT CPU”が増加し、”WAIT CPU”以外に平均して 10%程度 CPU を利用していること考慮すると、90 / 0.25 = 360 台程度で CPU の Idle が枯渇し、ボトルネックを迎えると推定できる。詳細な考察は 13 章に記載する。

表 5-6 監視・故障検知におけるスケーラビリティの評価

No.	観点	概要
1	有効性	故障検知・監視・可視化が行えることを確認した。
2	効率性	Web ベースのコンソールにて一目で異常を確認でき、また、系全体のリソース使用量が集約化され一目で系のリソース使用量が可視化されており、1 人で監視が可能であることを確認した。
3	適用性	100 台のサーバに対して、1 台の運用監視サーバにおいて監視・可視化が行われることを確認した。現行の運用監視サーバの CPU 使用率から推測すると、300 台の監視・可視化まで可能であると判断する。また、監視サーバと可視化サーバを別のサーバで提供することも可能である。

5.3.2.4クラスタ構成要素の変更に関する評価

Hadoop スレーブサーバの構成変更に関しては、監視・可視化サーバの設定を変更するだけで対応が可能であることを確認した。また、Hadoop スレーブサーバの設定は自動構築手順により、一切変更する必要がない。

規模に比例する構成要素である Hadoop 計算サーバの設定変更は必要ないことを確認し、監視・可視化サーバの設定ファイル（3 ファイル）の設定変更のみで可能であ

ることを確認した。

5.3.2.5 基盤の混在性に関する評価

監視・可視化に関しては基盤の混在性はアプリケーションの設定ですべて完全に設定することが可能である。アプリケーションの設定を行うだけで、全てのサーバを同一の作業手順で監視・可視化を行うことができることを確認した。

5.3.3 維持管理・メンテナンスにおける運用性実験

Hadoop 基盤を構築する Hadoop スレーブサーバ・Hadoop マスターサーバの維持管理・メンテナンスに関する運用を評価する。

5.3.3.1 維持管理・メンテナンスにおける実験のスコープ

維持管理・メンテナンスにおけるスケーラビリティ上の問題点は、大量に存在し、かつスペックが混在する Hadoop スレーブサーバのソフトウェアの構成管理である。そこで、11 章で検討した、オープンソースソフトウェアの構成管理ソフトウェアである Puppet を用いた構成管理手法を用いて、大量のサーバに意図通りの設定がなされていることを保障するような仕組み（図 5-9 参照）が有効に確認できることを確かめた。

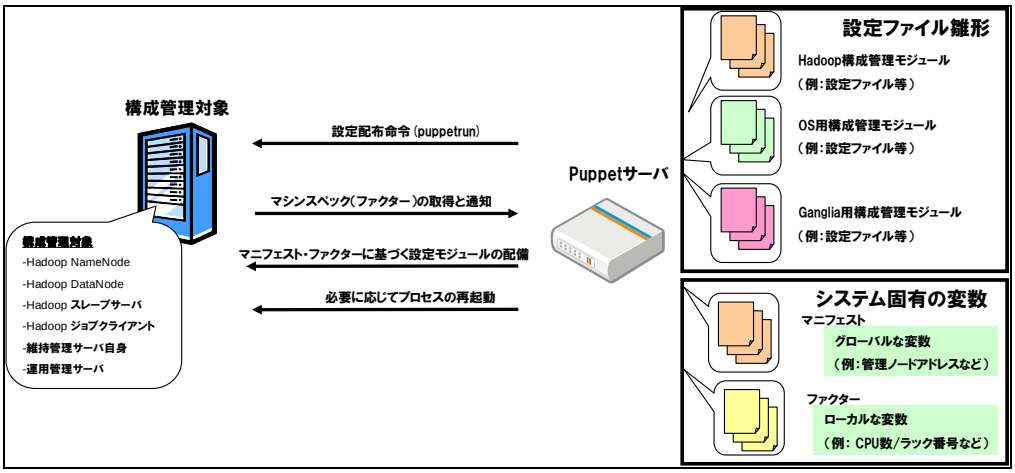


図 5-9 Puppet による構成の担保の実現方式

5.3.3.2 実験結果

Puppet を利用することで、Hadoop 基盤を構成する全てのサーバに対する構成管理を維持管理サーバ 1 台で管理することを確認した。特に、スペック混在環境における Hadoop 基盤構成に関しても、Puppet の facter 拡張を実装することで、スペックが異なる Hadoop 基盤に関しても、マシンごとに異なる設定ファイルを単一の操作で配

布できることを確認した。

5.3.3.3スケーラビリティの評価

本方式によって Hadoop 基盤を構成する全てのサーバに対する構成管理が効率的に行えることが実証され、有効に動作することを確認した。

設定の配布・操作に関しては、あらかじめ用意された、Hadoop スレーブサーバ共通のテンプレートを、維持管理サーバから単一のコマンドで操作するだけであり、およそ 3 分で設定の配布が完了する。従って、1 人の作業者において Hadoop 基盤の構成管理ができると結論づけた。

なお、配布に当たっては 1 サーバあたり、維持管理サーバとの通信に 10 秒前後の配布時間がかかり、その前後では、維持管理サーバの CPU コア 1 つを占有する。配布自体は多重度をかけることが可能であり、本実験では最大 10 多重で設定の配布を行った。上記を表 5-7 にまとめる。

表 5-7 維持管理・メンテナンスにおけるスケーラビリティの評価

No.	観点	概要
1	有効性	100 台のサーバの構成管理（設定ファイル・バイナリ配布）が行えることを確認した。
2	効率性	単一のコマンドにより一人の作業者で実施することが可能であることを確認した。
3	適用性	100 台の構成反映におよそ、3 分程度の処理時間が必要である。それを反映し、メンテナンス時間に上記時間が収まる程度まで適用可能である。

5.3.3.4クラスタ構成要素の変更に関する評価

Hadoop スレーブサーバの構成変更に関しては、維持管理サーバの設定を変更するだけで対応が可能である。また、テンプレートが共通化されているため、維持管理サーバ側では、構成ファイルにサーバ名を追記するだけでよいことを確認した。Hadoop スレーブサーバの設定は自動構築手順により、変更が不要であることを確かめた。

5.3.3.5基盤の混在性に関する評価

規模に比例した構成要素である、96 台の Hadoop スレーブサーバは前述の「表 5-5 構築対象の混合性」の通り、機器・種別が混在している。混在環境を意識せずに単一の配布手順で構成管理、操作ができることを確認した。これはマシン固有情報を動的に収集する Puppet の facter 拡張を利用したためである。

上記をもって、環境の混在性が、維持管理管理サーバの変数値として隠蔽され、作業者は同一の作業手順で、混在環境における設定変更・配布が可能であることを確認した。

5.3.4 回復・増設における運用性実験

Hadoop スレーブサーバの故障からの回復及び、増設を行う。

5.3.4.1 回復・増設における実験のスコープ

ラック単位での増設を行う。3 ラックの増設で、増設サーバ数は 70 台である。また、一台の故障からの回復を行う。

5.3.4.2 実験結果

図 5-10 に増設時における、インストール時間の分布をまとめる。

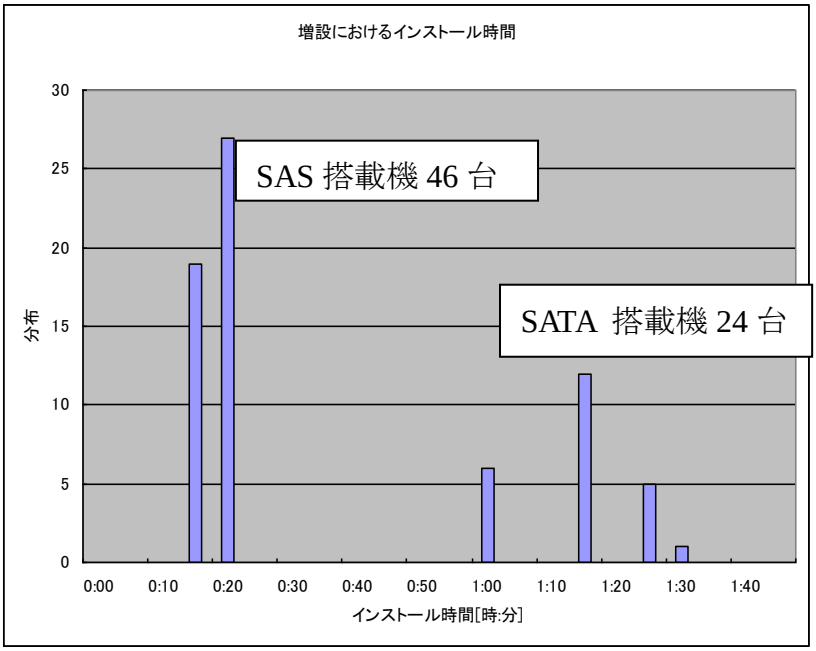


図 5-10 増設におけるインストール時間の分布

大きく 2 種類の分布となる。これは Hadoop 計算サーバの内蔵ディスク性能を反映したものである。増設中の運用系サーバの CPU 使用率を図 5-11 に掲載する。

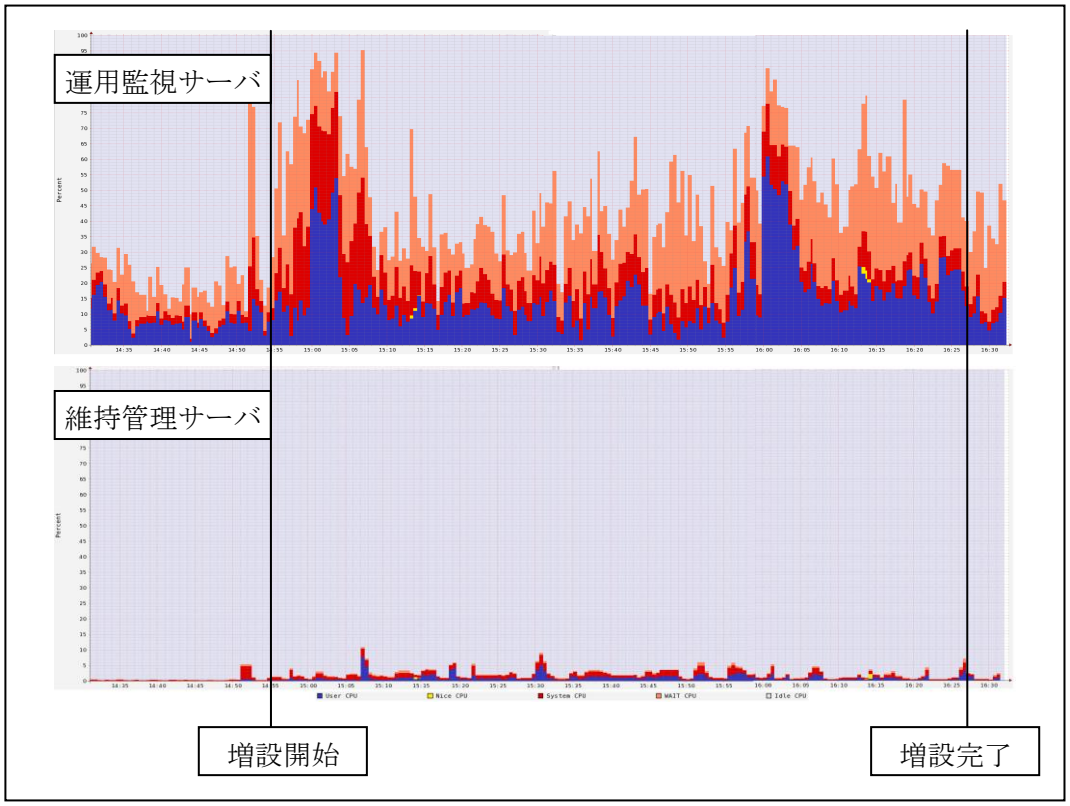


図 5-11 運用サーバの CPU 使用率

運用監視サーバの二つのピークは、定期的な統計情報のバックアップのための負荷である。

実証実験を実施する上で、以下の通りハードウェア・ソフトウェアの故障が発生した。11 章の手順に従って回復を実施した。

表 5-8 実証実験におけるサーバ故障事例一覧

No.	日付	故障概要	対応
1	11/27	HDD を認識しない。RAID カードの BIOS では本来二つ見えるはずの PhysicalVolume が一つしか見えない。	OS の再起動でも起動が不可。ベンダに連絡し、12/1 に RAID カードの交換を実施。再構築。
2	12/13	HDD を認識しない。RAID カードの BIOS では本来二つ見えるはずの PhysicalVolume が一つしか見えない。	OS の再起動でも起動が不可。ベンダに連絡し、12/15 HDD・HDD 接続ケーブルの交換。再構築。
3	1/ 26	サーバが応答しない。	swap 領域が壊れており、以下のメッセージがコンソールに出ていた。

No.	日付	故障概要	対応
			end_request I/O error dev, sda, sector NNNNN OS を再起動した際に、fsck が走り正常に終了することを確認。
4	1/ 28	サーバが応答しない。	サーバが応答しない。OS の再起動で正常にクラスタに組み込まれることを確認した。

5.3.4.3スケーラビリティの評価

故障からの復旧および増設に関しては、自動構築手順により、作業者は共通化された構築手法に従い、効率的に行えることを確認した。

適応性に関しては、作業者の作業時間に依存する。復旧に関しては、作業者が故障対象一台あたりに対応する作業時間は概ね 10 分程度であり、その他の作業は、同時並行的に行うことができる。従って 1 度の故障からの復旧で 5 台程度の復旧をすると想定すると、復旧は 2 時間程度と想定される。増設に関しては、初期構築と同様、70 台の構築を 90 分程度で行えることを確認した。

表 5-9 回復・増設におけるスケーラビリティの評価

No.	観点	概要
1	有効性	70 台のサーバの増設・1 台のサーバの故障回復を確認した。
2	効率性	復旧における作業時間は概ね 40 分程度である。作業時間のほとんど（30 分程度）は、自動構築時間であるため、大幅に削減できている。
3	適用性	復旧のためのメンテナンス時間に依存する。5 台程度の故障からの復旧が 2 時間であるため、メンテナンス時間を結果に応じて決定する必要がある。

5.3.4.4クラスタ構成要素の変更に関する評価

表 5-10 に増設時の運用手順を、表 5-11 に故障回復時における運用手順を記載する。

表 5-10 増設時の手順

No..	作業内容	対象	作業時間	総時間
1	初期構築と同様に構築を実施。構築最中に自動で構成管理サーバと通信を行うため、初期構築後の基盤への設定変更は全て反映済み	70 台	10 分	91 分

No..	作業内容	対象	作業時間	総時間
2	マスタサーバで HDFS のリバランス処理を実施する。	1 台	5 分	5 分

表 5-11 故障からの回復手順

No.	作業内容	作業時間	総時間
1	サーバにログインを試みるもログイン不可。サーバ故障と判断し、OS の再インストールを決定	5 分	5 分
2	故障回復手順にのっとり、維持管理サーバの該当サーバの証明書を削除	1 分	1 分
3	交換した代替機の電源を投入し、OS のインストールを開始	1 分	32 分
4	自動インストールが完了し、Ganglia, 監視システムにおいて問題ないことを確認	2 分	1 分

増設および、故障検知・復旧作業手順において、総時間のほとんどを占める OS の再インストール（復旧）に関しては、前述の初期構築・増設と全く同じ手順で実施可能であることを確認した。これは、初期構築から故障検知・復旧にいたる運用の中において、設定変更など Hadoop スレーブサーバへの変更は全て構成管理サーバ経由で行うという維持管理手順に従ったためである。

上記結果により、故障からの復旧・増設のいずれも、作業時間のほとんどを占める構築作業に関して、初期構築と同様の手順で実施可能であることを確認した。

5.3.4.5 基盤の混在性に関する評価

Hadoop スレーブサーバは前述の「表 5-5 構築対象の混合性」の通り、機器・種別が混在している。ハードウェア故障からの回復や、新規インストールである増設では、初期構築と同じ手法をとることができることを確認した。また、初期構築以降の構成変更、例えば設定ファイルの変更やライブラリのアップデートなども、Puppet を利用することで全て反映することが出来る。

上記結果により、作業時間のほとんどを占める構築作業に関して、初期構築・維持管理と同様、作業者は同一の作業手順で、混在環境においても同一の手順で構築が可能であることを確認した。

5.4 まとめと今後の課題

本章では、Hadoop 基盤の特徴を３つの観点で整理するとともに、各運用項目に関

して、本章で採用した運用方式が、有効であることを確認した。本節では、それらの結果を整理しなおすとともに、今後の課題を抽出する。

5.4.1 運用性実験のまとめ

本章では規模に比例する構成要素である Hadoop スレーブサーバに関して、「スケーラビリティ」、「クラスタ構成要素の変更への対応」、「基盤の混在性」を評価の軸とし、各運用項目に関してその妥当性を評価した。表 5-12 に詳細を記載する。

総論として、各運用項目に関して、実証実験における Hadoop 基盤を効率的に運用するための手法が「スケーラビリティ」、「クラスタ構成要素の変更」、「混在性」の評価軸において有効に動作することを確認した。

表 5-12 各運用項目の評価結果概要

No.	運用項目	評価軸	評価観点	確認結果
1	初期構築	スケーラビリティ	有効性	96 台の Hadoop スレーブサーバを適切に構築できることを確認した。
2			効率性	手動による想定構築時間と比較し、およそ 380 時間の作業時間の削減と試算する。
3			適用性	90 分で 96 多重での同時構築を確認した。
4		クラスタ構成要素の変更	共通性	初期構築では該当なし。
5		基盤の混在性	共通性	ベンダや種別の異なる機器においても適用可能であることを確認した。
6	監視・故障検知	スケーラビリティ	有効性	故障検知・監視・可視化が適切に行えることを確認した。
7			効率性	系全体のリソース使用量が集約化され、一目で監視が可能であることを確認した。
8			適用性	100 台のサーバに対して、1 台の運用監視サーバにおいて監視・可視化が行われることを確認した。
9		クラスタ構成要素の変更	共通性	Hadoop スレーブサーバの設定は自動構築手順により、一切変更する必要がないことを確認した。

No.	運用項目	評価軸	評価観点	確認結果
10		基盤の混在性	共通性	基盤の混在性はアプリケーションの設定ですべて完全に設定することが可能であることを確認した。
11	維持管理・メンテナンス	スケーラビリティ	有効性	100 台のサーバの構成管理（設定ファイル・バイナリ配布）が行えることを確認した。
12			効率性	単一のコマンドにより実施が可能であることを確認した。
13			適用性	100 台の構成反映におよそ、3 分程度の処理時間が必要であることを確認した。
14		クラスタ構成要素の変更	共通性	テンプレートが共通化されているため、維持管理サーバ側でわずかな変更を行うだけでよいことを確認した。
15		基盤の混在性	共通性	Puppet の facter 拡張を利用することで、混在環境を意識せずに単一の配布手順で行えることを確認した。
16	回復・増設	スケーラビリティ	有効性	70 台のサーバの増設・1 台のサーバの故障回復ができることを確認した。
17			効率性	復旧・増設における作業時間の大部分が、自動構築時間であることを確認した。
18			適用性	5 台の故障からの復旧が 2 時間で行えることを確認した。
19		クラスタ構成要素の変更	共通性	維持管理手順に従ったため、故障からの回復及び増設で初期構築における自動構築手法が利用できることを確認した。
20		基盤の混在性	共通性	初期構築と同等の手順が適用できるため、ベンダや種別の異なる機器においても適用可能であることを確認した。

5.4.2 今後の課題

クラウド型分散基盤はデータセンタをまたがった効率的な運用方式等に関しては、今回の実証検証の対象外であった。運用するべきクラウド型分散基盤が、複数のデータセンタをまたがっている場合や、運用者が物理的に非常に離れている際の運用手順及び効率的な運用方式に関して、本実証実験での検討結果が妥当性評価を行うことは意義があると考える。

6 実証実験

本章では、5 章までに検討した「Hadoop を用いて構築したクラウド型分散処理基盤」および「MapReduce を適用した渋滞解析アプリケーション」を、より実運用に近いシナリオに基づいて動作させ、クラウド型分散処理基盤の有用性を検証する。

6.1 実施概要

実証実験の目的と実施内容の概要を説明する。

6.1.1 実証実験の目的

2 章から 5 章では、MapReduce を適用したアプリケーションの開発、Hadoop 基盤上でのアプリケーションの動作特性、Hadoop により構築したクラウド基盤に対する可用性、効率的な運用手法など、クラウド型分散処理基盤の実利用に必要となる主要な技術トピックについて個別に検討、評価を実施し、クラウド型分散処理基盤の実現に向けた技術実証と、課題の抽出を行ってきた。

これらの検討結果を組み合わせた上で、現実の大規模プローブ情報に対して実証実験を行うことで、より現実に近い条件においても各手法が有効であることを、アプリケーションの動作特性、クラウド基盤の信頼性向上、クラウド基盤の運用効率化の 3 つの観点で確認することを目的としている。

アプリケーションの動作特性に関する実証実験

2 章、3 章では分散処理のケーススタディとして渋滞解析アプリケーションを用いて、データ量やサーバ台数に対するスケーラビリティ、処理時間と処理精度のバランスなどの特性の分析、評価を行ってきた。

本実証実験では、処理時間と処理精度のバランス等の特性が実運用を意識したデータ量・動作シナリオにおいても同様に適用可能であることを実証する。データ量増、サーバ増強、処理精度の向上といったシステムの長期的な発展を意識したシナリオに従いアプリケーションを動作させる。その際、データ量増とサービス拡充による計算量増大に対してサーバ台数の増強で対応可能となる、シナリオ実現に必要なスケーラビリティを有していることを確認する。また、データ量が非常に多い場合の処理として、1 年分のプローブデータの渋滞統計処理を行う。

クラウド基盤の信頼性向上に関する実証

4 章では Hadoop 基盤における可用性を担保する技術について基盤を構成する要素ごとに方式を検討し、マスタサーバの可用性確保のためにソフトウェア FT 技術である Kemari の適用検証を行ってきた。

本実証実験では、実際にアプリケーションが動作している状態で Hadoop 基盤の構

成要素であるスレーブサーバ(1 台単位、ラック単位)・Kemari で冗長化したマスタサーバ等に対して故障を発生させ、実行中のジョブや後続ジョブへの影響がないことを確認する。

クラウド基盤の運用効率化に関する実証

5 章では Hadoop 基盤における運用項目を、「初期構築」、「監視・故障検知」、「維持管理・メンテナンス」、「回復・増設」に分類した上で、各運用項目についてスケーラビリティを考慮した運用方式の検討と実証を行ってきた。

本実証実験では、システムの長期的な発展を意識したシナリオに従い「初期構築と増設」、「故障検知と復旧」を実施して、アプリケーションの動作に悪影響を及ぼさず効率的な運用が行えることを確認する。

6.1.2 実証実験のシナリオ

実証実験のシナリオとして、「処理時間と処理精度に関する実証実験のシナリオ」と「大規模データを使った処理のシナリオ」の 2 つを使用する。

6.1.2.1 処理時間と処理精度に関する実証実験のシナリオ

プローブ情報の提供元から随時クラウド環境に到着するプローブ情報に対して、5 分おきに短時間渋滞情報解析アプリケーションを起動して直近 5 分間の渋滞状況を解析する処理を行う。毎時 0 分、5 分、10 分、……、55 分にアプリケーションを起動する。処理対象データはアプリケーション起動時の直近 5 分間のデータを対象とする。アプリケーションの実行時間は 5 分間で処理が終了することを目指して実証実験を実施する。実証実験で使用するプローブ情報、アプリケーションの処理内容の詳細は 2 章を参照のこと。

実際のプローブ情報のデータ量を処理するシステム規模から始まり、システムの成長に伴いデータ量・サーバ台数が増加していくシナリオを想定して、サーバ台数・データ量・処理精度を変化させる。サーバ台数・データ量・処理精度の変化の順序を「表 6-1 処理時間と処理精度に関する実証実験のシナリオ」に示す。

表 6-1 処理時間と処理精度に関する実証実験のシナリオ

シナリオ No.	想定シナリオ	サーバ台数	データ量	処理精度（道路種別）	処理精度(道路区間)	実施内容
1	小規模構成によるサービススタート	3 台	約 22 万件(端末数:約 730 台)	主要道路	標準	実データを使い、小規模構成でデータ解析を行う。
2	データ量の増大とサーバ増強	25 台	約 900 万件(端末数:約 30000 台)	主要道路	標準	プローブ数の増加などにより入力データが増えた場合の挙動を確認する。
3	対象道路の拡大とサーバ増強	93 台	約 900 万件(端末数:約 30000 台)	一般道路	詳細	対象とする道路の範囲を広げた場合の挙動を確認する。

(注) 端末数はデータ数より概算した同時接続しているプローブ発信端末の数を示す。

なお、クラウド基盤の信頼性向上に関する実証実験は「3 サービス内容の向上とサーバ増強」でサーバ 100 台に増強した後の状態で実施を行う。また、クラウド基盤の運用効率化に関する実証実験は「2 データ量の増大とサーバ増強」「3 サービス内容の向上とサーバ増強」でのサーバ増強時に実施する。

6.1.2.2大規模データを使った処理のシナリオ

過去 1 年分のプローブ情報を解析して、季節、月、曜日、特異日等の単位で渋滞状況の集計処理を実施する。「6.1.2.1 処理時間と処理精度に関する実証実験のシナリオ」とは異なり定期実行は行わず、大量データを一度に長時間(数 10 時間程度)で処理を行う実証実験とする。実証実験で使用するプローブ情報、アプリケーションの処理内容の詳細は 2 章を参照のこと。

渋滞統計処理アプリケーションを用いた実証実験のシナリオを「表 6-2 大規模データを使った処理に関する実証実験シナリオ」に示す。

表 6-2 大規模データを使った処理に関する実証実験シナリオ

シナリオ No.	想定シナリオ	サーバ台数	データ量	処理精度（道路種別）	処理精度(道路区間)	実施内容
1	大規模データの実行	93 台	約 2.1TB（約 140 億件、1 年分相当）	詳細	一般道路	1 年分のデータで統計処理を実行し、処理時間をシナリオ 1 で見積もった時間と比較する。

6.2 実証実験環境

2 章から 5 章までの検討を踏まえて構築した実証実験環境の全体構成を図 6-1 に示す。

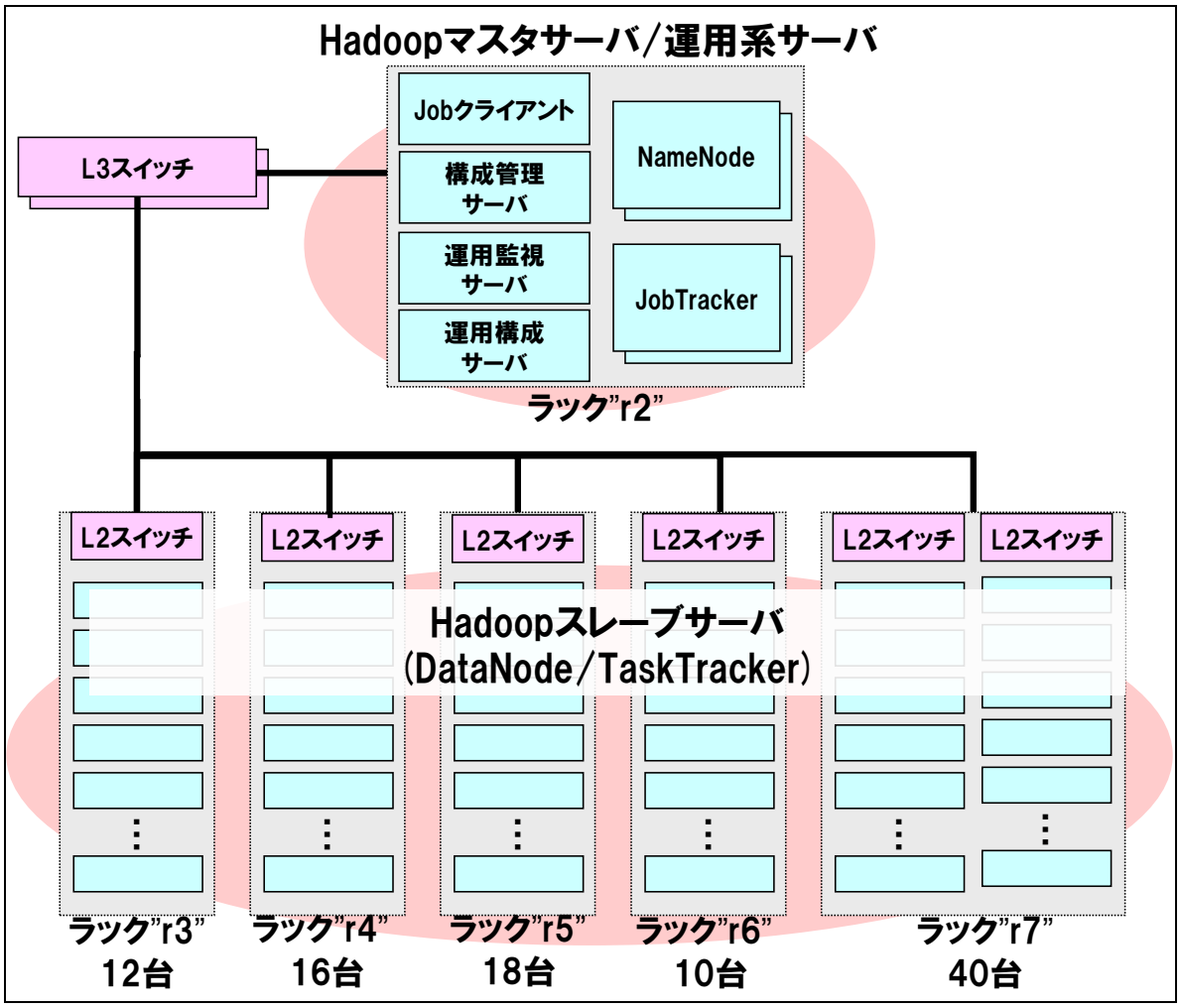


図 6-1 クラウド型分散処理基盤のシステム構成図

6.3 MapReduce アプリケーションの動作特性に関する実証実験

本章では、2 章で設計、実装した渋滞解析アプリケーションを「表 6-1 処理時間と処理精度に関する実証実験のシナリオ」、「表 6-2 大規模データを使った処理に関する実証実験シナリオ」に従い動作させ、アプリケーションの動作特性に関する実証を行う。

「処理時間と処理精度に関する実証実験」においては、実行プローブデータ量の増加や対象道路の拡大といった処理精度の上昇にともなう計算量の増加に対しても、サーバ台数の増強によるスケーラビリティでバランスが保てることを確認する。

「大規模データを使った処理に関する実証実験」においては、数テラバイトの大規模プローブデータを用い渋滞統計生成処理を実行し、大規模データに対して渋滞解析が行えることを確認する。

6.3.1 処理時間と処理精度に関する実証実験

本項では、「表 6-1 処理時間と処理精度に関する実証実験のシナリオ」に従い入力データ量の増大や、対象道路の拡大といった処理精度を上昇させていくシナリオにおいて、サーバ台数の増強によるスケーラビリティで、処理時間と処理精度のバランスを保てることを確認する。短時間渋滞情報生成処理で実行するシナリオを表 6-3 に再掲する。

表 6-3 処理時間と処理精度に関する実証実験シナリオ（再掲）

シナリオ No.	想定シナリオ	サーバ台数	データ量	処理精度（道路種別）	処理精度(道路区間)	実施内容
1	小規模構成によるサービススタート	3 台	約 22 万件(端末数:約 730 台)	主要道路	標準	実データを使い、小規模構成でデータ解析を行う。
2	データ量の増大とサーバ増強	25 台	約 900 万件(端末数:約 30000 台)	主要道路	標準	プローブ数の増加などにより入力データが増えた場合の挙動を確認する。
3	対象道路の拡大とサーバ増強	93 台	約 900 万件(端末数:約 30000 台)	一般道路	詳細	対象とする道路の範囲を広げた場合の挙動を確認する。

(注) 端末数はデータ数より概算した同時接続しているプローブ発信端末の数を示す。

6.3.1.1小規模構成によるサービススタート

実証実験の起点として、小規模構成(サーバ数３台)で、実際に取得した５分間のプローブデータを使用し、シナリオの目標値である５分以内で処理が完了することを確認する。実行した際の処理時間を表 6-4 に示す。

表 6-4 シナリオ１の実行時間

No.	構成	サーバ台数	処理時間（秒）
1	シナリオ１の構成	3	122

(注)　処理時間については実行した中で最大の時間を掲載している。

目標値(300 秒)に対して、処理時間が 122 秒となっており、シナリオの目標値である５分以内での処理を達成できることがわかった。

なお、作成した渋滞情報を可視化すると図 6-2 のとおりとなる。

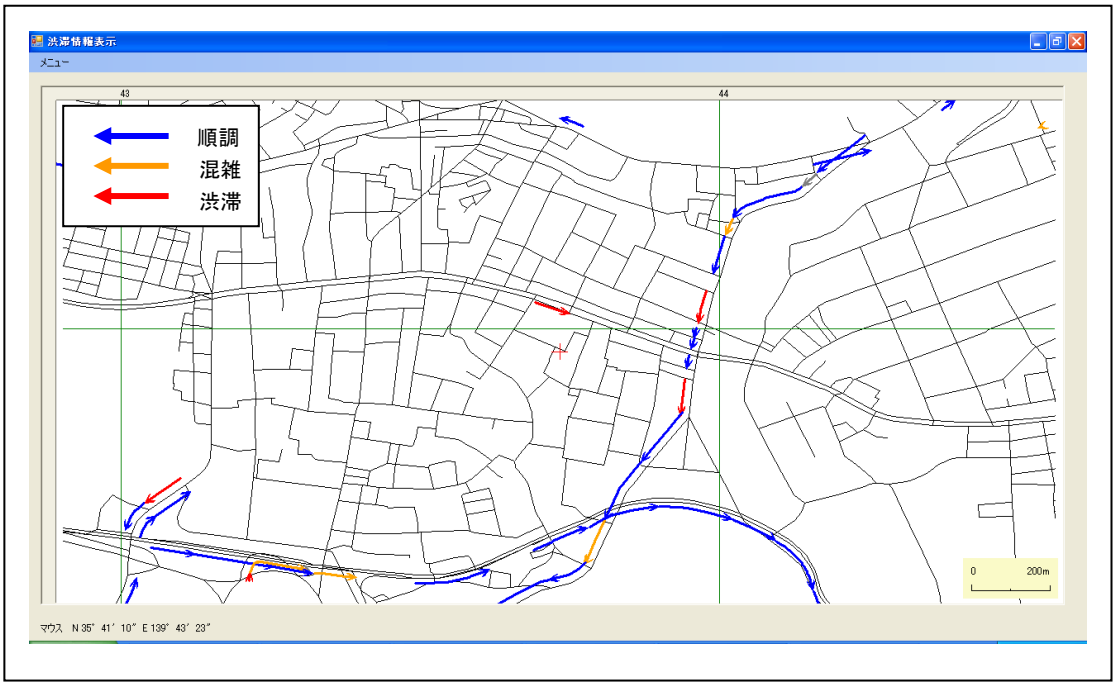


図 6-2 実データの可視化

6.3.1.2データ量の増大とサーバ増強

将来的にプローブデータを発信する車両、端末の増加によりプローブデータ量が拡大することを想定し、プローブ件数を約 900 万件に増加させ、次のことを確認する。

- データ量の増大への対応
  ３台構成(シナリオ１の構成)では対応できないような大幅なデータ量の増大に対して、サーバ台数でスケールさせることで対応できることを確認する。
- 精度への影響
  データ量が増大することで、作成される渋滞情報がどのように変化するか確認する。

データ量増大への対応

小規模構成（シナリオ１の構成）でプローブデータを増加させた場合と、データ増加へ対応し、サーバ台数を 25 台に増加させた構成(シナリオ２の構成)で実行した結果を表 6-5 に示す。

表 6-5 シナリオ 2 の実行時間

No.	実行時の構成	サーバ台数	処理時間（秒）
1	シナリオ１の構成	3	1,223
2	シナリオ２の構成	25	240

(注)　処理時間については実行した中で最大の時間を掲載している。

シナリオ１の小規模構成では、1223 秒となっており、シナリオの目標値である５分の処理時間を大幅にオーバーしたが、サーバ台数を増加させることで処理時間を 240 秒に短縮できた。データ量の増大に対して、サーバ台数を増加させることで対応できることが確認できた。

精度への影響

シナリオ１とシナリオ２のプローブ数を処理した際の渋滞解析結果のデータ量をまとめたものを表 6-6 に示す。

表 6-6 プローブ数増加による渋滞情報の件数

No.	プローブ数(件)	渋滞情報数(件)
1	約 22 万(シナリオ１のデータ量)	12,215
2	約 900 万(シナリオ２のデータ量)	295,745

入力となるプローブデータが増加したことで、渋滞情報数が約 24 倍に増加した。これまでプローブデータ不足のため、渋滞情報を作成できていなかった道路に対して

も渋滞情報が作成でき、より高精度の渋滞情報が作成できることが確認できた。

なお、作成した短時間渋滞情報を可視化すると、図 6-3、図 6-4 の通りとなる。新宿付近に限定してみても渋滞情報がある道路が増加しており、より高精度の渋滞情報が作成できていること確認できる。

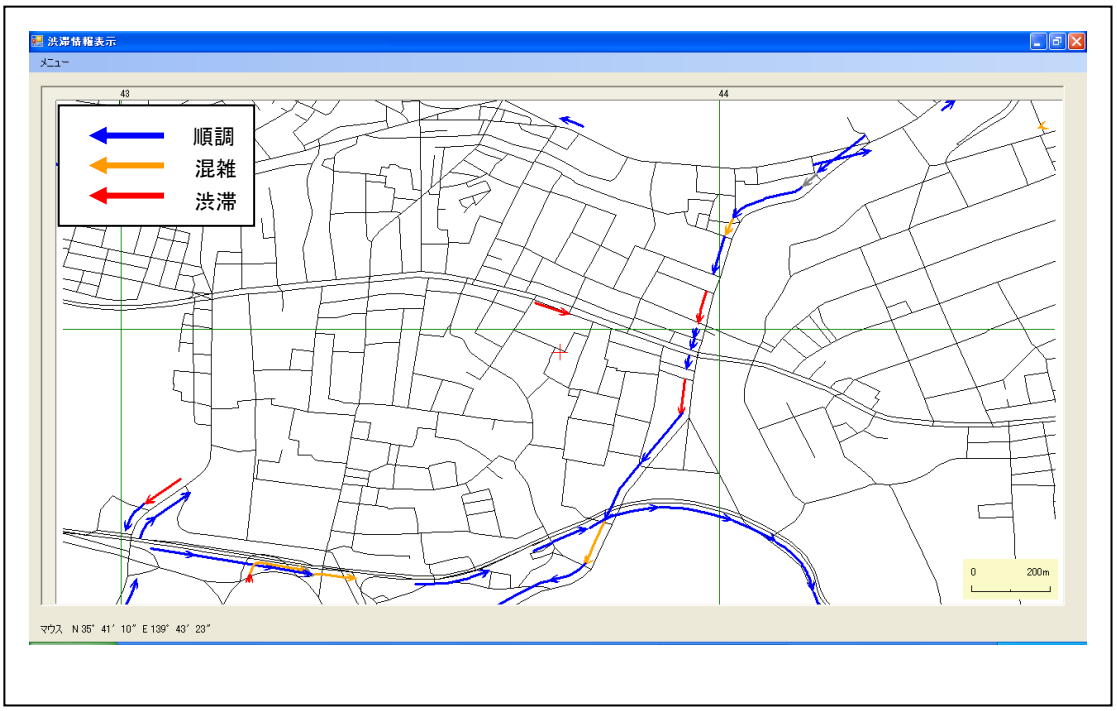


図 6-3 シナリオ 1 の渋滞情報の可視化

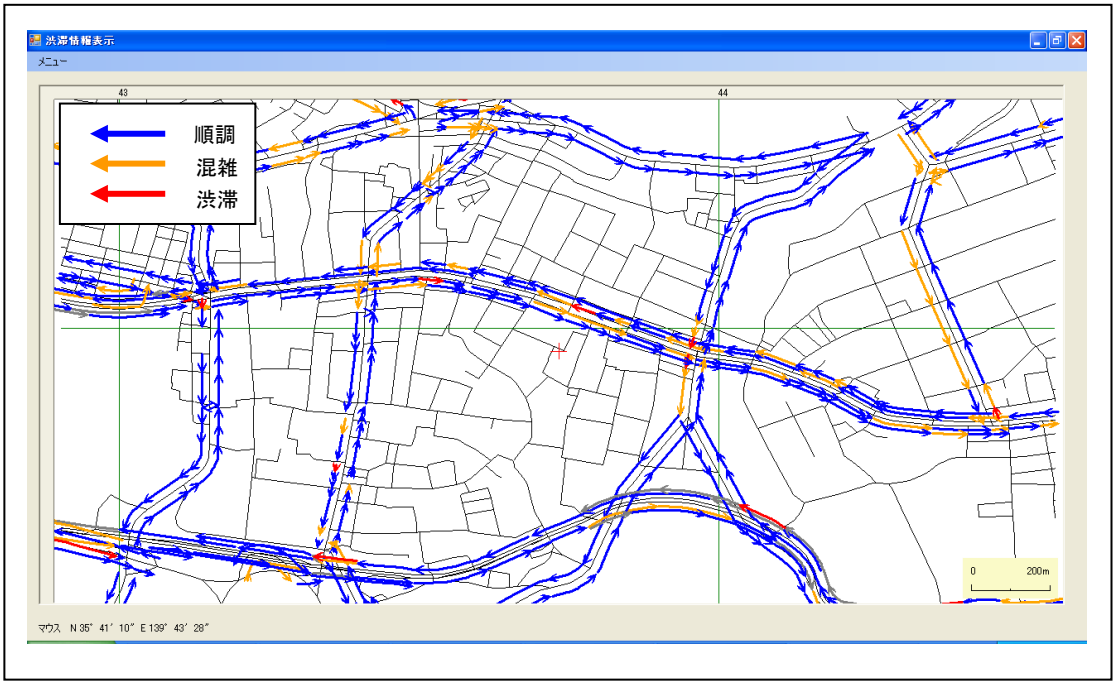


図 6-4 シナリオ 2 の渋滞情報の可視化

6.3.1.3対象道路の拡大とサーバ増強

渋滞情報の精度向上のため、渋滞解析アプリケーションの「道路種別を指定する機能」を「一般道路」にし、「道路区間距離を指定する機能」を「詳細」に変更することで処理対象となる道路を拡大して、次のことを確認する。

- 処理量増加への対応
  既存構成では対応できないような処理の増加に対して、サーバ台数でスケールアウトさせ、対応できることを確認する。
- 精度への影響
  「道路種別を指定する機能」と「道路区間距離を指定する機能」の設定を変更することで、作成される渋滞情報がどうのように変化するか確認する。

処理量増加への対応

25 台構成（シナリオ 2 の構成）で対象道路を拡大した場合と、対象道路の拡大に対応し、台数を 93 台に増加させた構成(シナリオ 3 の構成)で実行した結果を表 6-7 に示す。

表 6-7 シナリオ 3 の実行時間

No.	実行時の構成	サーバ台数	処理時間（秒）
1	シナリオ 2 の構成	25	548
2	シナリオ 3 の構成	93	221

(注)　処理時間については実行した中で最大の時間を掲載している。

シナリオ 2 の構成（サーバ台数 25 台)では、548 秒かかっており、対象道路の拡大と精度向上のための処理量の変化に対応できなくなっている。しかし、サーバ台数を増加させることで処理時間が 221 秒に短縮され、対象道路の拡大に伴う処理量の変化に対応できることがわかった。

精度への影響

シナリオ 2 とシナリオ 3 で実行した渋滞解析結果の渋滞情報のデータ数をまとめたもの表 6-8 に示す。

表 6-8 道路種別変更による短時間渋滞情報の件数

No.	道路の種類	渋滞情報数
1	主要道路(シナリオ 2 の道路種別)	295,745
2	一般道路(シナリオ 3 の道路種別)	457,816

渋滞解析の対象となる道路の種類を「主要道路」から「一般道路」へ拡大したこと

で、渋滞情報数が約 154%増加した。これまで、対象外としていた道路に対して渋滞情報が作成され、より高精度の渋滞情報が作成できることが確認できた。

なお、作成した短時間渋滞情報を可視化すると、図 6-5、図 6-6 のとおりとなる。渋滞情報を作成する対象の道路を増加させたことで、これまで対象となっていない一般道路の渋滞情報が作成されていることがわかる。また、個々の道路を分割する際の区間を短くしたことで、より細かい間隔での渋滞情報が作成されていることがわかる。

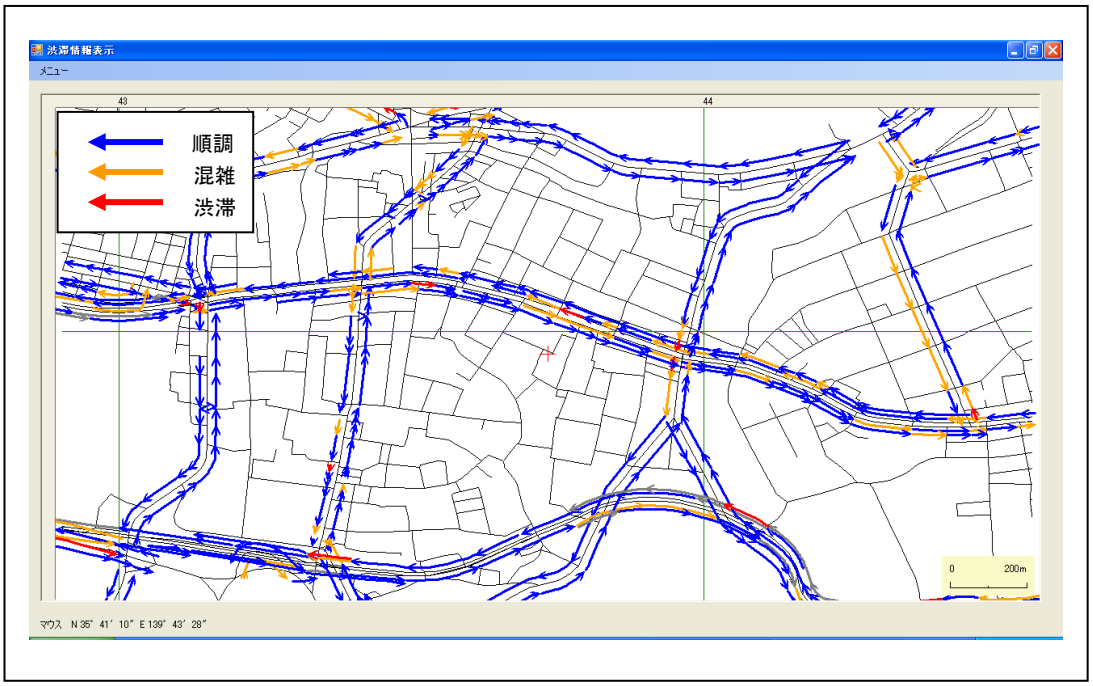


図 6-5 シナリオ 2 の渋滞情報の可視化

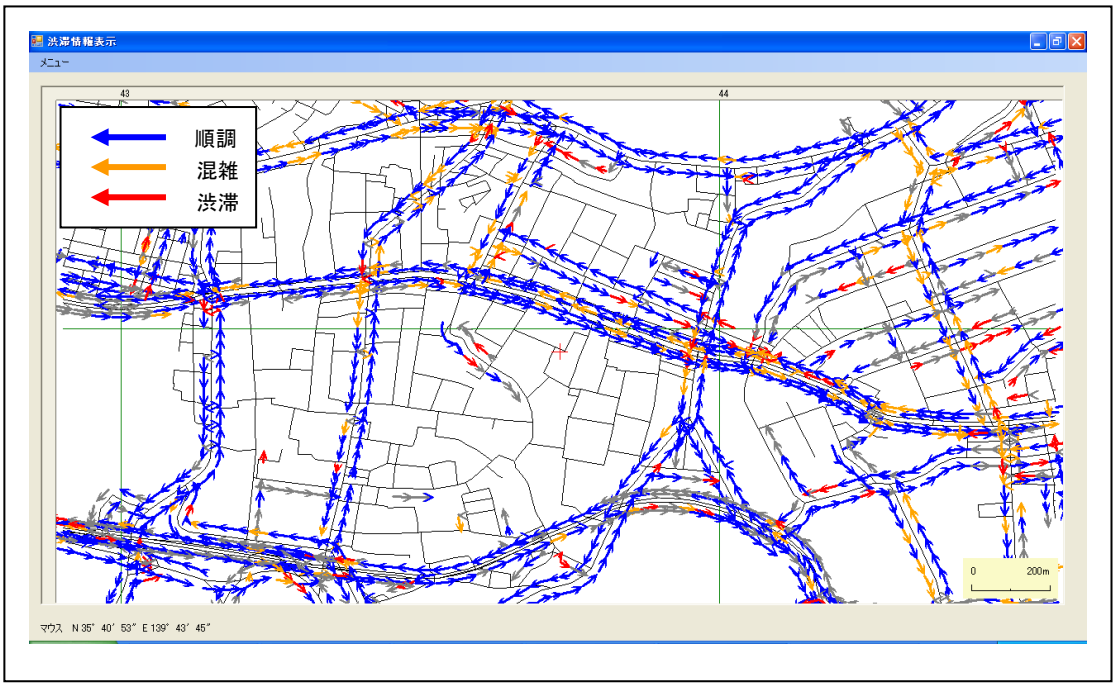


図 6-6 シナリオ 3 の渋滞情報の可視化

6.3.2 大規模データを使った処理に関する実証実験

本項では、「表 6-2 大規模データを使った処理に関する実証実験シナリオ」に従い、数テラバイトの大規模プローブデータを用い渋滞統計生成処理を実行し、大規模データに対して渋滞解析が行えることを確認する。また、「9.6 MapReduce 処理時間見積もり方法」を使用し、渋滞解析にかかる処理時間の見積もりを行い、大規模データに対しても「9.6.2 MapReduce 処理時間見積もり式」が適用できることを確認する。渋滞統計生成処理で実行するシナリオを表 6-9 に再掲する。

表 6-9 大規模データを使った処理に関する実証実験シナリオ（再掲）

シナリオ No.	想定シナリオ	サーバ台数	データ量	処理精度（道路種別）	処理精度(道路区間)	実施内容
1	大規模データの実行	93 台	約 2.1TB（約 140 億件、1 年分相当）	詳細	一般道路	1 年分のデータで統計処理を実行し、処理時間をシナリオ 1 で見積もった時間と比較する。

6.3.2.1 実行時間予測

シナリオ 1 の実行に先立ち、実行時間の予測を行う。予測時間の算出については「9.6.2 MapReduce 処理時間見積もり式」に基づき、「少量データ実行により確定するパラメータ」と「大規模データ実行環境」を使用する。なお、見積もり手順については 9 章を参照されたい。

少量データ実行により確定するパラメータ

少量データの実行では、1 カ月分のタクシープローブデータと携帯電話プローブデータを使用し、渋滞統計生成処理を実行した「3.5 アプリケーション評価」の結果を利用する。「9.6.2 処理時間見積もり式」で取り扱う少量プローブデータ実行結果及び実行設定値を表 6-10 に示す。

表 6-10 少量データ実行結果

No.	設定値	タクシープローブ解析	携帯電話プローブ解析	解析結果集計
1	Map 処理時間（秒）	546	492	119
2	Reduce 処理時間（秒）	572	5349	128
3	Map 入力データ量	約 850MB	約 173GB	約 7.8GB
4	Reduce 入力データ量	約 750MB	約 217GB	約 21GB
5	Map 分割数	1386	2782	2600
6	Reduce 分割数	1300	1300	50
7	Map スロット数	260	260	260
8	Reduce スロット数	260	260	260

大規模データ実行環境

大規模プローブデータ実行設定値は実際に使用する入力データ量と Hadoop パラメータとなる MapReduce 分割数、スロット数を使用する。大規模データ実行設定を表 6-11 に示す。

表 6-11 大規模データ実行設定一覧

No.	設定値	タクシープローブ解析	携帯電話プローブ解析	集計結果解析
1	入力データ量	約 4.6GB	約 1.9TB	約 85GB
2	Map 分割数	3900	31840	2600
3	Reduce 分割数	1300	1300	100
4	Map スロット数	260	260	260
5	Reduce スロット数	260	260	260

以上で挙げた表 6-11 の値を「9.6.2 MapReduce 処理時間見積もり式」に当てはめ、大規模プローブ処理時間の見積もりを行う。計算の途中過程についてはここでは省略するが、見積もり式の詳細については 9.6 を参照されたい。

MapReduce 処理見積もり時間

「少量データ実行により確定するパラメータ」と「大規模プローブ実行環境」を「9.6.2 MapReduce 処理時間見積もり式」に当てはめ、大規模プローブデータを使用した渋滞統計生成処理時間を見積もった結果を表 6-12 に示す。

表 6-12MapReduce 処理時間の見積もり結果

No.	データ名	Map 処理 見積もり時間	Reduce 処理 見積もり時間	Total 見積もり時間
1	タクシープローブ	51 分 2 秒	46 分 37 秒	56 分 11 秒
2	携帯電話プローブ	1 時間 33 分 48 秒	16 時間 59 分 43 秒	17 時間 4 分 25 秒
3	渋滞解析結果	21 分 37 秒	41 分 26 秒	42 分 31 秒

6.3.2.2 大規模プローブデータを利用した渋滞統計生成処理の実行

表 6-9 のシナリオ 1 に基づいて、タクシープローブ半年分、携帯電話プローブ 1 年分を入力データとし、大規模データを使用した渋滞統計生成処理を実行し、次のことを確認する。

- 大規模データ実行の確認
  数テラバイトの大規模データに対しても処理が可能であることを確認する。
- 見積もり時間との比較
  大規模データの実行時間を見積もりと、実際に実行した際の実行時間と比較し、大規模データに対しても 8 章の「MapReduce 処理時間見積もり方法」が適用できることを確認する。

大規模データ実行の確認

表 6-13 に大規模データを使用した渋滞統計生成処理の実行結果を示す。

表 6-13　大規模データの実行結果

No.	構成	サーバ台数	処理時間（秒）
1	シナリオ 1 の構成	93 台	21 時間 53 分 10 秒

処理途中に 10 台のマシンで､ディスク容量不足によるタスクの失敗が発生したが、失敗したタスクを他のサーバが代わりに処理したため、MapReduce ジョブについては、無事完了できた。

しかし、処理時間については、ディスク容量不足により 10 台のサーバでタスクを処理していないため、その分遅くなっている。Hadoop では Map 処理の出力は、Map 処理が完了したサーバのディスクに一時的に出力される｡そのため HDFS 領域以外にも、各サーバで Map 処理の出力を処理するための領域が必要となる。本実証実験環境では、想定以上に Map 処理の出力によるディスク書込みが発生し、本問題が発生した。大規模データに対して処理する際は、HDFS の領域だけでなく、Map 処理の出力を処理するための領域についても十分に確保する必要がある。

見積もりと実行結果の比較

渋滞統計生成処理の見積もりと、大容量データでの実行結果を表 6-14 に示す。

表 6-14 大規模データ実行時間と見積もり時間の比較

No.	処理名	実行時間	見積もり時間	誤差割合 (実行-見積)／見積
1	タクシー Map 処理	43 分 45 秒	51 分 2 秒	-14.5%
2	タクシー Reduce 処理	44 分 8 秒	46 分 37 秒	-17.7%
3	タクシー Total 処理時間	46 分 41 秒	56 分 11 秒	-16.9%
4	携帯電話 Map 処理	1 時間 54 分 1 秒	1 時間 33 分 48 秒	21.5%
5	携帯電話 Reduce 処理	20 時間 40 分 11 秒	16 時間 59 分 43 秒	21.6%
6	携帯電話 Total 処理時間	20 時間 43 分 24 秒	17 時間 4 分 25 秒	21.3%
7	解析結果集計 Map 処理	11 分 2 秒	21 分 37 秒	-48.9%
8	解析結果集計 Reduce 処理	20 分 51 秒	41 分 26 秒	-49.6%
9	解析結果集計 Total 処理時間	23 分 5 秒	42 分 31 秒	-45.7%
10	Total 処理時間	21 時間 53 分 10 秒	18 時間 43 分 7 秒	16.9%

見積もり時間と、実際の処理時間を比較すると、全体で 16.9%の誤差となり、9 章の「MapReduce 処理時間見積もり方法」の傾向と一致した。よって、数テラバイトのデータに対して行った「MapReduce 処理時間見積もり方法」が適用できることが確認できた。

個別に確認していくと、タクシープローブの Map 処理、Reduce 処理では、-14.5%、-17.7%となっており、9 章の傾向どおり 30%以内におさまっている。

携帯電話の Map 処理、Reduce 処理では、実行時間が見積もり時間を越える誤差が

発生している。これは「大規模データ実行の確認」で記載したディスク容量不足によるタスクの失敗が発生したことが原因となり、10 台分のサーバリソースが利用できなかったためである。

解析結果集計の Map 処理、Reduce 処理では、実行時間が見積もり時間より大幅に小さくなっている。解析結果集計は、タクシープローブ解析、携帯電話プローブ解析の処理の結果をまとめる処理のため、データ見積もり時に利用した入力データが極端に小さかった。そのため、実行時間が見積もり時間よりも大幅に小さくなったが、解析結果集計の Map 処理は全体への時間の割合が 1.7%と小さいため、全体への見積もりの精度へはさほど影響を与えていない。

6.4 クラウド基盤の信頼性向上に関する実証実験

本実証実験では、4 章の検討結果を用いてクラウド基盤の信頼性向上を確認することを目的とする。具体的には、実アプリケーション動作中に構成要素に対して故障を発生させ、実行中のジョブや後続ジョブへの影響がないことを確認する。なおマスタサーバの冗長化にはソフトウェア FT 技術である Kemari を使用する。

確認観点は以下の通りである。

(1) 故障が発生しても実行中のジョブが正常終了すること
(2) 故障により機器が縮退する場合は、クラスタ全体の性能が想定以上に劣化しないこと
(3) Kemari のオーバーヘッドが実証実験アプリケーションに影響を与えないこと
(4) 実証実験アプリケーションのスケーラビリティに Kemari が影響を与えないこと

故障を発生させる構成要素は以下のとおりである。括弧内の見出し番号は対応する節を表している。

(1) タスク(6.4.1)
(2) スレーブサーバ(6.4.2)
(3) マスタサーバ(JotTracker/NameNode)(6.4.3)
(4) ラック内 L2 スイッチ(6.4.4)

6.4.1 タスク失敗時の可用性確保の確認

1 台のスレーブサーバをディスクフルとし、実行されるタスクが異常終了する状態とした。異常終了したタスクは別のスレーブサーバで再実行され、ジョブが正常終了することを確認した。

6.4.2 スレーブサーバ故障時の可用性確保の確認

ジョブ実行中にスレーブサーバの 1 台(r4-1-0-01)を電源断により停止させた。停止したサーバで実行中だった Reduce タスクが遅延したが、実証実験アプリケーションがタイムアウトとなるような影響はなかった。

表 6-15 の遅延したタスクの実行結果を示す。タスク自体は他のスレーブサーバ(r5-1-0-12)に割り当てられたが、サーバ電源断によりシャッフルのタイムアウトが発生し、Map タスクの再実行が行われたためである。

表 6-15 停止させたスレーブサーバに割り当てられた Reduce タスク

No	実行サーバ	タスク開始	シャッフル時間	タスク実行時間	結果
1	r4-1-0-01	11:10:22	35 秒	―	電源断により強制終了
2	r5-1-0-12	11:11:41	10 分 9 秒	10 分 21 秒	正常終了

6.4.3 マスタサーバ故障時の可用性確保の確認

マスタサーバにソフトウェア FT を適用している状態で、ジョブ実行中にマスタサーバを停止させた。JobTracker サーバ、NameNode サーバでの結果を次に記載する。ソフトウェア FT の切り替え時間は待機系サーバでの停止検知～GratiousARP 送信完了までとした。

6.4.3.1 JobTracker サーバ故障時

JobTracker の現用系サーバを停止し、ジョブが継続するか確認を行った。仮想マシンの切り替えは 4 秒で完了し、JobTracker プロセスは切り替え後に待機系サーバにて動作し続け、Hadoop ジョブとしては個々のタスクが失敗することなく正常終了した。切り替えを発生させた時間に実行したジョブのタスク情報を表 6-16 に示す。

表 6-16 MapReduce ジョブ実行結果

No	タスク種別	成功タスク	失敗タスク
1	Map	184	0
2	Reduce	50	0

6.4.3.2 NameNode サーバ故障時

NameNode の現用系サーバを停止し、ジョブが継続するか確認を行った。仮想マシンの切り替えは 2 秒で完了し、NameNode プロセスは切り替え後に待機系サーバにて動作し続け、Hadoop ジョブとしては個々のタスクが失敗することなく正常終了した。切り替えを発生させた時間に実行したジョブのタスク情報を表 6-17 に示す。

表 6-17 MapReduce ジョブ実行結果

No	タスク種別	成功タスク	失敗タスク
1	Map	184	0
2	Reduce	268	0

ソフトウェア FT 技術により、切り替え時にも実証実験アプリケーションに影響を与えないことが分かった。

6.4.4 1 ラック全体の故障時の可用性確保の確認

ジョブ実行中にラック内スイッチ(r6)を停止し、スレーブサーバ 10 台を縮退させた。1 ラック停止前後でのジョブ実行結果を表 6-18 に示す。

ラック停止時に実行中であったジョブは 6.4.1 と同じく実行時間が長くなるが、ジョブは正常に終了し、次回のジョブの実行時間には影響がなかった。このことから 1 ラック停止しても実証実験アプリケーションの可用性が確保されていることが分かった。

表 6-18 ジョブ実行結果

No	ジョブ実行タイミングと故障の関係	ジョブ開始時刻	ジョブ終了時間	結果
1	ラック内スイッチ停止前	11:17:01	1 分 46 秒	正常終了
2	本ジョブ実行中に停止	11:21:21	20 分 1 秒	正常終了
3	停止後の新規ジョブ	11:41:24	2 分 8 秒	正常終了

またラック停止中のリソース使用という観点で同試験を行った。ラックを停止した時間帯の Hadoop 全体の CPU 使用率および HDFS のレプリカ対象ブロック数を図 6-7 に示す。停止した 10 台分のスレーブサーバが保持しているデータの再レプリケーションに 15 分程度かかり、Hadoop 全体として I/O 待ちが発生していることが分かる。

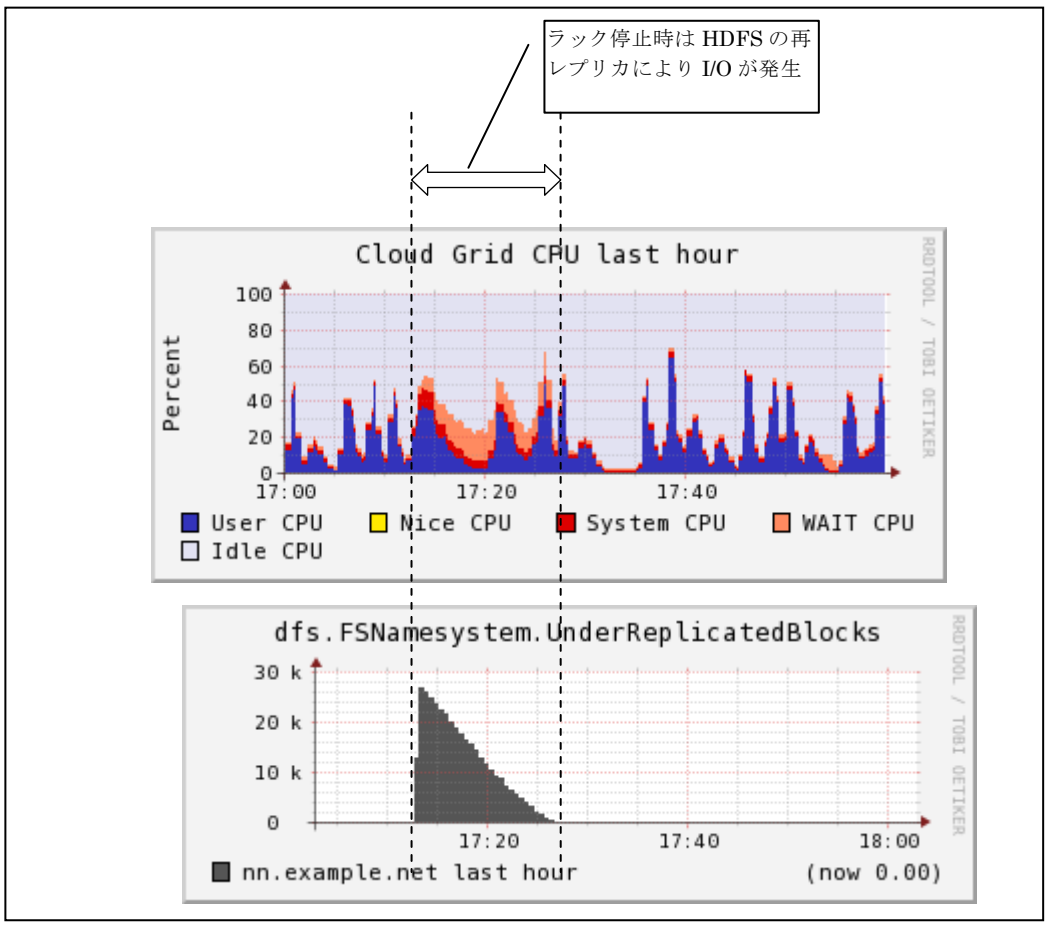


図 6-7 1 ラック停止前後の全体 CPU 使用率

6.4.5 Kemari によるオーバーヘッドの確認

ソフトウェア FT 技術である Kemari が行っている仮想マシン同期のオーバーヘッドが実証実験アプリケーションに影響を与えるか確認を行った。

ネットワーク I/O をトリガに仮想マシンの同期を行うという Kemari のアーキテクチャから、スレーブサーバのマシン台数が変化する表 6-1 のシナリオにしたがって、Kemari が性能に与える影響を確認した。結果を表 6-19 に示す。

表 6-19 Kemari オーバーヘッド

No.	想定シナリオ	マシン台数	通常環境	Kemari 環境	オーバーヘッド
1	小規模構成によるサービススタート	3	180 秒	256 秒	29%
2	データ量の増大とサーバ増強	25	249 秒	485 秒	48%
3	対象道路の拡大とサーバ増強	93	258 秒	553 秒	53%

Kemari のオーバーヘッドはスレーブサーバのマシン台数に依存していることが分かる。これは、台数の増加にしたがってマスタサーバとスレーブサーバの通信が増加するためと予想される。

Kemari を実行しているサーバでのネットワーク使用量のグラフを図 6-8 に示す。サーバ数増加に伴って転送量は増加しているが、4 章で実施したネットワークスループットの上限である 40Mbps には達していない。このため、オーバーヘッドの影響は通信量ではなく、通信のレイテンシが積み重なった結果性能に影響を与えていると予想される。

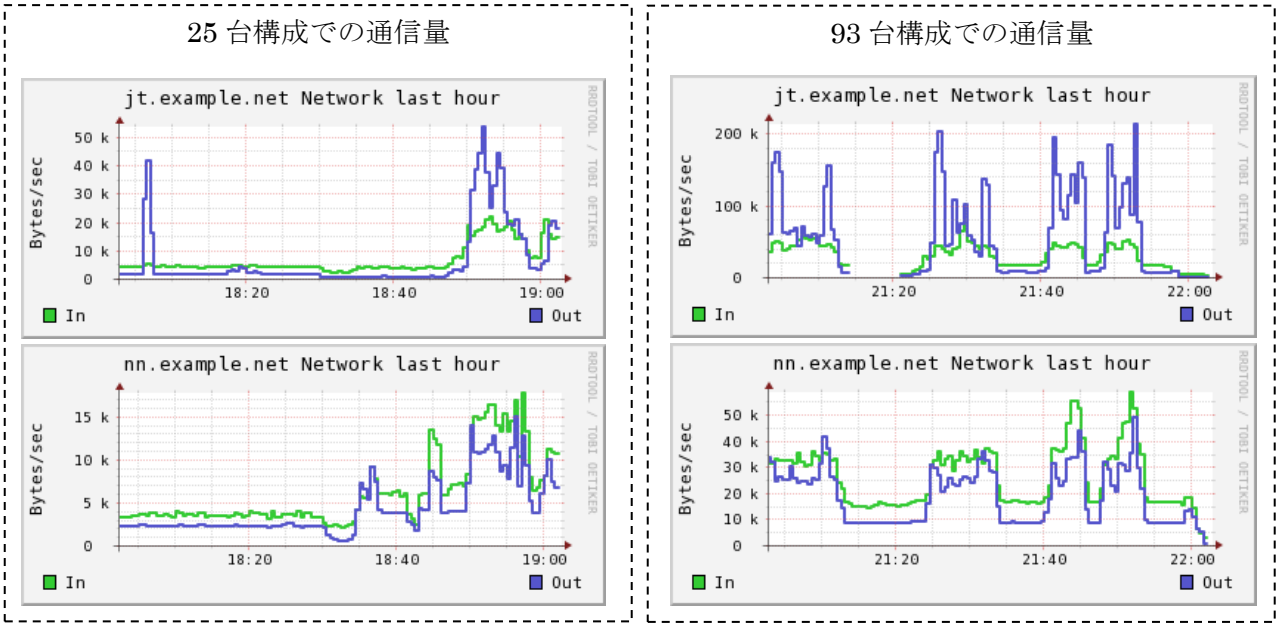


図 6-8 ネットワーク使用量グラフ

6.5 クラウド基盤の運用効率化に関する実証実験

本実証実験では、5 章で検討した運用効率化のための方式を、表 6-1 で示したシナリオに従い、それらの有効性を評価・検討する。具体的には、以下の二つのユースケースを実証実験として取り上げる。5 章で確認した「スケーラビリティ」「クラスタ構成要素の変更」「基盤の混在性」を意識した運用手法が有効であることを再確認するとともに、システムを無停止でメンテナンスすることを想定し、実行中のジョブに影響しないように各運用項目が実施できるかを確認する。

(1) 初期構築・増設の効率的な実施
(2) スレーブサーバの故障検知と効率的な復旧

6.5.1 効率的な初期構築と増設に関する実証実験

多数存在するスレーブサーバの初期構築に関しては、実証実験アプリケーションの動作環境とは独立しているため、詳細は 12 章を参照されたい。構築においては、種別・機種混在環境において、同時並列的に構築できるとともに、作業者の作業項目は、サーバの電源を投入するだけであり、種別・機種の混在下の大量サーバ環境において構築を効率化した。

以下では、増設作業が効率的に行えるかの実証実験を行う。検証観点は以下の通りである。

① 増設と初期構築が共通化された運用手順で増設が可能であることを確認する
② 増設作業が、既存のジョブ実行を干渉することがないことを確認する

6.5.1.1 シナリオ

Hadoop スレーブサーバ25台の小規模な分散処理基盤でアプリケーションを動作させていたが、データ増加に伴い、分散処理基盤を 100 台まで増設する。なお、増設作業中においてもジョブの実行が必要とされていると想定する。そのため、増設作業中でも既存のジョブ実行を干渉することがないことを保証する必要がある。

6.5.1.2 増設手順の共通化の確認

観点①を検証するため、増設手順の共通化・効率化を取り上げる。すなわち、運用作業時間が短縮されていること、及び増設手順自体が、共通化され、任意のタイミングで増設が可能であることを確認する。表 6-20 に増設手順と、その結果を記載する。前提として、スイッチ・サーバの物理配置（ラッキング）、LAN 結線（ケーブリング）は済んでいる。

表 6-20 増設時の実証実験手順

No	作業内容	作業対象	作業時間	総時間
1	初期構築と同様に構築を実施（サーバの電源を投入するのみ。構築最中に自動で構成管理サーバと通信を行うため、初期構築後の基盤への設定変更は全て反映済み）	70 台	10 分	96 分
2	TaskTracker 自動起動時に、実証実験アプリケーションに必要な資材を自動同期 自動でスレーブサーバプロセスが立ち上がり、クラスタに自動登録される。	70 台	0 分	約 5 分(サーバ起動時間)
3	Hadoop マスタサーバで HDFS のリバランス処	1 台	5 分	5 分

No	作業内容	作業対象	作業時間	総時間
	理を実施する。			
4	ジョブクライアントで、サーバ増減数に応じた設定の変更を実施する	1 台	5 分	5 分

増設の際に作業の大部分を占める項番 1,2 は、初期構築と全く同じ運用手順であることを確認した。これは、初期構築から増設にいたる運用の中において、設定変更などスレーブサーバへの変更は全て構成管理サーバ経由でおこなうという維持管理手順に従ったためである。従って初期構築と共通化された運用手順で、任意のタイミングで増設が可能であることを確認した（観点①)。

増設におけるインストール時間の分布を図 6-9 に記載する。

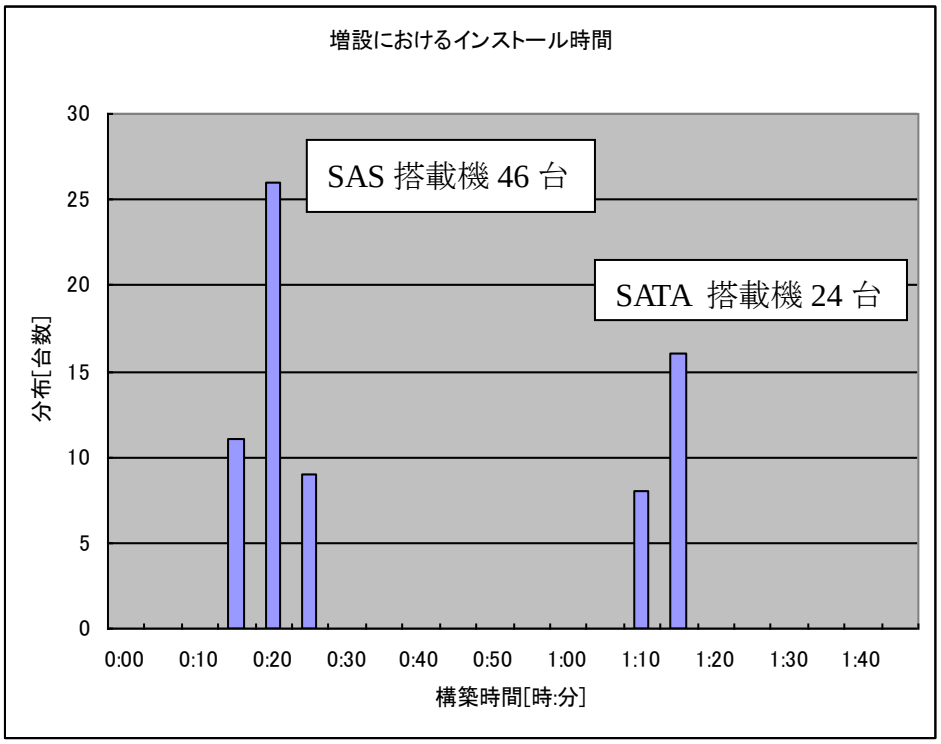


図 6-9 増設におけるインストール時間の分布

インストール時間がマシンによって分散するのは、内蔵ディスクの書き込み性能によるものである。参考情報として、表 6-21 にディスク性能の比較値を記載する。

表 6-21 tiobench ベンチマークによるディスク I/O 速度（代表値）

No.	名前	CPU	ディスク	スコア（Mbyte/s・1 本） seq read / seq write
1	SAS 搭載機(代表値)	2.33GHz×2 個 8 コア	SAS 146GB×2	60.939 / 62.358
2	SATA 搭載機(代表値)	Core2 Duo T9400 2.53GHz 2 コア	SATA 250GB ×2	95.419/ 26.693

なお、増設最中における、関連する管理系サーバのリソース使用状況を図 6-10 に併記する。

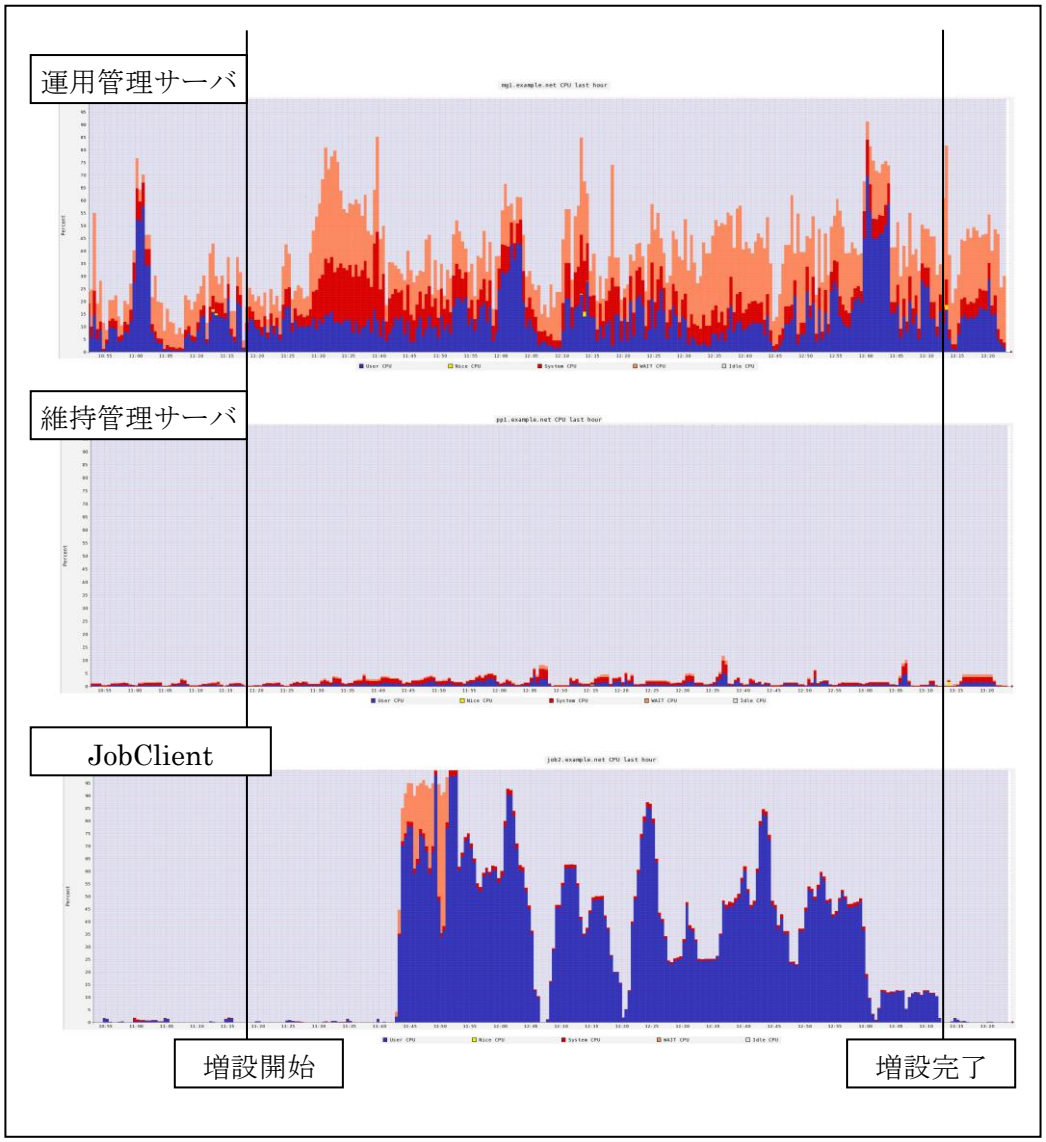


図 6-10 増設中の各管理サーバの CPU リソース使用率

運用管理サーバは、定常的に負荷が高めであるが、増設時にボトルネックになるような負荷に達していない。Ganglia による IO 負荷（橙色）は増設前後で、可視化対象の増大に伴い、定常負荷が増大している。なお、運用管理サーバにおける二つの大きなピークは定期的に実行されるリソースの収集ジョブの実行結果である。構成管理サーバおよび JobClient は、増設後には、リソースの定常負荷に戻った。

6.5.1.3 ジョブ実行への影響の確認

観点②を検証するため、増設がジョブ実行に影響を与えないことを確かめる。以下に、増設前、増設中、増設後におけるジョブの実行履歴を示す。

表 6-22 増設中のジョブ実行結果

No	ステータス	結果（注）	平均ジョブ実行時間
1	増設前	正常終了	233 秒（増設直前のジョブ 8 回の平均）
2	増設中	正常終了	149 秒（ジョブ 19 回の平均）
3	増設後	正常終了	98 秒　（増設直後のジョブ 9 回の平均）

注：正常終了は、5 分以内にジョブが正常に終了したことを表す

増設中においてもジョブは想定時間内に正常に終了しており、増設作業によって既存のジョブの実行が妨げられないことを確認した（観点②）。

また、増設中におけるリソース状況を記載する。以下に増設最中の CPU 使用率を記載する。増設に伴いジョブの負荷が分散され、全体の CPU 使用率が減少していくことを確認した。

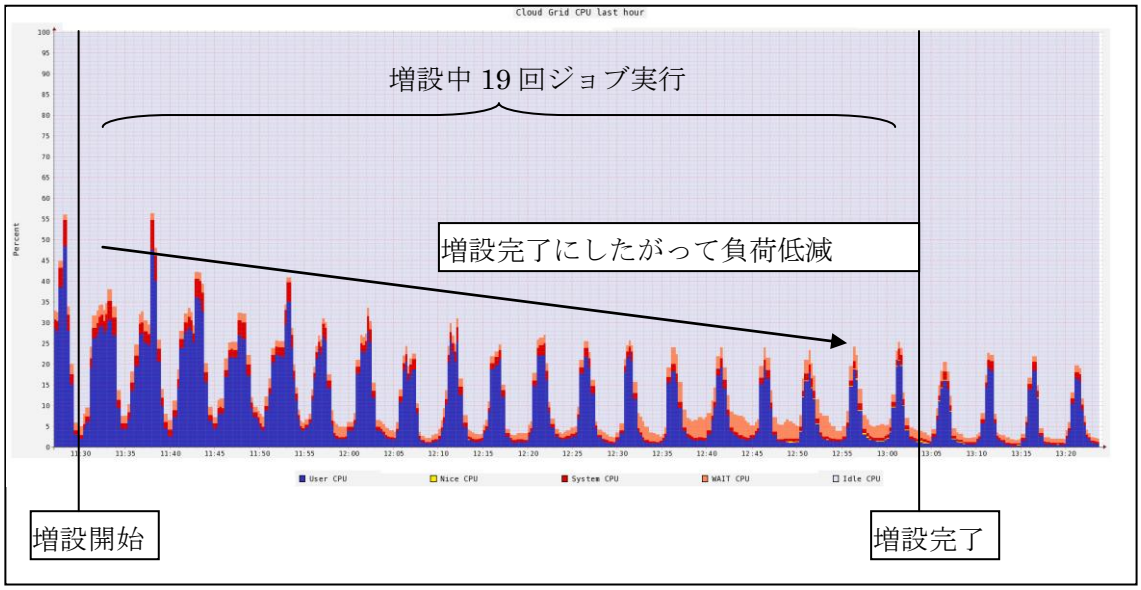


図 6-11 増設中における Hadoop 基盤全体の CPU 使用率の遷移

以下に増設対象であるラック番号 4 のリソース状況を記載する。リソース状況の可視化に関しては、増設時に自動でインストールされる Ganglia を利用して、ラック 4 に収納されている全てのサーバを集約表示したものである。

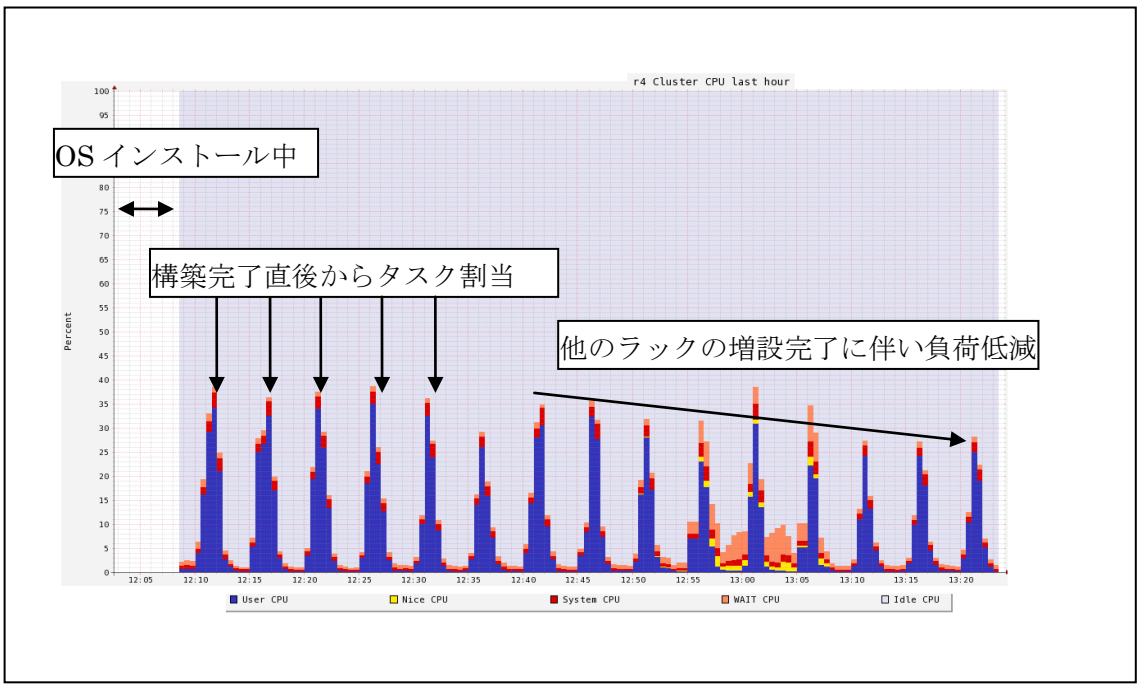


図 6-12 ラック番号 4 の増設における CPU 使用率の遷移

増設の際の OS のインストールが完了した直後からリソース状況が取得され、起動してすぐに Hadoop クラスタに組み込まれてタスクが割り当てられていることが実証された。また、他ラックの増設が完了するに従って該当ラックの負荷が軽減されている様子を確認した。

6.5.1.4 アプリケーションデータの同期に関する補記

一般に Hadoop 基盤においては、JobClient サーバにジョブを配置するだけで実行が可能であり、ジョブ実行時にアプリケーションが各スレーブサーバに自動的に配置される。

一方、Hadoop アプリケーションでは、処理するべき入力データとは別に、各ジョブで共通的に利用するアプリケーションデータが存在する。このデータのサイズが大きい場合、ジョブ実行のたびにジョブとともに配置するのではなく、各スレーブサーバにあらかじめ配置しておくことが効率的である。

これを受け、本実証実験においてはデータをあらかじめ配置するために、図 6-13 に示すような同期方式を採用した。

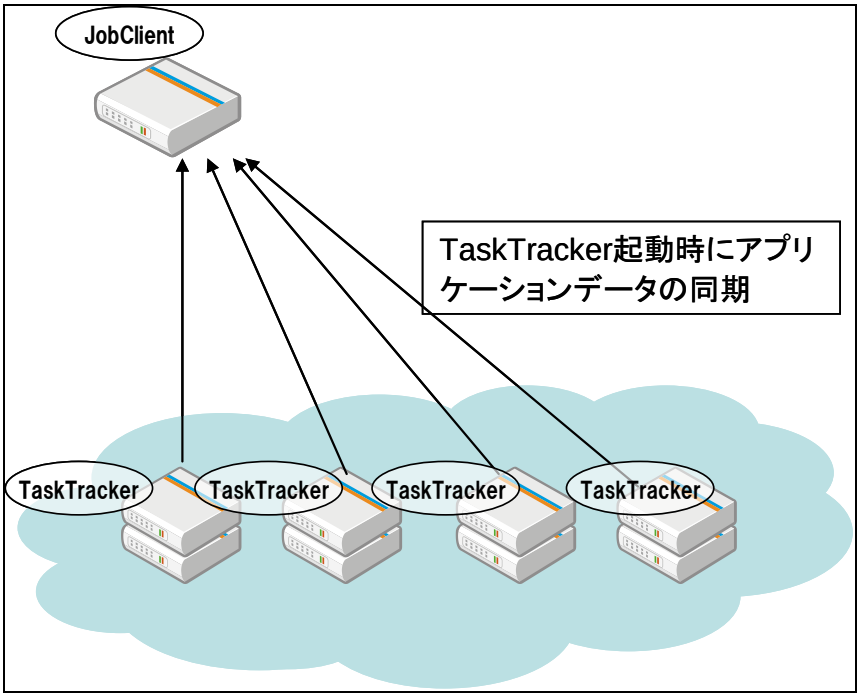


図 6-13 アプリケーションデータの同期

以下に実装の概要を示す。スレーブサーバでは分散計算を実施するための TaskTracker を起動する必要がある。この TaskTracker の起動時に JobClient に配置されているデータを同期していることを確認してから TaskTracker が起動するように起動スクリプトを作成した。なお、同期はオープンソースソフトウェアの rsync を利用した。

本実証実験では、この方式を採用したため、「図 6-10 増設中の各管理サーバの CPU リソース使用率」における JobClient の CPU リソースを使用することが観測される。このように、大量のデータを同期する場合には以下の点に留意する必要がある。

- 同期方法の選択
- ネットワーク帯域と圧縮転送による CPU 使用率のトレードオフ

本実証実験では、同期手法として Linux で標準的に利用されている rsync を利用した。rsync では転送チャネルに SSH を利用するが、転送の際に暗号化を行うために、CPU リソースを消費する。Hadoop 基盤において、クラスタメンバ間は同一のセキュリティゾーンに存在するため、暗号化は必須でないと判断した。そのため転送チャネルに暗号化を行わない RSH を利用した。また、同期に伴うネットワーク帯域と、圧縮に伴う CPU リソースのトレードオフが発生する。同期対象を圧縮転送することで、転送量を抑えることが可能である。本実証実験のデータでは、同期データ量が、5GB

程度存在するが、2GB 弱に圧縮されることを確認した。一方、圧縮に伴い、圧縮のために JobClient の CPU リソースを消費することが考えられる。本実証実験では、ネットワーク帯域への負荷を懸念し、圧縮転送を採用した。

6.5.2 故障検知と復旧に関する実証実験

Hadoop 分散環境の運用効率化のための二つ目の実証実験として、スレーブサーバの故障検知とその効率的な回復を取り上げる。5 章及び 10 章で検討した、スケーラブルな監視手法で故障検知が実現できることを、実証実験を通じて確認するとともに、その復旧が、初期構築（及び増設）と共通化された運用手順で可能であることを確認する。検証観点は以下の通りである。

① 初期構築（及び増設）と共通化された運用手順で復旧が可能であることを確認する
② スケーラブルな監視手法で故障検知が実現できることを確認する
③ 復旧中にジョブが妨げられることはないことを確認する

6.5.2.1 シナリオ

表 6-23 に示すように、スレーブサーバを擬似的に故障したと想定し、その復旧を行う。

表 6-23 擬似故障内容

No.	故障サーバ	擬似故障概要
1	r4-1-0-09	Hadoop スレーブサーバ故障。機器変更を行い、OS の再インストールが必要となる

6.5.2.2 故障検知手順と、復旧手順の共通化

観点①を確認する。11 章で検討した故障検知・回復フロー、及びその実測値を表 6-24 に記載する。

表 6-24 故障検知及び回復フロー

No.	時刻	作業内容	作業時間	総時間
1	16:50	サーバ r4-1-0-09 の強制電源断（擬似故障の発生をシミュレート）	-	-
2	16:52	Ganglia において故障を検知(図 6-14)	1 分	1 分
3	16:55	監視システムにおいて故障を検知(図 6-15)	1 分	1 分
4	17:00	サーバにログインを試みるもログイン不	5 分	5 分

No.	時刻	作業内容	作業時間	総時間
		可。サーバ故障と判断し、ハードウェアの交換の実施を決定（実際には再インストールのみを実施）		
5	17:02	故障回復手順にのっとり、維持管理サーバの該当サーバの証明書を削除	1 分	1 分
6	17:03	交換した代替機の電源を投入し、OS のインストールを開始	1 分	32 分
7	17:35	自動インストールが完了し、Ganglia, 監視システムにおいて問題ないことを確認	2 分	1 分

上記故障検知・復旧作業手順において、総時間のほとんどを占める OS の再インストール（復旧）に関しては、前述の初期構築・増設と全く同じ手順で実施可能であることを確認した（観点①）。これは、初期構築から故障検知・復旧にいたる運用の中において、設定変更など Hadoop スレーブサーバへの変更は全て構成管理サーバ経由でおこなうという維持管理手順に従ったためである。

6.5.2.3 故障検知と復旧の検知

観点②を確認するため、本節では故障検知の妥当性の評価を行う。表 6-24 の No2 における Ganglia による故障検知の様子を図 6-14 に示す。

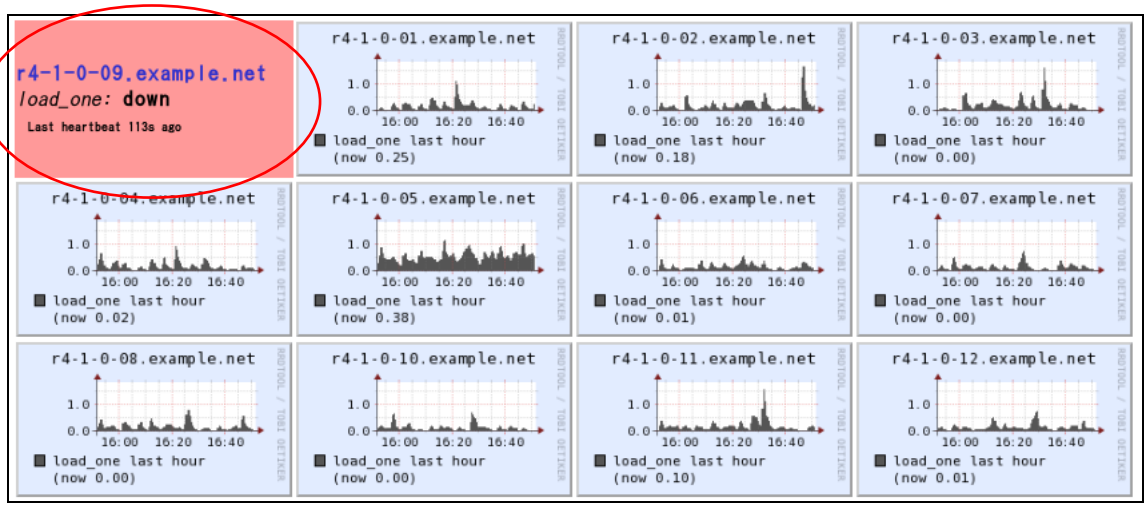
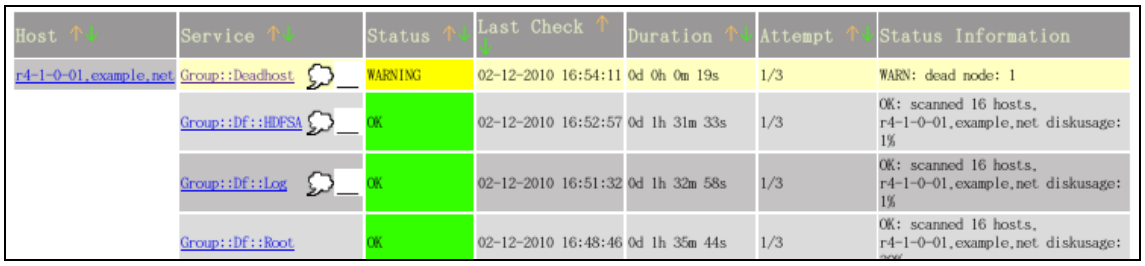


図 6-14 Ganglia での故障検知

本実証実験では、Ganglia の集約サーバは、故障機器と直接通信せず、故障機器を含むラックの代表サーバ（及びその代替サーバ）に情報を問い合わせるというスケー

ラブルな監視手法を採用したが、図に示されるように故障サーバの故障検知を明確に識別できる形で故障検知できることを実証実験においても確認した。

また、本実証実験では、監視システムと Ganglia を連携する方式を採用した。図 6-15 に監視システムにおける故障検知の様子を示す。ラック 4 の代表サーバに監視をすることで、ラック 4 の故障サーバを検知できることを確認した。



Host	Service	Status	Last Check	Duration	Attempt	Status Information
r4-1-0-01.example.net	Group::Deadhost	WARNING	02-12-2010 16:54:11	0d 0h 0m 19s	1/3	WARN: dead node: 1
	Group::Df::HDFSA	OK	02-12-2010 16:52:57	0d 1h 31m 33s	1/3	OK: scanned 16 hosts, r4-1-0-01.example.net diskusage: 1%
	Group::Df::Log	OK	02-12-2010 16:51:32	0d 1h 32m 58s	1/3	OK: scanned 16 hosts, r4-1-0-01.example.net diskusage: 1%
	Group::Df::Root	OK	02-12-2010 16:48:46	0d 1h 35m 44s	1/3	OK: scanned 16 hosts, r4-1-0-01.example.net diskusage:

図 6-15 監視システムでの故障検知

上記の通り、Nagios, Ganglia において故障検知がなされることを確認した（観点②）。

監視システムに関しても Ganglia と同様、故障機器と直接通信せず、故障機器を含むラックの代表サーバ（及びその代替サーバ）に情報を問い合わせるというスケーラブルな監視手法を採用したが、図に示されるように故障サーバの故障検知を明確に識別できる形で故障検知できることを確認した。図 6-16 に復旧後のサーバの様子を示す。

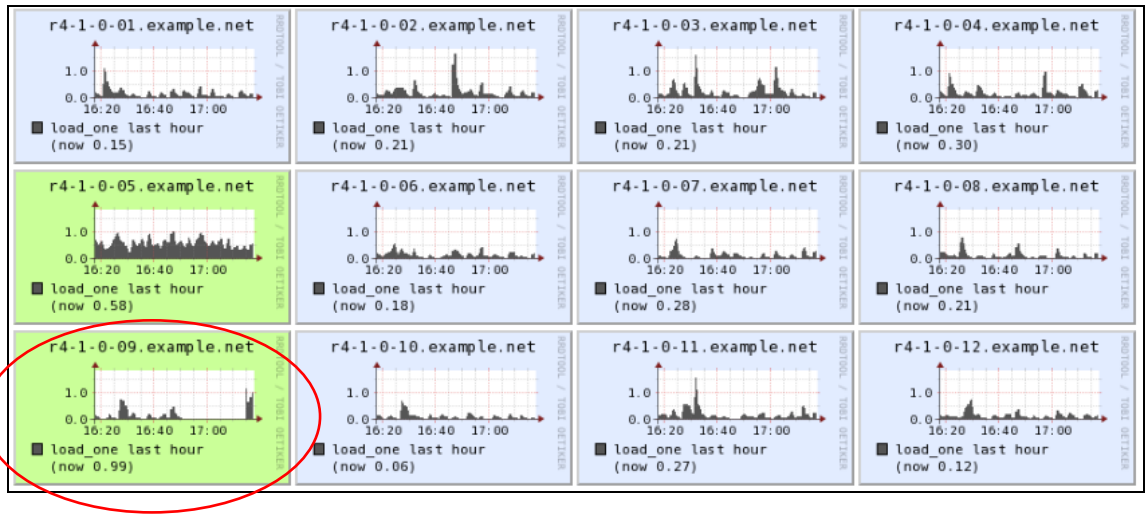


図 6-16 復旧後のタスク割当

復旧後 Ganglia が再度リソース情報を収集できるとともに、タスクが再割当されている様子が観測され、復旧が自動的に行われたことを確認した。

6.5.2.4 ジョブへの影響

観点③を確認する。故障前、故障中、復旧中、復旧後におけるジョブへの影響を表 6-25 に記述する。

表 6-25 故障検知・復旧中のジョブ実行結果

No	ステータス	結果（注）	平均ジョブ実行時間
1	故障前	正常終了	85.6 秒（故障直前のジョブ 4 回の平均)
2	故障中	正常終了	84.5 秒（故障中のジョブ 2 回の平均)
3	復旧中	正常終了	86.3 秒（復旧中ジョブ 7 回の平均)
3	復旧後	正常終了	86.8 秒（復旧直後のジョブ 5 回の平均)

(注) 正常終了は、5 分以内にジョブが正常に終了したことを表す

上記結果より、故障中および復旧中、復旧後のいずれの時間帯においても、目立った性能劣化をせずに、5 分以内にジョブが正常に終了したことを確かめた。すなわち、故障及び復旧作業がジョブ実行に干渉しないことを確認した（観点③)。

6.6 実証実験のまとめ

クラウド型分散処理基盤の有用性を確認するために本実証実験を行い、以下の結果が得られた。

- アプリケーションの動作特性に関する実証

  渋滞解析処理アプリケーションが、実際のタクシー及び携帯電話から取得したプローブデータに対して動作することを確認した。また、システムの長期的な発展を意識したシナリオに沿ってデータ量の大幅増加、処理精度の向上、サーバ台数の増加を行い、システムの発展に必要なスケーラビリティを有していることを確認した。データ量が非常に多い場合の処理として 1 年分のプローブデータの渋滞統計処理を行い、ディスク容量が少ない一部のサーバでエラーが発生したが、処理全体としては問題なく完了することを確認した。

- クラウド基盤の信頼性向上に関する実証

  ソフトウェア FT 技術である Kemari を Hadoop マスタサーバに適用することで、アプリケーションが動作中に Hadoop マスタサーバが故障しても処理が継続でき、大幅に可用性が向上したことを確認した。一方、Kemari による

オーバーヘッドについては、Hadoop スレーブサーバの台数が増えるほどオーバーヘッドが増加する傾向にあり、性能面での課題が判明した。また、Hadoop スレーブサーバの故障、ラック単位での故障に関してもアプリケーションのジョブがエラーになることなく処理が行われ、データが失われないことを確認した。

・ クラウド基盤の運用効率化に関する実証

スペックの異なるハードウェアが混在した環境においても自動構築手法が適用できることを確認した。アプリケーションが動作している状態でサーバの増設を行い、動作中のアプリケーションに影響を及ぼすことなく増設が行えること、増設終了すぐにアプリケーション処理のためのリソースとして使用されたことを確認した。故障検知、回復作業が想定どおりの手順で少ない作業量で行えることを確認した。

今回の実証実験では、Hadoop は数 10 秒～2,3 分程度の短時間ジョブの実行効率が低いことを確認した。短時間渋滞解析処理でサーバ台数を約 8 倍(3 台から 25 台)に増やした場合に実行時間は約 5 分の 1(1223 秒から 240 秒)となったように完全な比例はしていない。これは、Hadoop の実装で秒単位での間隔のポーリングやスリープ処理を複数個所で行っていることが原因と考えられる。大量のデータを大量のサーバを用いて短時間で処理を行う目的に対しては、Hadoop の実装の改良が必要であると思われる。

また、ソフトウェア FT 技術の Kemari を適用した場合、サーバ 93 台で 53%の速度低下が発生することを確認した。ソフトウェア FT を採用する際には適用可能な範囲の見極めが重要になる。適用範囲の拡大のためには実装の改良が必要になると思われる。

7　おわりに

本事業では、データセンターにおけるクラウド型サービスの一環として、大量なデータを蓄積するとともに多様な用途に合わせて柔軟な処理をスケーラブルに扱うサービスをビジネス用途に提供するため、必要となる分散処理基盤技術について「適用性、信頼性、運用性」の観点から技術開発を行った。また、100 台規模の実証実験用クラウド環境を構築するとともに、プローブデータによる渋滞解析をケーススタディとして実証実験を実施することで開発した成果の有効性を確認した。

「適用性」においては、MapReduce アプリケーションの設計手法と、アプリケーションの性能評価手法の整備を行い、以下の成果を得た。

- MapReduce アプリケーションの設計手法として、アプリケーション開発の流れと考慮すべきポイントを説明した。特に、Map と Reduce の処理フェーズを分ける箇所を導き出す手法として、アプリケーションの処理フローを並列に並べた上で他ノードの処理結果に依存している箇所を順に調査する方法を説明し、実際に渋滞解析アプリケーションの開発事例で手法が適用できることを確認した。本手法は一般的な設計プロセスの成果物である処理フローを基としたため、開発プロセスへの取り込みが容易に行え、短期間で Map 処理、Reduce 処理でおこなう処理を弁別し、その入出力データを決定することができる。

- アプリケーションの性能評価手法として、効率的にジョブが実行できる基盤パラメタの設定手法を説明した。Map スロット数・Reduce スロット数の決定方法、実メモリ量やディスクに関する制約事項、Map 処理分割・Reduce 処理分割の方針を説明した。また、ハードウェアスペックが混在する Hadoop 基盤では Map スロット数・Reduce スロット数を CPU のコア数に比例して設定することでスペックの混在の影響を軽減できることを示した。そして、数 GB 程度の少量データを処理する MapReduce ジョブの実行結果から、数百 GB 以上のデータを処理する場合の MapReduce 処理時間を見積もる手法を提案し、実際に渋滞解析アプリケーションで見積もり手法が有効であることを確認した。

「信頼性」においては、Hadoop 基盤の可用性を担保するための技術の整備を行い、以下の成果を得た。

- Hadoop 基盤の可用性確保の検討を行い、元々の Hadoop 基盤が有している Hadoop スレーブサーバの冗長性に加え、Hadoop マスタサーバとコア L3 スイッチの冗長化を行うことで、基盤全体の可用性が確保できることを確認した。

・ Hadoopマスタサーバの冗長化手法としてHAクラスタとソフトウェアFTの検討を行った。ソフトウェアFTについては、オープンソース実装のKemariを用いて検証を行ったところ、Hadoopマスタサーバに故障が発生した場合もジョブの再実行を行うことなく動作が継続できることが確認できた(HAクラスタの場合ジョブの動作継続はできない。ただし、Kemariを用いて冗長化を行ったことによるHadoopジョブの処理速度の低下は、3台の環境で約3割、100台の環境で約半分であり、今後解決すべき技術課題もある)。

「運用性」においては、Hadoop基盤の効率的な運用を実現する技術の整備を行い、以下の成果を得た。

・ Hadoop基盤の運用項目を検討し、自動化等で効率化すべき項目として、台数の多いHadoopスレーブサーバに対する運用項目の抽出を行った。自動構築手法として、Kickstartによるインストールと Puppetによる構成管理を組み合わせた自動化および一元管理の手法を整備した。100台の自動構築検証を行い、90分で完了することを確認した。

・ Hadoop基盤の動作状況を把握するために必要な項目を抽出し、可視化対象情報として整理した。可視化対象の台数が増大してもネットワーク通信量がボトルネックとならないGangliaを選択し、すべての可視化対象情報を可視化できるよう機能追加した。また、100台のHadoop基盤において性能の検証を行い、ネットワーク通信量がボトルネックにならないことと、本検討で必要と判断した可視化対象情報の収集であればPCクラスの監視サーバであっても400台規模までスケール可能なことを確認した。

今回の技術開発及び実証実験では、以下に挙げる課題が判明した。クラウド型分散処理基盤が提供する大規模な計算能力をより広い範囲で活用できるようにするには、これらの課題の解決が重要であると考えられる。

・ Hadoopは数10秒～2,3分程度の短時間ジョブの実行効率が低いことを確認した。短時間渋滞解析処理でサーバ台数を約8倍(3台から25台)に増やした場合に実行時間は約5分の1(1223秒から240秒)となったように完全な比例はしていない。これは、Hadoopの実装の中に、秒単位でのポーリングやスリープを用いた待ち合わせ処理が複数個所で用いられていることが原因と考えられる。大量のデータを大量のサーバを用いて短時間で処理を行う目的に対しては、Hadoopの実装の改良が必要であると思われる。

- ソフトウェア FT 技術の Kemari を適用した場合、サーバ 93 台で 53%の速度低下が発生することを確認した。ソフトウェア FT を採用する際には適用可能な範囲の見極めが重要になる。適用範囲の拡大のためには実装の改良が必要になると思われる。

- 今回の実証実験ではクラウド固有の問題ではないため対象外としたが、プローブデータの提供元システムから定期的にプローブデータを取得しクラウド型分散処理基盤に効率的に格納する仕組みと、クラウド型分散処理基盤上での処理結果を後続の利用システムに提供する仕組みは、現実のシステムへの適用を考えた場合には重要であり、今後の技術整備が必要と考えられる。

- 今回の技術開発及び実証実験では単一のアプリケーションのみがクラウド型分散処理基盤を使用することを前提としていたが、クラウド型分散処理基盤の利用効率の更なる向上のために、複数の利用者で共同利用を行い複数のアプリケーションを同時に実行することが考えられる。そのためには、ジョブのスケジューリング技術の検討、クラウド型分散処理基盤に配置されるデータのアクセス制御手法の検討を行うとともに、その実用性を実証する必要がある。

第２編　技術トピック

8　MapReduce アプリケーション開発手法

本章では、分散処理プログラムモデル MapReduce を使用して、何らかの処理を MapReduce に適用する方法について述べる。処理を MapReduce に適用し、分散処理用のオープンソースソフトウェアである Hadoop で動作させる。

まず、MapReduce 自体の処理フローや特長を述べる。そして、PageRank 計算を例に、Hadoop 上での MapReduce に適用する場合について説明する。Hadoop への適用で注意すべきポイントを交えながら MapReduce アプリケーション作成方法を説明する。

また、Hadoop で MapReduce 処理を包含したツールである Pig や Hive について機能や使い方、適用するポイントを述べる。

8.1 MapReduce 概要

本節では、MapReduce の処理の流れや特長について説明する。

8.1.1　MapReduce とは

MapReduce は、Google が提唱した分散処理プログラムモデルである。MapReduce は、サーバやインターネットにある膨大な情報を分散並列処理によって効率よく処理させるために考案された。

MapReduce は、データを Key・Value 形式で扱う。MapReduce 処理フローは、図 8-1 に示すように 3 つのステップから構成される。

(1) Map 処理

入力データを、何らかの処理によって Key・Value 形式に変換させる。

(2) Shuffle

Map 処理で生成された Key･Value 形式のデータを Key 単位で集約させる。集約したデータは、Reduce 処理を実行するワーカに渡す。

(3) Reduce 処理

Key 単位で集約した Key･Value 形式のデータに対して何らかの処理をする。

MapReduce で処理させたい内容は、Map 処理、Reduce 処理で定義する。

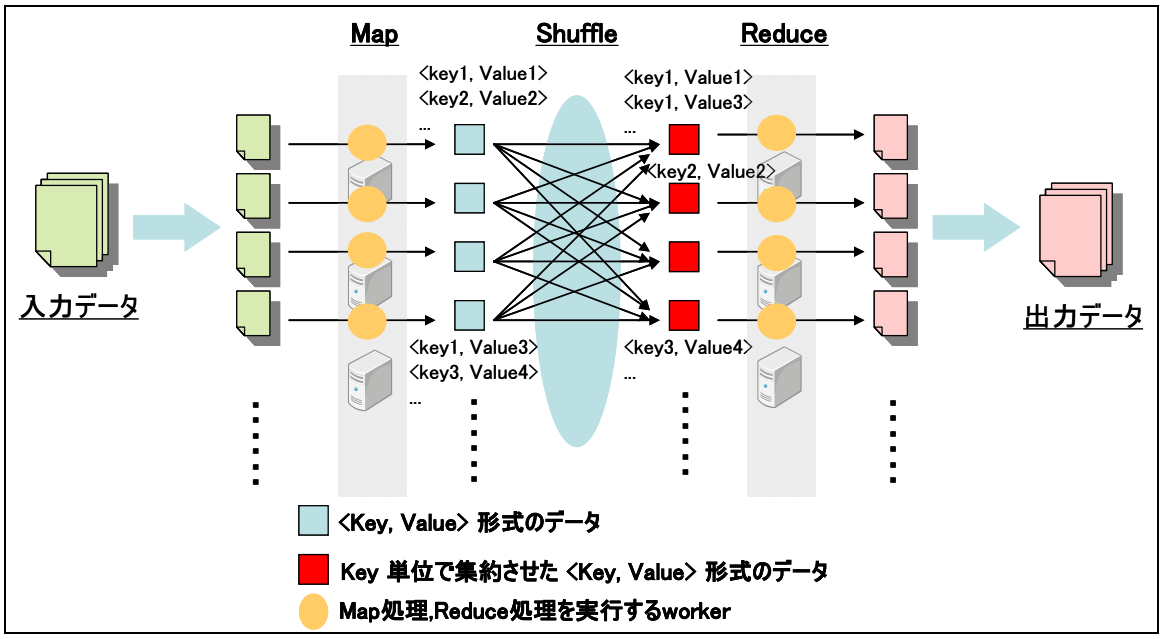


図 8-1 MapReduce の処理フロー概要

8.1.2 MapReduce の特長

MapReduce の特長を以下に示す。

- 膨大なデータの処理に有効

  一台のサーバでは扱いきれない膨大なデータを処理させるときに複数のサーバに分散して処理する MapReduce は有効である。特に、データベースなどで数日以上掛かっていたバッチ処理のようなものを数分～数時間程度で処理させるケースに向いている。

- スケーラビリティ

  一定のデータに対して、Map 処理や Reduce 処理を実行する処理ノードを増やすほど処理時間が線形に短縮するスケーラビリティを持つ。

- 耐障害性

  Map 処理、Reduce 処理を実行する処理ノードが何らかの異常で処理できなくなった場合、他の処理ノードに同じ処理を割り当てるようにスケジューリングされる。そのため、処理ノード単位で異常が発生しても最初から全ての処理をやり直すことなく処理を継続できる。

8.2 MapReduce 適用性検討

本節は、既存の処理を MapReduce に適用する場合の適用フローについて説明する。

具体的な事例を Map 処理、Reduce 処理に分割して、各処理の入出力で必要な Key・Value を定義する。そして、MapReduce に適用させる手順を述べる。

8.2.1 MapReduce 適用のポイント

何らかの処理を MapReduce に適用する場合、3 つのステップより適用する。

(1) 適用したい何らかの処理に対して、処理内の役割に応じて処理単位に分割する。処理全体から役割を抜き出す。役割が処理単位となる。

(2) 全体の処理を並列で実行した場合に、分割した処理の影響範囲が独立・依存するか判断する。(1)で定義した処理を並列に実行した場合、何らかの処理単位が処理の全体や他の並列した処理に影響するか、個別で実行して問題ないか判断する。

(3) どの処理を Map 処理、Reduce 処理に割り当てるか決定する。“Reduce で処理する処理単位→ Map で処理する処理単位”となるような順序の場合、複数の MapReduce ジョブを実行する。Map 処理には、並列して同じ処理を実行する場合に、他方に影響を与えないものを割り当てる。Reduce 処理には、並列して同じ処理を実行する場合に、他方からデータが必要となるものを割り当てる。

以上のポイントで、何らかの処理を MapReduce ジョブで定義する。処理の順番と影響範囲の選択によっては、複数回 MapReduce を実行することになり処理コストが大きくなる。

なお、以下のような処理には MapReduce を適用することは向いていない。

- 再帰的に MapReduce を実行しないと結果を得られない処理

  MapReduce を繰り返し実行しなければならない場合は、処理コストが大きくなる。特に、MapReduce を繰り返し実行しても処理データ量がほとんど変わらないようなケースは、処理コストが大きいと言える。

- 連続した Key を利用した処理

  MapReduce では、Key・Value 形式で情報を保持する。Key を中心に処理されるので、任意の Key の前後関係を参照するような処理を MapReduce で実現する場合は、処理を分散できないため処理コストが高くなる。

8.2.2 MapReduce の適用事例

本節では、MapReduce ジョブに適用する具体的な事例を 2 例述べる。

(1) Web ページの頻出キーワード抽出

Web ページの単語を抽出して、出現頻度が高い単語を抽出する。図 8-2 に MapReduce での処理の流れを示す。Web ページの情報を Map 処理で<出現単語, 1>とカウントする。Shuffle で、出現単語を頭文字で分割・集約する。Reduce 処理で<出現単語, 1>をまとめて<出現単語, 出現回数>となるように集計する。

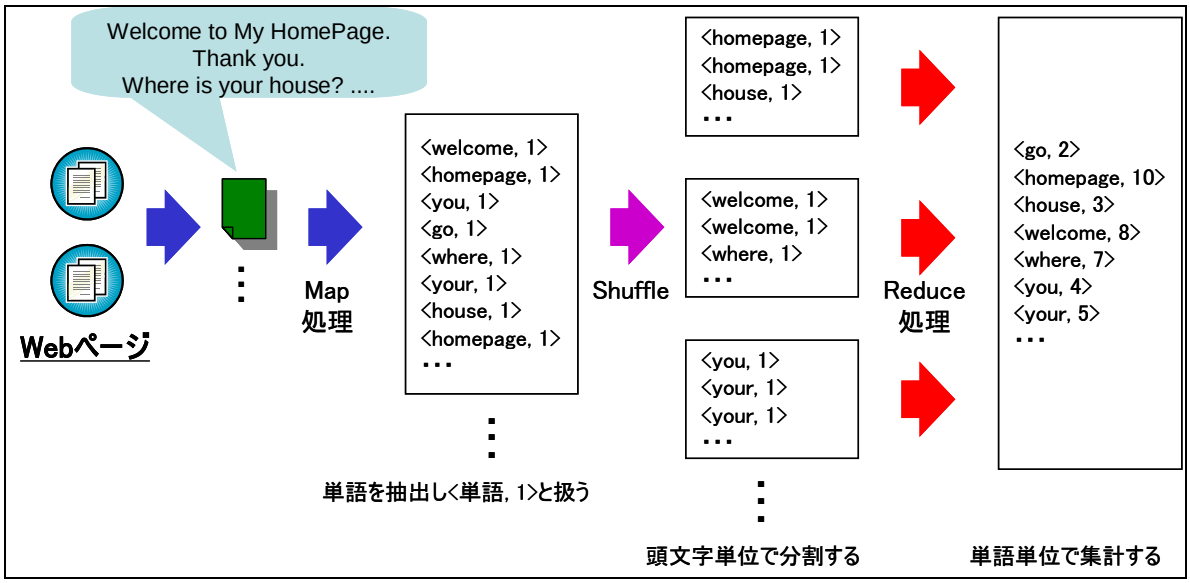


図 8-2 Web ページの頻出キーワード抽出フロー

この処理を応用することで、以下の処理も MapReduce で実現できる。

- Web ページの属性抽出 ： 出現単語に重みをつけ出現割合から属性を決める
- アクセスログ解析 ： どの Web ページにアクセスが多いか分析する

(2) メールフィルタリング

メールログより、スパムメールを送信するメールアドレスのリストを作成するために MapReduce を適用する。図 8-3 に示すように、Map 処理では、メール 1 通単位でメールアドレスやヘッダ・本文と言ったメールデータを評価関数に適用し、スパムメール指数を算出する。Reduce 処理では、メールアドレス単位でのスパムメール指数を集計し、閾値によりフィルタ対象のメールアドレスを決定する。

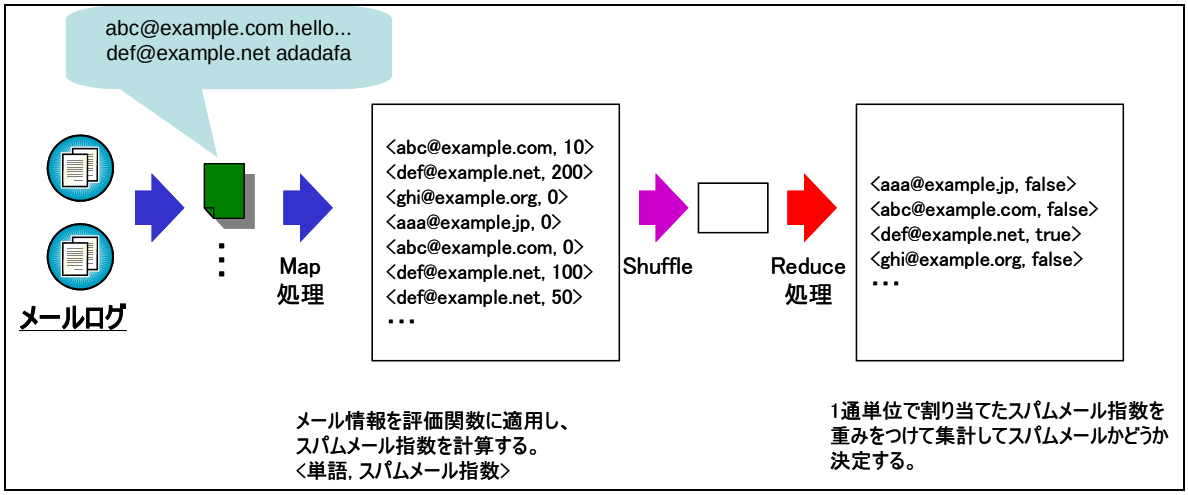


図 8-3 メールアドレスフィルタリング

フィルタリング処理は、Map 処理や Reduce 処理で、個別に評価関数を使用する。フィルタルールを変更する場合は、Map 処理や Reduce 処理の全てを変更する必要はなく、その評価関数部分のみを置き換えるだけで対処できるので、ソースコードレベルで修正量が少ない。また、複数のフィルタルールを組み合わせてフィルタ対象のメールアドレスを決定するルールといった柔軟な処理も容易に実現できる。

8.2.3 では、PageRank 計算を具体例とし、MapReduce ジョブへの適用ポイントを踏まえて適用する流れを具体的に述べる。

8.2.3 MapReduce に適用させる計算モデル

本節では、8.2.1 で述べた事例のような何らかの処理として、図 8-4 に示すような PageRank 計算を例に、処理を MapReduce に適用する。PageRank 計算は、以下のような計算である。

日記やブログを扱っている Web サーバ内の Web ページに含まれるリンクを利用して、アクセス傾向強い Web ページを判断する PageRank 情報を作成する。“有意な Web ページはそのページを参照するリンクも多い”という考えの下、Web ページの重要度を計算する。

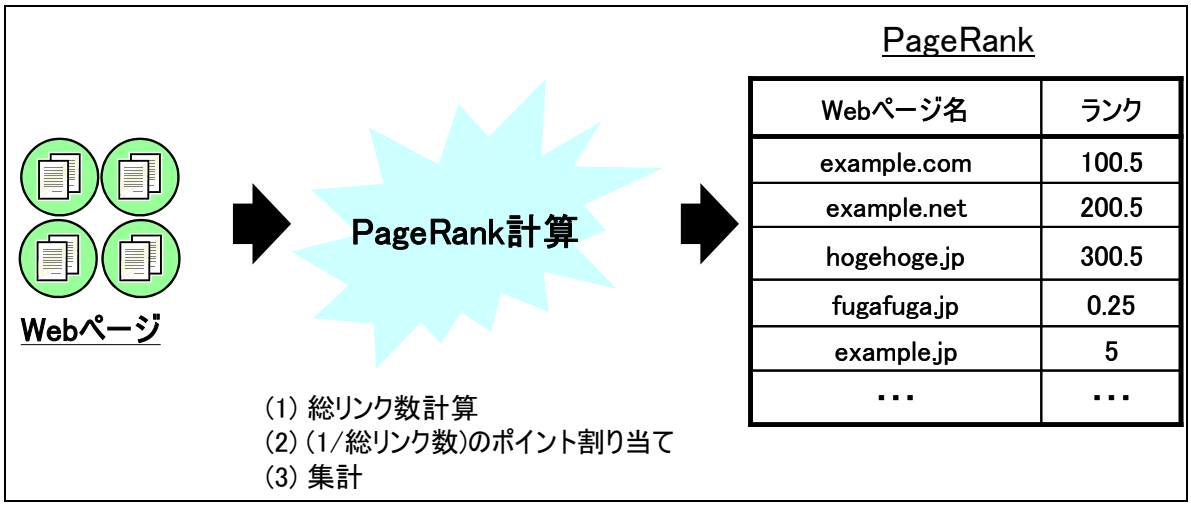


図 8-4 MapReduce 適用モデル

PageRank は、以下の手順で計算する。

(1) Web ページ 1 つ 1 つごとに総リンク数を計算する。

(2) Web ページ 1 つに出現するリンクに対して、“(1/総リンク数)”をポイントとして割り当てる。

(3) (2)で割り当てたポイントを Web ページ単位で集計して PageRank を算出する。

次節以降では、図 8-4 内の“PageRank 計算”を MapReduce に適用させる方法について述べる。

8.2.4 MapReduce 適用観点

8.2.3 で示した PageRank 計算を MapReduce に当てはめる。まず、PageRank 計算を処理単位に分割する。なお、PageRank 計算は、図 8-5 に示す 3 つの処理に分割できる。

- 総リンク数計算
- ポイント割り当て
- ポイント集計

図 8-5 PageRank 計算内容

次に、分割した処理でどのような情報が必要か図 8-5 に追加する。情報を付与した

ものを図 8-6 に示す。

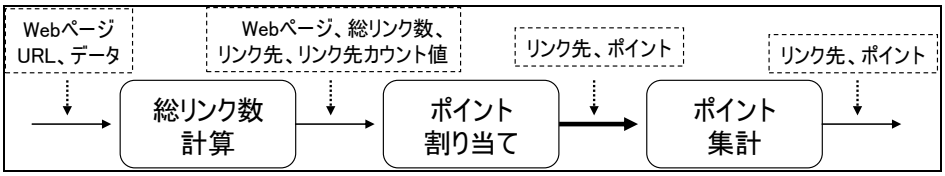


図 8-6 PageRank 計算のデータの流れ

分散処理では、図 8-6 で示した処理が複数の処理ノードで実行される。PageRank 計算では、ポイント集計処理のときに他処理ノードで実行する “リンク先、ポイント”情報が自処理ノードで必要となる。そのためデータの流れは、図 8-7 に示すように他処理ノードの処理にデータを渡し、自処理ノードでは他処理ノードのデータを受け取ることになる。

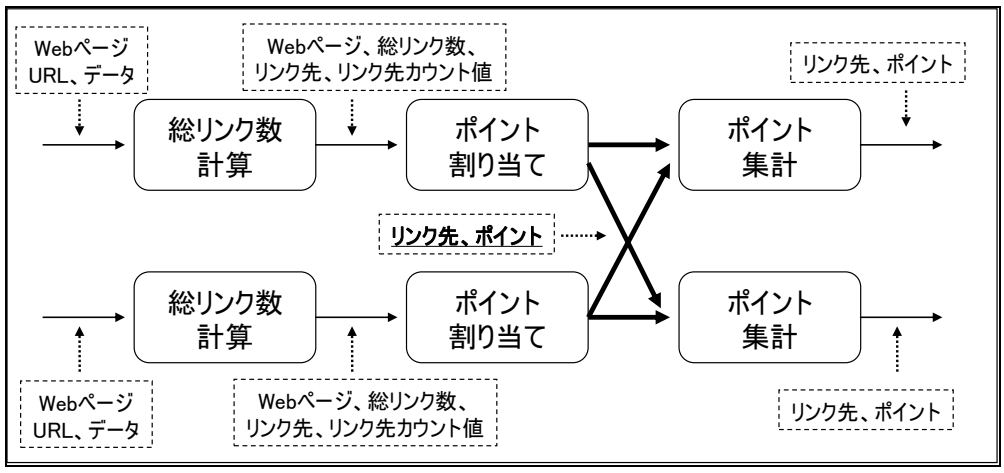


図 8-7 複数の処理ノードを意識した PageRank 計算でのデータの流れ

複数の処理ノードを意識して Map 処理、Reduce 処理への割り当てを決定する。処理割り当ての観点は以下の通りである。

- 処理に他ノードの計算結果が必要な場合や、入力データを特定の単位でまとめる処理は、Reduce 処理に割り当てる
- 他の処理に影響しない独立した処理は、Map 処理に割り当てる

この観点を PageRank 計算に適用すると、“ポイント集計”は他ノードの処理結果を利用し、Key 単位にまとめたデータを処理するので、Reduce 処理に割り当てられる。“総リンク数計算”と“ポイント割り当て”は、各ノードで独立して処理できるため、Map 処理に割り当てられる。以上を整理したものを図 8-8 に示す。

本処理では、Reduce 処理の後に Map 処理が発生することがないため、一回の MapReduce で処理を実現できる。

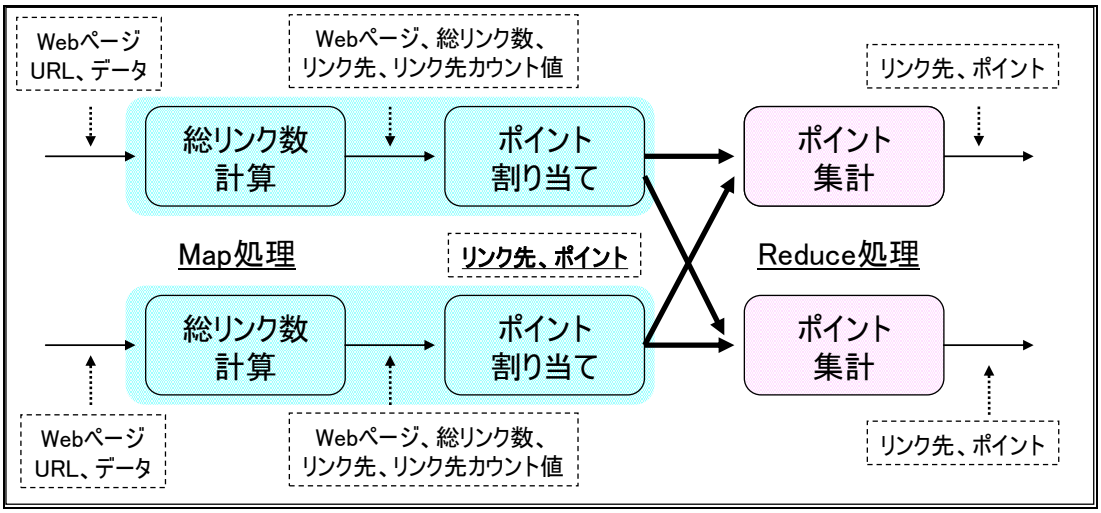


図 8-8 PageRank 計算の処理分割

以上で説明した Map 処理と Reduce 処理への適用方法より PageRank 計算は、図 8-9 で示すような MapReduce での処理フローとなる。

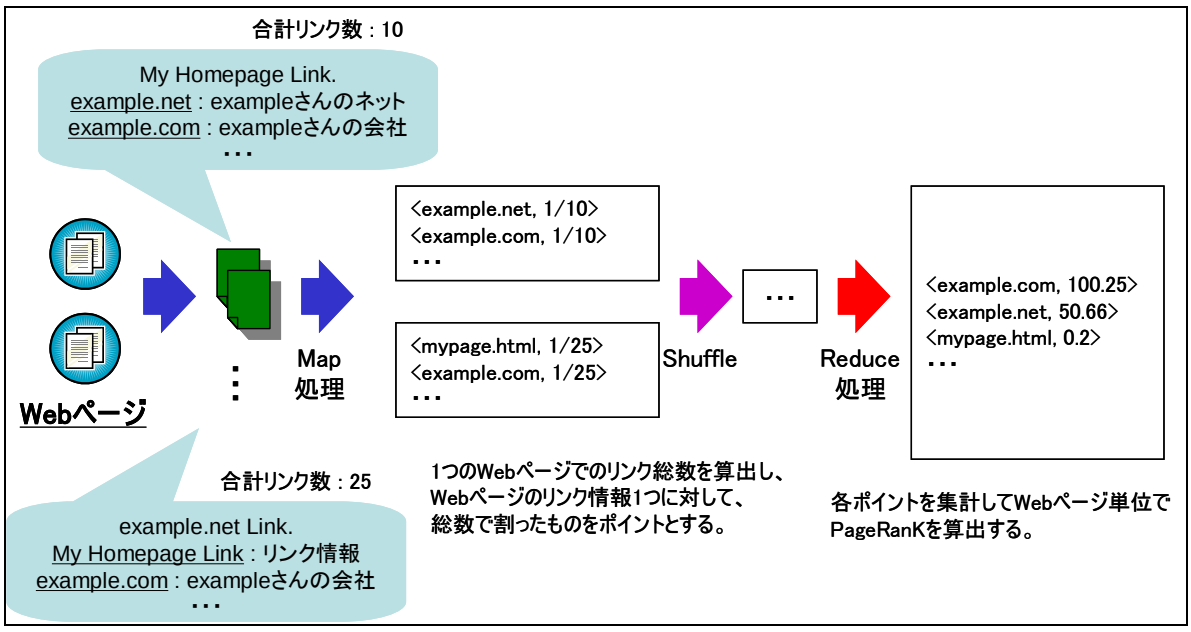


図 8-9 PageRank 計算の MapReduce 適用イメージ

本節では、具体的な処理の MapReduce 適用方法について説明した。なお、PageRank 計算は、本節で説明した処理内容に以下の処理を追加することで、計算結果の精度を

より高めることができる。

- PageRank 計算の MapReduce を繰り返し実行することで、スコアの精度を高める。(例: X 回目の処理は、(X-1)回目のスコアを重みとして与える など)
- Web ページのアクセスログと組み合わせて、特定のキーワードに対してはスコアを補正する。

8.3 Hadoop で実行する MapReduce アプリケーションの作り方

本節は、Hadoop 上で実行させる MapReduce アプリケーションの作り方について述べる。Hadoop 上の MapReduce アプリケーションは、図 8-10 で示すようなポイントをもとに作成する。

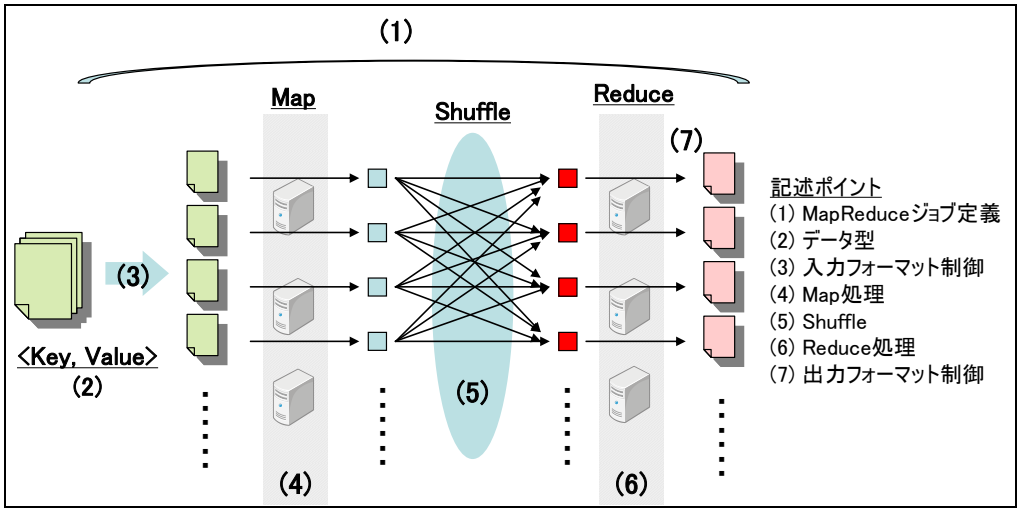


図 8-10 Hadoop 上での MapReduce アプリケーション設計のポイント

- MapReduce ジョブ定義 ：MapReduce としてのジョブ内容の定義を記述
- データ型 ：MapReduce で扱うデータ型を記述
- 入力フォーマット制御 ：MapReduce の入力方法を記述
- Map 処理 ：MapReduce の Map 部分に該当する処理を記述
- Shuffle：Shuffle で利用する Partition に該当する処理を記述
- Reduce 処理 ：MapReduce の Reduce 部分に該当する処理を記述
- 出力フォーマット制御 ：MapReduce の出力方法を記述

Hadoop では、MapReduce フレームワークが Java の API として用意されている。そのため、フレームワークに必要な処理内容を定義すればよい。本節では、8.2.3、8.2.4 で検討した PageRank 計算処理を Hadoop 上の MapReduce フレームワークに適用する例をベースに説明する。

なお、Hadoop で MapReduce アプリケーションを作成するにあたり、2010 年 2 月時点で最新版であるバージョン 0.20.1 の Hadoop を使用することを前提とする。また、以降で述べる MapReduce アプリケーションは、Hadoop 分散ファイルシステム(HDFS)上で管理しているファイルに対して MapReduce ジョブを実行する。

Hadoop の MapReduce フレームワークでは MapReduce ジョブ定義、Map 処理、Reduce 処理の項目を各自で記述しなければならない。 それ以外に関しては、Hadoop でデフォルトとして用意されているものがある。デフォルト以外を使用した場合のみ、各自で定義する。それぞれの項目を記述する上でのポイントについて、以降で述べる。

8.3.1 MapReduce ジョブ定義

Hadoop 上での MapReduce 処理を MapReduce ジョブと呼ぶ。MapReduce ジョブは、8.3.7 までで定義する Map 処理や Reduce 処理といった処理に紐付いた各クラスを 1 つの MapReduce ジョブとして定義する。

MapReduce ジョブを実行するために、ジョブ単位で Job クラス(org.apache.hadoop.mapreduce.Job)を定義する。

Map 処理や Reduce 処理の定義や入出力フォーマットといった一通りの処理内容は、Job クラス内のメソッドで定義できる。しかし、MapReduce ジョブで使用する個別の設定のような項目は、Job クラス内のメソッドで用意されていない。そこで、Job.getConfiguration().set() メソッドを利用して個別に設定情報を定義する。

以降で、MapReduce ジョブ内で記述しなければならない項目についてアプリケーションとして設定すべき項目、実行環境として設定すべき項目、分散処理として設定すべき項目に分けて説明する。

8.3.1.1アプリケーションとして設定すべき項目

MapReduce ジョブを実行するために、アプリケーションとして設定する項目は以下のとおりである。

(1) 入力ファイルのパス

FileInputFormat クラスの setInputPaths メソッドで入力ファイル/ディレクトリのパスを指定する。パスはカンマ区切りで複数指定できる。

(2) 出力先のパス

FileOutputFormat クラスの setOutputPath メソッドで出力先のパス(ディレクトリ)を指定する。ファイルやディレクトリが存在しているパスを指定し

た場合は、FileAlreadyExistsException が呼び出され、MapReduce ジョブは実行されない。

(3) 入出力フォーマット

8.3.6 と 8.3.7 で説明する入力フォーマットや出力フォーマットを定義する。

(4) Map 処理、Reduce 処理のクラス

8.3.2 で説明する Map クラス、8.3.3 で説明する Reduce クラスを定義する。

8.3.1.2 実行環境として設定すべき項目

MapReduce ジョブを実行するために、処理を実行する環境として設定する項目は以下のとおりである。

(1) Map 処理、Reduce 処理での Java に関する設定

各処理に適用する JavaVM オプションは、設定ファイル mapred-site.xml 内か MapReduce ジョブ内に“mapred.child.java.opts”プロパティを記述することで設定できる。このプロパティの利用方法は以下のとおりである。

- ワーカのヒープメモリサイズを変更する(デフォルト 200MB)
- ガベージコレクションに関するログを出力する
- ガベージコレクション実行方式を変更する

(2) Map 処理、Reduce 処理のタイムアウト設定

Map 処理、Reduce 処理は、TaskTracker が生成した Child プロセスで実行される。このとき Child と TaskTracker 間では、処理の進捗状況について通信する。通信が一定期間ない場合は、その処理をタイムアウトと判断して処理を Failed とする。この値は、“mapred.task.timeout”プロパティで設定できる。長時間応答が無いような処理を実行する場合は、長い値を設定する。(デフォルト 60000 ミリ秒)

(3) Map 処理、Reduce 処理を実行する処理ノードのリミット設定

処理ノードでの OS レベルでのリミットにより処理が実行できないことがある。その場合、“mapred.child.ulimit”プロパティとして、リミットに関する設定を記述する。リミットには、ファイル数を開く数や OS レベルでのメモリ割り当てサイズなどがある。

8.3.1.3 分散処理として設定すべき項目

MapReduce ジョブを実行するときに、分散処理として設定すべき項目は以下のとおりである。

(1) Reduce 処理数

Job クラスの setNumReduceTasks メソッドで、Reduce 処理数を指定する。0 を指定すれば、Map 処理の結果をそのまま出力する。Hadoop では、Reduce 処理の数だけファイルが出力される。

Reduce 処理数は、実行する Hadoop 基盤の処理ノード数に応じて設定する必要がある。“Reduce 処理数 < 処理ノード数”である場合は、Hadoop 基盤の処理ノード全てを利用しないため空き処理ノードが存在することになる。Hadoop 基盤のスループット向上のため、処理を分割できるのであれば“Reduce 処理数 >= 処理ノード数”となるように、Reduce 処理数を設定する。

(2) Map 処理数

Map 処理数は、デフォルトでは入力データのブロック数が Map 処理数となる。しかし、“Map 処理数 < 処理ノード数”となる場合、Hadoop 基盤の処理ノード全てを利用しないため、空き処理ノードが存在することになる。

そこで、プロパティ“mapred.max.split.size”を指定して、Map 処理 1 つあたりの最大入力データ量を設定する。1 つの Map 処理での最大入力データ量を指定することで、通常はブロック単位の入力データから分割したデータを Map 処理で扱えるようになる。例えば、最大入力データ量を 16MB(16777216)とする場合、設定方法は以下のようになる。

```
Job job = new Job(); // ジョブの定義
・・・
job.getConfiguration().set(“mapred.max.split.size”, “16777216”);
```

最大入力データ量を指定することで Hadoop 基盤の空き処理ノードを減らし、Hadoop 基盤のスループットを向上させることができる。なお、設定値は“Map 処理数 >= 処理ノード数”となるよう、入力データ量と処理ノード数から算出する。

(3) MapReduce ジョブ処理結果データの多重度

HDFS 上で保持するデータの多重度を設定する。多重度は、プロパティ“dfs.replication”で設定する。

HDFS 多重度のデフォルトは 3 である。HDFS ではファイル単位で多重度を設定できるので、複数の MapReduce ジョブから構成される処理の場合、処理の途中で生成されるジョブ結果の多重度を 3 以下にすることで、レプリケ

ーションに関する処理を軽減し、処理時間を短縮できる。

8.3.1.4 その他設定項目

MapReduce ジョブに関して、個別に設定する項目を定義する。前節の Map 処理数の設定で示した、Job.getConfiguration().set()メソッドを使用すれば、MapReduce ジョブを細かく設定できる。例えば、MapReduce ジョブ実行時にコマンドラインで設定した設定を Map 処理に適用するといった場合などで使用する。

8.3.1.5 MapReduce ジョブ設定に関する記述

ここまでで述べた設定項目を MapReduce ジョブに定義する。

MapReduce ジョブクラス実装項目

MapReduce ジョブでは、以下を実装する。

- org.apache.hadoop.conf.Configured クラスを継承
- org.apache.hadoop.util.Tool クラスを実装
- main メソッド：アプリケーション呼び出し、run メソッドへ引数を渡す
- run メソッド：コマンドラインからの入力加工、MapReduce ジョブ設定
  - MapReduce ジョブ名 ：setJobName
  - 入力フォーマットクラス：setInputFormatClass
  - 出力フォーマットクラス：setOutputFormatClass
  - Map 処理クラス ：setMapperClass
  - Reduce 処理クラス ：setReducerClass
  - 入力データのパス：FileInputFormat.setInputPaths (HDFS 上のファイルを扱う場合)
  - 出力データのパス：FileOutputFormat.setOutputPath(HDFS 上のファイルとして出力する場合)
  - 出力データの Key クラス ：setOutputKeyClass
  - 出力データの Value クラス：setOutputValueClass
  - Map 出力データの Key クラス：setMapOutputKeyClass
  - Map 出力データの Value クラス ：setMapOutputValueClass
- configuration メソッド: ジョブ設定のために必要な場合各自で用意
- execute メソッド：MapReduce ジョブ実行時に必要な場合各自で用意

main メソッドは、MapReduce ジョブを実行するために必ず定義する。run、configuration、execute メソッドは各自で記述する。

8.2.4 の PageRank 計算での MapReduce ジョブ実装イメージは以下の通りになる。

```
17.  public class PageRankJob extends Configured implements Tool {
18.
19.
20.      public int run(String[] args) throws Exception {
21.
22.          execute(configure(args), true);
23.          return 0;
24.      }
25.
26.      public static void main(String[] args) throws Exception {
27.          ToolRunner.run(new Configuration(), new PageRankJob(), args);
28.          System.exit(-1);
29.      }
30.
31.      public Job configure(String[] args) throws IOException {
32.          JobConf jobconf = new JobConf(getConf());
33.
34.          Job job = new Job(jobconf);
35.          job.setMapperClass(PageRankMapper.class);
36.          job.setPartitionerClass(PageRankPartitioner.class);
37.          job.setReducerClass(PageRankReducer.class);
38.          job.setInputFormatClass(TextInputFormat.class);
39.          job.setOutputFormatClass(TextOutputFormat.class);
40.          FileInputFormat.setInputPaths(job, "");
41.          FileOutputFormat.setOutputPath(job, new Path(""));
42.
43.          return job;
44.      }
45.
46.      public int execute(Job job, boolean verbose)
47.          throws IOException, InterruptedException,
48.                   ClassNotFoundException {
49.          job.waitForCompletion(verbose);
50.          return 0;
51.      }
52.
53.  }
```

図 8-11 MapReduce ジョブ設定クラス実装イメージ

本節で説明した内容を定義することで、Hadoop 上で MapReduce ジョブを実行できる。続いて、Map 処理、Reduce 処理と実装する方法について述べる。

8.3.2 Map 処理

Map 処理は、入力データや Map 処理結果を Key・Value 形式で扱う。また、Map 処理の前処理を実行する setup メソッドや、Map 処理の後処理を実行する cleanup メソッドも必要に応じて記述する。(例: Map 処理出力にヘッダやフッタを付与する)

Map クラス実装項目

Map クラスは、以下の項目を実装する。

- org.apache.hadoop.mapreduce.Mapper クラスを継承
- setup メソッド：Map 処理で必要な前処理を記述
- map メソッド：Map 処理を記述

- cleanup メソッド：Map 処理で必要な後処理を記述
- run メソッド：setup, map, cleanup メソッド以外に Map 処理を制御する場合に記述（通常、記述の必要はない）

Map 処理での入力は、8.3.5 で定義する入力フォーマットクラスに沿ってデータ型を定義する（例: TextInputFormat クラスを入力フォーマットにした場合、Key は LongWritable 型、Value は Text 型になる)。また、入力と同等に出力も Key・Value 単位でデータ型を定義する。これは、8.3.1.5 に示す出力データの Key・Value クラス、Map 出力データの Key・Value クラスで定義する。

8.2.4 で検討した PageRank 計算を Map 処理に適用する。PageRank 計算では、Web ページ単位での“総リンク数算出”と“スコア作成”を実行する。“総リンク数算出”は、Map の前処理として実装する。スコアは、前処理でカウントした総リンク数より作成する。図 8-12 に、PageRank 計算での Map クラスの実装イメージを示す。

```
10.  public class PageRankMapper extends
11.          Mapper<LongWritable, Text, WebDataWritable, DoubleWritable> {
12.
13.      private int totalLinks; /* Webページ内の総リンク数 */
14.      private DoubleWritable score; /* PageRank用スコア */
15.
16.      protected void setup(Context context)
17.          throws IOException, InterruptedException {
18.          this.totalLinks = 0;
19.          while( context.nextKeyValue() ) {
20.              this.totalLinks +=
21.                          this.countTotalLinks(context.getCurrentValue());
22.          }
23.      }
24.
25.      private int countTotalLinks(Text value) {
26.          /* Webページ内のリンク数を計算するメソッド */
27.          return count;
28.      }
29.
30.      public void map(LongWritable key, Text value, Context  context)
31.          throws IOException, InterruptedException {
32.
33.          String address = null;
34.          if ( ( address = this.getLink(value) ) != null ) {
35.              WebDataWritable link = new WebDataWritable();
36.              link.wd.address = address;
37.              this.score = new DoubleWritable(1/totalLinks);
38.              context.write(link, score);
39.          }
40.      }
41.
42.      private String getLink(Text value) {
43.          /* Webページの行にリンク情報があるか確認するメソッド */
44.          return link;
45.      }
46.
47.      protected void cleanup(Context context)
48.          throws IOException, InterruptedException {
49.
50.      }
51.
52.  }
```

図 8-12 Map クラス実装イメージ

PageRankMapper クラスでは、Map 処理の前処理として、setup メソッドで、Key となる Web ページのリンク数をカウントする countTotalLinks メソッドを実行する。そして、map メソッドで各リンクにスコアを設定する。

Map 処理では、個別に出力するログ設定にも注意しなければならない。ログは各処理ノードのログ領域に出力される。処理内容を確認する目的であれば、Hadoop ではログに関する設定も簡単に記述できる。しかし大量のデータを扱う場合、ログレベル次第ではログ領域に膨大な量のログを出力する。また、ログ出力によりディスクアクセスが多発し、処理ノードでの Map 処理の処理時間が延びることもある。ログ領域があふれてしまった場合、Hadoop の実装上処理ノードで Map 処理が起動せず Failed 扱いとなるため、デバッグ時以外は極力ログ出力を抑える。

8.3.3 Reduce 処理

8.2 で検討した Reduce 計算を Hadoop で処理する形式に置き換える。Reduce クラスは、Reduce の前処理を実行する setup メソッドや、Reduce の後処理を実行する cleanup メソッドも必要に応じて記述する。

Reduce クラス実装項目

Reduce クラスは、以下の項目を実装する。

- org.apache.hadoop.mapreduce.Reducer クラスを継承
- setup メソッド：Reduce 処理で必要な前処理を記述
- reduce メソッド：Reduce 処理を記述
- cleanup メソッド：Reduce 処理で必要な後処理を記述
- run メソッド：setup, reduce, cleanup メソッド以外に Reduce 処理を詳細に制御する場合に記述（通常、記述の必要はない）

Reduce 処理では、Reduce の結果が出力フォーマット形式に沿ってファイルに出力される。そのため、ファイルにヘッダやフッタ等の情報を入れる場合は、それぞれ setup メソッドや cleanup メソッドにヘッダやフッタを出力させるように、Reduce クラスを実装する。

8.2.4 で検討した PageRank 計算を Reduce 処理に適用する。PageRank 計算の Reduce 処理は、スコアを集計することである。

図 8-13 に、PageRank 処理での Reduce クラスの実装イメージを示す。

```
 9.  public class PageRankReducer
10.      extends Reducer<WebDataWritable, DoubleWritable,
11.                      WebDataWritable, DoubleWritable> {
12.
13.      public void setup(Context context)
14.          throws IOException, InterruptedException {
15.
16.      }
17.
18.      public void reduce(WebDataWritable key,
19.                         Iterable<DoubleWritable> values,
20.                         Context context)
21.          throws IOException, InterruptedException {
22.
23.          double score = 0;
24.          DoubleWritable total;
25.
26.          for ( DoubleWritable val : values ) {
27.              score += val.get();
28.          }
29.          total = new DoubleWritable(score);
30.          context.write(key, total);
31.      }
32.
33.      public void cleanup(Context context)
34.          throws IOException, InterruptedException {
35.
36.      }
37.  }
```

図 8-13 Reduce クラス実装イメージ

PageRankReducer クラスの reduce メソッドでは、Key 単位でデータを集計する。集計した結果を Reduce 処理の結果として扱う。Reduce 処理の入力となるデータは、Key 単位に Iterable に格納され、reduce メソッドに与えられる。

Reduce 処理でも Map 処理と同様にログを出力する際は、設定に注意する。

8.3.4 データ型

Hadoop で MapReduce を実行する場合には、Key・Value 形式でデータを扱う。データは、以下の場面でそれぞれ用意する。

(1) 入力データ：Map 処理への入力
(2) 中間データ：Map 処理の出力、Reduce 処理への入力（Shuffle するデータ）
(3) 出力データ：Reduce 処理の出力

そのため、各入出力に対して Hadoop で扱えるデータ型を指定しなければならない。データ型は、Hadoop 標準で用意されているデータ型の他に、各自でデータ型を作成できる。

以下で、標準のデータ型と各自で作成するデータ型について示す。

8.3.4.1 標準のデータ型

Hadoop では、以下のデータ型を利用することができる。通常利用が想定される文字列や数値、論理演算といった型に加え、配列(一次元、二次元)、コレクションモデルである java.util.Map のデータ型も特に意識せず利用できる。なお標準のデータ型では、複数の型の組み合わせを利用できない。

(1) 文字列型（Text クラス）
(2) 数値型（IntWritable クラス, DoubleWritable クラスなど）
(3) バイト型(BytesWritable クラス）
(4) 論理演算型(BooleanWritable クラス）
(5) 配列型(ArrayWritable クラス, TwoDArrayWritable クラス）
(6) Map 型(MapWritable クラス）

8.3.4.2 新たに作成するデータ型

複数の型を Key として使用するなど、Hadoop では各自で新たにデータ型を作成できる。データ型を Key と Value で利用する場合は、WritableComparable インタフェースを implement する。データ型を Value のみで利用する場合は、Writable インタフェースを implement する。

データ型クラス実装項目

新たなデータ型は、以下の項目を実装する。

- org.apache.hadoop.io.Writable を実装（Value のみで利用する場合）
- org.apache.hadoop.io.WritableComparable を実装（Key と Value で利用する場合）
- set メソッド：オブジェクトに値を格納（引数: 格納したい情報）
- get メソッド：オブジェクトから値を取得（戻り値 ： 格納している情報）
- write メソッド：オブジェクトのシリアライズ（引数: シリアライズしたい情報）
- readFields メソッド：オブジェクトのデシリアライズ（戻り値: デシリアライズされた情報）
- compareTo メソッド：オブジェクト同士の比較（引数: Object クラスのオブジェクト, 戻り値: -1:引数で与えられたほうが大, 0:同じ, 1: 引数で与えられたほうが小)、WritableComparable インタフェースで実装
- toString メソッド：オブジェクトを文字列型に変換
- equals メソッド：オブジェクトが同一か確認（引数: Object クラスのオブジェクト, 戻り値: 論理型 true / false）

また、Map 処理から Reduce 処理に渡すときにデータをシリアライズする。その中で Key をソートする処理がある。このときシリアライズ状態でオブジェクトを比較する Comparator サブクラスも実装できる。これを実装することで Key を比較するときに、デシリアライズする必要がないため、処理時間を短縮できる。

- Comparator サブクラス：シリアライズされたオブジェクトを比較するサブクラス、org.apache.hadoop.io.WritableComparator を継承
- compare メソッド：シリアライズオブジェクトの比較

データ型実装で注意すべき点は、各データを比較する Comparator のサブクラスである。Hadoop での MapReduce ジョブは、Shuffle されたデータを Key によってソートする。Key のソートは、各データモデルの比較処理部を利用するため整列させるルールに沿って実装しなければならない。

8.2.4 で説明した PageRank 計算では、以下のデータ型が必要となる。

(1) Web ページに関するデータ型
(2) スコアに関するデータ型

(1)の Web ページに関するデータ型は、Web ページの情報を属性として保持するため、新しいデータ型とする。(2)のスコアに関するデータ型は、Hadoop 標準データ型である DoubleWritable を使用すれば良い。図 8-14 に、Web ページ用データ型 WebDataWritable の実装イメージを示す。

```
10. public class WebDataWritable
11.     implements WritableComparable<WebData> {
12.
13.     WebData wd;
14.
15.     public WebDataWritable() {}
16.     public WebDataWritable(WebData webdata) { wd = webdata; }
17.     public void set(WebData webdata) {  wd = webdata; }
18.
19.     public WebData get() {  return wd; }
20.
21.     public boolean equals(WebData webdata) {
22.         return wd.equals(webdata);
23.     }
24.
25.     public void write(DataOutput out) throws IOException {
26.         out.writeUTF(wd.address);
27.         out.writeUTF(wd.created);
28.         out.writeUTF(wd.updated);
29.         out.writeInt(wd.size);
30.     }
31.
32.     public void readFields(DataInput in) throws IOException {
33.         wd = new WebData();
34.         wd.address = in.readUTF();
35.         wd.created = in.readUTF();
36.         wd.updated = in.readUTF();
37.         wd.size = in.readInt();
38.     }
39.
40.     public String toString() {
41.         return (wd.address + ", " + wd.created + ", "
42.         + wd.updated + ", " + Integer.toString(wd.size));
43.     }
44.
45.     public int compareTo(WebData webdata) { /* 比較処理を記述 */ }
46.
47.     public static class Comparator extends WritableComparator {
48.         public Comparator() {
49.             super(WebDataWritable.class);
50.         }
51.
52.         public int compare(byte[] b1, int s1, int l1,
53.                            byte[] b2, int s2, int l2) {
54.             /* 比較処理を記述 */
55.         }
56.     }
57. }
```

図 8-14 データ型クラス実装イメージ

WebDataWritable クラスで扱う WebData は、図 8-15 で示すような情報を持つ。比較処理は、Web ページのアドレスや作成日、更新日を利用して比較できるようにする。WebData クラスで定義する変数は、WebDataWritable に直接記述することもできる。

WebData クラスを MapReduce ジョブ以外で利用するかどうかを踏まえて、同一のクラスで扱うかクラスを分割するかを決定する。

```
3.  public class WebData {
4.
5.      String address; // URL
6.      String created; // 作成日
7.      String updated; // 更新日
8.      int size; // サイズ
9.  }
```

図 8-15 WebData クラス実装例

以上で説明した Map クラスや Reduce クラスとデータ型を指定すれば、Hadoop で MapReduce を単純に実行できる。ただし、MapReduce ジョブの実行に関して細かな制御は難しい。8.3.5 以降では、MapReduce ジョブの細かな制御に関する設計について述べる。

8.3.5 Shuffle

Shuffle は、Map 処理結果を Key 単位で Reduce に渡すための処理である。Hadoop の Shuffle は、以下の 3 つの処理から構成される。

- Partition：Reduce 処理を実行する処理ノードに Key や Value の内容に沿ってデータを配置する
- Grouping： 異なる Key を同一の Reduce 処理で扱う
  - Sort：Partition ルールによって Shuffle された Key をソートする

Map 処理を実行した処理ノードで Partition は処理される。デフォルトでは、Key のハッシュコードを Reduce 処理数で剰余したものとなる。Hadoop 標準のデータ型を Key として使用し特別な Partition が必要でなければ、Partition 処理を実装する必要はない。しかし、剰余以外での Partition 処理や、ユーザが作成したデータ型を Key として利用する場合には、以下に示す項目を実装する。

- Sort 処理は、8.3.4 で説明したデータ型クラス内の比較処理部分で処理されるため、特別にソートを実行しない場合は、定義する必要はない。
- Grouping 処理も同様に、特別なソートを実行しない場合は、定義する必要はない。

Partitioner クラス実装項目

Partitioner クラスは、以下のように実装する。

- org.apache.hadoop.mapreduce.Partitioner クラスを継承
- getPartition メソッド：Key や Value と Reduce 分割数を利用して、Key をどの Reduce 処理に渡すか記述

Hadoop の Partition 実装は、Key だけでなく Value も分割に使用できるので、細

かな制御を実現できる。

8.2.4 の PageRank 計算で、Partition する実装イメージを図 8-16 に示す。

```
6.  public class PageRankPartitioner
7.      extends Partitioner<WebDataWritable, DoubleWritable> {
8.
9.      public int getPartition(WebDataWritable key,
10.                             DoubleWritable value,
11.                             int num_reducer) {
12.
13.         char address_head = key.wd.address.charAt(0);
14.         return (address_head % num_reducer);
15.     }
16.
17. }
```

図 8-16 Partitioner クラス実装イメージ

図 8-16 の PageRank 処理では、Web ページのアドレスの頭文字単位で Key を分割させた。アドレスの他の情報や Web ページ作成日・更新日なども Partitoner に使用する場合は、getPartition メソッドを修正する。

Key となる型が int や float といったプリミティブ型のみを使用する場合、Partition 処理に影響は無い。しかし、各自でクラスを定義したものを利用する場合、変数内の値は同じでもハッシュコードが異なることがある。そのため、同じ処理ノードに配置されないことがあるので注意しなければならない。

Grouping クラス実装項目

Grouping に関する実装は以下の通りである。

- org.apache.hadoop.io.RawComparator クラスを実装
- compare メソッド：Key・Value のデータを比較するメソッド。同一の Key として処理させるかどうか記述

8.2.4 の PageRank 計算では、WebData クラスの address のみを利用するため、address が同一なら同じ Key として扱われる。そのため、Grouping の必要はない。

8.3.6 入力フォーマットモデル

Hadoop では、Map 処理へデータを渡すために入力フォーマットクラスを定義しなければならない。Hadoop では、標準で利用できる入力フォーマットクラスと各自で作成した入力フォーマットクラスを定義できる。

以下で、標準の入力フォーマットクラスと各自で作成する入力フォーマットクラスのポイントについて述べる。

8.3.6.1標準入力フォーマット

Hadoop では、標準で以下の入力フォーマットを利用できる。

(1) テキスト形式

テキスト 1 行を入力データとして使用する場合、TextInputFormat クラスを利用する。

TextInputFormat クラスを利用する場合、読み込まれた Key・Value のうち Key はファイルから読み込んだ位置が LongWritable 型で与えられる。また、Value は Text 型で与えられる。特に Key の値が必要でなければ、Value のみを利用する。

(2) シーケンシャルファイル形式

入力データとして、Hadoop 独自のバイナリ形式である SequenceFile を利用する形式である。この場合は、SequenceFileInputFormat クラスを利用する。

8.3.6.2 新しい入力フォーマット

新しく入力フォーマットを作成する場合は、FileInputFormat クラスや Hadoop デフォルトのクラスを親クラスとして継承する。

入力フォーマットクラス実装項目

入力フォーマットクラスは、以下の項目を実装する。

- org.apache.hadoop.mapreduce.lib.input.FileInputFormat クラスを継承
- createRecordReader メソッド：入力データの読み込み(RecordReader)を記述
- isSplitable メソッド：入力データを分割処理できるか制御を記述、圧縮ファイルを入力データに適用する場合以外は記述不要

1 行単位でのデータの読み込み以上に複数行単位でデータを読み込む必要がある場合は、入力フォーマットクラス内にデータ読み込みに関するクラス(RecordReader)を設計する必要がある。以下に、RecordReader 実装項目を示す。

入力データ読み込みクラス実装項目

入力データ読み込みクラスは、以下のメソッド等を実装する。

- org.apache.hadoop.mapreduce.RecordReader クラスを継承
- initialize メソッド：入力となる Key・Value の定義、Key・Value の初期化
- nextKeyValue メソッド：Map 処理へ渡す Key・Value の設定

- getCurrentKey メソッド：現在の Key 情報を渡す
- getCurrentValue メソッド：現在の Value 情報を渡す
- close メソッド：読み込み終了処理を記述

8.2.4 の PageRank 計算では、標準入力フォーマットクラスである TextInputFormat クラスを利用する。細かい制御を実現するために PageRank 計算の入力制御が必要な場合は、図 8-17 に示すようなイメージで入力フォーマットクラスを作成する。

```
13.  public class PageRankInputFormat
14.      extends FileInputFormat<WebDataWritable, Text> {
15.
16.      public RecordReader<WebDataWritable, Text>
17.          createRecordReader(InputSplit split, TaskAttemptContext context)

18.          throws IOException, InterruptedException {
19.
20.          return new PageRankRecordReader();
21.      }
22.
23.      public class PageRankRecordReader
24.              extends RecordReader<WebDataWritable, Text> {
25.
26.          WebDataWritable key;
27.          Text value;
28.
29.          LineReader in;
30.
31.          public void close() throws IOException {
32.              if ( in != null ) { in.close(); }
33.          }
34.
35.          public WebDataWritable getCurrentKey()
36.              throws IOException, InterruptedException {
37.              return key;
38.          }
39.
40.          public Text getCurrentValue()
41.                  throws IOException, InterruptedException {
42.              return value;
43.          }
44.
45.          public float getProgress()
46.                  throws IOException, InterruptedException {
47.              return 0;
48.          }
49.
50.          public void initialize(InputSplit arg0,
51.                                 TaskAttemptContext arg1)
52.                  throws IOException, InterruptedException {
53.
54.          }
55.
56.          public boolean nextKeyValue()
57.                  throws IOException, InterruptedException {
58.              if (key == null) {
59.                  key = new WebDataWritable();
60.              }
61.              if (value == null) {
62.                  value = new Text();
63.              }
64.              return false;
65.          }
66.
67.      }
68.  }
```

図 8-17 入力フォーマットクラス実装例

8.3.7 出力フォーマットモデル

Hadoop では、Reduce 処理の結果を出力する出力フォーマットクラスを定義しなければならない。Hadoop では、標準で利用できる出力フォーマットクラスの他に、各自で作成する出力フォーマットクラスを定義できる。

以下で、標準の出力フォーマットクラスと新しい出力フォーマットクラスのポイン

トについて述べる。

8.3.7.1 標準出力フォーマット

Hadoop では、以下の標準出力フォーマットクラスを利用できる。

(1) テキスト形式: TextOutputFormat クラス

出力結果となる Key・Value をテキスト形式で出力する。なお、出力される Key と Value はタブで区切られている。

(2) シーケンシャルファイル形式: SequenceFileOutputFormat クラス

Hadoop 独自のバイナリ形式である SequenceFile に出力する。複数の MapReduce ジョブで構成されるアプリケーションの場合、SequenceFile を使用することで出力データ量を軽減できる。

8.2.4 の PageRank 処理での出力フォーマットは、Hadoop 標準の TextOutputFormat クラスを利用する。

出力フォーマットで Key もしくは Value を利用しない場合は、NullWritable を宣言する。NullWritable は何もデータを返さない WritableCompareble クラスである。

8.3.7.2 新しい出力フォーマット

出力フォーマットは、FileOutputFormat クラス(または、TextOutputFormat クラス)を継承してクラスを作成する。テキストでの出力では、TextOutputFormat クラスの使用で特に問題ないが、特に以下の機能を実現したい場合は、クラス継承する。

(1) 出力ファイル名を変更する場合

(2) Key・Value での出力形式を変更する場合

(3) 圧縮ファイルとして出力させる場合(圧縮用コーデックが別途必要)

8.3.8 アプリケーションやアプリケーション実行基盤チューニング

色々なパターンで MapReduce ジョブを実行することを考慮して、以下の実装も MapReduce ジョブに追加する。

(1) Map 処理、Reduce 処理に関係するパラメータの設定

MapReduce ジョブ設定内で、Hadoop に関係したパラメータを設定可能にしておくことで、アプリケーションの実行毎に調整できる。

(2) アプリケーションに関係するパラメータの設定

Key を生成する条件をチューニングするように、アプリケーション単位でチューニングできるようなパラメータを検討する。

8.4 MapReduce をラップしたアプリケーション

ここまでで、Hadoop による MapReduce アプリケーション実装方法について説明した。説明した一連の MapReduce 処理を作成することで、Hadoop 上で動作する MapReduce アプリケーションを作成できる。

一方、手元にあるデータを特定の Key で集計するような簡単な処理を実行する場合に、その都度 MapReduce を実行するプログラムを作成することは非効率である。そこで、Hadoop の MapReduce 実装をラップしたソフトウェアを使用し、MapReduce アプリケーションを作成する手間を省く方法がある。

本節では、Hadoop 上で動作する MapReduce 実装をラップしたソフトウェアの機能や実装例を説明する。

8.4.1 Pig

Pig は、Pig Latin と呼ばれる構文を記述することで、Hadoop 上で MapReduce ジョブを実行できる。

Pig では特定の情報を Key として扱い、一致する Key をもつデータを結合させる JOIN 文や、Key を集約させる GROUP 文といった Hadoop 上の MapReduce アプリケーションで実装する場合に複雑である処理に対しても容易に記述できる。以上のような処理の流れを Pig Latin で記述すれば、Pig の実行エンジンにて MapReduce での実行計画を作成して、Hadoop 上での MapReduce ジョブを実行する。

以降では、2010 年 2 月時点において Hadoop コミュニティから公開されている Pig-0.5.0 をベースに説明する。

8.4.1.1 Pig 機能

Pig は、以下の機能を持つ。

リレーション処理

(1) Key の集約（GROUP 文, COGROUP 文）
(2) Key の重複排除（DISTINCT 文）
(3) Key の結合(JOIN 文, JOIN OUTER 文)
(4) Key による分割（SPLIT 文）
(5) データの結合(UNION 文, CROSS 文)
(6) データのフィルタ(FILTER 文)
(7) データの整列（ORDER 文）
(8) データの確認(FOREACH～GENERATE 文)

例えば、Pig Latin では、以下のようにリレーション処理を記述できる。

```
A = LOAD ‘dataa.csv’ AS (id:charaarray, test1:int, test2:int);
B = LOAD ‘datab.csv’ AS (id:charaarray, test3:int, test4:int);
JOINDATA = JOIN A BY id, B BY id;
DISTINCTDATA = DISTINICT JOINDATA;
FILTERDATA = FILTER DISTINCTDATA BY (test1 > test3)
GENDATA = FOREACH FILTERDATA GENERATE id, test1, test2, test4;
ORDERDATA = ORDER GENDATA BY test4 DESC;
```

以上で示したデータを結合する JOIN、任意の属性の重複排除、フィルタリングや整列といった、Hadoop の MapReduce 実装で記述すると記述量が多い処理でも、Pig では数行で記述することができる。

数値演算処理

Key に対して、加減乗除剰余を実行できる。例えば、ある情報に対して特定の値を加算する場合は以下のように記述する。

```
LOG = LOAD ‘score.csv’ AS (id:charaarray, test1:int, test2:int);
SCORELIST = FOREACH LOG GENERATE id, test1+test2, test1-test2;
```

条件判定処理

Key が等しいかどうかなどの比較処理、パターンマッチングを実行できる。また、複数の評価式を組み合わせる(AND, OR, NOT)ことも可能である。

例えば、ある情報を特定の条件でフィルタリングする場合は、以下のように記述する。

```
LOG = LOAD ‘hogehoge.csv’ AS (id:charaarray, score:int);
IDLIST = FILTER LOG BY (score >= 50) AND (NOT score > 100)
```

評価処理

Key の最大(MAX)、最小(MIN)、平均(AVG)、合計(SUM)、集計(COUNT)、結合(CONCAT)という処理も実行できる。

例えば、Key の最大値を求める場合は、以下のように記述する。

```
LOG = LOAD ‘hogehoge.csv’ AS (id:charaarray, score:int);
```

```
IDLIST = GROUP LOG BY id;
MAXSCORE = FOREACH IDLIST GENERATE group, MAX(LOG.score);
```

データ入出力

Pig で扱うデータの入力や Pig での処理結果の出力に関しては、それぞれ LOAD 文と STORE 文を使用する。LOAD 文、STORE 文両方とも、テキストデータを扱う PigStorage、バイナリデータを扱う BinStorage と合わせて選択する。

また、データを画面に表示する DUMP 文も用意されている。

```
DATA = LOAD '/path/to' USING PigStorage(',');
STORE DATA INTO '/path/to2' USING BinStorage(',');
```

HDFS 操作

Pig のシェル上で HDFS 操作に関するコマンドを実行できる。Hadoop のクライアントで利用できる cat,cd,copyFromLocal,copyToLocal,cp,ls,mkdir,mv,pwd,rm,rmf コマンドに加えて、Pig スクリプトを実行する exec, run コマンドも用意されている。

また、実行中の Job を停止する kill コマンド、個別プロパティの値をセットできる set コマンドも用意されている。

```
grunt > cat hogehoge.txt
aaa bbb
ccc ddd
grunt > run pigscript.pig
# Pig 用スクリプトが実行される
```

ユーザ定義関数

Pig の標準に含まれる関数のほかに、ユーザが独自に定義した関数を利用することができる。KEY の評価や変換といった操作について Java で記述したものを用意する。ユーザ定義関数は、Pig 構文内の REGISTER 文でユーザ定義関数を含めた Java アーカイブファイル(jar ファイル)を宣言することで利用できる。

8.4.1.2 Pig 実行方法

Pig は、実行元となるサーバ(Hadoop クライアント)に Pig パッケージを配置して、Java コマンドより実行する。Java コマンド実行後 Pig 用のシェルが起動する。シェル上で Pig 構文の記述や事前に用意した Pig 構文ファイルを読み込ませて実行する。

また、Pig 構文をまとめたファイルを用意してコマンドライン上から Java コマンドの引数として指定して実行する方法や、Java のアプリケーション内に Pig の構文を埋め込むこともできる。

```
# Pig 用シェルを起動する
user $ java -cp $PIGDIR/pig.jar:$HADOOP_CONF_DIR org.apache.pig.Main
grunt > “Pig の構文を入力する”

# Pig 用スクリプト
user $ java -cp $PIGDIR/pig.jar:$HADOOP_CONF_DIR org.apache.pig.Main
script.pig
・・・script.pig の内容によって MapReduce が実行される
```

8.4.2 Hive

Hive は、Pig と同様に HiveQL と呼ばれる SQL ライクの言語を記述することで、MapReduce をラップしたアプリケーションを実行できる。Hive は、RDBMS のようにデータをテーブルで保持する。テーブルに関するリレーション処理は、HiveQL を記述することで実現する。HiveQL の記述内容から、Hive のエンジンにて MapReduce の実行計画を作成し、Hadoop での MapReduce を実行する。

8.4.2.1 Hive 機能

Hive は、以下の機能を持つ。

データ定義

CREATE TABLE (テーブル作成)/ DROP TABLE (テーブル削除)/ ALTER TABLE(テーブル構成変更)といったSQLのデータ定義構文と同等の処理を Hive でも実行できる。

また、Hive 構文内で利用する関数も SQL の関数定義構文と同様に実行できる。

```
CREATE TABLE sample(id STRING, score INT)
 ROW FORMAT DELIMITED FIELDS TERMINATED BY ‘¥0A’
 STORED AS TEXTFILE;
```

データ操作

LOAD 文や INSERT OVERWRITE 文によりローカルファイルシステムや HDFS

上のファイルをテーブル内に格納できる。SQL のようなテーブル内の特定のデータを更新する UPDATE 文やテーブル内の特定のデータを削除する DELETE 文は Hive には存在しない。

```
LOAD DATA INPATH 'test.csv' INTO TABLE sample;
```

リレーション操作

テーブルに関するリレーション処理は、SELECT 文を実行する。SELECT 文の中には、GROUP BY による集約処理、DISTINCT による重複排除処理、SORT BY や ORDER BY によるデータの整列処理がある。

また、テーブル同士を結合させる JOIN 文や UNION 文、副問い合わせ文も実行可能である。

```
SELECT DISTINCT a.id, b.score FROM sample a JOIN sample2 b ON (a.id =
b.id) WHERE b.score > 80 LIMIT 10;
```

```
SELECT id, score FROM (
  SELECT name AS id , point AS score
  FROM sample3
  ) result
```

8.4.2.2 Hive 実行方法

Hive は、Hive 実行元となるサーバ(Hadoop クライアント)に Hive 用の実行バイナリを配置して、Hive のシェルスクリプトより起動する。

Hive のテーブル操作履歴は、metadata と呼ばれるファイルに格納される。この metadata は、Hive 実行元のサーバにファイル形式で、または別途用意したデータベースに保存することができる。

```
user $ cd $HIVE_HOME
user $ ./bin/hive          ← Hive 用のシェルを起動する
hive > create table table_name (col1 INT, ・・・) ;
hive > ・・・
```

8.4.3 Pig や Hive を適用させる場合のポイント

Pig や Hive を適用するポイントは、MapReduce アプリケーションをスクラッチで実装する場合に掛かる作業時間を短縮できるかどうかである。特にスクラッチで実装する場合とは違い、複雑であるリレーション操作(JOIN, UNION, GROUP)を Pig や Hive では既に実装されているので作業時間を短縮できる。

以上より、Pig や Hive を適用するポイントは次の通りである。

- 複雑なリレーション処理であること
- 処理内容が単純であること
- 扱うデータの型(モデル)を各自で作成する必要があること

即座にデータを MapReduce で処理させる場合に簡単に実行できることが、Pig や Hive の利点である。また、処理させたい内容をその都度変更するような場合に有効である。

一方、スクラッチで MapReduce アプリケーションを作成する場合は、MapReduce で扱う Key や Value に関して細かな制御を実現できる。また、MapReduce の実行に関しても細かい制御を実現できる。そのため、定期的なバッチ処理で処理するデータの型が決まっているような場合には、スクラッチで記述するほうが良い。

8.5 MapReduce アプリケーション作成の踏み込んだポイント

本節では、Pig や Hive を利用せずスクラッチで MapReduce アプリケーションを作成する場合の注意点について述べる。

8.5.1 Map 処理・Reduce 処理分割

Hadoop では、Map・Reduce のステージ単位で処理単位を分割する。MapReduce ジョブ実行時に Hadoop 基盤内で空き処理ノードを減らすことで、Hadoop 基盤全体のスループットを高められる。

8.5.1.1 Map 処理分割

Hadoop では、デフォルトでは入力データのブロックサイズで Map 処理が分割される。ブロックサイズのデフォルトは 64MB である。特に設定を変更しなければ 1 つの Map 処理では、64MB＋αB の入力データを処理する。なお、ブロックの最後が途中で区切られていたときに次のブロックからデータを読み出す量をαバイトとする。

処理分割の設定を変更する場合は、以下の方法で対処する。

・ 分割パラメータによる変更

FileInputFormat クラスの setMaxSplitSize メソッドで変更できる。このメソッドは、“mapred.max.split.size”プロパティの値を変更するもので、1 つの Map 処理あたりの最大データ量を指定できる。“mapred.max.split.size”が入力データのブロックサイズより小さいときに入力データとなるブロックを指定した値で分割する。

・ ブロックサイズによる変更

ブロックサイズを変更して HDFS に格納することで、1 つの Map 処理の入力データ量を変更できる。ただし、この方法は HDFS で管理するブロックの数が変化する。ブロックサイズは、“dfs.block.size”プロパティの値を変更する。なお、1 度 HDFS に格納したデータのブロックサイズを変更する場合は、HDFS のデータをコピーする、再度 HDFS にデータを格納することで対処する。

8.5.1.2 Reduce 処理分割

Reduce 処理は、“mapred.reduce.tasks”プロパティの値を使用する。MapReduce アプリケーションでは、Job クラスの setNumReduceTasks メソッドで、このプロパティの値を設定できる。この値を設定しないと Reduce 処理分割数は“1”が設定されることになるため、Reduce 処理実行時に 1 台の処理ノードでしか処理されず、分散処理されない。

8.5.1.3 分割した処理の確認方法

Map 処理、Reduce 処理の分割状況は、以下の方法で確認できる。

・ JobTracker の Web インタフェース

Web インタフェースでは、Map 処理・Reduce 処理の処理時間や処理単位のカウンタ値を一覧で確認できる。また、処理の平均処理時間、各ステージの最長・最短処理情報と言った分析結果を確認できる。

・ JobTracker のログ

JobTracker のログディレクトリ内に、MapReduce ジョブの実行履歴を記録したヒストリファイルが存在する。ヒストリファイルには時系列に処理の実行履歴などが記録される。ヒストリファイルを解析することで各処理ノードでの処理実行状況を判断できる。

・ MapReduce で使用する API

Job クラスに getTaskCompletionEvent メソッドがある。MapReduce ジョ

ブ実行後に、getTaskCompletionEvent メソッドを利用すると処理ノードへの処理割り当て状況や処理実行時間を確認できる。

8.5.2 処理によるログ出力の内容と出力場所

Map 処理・Reduce 処理に関するログは、各処理を実行した処理ノードのログ領域に記録される。ログは、3 種類ある。

- stdout：Map 処理、Reduce 処理を定義した際に“System.out.println”など標準出力させるメソッドを利用した場合に記録される。また、各処理で設定した JavaVM のガベージコレクションのログも stdout に出力される。
- stderr：Map 処理、Reduce 処理を定義した際に“System.err.println”など標準エラーに出力させるメソッドを利用した場合に記録される。また、各処理で発生したエラーに関するスタックトレースも記録される。
- syslog：各処理でログ用 API である Log4J などによって出力されたログを記録する。Log4J は各タスクの実行記録で利用されている。また、アプリケーションで Log4J を利用することもできる。

どのログを使用する場合でも、MapReduce で処理するデータ量によっては大量のログが出力される。そのため安易にログを出力するような記述はしない。また、出力したいメッセージ内容に応じてログレベルを変更する。

各処理ノードで使用するログ領域の容量が一杯になり新たにデータを書き込めなくなった場合、その処理ノードに割り当てられた処理は全て Failed となる。また、各処理ノードのログ領域に出力された MapReduce に関するログは、デフォルト 24 時間で削除される。

8.5.3 MapReduce ジョブで使用するカウンタの扱い方

Hadoop では、1 つの MapReduce ジョブに対してカウンタが利用できる。カウンタを利用することで、処理データ量の把握や各ワーカへのタスク割り当て状況を判断することができる。

Hadoop 標準に含まれているカウンタは、以下の通りである。

- Job Counters

  Map 処理・Reduce 処理を起動した数、Map 処理で、ローカルにあるデータを利用した Map 処理数、ラック内にあるデータを利用した Map 処理数といったデータの配置と Map 処理の起動の関係も記録される。
- FileSystemCounters

  HDFS に読み書きしたデータ量、ローカルディスクに読み書きしたデータ量が記録される。処理単位やジョブ全体での合計を確認できる。
- Map-Reduce Framework

Map 処理、Reduce 処理での入出力レコード数、入出力データ量、ディスクに書き出したレコード数を記録する。(2)の“FileSystem Counters”のカウンタと同様に処理単位やジョブ全体での合計を確認できる。

Hadoop 標準のカウンタのほかに、ユーザが独自で定義したカウンタも利用できる。Context クラスの getCounter メソッドを利用することで、各自でカウンタを利用できる。カウンタは、以下に示すような記述を処理に追加することで利用可能となる。カウンタを適用するクラス名にグループ名、カウント名を記述する。

```
/* 例：Map メソッドでカウンタを利用する */
void map(Key Key・Value value, Context context) {
   context.getCounter(TestMapper.class.getSimpleName(),
                          “カウンタ名”).increment(値)
}
```

カウンタの値は、JobTracker の Web インタフェースや Job 実行時の画面で確認できる。

8.5.4 MapReduce で利用する静的データの扱い方

MapReduce ジョブでは、入力データに対してルックアップテーブルのような静的情報を参照して処理するケースも想定される。このとき静的データの扱い方は、状況に応じて変更しなければならない。

静的データを扱う場合の観点は、以下の 2 つである。

- 静的データのサイズ：数 MB 程度、数 MB-数百 MB, GB 以上
- 静的データの分割：データを分割して扱えるかどうか

この観点を元に、静的データを利用する方法を以下に示す。

(1) ローカルディスク：各処理ノードのローカルディスク内に静的データを保持する。
(2) memcached：分散メモリアプリケーションである memcached を各処理ノードに設置してメモリ上で静的データを保持する。
(3) RDBMS：Hadoop 基盤とは別に RDBMS を用意してその中で静的データを保持する。
(4) 分散 Key・Value ストア：Hadoop 基盤上に構築できる分散 Key・Value ストア(HBase、Hypertable)を用意して、その中で静的データを保持する。

(5) MapReduce アプリケーション：MapReduce アプリケーション内に静的データを用意し、MapReduce ジョブ実行時にアプリケーションと一緒に配布する。

以上の方法より、静的データを適用する場合の比較を表 8-1 に示す。

表 8-1 静的データ利用に関する比較

No.	静的データサイズ	数 MB 程度 (小規模)	数 MB ～ 数百 MB (中規模)		GB 以上 (大規模)	
	分割有無	不問	可能	不可能	可能	不可能
1	ローカルディスク	○	△	△	×	×
2	memcached	○	○	×	△	×
3	RDBMS	×	×	○	×	○
4	分散 KV ストア	×	△	△	○	△
5	MapReduce-AP	○	×	△	×	×

○、△、×は、静的データの管理、MapReduce ジョブ実行時の性能という観点で相対的に付与している。以下で、静的データサイズによる評価について述べる。

まず、小規模の静的データの場合、RDBMS や分散 KV ストアは、導入や管理コストが大きくなるため利用には向かない。それ以外の方法は特に問題ない。

中規模程度の静的データでは、データ自体が分割可能であるならばローカルディスクや memcached、分散 KV ストアを利用することが考えられる。データの分割が難しい場合は、RDBMS を利用することが良い。

大規模の静的データでは、データ分割できるならば分散 KV ストア、分割できないのであれば RDBMS を利用する。

8.5.5 Map 処理結果の Reduce 処理へのデータの渡し方

Hadoop では、Map 処理の結果を Partition させることで Reduce 処理の入力に影響を与える。理想的な Reduce 処理は、入力データが全ての処理ノードに均等に配分することである。しかし Reduce 処理は、Map 処理とは違い定義した Key によって処理ノードに配置されるためデータに偏りが発生する。また、均等にデータを配置した場合でも Reduce 処理内容によっては逆に偏らせたほうが良いこともある。
Partition による分割方針について以下で述べる。

- Reduce 処理内容が Key 同士、Value 同士の依存性の影響が強い場合
  データの平均や最大・最小など Reduce 処理を実行するときにデータがまとまっていなければならない場合は、処理ノードごとにデータが偏るのは問題

ない。それでも偏りを抑えたい場合は、8.5.1 で説明した Reduce 処理数を多くして、偏りの影響を抑える。

- Reduce 処理内容が Key 同士、Value 同士の依存性の影響が無い・弱い場合、この場合は、各処理ノードに均等にデータを配置できるように Partition させる。

8.6 まとめと課題

本資料では、何らかの処理を MapReduce に適用する場合の適用までの流れを適用のポイント、Hadoop で動作させる MapReduce アプリケーションの作成方法を説明した。例として PageRank 計算での MapReduce への適用を示した。また、Pig や Hive と言った MapReduce を包含したツール、MapReduce アプリケーションを作成する場合の注意点について述べた。

Hadoop での MapReduce アプリケーションの課題は、アプリケーションで利用する静的データの管理方法である。扱う静的データの容量や MapReduce ジョブの処理データ量に応じて、管理方式を決定しなければならない。

また、単純に動作する MapReduce アプリケーションを作成するだけでなく、ジョブ実行時間の短縮や、Hadoop クラスタの資源を十分利用するように設定しなければならない。

9 Hadoop 基盤の性能チューニング検討

本章では、クラウド型分散処理環境として Hadoop 基盤の性能チューニングを検討する。分散処理として MapReduce を Hadoop 上で実行する。

Hadoop の性能チューニングとして、以下の内容を報告する。

- Hadoop 上の MapReduce 処理特性の把握

  MapReduce を Hadoop 上で実行するときの動作特性や Hadoop 基盤のリソース使用状況について把握する。
- Hadoop 基盤としての性能チューニング

  MapReduce 処理特性を踏まえて、Hadoop 基盤の性能に関するチューニング方針を述べる。
- MapReduce ジョブに関する Hadoop 性能チューニング

  Hadoop 上の MapReduce に関するボトルネックポイントを考慮して、Map 処理や Reduce 処理の分散をチューニングする。
- MapReduce ジョブの処理時間の見積もり方法

  最初に大量のデータで MapReduce を実行すると処理がいつ完了するか判断できないため、少量データによる MapReduce ジョブ実行結果より、大量データの処理時間を見積もる方法を示す。

クラウド型分散処理としてのあるべき姿や課題について示し、それを Hadoop 基盤に置き換えたときに検討することや対処方法について述べる。なお Hadoop 基盤は、ハードウェアスペックが混在する処理ノードより構成されていることを前提とし、環境制約を考慮したチューニング方法を検討する。

9.1 あるべき姿と課題

本節では、クラウド型分散処理での性能に関して、あるべき姿と課題や問題について述べる。

9.1.1 クラウド型分散処理環境の性能定義

クラウド型分散処理環境の処理性能を定義する。性能として以下の 2 点を挙げる。

(1) 処理に要する時間

分散処理に要する処理時間が短いほど性能が良いとする。

(2) 処理を実行するノードの利用状況

分散処理を実行するときに、処理ノードの利用率が 100%に近いほど分散処理されていると判断し、性能が良いとする。

9.1.2 混在するクラウド型分散処理環境でのあるべき姿

クラウド型分散処理アプリケーションは、クラウド上の CPU やメモリ、ディスクといったリソースを利用して処理を実行する。

クラウド型分散処理では、表 9-1 に示すような確認すべき項目がある。

表 9-1 クラウド型分散処理の性能として確認すべき項目

No.	項目	レイヤ	対象	確認内容
1	システム基盤	物理レイヤ	CPU、メモリ、ディスク、ネットワーク	クラウドを構成するサーバついて性能を把握する
2		アプリケーション実行基盤レイヤ	OS、アプリケーション実行基盤	OS やアプリケーション実行基盤に関する性能や特性を把握する
3	アプリケーション	アプリケーションレイヤ	アプリケーション	アプリケーションの性能を把握する

まず確認しなければならないことは、利用するクラウド型分散処理環境がどのような特性を持っているのか把握することである。図 9-1 に示すようなクラウド型分散処理環境のリソースを十分利用していない場合では、クラウド型分散処理特性を把握することで、クラウド型分散処理環境のリソースを十分活用する設定ができるようになる。特に、クラウド型分散処理特性を把握することは、ハードウェアスペックが混在する環境では性能を向上させるため重要である。

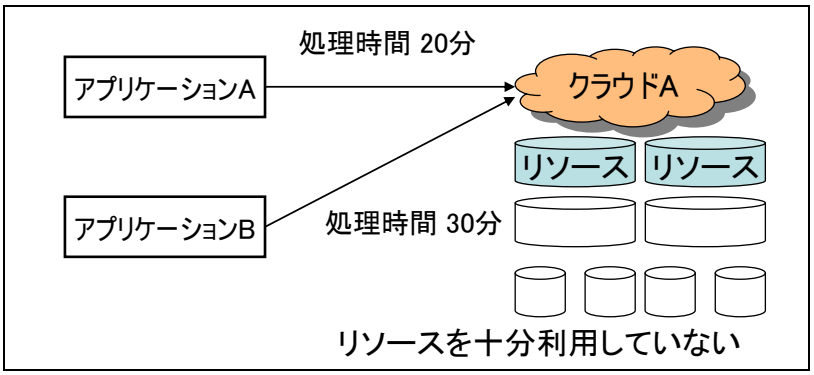


図 9-1 リソースを活用していないクラウド型分散処理環境

次に、クラウド型分散処理基盤を性能に関してチューニングする。これにより、図 9-2 に示すように全ての処理ノードを利用し、処理時間を短縮できる。

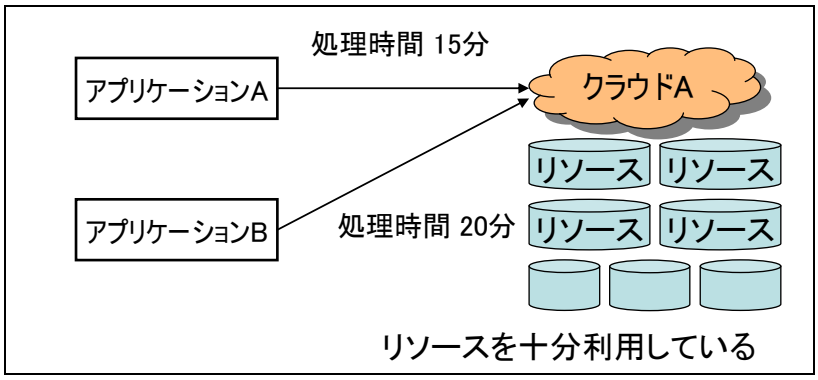


図 9-2 クラウド型分散処理環境チューニングの効果

また、チューニングされたクラウド型分散処理環境では、アプリケーションの処理時間を見積もることができる。処理時間を見積もる理由は、一度に大量のデータを実行する場合に処理がいつ完了するか見通しが立たないためである。そのため、最初に少量のデータを実行して、そこで得られた情報より処理時間を見積もる。

さらに、ハードウェアスペックが混在するクラウド型分散処理環境で性能を向上させるために、スペックの差が影響とならないように処理を分割する。そのためには、ハードウェアのスペックに応じた設定やチューニングがされていなければならない。

以上を整理すると、表 9-2 のようなあるべき姿となる。

表 9-2 クラウド型分散処理の性能に関するあるべき姿

No.	あるべき姿
1	クラウド型分散処理基盤の特性を把握している
2	性能に影響するチューニングポイントが分かっている

このあるべき姿を実現するために、課題や問題が存在する。次項でその課題や問題について述べる。

9.1.3 混在する環境での問題や課題

本項では、前項で述べた混在環境でのクラウド型分散処理であるべき姿を達成するための問題や課題について述べる。

9.1.3.1 ハードウェアスペック混在の定義

まず、ハードウェアスペックが混在するポイントについて、表 9-3 に示す。クラウド型分散処理では、分散処理を実行するノードに混在環境が存在することを想定しておく必要がある。

表 9-3 ハードウェアスペックの混在定義

No.	混在部位	備考
1	CPU モデル名(コア数、クロック周波数、キャッシュサイズ、個数)	
2	チップセット	チップセット単体での性能評価は、他の項目でカバーできるため実施しない。
3	メモリ（規格、容量)	メモリ規格による影響は、CPU 処理性能内で扱う。
4	ディスク（規格、容量、台数)	
5	ネットワークインタフェース(速度)	

これらのハードウェアスペック混在がクラウド型分散処理基盤にどう影響するかを確認しなければならない。

9.1.3.2クラウド型分散処理の特性

クラウド型分散処理の処理フローを確認し、どのリソースをどのように利用するかを把握することで、性能向上に関するアプローチを実現できる。

アプローチとして、処理基盤やアプリケーションの分散処理に関して性能向上のためにチューニングする。また、9.1.3.1 で示した、ハードウェアスペックが異なる環境による制約が処理に与える影響についても考慮する。

9.1.3.3問題や課題を解決するためのアプローチ

以上で述べたクラウド型分散処理の性能に関して、混在する環境や処理特性について表 9-4 に示すアプローチで進める。

表 9-4 性能に関する課題や問題解決アプローチ

No.	課題
1	クラウド型分散処理特性の把握
2	ハードウェアスペックが混在するクラウド型分散処理基盤の影響把握
3	クラウド型分散処理基盤のチューニング検討
4	アプリケーションの処理に関するチューニング検討

以上のアプローチを実行するために、まず前提条件を定義する。

9.2 前提条件

本節では、「表 9-2 クラウド型分散処理の性能に関するあるべき姿」で示した内容を実現するために、前提条件を定義する。

9.2.1 混在するクラウド型分散処理環境構成

混在するクラウド型分散処理環境構成は、表 9-5 に示すような環境を前提とする。

表 9-5 混在するクラウド型分散処理環境構成

No.	前提条件
1	クラウド型分散処理環境を構成するサーバのスペックは同一でない。 CPU やメモリ、HDD 容量や台数は各サーバで異なるとする。
2	クラウド型分散処理環境を構成するサーバ台数は、あらかじめ決まっているとする。

No.1 は、表 9-3 で定義したハードウェアスペックが混在するクラウド型分散処理環境を利用することを意味する。No.2 は、クラウド型分散処理の途中で環境構成が変化しないことを意味する。

本章で使用するクラウド型分散処理環境は、表 9-6 で示す 5 種類のサーバから構成されている。

表 9-6 混在するクラウド型分散処理環境のスペック一覧

No.	名称	CPU	メモリ	ディスク	NIC	台数	ベンダ名 型番	発売時期
1	S1	Xeon E5504 2GHz 4 コア	6GB	SAS 300GB 2 台	1Gbps	18	HP DL360G6	2009 年 4 月
2	S2	Xeon E5345 2.33GHz 4 コア×2	8GB	SAS 146GB 2 台	1Gbps	4	HP DL380G5	2008 年 1 月
3	S3	Xeon 5148 2.33GHz 2 コア	2GB	SAS 72GB 2 台	1Gbps	16	HP DL360G5 AH480A	2006 年 10 月

No.	名称	CPU	メモリ	ディスク	NIC	台数	ベンダ名 型番	発売時期
4	S4	Core 2 Duo T9400 2.53GHz 2 コア	2GB	SATA 250GB 2 台	1Gbps	50	NEC Express5800	2009 年 1 月
5	S5	Xeon X5460 3.16GHz 4 コア	6GB	SAS 146GB 2 台	1Gbps	8	HP DL360G5 AK839A	2008 年 2 月

9.2.2 クラウド型分散処理環境のためのソフトウェア

クラウド型分散処理として Hadoop を使用する。Hadoop は、分散ファイルシステムである HDFS(Hadoop Distributed File System)と分散処理フレームワークである MapReduce で構成される。以降、本章で述べるクラウド型分散処理基盤は、Hadoop 基盤を指すこととする。アプリケーションは Hadoop 上で動作する MapReduce アプリケーションを指すこととする。

本章では、2010 年 2 月中旬に最新版であった Hadoop のバージョン 0.20.1 を取り上げる。

9.2.3 Hadoop 基盤での性能定義

9.1.1 で示した“クラウド型分散処理における性能”を“Hadoop 上での MapReduce 処理における性能”として扱う。

(1) 処理に要する時間

1 つの MapReduce ジョブ処理時間とする。MapReduce ジョブ処理時間が短いほど良いとする。

(2) 処理ノードの実行状況

1 つの MapReduce ジョブを実行するときに、全ての処理ノードで処理を実行しているほど良いとする。

以上の 2 つを達成するほど分散処理として性能が良いとする。

9.3 Hadoop 基盤の性能に関する特性の把握

本節では、Hadoop 基盤の特性を判断するための確認ポイントや性能特性、ハードウェアスペックが混在することによる影響、性能特性を判断するためのベンチマークについて述べる。

9.3.1 Hadoop 上での MapReduce 処理概要

MapReduce は、データを Key・Value の形式で管理し、任意の入力データを複数に分割して分散処理するプログラミングモデルである。

MapReduce は、表 9-7 に示すような 3 つのステージで分散処理を実現する。

表 9-7 MapReduce 処理ステージ

No.	ステージ名	処理概要
1	Map	分割された入力データに対して、各自で定義した処理を処理ノード上で実行する。
2	Shuffle	Map 処理結果を各自で定義した条件によって振り分ける。また、振り分けたデータを Key でソートする。
3	Reduce	Shuffle されたデータに対して、各自で定義した処理を処理ノードで実行する。そして、実行結果を出力する。

Hadoop 上の MapReduce は、図 9-3 に示すように Map 処理や Reduce 処理を実行する TaskTracker と TaskTracker を制御する JobTracker から構成されている。Map や Reduce 処理は、1 台の TaskTracker で複数同時に実行できる。

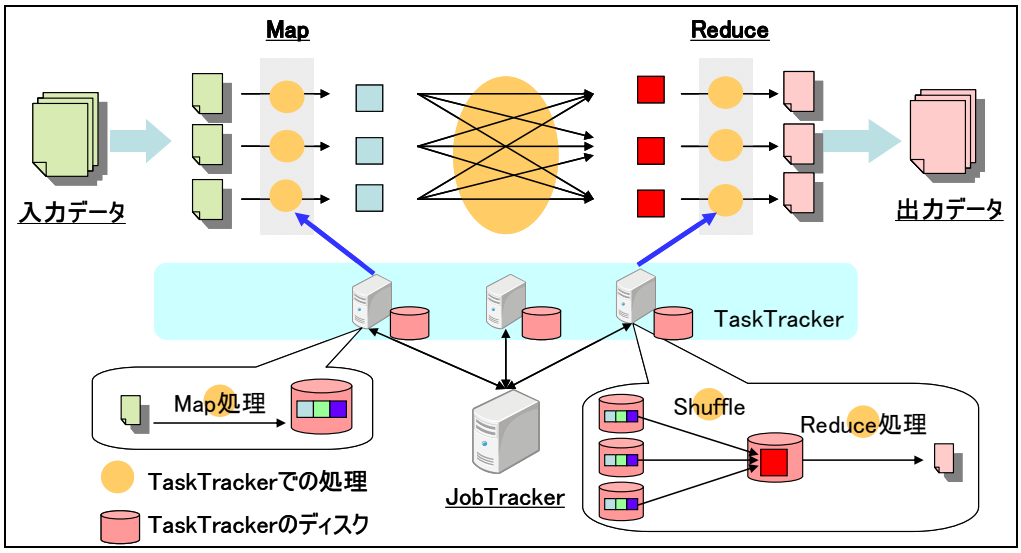


図 9-3 Hadoop 上での MapReduce 処理の流れ

また、Hadoop 上での MapReduce は、HDFS と連携する。HDFS は、メタ情報を管理する NameNode とデータをブロック単位で保持する DataNode から構成される。Hadoop 上での MapReduce は、図 9-4 に示すように入力データの読み出しと出力データの書き込みを HDFS に対して実行する。

DataNode と TaskTracker は、同一の処理ノード上に起動したものを利用する。処理ノードは、DataNode と TaskTracker を起動しているサーバのことである。

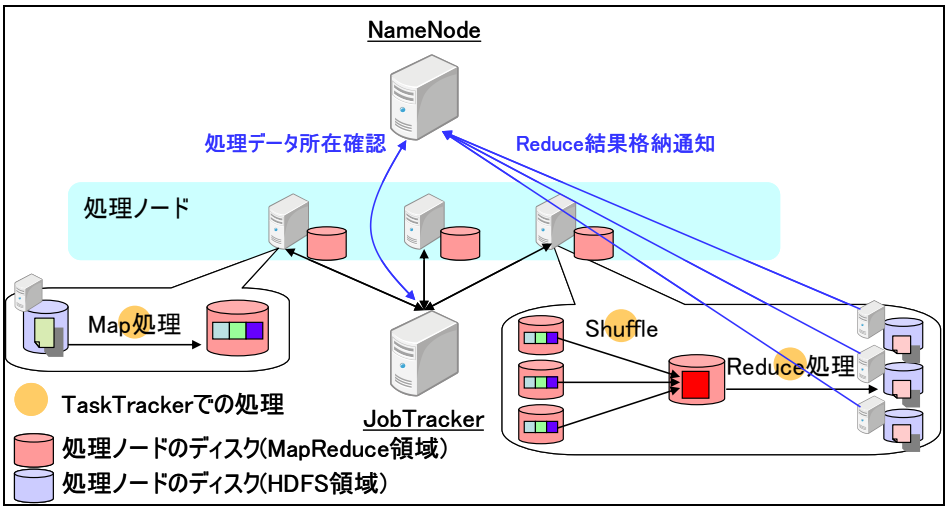


図 9-4 MapReduce と HDFS の連携

9.3.2 MapReduce ジョブでの処理ノードのリソース消費

前項で示した、MapReduce ジョブの流れを詳細化し、Hadoop 基盤の特性を判断する。表 9-7 と図 9-3 と図 9-4 で示した関係より、MapReduce の各処理ステージの処理特性をリソース消費の観点から確認する。

Map 処理でのリソース消費

Map 処理は、図 9-5 に示すように、以下の処理がある。

- HDFS 上のデータを読み出す
- Map 処理を実行する
- Map 処理結果を処理ノード自身のローカルディスクに書き出す

Hadoop では、ファイルを分割したブロックを DataNode に保存する。Map 処理は、処理ノードの DataNode で保存しているブロックを優先的な処理対象とするように処理が割り当てられる。この割り当て方針によって、他サーバとの通信を発生させないように設計されている。しかしブロックが存在しない場合は、他の処理ノードに保存されているブロックを通信で取得して Map 処理を実行する。

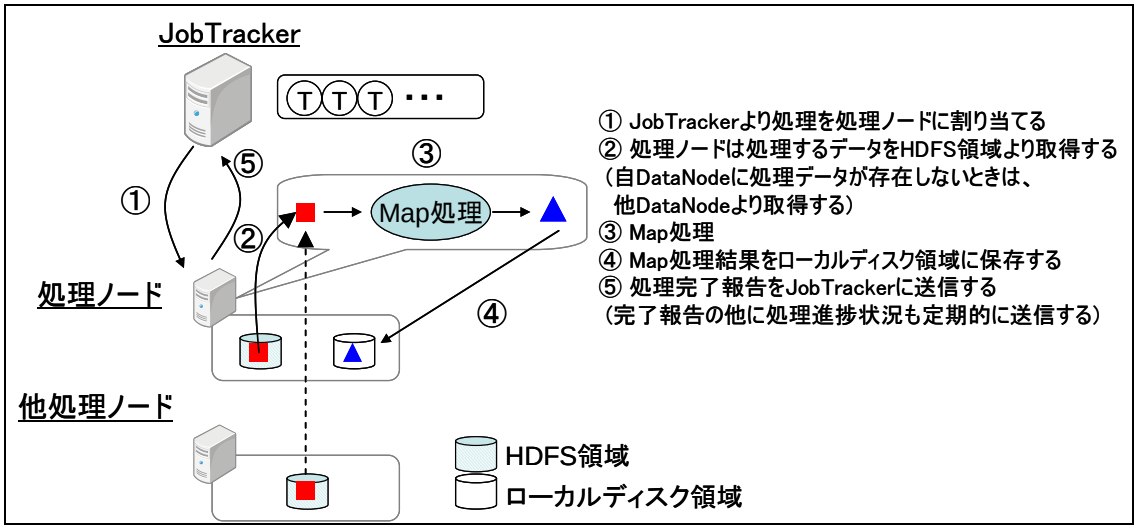


図 9-5 処理ノードでの Map 処理の流れ

Map 処理実行時に消費されるリソースについて、表 9-8 に示す。

表 9-8 Map 処理でのリソース消費

No.	リソース	処理内容
1	CPU	定義した Map 処理
2	ディスク	・Map 処理のためのデータをディスクからの読み出す ・Map 処理結果をディスクに書き出す
3	メモリ	定義した Map 処理

No.3 は、Hadoop の設定でリソース消費を制御できる。しかし、No.1 や No.2 は処理内容に依存するため Hadoop の設定だけでは制御することはできない。そのため、ボトルネックが発生しやすい。

Shuffle でのリソース消費

Shuffle は、Reduce 処理を実行する処理ノードで図 9-4 や図 9-6 に示すような処理を実行する。

- Map 処理結果である中間ファイルを他の処理ノードから取得する
- 中間ファイルを結合する
- 結合した中間ファイルを Key でソートする

集約処理、ソート処理のどちらも、割り当てられた JavaVM ヒープメモリサイズにデータが収まる場合はメモリ内で処理が完了する。しかし、メモリ量以上のデータを扱う場合には、データを退避させるためのディスク IO が発生する。

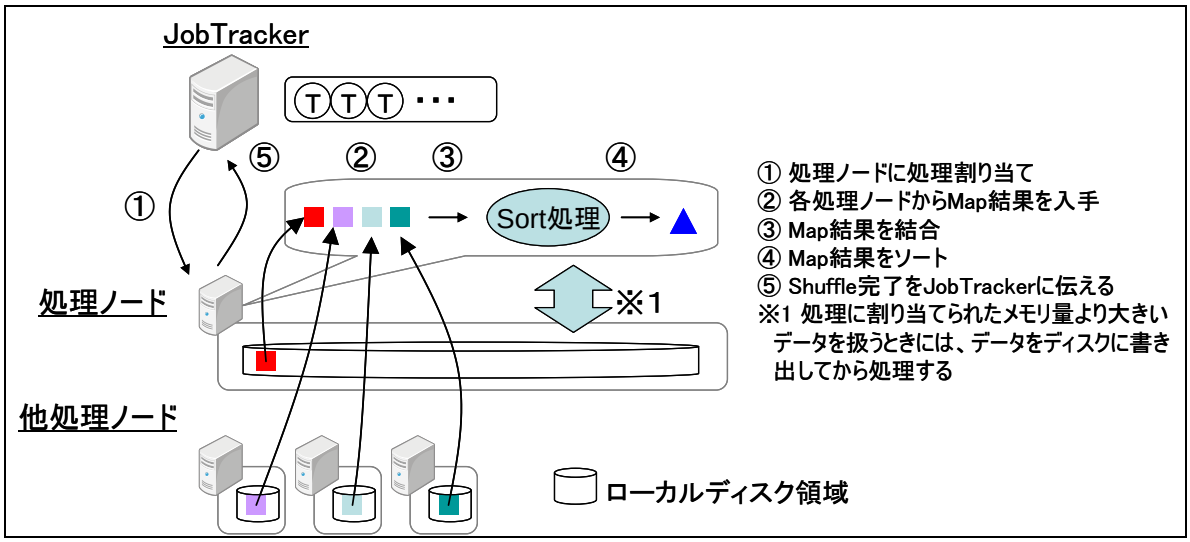


図 9-6 処理ノードでの Shuffle の流れ

Shuffle では、表 9-9 に示すようなリソースを消費する。

表 9-9 Shuffle のリソース消費

No.	リソース	処理内容
1	ディスク	・Map 結果を読み出す ・メモリサイズ以上の処理データを扱う場合のディスクへの読み書き
2	ネットワーク	Map 結果の転送

Reduce 処理でのリソース消費

Reduce 処理では、図 9-7 に示すとおり Shuffle された結果を入力として、各自で定義した Reduce 処理を実行する。処理結果は、Reduce 処理を実行した処理ノードのディスク(HDFS 領域)に書き出し、他の処理ノードに結果をレプリケーションする。

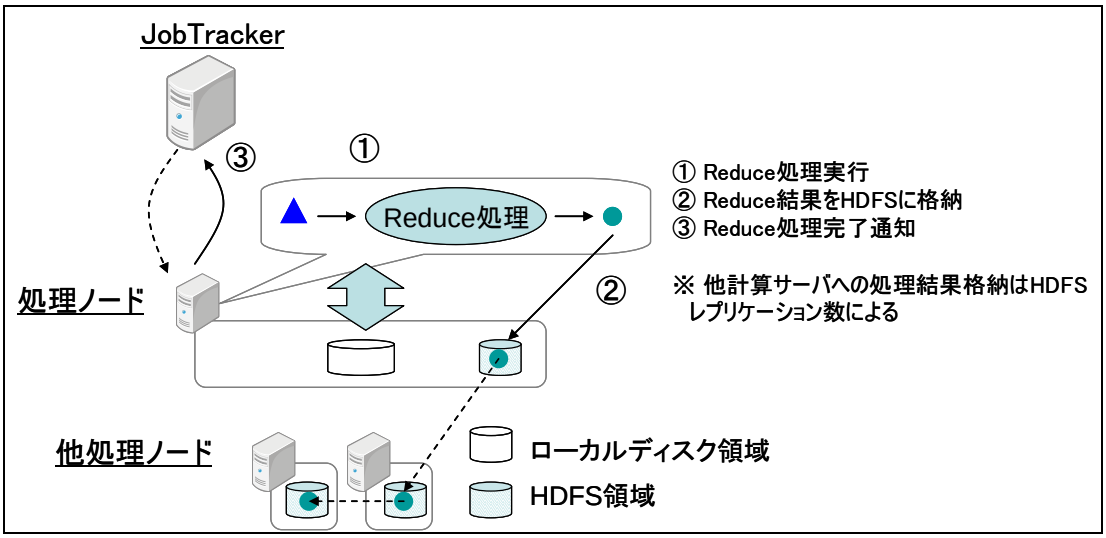


図 9-7 処理ノードでの Reduce 処理の流れ

Reduce 処理では、表 9-10 に示すようなリソースを消費する。

表 9-10 Reduce 処理でのリソース消費

No.	リソース	処理内容
1	CPU	定義した Reduce 処理
2	ディスク	・Reduce 結果を書き出す ・処理データがメモリに載らない場合にディスクへの読み書き ・レプリケーションされる Reduce 結果を書き出す
3	メモリ	定義した Reduce 処理
4	ネットワーク	Reduce 結果のレプリケーション

No.3 は MapReduce アプリケーションでの設定、No.4 はレプリケーション数やブロックサイズの設定で回避できる。しかし No.1、No.2 は、Hadoop の設定だけでは回避できず、ボトルネックとなる。

以上より、Hadoop 上での MapReduce 処理で処理ノードのリソースを消費するポイントについて示した。この結果、CPU やディスク、ネットワークでボトルネックが発生しやすいことを確認した。

これらは、MapReduce での処理内容に応じてボトルネックとなる。そのため、Hadoop 基盤の特性を判断し、ボトルネックに着目した性能特性を検討する。

9.3.3 ハードウェアスペックが混在する Hadoop 基盤への環境制約

ハードウェアスペックが混在する処理ノードによって、Hadoop 基盤の性能に与える影響について述べる。

9.3.3.1 CPU スペックのばらつき

CPU スペックが異なると、処理ノードでの Map 処理時間や Reduce 処理時間に差が発生する。MapReduce ジョブ実行時間は、すべての Map 処理や Reduce 処理が完了するまでの時間であるため、スペックにばらつきがある環境では CPU スペックが低い処理ノードにより MapReduce ジョブ実行時間が長くなる。また、CPU スペックの高い処理ノードでは早く処理が完了するため、図 9-8 に示すように全ての処理が完了するまで、処理を何も実行しない空き時間が発生する。

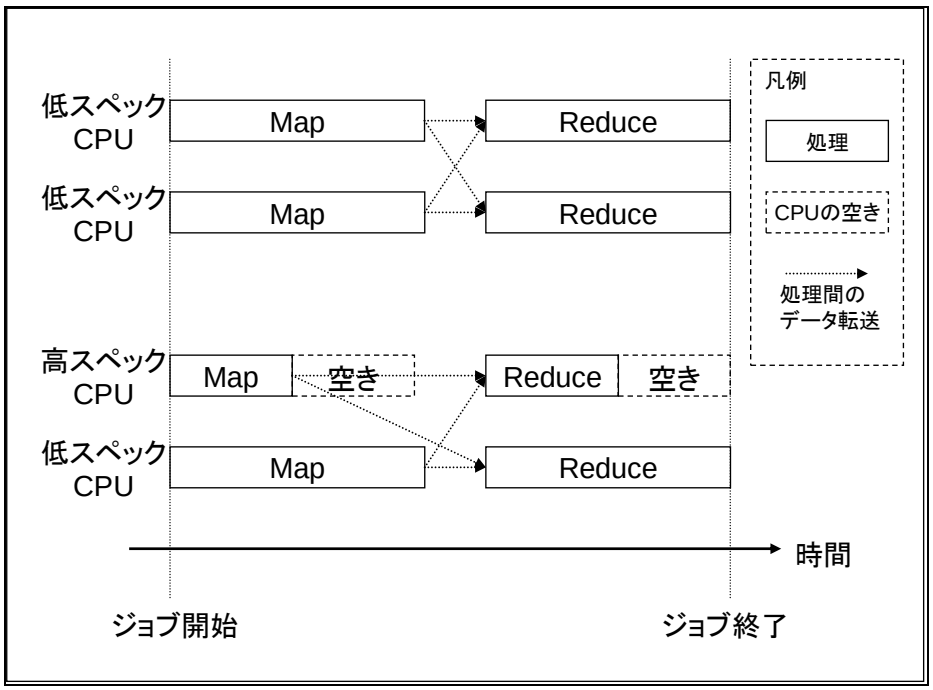


図 9-8 CPU スペックの混在による MapReduce 処理への影響

空き時間が発生する状態は、9.2.3 の処理ノードの実行状況の考え方では良いとは言えない。そのため、Map 処理、Reduce 処理を CPU スペックの影響が無い程度まで分割することで空き時間の発生を抑える。Map 処理、Reduce 処理ともに処理を分割した場合、CPU スペックが異なっていても図 9-9 に示すように処理を割り当てられるので、CPU スペックの高い処理ノードの処理能力が余る状況を軽減できる。また、MapReduce ジョブ全体での処理時間も短縮できる。

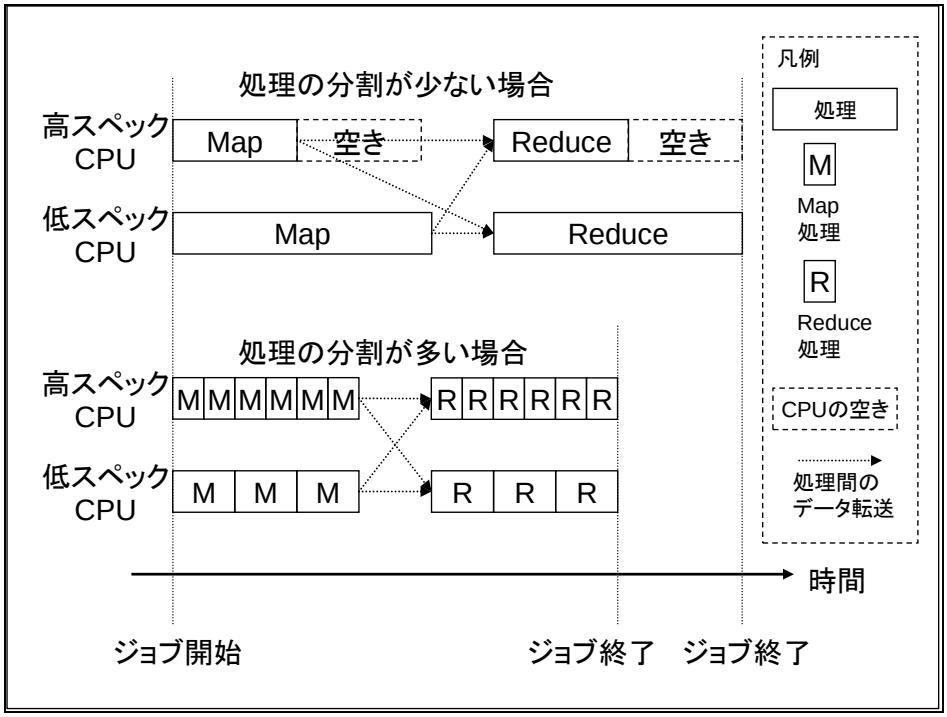


図 9-9 Map 処理や Reduce 処理の分割による効果

9.3.3.2 メモリ性能のばらつき

処理ノードのメモリ利用例について、図 9-10 に示す。

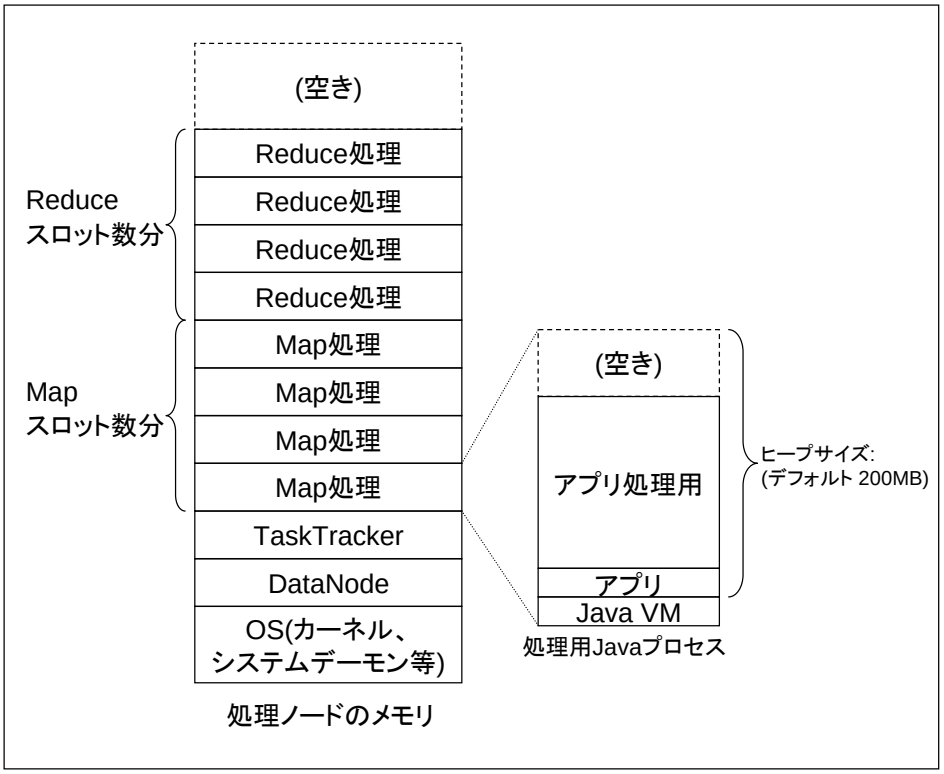


図 9-10 処理ノードメモリ利用例

Hadoop の Map 処理、Reduce 処理で割り当てられる JavaVM ヒープメモリ内には、MapReduce ジョブで扱うバッファ領域がある。バッファ領域は固定サイズで割り当てられている。

Map 処理では、図 9-11 に示すようなバッファ領域がある。Map 処理結果を格納するときにバッファ領域が足りない場合は、ディスクに書き出す操作が発生する。このディスクに書き出す操作によって処理時間が延びる。また、アプリケーションの処理内容次第で、GC の頻発や java.lang.OutOfMemoryError などの事象が発生する。

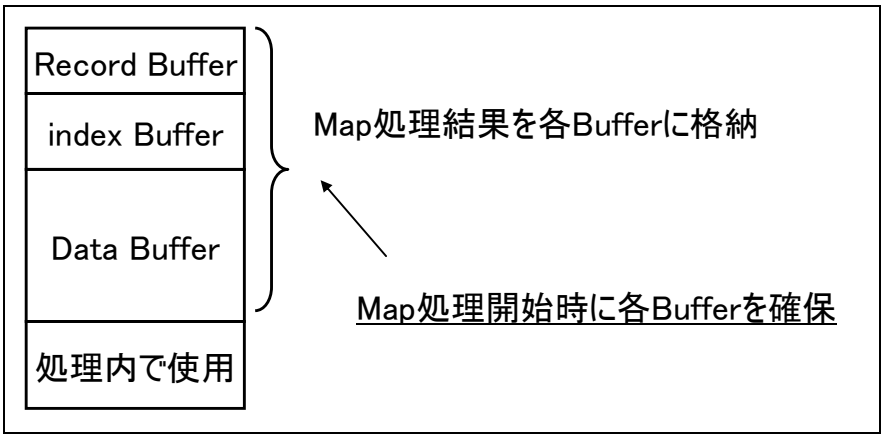


図 9-11 Map 処理でのヒープメモリ割り当て状況

Reduce 処理では、Map 処理のヒープメモリの割り当て方法と同じように、Reduce 処理で扱うデータの格納領域として、バッファが割り当てられる。

Map 処理や Reduce 処理に割り当てる Java VM ヒープメモリサイズは、どちらも同じサイズとなる(デフォルトでは 200MB)。一方、Map 処理や Reduce 処理で割り当てるヒープメモリを増やしすぎると、処理ノードの実メモリが不足することもある。

Hadoop では、Map スロット数と Reduce スロット数に比例して実メモリを使用する。CPU コア数に対して実メモリが極端に少ない場合は、実メモリ不足によるスワップアウトが発生する可能性がある。スワップアウトが発生すると処理が長期化するため、Map スロット数や Reduce スロット数を抑える必要がある。

実メモリ量のばらつきによる MapReduce ジョブへの影響は以下の通りである。

- 処理ノードに搭載している CPU1 コアあたりの実メモリ量が十分あり、実メモリ不足とならない場合、MapReduce ジョブ実行への影響はない。
- CPU1 コアあたりの実メモリ量が少ない処理ノードは、実メモリ不足によるスワップアウトを避けるため Map スロット数や Reduce スロット数を減らさなければならない。その場合 CPU リソースを十分に使いきれない可能性がある。

9.3.3.3ディスク容量のばらつき

MapReduce ジョブを実行する処理ノードのディスク容量が十分でない場合は、処理ノードに Map 処理や Reduce 処理が割り当てられない状況が発生する。

Hadoop では、Map 処理、Reduce 処理がそれぞれ一定の割合が(デフォルト 10%)完了すると、JobTracker で 1 処理あたりの処理データ量の見積もり値が設定される。JobTracker は、新たな Map 処理、Reduce 処理の割り当ての可否を見積もり値と処理ノードのディスク容量で判断する。そのため、処理ノードのディスク容量が少ない場合は新たな処理が割り当てられない。

処理が割り当てられない様子は、JobTracker の以下のログから確認できる。

```
# Map 処理が割り当てられない例 - JobTracker ログ
2010-02-02 17:14:24,230 WARN org.apache.hadoop.mapred.JobInProgress: No room for map task. Node slave001 has 14135296 bytes free; but we expect map to take 109193991

# Reduce 処理が割り当てられない例 - JobTracker ログ
2010-02-02 17:14:24,231 WARN org.apache.hadoop.mapred.JobInProgress: No room for reduce task. Node tracker_slave001:localhost.localdomain/127.0.0.1:44741 has 14135296 bytes free; but we expect reduce input to take 1091939917
```

JobTracker の 1 処理あたりの処理データ量見積もり値の設定前に割り当てられた処理や、見積もり値が十分でなく処理ノードのディスク容量が足りない状態で割り当てられた処理は失敗する。

```
# ディスク容量不足による処理失敗例 - 処理に関するログ
2010-02-02 16:51:37,581 FATAL org.apache.hadoop.mapred.TaskRunner: Task attempt_201001061158_0367_m_000003_0 failed : org.apache.hadoop.fs.FSError: java.io.IOException: No space left on device
at org.apache.hadoop.fs.RawLocalFileSystem$LocalFSFileOutputStream.write (RawLocalFileSystem.java:192)
(・・・途中省略・・・)
Caused by: java.io.IOException: No space left on device
        ... 8 more
```

ディスク容量が少ないサーバを処理ノードして利用する場合、MapReduce ジョブ

での処理データ量が処理ノードのディスク容量以上になることを考慮して以下のような対策を検討する。

- ディスク容量が少ない処理ノードへの処理割り当てによる処理失敗を考慮して、Map 処理の試行回数、Reduce 処理の試行回数を変更する(デフォルト 4 より大きな値とする)。これは、設定ファイル mapred-site.xml 内か MapReduce ジョブ設定時に、“mapred.map.max.attempts”, “mapred.reduce.max.attempts”プロパティで設定できる。
- Hadoop の MapReduce ジョブでは、処理ノードは、処理を一定回数失敗すると MapReduce ジョブとしてのブラックリストに登録され、新たな処理がアサインされなくなる。このブラックリストに登録されるまでの処理失敗数を変更する(デフォルト 4)。この値を小さくすることで、ディスク容量が少ない処理ノードへの処理割り当てを早々に停止することが可能である。これは、“mapred.max.tracker.blacklists”プロパティで設定できる。

9.3.3.4 ネットワーク性能のばらつき

「表 9-6 混在するクラウド型分散処理環境のスペック一覧」で示したとおり、どのサーバもネットワーク転送速度は同じである。

現在、コモディティ製品の範囲で入手可能なネットワークインタフェースは 100Mbps, 1Gbps, 10Gbps がある。現在市販されているサーバのほとんどは 1Gbps の NIC が標準で搭載されているため、1Gbps の NIC を前提とする。

そのため本報告書では、ネットワークインタフェース性能のばらつきは考慮に入れない。

Hadoop では Shuffle と Reduce 処理結果を HDFS に書き込む時に大量の通信が発生する。特に、Shuffle での通信量は大きい。Hadoop の処理ノードに 100Mbps の製品を利用する場合、ネットワーク性能の低い処理ノードによって処理時間が増加する可能性がある。

9.3.4 Hadoop 基盤特性を把握するためのベンチマークモデル

本項は、前項までで示した確認項目より、Hadoop 基盤特性に関するボトルネックに着目したベンチマークモデル(CPU、ディスク IO、ネットワーク)を示す。

9.3.4.1 CPU ボトルネックとなるベンチマーク

CPU ボトルネックは、何らかの処理で CPU 使用率(top や sar コマンド実行時に%user や%system の和)が 100%を維持し続ける状態のことを示す。入出力(%iowait)で CPU 使用率が 100%を維持し続ける状態は、CPU ボトルネックとしない。

Hadoop では処理ノードが Map 処理、Reduce 処理実行時に CPU 使用率 100%となるケースを想定する。処理ノードの CPU が複数のコアで構成されている場合は、全てのコアで Map 処理や Reduce 処理を実行している。

Map 処理や Reduce 処理で CPU ボトルネックを引き起こす方法は、それぞれの処理で、複雑な計算を実行することで実現できる。

CPU ボトルネックは、以下のような処理で発生する。

- 機械学習
- 数学関係の演算(素数計算、π計算など)

これらの処理で、容易に入手可能かつ実行可能である CPU ボトルネックベンチマークとして、Hadoop パッケージ内にπ計算(PiEstimator)を実行するサンプルがある。PiEstimator の処理フローを図 9-12 に示す。PiEstimator は、モンテカルロ法を利用して円周率計算するベンチマークで、モンテカルロ法による処理部分で CPU を利用する。π計算に利用するサンプル数を増加させることで、CPU ボトルネックを発生できる。

以上より、PiEstimator を CPU ボトルネック用のベンチマークとして扱う。

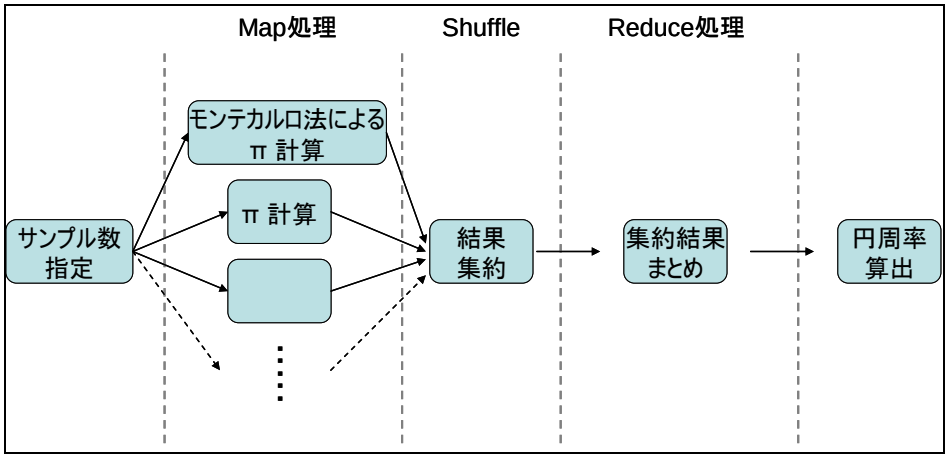


図 9-12 PiEstimator 処理フロー

9.3.4.2ディスクボトルネックとなるベンチマーク

ディスクボトルネックは、何らかの処理でディスク IO に関するパラメータの数値が高い状態を指す。ディスクボトルネックは、iostat コマンドで実行した場合に出力される await(ディスク IO 処理での平均待ち時間)や avgqu-sz(IO リクエストキューの長さ)、%util(ディスク IO での CPU のビジー率)を見ることで判断できる。

MapReduce は、Map 処理、Reduce 処理と経ることで処理データ量を減少させるよう MapReduce ジョブを定義することが望ましい。しかし、処理データ量を減少で

きない場合では、ディスクボトルネックが発生する。

ディスクボトルネックは、以下のような処理で発生する。

- 整列
- データ変換
- データ生成

以上の処理の中で、容易に入手可能かつ実行可能であるディスクボトルネックベンチマークとして、Hadoop パッケージ内に TeraSort と呼ばれるサンプルがある。TeraSort の処理フローを図 9-13 に示す。TeraSort は、Key・Value 形式で記述されているデータを MapReduce によって、昇順にソートするプログラムである。大量のデータを扱うことで、ディスクボトルネックを発生させることができる。

以上より、TeraSort をディスクボトルネック用のベンチマークとして扱う。

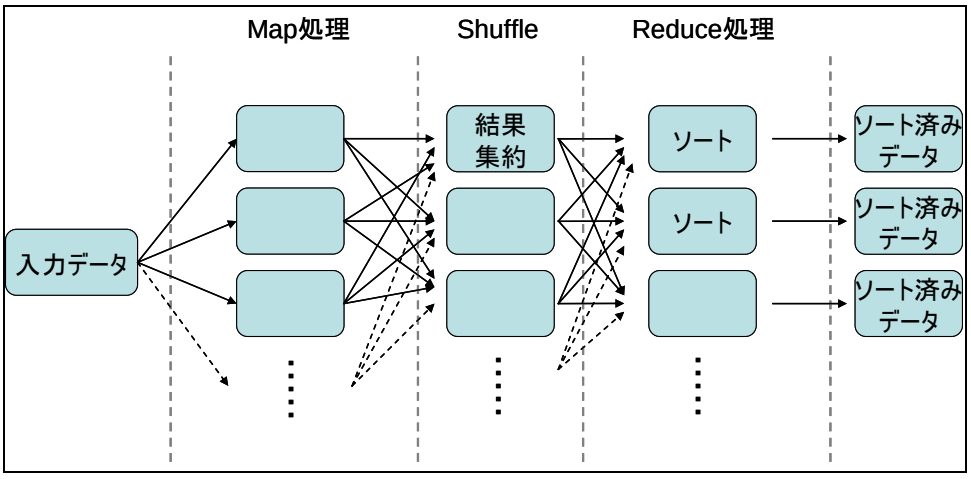


図 9-13 TeraSort 処理フロー

9.3.4.3ネットワークボトルネックとなるベンチマーク

ネットワークボトルネックは、何らかの処理でサーバ間の通信量が多い状態で一定となり、その状態で通信パケットのドロップやエラーが発生する状態である。sar コマンド(sar -n DEV)を実行したときに秒間あたりの通信量(rxbyt/s、txbyt/s)が大きな値で推移する状態となる。

Hadoop では、以下のケースで大量の通信が発生する。

- Shuffle を実行する処理ノードが、Map 処理を実行した処理ノードから計算結果を取得する場合
- Reduce 処理結果を HDFS に格納する場合

処理データの集約率を考慮すると、Shuffle で瞬間的に発生する通信量が Reduce 処

理の通信量より大きいため、Shuffle 時にネットワークボトルネックが発生する可能性が高いと考える。そのため、Shuffle に着目したベンチマークを検討する。

Hadoop パッケージには、ネットワークボトルネックに関するベンチマークソフトウェアは含まれていない。よって、Shuffle での通信に着目したベンチマークを用意すれば良い。Map 処理結果を Shuffle するときに掛かる時間をベンチマーク結果として扱うことでベンチマークを実現できる。

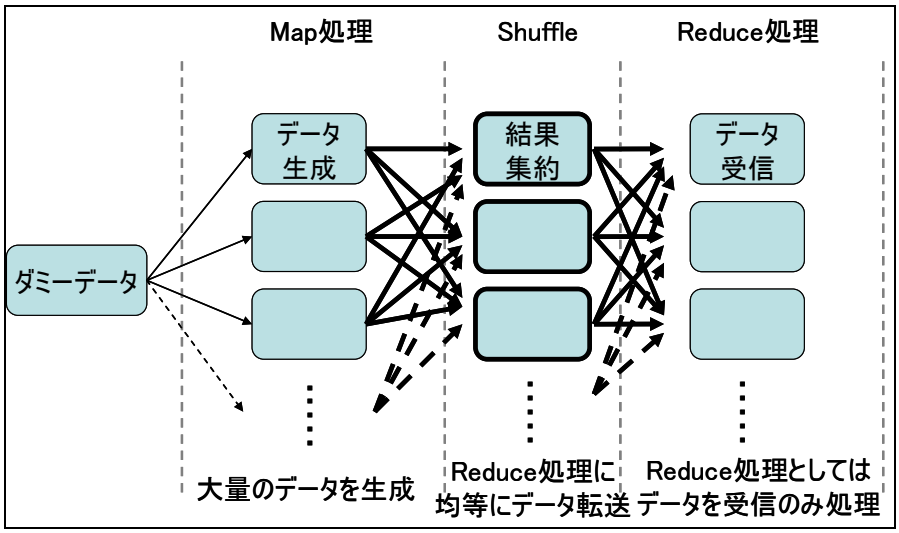


図 9-14 ネットワークベンチマーク処理フロー

9.4 Hadoop 基盤の性能チューニング検討

本節では、MapReduce ジョブを実行する Hadoop 基盤で性能に影響を与えるチューニングについて検討する。

9.3.2 で説明したとおり、Map 処理、Reduce 処理で処理ノードの CPU やディスク IO、ネットワークがボトルネックになる可能性が高いことを説明した。

これらのボトルネックを回避するために、Hadoop 基盤ではどのような対処をすべきか、Hadoop 基盤のチューニングについて検討する。

9.4.1 各処理での Hadoop 基盤チューニングポイント

本項では、Hadoop での Map 処理、Reduce 処理のチューニングポイントについて説明する。なお Shuffle は、Reduce 処理の一部として検討する。

Map 処理でのチューニングポイント

9.3.2 で示したとおり Map 処理でのチューニングポイントは、以下の 2 つである。

(1) Map 処理での CPU 使用率

(2) Map 処理でのディスク IO 性能

まず、(1)から検討する。何らかの Map 処理をする場合に、top や sar コマンドで確認できる CPU 使用率が 100%になることで、リソースを有効活用していると判断できる。現在サーバとして使用する CPU は複数コアで構成されているので、その全てのコアを利用するようにすることが大切である。図 9-15 に示すように、Map 処理を全ての CPU コアに割り当てられるようチューニングする。

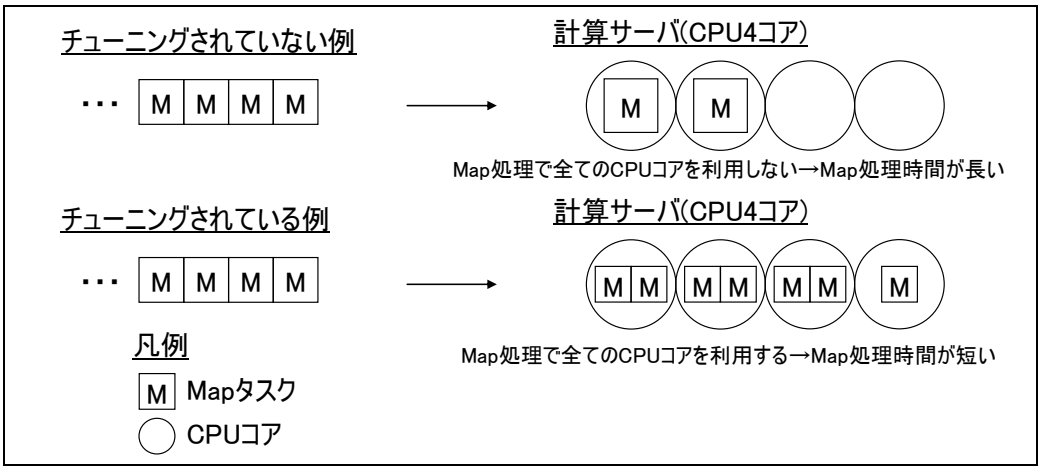


図 9-15 Map 処理での CPU 利用の考え方

次に、(2)について検討する。複数のディスクで構成されている処理ノードは、Map 処理の結果を書き出す場合に、複数のディスクを書き込み先に指定することでディスク IO の分散を実現する。図 9-16 に示すように複数の CPU コアで処理した結果をディスクに書き出す場合、複数の出力先を指定することで、Map 処理時間の短縮につながる。

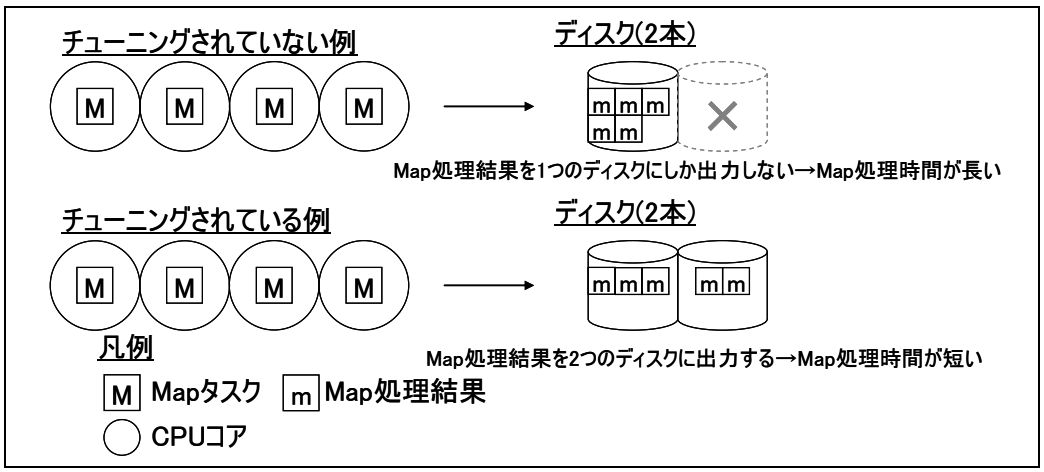


図 9-16 Map 処理でのディスク利用の考え方

Reduce 処理でのチューニングポイント

表 9-9 や表 9-10 で示したとおり、Reduce 処理のチューニングポイントは以下の通りである。

(1) Reduce 処理での CPU 使用率
(2) Reduce 処理でのディスク IO
(3) Map 処理結果を受け取る方法

まず、(1)から検討する。これは、Map 処理の CPU チューニングと同様に、図 9-17 に示すように Reduce 処理を全ての CPU コアに割り当てられるようにチューニングする。

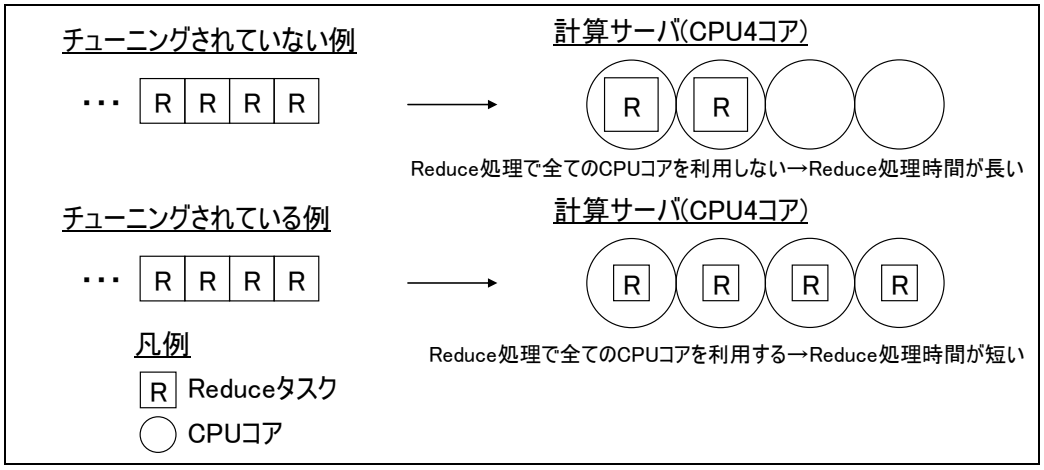


図 9-17 Reduce 処理での CPU 利用の考え方

次に、(2)について検討する。Reduce 処理結果は、HDFS に書き込まれる。そのため HDFS に格納するファイルのブロックサイズやレプリケーション数が Reduce 処理時間に影響を与える。これらのパラメータは、MapReduce を実行するアプリケーション単位で設定できる。

また、図 9-18 に示すように複数ディスクを持つサーバの場合、HDFS で利用するディスクを複数選択することで、ディスク IO の分散を実現できる。

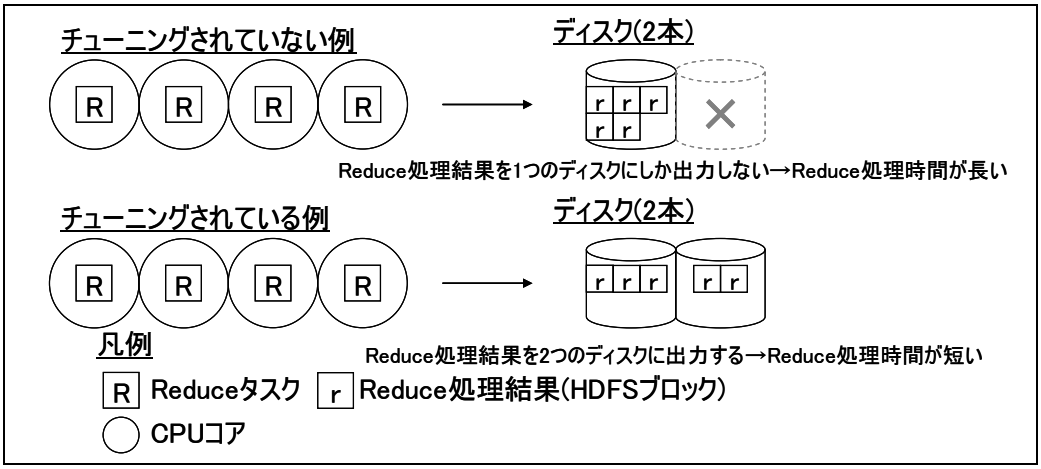


図 9-18 Reduce 処理でのディスク利用の考え方

(3)について、Shuffle は Map 処理の結果を複数の処理ノードから一度に大量に取得することで、処理時間を短縮できる。しかし、大量取得した場合は、処理ノード間でネットワークボトルネックや受け取る処理ノードのディスクボトルネックとなる可能性がある。そのため、ボトルネックにならないようにデータを取得する。

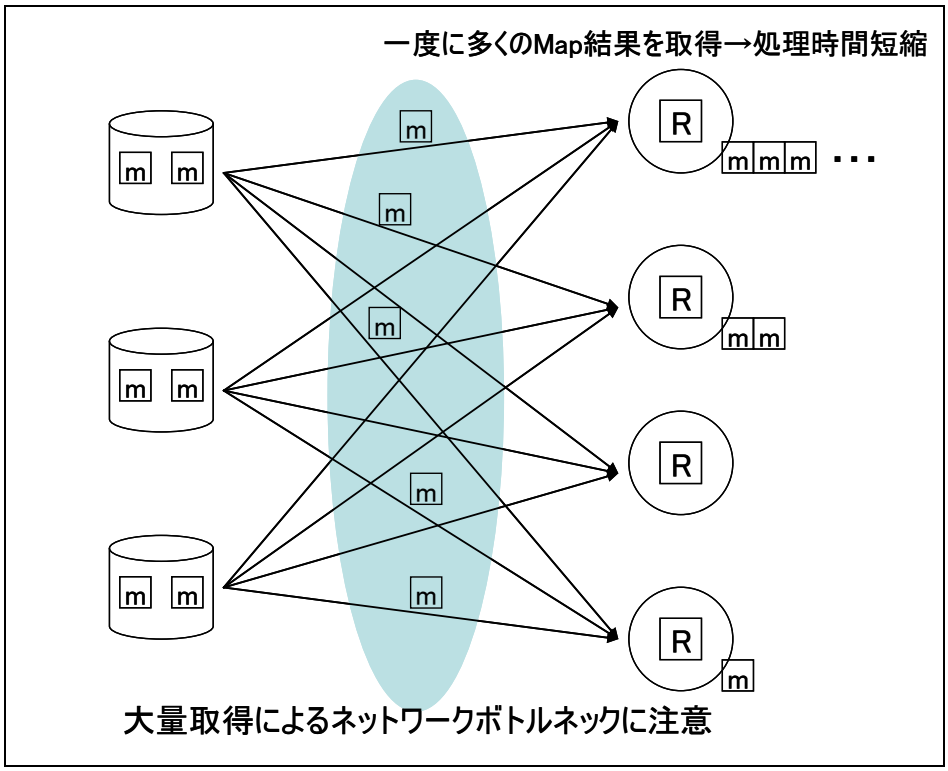


図 9-19 Map 結果取得の考え方

以上より、Hadoop 基盤の MapReduce 処理に対応するチューニングポイントを表

9-11 にまとめる。

表 9-11 Hadoop 基盤チューニングポイント

No.	処理	リソース	チューニング内容
1	Map 処理	CPU	Map 同時処理数
2	Map 処理	ディスク	Map 処理結果
3	Reduce 処理	CPU	Reduce 同時処理数
4	Reduce 処理	ディスク	Reduce 処理結果の出力先
5	Reduce 処理	ネットワーク	Map 処理結果同時入手数

具体的な Hadoop の設定について次項で述べる。

9.4.2 Hadoop 基盤での性能チューニングに関するパラメータ

前項で示した、Hadoop 基盤の性能チューニングポイントの設定方針を検討する。

まず、表 9-11 に示した、Hadoop 基盤のチューニングポイントに対して、有効となるパラメータを表 9-12 に示す。

表 9-12 MapReduce 処理性能に関する Hadoop 基盤パラメータ

No.	処理	リソース	パラメータ名	内容
1	Map 処理	CPU	mapred.tasktracker.map.tasks.maximum	処理ノード 1 台あたりの Map 処理最大同時処理数
2	Map 処理	ディスク	mapred.local.dir	Map 処理結果の出力先、複数指定すると分散配置する
3	Reduce 処理	CPU	mapred.tasktracker.reduce.tasks.maximum	処理ノード 1 台あたりの Reduce(Shuffle)処理最大同時処理数
4	Reduce 処理	ディスク	dfs.data.dir	HDFS での格納領域、複数指定すると分散配置する
5	Reduce 処理	ネットワーク	tasktracker.http.threads	Shuffle で Map 結果取得リクエストに対する同時処理スレッド数
6	Reduce 処理	ネットワーク	mapred.reduce.parallel.copies	Map 処理結果の最大同時収集数

上記のパラメータに対して、特にボトルネックの可能性があるディスク、CPU の観点について検討する。

9.4.2.1ディスク台数の決定

まず、Hadoop で利用する処理ノードのディスク台数について決定する。「表 9-6 混在するクラウド型分散処理環境のスペック一覧」で述べたサーバを利用して、ディスク台数を変化させた場合の影響を確認する。ディスク増減の影響によりディスクボトルネックの影響が変化する。そのため、MapReduce ジョブとして、500GB の TeraSort を実行した。なおサーバ種別 S3 は、ディスク容量が少ないため、実行対象から除外した。

表 9-13 ディスク台数違いによる TeraSort 実行結果

		ディスク台数			
サーバ	台数	テスト 1	テスト 2	テスト 3	テスト 4
S1	18 台	1 台	2 台	2 台	2 台
S2	4 台	1 台	2 台	2 台	2 台
S4	9 台	1 台	1 台	1 台	2 台
S4	19 台	1 台	1 台	2 台	2 台
S5	8 台	1 台	2 台	2 台	2 台
総ディスク台数 (対テスト 1 増加率)		58 台 (100%)	96 台 (165.5%)	107 台 (184.5%)	116 台 (200%)
実行時間 (対テスト 1 減少率)		2531 秒 (100%)	1961 秒 (77.5%)	1811 秒 (71.6%)	1775 秒 (70.1%)

以上の結果より、テスト 1 からテスト 4 と MapReduce ジョブで使用するディスク台数を増やすことで、処理時間の短縮(テスト 1：2531 秒からテスト 4: 1775 秒)を確認できた。この結果より、処理ノードのディスクに関する設定方針を以下のようにする。

“Hadoop の MapReduce ジョブで処理を実行する処理ノードは、処理時間改善のため、処理ノードに搭載しているディスク全てを Hadoop 用に割り当てる。”

Hadoop のパラメータとしては、以下の通りに設定する。

表 9-14 ディスクに関係した Hadoop パラメータ設定方針

No.	パラメータ名	設定方針
1	mapred.local.dir	処理ノードのディスクを全て使用するように、領域設定する。
2	dfs.data.dir	

9.4.2.2スロット数の決定

Map スロット数、Reduce スロット数は、Map 処理、Reduce 処理の特性を考慮して、以下のベンチマークを実行することで決定する。

- Map 処理 ：Map 処理は、入力データを分割して処理させる特性から、CPU 負荷となる PiEstimator を実行する。
- Reduce 処理 ：Reduce 処理は、データの結合やソートする処理特性から、ディスク負荷となる TeraSort を実行する。

まず、Map スロット数に関して PiEstimator を実行する。PiEstimator を以下の条件で実行する。

- 処理ノード数 88 台（S1 17 台, S2 4 台, S3 16 台, S4 43 台, S5 8 台）
- Map スロット数を CPU コア数×1, 1.5, 2, 3, 4 と変化
- 各測定とも 5 回実行し、5 回実行による最速実行時間、平均実行時間、標準偏差を求める

PiEstimator 実行結果を表 9-15 に示す。

表 9-15 PiEstimator 実行結果

No.	Map スロット数	最速実行時間(秒)	平均実行時間(秒)	標準偏差
1	CPU コア数×1	483	487	3.35
2	CPU コア数×1.5	481	483	1.27
3	CPU コア数×2	485	497	6.68
4	CPU コア数×3	500	505	2.93
5	CPU コア数×4	481	511	17.65

以上の結果より、No.1 や No.2 は平均実行時間が短かった。No.5 は、最速実行時間が早いものの標準偏差より実行時間にばらつきが生じた。このことから、Map スロット数は、Map スロット数は、CPU コア数～CPU コア数の 1.5 倍であれば、実行時間

を短くできる。

次に、Reduce スロット数に関して、TeraSort を実行する。検証環境ならびに検証条件について、以下に示す。

- 処理ノード数 8 台 (S1 8 台), CPU4 コア
- 40GB の TeraSort を実行
- Map スロット数: 4(CPU コア数×1) , 6 (CPU コア数×1.5)
- Reduce スロット数: 2(CPU コア数×0.5) ～ 6(CPU コア数×1.5)
-

TeraSort 実行結果を図 9-20 に示す。

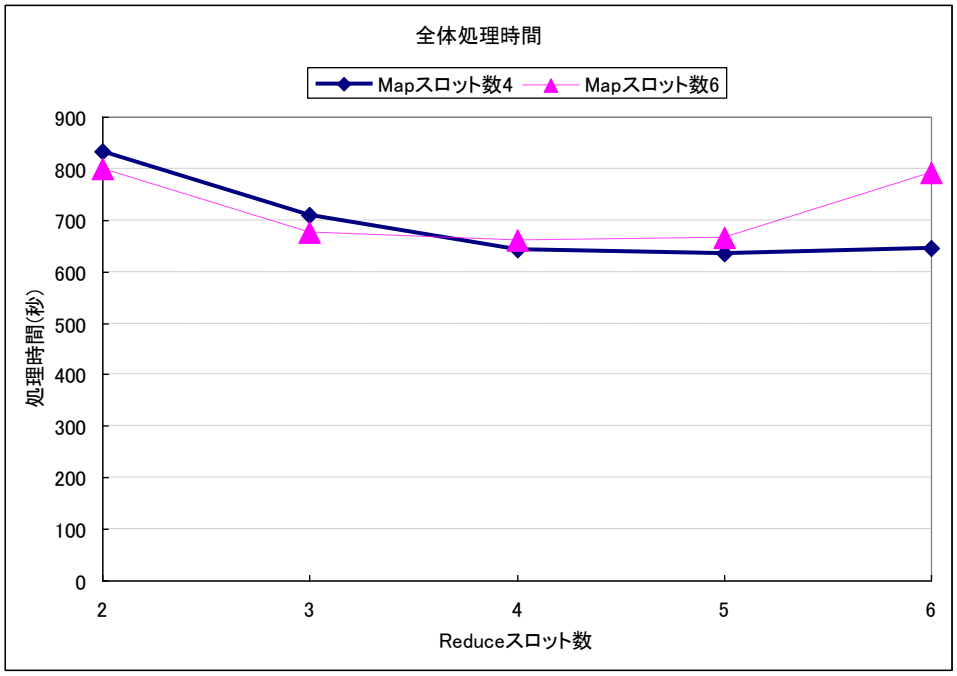


図 9-20 スロット数決定のための TeraSort 実行結果

以上の結果より、Map スロット数によらず Reduce スロット数が 3～5 の場合に処理時間が短くなる結果となった。このときの処理ノードのリソース使用状況(CPU 使用率を図 9-21 に示す。

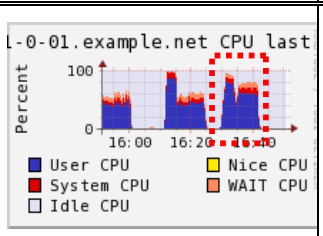
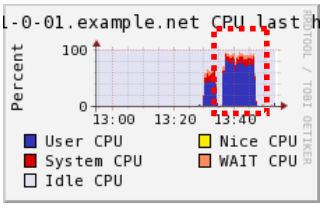
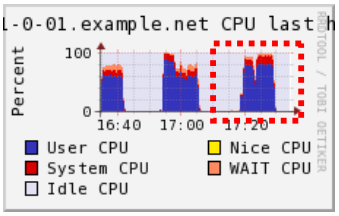
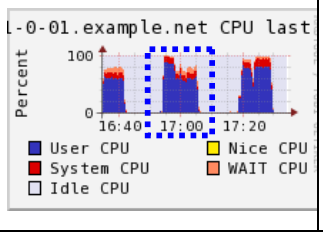
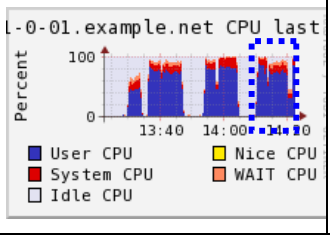
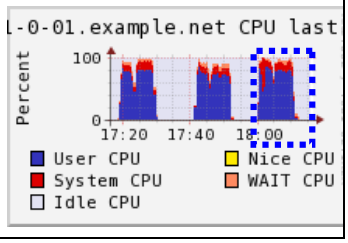
No.	Map 数	Reduce 3 スロット	Reduce 4 スロット	Reduce5 スロット
1	4			
2	6			

図 9-21 スロット数決定のための TeraSort 実行時のリソース使用状況

この結果より、スロット数が CPU コア数より少ない場合(Map スロット、Reduce スロットともに 4 以下)は、“Idle CPU”がはっきりと表れており、CPU リソースを十分に活用していないことがわかる。また、スロット数を増やすと、CPU リソースはほぼ 100%となり、リソースを活用できている。この結果より、Reduce スロット数は、コア数～コア数+1 で良いと言える。

以上より、処理ノードのスロット数設定方針は表 9-16 とする。

表 9-16 処理ノードのスロット数に関する設定方針

No.	パラメータ名	設定方針
1	mapred.tasktracker.map.tasks.maximum	CPU コア数×1～1.5
2	mapred.tasktracker.reduce.tasks.maximum	CPU コア数 ～ CPU コア数 + 1

これは、処理ノードの実メモリ容量が十分ある場合の設定である。実メモリ容量が少ない場合は、以下の手順で Map スロット数、Reduce スロット数を設定する。

(1) 処理ノードでは、Hadoop(DataNode、TaskTracker)や OS 用として 1GB 利用する。

(2) 各処理では JavaVM ヒープメモリサイズの他に Permanent 領域や C ヒープ領域として割り当てられるメモリサイズを 50MB とする。そのため、“(実メモリ－1GB)÷(Map 処理・Reduce 処理のヒープサイズ+50MB)＝処理ノードでのス

ロット数”とする。

(3) “切り上げ{(Map スロット数＋Reduce スロット数)} > (2)で求めたスロット数”となる場合は Map スロット数を CPU コア数×1 とする。このとき多少のスワップは処理時間に影響しないことを考慮すれば、先の式の“(Map スロット数＋Reduce スロット数)”を“(Map スロット数＋Reduce スロット数)÷2”で計算しても良い。その場合は、(2)の“Map 処理・Reduce 処理のヒープサイズ”の値を大きくして再計算する。

(4) (3)の対処後でもスロット数に影響する場合は、“Reduce スロット数－１”→“Map スロット数－１”と設定しなおす。

本検証で使用する混在環境では、Map スロット数や Reduce スロット数は表 9-17 のとおりに設定する。なお、1 スロットあたり JavaVM のヒープメモリには 200MB 割り当てる。(3)の条件式を変更すれば、1 スロットあたりの JavaVM ヒープメモリを 450MB まで割り当てられる。

表 9-17 混在環境スロット数設定方針

No.	設定部分	S1	S2	S3	S4	S5
1	Map スロット数	4	8	2	2	4
2	Reduce スロット数	4	8	2	2	4

処理で利用する JavaVM ヒープメモリは MapReduce ジョブ単位で設定できる。そのため、まずは少量データで MapReduce ジョブを実行してヒープメモリ使用状況を判断する。設定した割り当てよりも必要な場合は、ヒープメモリサイズを変更する。

9.5 アプリケーション実行に関係する Hadoop チューニング検討

本節は、MapReduce ジョブについて、Hadoop の設定で対処できるチューニングポイントとチューニング観点について述べる。

9.5.1 Map 処理チューニングに関する Hadoop パラメータ

MapReduce 環境において Map 処理とリソース使用の関係を図 9-22 に示す。また、Map 処理でチューニングに関するパラメータを表 9-18 に示す。

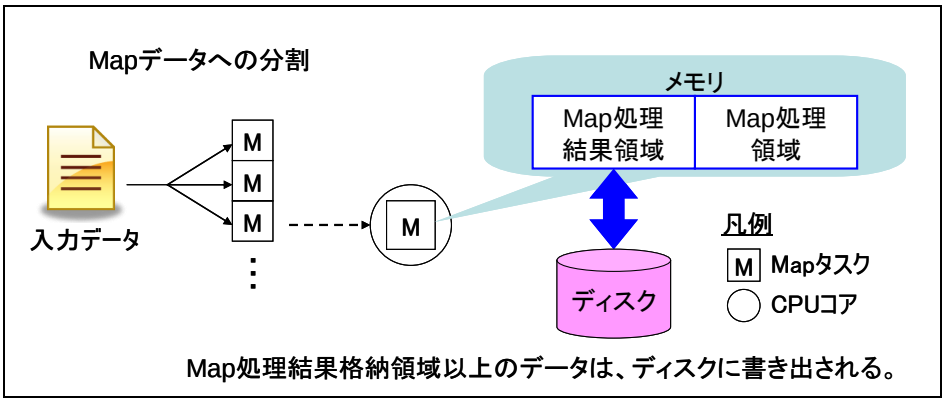


図 9-22 Map 処理とリソース使用

表 9-18 Map 処理チューニングのためのパラメータ

No.	チューニング観点	Hadoop パラメータ	内容
1	Map 処理分割	パラメータとしては無い	入力データを分割するサイズ MapReduce ジョブ内のメソッドで設定 (0.19 系までは、mapred.map.tasks で設定可能)
2	メモリ	mapred.child.java.opts	1 つの Map 処理で利用できるヒープメモリサイズなど (デフォルト 200MB)
3	メモリ	io.sort.mb	Map 処理結果を Partition により分割して Key 単位でソートするために必要な領域 (デフォルト 100)
4	メモリ	io.sort.record.percent	Map 処理のレコードバッファ領域 (デフォルト 0.05)

Map 処理数は、Map 処理で利用する入力データのブロック数に影響する。Map 処理数が Map スロット数よりも少ない場合は、Map 処理数をブロック数以上に設定する。

メモリは、1 つの Map 処理で利用する領域と Map 処理でデータを格納する領域の 2 つについて検討する。Hadoop では、Map 処理の出力データがヒープメモリに格納できない場合は、ディスクに書き出す。ディスクアクセスを軽減するようにメモリ領域を調整する。

Hadoop のデフォルトでは、1 つの Map 処理で JavaVM ヒープメモリが 200MB、

そのうち Map 結果格納のためのバッファ領域として 100MB 程度確保される。

9.5.2 Reduce 処理チューニングに関する Hadoop パラメータ

Reduce 処理のリソース使用について図 9-23 とチューニングのためのパラメータを表 9-19 に示す。

Reduce 処理では、Map 処理結果を Shuffle で取得するときに、処理結果のサイズが Reduce 処理で割り当てたヒープメモリ容量の 25%以下であればヒープメモリ上で保持する。25%以上であればディスクに書き出す。ヒープメモリ上のデータが一定の割合より大きくなるとデータを結合してディスクに書き出す。

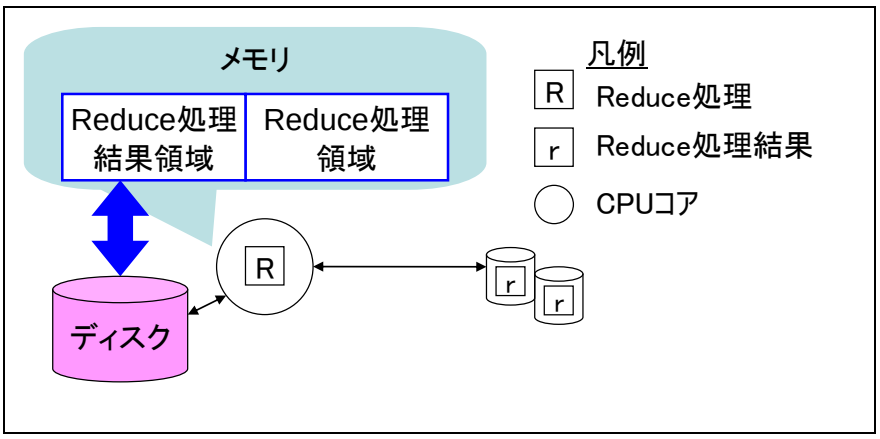


図 9-23 Reduce 処理とリソース使用

表 9-19 Reduce 処理チューニングのためのパラメータ

No.	チューニング観点	Hadoop パラメータ	内容
1	Reduce 処理分割	mapred.reduce.tasks	Reduce 処理を分割する(デフォルト 1)
2	メモリ	mapred.child.java.opts	1 つの Reduce 処理で利用できるヒープメモリサイズなど(デフォルト 200MB)
3	メモリ	mapred.job.shuffle.merge.percent	Map 結果の結合で利用できるメモリサイズ(デフォルト 0.66)
4	メモリ	mapred.job.shuffle.input.buffer.percent	Shuffle として割り当てるメモリサイズ（デフォルト 0.70)
5	Map 結果同時取得	mapred.reduce.parallel.copies	1 回の Shuffle で Map 処理結果を同時に取得する個数（デフォルト 5)

9.5.3 MapReduce ジョブでの処理数設定

本項は、9.5.2 までで説明したパラメータに対して、ベンチマークソフトウェアを利用して方針を決定する。9.5.2 までで述べたパラメータのうち、特に Map 処理分割や Reduce 処理分割が性能に影響を与える。そのため Map 処理数、Reduce 処理数をチューニングする。

9.5.3.1Map 処理数設定

Map 処理は、処理の特性上 CPU やディスクのリソースを使用する。そのため、MapReduce ジョブがどのリソースを消費するか踏まえて Map 処理を分割する。リソース消費の観点で、CPU リソース消費に着目して PiEstimator での Map 処理分割、ディスクリソース消費に着目して TeraSort による処理分割を考える。

数 GB 以上のデータを処理する場合の処理分割は、Hadoop 基盤としてのスループットに着目して、全ての処理ノードのリソースを利用することが望ましい。そのため、以下のように考える。

“分割できるのであれば、Hadoop 基盤の Map スロット総数以上に Map 処理を分割する。”

以降で述べる検討結果より、Map 処理分割方針を先に示す。

(1) 1 つの Map 処理で書き出すデータ量が 1 度にマージするセグメント(中間ファイルの一部)量を超えない範囲となるように Map 処理数を設定する。ブロックサイズ分の入力データを処理するときにセグメント量が超える場合は、ブロックを分割して Map 処理の入力とする。

(2) 1 つの Map 処理で扱うデータ量が JavaVM ヒープサイズよりの 5 分の 1 よりも小さい場合は、Map 処理を実行する処理ノードのリソースを活用しきれなくなる。そのため、(1)の影響が無い限りブロックサイズを小さな値にしない。

(3) ディスク読み書きが少ない MapReduce ジョブを実行する場合は、Map 分割数をスロット数の 30 倍程度より少なく設定する。

以降は、(1)～(3)に示した Map 処理分割方針を検討したものである。

CPU ボトルネックとなる Map 処理の分割数設定方針

CPU ボトルネックとなる Map 処理分割検討として、PiEstimator で 1Map 処理あたりの処理量を変更することで、処理分割方針を検討する。検証パターンや検証構成は以下の通りである。

- 1Map 処理あたりの計算量は、総計算量÷(スロット数×倍率)とする。
- 検証 1、検証 4：総計算量 2.5 兆となる PiEstimator 処理
- 検証 2：総計算量 5.0 兆となる PiEstimator 処理
- 検証 3：総計算量 7.5 兆となる PiEstimator 処理
- 検証 1～3：処理ノード 88 台（S1 17 台, S2 4 台, S3 16 台, S4 43 台, S5 8 台），スロット数総数 250
- 検証 4：処理ノード 44 台（S1 9 台, S2 2 台, S3 8 台, S4 21 台, S5 4 台），スロット数総数 126

1Map 処理あたりの計算量を変化させて、PiEstimator を実行した結果を図 9-24 に示す。

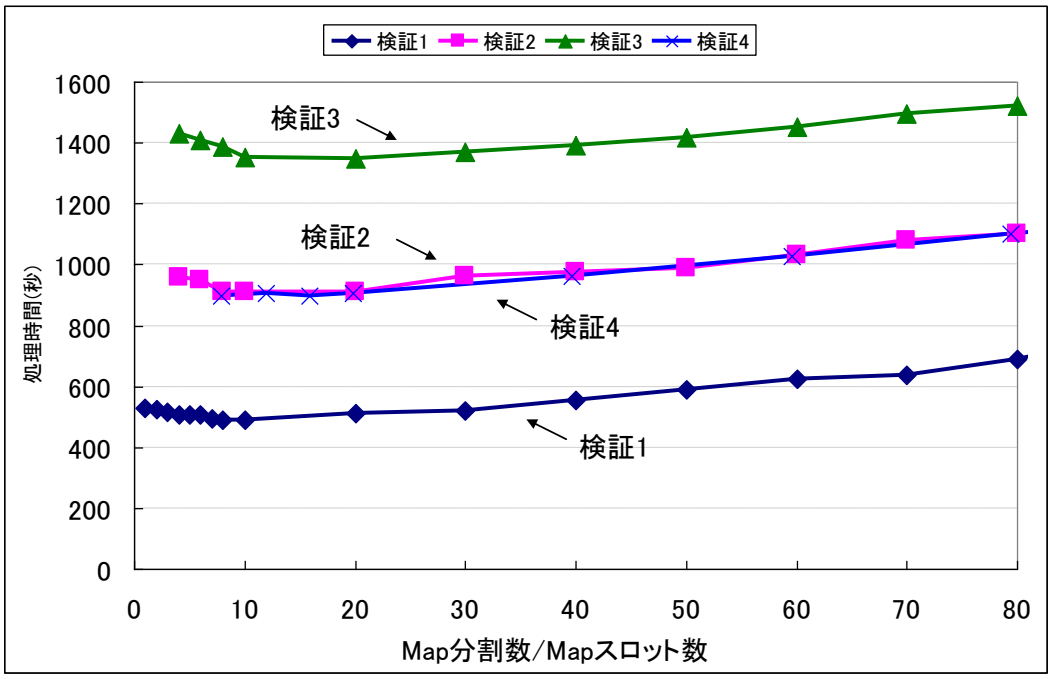


図 9-24 Map 処理分割影響調査結果(PiEstimator)

この結果より PiEstimator では、Map 処理数をスロット数の数十倍のような極端に設定しなければ、ほぼ同じような処理時間となる。なお、Map 処理数を増やすと総計算量が同じにも関わらず処理時間が増加している、これは、図 9-25 に示すように、Map 分割数が増えるほど Map 総処理時間が増えているためである。

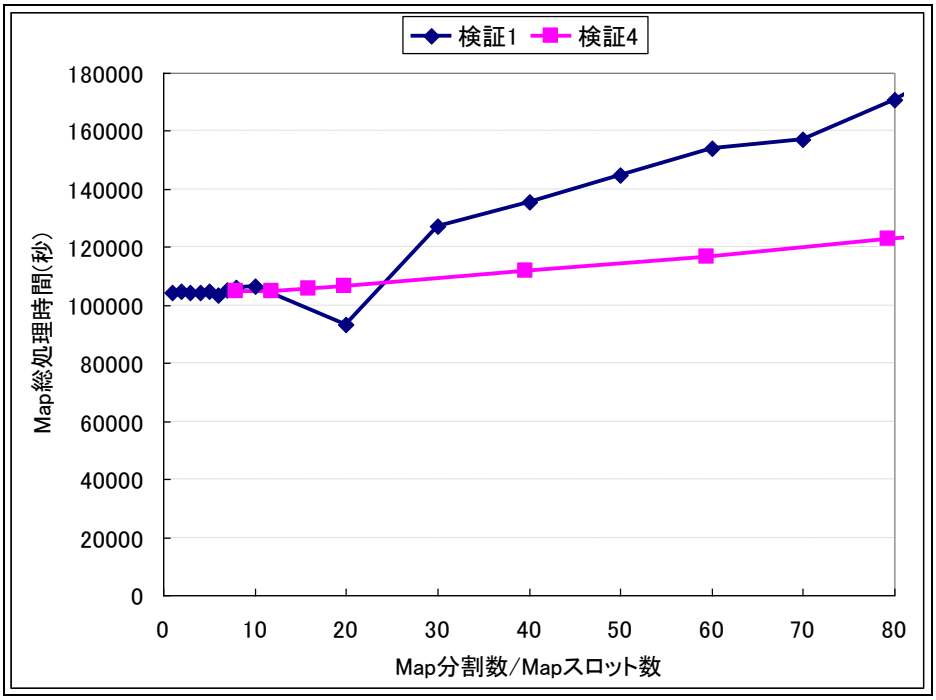


図 9-25 Map 総処理時間(PiEstimator)

図 9-25 より“Map 分割数が多い＝1 つの Map 処理での処理量が少ない”と Map 処理時間が長くなると言える。これは Map 処理に関係する Hadoop 実装のオーバヘッドが影響していると考える。

以上より、ディスク読み書きが無いケースでは処理時間に大差はない。そのときに処理時間を短くしたい場合は、Map 総処理時間の変化より分割数を Map スロット数の 30 倍よりも少ない範囲で設定すれば良いと言える。

ディスクボトルネックとなる Map 処理の分割数設定方針

ディスクリソースに着目した Map 処理数検討として、TeraSort を使用して Map 分割を検討する。検証パターンや検証構成は以下の通りである。

- 1Map 処理あたりの計算量は、総計算量÷(スロット数×倍率)として Map 処理数を設定する。Reduce 処理数は固定。
- 検証 1：処理ノード 8 台(S4：CPU2 コア)，スロット数総数 16, 40GB 実行
- 検証 2：処理ノード 8 台(S4：CPU2 コア)，スロット数総数 16, 80GB 実行
- 検証 3：処理ノード 16 台(S4：CPU2 コア)，スロット数総数 32, 40GB 実行
- 1Map 処理あたりの JavaVM ヒープメモリを 200MB(データ領域 100MB 程度)に設定

なお、Map 分割数と 1Map 処理あたりの処理データ量は表 9-20 の通りである。

表 9-20 Map 分割数と処理データ量の関係

No.	処理データ量 (MB)	検証 1 Map 分割数 (倍率)	検証 2 Map 分割数 (倍率)	検証 3 Map 分割数 (倍率)
1	500	80 (5 倍)	160 (10 倍)	80 (2.5 倍)
2	250	160 (10 倍)	304 (19 倍)	160 (5 倍)
3	130	304 (19 倍)	608 (38 倍)	304 (9.5 倍)
4	100	400 (25 倍)	800 (50 倍)	400 (12.5 倍)
5	83.3	480 (30 倍)	960 (60 倍)	480 (15 倍)
6	65.8	608 (38 倍)	1200 (75 倍)	608 (19 倍)
7	33.3	1200 (75 倍)	2400 (150 倍)	1200 (37.5 倍)
8	21.9	1824 (114 倍)	3648 (228 倍)	1824 (57 倍)

この処理結果について、図 9-26 に示す。

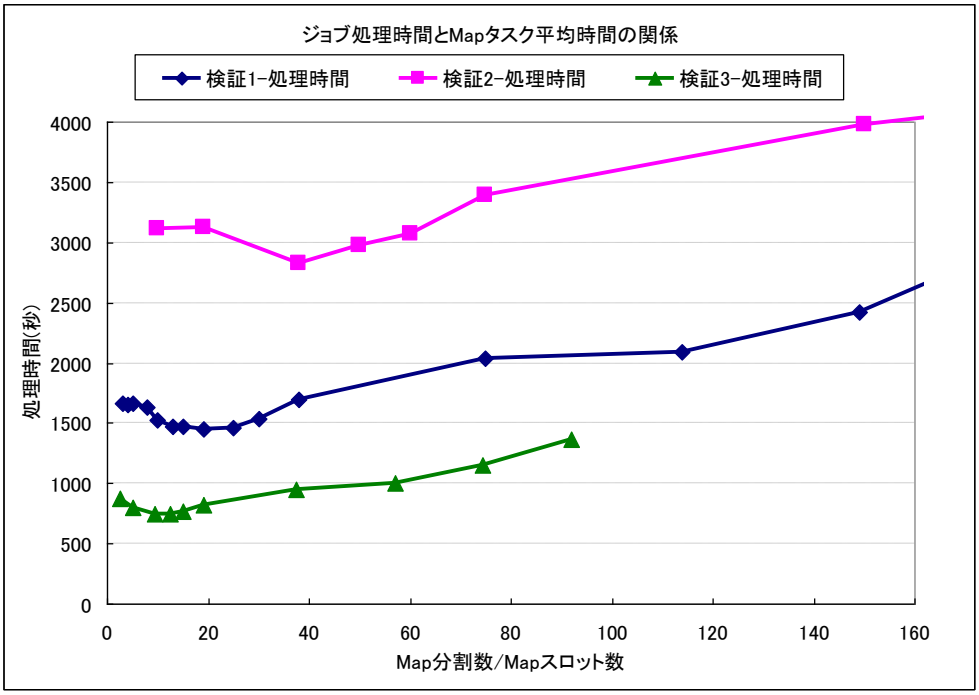


図 9-26 Map 処理分割影響調査結果(TeraSort)

この結果より、Map 分割数を極端に多くすると処理時間の増加が確認できた。しかし、PiEstimator のように計算量の増減や処理ノード数の変化による処理時間の影響が、Map スロット数の倍率だけでは説明できない。そこで幾つかの観点により Map 処理分割を検討する。

まず、Map 処理分割の変化によって、処理ノードのリソース使用状況がどのように

変化したか確認する。検証 1(図 9-27 再掲)のリソース使用状況を図 9-28 に示す。このとき、以下の 3 点でリソース使用状況を確認する。リソース使用状況で著しい変化を確認できた CPU 使用率と通信量について示す。

(1) Map 分割が十分でなく、処理時間が多少長い場面
(2) Map 分割によって処理時間が短い場面
(3) Map 分割が多すぎて、処理時間が長い場面

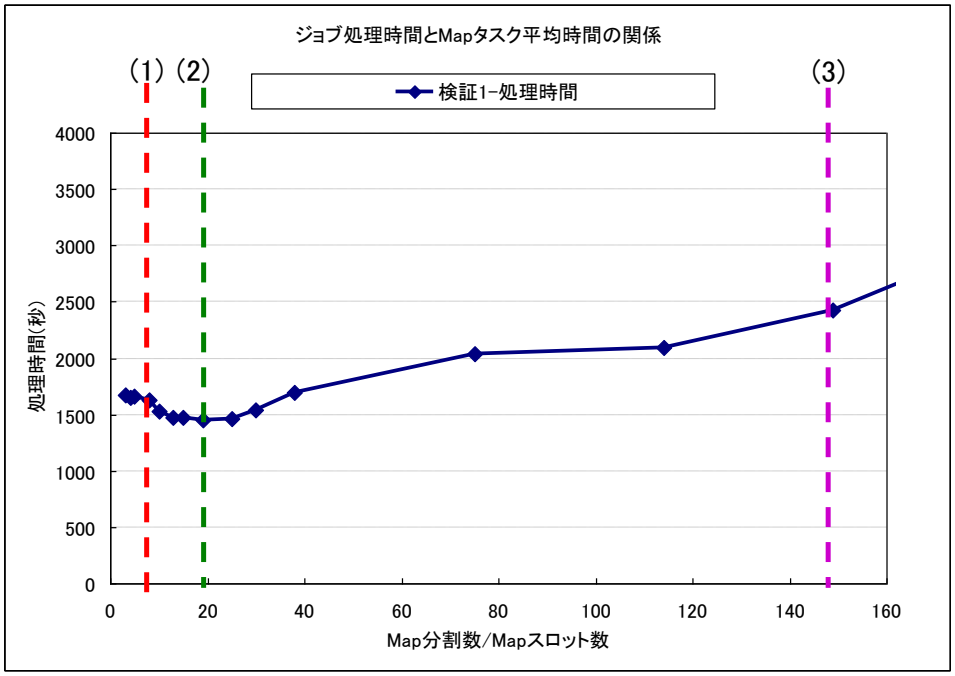


図 9-27 TeraSort 処理結果(検証 1・再掲)

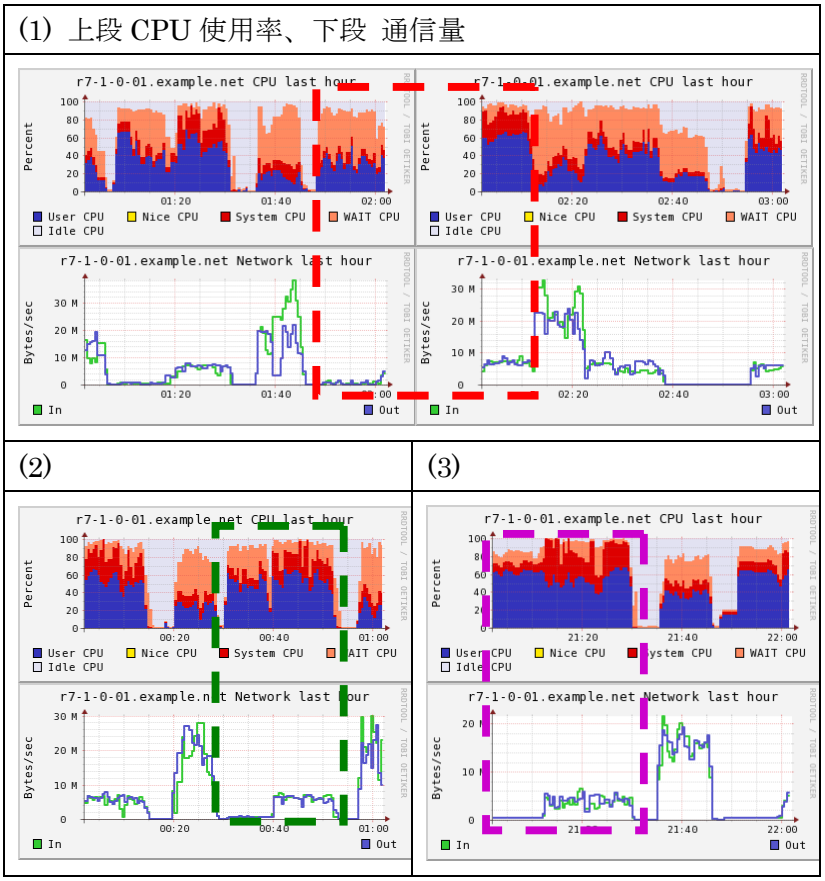


図 9-28 TeraSort でのリソース使用状況

このリソース使用状況より、以下の 2 点が分かる。

(1) Map 処理数が少ないケースでは、CPU の“WAIT CPU”が非常に高い期間がある。これは、ディスク IO に関係するものであるため、処理数が少ないとディスクアクセスに何らかの影響を与えている。

(2) Map 処理数が大きいケースでは、秒間あたりの通信量が他のケースに比べて少ない。MapReduce では、特に Shuffle で通信が発生するため、Shuffle に何らかの影響を与えている。

以上の(1)、(2)に対して、Hadoop としての動作について検証 1 の結果を踏まえて確認する。

まず、(1)は 1Map 処理あたりの処理量(ディスク書き込み量)から確認できる。図 9-29 に示すとおり、Map 処理の分割が小さいほど書き込み量と処理時間の関係が強い。この書き込み量増加の原因は、図 9-30 に示す Map 処理の“Spill Records”を確認することで判断できる。

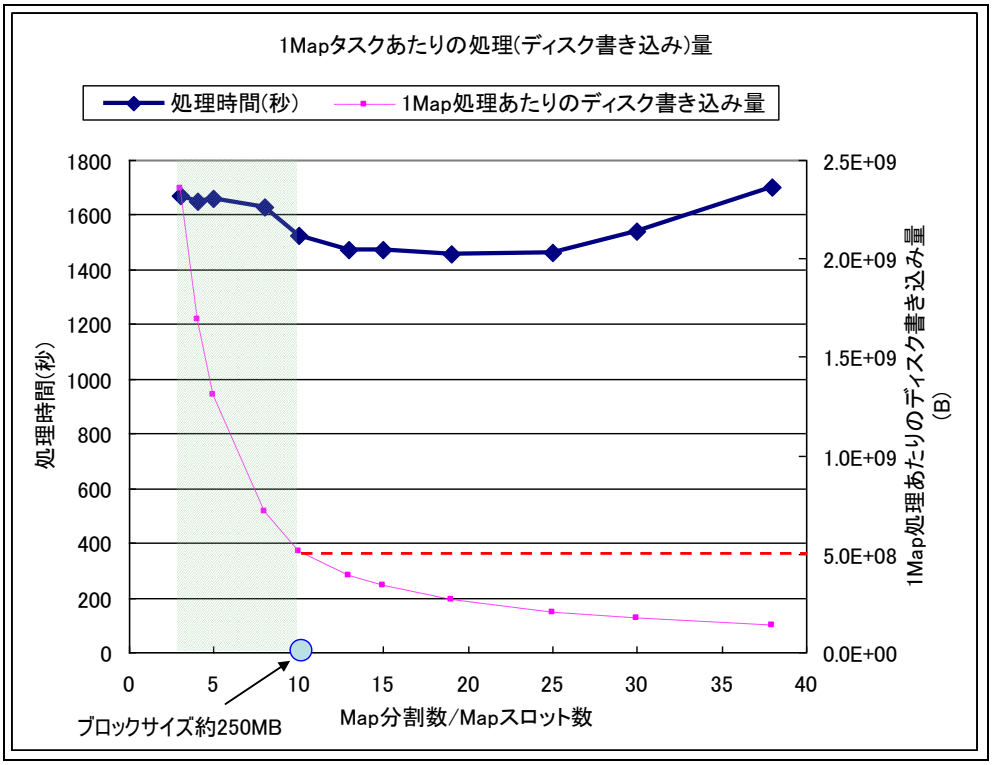


図 9-29 Map 処理分割とディスク使用量の関係（検証 1）

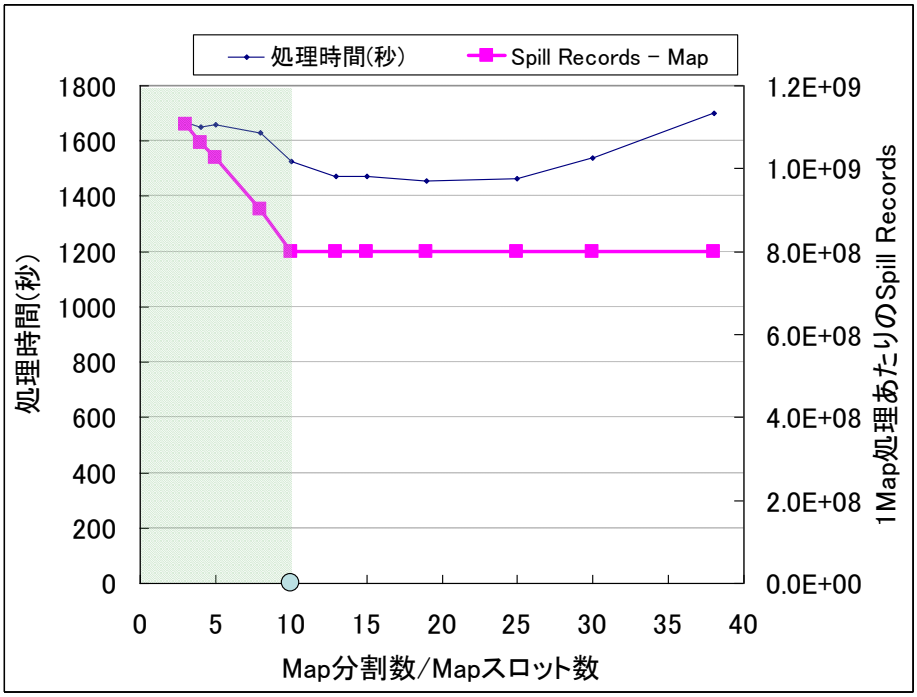


図 9-30 Map 処理分割と Spill Records の関係（検証 1）

Map 処理の“Spill Records”は、以下の操作によるディスクへの書き出しでカウントされる。以下の操作は、すべてディスク IO が発生する。

(1) Map 処理結果となるデータを格納する Data Buffer の容量が閾値を超えた場合

Data Buffer は、Hadoop パラメータである“io.sort.mb”で指定した数字分 JavaVM ヒープメモリ内に用意される(デフォルト約 100MB)。Map 処理結果となるデータは、Data Buffer に格納される。このとき、Data Buffer が閾値(Hadoop パラメータ“io.sort.spill.percent”で設定、デフォルト 0.8)を超えた場合(80MB 以上)、Spill 処理して Data Buffer、Record Buffer を空にする。

(2) Map 処理結果のレコード情報を格納する Record Buffer の容量が閾値を超えた場合

Record Buffer は、“io.sort.mb”で指定した領域から“io.sort.record.percent”÷16 としたものが JavaVM ヒープメモリ内に用意される(デフォルト 約 330KB)。Record Buffer は、Map 処理結果となるデータ数が閾値(Hadoop パラメータ“io.sort.spill.percent”)を超えた場合、Spill 処理して Data Buffer、Record Buffer を空にする。

(3) Map 処理結果を Reduce 処理単位に分割する場合

(1)や(2)の操作で Spill されたデータはディスク内でセグメントとして扱われる。セグメントは、Map 処理結果を Reduce 処理単位に分割する時にマージされるが、1 度にマージする単位(Hadoop パラメータ“io.sort.factor”)以上のセグメントがある場合に、複数のセグメントをマージして 1 つのセグメントとする。この操作によりまとめられたセグメント内のデータ量についてカウントされる。

図 9-28 で示した CPU 使用率と図 9-30 の Spill Records の関係より、Map 分割数が少ない場合に、1 つの Map 処理で扱うデータ量が処理に割り当てられたメモリサイズより大きくなる。そのため、ヒープメモリ上でデータを処理できずにディスクへの書き出しが多発する。

この事象を回避するためには、以下のように Map 処理を分割する。

- 1 つの Map 処理結果のサイズが Data Buffer の閾値を超えないように Map 処理を分割する。または、Data Buffer(“io.sort.mb”)、JavaVM ヒープメモリを拡張する。
- 1 つの Map 処理結果で Reduce に渡すレコード数が、Record Buffer の閾値を超えないように Map 処理を分割する。または、Record Buffer (“io.sort.record.percent”)を拡張する。
- 1 つの Map 処理結果をマージする場合に、1 度にマージするセグメント数を大きくする。

特に 1 レコードあたりのデータサイズが小さい場合、Record Buffer の閾値超えによるディスク書き出しが発生するため、MapReduce 処理での 1 レコード長に応じて

Record Buffer サイズを変更する。

図 9-30 より、Map 処理でのレコード数の 2 倍までであれば、Spill Records が発生しても性能は良い。つまり(3)の操作が発生しなければ Spill Records の影響は小さいといえる。一度にマージするセグメントはデフォルトの 10 なので、9 回までの Spill Out では(3)は発生しない。本検証の Spill Out は Record Buffer 起因であるため、“レコード数の閾値×1 レコード長×10” より Map 処理での書き出し量が小さければ良い。1 レコード長は 100 バイトなので、約 260MB 程度の書き出し量までは処理に極端な遅延が発生しないといえる。

次に、図 9-28(3)の Map 分割数が極端に多い場合、秒間あたりの通信量が減少している点について考察する。これは、Reduce 分割数を固定で測定していることから 1Reduce あたりの処理量は、どの場合でも同じとなる。つまり、Map 分割数が多いと Reduce 処理用のデータを収集するまでにかかる時間が延びることが分かる。検証 1 にて 1Reduce 処理あたりの Shuffle 時間に関する結果を図 9-31 に示す。

これは、Shuffle で Map 処理結果を一度に取得できる数を設定しているためである。また Hadoop では、Map 処理結果は一定時間間隔で取得するように実装されている。それらによるオーバヘッドは、Map 分割数が多くなることで顕著になり、Shuffle 時間の増加により全体としての処理時間が増加する。

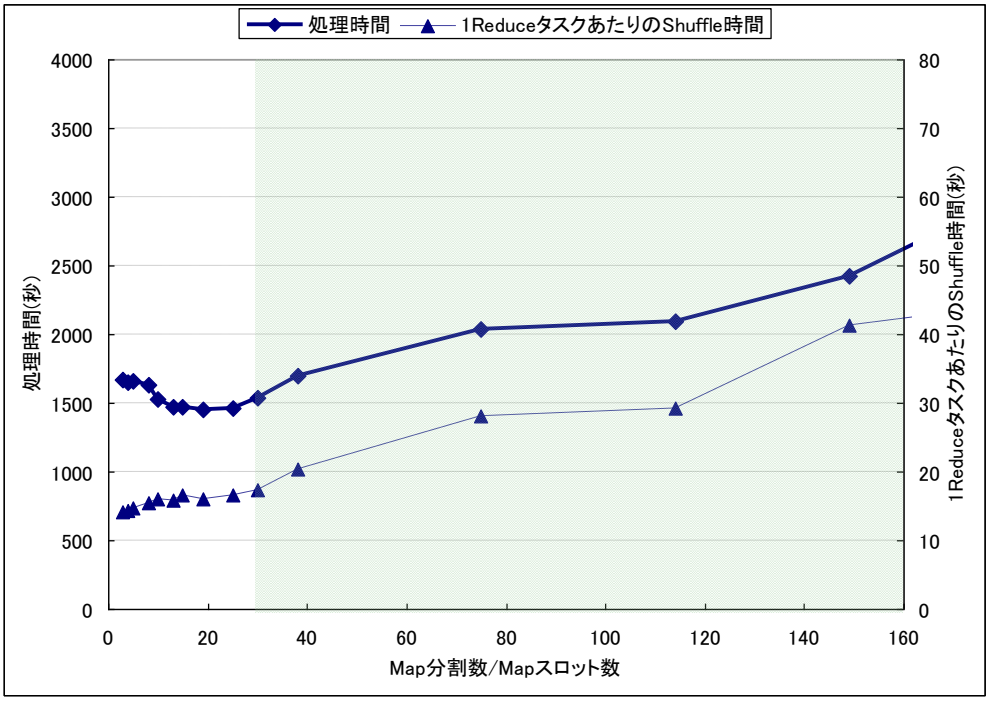


図 9-31 Map 分割数と Shuffle 時間の関係(検証 1)

この分割数の影響は、1Map 処理あたりの処理データ量で置き換えられる。検証 1

～検証 3 の処理データ量について図 9-32 に示す。

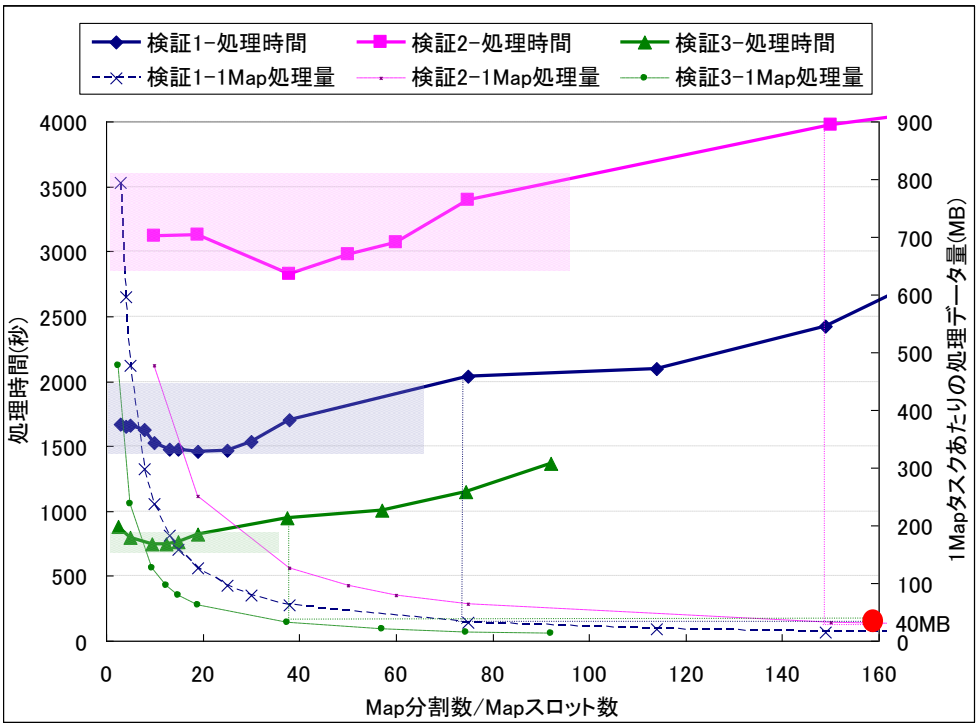


図 9-32 Map 分割数と Map 処理データ量の関係

図 9-32 では、各検証パターンともに斜線の枠で囲んだ最速処理時間からの 3 割長くなった処理時間の測定パターンまでを許容範囲とする。このとき 1Map 処理あたりの処理データ量と比較すると、1Map 処理あたりの処理データ量が約 40MB 程度よりも小さくなるときに MapReduce ジョブ全体としての処理時間が顕著に長くなる。これは、1 つの Map 処理で扱うデータサイズとより処理データ量が小さいため、キャパシティとして無駄が生じていると言える。本検証の 1Map 処理あたりのヒープサイズは 200MB であるので、40MB 程度のデータでは処理能力を活用していないことが分かる。

以上 PiEstimator と TeraSort の結果より、Map 処理分割は以下のことが述べられる。

- ディスクに書き出すような処理が発生しない場合、Map 分割を多くすると処理時間が長くなる。
- 1Map 処理で Reduce 処理に渡すレコードが、Map 処理で扱う Data Buffer、Record Buffer を超えるとディスクにデータを書き出す。ディスクに書き出す頻

度が多くなることで、ディスク IO が頻発して処理時間が延びる。

- 1Map 処理あたりの処理データ量が 1Reduce 処理に割り当てられた JavaVM ヒープメモリサイズよりも小さい(5 分の 1 以下)場合、リソースを活用しきれない。そのため、Shuffle 時間が長くなる。

なお Hadoop の Map 処理は、HDFS 上に格納されたブロック単位で処理される。Map 分割は、ブロック内のデータを分割して処理することになる。データの区切りがブロック間に跨る場合を除き、1 つの Map 処理が複数のブロックを処理することは無い。そのため、HDFS に格納するブロックサイズの設定を Map 処理分割方針に言い換えられる。

9.5.3.2 Reduce 処理分割方針

Reduce 処理は、ディスクのリソースを特に使用する。そのため、Map 処理の検討方法と同じく TeraSort によるディスクリソース消費に着目した処理分割を考える。

Reduce 処理も Map 処理の時と同様に、以下の方針とする。

“Reduce 処理分割で、特に制約が無い限り Reduce スロット数と同等以上の Reduce 分割数とする。”

以降で述べる検討結果を踏まえて、Reduce 処理分割方針を以下に示す。

(1) Shuffle をメモリ上で処理させるために、上限サイズを超えないように Reduce 分割数を設定する。
(2) Shuffle をメモリ上で処理させるための上限値を増やすために、JavaVM ヒープメモリ、Hadoop パラメータ(mapred.job.shuffle.input.buffer.percent)を増やす。

以降は、(1)、(2)で示した Reduce 処理分割方針を検討したものである。

Reduce 処理の分割数設定方針

Reduce 処理の分割数を以下の条件による検証で検討する。

- Map 処理数は一定、Reduce 分割数を Reduce スロット数の倍率で変化させる。
- 1 つの処理あたりの JavaVM ヒープメモリを 200MB とする。
- 検証 1: 処理ノード 8 台(S4: CPU2 コア), スロット数総数 16, 40GB 実行
- 検証 2: 処理ノード 8 台(S4: CPU2 コア), スロット数総数 16, 80GB 実行
- 検証 3: 処理ノード 16 台(S4: CPU2 コア), スロット数総数 32, 40GB 実行

Reduce 分割数とスロット数の倍率の関係について表 9-21 に示す。

表 9-21 Reduce 分割数とスロット数の倍率の関係

No.	Reduce 分割数	検証 1 倍率	検証 2 倍率	検証 3 倍率
1	8	0.5	0.5	0.25
2	16	1	1	0.5
3	32	2	2	1
4	64	4	4	2
5	128	8	8	4
6	256	16	16	8
7	512	32	32	16
8	1192	74.5	74.5	37.25

測定結果を図 9-33 に示す。図内の斜線の部分は、良い結果が得られたと判断するための基準として、各検証での最速処理時間の 2 割長い部分までを範囲としている。

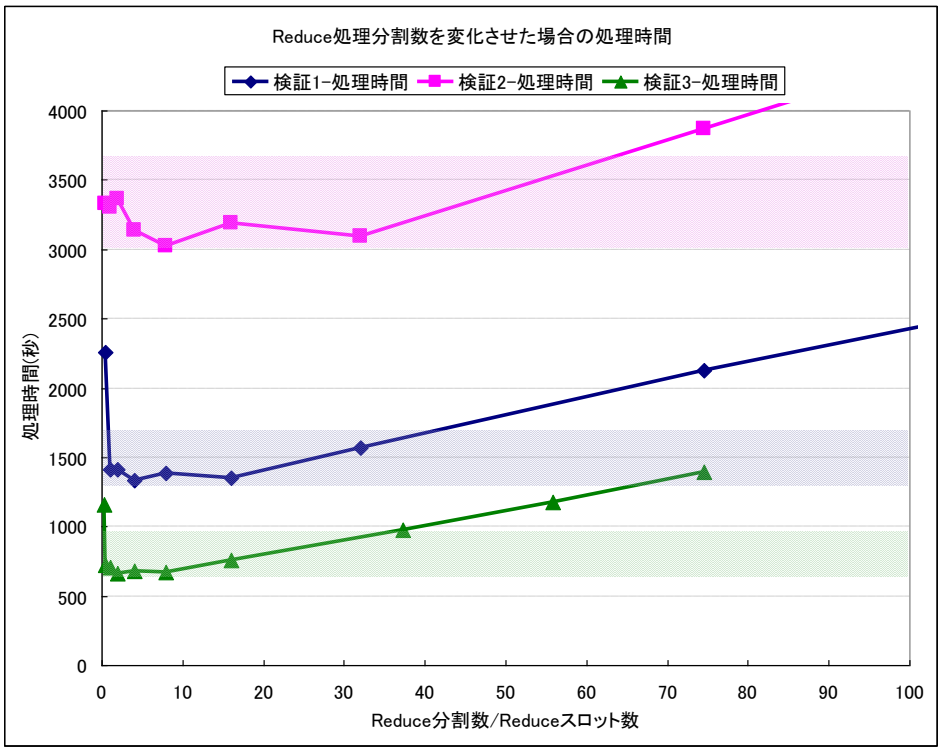


図 9-33 Reduce 処理分割による影響調査結果

この結果より、Reduce 分割数が極端に小さい場合と大きな場合は、TeraSort 処理時間が非常に長くなることが分かった。このときの処理ノードのリソース使用量につ

いて Map 処理分割の検証と同様に CPU 使用率と通信量に着目する。検証 3 でのリソース使用状況について、図 9-34、図 9-35 に示す。このときリソース情報を取得するポイントを 3 つに分類する。

- Reduce 分割数が Reduce スロット数と同程度～少ない場合((1)に該当)
- Reduce 分割数が Reduce スロット数の数十倍以上の場合 ((4)に該当)
- Reduce 分割数が Reduce スロット数の数倍～十数倍の場合 ((2),(3)に該当)

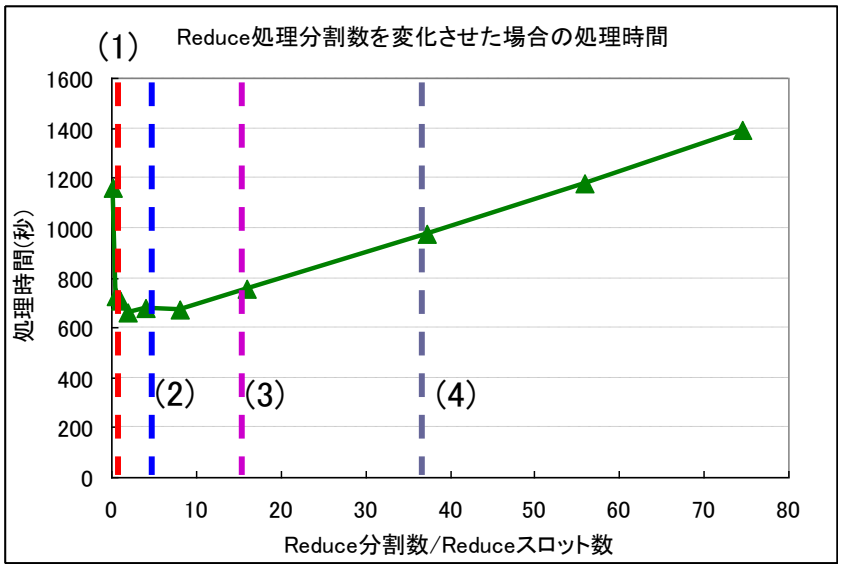


図 9-34 Reduce 処理分割数を変化させた場合の処理時間(検証 3)

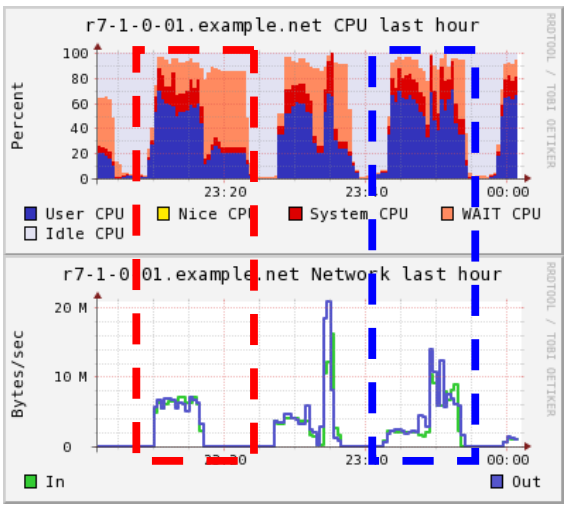
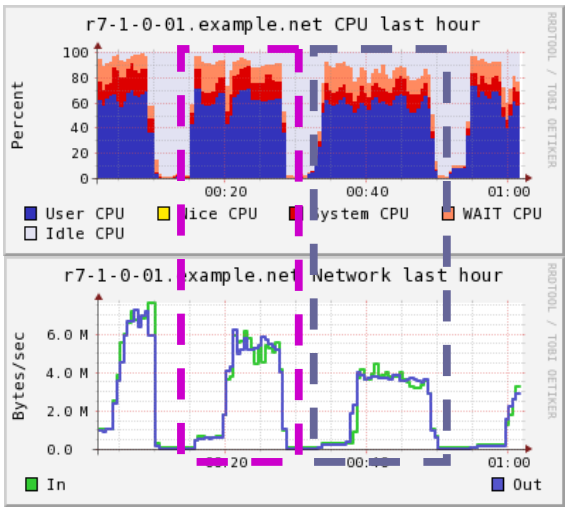
(1):赤枠、(2):青枠 に該当する部分	(3):紫枠、(4):灰枠 に該当する部分
	

図 9-35 Reduce 処理数分割に対するリソース使用状況(検証 3)

処理ノードのリソース使用状況から、以下のことが述べられる。

- Reduce 分割数が少ないと CPU の“WAIT CPU”が顕著になる。これは、ディスク IO 待ちで CPU が割り当てられているように見えるため、ディスクボトルネック状態であるといえる。
- 処理時間が短いケースでは、CPU リソースを十分利用している。
- Reduce 分割数が多くなると、秒間あたりの通信量が少なくなる。さらに、CPU 使用率で空き(“Idle CPU”)状態が増える。

Reduce 分割数が少ない場合は、図 9-36 の左側部分の斜線で示した領域のように、1Reduce 処理あたりの処理データ量が大きくなる。図 9-35 のリソース状況と突き合わせると CPU 使用率の欄で、ディスクアクセスに関係する“WAIT CPU”が大きな値となる。この影響は、Map 処理のときと同様に“Spill Records”で判断できる。Reduce 処理での“Spill Records”と処理時間の関係を図 9-37 に示す。

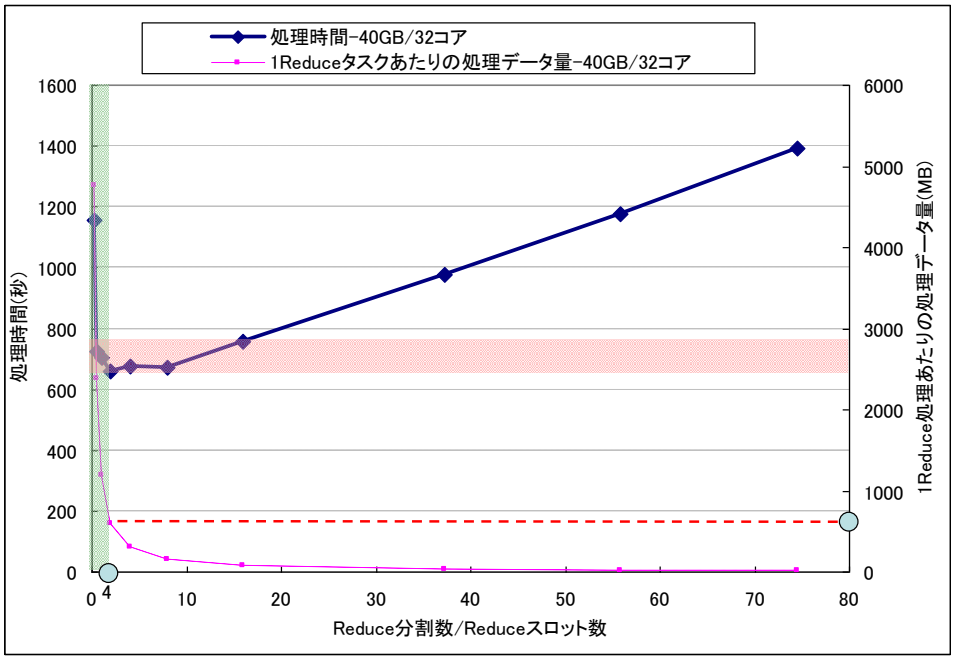


図 9-36 Reduce 分割数が少ない場合の影響(検証 3)

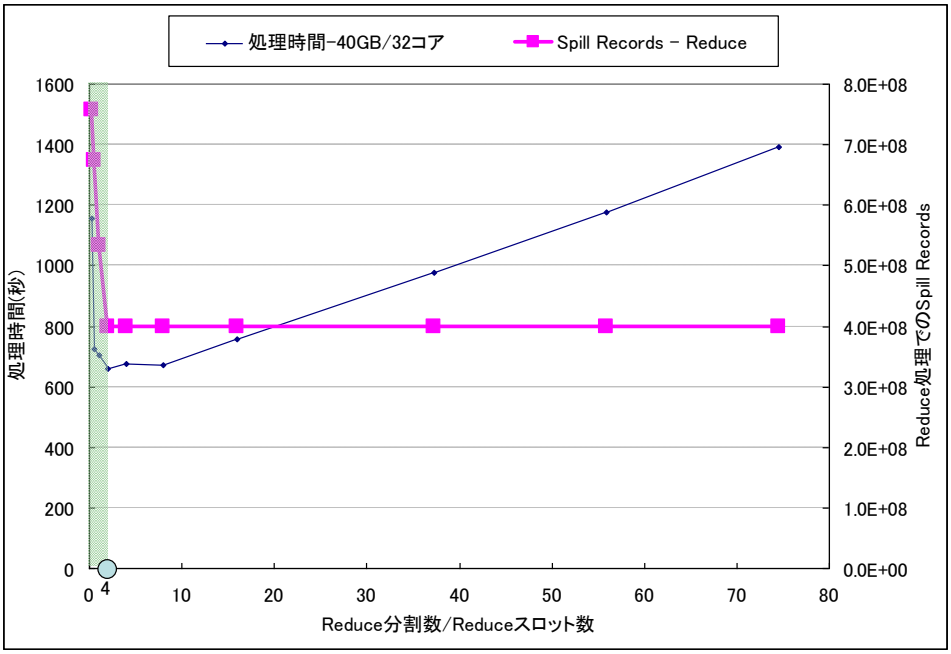


図 9-37 Reduce 分割数と Spill Records(検証 3)

図 9-37 より、Reduce 分割数が少ない場合は“Spill Records”の値も大きいことが確認できる。この事象は以下の理由により発生する。

Shuffle では、Map 処理結果をメモリかディスクに格納する。また、Map 処理結果のマージや Key でソートする場合に、メモリ上で処理するかディスクを使用して処理するか選択される。ディスクでマージする場合には、マージした中間ファイルを再びディスクに書き出す。“Spill Records”は、ディスクに書き出すレコード数をカウントするので、マージした中間ファイル分についてもカウントされる。

Map 処理結果をメモリ上で格納するかディスクに書き出す判断は、以下のように決定される。

(1) 1Reduce 処理あたりの JavaVM ヒープメモリ（デフォルト: 最大 200MB)
(2) mapred.job.shuffle.input.buffer.percent：Map 処理結果をメモリで格納するための Hadoop パラメータ（デフォルト:0.7)
(3) 1 つの Map 処理結果をメモリに格納するための閾値（0.25 で固定)

1 つの Map 処理結果メモリに割り当てる上限サイズ ＝(1)×(2)×(3)

デフォルトのヒープメモリサイズ(最大 200MB)の場合、この上限サイズは約 32MB となる。

本検証で Reduce 分割数が少ない場合、1 つの Map 処理結果が上限サイズ超えるため、ディスクでのマージとなり“Spill Records”の増加、CPU 使用率の“WAIT CPU”が大半を占めることになった。

なお、Reduce 処理で、“Spill Records”がカウントされる要因として、“mapred.job.reduce.input.buffer.percent”プロパティも影響する。これは、Reduce 処理開始時点で、Map 処理結果をメモリ上に残すかどうか判断するプロパティである。このプロパティは、デフォルト 0.0 であるため Reduce 処理時点で Map 処理結果全てをディスクに書き出すことになる。この書き出しによって“Spill Records”がカウントされる。

次に、Reduce 分割数が多くなると秒間あたりの通信量が少なくなる原因について考察する。これは、Reduce スロット数に対して Reduce 分割数が多くなると Shuffle による待ち時間だけが消費されてしまうためである。処理データ量よりも Shuffle でのオーバヘッドが顕著になるためである。検証 3 での処理時間と全 Reduce 処理での Shuffle に要した時間を図 9-38 に示す。

ここで述べる Shuffle 時間は、Map 処理と重複している部分を排除して計算したものである。

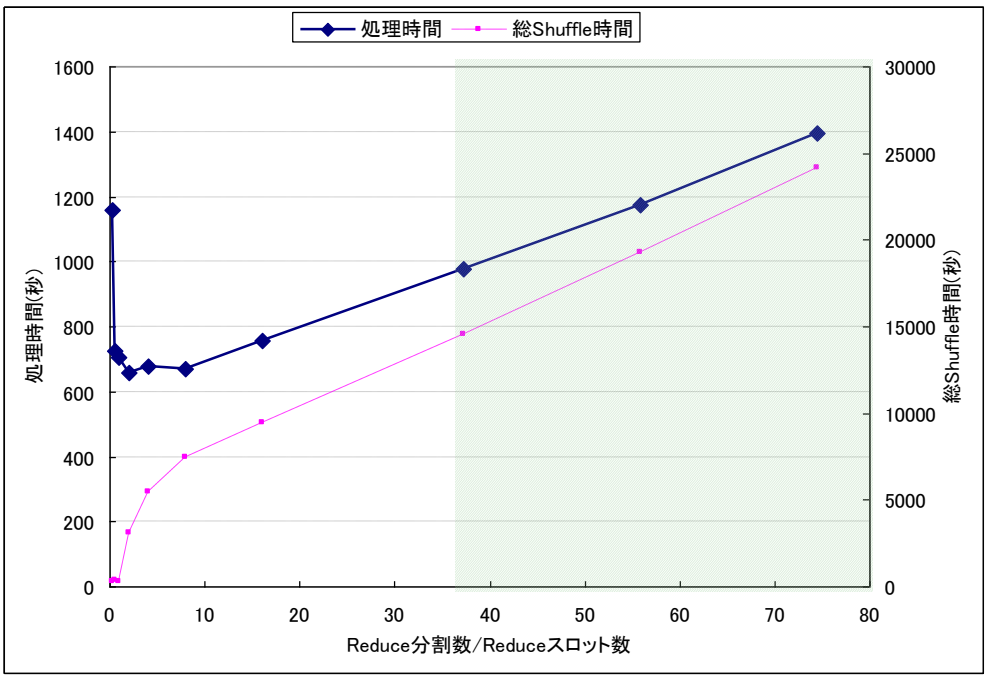


図 9-38 Reduce 分割数と総 Shuffle 時間(検証 3)

図 9-36 と図 9-38 より、Reduce 分割数が増加したにも関わらず、総 Shuffle 時間

は増加している。これは、Map 処理結果を取得する場合に Hadoop の実装により生じる待ち時間が影響している。そのため Reduce 分割数が大きくなると Shuffle 時間が増える。ただし、Reduce 分割数を Reduce スロット数の数十倍にしても処理時間は、最速処理時間の 2 倍にもならない。

本検証より Reduce 処理の分割で確認できることは以下の通りである。

- Reduce 処理で 1 つの Map 処理結果が Shuffle 時にメモリ上で扱える上限を超える場合は、ディスクに書き出される。そのため Map 処理結果をマージする場合もディスクで処理することになり、ディスク IO がボトルネックになるため、処理時間が長くなる。
- Reduce 分割数を Reduce スロット数の数十倍以上にすると Shuffle 時間が顕著になる。これは、Shuffle で Map 処理結果を受け取る場合に生じる待ち時間など Hadoop 実装による影響を受けるためである。

9.6 MapReduce 処理時間見積もり方法

本節は、MapReduce を実行するアプリケーションの処理時間を見積もる方法について説明する。処理時間を見積もる目的は、大量のデータを MapReduce ジョブで扱う場合に、どの程度の時間で完了するか実行しなければ処理時間を判断できないためである。そのため、図 9-39 に示すように事前に少量のデータを実行して基礎情報を確定し、大規模データを実行する場合の処理時間を見積もる。

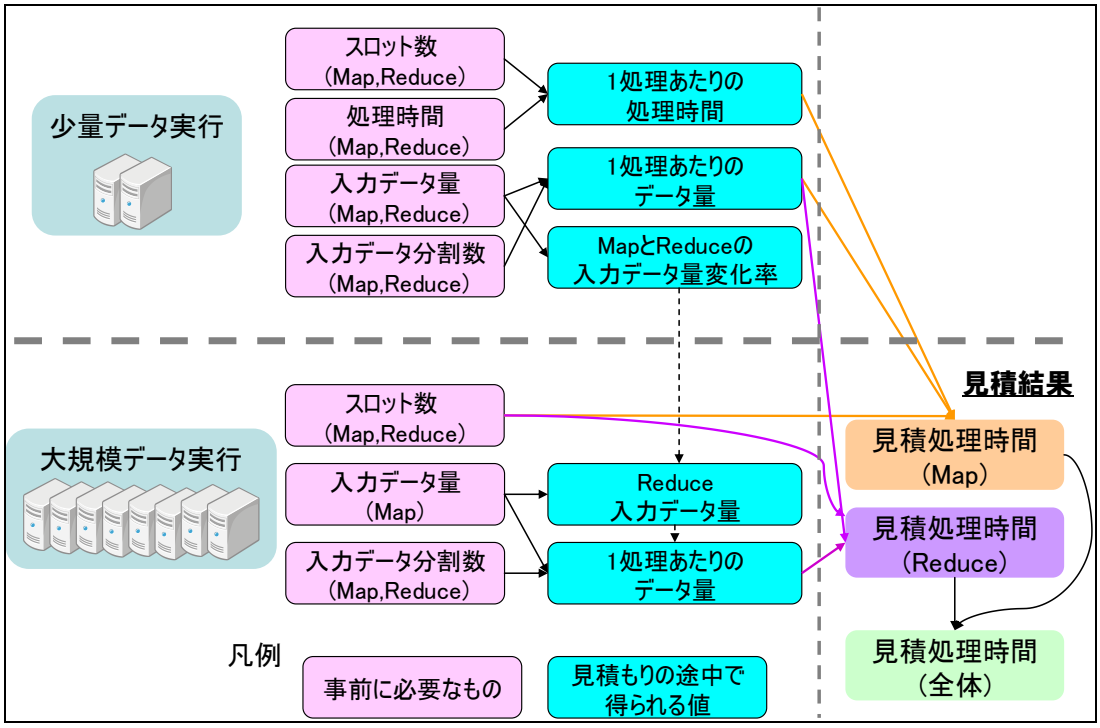


図 9-39 MapReduce 処理時間見積もりの考え方

9.6.1 少量データ実行により確定するパラメータ

まず、少量(数 10MB～数十 GB 程度)の入力データに対して、MapReduce ジョブを実行する。その結果、図 9-39 の少量データ実行部分で示すような、以下のパラメータを確定できる。

- Map 処理時間：少量データのジョブより Map 処理時間を設定する。
- Reduce 処理時間：少量データのジョブより Reduce 処理時間を設定する。
- Map 入力データ量：少量データを利用する。
- Reduce 入力データ量：少量データのジョブより Reduce 入力データ量を設定する。
- Map 分割数：Map 処理の分割数を設定する。
- Reduce 分割数：Reduce 処理の分割数を設定する。
- Map スロット数：Map 処理の多重度を設定する。
- Reduce スロット数：Reduce 処理の多重度を設定する。

Map 分割数や Reduce 分割数は、少量データ実行時の処理結果で確認できる。Map スロット数や Reduce スロット数は、MapReduce に関する Web 画面で確認できる。

9.6.2 MapReduce 処理時間見積もり式

前項までで確定したパラメータを元に、MapReduce ジョブの処理時間見積もり式を検討する。少量データ実行によるパラメータより、処理時間を算出する。

大量データ実行前に、以下の必要な情報を確認する。

- Map 入力データ量：大量データ実行時の Map 入力データ量
- Map 分割数：大量データ実行時の Map 分割数を確認する。
- Reduce 分割数：大量データ実行時での Reduce 分割数を確認する。
- Map スロット数：大量データ実行環境の Map スロット数
- Reduce スロット数：大量データ実行環境の Reduce スロット数

Map 処理時間見積もり式

Map 処理時間は、Map への入力データ量を元に以下の式より求める。

[少量]1Map あたりの処理時間 = [少量]Map 処理時間 ÷ ([少量]Map 分割数÷[少量]Map スロット数)

[大規模]Map 処理時間概算 = [少量]1Map あたりの処理時間 × [大規模]Map 分割数 ÷ [大規模] Map スロット数

[少量]と[大規模]1Map 処理のデータ量変化率 =([少量]Map 入力データ量÷[少量]Map 分割数) ÷ ([大規模]Map 入力データ量÷[大規模]Map 分割数)

Map 処理時間 ＝ [大規模]Map 処理時間概算 ÷ 1Map 処理のデータ量変化率

Reduce 処理時間見積もり式

Reduce 処理では、最初に少量データでの処理を元に以下の情報を決定する。

[大規模]Reduce 入力データ量 =[大規模]入力データ量 ÷ ([少量]入力データ量 ÷ [少量]Reduce 入力データ量)

算出した大規模での Reduce 入力データ量を利用して、以下の式より求める。

[少量]1Reduce あたりの処理時間 =[少量]Reduce 処理時間 ÷ ([少量]Reduce 分割数 ÷ [少量]Reduce スロット数)
[少量]1Reduce あたりの処理量 = [少量]Reduce データ入力量 ÷ [少量]Reduce 分割数

[大規模]1Reduce あたりの処理量 =[大規模]Reduce 入力データ量 ÷ [大規模]Reduce 分割数

[大規模]1Reduce あたりの処理時間 =[少量]1Reduce あたりの処理時間 × [大規模]1Reduce あたりの処理量 ÷ [少量]1Reduce あたりの処理量

Reduce 処理時間 =[大規模]1Reduce あたりの処理時間 × [大規模]Reduce 分割数 ÷ [大規模]Reduce スロット数

MapReduce 処理時間

Hadoop の MapReduce ジョブで Reduce 処理は、Map 処理の途中から開始される。Reduce 処理開始時の Map 処理完了率を踏まえて、MapReduce 処理時間を見積もる。Map 処理完了率は、“mapred.reduce.slowstart.completed.maps”プロパティで設定できる。デフォルトは、0.05 である。

MapReduce 処理時間 ＝ Map 処理時間 × Reduce 処理開始時の Map 処理完了率 + Reduce 処理時間

9.6.3 MapReduce 処理時間見積もり式の評価

本項では、前項までで定義した MapReduce 処理時間見積もり式をベンチマークとして利用した TeraSort に適用することで、見積もり式を評価する。

少量データ実行環境

少量データとして 40GB での TeraSort を実行する。少量データ試行により以下のパラメータを確定する。なお少量データ実行環境は、処理ノード 8 台(S4 :8 台)を使用する。

- Map 処理時間 ：571 秒
- Reduce 処理時間 ：1364 秒
- Map 入力データ量 ：4.0×10^10 Byte
- Reduce 入力データ量 ：4.0×10^10 Byte
- Map 分割数 ：608
- Reduce 分割数 ：256
- Map スロット数 ：16
- Reduce スロット数 ：16

大規模データ実行環境

大規模データ実行環境は、処理ノード 93 台(S1:17 台, S2: 4 台, S3: 16 台, S4:48 台, S5:8 台)のハードウェアスペックが混在する分散処理環境で 500GB の TeraSort を実行する。

- Map 入力データ量 ：5.0×10^11 Byte
- Map 分割数 ：7456
- Reduce 分割数 ：1300
- Map スロット数 ：260
- Reduce スロット数 ：260

Map 処理時間見積もり

少量データ試行によるパラメータ値から以下のように Map 処理時間を算出する。

[少量]1Map あたりの処理時間 = [少量]Map 処理時間 ÷ ([少量]Map 分割数÷[少量]Map スロット数)

= 571 ÷ (608 ÷ 16)

≒ 15.03 (秒)

[大規模] Map 処理時間概算 = [少量]1Map あたりの処理時間 × [大規模]Map 分割数 ÷ [大規模] Map スロット数

= 15.03 × 7456 ÷ 260

≒ 430.90 (秒)

[少量]と[大規模]1Map 処理のデータ量変化率 = ([少量]Map 入力データ量÷[少量]Map 分割数) ÷ ([大規模]Map 入力データ量÷[大規模]Map 分割数)

= (4.0×10^10 ÷ 608) ÷ (5.0×10^11 ÷ 7456)

≒ 0.98

Map 処理時間 =[大規模]環境での Map 処理時間概算 ÷ 1Map 処理のデータ量変化率

= 430.9 ÷ 0.98

≒439.7 (秒)

Reduce 処理時間見積もり

少量データ試行によるパラメータより、まず大規模環境での Reduce 入力データ量を決定する。

[大規模]Reduce 入力データ量 = [大規模]入力データ量 ÷ ([少量]入力データ量÷[少量]Reduce 入力データ量)

= 5.0×10^11 ÷ (4.0×10^10 ÷ 4.0×10^10)

= 5.0×10^11 (Byte)

次に、Reduce 入力データ量より Reduce 処理時間を求める。

[少量]1Reduce あたりの処理時間 = [少量]Reduce 処理時間 ÷ ([少量]Reduce 分割数÷[少量]Reduce スロット数)

= 1364 ÷ (256 ÷ 16)

= 85.25 (秒)

[少量]1Reduce あたりの処理量= [少量]Reduce データ入力量 ÷ [少量]Reduce 分割数

= 5.0×10^11 ÷ 256

≒ 1.56×10^8 (Byte)

[大規模]1Reduce あたりの処理量 = [大規模]Reduce 入力データ量 ÷ [大規

模]Reduce 分割数

= 5.0×10^11 ÷ 1300

≒ 3.85×10^8 (Byte)

[大規模]1Reduce あたりの処理時間 = [少量]1Reduce あたりの処理時間 × [大規模]1Reduce あたりの処理量 ÷ [少量]1Reduce あたりの処理量

= 85.25 × 3.85×10^8 ÷ 1.56×10^8

= 209.85 (秒)

Reduce 処理時間 = [大規模]1Reduce あたりの処理時間 × [大規模]Reduce 分割数 ÷[大規模]Reduce スロット数

= 209.85 × (1300 ÷ 260)

= 1049.25 (秒)

MapReduce 処理見積もり時間

Reduce 処理開始時の Map 処理完了率は 0.05(5%)である。

MapReduce 処理時間 = Map 処理時間 ÷ Reduce 処理開始時の Map 処理完了率 + Reduce 処理時間

= 439.7 × 0.05 + 1049.25

≒ 1071.2 (秒)

実際の測定結果

実際に 500GB の TeraSort を実行した。実行した結果は、以下の通りになった。

- Map 処理時間 ：588 秒（見積もり時間と約 34%の誤差)
- Reduce 処理時間 ：1325 秒（見積もり時間と約 26%の誤差)
- MapReduce 処理時間 ：1353 秒（見積もり時間と約 26%の誤差)

以上より、見積もり時間と実測時間では約 3 割程度の誤差となった。これは、大規模データ実行環境はハードウェアスペックが混在するサーバ構成での実行、少量データは 1 種類のハードウェアスペックのサーバ構成で実行であるため、混在環境での影響が誤差になっていると考える。

9.7 Hadoop 基盤の性能チューニング検討のまとめ

本節では、Hadoop 基盤の性能チューニングに関して考察と課題を述べる。

9.7.1 Hadoop 上の MapReduce 処理特性の把握

Hadoop での MapReduce 処理の流れを図示し、ハードウェアスペックが混在する Hadoop 基盤としてのボトルネック要因を示した。特に、処理ノードの実メモリ容量やディスク容量によって、Hadoop 基盤として本来達成できる性能に制約が生じることを説明した。

9.7.2 Hadoop 基盤としての性能チューニング

MapReduce ジョブに着目して、Hadoop 基盤を活用できるために CPU リソースやディスクに着目して、スロット数や Hadoop として使用するディスクを設定する Hadoop パラメータについて示した。PiEstimator や TeraSort を利用して TaskTracker の Map スロット数、Reduce スロット数の設定方針を決定した。

処理ノードに対して環境制約が無い場合、Map スロット数は、CPU コア数～CPU コア数×1.5 の範囲で、Reduce スロット数は、CPU コア数～CPU コア数+1 の範囲でスロット数を設定すれば、処理ノードのリソースを十分活用できることを確認した。また、処理ノードで使用するディスク台数は処理ノード内のすべてのディスクを利用することが良いことも確認した。

さらに Hadoop 基盤の制約として、処理ノードの実メモリ量やディスク容量に関する制約が、先に示した Map スロット数や Reduce スロット数やディスク台数の設定に対して影響を与えることについて示した。

9.7.3 MapReduce ジョブ実行に関する Hadoop チューニング

MapReduce アプリケーションを実行するときに、処理ノードのリソースでどのようなボトルネックが発生するかを示した。そして、MapReduce ジョブ実行時に Hadoop 基盤の全てのノードを活用するために、Map 処理の分割、Reduce 処理の分割について PiEstimator と TeraSort を利用して処理分割方針について示した。

9.7.4 MapReduce ジョブの処理時間見積もり方法

数 GB 程度までの少量データによる MapReduce ジョブ実行結果から、数十 GB 以上のデータを処理する場合の MapReduce 処理時間を見積もる手法を説明した。MapReduce ジョブとして TeraSort を利用して、少量データを小規模環境で実行した結果より、大規模環境での処理時間を見積もった。そして、実際に TeraSort を実行した結果と比較した。

見積もり方法の課題としては、混在するハードウェアスペックの情報を見積もり式

に反映することである。9.6 で示した見積もり式にその情報を追加することで、誤差を小さくできると考える。

10 Hadoop 基盤における管理ノード冗長化検討

本章では Hadoop 基盤の可用性向上を目的とした、Hadoop 管理ノードの冗長化についての検討結果を報告する。

Hadoop 基盤は処理ノードが故障しても、該当サーバを切り離すことにより縮退運転を行う仕組みを持つ。これは処理ノードのデータ・タスクが分散管理されていることに起因する。一方、データの管理情報(メタデータ)や処理全体を制御するプロセスは管理ノードにより集中管理されている。これらは、管理ノード停止時には Hadoop 基盤としての処理が継続できなくなるという事実を示している。

本章では、管理ノード冗長化構成を 2 つ提示する。また、提示した冗長化構成が求められる可用性を担保しうることを示す。HA クラスタ構成では、管理ノードの故障時に実行中のジョブは中断するが、切り替え後のジョブは継続して実行されることを確認する。ソフトウェア FT 構成では、管理ノード故障時にもジョブが中断されることなく切り替えが行われることを確認する。

10.1 管理ノード冗長化の必要性

Hadoop 基盤の管理ノードとして、HDFS へのアクセスを管理する NameNode、MapReduce 処理の状況を管理する JobTracker が 1 つずつ存在する。これは、Hadoop 基盤の制御情報が NameNode/JobTracker により集中管理されていることを示している。そのため、これら 2 つの管理ノードがどちらか 1 つでも停止してしまった場合、Hadoop 基盤全体の処理が停止する。

図 10-1 に HDFS へのデータ書き込み時の概略を示す。

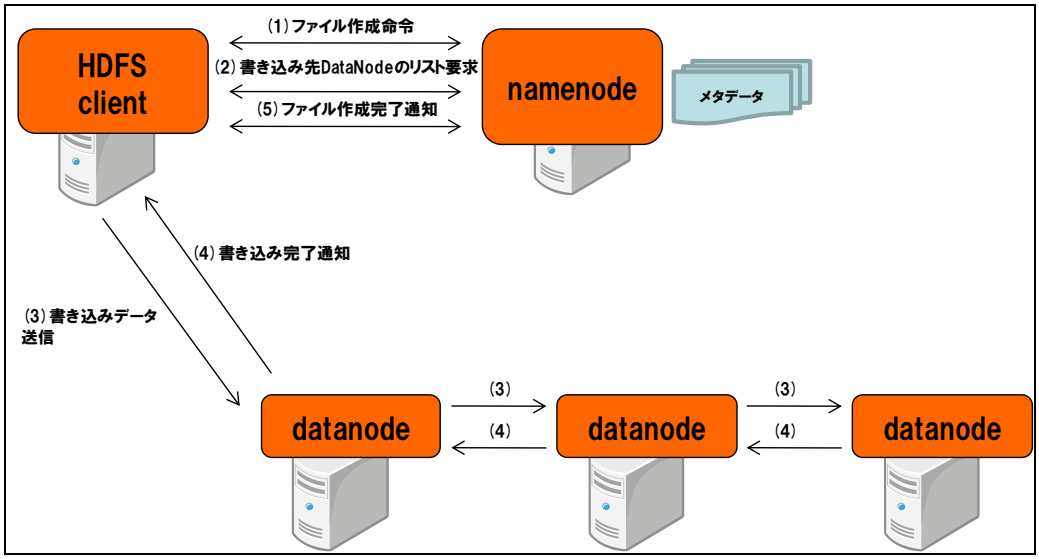


図 10-1 HDFS データ書き込みフロー

HDFS へのデータ書き込みは以下の手順で行われる。

(1) HDFS クライアントが NameNode に対して、ファイル作成命令を発行する。NameNode は指定されたファイルパス、クライアントの書き込み権限などを調べ、問題がなければメタデータの作成を行う。

(2) HDFS クライアントが NameNode に対して、レプリケーション先 DataNode のリストを要求する。NameNode はメタデータからレプリケーション先 DataNode の情報を抽出し、HDFS クライアントに渡す。

(3) HDFS クライアントがレプリケーション先 DataNode のリストの先頭にある DataNode に対して、書き込み情報を送信する。書き込み情報を受けとった DataNode は、次の DataNode に書き込み情報をフォワードする。

(4) DataNode が書き込みに対する完了通知を送信する。最終的には先頭の DataNode が HDFS クライアントに対して、書き込み完了通知を送信することとなる。

(5) HDFS クライアントが NameNode に対して、ファイル作成の完了通知を送信する。

NameNode へのアクセスは、HDFS クライアントから NameNode への通信により行われる。NameNode 停止時には、格納しているブロックと DataNode の関係を参照することができないため、HDFS 上のデータへのアクセスが不可能となる。

図 10-2 に MapReduce ジョブ実行時の概略を示す。

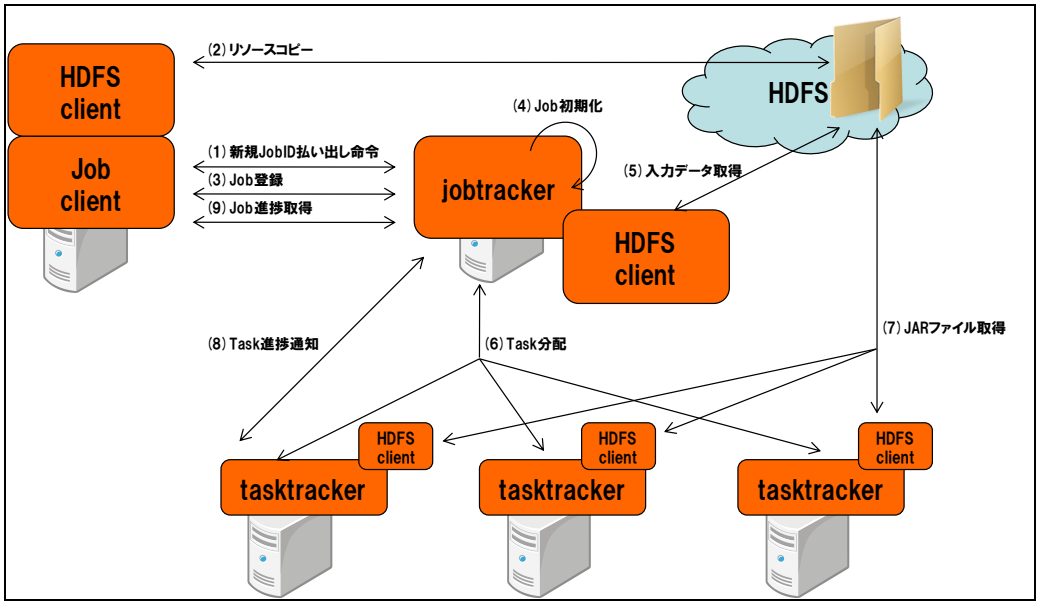


図 10-2 Map/Reduce ジョブ実行フロー

Job の実行は以下の手順で行われる。

(1) Job クライアントが JobTracker に対して、新規 JobID の払い出し命令を発行する。JobTracker は入出力のファイルパスなどを調べ、問題がなければ新規 JobID の払い出しを行う。

(2) (Job クライアントと同一マシン上の)HDFS クライアントが、Job 実行に必要なリソースを JobTracker 上のディレクトリにコピーする。Job 実行に必要なリソースは Job の JAR ファイル、設定ファイル、入力データなどである。コピーの際には、図 10-1 にて示したように NameNode と通信する必要がある。

(3) Job クライアントが JobTracker に対して、Job の登録を行う。Job の情報は JobTracker のメモリ上で管理される。

(4) JobTracker が登録された Job の初期化を行う。Job の初期化処理では、必要なオブジェクトの生成などが行われる。

(5) (JobTracker と同一マシン上の)HDFS クライアントが、入力データを取得する。取得後、入力データを割り振るための Task の生成を行う。Task には一意な TaskID が割り振られる。

(6) JobTracker が TaskTracker に対して、Task を分配する。Task は tasktracker からの Heartbeat パケットに返信する形で分配される。

(7) (TaskTracker と同一マシン上の)HDFS クライアントが、Job の JAR ファイルを取得する。JAR ファイル取得後、Task の実行を開始する。

(8) TaskTracker が JobTracker に対して、Task の状態を通知する。Task の状態として通知される情報は、TaskTracker の状態や Task の進捗などである。

(9) Job クライアントが JobTracker から、Job の状態を取得する。Job の状態として通知される情報は、JobTracker の状態や Job の進捗などである。

Job の実行は Job クライアントから JobTracker への通信により行われる。そのため、JobTracker 停止時には実行中のジョブが停止し、以降の新規 Job の投入が不可能となる。

以上のように管理ノードが停止すると Hadoop 基盤の処理全体が停止する。処理停止を防ぐためには、サーバ単体として部位を冗長化するとともに、サーバ複数台による冗長化構成をとる必要がある。

10.2 前提条件および冗長化構成の実現方針

10.1 にて述べたとおり、管理ノードがサーバ複数台による冗長化構成をとり、管理ノード故障時にも Hadoop 基盤が継続動作することがあるべき姿である。あるべき姿を目指すにあたり、今回は以下の事項を前提とする。

- 特殊なハードウェアではなくコモディティ製品による冗長化構成をとる。

- オープンソースソフトウェアを使用した冗長化構成をとる。
- 故障時の復旧処理が自動化された冗長化構成をとる。

あるべき姿を実現するための方針を以下に示す。なお、括弧内の見出し番号は対応する節を表している。

- 管理ノードにおいて、可用性を担保する対象を決定する。(10.3)
- 可用性を担保する冗長化方式の検討を行う。(10.4)
- 冗長化構成を実現し、対象の可用性が担保されていることを明らかにする。(10.5)

10.3 可用性を担保する対象

本節では、管理ノード故障時にも継続動作する Hadoop 基盤を実現するにあたり、可用性を担保するべき対象を決定する。図 10-3 に NameNode の構成要素を示す。

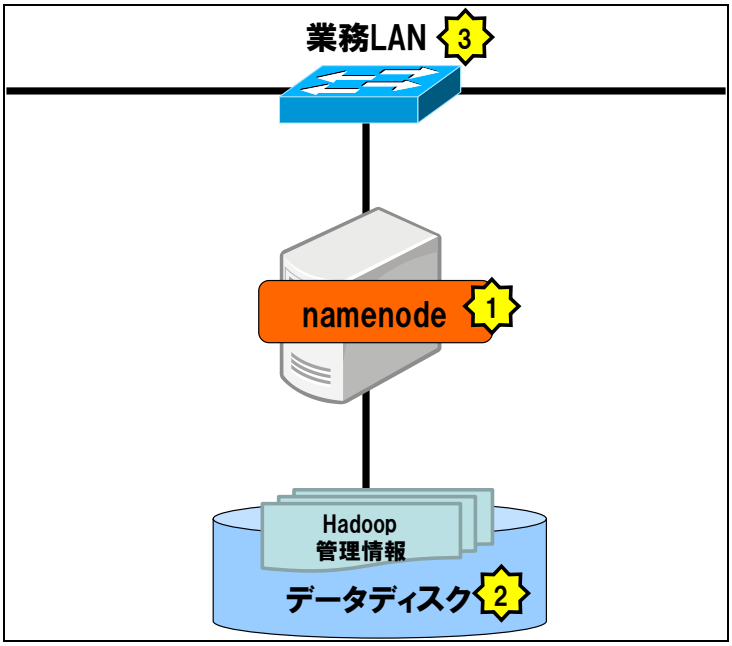


図 10-3 可用性を担保する対象

10.1 にて示したように、HDFS へのアクセスに当たり、namenode プロセスは業務 LAN を通して DataNode との通信を行い、NameNode のローカルディスクに記録された管理情報の読み書きを行う。これは、namenode プロセス、業務 LAN、管理情報の 3 つの内、どれか 1 つでも使用不可能となれば HDFS へのアクセスは実行できないということを示している。よって、可用性を担保するべき対象を、以下の 3 つと決定する。なお、JobTracker も同様に考えて、以下の 3 つを対象とすることとする。

- プロセス
- 永続データ
- ネットワーク

10.3.1 プロセスの可用性

管理ノード故障の際、Hadoop 基盤の処理に必要なプロセスが存在しないという事態が起こりうる。よって、プロセスの可用性が担保された状態を「管理ノードに故障が発生した場合においても、処理に必要なプロセスが使用可能である」状態と定義する。

なお、ここでのプロセスとは、必ずしも Hadoop 基盤固有のプロセスに限った話ではないことに留意する。つまり、処理に必要なプロセス全てについて、可用性が担保されなくてはならない。

10.3.2 永続データの可用性

管理ノード故障の際、Hadoop 基盤の処理に必要な情報にアクセスできないという事態が起こりうる。よって、永続データの可用性が担保された状態を「管理ノードに故障が発生した場合においても、処理に必要なデータへのアクセスが保障されている」状態と定義する。

10.3.3 ネットワークの可用性

管理ノード故障の際、管理ノード-処理ノード間の通信が途絶えるという事態が起こりうる。よって本章では、ネットワークの可用性が担保された状態を「管理ノードに故障が発生した場合においても、管理ノード-処理ノード間の通信路が確保されている」状態と定義する。

10.4 冗長化方式の検討

10.2 にて示した前提より、今回目指すべき冗長化構成では故障時の復旧処理が自動化されている必要がある。故障時の復旧が自動で行われる方式として、一般的に以下の 3 つの方式が考えられる。表 10-1 にそれぞれの方式の特徴を示す。

表 10-1 ハードウェア冗長化方式

No.	方式	特徴
1	HA クラスタ方式	同じ構成のサーバを現用系と待機系の 2 台用意し冗長化構成をとる。 現用系のハードウェアあるいはアプリケーショ

No.	方式	特徴
		ンに故障が発生しても、待機系でアプリケーションが自動的に起動する。
2	ハードウェア FT 方式	完全に二重化したハードウェアを用い、CPU 命令レベルで常に同期をとる。 一方のモジュールで故障が発生しても、残りのモジュールが正常に動作しているため、OS やアプリケーションには影響を与えない。
3	ソフトウェア FT 方式	仮想化技術を使用して冗長性を確保する。具体的には、仮想マシンを二重化し、同期制御する。 サーバで故障が発生しても、対向サーバで故障を検知し、自動で切り離しを行う。

10.2 にて示した前提より、特殊なハードウェアが必要なハードウェア FT 方式は対象外とする。よって、HA クラスタ方式とソフトウェア FT 方式について検討することとする。本報告書では、具体的な構成として以下の 2 つを扱う。

- Heartbeat と DRBD による HA クラスタ構成
- HA クラスタ構成と Kemari によるソフトウェア FT 構成

10.4.1 可用性を担保する技術

HA クラスタ方式・ソフトウェア FT 方式の実現にあたり、可用性の担保を目的として用いられる技術を以下に示す。

10.4.1.1 Heartbeat

Heartbeat とは、オープンソースの HA クラスタソフトウェアである。Heartbeat を導入することで、ハードウェアやソフトウェアに故障が発生した場合、自動的に予備機に切り替わり、サービスの中断を最小限に抑えることができる。

Heartbeat の持つ機能を次に示す。

- 現用機と予備機の 1 対 1 の冗長化構成、N 対 1 の冗長化構成
- サービスの起動、停止、監視
- ネットワーク経路の監視
- 故障発生を自動的に検知しフェールオーバの実施
- スプリットブレイン時における両系マウントの防止
- ディスクパスの監視
- リソース停止故障サーバの電源リセット

サービス故障時の Heartbeat 動作イメージを図 10-4 に示す。現用系/待機系で起動している heartbeat プロセスが一定間隔で生存確認の通信を行うことにより、電源断などの検知を行う。またサービス提供に必要なプロセスやネットワークインターフェースカード(NIC)・ディスクなどのハードウェアを監視対象として設定することで、サービス故障を検知することも可能である。Heartbeat は、故障を検知した後に待機系への自動切り替えを行う。

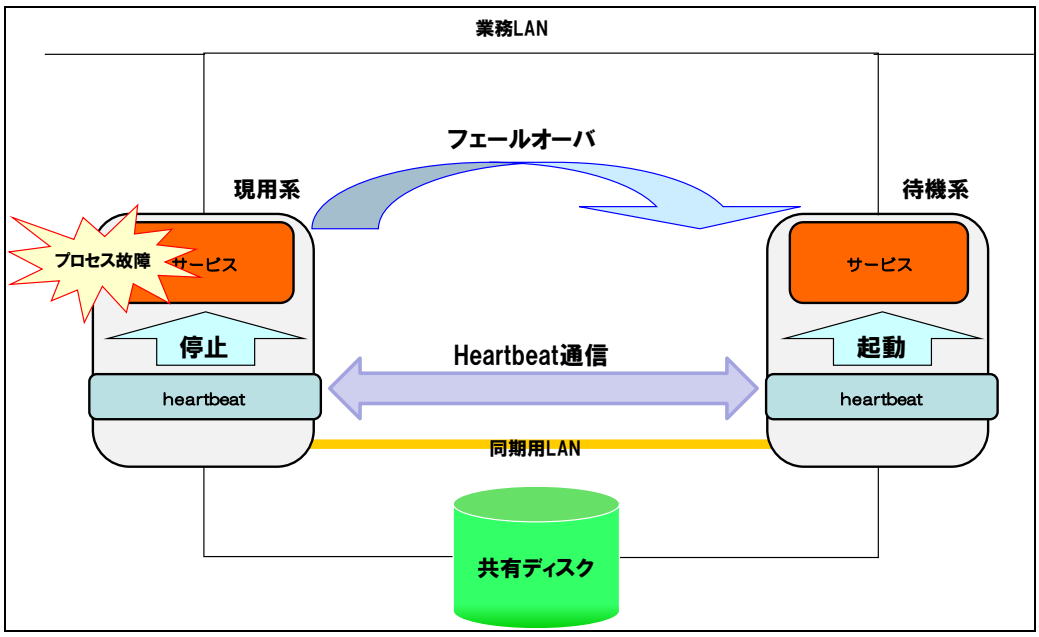


図 10-4 Heartbeat 動作イメージ

10.4.1.2 Kemari

Kemari とは仮想化を使用した環境でサーバ故障発生時に高速なフェールオーバを行うことができるオープンソースソフトウェアである。Kemari 自身はメモリ同期の機能のみを保持するため、故障時に切り替えを行うには HA クラスタソフトウェアの併用が必要である。

Kemari の特徴を次に示す。

- Xen 上に構築された仮想マシンを 2 台のサーバで同期させ冗長化構成をとる
- 同期用 LAN を通して、仮想マシンの実行状態をメモリ状態を含めて同期する
- ディスク I/O、ネットワーク I/O を別サーバに同期する
- 同期対象データは更新差分のみ
- 特殊な CPU 機能を必要とせず、ハードウェアの制約が少ない

Kemari の動作イメージを図 10-5 に示す。メモリ状態が同期されるということは、

メモリ上で動作しているアプリケーションの状態が同期されるということである。そのため、ハードウェア故障時にもアプリケーションの処理が中断することなく継続動作することができる。ただし、Heartbeat のような故障検知・自動切り替えの機能を持たないため、フェールオーバは手動で行う必要がある。

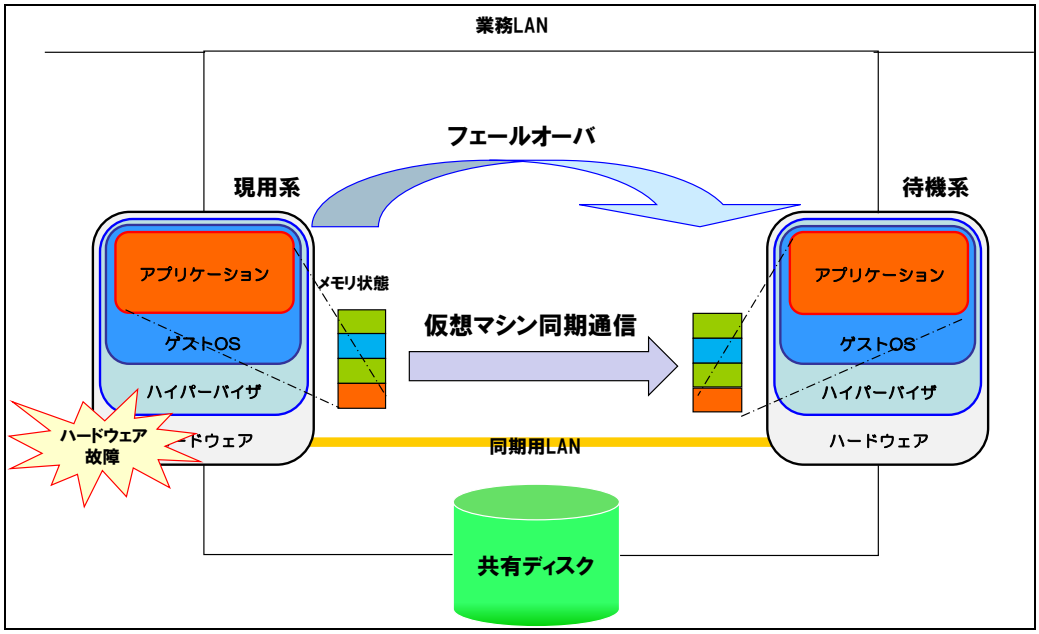


図 10-5 Kemari 動作イメージ

10.4.1.3 DRBD

DRBD とは、特定のディスクパーティションをネットワーク経由で他のサーバにレプリケーションすることができるオープンソースソフトウェアである。

DRBD の持つ機能を次に示す。

- リアルタイムのレプリケーション
- 単一プライマリモード、デュアルプライマリモードの構成
- ネットワーク帯域と信頼性を考慮した同期/非同期レプリケーションモード
- 複数の転送プロトコル
- 効率的なデータ同期

DRBD によるレプリケーションのイメージを図 10-6 に示す。現用系へのディスク書き込みが待機系にネットワーク経由で転送され、ディスクが同期される。DRBD は共有ディスクのようなサーバ間でのデータ引き継ぎを行うことができる。また、現用系/待機系のローカルディスクで RAID を構築することにより、更に障害耐性を高めることが可能である。

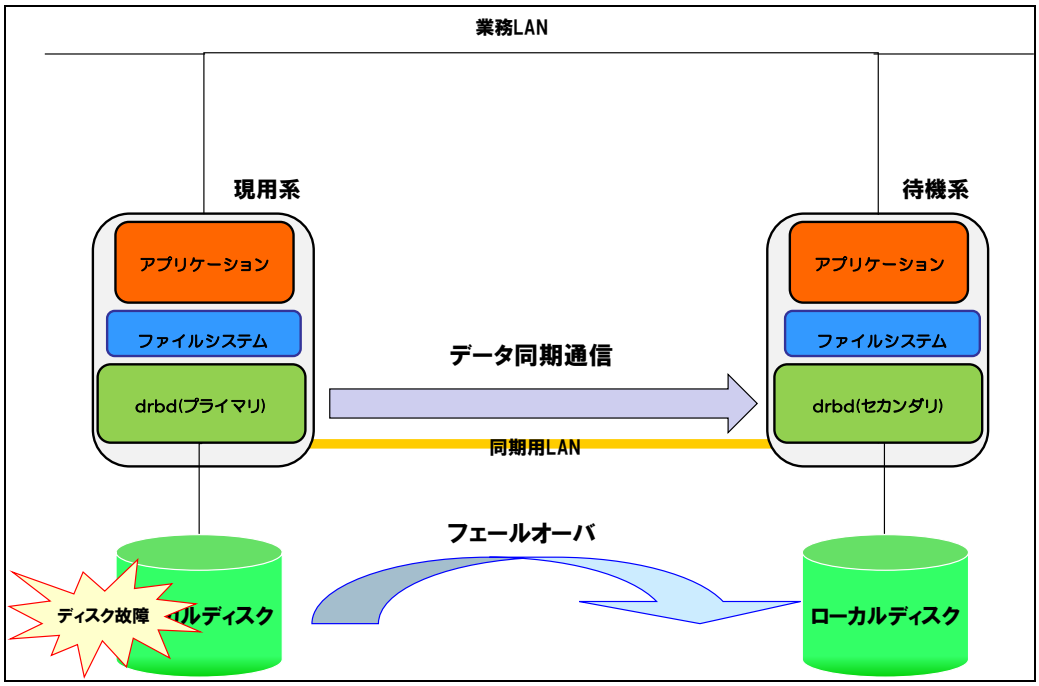


図 10-6 DRBD 動作イメージ

10.4.2 Heartbeat と DRBD による HA クラスタ構成

Heartbeat による HA クラスタ構成では、フェールオーバ前とフェールオーバ後に同一データへのアクセスを保障する必要がある。同一データへのアクセスの保証のために、ストレージとして共有ディスクが用いられるケースが多い。しかし 10.2 にて示したように、特殊なハードウェアを使用せずにオープンソースソフトウェアを使用した冗長化構成を目指すという前提がある。そのため、共有ディスクの代わりに DRBD をストレージとして用いる構成を採用した。Heartbeat と DRBD による HA クラスタ構成を図 10-7 に示す。

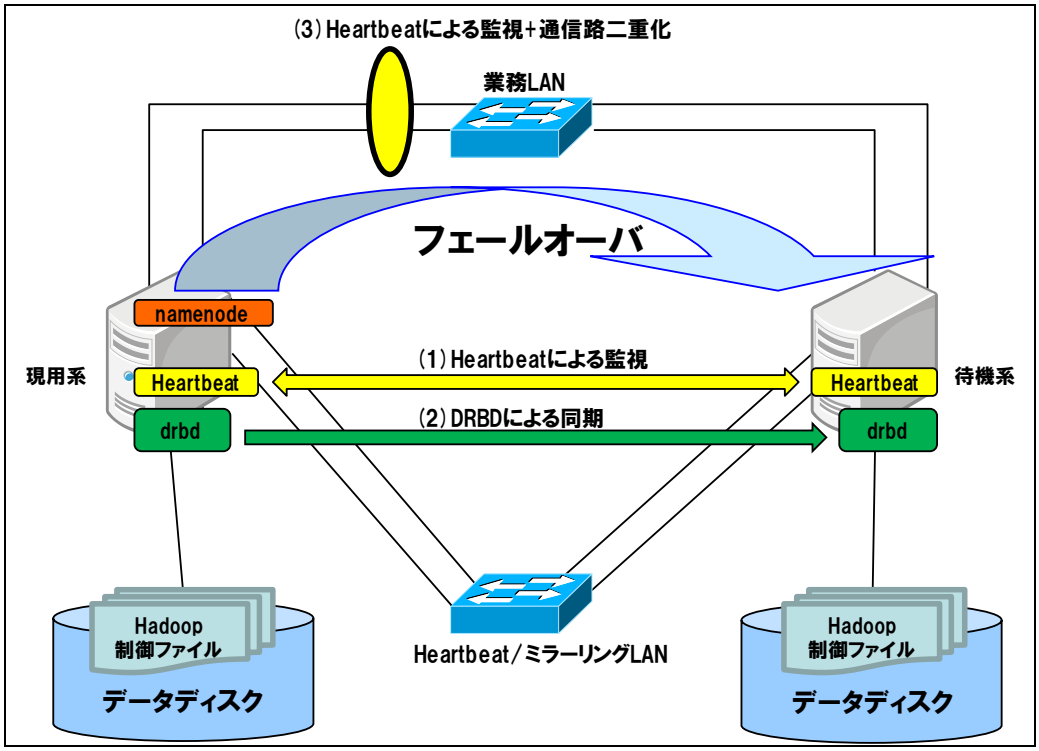


図 10-7 HA クラスタ構成

本構成では、10.3 にて示した対象の可用性を、以下のように担保する。

(1) Heartbeat による Hadoop 管理プロセスの監視を行うことで、プロセスの可用性を担保する。

(2) DRBD により Hadoop 制御ファイルのミラーリングを行うことで、永続データの可用性を担保する。

(3) Heartbeat による業務 LAN の監視および通信路の二重化を行うことで、ネットワークの可用性を担保する。

なお、NameNode、JobTracker ともに図 10-7 の構成をとる。

10.4.3 HA クラスタ構成と Kemari によるソフトウェア FT 構成

10.4.2 にて示した HA クラスタ構成に Kemari を加えることで、ソフトウェア FT 構成を実現することができる。物理マシンレベルのクラスタを構築する HA クラスタ構成に対して、ソフトウェア FT 構成は仮想マシンレベルのクラスタを構築する。

Kemari によるソフトウェア FT 構成の最大の特徴は、Hadoop ジョブを始めとする実行中の処理が無停止で継続されるという点である。

Kemari によるソフトウェア FT 構成を図 10-8 に示す。

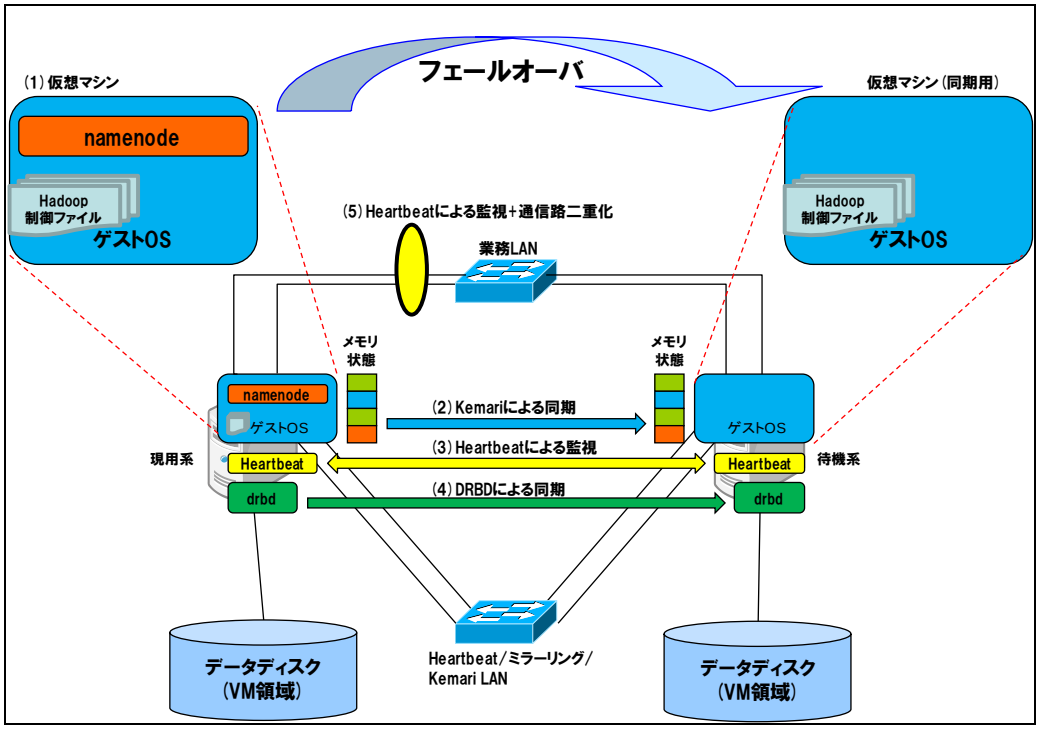


図 10-8 ソフトウェア FT 構成

本構成では、10.3 にて示した対象の可用性を、以下のように担保する。

(1) Hadoop 管理プロセスを仮想マシン上で起動する。

(2) 仮想マシンの実行状態が Kemari プロセスにより同期される。メモリ状態を含む実行状態の同期を行うため、フェールオーバ時に実行中の処理が無停止で継続する。

(3) Heartbeat による Kemari プロセスの監視を行うことで、プロセスの可用性を担保する。

(4) DRBD による仮想マシン領域全体のミラーリングを行うことで、永続データの可用性を担保する。なお仮想マシン領域には Hadoop 制御ファイルも含まれることに留意する。

(5) Heartbeat による業務 LAN の管理および通信路の二重化を行うことで、ネットワークの可用性を担保する。

なお、NameNode、JobTracker ともに図 10-8 の構成をとる。

10.5 冗長化構成の実現と検証

本節では、10.4 にて示した HA クラスタ構成とソフトウェア FT 構成についての実現方法を示す。その後、可用性が担保されていることを確認するための検証実験を行い、その結果を示す。

10.5.1 HA クラスタ構成の実現と検証

Heartbeat と DRBD による HA クラスタ構成の実現および検証実験の結果について、以下に示す。

10.5.1.1 保護対象のリソース

HA クラスタ構成において 10.3 にて示した可用性を担保するために、保護しなくてはならないリソースを表 10-2 に示す。

表 10-2 リソース一覧(HA クラスタ構成)

No.	担保対象	保護リソース	用いられる技術
1	プロセス	・Hadoop プロセス ・DRBD プロセス	・Heartbeat
2	永続データ	・管理ノードが使用するデータ領域(ファイルシステム)	・DRBD
3	ネットワーク	・仮想 IP ・ネットワークインターフェース	・Heartbeat ・Bonding(注)

(注) Bonding とは、複数の物理ネットワークインターフェースを単一の論理ネットワークインターフェースとしてまとめることで冗長化する技術である。

DRBD は両サーバで起動するため、Heartbeat のマスター/スレーブ方式により管理する。それ以外のリソースをリソースグループとすることで、グループ内のいずれかのリソースに故障が発生した場合、全てのリソースをフェールオーバさせることができる。また、リソースグループの起動前に DRBD がプライマリとなるように Heartbeat の制約を設定する。リソースグループのイメージを図 10-9 に示す。

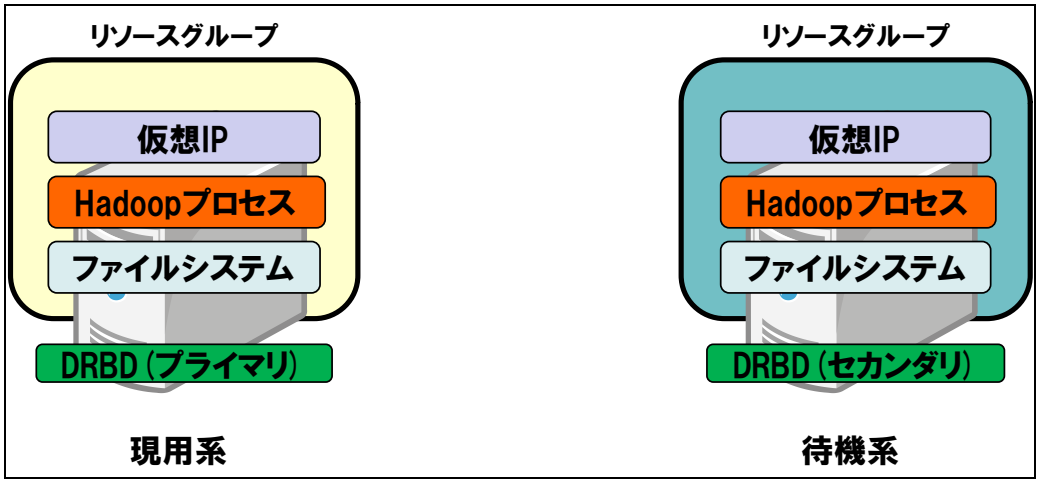


図 10-9 リソースグループ(HA クラスタ構成)

10.5.1.2 プロセスの可用性に関わる設計

HA クラスタ構成の内、プロセスの可用性に関わる箇所について以下に示す。

Hadoop プロセスのリソースエージェント

Heartbeat により Hadoop プロセスを管理するためには、リソースエージェント(RA)と呼ばれるスクリプトを作成する。

スプリットブレイン対策

HA クラスタではハートビート LAN 切断時など、互いのサーバが認識できなくなった場合に両系のサーバでリソースを起動するスプリットブレイン状態が発生する。スプリットブレイン状態ではサービスの継続は不可能であるため、対策を検討する必要がある。Heartbeat ではスプリットブレインを防止する方法として以下の 3 つが考えられる。

- VIPCheck による仮想 IP の衝突防止処理
- SFEX による共有ディスクのロック処理
- STONITH による強制再起動処理

今回はリソース停止失敗時の対策としても導入が必要な STONITH をスプリットブレインの対策として採用する。STONITH によるサーバ停止を確実に実施するには、OS とは別の経路で電源を操作できる機能があることが望ましい。しかし本機能は HP サーバでは iLO2、IBM サーバでは IMM というようにハードウェアベンダごとに実装が異なっている。10.2 にて示した前提より、今回はハードウェアに依存しない方式を採用する。そのため、OS に ssh でログインした後に STONITH 機能を使用することとする。

リソース停止失敗時の対策

前述したとおり、STONITH はリソース停止失敗時の対策としても用いられる。リソース停止失敗時の STONITH 動作イメージを図 10-10 に示す。

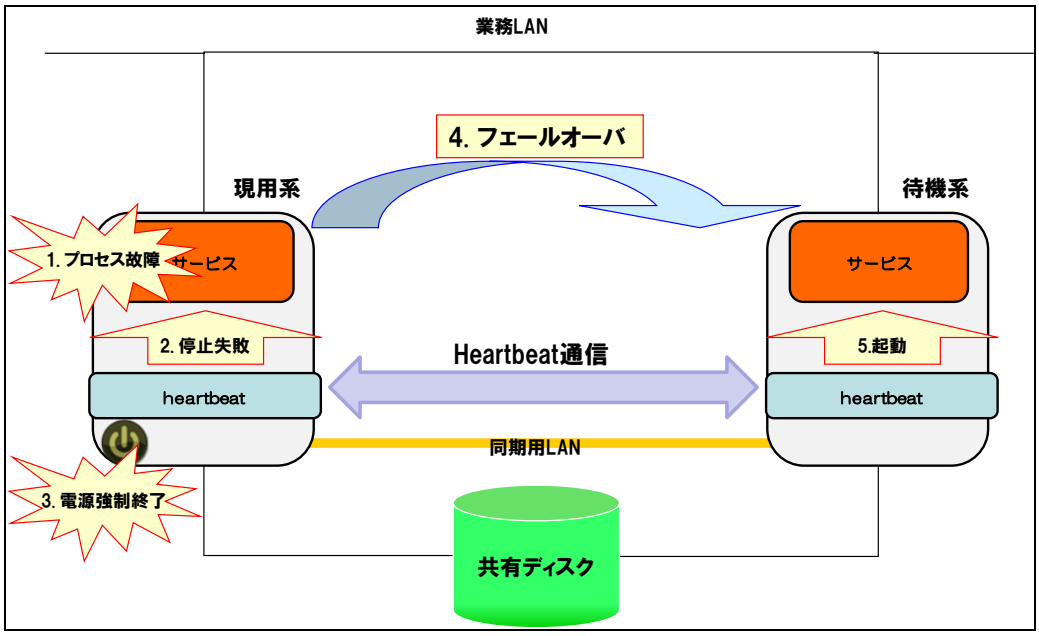


図 10-10 STONITH 動作イメージ

フェールオーバ時に現用系でのリソース停止に失敗した場合、リソースが両系で起動することを防止するためにフェールオーバを中断する。このため、リソース停止失敗時にはサービスが停止してしまう。そこでサービス停止を回避するために、リソース停止失敗時にサーバを強制的に停止(STONITH 機能の使用)することでフェールオーバを継続させる。

10.5.1.3 永続データの可用性に関わる設計

HA クラスタ構成の内、永続データの可用性に関わる箇所の設計について以下に示す。

DRBD 設計

表 10-3 に HA クラスタ構成での DRBD 設計を示す。

表 10-3 DRBD 設計(HA クラスタ構成)

No.	項目	詳細
1	同期方式	同期モードとして現用系をプライマリ・待機系をセカンダリとした単一プライマリモードを採用する。 また信頼性を確保するため、同期プロトコルとして Protocol C を採用する。
2	起動方法	10.5.1.1 にて示したように、DRBD は Heartbeat の管理化に置かれるため、起動は Heartbeat 経由で行われる。
2	同期対象	Hadoop の制御ファイルを格納したディスクパーティションを同期対象とする。

DRBD の同期方式としては同期モードと同期プロトコルを決定する必要がある。同期モードはデータ同期の方向を、同期プロトコルはデータ同期のタイミングを決定する。

同期モードは、現用系をプライマリ・待機系をセカンダリとした単一プライマリモードを採用する。単一プライマリモードでは、プライマリのデータをセカンダリに転送し同期する。セカンダリ側はディスクにアクセスすることができないため、Heartbeat による切り替え時には、待機系のセカンダリをプライマリに昇格する。

一方、同期プロトコルは以下の 3 つの内、もっとも信頼性の高い ProtocolC を採用する。ProtocolC はネットワーク帯域を必要とするというデメリットがあるが、同期用のネットワークにはギガビット以上のイーサネットを使用するため、必要な帯域は確保されている。

- ProtocolA:ローカルディスクへの書き込み完了後、TCP の送信バッファへ命令が到達した時点で同期完了
- ProtocolB: ローカルディスクへの書き込み完了後、レプリケーション用パケットの到達確認時点で同期完了
- ProtocolC:ローカルディスクへの書き込み完了後、他系で書き込み完了した時点で同期完了

10.5.1.4 ネットワークの可用性に関わる設計

HA クラスタ構成の実現の内、ネットワークの可用性に関わる箇所の設計について以下に示す。

業務 LAN 設計

Heartbeat による業務 LAN の監視および通信路の二重化を行うことで、ネットワ

ークの可用性を担保する。現用系の業務 LAN がダウンした場合はフェールオーバが発生し、仮想 IP が待機系に関連付けられることでサービスが継続する。

同期用 LAN 設計

ハートビート通信が切断するとスプリットブレイン状態となるため、専用 LAN を 2 本確保することによる通信路の二重化が必要である。ただし、bonding による冗長化は行わない。これは Heartbeat により多重化を行った方が信頼性が高いためである。なお Heartbeat ではハートビート通信量が多いため、シリアルケーブルは採用しない。

10.5.1.5 その他の設計

HA クラスタ構成の実現の内、上述以外の箇所の設計について以下に示す。

フェールバック方針

故障したサーバが復旧した場合に自動的にリソースをフェールバックさせるかを検討する。フェールバック時には切り替えによる一時的なサービス停止が発生するため、自動フェールバックは行わず、メンテナンス時間帯に手動でフェールバックを行うこととする。

10.5.1.6 故障時の動作

故障パターンとして表 10-4 の故障を想定し、それぞれの場合での動作例について述べる。各故障と構成とのマッピングを図 10-11 に示す。

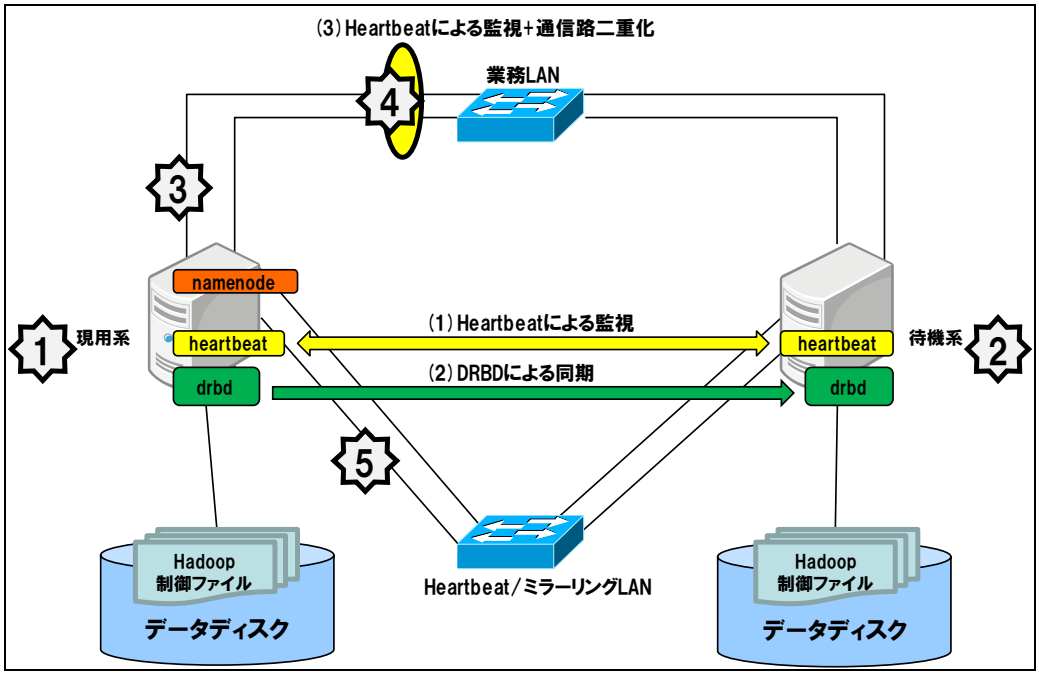


図 10-11 故障パターン(HA クラスタ構成)

表 10-4 故障内容と故障時の動作(HA クラスタ構成)

No.	故障内容	故障時の動作
1	現用系サーバ電源断	待機系にフェールオーバする
2	待機系サーバ電源断	サービスが継続される
3	業務用 LAN 片系故障	サービスが継続される
4	業務用 LAN 両系故障	待機系にフェールオーバする
5	Heartbeat LAN 片系故障	サービスが継続される。

10.5.1.7 検証実験

本章にて示した HA クラスタ構成により、求める可用性が担保されていることを確認するために、検証実験を行った結果を示す。

冗長化検証

Hadoop 付属の TeraSort アプリケーション実行中に、サーバに故障を発生させ処理が継続するか検証を行った。検証項目、結果を表 10-5 に示す。

表 10-5 冗長化検証項目(HA クラスタ構成)

No.	大分類	小分類	検証項目	検証結果
1	現用系サーバ故障	サーバ電源断	待機系サーバにフェールオーバする	○
2	待機系サーバ故障	サーバ電源断	特に変化なく、実行中の処理が継続する	○
3	現用系ネットワーク故障	業務 LAN 片系故障	特に変化なく、実行中の処理が継続する	○
4		業務 LAN 両系故障	待機系サーバにフェールオーバする	○
5		Heartbeat LAN 片系故障	特に変化なく、実行中の処理が継続する	○

10.5.2 ソフトウェア FT 構成の実現と検証

HA クラスタ構成に Kemari を加えたソフトウェア FT 構成の実現および検証実験の結果について、以下に示す。

10.5.2.1 保護対象のリソース

ソフトウェア FT 構成において 10.3 にて示した可用性を担保するために、保護しなくてはならないリソースを表 10-6 に示す。

表 10-6 リソース一覧(ソフトウェア FT 構成)

No.	担保対象	保護リソース	用いられる技術
1	プロセス	・Kemari プロセス	・Heartbeat ・Kemari
2	永続データ	・仮想マシンの起動イメージを格納するデータ領域	・DRBD
3	ネットワーク	・ネットワークインターフェース	・Heartbeat ・Bonding

10.5.2.2 プロセスの可用性に関わる設計

ソフトウェア FT 構成の実現の内、プロセスの可用性に関わる箇所について以下に示す。

Kemari プロセスのリソースエージェント化

Kemari によるソフトウェア FT 構成では、公開されている Kemari 用のリソースエージェント（以下 Kemari RA)を使用して、Heartbeat との連携を行う。Kemari RA は、以下のフローにより仮想マシンの起動・状態の同期を行う。

(1) Kemari RA 経由で、現用系の仮想マシンが起動する。
(2) Heartbeat RA 経由で、Kemari プロセスが生成される。
(3) Kemari プロセス経由で、待機系の仮想マシンが起動する。待機系の仮想マシンは常時 pause 状態で現用系からの同期通信を受け付ける。
(4) Kemari プロセスが、仮想マシンの実行状態の同期を開始する。
(5) 初期同期完了後、NameNode/JobTracker プロセスがアクセス可能となる。

Kemari によるソフトウェア FT 構成の起動時動作概要を図 10-12 に示す。

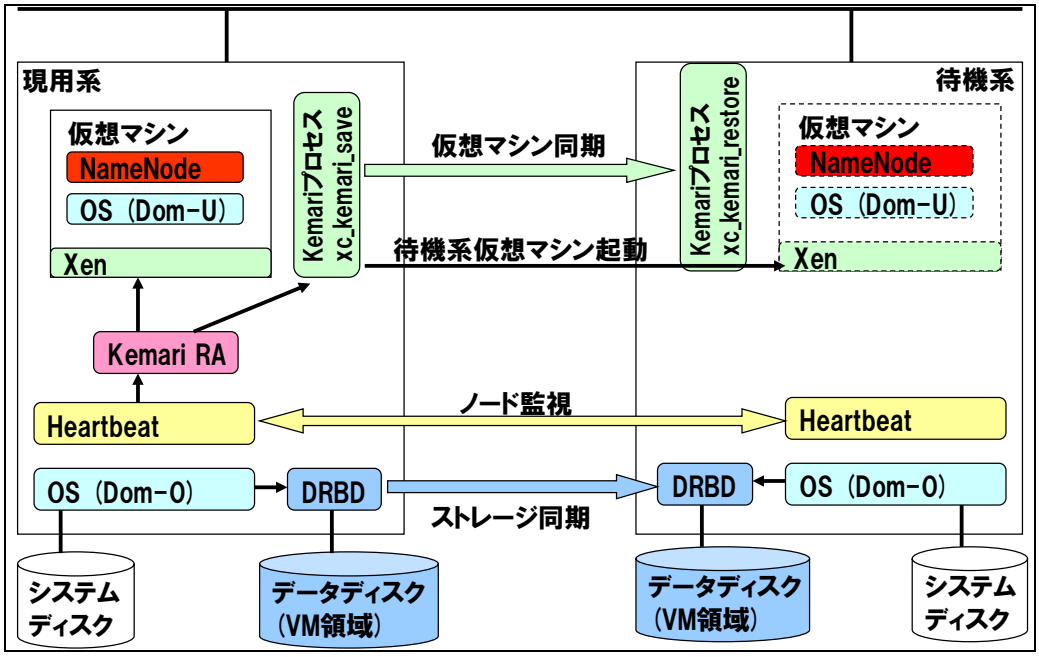


図 10-12 ソフトウェア FT 構成の起動動作概要

現用系に故障が起こった場合、待機系の仮想マシンに切り替える事でサービスが継続される。故障時の切り替え処理は、以下のフローで行われる。

(1) Heartbeat の監視機能により、現用系の故障を検知する。
(2) 待機系で、Kemari RA が起動する。
(3) Gratious ARP パケット送信により MAC アドレスの更新を行う。
(4) Kemari プロセス経由で、待機系の仮想マシンの停止状態が解除される。
(5) Gratious ARP パケット送信により MAC アドレスの更新を行う。
(6) 切り替えが完了し、仮想マシンにアクセス可能となる。

Kemari によるソフトウェア FT 構成の故障時切り替え動作の概要を図 10-13 に示す。

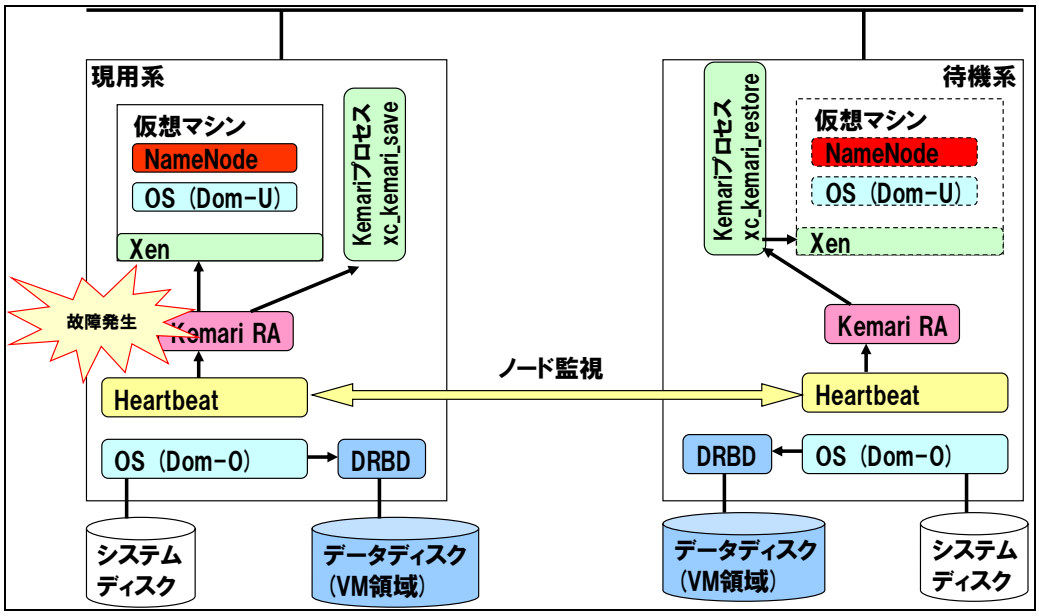


図 10-13 ソフトウェア FT 構成の故障時切り替え動作概要

仮想マシン切り替え時には、同一 IP の仮想マシンが別の物理サーバで動作する。同一セグメントの機器上の ARP キャッシュを更新するために、Heartbeat 経由で Gratious ARP を送信し、切り替え後も仮想マシンとのネットワーク接続を維持する。

10.5.2.3 永続データの可用性に関わる設計

ソフトウェア FT 構成の内、永続データの可用性に関わる箇所について以下に示す。

DRBD 設計

Kemari は、Xen のマイグレーション機能を用いて実装されている。Xen のマイグレーション機能を実行するためには、マイグレーション元(現用系)/マイグレーション先(待機系)の両方からのディスクアクセスを許可する必要があり、そのための手段と

しては共有ディスクが一般的である。しかし 10.2 にて示した前提より、本検証実験では DRBD を用いて同等の機能を実現することとする。そのための DRBD 設計を表 10-7 に示す。

表 10-7 DRBD 設計(ソフトウェア FT 構成)

No.	項目	詳細
1	同期方式	現用系・待機系の両方から常時ストレージにアクセス可能である必要があるため、同期モードとしてデュアルプライマリモードを採用する。 また信頼性を確保するため、同期プロトコルとして Protocol C を採用する。
2	起動方法	Heartbeat ではアクティブ/スタンバイ、マスター/スレーブ構成のリソースしか扱えない。そのため、デュアルプライマリモードで起動する DRBD を管理下に置く事はできない。この制約により、DRBD は Heartbeat 経由で起動せずに OS 起動時に常駐サービスとして起動する
3	同期対象	仮想マシンのイメージを格納したディスクパーティションを同期対象とする。

10.5.2.4 ネットワークの可用性に関わる設計

ソフトウェア FT 構成の内、ネットワークの可用性に関わる箇所について以下に示す。

業務 LAN 設計

Heartbeat による業務 LAN の監視および通信路の二重化を行うことで、ネットワークの可用性を担保する。現用系の業務 LAN がダウンした場合はフェールオーバが発生し、待機系で仮想マシンが起動することによりサービスが継続する。

同期用 LAN 設計

Kemari によるソフトウェア FT 構成では、現用系と待機系のサーバ間で以下の 3 つの通信が必要である。

- Heartbeat:ノード監視用の通信
- DRBD:ディスク同期用の通信
- Kemari:仮想マシン実行状態同期用の通信

Hadoop の NameNode/JobTracker のようなディスク I/O、ネットワーク I/O が頻繁に発生するプロセスを稼動させる場合、ストレージ同期用の通信及び仮想マシン実行状態同期用の通信が大量に発生する。同期用 LAN の通信量の一例を図 10-14 に示

す。

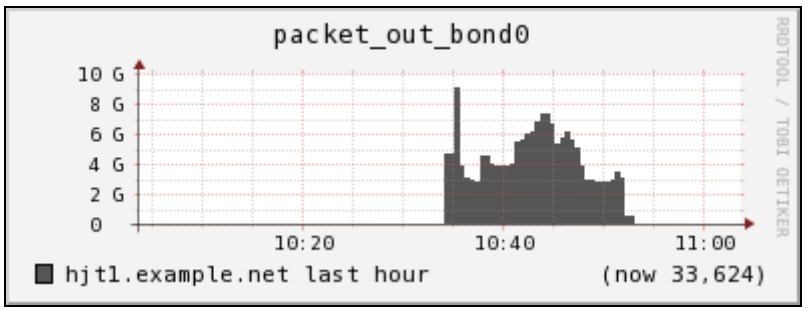


図 10-14 同期用通信路の通信量

帯域を確保し、かつ可用性を担保するためには、同期用 LAN について以下の要件を満たす必要がある。

- 同期用 LAN として専用 LAN を確保する。これは、他の通信により同期通信の使用する帯域が圧迫されることを防ぐためである。
- 同期用 LAN は 10GNIC で構成する。これは、常時ギガビット単位の通信が流れるため、1GNIC では同期用通信路がボトルネックとなるうるためである。
- 同期用 LAN は Bonding により二重化する。これは、同期用通信が途切れることによるサービスの停止を防ぐためである。

3 つの通信それぞれに専用 LAN を確保する設計も考えられるが、現状の Kemari RA は用途の違う LAN の故障に対応していないため、3 つの用途を共存させる設計とする。

10.5.2.5 その他の設計

HA クラスタ構成の内、上述以外の箇所の設計について以下に示す。

Xen リソース設計

現在の Kemari は、Xen 及び Xen カーネルへのパッチによる実装である。Kemari によるソフトウェア FT 構成での Xen リソース設計について、以下に示す。

(1) 必須要件

Kemari によるソフトウェア FT 構成は、仮想マシンについて以下の要件を満たす必要がある。

- 仮想マシンの仮想化方式として完全仮想化を採用している。そのため、物理マシンの CPU は仮想化支援機能(Intel-VT, AMD-V)に対応している必要がある。
- 仮想マシンに PV ドライバがインストールされている必要がある。なお、PV

ドライバとは、完全仮想化により生成された仮想マシンの I/O 性能を向上させることを目的とした特殊なドライバである。

(2) リソース割り当て検討

本報告書の検証環境では、以下の方針に基づき仮想マシンへのリソース割り当てを行っている。

- 仮想マシン上で動作させる NameNode/JobTracker は、CPU リソースを多く使用するプロセスではなく、メモリやネットワーク I/O がボトルネックになるケースが多い。また Kemari では仮想マシンに複数 CPU を割り当てても有効に活用できないため、仮想マシンの CPU 数を 1 とする。
- NameNode は HDFS のメタデータをメモリ上で管理するため、HDFS で扱うデータ量を考慮して、仮想マシンに必要なメモリ量を見積る。一方 JobTracker では、MapReduce タスクの管理情報をメモリ上に保存するため、タスクの分割数などを考慮する必要がある。今回使用するアプリケーションでは、NameNode/JobTracker の必要メモリ量は 2.5GB 程度であると見積っている。データの増加などを考慮して、仮想マシンに 8G のメモリを割り当てる。

10.5.2.6 故障時の動作

故障パターンとして表 10-8 の故障を想定し、それぞれの場合での動作例について述べる。各故障と構成とのマッピングを図 10-15 に示す。

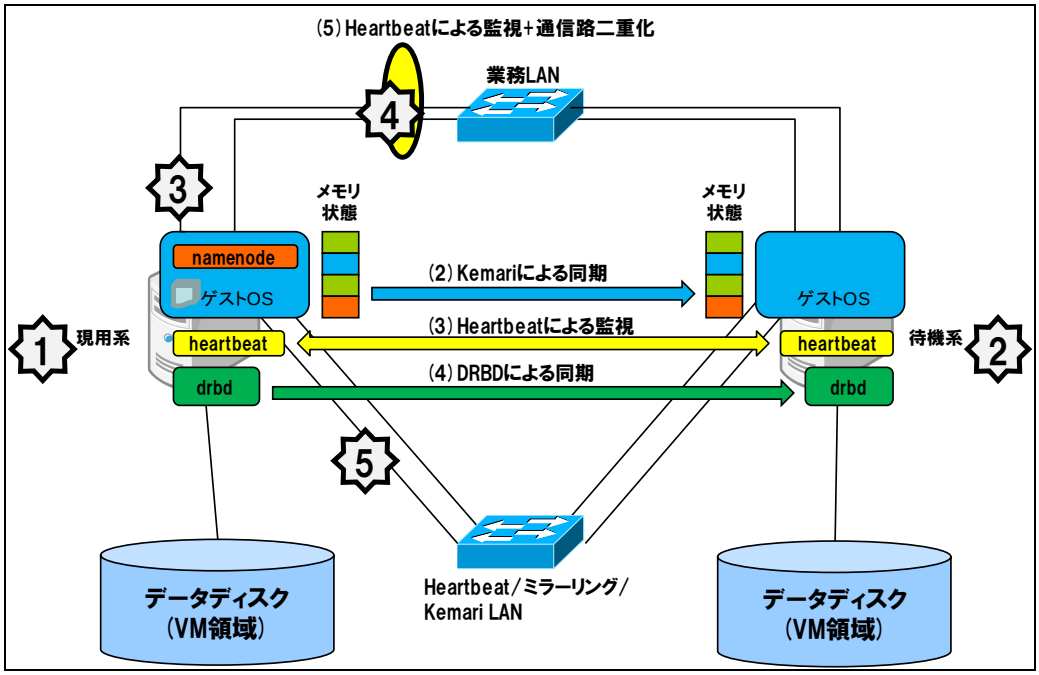


図 10-15 故障パターン(ソフトウェア FT 構成)

表 10-8 故障内容と故障時の動作(ソフトウェア FT 構成)

No.	故障内容	故障時の動作
1	現用系サーバ電源断	待機系にフェールオーバする
2	待機系サーバ電源断	サービスが継続される
3	業務用 LAN 片系故障	サービスが継続される
4	業務用 LAN 両系故障	待機系にフェールオーバする
5	Heartbeat LAN 片系故障	サービスが継続される。

10.5.2.7 Kemari 動作検証

本章にて示したソフトウェア FT 構成により、求める可用性が担保されていることを確認するために、検証実験を行った結果を示す。

冗長化検証

Hadoop 付属の TeraSort アプリケーション実行中に、サーバに故障を発生させ処理が継続するか検証を行った。検証項目、結果を表 10-9 に示す。

表 10-9 冗長化検証項目(ソフトウェア FT 構成)

No.	大分類	小分類	検証項目	検証結果
1	現用系サーバ故障	サーバ電源断	待機系に切り替わり、実行中の処理が継続する	○
2	待機系サーバ故障	サーバ電源断	特に変化なく、実行中の処理が継続する	○
3	現用系ネットワーク故障	業務用 LAN 片系故障	特に変化なく、実行中の処理が継続する	○
4		業務用 LAN 両系故障	待機系に切り変わり、実行中の処理が継続する	○
5		Heartbeat LAN 片系故障	特に変化なく、実行中の処理が継続する	○

10.6 結果の考察と今後の課題

HA クラスタ構成と Kemari によるソフトウェア FT 構成での冗長化の考察・比較を以下に示す。

10.6.1 HA クラスタ構成に関する考察

Heartbeat により NameNode や JobTracker のハードウェア/ソフトウェア故障時に自動的に待機系に切り替わり、プロセスを再開することができた。ただし、一時的なプロセス停止が発生すると、JobTracker の場合は実行中のジョブは中断する。

また NameNode は起動後に HDFS のメタデータと DataNode の実データの整合性を確認し、レプリケーションの既定値を満たすまで Safemode となる。Safemode 中は NameNode のログが大量に出力されるため、サーバのディスク容量を圧迫する可能性がある。

10.6.2 ソフトウェア FT 構成に関する考察

Kemari は仮想マシンの実行状態を同期するため、ハードウェア故障時においても実行中のジョブが継続できる事がわかった。Hadoop で実行するジョブは大量データの処理など実行時間が長いことが想定されるため、ハードウェア故障が発生してもジ

ョブを継続できるメリットは大きい。短いジョブであっても規定時間内に実行完了しなくてはならないなどの制約がある場合は、ダウンタイム短縮のためにソフトウェア FT 構成をとるという選択肢が考えられる。

しかし、表 10-10 に示すように Kemari ではネットワークのスループット低下が大きい。そのため、実アプリケーションでどの程度帯域が必要か影響を確認する必要がある。

オーバヘッド検証

Kemari のオーバヘッド測定についての検証項目及び結果を表 10-10 に示す。オーバヘッド測定に使用するベンチマークソフトと使用目的は以下の通りである。

- Netperf: 業務用 LAN のネットワークスループットへの影響を測定するために使用する
- NNBench: NameNode の処理性能への影響を測定するために使用する
- TeraSort: Hadoop 系全体の処理性能への影響を測定するために使用する

上記のベンチマークソフトについて、Kemari による仮想マシン実行状態の同期が行われている場合と行われていない場合を比較する。

表 10-10 Kemari のオーバヘッド測定

No.	大分類	小分類	検証項目	Kemari 無	Kemari 有
1	Kemari オーバヘッド測定	業務用 LAN の性能	Netperf によるスループット測定	750(Mbps)	20(Mbps)
2		NameNode の性能	NNBench の処理時間 [Map 数:4 ブロックサイズ:1[byte] ファイルサイズ:1[byte] ファイル生成数:5000]	203(s)	2393(s)
3		Hadoop 系全体の性能	Terasort の処理時間 [データサイズ:10G]	17.5 分	16 分

Kemari によるソフトウェア FT 構成では、ネットワーク I/O が発生したタイミングで実行状態の同期を行うため、業務用 LAN のスループットは通常時に比べて低下する。表 10-10 より、スループットが約 1/40 に低下していることが分かる。今回の検証ではボトルネックにならなかったが、スケールアウトしていく中でネットワークがボトルネックになる事が予想される。

小サイズのファイルを大量に生成する処理を行った場合、NNBench の処理時間が約 10 倍になっており、Kemari のオーバヘッドが顕著である。一方、ブロックサイズ 64MB(Hadoop 基盤のデフォルト値)で実行される Terasort は、処理時間に有意な差が見られない。

Hadoop 基盤で取り扱うファイルは数百 MB オーダである事が多く、ブロックサイズをデフォルト値から小さく設定する必要が生じることは少ない。そのため、Hadoop 基盤の管理ノードの冗長化構成としては問題ないと考えられる。

10.6.3 今後の課題

ソフトウェア FT 方式は、ハードウェア故障への対策であり OS やアプリケーションの故障には対応できない。たとえば、現用系の仮想マシン上で動作している Hadoop プロセスが故障した場合、待機系の仮想マシンは Hadoop プロセスがダウンした状態を再現するため、フェールオーバはなんら問題を解決しない。こういった故障を正常状態に戻すためには、リソースの再起動が必要となる。そのため、ソフトウェア FT 方式に仮想マシン自体や Hadoop プロセスの故障を検知して再起動を実施する対策を追加することが課題である。

また、ハードウェア FT 方式では CPU・メモリなどのハードウェア情報を監視することで、故障の予兆を検知して待機系にフェールオーバするという仕組みを備えているが、今回提示した Kemari によるソフトウェア FT 方式には同等の機能が存在しない。更なる可用性向上を求めて、上記の予兆検知機能を追加実装することが課題である。

11 Hadoop 基盤における運用手法検討

本章では、Hadoop 基盤における運用項目を一覧化し、Hadoop 基盤を運用する上で効率化するべき項目を抽出する。そして、効率化のための実現方法を検討し、その方式を報告する。

なお、効率化のための実現方式のうち、「Hadoop スレーブサーバの自動構築」に関しては 12 章、「系を全体的に把握する手法」に関しては 13 章に詳細な報告を行う。併せて参照されたい。

11.1 Hadoop 基盤における運用上の特徴と課題

本節では Hadoop 基盤における運用の範囲を定義するとともに、一般的なシステムにおける運用と比較して、Hadoop 基盤において特徴的な点を明らかにする。

11.1.1 運用からみた Hadoop 基盤の特徴

本節では、クラウド型分散処理基盤の運用から見た際の特徴を明らかにするとともに、Hadoop 基盤を運用から見た際の特徴を明らかにする。

クラウド型分散処理基盤の特徴

クラウド型分散処理基盤における運用の特徴的な項目として、基盤を構成する要素が非常に多いことが挙げられる。概してクラウド型分散処理基盤を構成するサーバ及びネットワーク機器は数十～数千台で構成される。また、分散処理基盤の規模に比例して構成要素が多くなる。

それに加え、クラウド型分散処理基盤においては、多数の機器の故障・回復及び増設が定常的に行われることを前提としなくてはならない。そのため、種別・機種が単一で構成するという前提を置くことは適切でなく、個々の機器の種別や機器は混在しうるという特徴を持つ。図 11-1 にその概念図を示す。

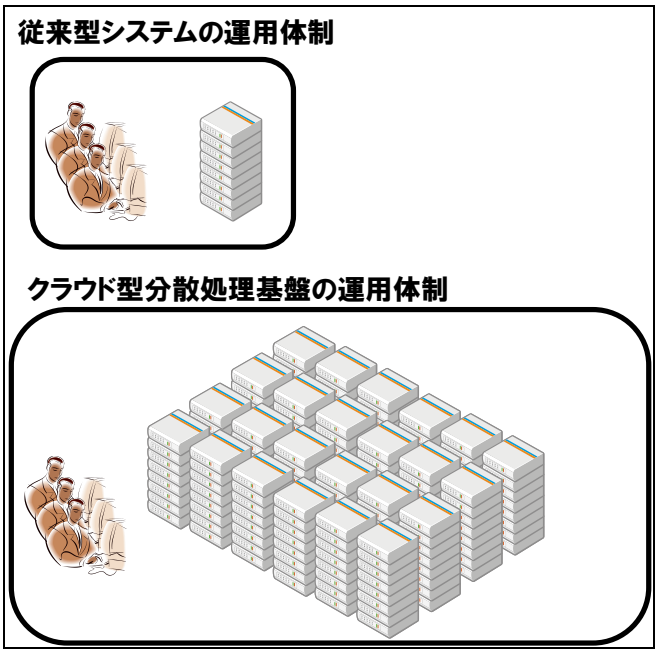


図 11-1 Hadoop 基盤は構成要素が多い

Hadoop 基盤の特徴

Hadoop 基盤は、大量の Hadoop スレーブサーバとその結果を取りまとめる少数の Hadoop マスターサーバ（NameNode, JobTracker）で構成されている。Hadoop マスターサーバは単一故障点となりうるという意味で、厳密な監視や即時の故障検知、回復を検討する必要があるが、Hadoop スレーブサーバはそうではないという特徴を持つ。Hadoop スレーブサーバは台数が非常に多く、故障時においては、縮退動作となり、システム全体の動作がとまることはない。また、数台の故障に関しては、縮退時での性能低下も軽微であると考えられる。したがってこれらのサーバに関しては、運用項目として重点を置かなくてよい項目も挙げられる。図 11-2 にその概念図を示す。

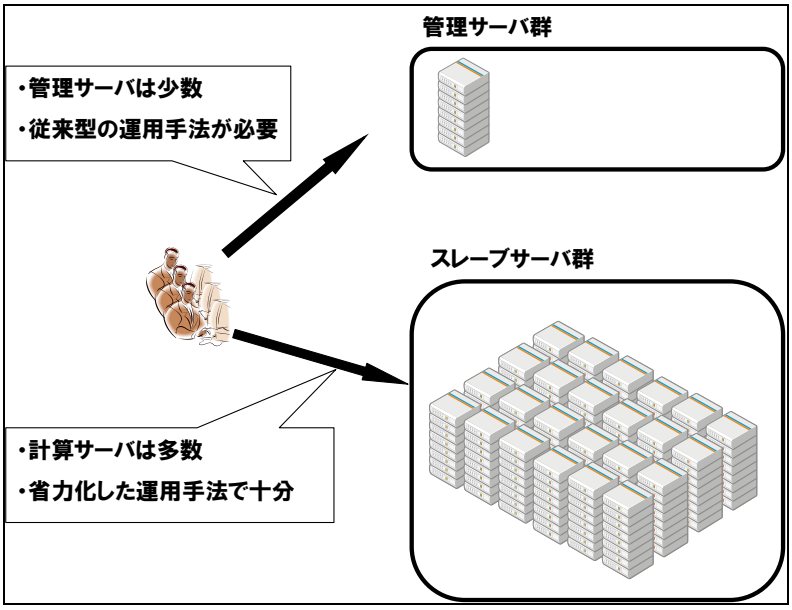


図 11-2 Hadoop 基盤における運用レベル

上記の検討結果より、Hadoop 基盤の運用の特徴として、総論として表 11-1 のようにまとめることができる。

表 11-1 Hadoop 基盤の運用上の特徴

No.	Hadoop 基盤の特徴	概要
1	Hadoop を構成する構成要素が多い	大量・多種のサーバ運用が必要。また、故障及び復旧に対する考慮が必要である。
2	簡略化できる運用項目が存在する	大量に存在する Hadoop スレーブサーバが故障しても動作を継続することが可能な上、縮退動作も影響が少ない。

11.1.2 Hadoop 基盤における効率的な運用手法検討における基本方針

Hadoop 基盤運用では、Hadoop スレーブサーバが大量に存在する。そのため、Hadoop スレーブサーバに対して、運用者が効率よく運用することが必要である。

そこで、本章では、Hadoop 基盤の効率的な運用を以下の三点に分割して検討する。一点目は、Hadoop 基盤における運用項目を列挙してその中で運用効率化が必要な項目を抽出することである。二点目は、効率化が必要な運用を実現するための実現方式を検討することである。三点目は、実現方式を利用して運用項目を実現することである。

11.3 において、運用項目を列挙し、効率化が必要な項目を列挙する。11.4 でその方式検討を行い、11.5 で各運用項目の実現方式を報告する。

11.1.3 運用の検討範囲

本章では、運用を表 11-2 に挙げるように大別し、各項目に関して運用方法を検討する。

表 11-2 本章における運用の定義

No.	項目	運用の概要
1	初期構築	ハードウェアに OS やソフトウェアをインストールして分散計算のために使えるようにセットアップする。
2	監視・故障検知	クラスタを構成している個々のハードウェア・ソフトウェアの死活監視を実施するとともに、クラスタ全体のサービス継続性や性能などの各種指標値を定常的に監視する。
3	維持管理・メンテナンス	ハードウェア・ソフトウェアの構成情報を管理したり、物理的な配置や故障・回復履歴を管理したりする。また、構築済みのクラスタで使用された設定ファイルを維持管理するなど、構成を管理する。また、Hadoop 基盤のメンテナンスを実施する。メンテナンスには、サービスの閉塞・開放や、システムバックアップや、データのバックアップ作業も含まれる。
4	回復・増設	クラスタを構成する各要素に関して故障検知や、その他アラートが挙がった際にどのような手順で復旧させるかを記載する。ハードウェアの交換や、回復後のハードウェアの再構築とクラスタへの組み込みを実施する。増設では、計算性能をあげる為にラック単位でハードウェアを増設することである。

初期構築では、Hadoop 基盤の初期構築を実施する。ラッキング・ケーブリングされているサーバ、ネットワーク機器に対して、クラウド型分散処理を実施するために必要な基盤構築を行うことを目的とする。初期構築では大量のサーバ及びスイッチの構築が必要となる。構築の効率化については、12 章でより詳細な検討を実施する。

監視・故障検知では、Hadoop 分散処理の状況を監視・可視化する項目を想定する。Hadoop 基盤を構成する各要素の故障検知（HDD の故障や NIC ポート故障）やアプリケーションレイヤにおける異常（プロセス障害や例外ログ）を検知する。さらに Hadoop 基盤全体としてのサービス監視や、系をサマライズして理解できるような手法による Hadoop 基盤全体の評価をする。これらを実現し、定常的に行う運用を監視・故障検知と定義する。

維持管理・メンテナンスでは、Hadoop 基盤を構成する各要素の維持管理を実施す

る。Hadoop 基盤は構成要素が多数存在するため、故障及び回復によって頻繁に基盤を構成する要素が変更される。そのため、Hadoop 基盤の構成要素を時系列に維持・管理する必要がある。また、大量のサーバに各種構成（アプリケーションや設定ファイル）を配布・更新・状態管理するライブラリ管理が必要となる。また、サービスの閉塞・開放や、システムバックアップや、データのバックアップ作業も含まれる。

回復では、Hadoop 基盤において動作不良に陥った構成要素に対して、必要に応じて修理・交換を行うことで、再び Hadoop 基盤の構成要素へ復帰させることと定義する。増設は、Hadoop 基盤の構成要素を大きく追加する運用である。ラック単位（十～百台程度）の大幅な Hadoop スレーブサーバの追加などがこれに該当する。ラック単位のマシンを Hadoop スレーブサーバに追加する運用項目として、増設を定義する。

11.2 前提条件

本節では運用のための前提条件として、想定する運用目標を設定する。

11.2.1 想定する運用

Hadoop 基盤の運用手法検討のために、表 11-3 のような運用を想定する。

表 11-3 想定するシステム保守

No.	項目	想定する運用の概要
1	利用時間	原則として 24H/365D. 不定期なメンテナンス時間帯がある
2	保守時間	9:00-17:00 の保守対応時間。
3	ベンダ保守	翌日以降対応。

運用担当者は、定例的な運用項目（典型的には、日々の監視・故障検知）を実行する。回復時には必要に応じて物理的にデータセンタにおける作業を実施する。

11.2.2 想定する Hadoop 基盤の構成

本章では図 11-3 のような Hadoop 基盤を想定する。

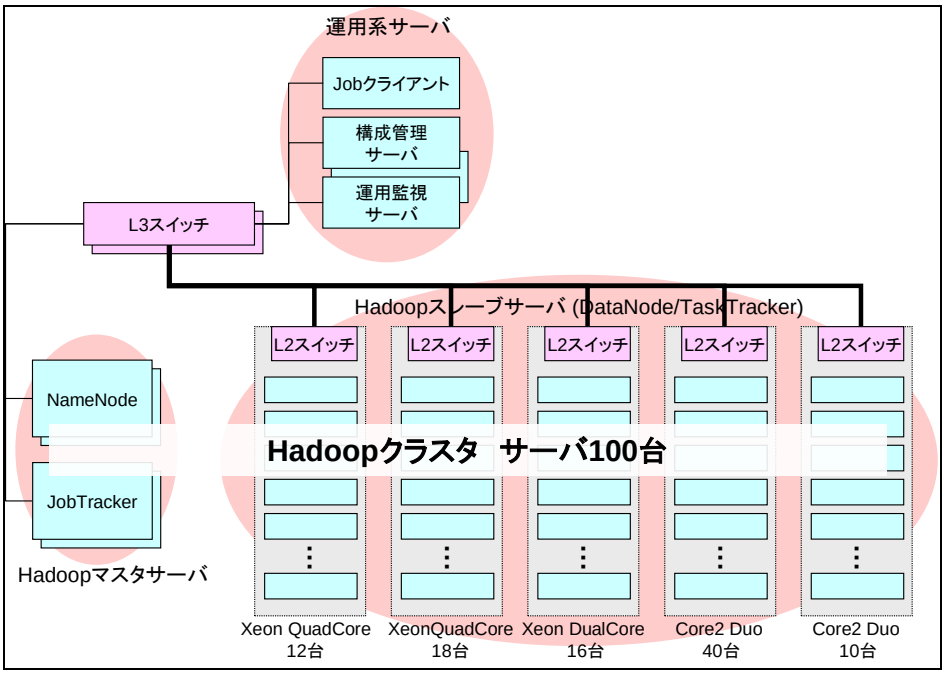


図 11-3 前提とする Hadoop 基盤の構成図

11.3 Hadoop 基盤での運用項目の抽出

本節では、Hadoop 基盤の各運用項目に対して、行うべき詳細な運用項目を抽出するとともに、効率化対象を選定する。

11.3.1 効率化対象の抽出の基本方針

Hadoop 基盤の効率的な運用のためには、大量に存在する Hadoop スレーブサーバを対象とする様な運用に対して効率的な運用を検討する必要がある。本節では、Hadoop スレーブサーバに関して行うべき運用項目を効率化対象として選定し、次節以降にその効率化手法を検討する。

11.3.2 初期構築における運用項目の抽出と効率化対象の選定

初期構築の目標を、ラッキング・ケーブリングされたサーバ群に対して、OS や分散計算に必要なソフトウェアや管理のためのソフトウェアがインストールされ、利用できること、また、可視化・監視の対象となるように構成することとする。

したがって、初期構築において行うべき運用項目は表 11-4 の通りである。

表 11-4 初期構築における運用項目一覧

No.	項目	概要	効率化対象
1	コア L3 スイッチの構築	Hadoop マスターサーバや運用 Hadoop マスターサーバ等が直接つながるスイッチの構築	
2	L2 スイッチの構築	Hadoop スレーブサーバが直接つながるスイッチの構築	
3	維持管理サーバの構築	名前解決サービスや構成管理サービスを提供するサーバの構築	
4	運用管理サーバの構築	運用監視・リソース監視サービスを提供するサーバの構築	
5	Hadoop マスターサーバの構築	NameNode, JobTracker が動作するサーバの構築	
6	Hadoop スレーブサーバの構築	DataNode, TackTracker が動作するサーバの構築	○

初期構築における運用項目は Hadoop 基盤を構成する構成要素であるスイッチ及びサーバの構築である。これらのうち Hadoop スレーブサーバに関しては、Hadoop の

規模が大きくなるにしたがって増加するが、その他の Hadoop マスターサーバや運用管理サーバ等は規模によらず数台である。したがって Hadoop スレーブサーバの構築は効率化対象として選定する。

11.3.3 監視・故障検知における運用項目の抽出と効率化対象の選定

本節では運用項目として監視・故障検知に焦点をあて、運用項目の明確化及び運用効率化のための課題を洗い出し、その課題解決手法を検討する。

監視・故障検知に関しては、ハードウェアの監視と、Hadoop 基盤の全体監視が必要である。ハードウェアの監視では、スイッチの故障監視や、サーバ故障監視など、Hadoop 基盤を構成している個別の構成要素の監視を定常的に実施する。Hadoop 基盤の全体監視では、Hadoop 基盤を構成している多数の構成要素が全体的としてサービスを提供できるかを定常的に監視する。

ハードウェアの監視

ハードウェアの監視は Hadoop 基盤を構成するスイッチ及びサーバ機器個別の監視である。表 11-5 にハードウェアの監視するべき対象を記載する。

表 11-5 監視・故障検知における運用項目の監視対象

No.	項目	概要	効率化対象
1	スイッチの監視	スイッチに何らかの異常がないかを監視する。	
2	Hadoop マスターサーバ・管理系サーバの監視	Hadoop 管理サーバや運用管理サーバ、維持管理サーバにハードウェア・ソフトウェア的な異常がないかを監視する。	
3	Hadoop スレーブサーバの監視	Hadoop スレーブサーバにハードウェア・ソフトウェア的な異常がないかを監視する。	○

監視に関しては、既存のシステムにおいては確立された手法があり、様々な製品が広く使用されている。商用製品や、Hinemos、Nagios といったオープンソースソフトウェアによる監視システムの実装が既に存在する。監視対象はスイッチなどのネットワーク機器、サーバなどのサーバ機器を対象として、SNMP 監視、SNMP-TRAP 監視、ログ監視、プロセス監視、死活監視、サービス監視など様々な監視手段も提供されている。本報告書においても以下に示すように各構成要素の監視を行うこととす

る。

一方、Hadoop 基盤に対して上記の監視・通知システムがそのまま適用できない。Hadoop 基盤は非常に大量の Hadoop スレーブサーバで構成される。したがって、監視サーバに対する負荷が過大となり、これまでのような監視が出来ない。

具体的な例をあげる。通常監視システムにおいて一台あたりのサーバに関して監視するべき項目が 20 項目程度存在し、監視項目を 60 秒の間隔で取得するという状況を考える。Hadoop 基盤の構成要素は、120 程度を前提としているために、監視システムは 1 秒あたり 40 もの監視を実行しなくてはならない。一般的にはクラウド型分散基盤においては、千台程度の規模のシステムであることも十分考えられる。これらの考察から、監視システム自体が十分にスケーラブルなものであることが必要であるが、多くの監視ツールにおいては、完全にスケーラブルな実装であるものはない。そのため次節以降で検討を行う。

Hadoop 基盤の全体監視

Hadoop 基盤は多数のサーバが協調しあい、ひとつのサービスを提供するシステムである。従って、系を全体としてとらえ、全体としての監視を実施する必要がある。本章では、系としての監視対象の選定として、サービスの停止につながる項目を選出することとした。表 11-6 に監視対象を記載する。

表 11-6 全体の監視対象の抽出

No.	監視対象	概要	効率化対象
1	性能監視	Hadoop クラスタを全体としてみた際に、各種リソース状況を可視化することで明らかにする	○
2	サービス監視	HDFS のサービス監視	
3		MapReduce のサービス監視	
4	リソース監視	DataNode の台数を閾値監視	
5		HDFS の残容量	
6		TaskTracker の台数を閾値監視する	

11.3.4 維持管理・メンテナンスにおける運用項目の抽出と効率化対象の選定

Hadoop 基盤は大量のサーバから構成される。特に Hadoop スレーブサーバは大量に存在するために、それらの構成情報を一覧化し、変更がされたことを保障することは重要である。

ハードウェアの構成管理に関しては、各ハードウェアの製造ベンダ・CPU・メモリなどのハードウェアスペックを一覧化するだけでは十分でない。大量に Hadoop スレーブサーバが存在し、故障があることを前提とした、ハードウェアの故障/回復履歴を一覧化することが必要となる。ソフトウェアの構成管理に関しも同様である。多数のハードウェアが存在し、それらの上に複数のミドルウェア及びそれを動作させるための設定ファイルが存在する。これらの情報を一覧化し、必要に応じて構成情報を配布・反映する必要がある。

上記より維持管理に必要な運用項目を表 11-7 に記載する。

表 11-7 維持管理における運用項目一覧

No.	運用項目	概要	効率化対象
1	ハードウェア構成情報の一覧化	構成要素の一覧化を行う。	
2	ハードウェア構成情報の変更	ハードウェアの故障/回復履歴を一覧化する。	
3	ソフトウェア構成情報の一覧化	ハードウェアごとに、インストールされたミドルウェアのバージョンや、アプリケーションのバージョンを一覧化し閲覧可能な状態にする。	
4	ソフトウェア構成情報の配布と反映確認	実機に設定が反映されていることを保障する。	○
5	設定変更	全ての対象サーバに対して適用するべき設定を適用する。	○
6	ミドルウェア配布	全ての対象サーバに対して配布するべきミドルウェアを適用する。	○

ソフトウェアの構成情報の配布と反映確認には、手作業では膨大な時間がかかるため、効率化対象として検討することとした。例を挙げる。Hadoop を動作させるためには、設定すべき設定ファイルは 10 ファイル程度存在するため、100 台の Hadoop 基盤から構成されているために、総じておよそ 1000 ファイルが意図通りに配布されていることを保障することが必要である。

また、Hadoop 基盤を継続的に利用するためには様々なメンテナンスが必要となる。

これらを表 11-8 に記載する。

表 11-8 メンテナンスにおける運用項目一覧

No.	運用項目	概要	効率化対象
1	系全体の閉塞	Hadoop サービス全体を閉塞する。	○
2	系全体の開放	Hadoop サービス全体を開放する。	○
3	サーバ単位の閉塞	対象サーバを閉塞し、Hadoop サービスを縮退させる。	
4	サーバ単位の開放	閉塞したサーバを開放し、Hadoop クラスタに組み込む。	
5	HDFS リバランス	HDFS のブロックに偏りのため、効率的に HDFS が利用できなくなる可能性があるために実施する。	
6	ログローテション	ログローテション設定を行うことで、ディスク増大を防ぐ。	
7	ログ収集	全てのサーバに対して、ログの収集を実行する。	
8	時刻同期	Hadoop クラスタ全体で、時刻同期を実施する。	
9	統計情報のバックアップ	統計情報を定期的にバックアップする。	
10	構成情報のバックアップ	配布した設定の履歴情報をバックアップする。	
11	HDFS メタ情報のバックアップ	HDFS のメタ情報をバックアップする。	

11.3.5 回復・増設における運用項目の抽出と効率化対象の選定

回復では、Hadoop 基盤において監視システムによって故障検知がなされた構成要素に対して、故障必要に応じて修理・交換を行うことで、再び Hadoop 基盤の構成要素へ復帰させることである。

なお、Hadoop 基盤は多数の構成要素で構成されており、機器の故障及びその修理に関しては、例外的な事態でなく、運用上常に対応が必要である。そのため、故障の判別及びその回復は本節に切り出して検討することとした。

また、増設では、Hadoop 基盤の構成要素を大きく追加する運用である。ラック単位（十～百台程度）の大幅な Hadoop スレーブサーバの追加などがこれに該当する。

ラック単位の多数のサーバを Hadoop スレーブサーバとして構築する運用項目を増設と定義する。回復における運用項目を表 11-9 に記載する。

表 11-9 回復・増設における運用項目一覧

No.	故障対象	運用項目	概要	効率化対象
1	コア L3 スイッチ	機器故障からの回復	スイッチ全体が故障する	
2		ポート故障からの回復	ポートが故障する	
3	ラック内 L2 スイッチ	機器故障からの回復	スイッチ全体が故障する	
4		ポート故障からの回復	ポートが故障する	
5	Hadoop マスターサーバ	ソフトウェア故障からの回復	各種プロセスや、OS が故障する	
6		機器故障からの回復	電源や NIC 等が故障する	
7	Hadoop スレーブサーバ	ソフトウェア故障からの回復	各種プロセスや、OS が故障する	○
8		機器故障からの回復	電源や NIC 等が故障する	○
9	構成管理サーバ・運用管理サーバ	ソフトウェア故障からの回復	各種プロセスや、OS が故障する	
10		機器故障からの回復	電源や NIC 等が故障する	

故障機器からの回復や、増設においては、初期構築と同じく、自動構築等の技術を使うことで、その運用項目を共通化することを検討する。そのため、表 11-10 のような運用項目を検討する。

表 11-10 自動構築と連携させるための回復・増設における運用項目一覧

No.	運用項目	概要	効率化対象
1	自動構築のための設定	自動構築のための前準備を行う	
2	監視・可視化設定	監視サーバ・可視化サーバ側に設定を行う	

11.4　効率的な運用方式検討

本節では、前節で効率化が必要であるとした各運用項目を実現するための、効率化手法を検討し、その実現方式を記載する。

11.4.1 効率的な運用方式の検討

前節において効率化対象とした 10 項目の運用項目に対して、効率化方式を検討する。表 11-11 に効率化が必要な運用項目を記載する。

表 11-11 Hadoop 基盤における効率化が必要な運用項目

No.	効率化対象の運用項目
1	Hadoop スレーブサーバの構築
2	Hadoop スレーブサーバの監視
3	性能監視
4	ソフトウェア構成情報の配布と反映確認
5	設定変更
6	ミドルウェア配布
7	系全体の閉塞
8	系全体の開放
9	Hadoop スレーブサーバのソフトウェア故障からの回復
10	Hadoop スレーブサーバの機器故障からの回復

効率化手法を検討するにあたり、可能な限り上記運用項目を広くカバーできるように選定を行い、表 11-12 の手法を選定した。次の小節ごとに詳細な方式検討を行う。

表 11-12 効率的な運用方式

No.	検討する効率化手法	対応する運用項目 (表 11-11 の No)
1	Hadoop スレーブサーバの自動構築方式	1,9,10
2	Hadoop クラスタの可視化方式	3
3	Hadoop スレーブサーバの監視方式	2
4	Hadoop クラスタにおける構成管理とデプロイ方式	1,4,5,6
5	Hadoop スレーブサーバに対する各種コマンド実行方式	7,8,

11.4.2 Hadoop スレーブサーバの自動構築方式

Hadoop スレーブサーバを自動構築することで、運用性の効率化を図る。作業者の作業時間短縮および、作業ミスや漏れを防ぐためにも自動化は有効である。本報告書では、Hadoop スレーブサーバの自動構築を、RedHat Enterprise Linux のネットワークツールである Kickstart とオープンソースソフトウェアの構成管理ツールである Puppet を適切に組み合わせることで実現する。本方式に関しては、12 章に詳細を記載しているのでそちらを参照されたい。なお、Puppet に関しては、11.4.5 に記載する。

11.4.3 Hadoop クラスタの可視化方式

Hadoop 基盤は少数の Hadoop マスターサーバ及び大量の Hadoop スレーブサーバから構成されている。これらのサーバを個々に情報を取得するだけでは、Hadoop 基盤の処理全体概要を捉えることができない。全体を捉えるためには、各サーバのリソース情報等を集計して表示する手法が必要となる。本報告書では、オープンソースソフトウェアである Ganglia を採用することで、Hadoop クラスタ全体の可視化を実現した。本方式については、12 章に詳細を記載するのでそちらを参照されたい。

11.4.4 Hadoop スレーブサーバの監視方式

Hadoop 基盤における Hadoop スレーブサーバは大量に存在し、Hadoop マスターサーバと同様、既存の監視システムで監視すると、監視サーバに多大な負荷がかかる。したがって Hadoop 基盤の特性を利用して、スケーラブルな監視方式が必要である。

Hadoop 基盤において、計算ノードは故障しても縮退するだけであり、サービスは継続して提供できる。したがって、Hadoop スレーブサーバに必要な監視項目を表 11-13 の項目で十分であるとした。

表 11-13 Hadoop スレーブサーバに関する監視項目の一覧

No.	監視対象	概要
1	死活監視	Hadoop のフレームワークを利用する
2	リソース監視	サーバの基本構成要素の使用率の閾値監視を実施する。具体的には、ハードディスクの各パーティションの残容量の閾値監視を実施する。
3	プロセス監視	Hadoop スレーブサーバにおいて必要なプロセスが立ち上がっていることを確認する。

これらの監視項目に関して、スケーラブルな監視方式の検討を行う。

本報告書では、オープンソースソフトウェアである可視化ソフトウェアである Ganglia を、可視化だけでなく、監視のための基盤として再利用することとする。なお、Ganglia の可視化機能に関しては 12 章に検討結果が記載されているので参照されたい。

Ganglia は基本的なアーキテクチャがスケーラブルである。Ganglia の情報収集エージェントである gmond は、（一般的にはラック単位の）メンバ全員のリソース情報を共有している。したがって、監視サーバは、代表の gmond に対して情報を照会することで、情報を共有している全てのメンバの情報を取得することができる。

上記 Ganglia のスケーラブルな情報収集の仕組みを監視システムでも活用する方向で検討した。具体的には、Ganglia で収集する情報として、表 11-13 で取得する情報を Ganglia のメトリック（可視化対象情報）として収集する。次にこれらの情報を監視サーバが代表であるところの gmond に照会し、その結果(XML)をパースすることで、メンバ全員の監視情報を取得するように設定した。図 11-4 に概念図を、表 11-14 にそのフローを記載する。

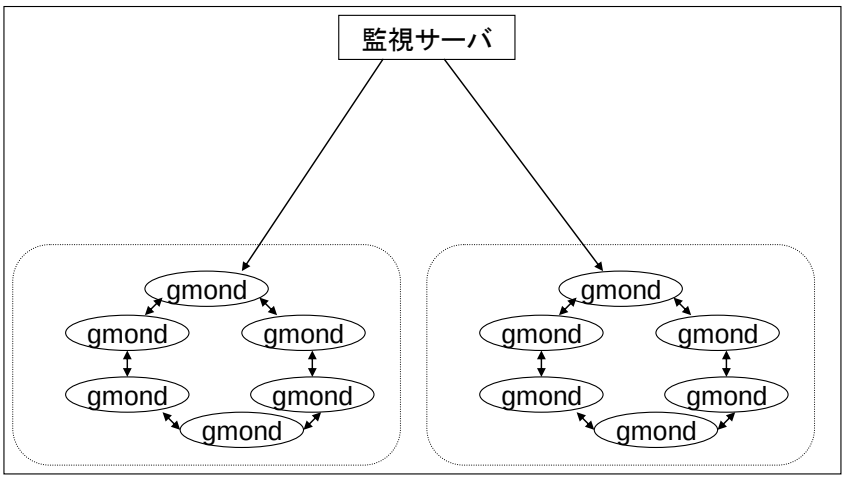


図 11-4 監視サーバと Ganglia 連携方式

表 11-14 Ganglia を用いたスケーラブルな監視方式の実現フロー

No.	項目	概要
1	Ganglia で監視項目の metric を収集	プロセス数や HDD 残容量などをメトリックとして定義する。
2	各サーバの metric を gmond クラスタで共有	Ganglia によって自動的に収集される。
3	監視サーバは代表サーバを選定し、gmond にメトリックを照会する	gmond クラスタメンバ全員のメトリックが得られる情報を含んだ応答を返す。
4	閾値監視を実施する	応答を解析するプログラムから、必要情報を取

No.	項目	概要
		得して、得られた値から閾値判定を実施する。

このように Hadoop スレーブサーバに関して、Ganglia を監視のプラットホームとしたスケーラブルな監視方式を実現することで、Hadoop 基盤におけるスケーラブルな監視を実現した。図 11-5 に典型的なスクリーンショットを記載する。

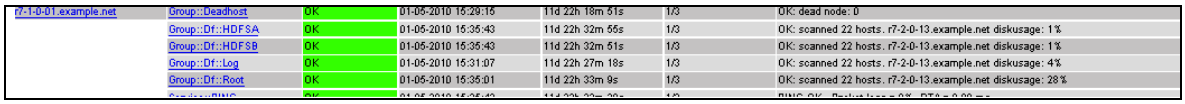


図 11-5 Ganglia と連携した監視システムのスクリーンショット

上記例では、ラック番号 7 のサーバ群から、代表サーバ r7-1-0-01 を選出し、代表サーバに問い合わせることで、ラック番号 7 の全てのノードにおけるログ領域の残容量を検査し、全てのサーバにおいて閾値を下回っている様子を表現している。

11.4.5 Hadoop クラスタにおける構成管理とデプロイ方式

Hadoop 基盤では、多数のハードウェアが存在し、それらの上に複数のミドルウェア及びそれを動作させるための多数の設定ファイルが存在する。これらの情報を維持管理することは、手作業では膨大な時間がかかる。例を挙げる。Hadoop 基盤を構成するサーバに関して、OS や、Hadoop 以外のミドルウェアの設定ファイルは 30 ファイル程度存在する。また、Hadoop を動作させるためには、設定すべき設定ファイルは 10 ファイル前後存在する。例を挙げる。本章で想定する構成では、100 台の Hadoop 基盤から構成されているために、総じておよそ数千ファイルが意図通りに配布されているか、また、それを実行する実行ファイルが想定どおりの設定であることを保障することが重要である。

すなわち、これらの資源を配布する際に 100 台のサーバに同一のファイルを斉一な手順で（ファイルだけでなく、パーミッションなどのメタ情報を含めて）配布することは解決すべき課題である。

そこで、オープンソースソフトウェアの構成管理ソフトウェアである Puppet を利用して、構成管理を実施する。維持管理においても Puppet を利用することで、サーバにおける設定ファイル及びソフトウェアのミドルウェアの構成管理をする。図 11-6 に Puppet による構成管理の概念図を記載する。

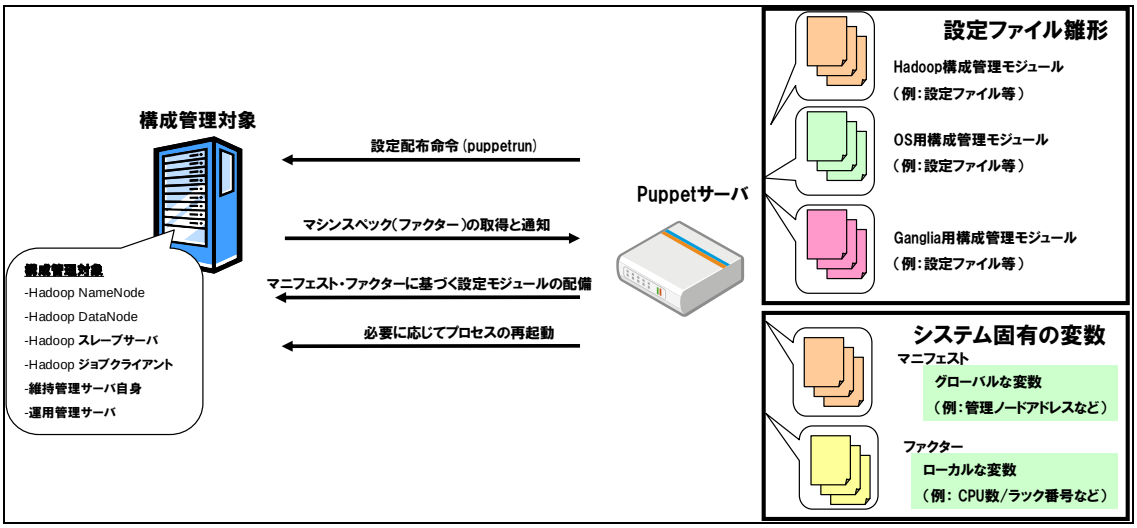


図 11-6 Puppet による構成管理

Puppet は表 11-5 に記載する機能を持っている。

表 11-15 Puppet による構成管理の特徴

No.	Puppet の機能	概要
1	一元管理	サーバサイドに設定ファイルや配布バイナリのマスターを保持することで構成を一元管理することができる。
2	冪等性・push 型配信	適用を繰り返しても同じ設定ファイルになることが保障される。またサーバ側の操作によって設定したいクライアントへ反映を実行することができる。
3	チェックサムによる同一性の保障	配布先のサーバで構成されている設定ファイルやバイナリとマスターで一元管理されているリソースの md5 チェックサムを比較しているために、想定の設定が確実に配布されることが保障される

Puppet を利用するに当たり、以下の項目を充足するように Puppet の設計を行う必要がある。これによって増設や回復という運用項目に関して、後から Hadoop 基盤に参加した機器に関しても既存の構成を保つことが保障される。

- アプリケーション単位で設定ファイル群をモジュール化し、テンプレート（雛形）と、変数値（案件固有情報）を分離し、汎用的なつくりとする
- サーバスペックが混在しており、スペックに応じた設定ファイルの設定を行う必要がある（例：MapReduce 設定ファイル、HDFS 設定ファイル...）
- スケーラブルな方式とするために、サーバ側から一括して設定変更を行う必要

がある

- Hadoop スレーブサーバへの変更は Puppet からしか行わない

そこで、アプリケーションごとのモジュールの作成、サーバスペックの特徴量を反映したローカル変数（factor）の実装、puppetrun による配布の実行を実装することとした。Puppet モジュールの実装等に関しては、初期構築時にも利用しているため、実装の詳細は、15.4.9 を参照されたい。

11.4.6 Hadoop スレーブサーバに対する各種コマンド実行方式

Hadoop スレーブサーバに関して、設定の配布及びそれに紐付く各種設定の反映は Puppet により実現したが、コマンドの実行は Puppet ではカバーできない。そこで、全ての（もしくは任意の）Hadoop スレーブサーバに対して同一の操作を実施するための運用ツールが必要である。そこで、表 11-16 の要件を満たすような運用ツールを作成し、主に、維持管理・メンテナンスにおける各種コマンド実行手法を検討した。

表 11-16 維持管理・メンテナンスのための運用ツール要件

No.	機能要件	概要
1	対象の選択	全てのサーバか個々のサーバかを指定することができる
2	実行単位	複数の任意のコマンドを指定された順で実行する
3	成果	出力結果が出力され、正常に実行できたかを確認する

なお、本ツールを構成変更のために使用すると、Puppet による構成管理がなされなくなる恐れがあるため、本ツールによる構成変更を禁止する。構成変更は、Puppet によってのみ許可するという運用規約を設ける。後から Hadoop 基盤に参加した、機器に関しても、Puppet による構成管理下となるため、既存の構成と斉一となることが保障される。

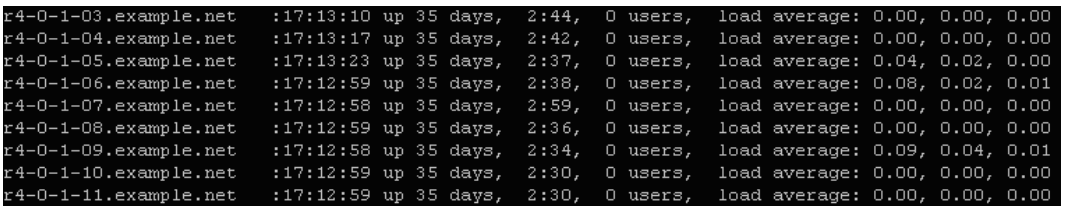


図 11-7 運用ツールによる一括コマンド実行のスクリーンショット

11.5 Hadoop 基盤における運用手法

本節では、11.3 で抽出した Hadoop 基盤の各運用項目に対して、11.4 で検討した効率化のための運用方式を利用してどのように運用手法とするかの詳細を記載する。

11.5.1 初期構築における運用手法

初期構築における運用項目を表 11-17 の手法で実現した。効率化が必要な項目番号 No6 に関しては、前節における自動構築方式により構築した。

表 11-17 初期構築における運用項目の実現方法

No.	項目	実現方法
1	コア L3 スイッチの構築	事前に設定ファイルを用意しておき、最低限の設定をした後、ネットワーク経由で設定を反映させる。
2	L2 スイッチの構築	事前に設定ファイルを用意しておき、最低限の設定をした後、ネットワーク経由で設定を反映させる。
3	維持管理サーバの構築	設計書に従い、サーバを構築する。
4	運用管理サーバの構築	設計書に従い、サーバを構築する。
5	Hadoop マスターサーバの構築	設計書に従い、サーバを構築する。
6	Hadoop スレーブサーバの構築	11.4 で検討した Hadoop スレーブサーバ自動構築手法により構築した。

11.5.2 監視・故障検知における運用手法

監視・故障検知における運用項目を表 11-18 の手法で実現した。

表 11-18 監視・故障検知における運用項目の実現方法

No	項目	実現方法
1	スイッチの監視	監視ソフトウェアにより、SNMP トラップ監視、死活監視を実施する。
2	Hadoop マスターサーバ・管理系サーバの監視	監視ソフトウェアにより、死活監視、リソース監視、プロセス監視、サービス監視を実施する。
3	Hadoop スレーブサーバの監視	14.4 で検討した Hadoop スレーブサーバの監視手法により、プロセス監視、リソース監視、死活監視を実施する。
4	全体の監視	11.4 で検討した Hadoop クラスタの可視化方式で監視する。また、監視ソフトウェアにより、JobTracker

No	項目	実現方法
		及び NameNode に情報を問い合わせるスクリプトを経由してサービス監視を行う。

図 11-8 では、Hadoop マスターサーバの JobTracker の監視画面を記載している。JobTracker のプロセスが落ちている際に監視コンソールから、プロセス障害を検知できていることがわかる。

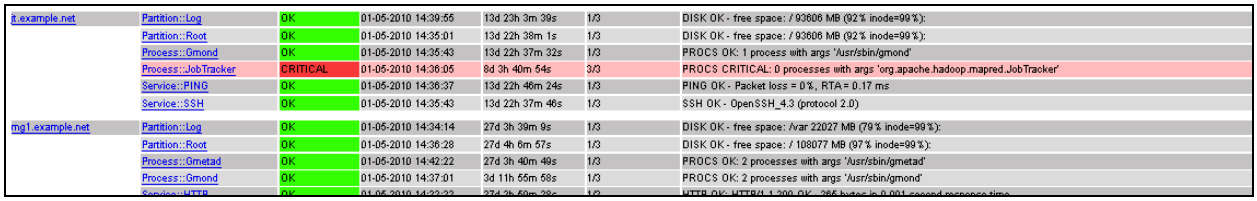


図 11-8 Hadoop マスターサーバ監視のスクリーンショット

図 11-9 では、ラック番号 7 のサーバ群から、代表サーバ r7-1-0-01 を選出し、代表サーバに問い合わせることで、ラック番号 7 の全てのノードにおけるログ領域の残容量を検査し、全てのサーバにおいて閾値を下回っている様子を表現している。

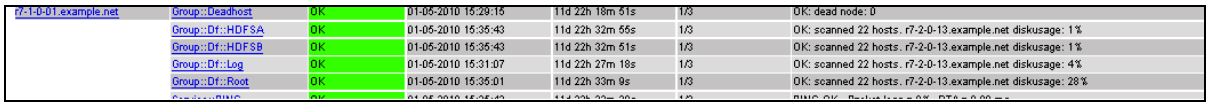


図 11-9 Hadoop スレーブサーバのスクリーンショット

11.5.3 維持管理・メンテナンスにおける運用手法

維持管理における運用項目を表 11-19 のように実現した。

表 11-19 維持管理における運用項目一覧

No.	運用項目	概要
1	ハードウェア構成情報の一覧化	サーバの NIC の eth0 の MAC アドレスを主キーとしたマシン構成情報の管理表による一覧化を行う。
2	ハードウェア構成情報の変更	ハードウェアの故障/回復履歴を一覧化する。
3	ソフトウェア構成情報の一覧化	開発機によるバージョン管理システムにおいて履歴管理を行う。
4	ソフトウェア構成情報	11.4 で検討した Hadoop クラスタにおける構成管理とデ

No.	運用項目	概要
	の配布と反映確認	プロイ方式により実施する。
5	設定変更	11.4 で検討した Hadoop クラスタにおける構成管理とデプロイ方式により実施する。
6	ミドルウェア配布	11.4 で検討した Hadoop クラスタにおける構成管理とデプロイ方式により実施する。

メンテナンスにおける運用項目を表 11-20 のように実現した。

表 11-20 メンテナンスにおける運用項目一覧

No.	運用項目	概要
1	系全体の閉塞	Hadoop マスターサーバを停止後、11.4 で検討した Hadoop スレーブサーバに対する各種コマンド実行方式により Hadoop スレーブサーバを停止する。
2	系全体の開放	Hadoop マスターサーバを開始後、11.4 で検討した Hadoop スレーブサーバに対する各種コマンド実行方式により Hadoop スレーブサーバを開始する。
3	サーバ単位の閉塞	対象サーバを停止すると、Hadoop サービスが縮退する。
4	サーバ単位の開放	閉塞したサーバを開始すると、Hadoop クラスタに自動的に組み込まれる。
5	HDFS リバランス	NameNode 上で HDFS リバランスコマンドを実行する。
6	ログローテション	OS の機能を利用する。
7	ログ収集	定期的に、全てのサーバに対して、ログの収集を実行する。
8	時刻同期	OS の機能を利用する。
9	統計情報のバックアップ	統計情報を定期的にバックアップする。11.4 で検討した Hadoop スレーブサーバに対する各種コマンド実行方式を利用する。
10	構成情報のバックアップ	11.4 で検討した Hadoop クラスタにおける構成管理とデプロイ方式により実施する。
11	HDFS メタ情報のバックアップ	SecondaryNameNode を JobTracker 上で動作させる。

11.5.4 回復・増設における運用手法

回復・増設における運用項目を表 11-21、表 11-22 のように実現した。

表 11-21 回復・増設における運用項目一覧

No.	故障対象	運用項目	概要
1	コア L3 スイッチ	機器故障からの回復	機器交換を実施する。
2		ポート故障からの回復	使用可能なポートで代用する。
3	ラック内 L2 スイッチ	機器故障からの回復	即座に機器交換を実施する。
4		ポート故障からの回復	使用可能なポートで代用する。
5	Hadoop マスターサーバ	ソフトウェア故障からの回復	プロセス再起動を試み、さらに動作不良であれば OS の再起動。
6		機器故障からの回復	ベンダの診断ツールを利用して故障箇所を明らかにし、機器交換を実施する。
7	Hadoop スレーブサーバ	ソフトウェア故障からの回復	プロセス再起動を試み、さらに動作不良であれば OS の再起動、さらに動作不良であれば、再構築する。
8		機器故障からの回復	故障管理簿に記載し、一定数故障ノードがたまったらまとめて交換し、自動構築方式により再構築する。
9	構成管理サーバ・運用管理サーバ	ソフトウェア故障からの回復	プロセス再起動を試み、さらに動作不良であれば OS の再起動。
10		機器故障からの回復	ベンダの診断ツールを利用して故障箇所を明らかにし、機器交換を実施する。

表 11-22 回復・増設における運用項目一覧

No.	運用項目	概要
1	自動構築のための設定	構成管理サーバのクライアント証明書を削除する。
2	監視・可視化設定	監視サーバにおいて、対象ノードのエントリを追加する。可視化サーバ側は設定する必要がない。

11.6　まとめと今後の課題

本章の結果及び考察を行う。

11.6.1 まとめ

本章では、Hadoop 基盤における運用項目を一覧化した。また、その中で効率化が必須である Hadoop スレーブサーバに関する運用項目に焦点をあて、運用上効率化するべき項目を抽出した。そして、効率化のため以下のような実現方式を検討した。

- Hadoop スレーブサーバの自動構築方式
- Hadoop クラスタの可視化方式
- Hadoop スレーブサーバの監視方式
- Hadoop クラスタにおける構成管理とデプロイ方式

この方式によって効率化するべき運用項目の運用手法を実現した。

11.6.2 今後の課題

本章では Hadoop スレーブサーバの運用項目について効率的な運用を確認した。一方、Hadoop の長期的な運用を考慮すると、運用中の Hadoop 基盤のバージョンアップに伴う移行手法の確立も必要である。現在の Hadoop では、バージョン間の互換性を保障していない。そのため、それらに関する運用の詳細は、現時点では検討することができない。Hadoop のコミュニティでは将来、互換性を保ったバージョン間移行が予定されている。今後のコミュニティの動向に注目することが重要である。

12 Hadoop 基盤における自動構築手法検討

本章では、Hadoop 基盤のための基盤構築の手法に関して検討結果及びその実装方式を報告する。

クラウド型分散処理基盤では、大量の Hadoop スレーブサーバが存在するが、これらのサーバの自動構築に焦点をあてる。 自動構築手法を検討し、RedHat Enterprise Linux のネットワークインストールツールである Kickstart と、オープンソースソフトウェアである Puppet を組み合わせることで実現することとした。この方式を実現するために三つの実装上の問題を明らかにし、それに対する実現方式を提示する。

12.1 初期構築のあるべき姿と課題

本節では Hadoop 基盤構築の際に検討するべき項目を、Hadoop 基盤の特徴から明らかにするとともに現在の課題を洗い出す。

12.1.1 Hadoop 基盤の特徴

Hadoop 基盤は、少数の Hadoop マスタサーバ（NameNode・JobTracker）と大量の Hadoop スレーブサーバ（DataNode・TaskTracker）から構成される。表 12-1 に典型的な Hadoop 基盤における構成要素を列挙する。

表 12-1 100 台規模の Hadoop 基盤の構成要素

No.	構成要素	概要	台数	規模に応じて増加
1	スイッチ	各サーバを結線する	8 台	○
2	Hadoop マスタサーバ	NameNode , JobTracker	4 台	-
3	Hadoop スレーブサーバ	DataNode, TaskTracker	96 台	○
4	ジョブ実行端末	Hadoop ジョブを実行する	1 台	-
5	構成管理サーバ	IP アドレス管理や名前管理を行う	2 台	-
6	運用サーバ	監視や可視化のサービスを提供する	2 台	-

初期構築において方式的な検討が必要な点は、Hadoop 環境で大量に存在する構成要素である Hadoop スレーブサーバの構築である。

上記の構成では、Hadoop スレーブサーバは 96 台であるが、より大規模な分散計算を実施しようとする場合、スイッチ及び Hadoop スレーブサーバは、規模に応じて台数が増加する。上記よりも大規模なシステム、例えば 1000 台規模の Hadoop 基盤においては、スイッチは 80 台程度、Hadoop スレーブサーバは 1000 台規模にも達する。一方、その他の構成要素に関しては台数に大きな変化はない。

このように Hadoop 基盤の構成要素は、規模に応じて増加する構成要素（L2 スイッチ及び Hadoop スレーブサーバ）と規模が増大しても増加しない構成要素（Hadoop マスタサーバや構成 Hadoop マスタサーバ、運用サーバ）に大別される。

本章では、台数の多い計算サーバの自動構築に対して、効率的な初期構築を目的として取り上げ、報告する。

12.1.2 Hadoop スレーブサーバ構築のあるべき姿と課題

前節の通り、初期構築を実施する際に方式的な検討が必要な箇所は、Hadoop 環境で大量に存在する構成要素である Hadoop スレーブサーバの構築である。

これらのサーバは大量に存在するため、OS のインストールやミドルウェアのインストール及びその構成に関して、手作業による基盤構築では大量の工数がかかる上、作業の抜けや漏れなどが発生しうる。さらにそれらの作業ミスを検知するのは困難である。

また、Hadoop スレーブサーバは一般的なサーバと比して安価な IA サーバで構成される。それに加え、Hadoop スレーブサーバは大量に存在することから、その故障および回復作業が定常的におこると想定され、環境の再構築が頻繁に必要となる。

増設や、故障からの回復における構築は、初期構築だけでなく、他の Hadoop スレーブサーバに初期構築後になされた全ての設定反映などを漏れなくすることが必要である。これらの初期構築以外の運用に伴うコストは、システム規模が大きくなるに伴い増大するため、これらは手順化するだけでなく、自動化が必要である。

上記の特徴を鑑み、Hadoop スレーブサーバの構築は、完全に自動化すること、及び構築後の構成管理と連動した構築手法が必要であると考えた。上記を表 12-2 にまとめる。

表 12-2 Hadoop スレーブサーバ構築のあるべき姿と課題

No.	あるべき姿	課題	課題の解決方法
1	運用者の作業時間を短縮することができる	手順書に基づく手作業のため、時間がかかる	自動化による作業時間の短縮
2	全てのサーバが同一の手順で構築されていることが保障されている	作業漏れや、作業ミスが混在しうる。またその検知が難しい	自動化によって斉一な手順を保障
3	故障からの回復や、増設時にも同じ手順で Hadoop スレーブサーバが構築できる	初期構築後の設定反映を漏れなく反映させるのが難しい	自動構築及びシステマティックな構成管理

12.2　前提条件

本節では自動構築のための前提条件を記載する。自動構築を実施する際のシステム構成や、自動構築のための前提条件を列挙するとともに、自動構築の目標を定義する。

12.2.1 適用されるシステム構成

本章における想定するシステム構成は、図 12-1 のとおりである。

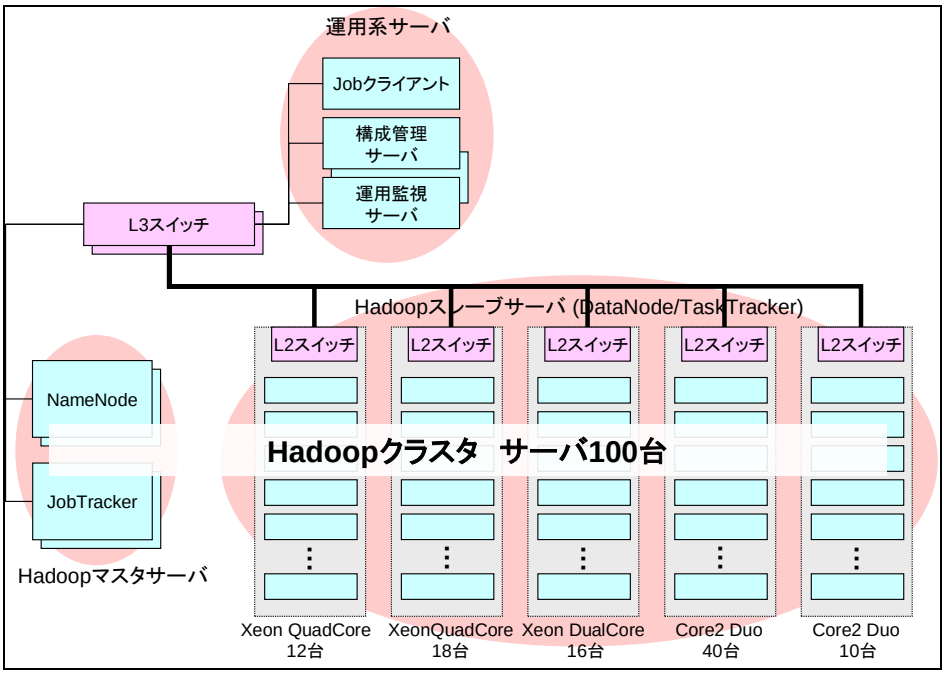


図 12-1 システム構成図

システム構成は以下の特徴を有する。

・構築対象と維持管理サーバ、運用管理サーバが L3 スイッチでルーティング可能である

・L3 スイッチは DHCP リレー機能を有する

・構築対象はネットワークブート可能なサーバである

・ラッキング及びケーブリングは完了済である

・各装置間の物理配線が一覧化されている

12.2.2 本章における初期構築のスコープ定義

本章では、Hadoop 基盤の構築のうち、12.1 で述べたように、スケーラビリティの観点から検討が必要な Hadoop スレーブサーバの構築方式の検討をする。

また、本章では、Hadoop 基盤の構築のうち、上記以外の方式検討はスコープ外と

する。例示すると以下は検討の対象外である。

・分電盤の工事、データセンタ内のエアフロー設計、ラックの設置

・各装置のラッキング及びケーブリング

・Hadoop マスタサーバの構築

12.2.3 構築に求められる要件

初期構築に求められる要件は、OS や分散計算に必要なソフトウェアや管理のためのソフトウェアがインストールされ、利用できること。また、可視化・監視の対象となることとする。

12.3 課題解決方法の検討

本節では、必要要件を満たすような自動構築方法および構成管理手法を比較検討し、実装方式を検討する。

12.3.1 Hadoop スレーブサーバ自動構築のための実現方式の比較観点

Hadoop スレーブサーバの自動構築の方式検討に先立ち、Hadoop スレーブサーバの構築に求められる要件を表 12-2 を元に詳細化する。

まず、Hadoop スレーブサーバは完全に同一スペックのサーバであることを前提としてはならない。修理による機器変更や、増設に伴う機器の変更も頻繁に生じうる。すなわち、Hadoop スレーブサーバはスペックや調達ベンダが異なる環境となることを前提として自動構築を実現する必要がある。

また、上記に伴い、構成管理に関しても要件の詳細化が必要である。スペックが非均一であるため、サーバごとに必要な設定も異なる。すなわち混在環境において、サーバごとに異なる最適な設定を配布・維持することが必要となる。

更に、自動構築サーバが容易に構築できることも重要な比較観点となる。具体的には、設計や実装が容易であり、短期間で構築が可能であること、及び、それらの変更点が明確で、類似システムを再度構築する際に容易にカスタマイズできるかという点を比較観点とし、自動構築対象が数百～数千台でも実現方式が変わらないことも比較観点とした。

上記前提と、前述の要件を併せて、Hadoop スレーブサーバの自動構築手法の比較観点を表 12-3 のように整理した。

表 12-3 構築自動化の実現方式の比較観点

No.	概要
1	手作業の構築と比較して、運用者の作業時間を短縮することができるか

No.	概要
2	同一の手順で構築されていることが保障されるか
3	サーバの機種等が混在していることを前提に基盤構築ができるか
4	機器が混合環境している際に機器ごとに最適な構成を配布維持できるか
5	設計や実装が容易か
6	プロジェクトごとにカスタマイズが容易か
7	数百～数千台でも実現方式が変わらないか

12.3.2 実現方式の比較

ハイパフォーマンスコンピューティング（HPC）の分野で、自動構築手法はいくつかのツールが存在する。HPC は大量の計算ノードが協調して、グリッド計算と呼ばれる並列処理を実現するものであり、Hadoop 基盤と性質が一部類似している。HPC における計算ノードの自動構築ツールに関しては、オープンソースソフトウェアでの実装もいくつか存在する。

これらの多くはすでに実績があるが、Hadoop 環境で最適であるかは検討の余地がある。そこで、既に実績のある以下の製品に関して机上による比較検討を実施し、自動構築の方式として採用することとした。なお、非対話なインストール手法の標準である Kickstart を利用するツールを選定した。比較対象としては以下である。

- Kickstart+ Puppet
  Kickstart に加え、オープンソースソフトウェアの構成管理ツールである Puppet を組み合わせる。
- rocks
  Kickstart によるインストールを管理するサーバを用意し、必要なパッケージをまとめたロールと呼ばれるアドオンを利用して、特定用途のクラスタを構築できるツールである。
- OSCAR
  Kickstart の設定を GUI ベースにすることができる製品である。簡易に Kickstart が設定できる上、自動インストール対象をデータベース化することで、クラスタ全体の管理を行うことができる。

表 12-4 に評価結果を記載する。

表 12-4 実現方式の評価結果

No.	比較観点	Kickstart+Puppet	rocks	OSCAR
1	ツール概要	OS のネットワークインストール後に構成管理サービスで配布	必要なモジュールを組み合わせてネットワークインストール	ウィザードに従って構成サーバ上でクラスタを GUI で設定
2	作業時間を短縮できるか	○	○	○
3	同一の手順で構築されていることが保障されるか	○	○	○
4	サーバの機種等が混在していることを前提に基盤構築ができるか	△	△	△
5	機器が混合環境している際に機器ごとに最適な構成を配布維持できるか	△	× 単体では不可能	× 単体では不可能
6	設計や実装が容易か	△	○ GUI 画面で容易にセットアップが可能	○ GUI 画面で容易にセットアップが可能
7	プロジェクトごとにカスタマイズが容易か	○	△ Roll と呼ばれるものを作成する必要がある	× 単体では不可能
8	数百～数千台でも実現方式が変わらないか	○	△	△

Kickstart は、方式が単純なため、追加に機能の実装が可能であり、細かい調整を行うことが可能である。rocks や OSCAR は対象サーバが数百台にもなると、構築時に手動で設定するべき項目が増大することや、初期構築時に専用のサーバが必要となるうえ、実現が難しい項目も見られる。特に Rocks, OSCAR はクラスタメンバが全て均一のマシンスペックを想定しており、ベンダやスペックが非均一な環境を想定して作られていない。

上記比較に基づいて、Kickstart と Puppet の組み合わせで自動構築環境を作成することとした。

12.4 Kickstart と puppet による自動構築の検討

本節では、自動構築手法で採用した KickStart による自動インストールの流れを記載し、適用上追加実装が必要な３つの問題を提示し、その実装方法を報告する。また、自動構築を実施するためには DNS サービス、DHCP サービス、TFTP サービス、HTTP サービス、Puppet サービスをあらかじめ構成しておく必要がある。本節ではこれらのサービスの設定方針を記載する。

12.4.1 Kickstart による標準的な自動構築概要

Kickstart とは、ネットワークインストールとあらかじめ作成しておいた設定ファイル（キックスタートファイル）にもとづいてサーバのネットワークインストールを行う手法である。

従来の Kickstart では以下のようにサーバを構成することができる。

・ディスクレイアウトの決定とフォーマット
・OS のインストール
・必要なパッケージのインストール
・全てのサーバに同一の設定で設定

このように Kickstart によるインストールは、大量のサーバに対して同一の設定ファイルに基づいて OS のインストール等の設定を実施することができる。

しかし、サーバごとに異なる設定を実施するためには、事前に全てのマシンの MAC アドレスを用意し、サーバごと異なるにキックスタートファイルを作成する必要性がある。サーバごとに異なる設定は以下のものが挙げられる。

・IP アドレス, ホスト名
・サーバスペックに基づく設定ファイル

しかし、96 台のサーバの MAC アドレス一覧が事前に入手することは、調達要件として一般的でないと考えた。そこで、これらを前提とせずに Kickstart ファイルによるインストールが可能であるようにする必要がある。

12.4.2 Kickstart を利用した自動構築の際の流れ

本節では、一般的な Kickstart による自動構築の仕組みを表 12-5 及び、表 12-6 に記述する。なお、DHCP サーバや DNS サーバなどの構成要素に関しては、12.4.8 に記載があるので参照されたい。

表 12-5 一般的な Kickstart における処理フロー

No.	実施項目	結果	手動/自動
1	電源を ON にしてネットワークブートを指定	PXE ブートモードで起動	手動
2	DHCP サーバに IP の払出を申請	DHCPDISCOVER の送出	自動
3	DHCP サーバの応答に基づいて IP を取得. ネットワークブート用のローダを取得、一次ローダの起動	TFTP サーバ上の一時ローダで起動	自動
4	ローダは HTTP サーバに配置された設定(Kickstart ファイル)に基づきインストーラを動作	OS のインストーラの起動	自動
5	Kickstart ファイルに基づき OS・各種ミドルウェアがインストールされる。	OS のインストール	自動

以下に Kickstart で必要なデータを記載する。

表 12-6 一般的な Kickstart におけるデータ

No.	データ	概要	保持するサーバ
1	割当 IP アドレス	PXE ブートモードで起動するローダ	DHCP サーバ
2	1 次ローダ	PXE ブートする際に読み込む、基本 OS を読み込むための 1 次ローダ	TFTP サーバ
3	OS のインストーラ	OS をインストールするためのプログラム	HTTP サーバ
4	OS インストーラ用設定ファイル（Kickstart ファイル）	インストーラを動作させるための設定ファイル	HTTP サーバ
5	インストールパッケージ	OS のインストールに必要なパ	HTTP サーバ

No.	データ	概要	保持するサーバ
		ッケージ	

12.4.3 Puppet による標準的な設定の配布

Puppet は資材配布及び構成管理を実現するオープンソースソフトウェアである。同じくオープンソースソフトウェアである Ruby 言語で実装されている。Puppet サーバは以下の機能を提供する。また、動作概要を図 12-2 に記載する。

- 設定ファイル群をモジュール化し、テンプレート化して一元管理
- 複数のサーバの設定を一つの管理サーバにて集約管理
- 管理サーバは設定ファイルを対象サーバに一括配布することが可能

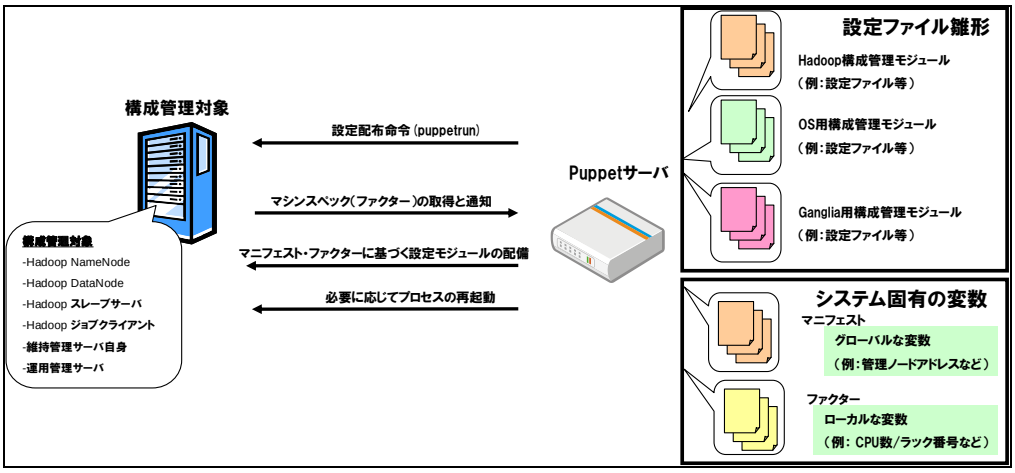


図 12-2 Puppet による設定の配布

12.4.4 Kickstart と Puppet での自動構築における主な問題

前述の通り、Hadoop スレーブサーバの自動構築を RedHat Enterprise Linux の自動インストールツールである Kickstart と、オープンソースソフトウェアである Puppet を組み合わせることで実現する。しかし、これらを適切に組み合わせるためには、解決すべき問題が存在する。これらを明らかにし、実現方式を検討する。

解決するための問題を、台数が多いことに基づく問題と、サーバ構成の非均一性に基づく問題の二つに大別する。

台数が多いことに基づく問題を「表 12-7 Kickstart で解決するべき問題」の No1 に記載し、サーバ構成の非均一性に基づく問題を No2 に記載し、詳細に関してはサブセクションに関して記述する。

表 12-7 Kickstart で解決するべき問題

No	問題
1	**物理配置と論理配置の対応付け**
1-1	名前解決の方式検討が必要
1-2	物理配置を反映した命名方式が必要
2	**Hadoop スレーブサーバのスペックが非均一であることに基づく設定配布**
2-1	スペックの非均一性に基づく最適な設定が必要

問題 1-1 の概要を記載する。大量の Hadoop スレーブサーバが存在し、複数のラック間に分散して配置される。これらのサーバに標準の Kickstart インストールを実施すると、サーバの IP アドレスが任意に付与される。そのため、任意に付与された IP アドレスに意味づけを持たせるために、ホスト名をどのように対応づけるか（名前解決をどのように実現するか）を決定する必要がある。

問題 1-2 の概要を記載する。論理配置（IP アドレスやホスト名）と、そのサーバの物理的な配置（どのラックの何番目に格納されているか、どのようなネットワークトポロジにて定義されているか）の対応付けが自明でないということがあげられる（例：IP アドレス 192.168.3.40 のマシンがどのラックのどの場所に配置されているかわからない)。故障監視や運用監視などの運用工程を考えると、サーバが物理的にどの場所に配置されているかをトラッキングすることが重要である。

問題 2-1 の概要を記載する。Hadoop スレーブサーバは同一機種・同一ベンダのサーバではなく、非均一なスペックを持ったサーバである。よってスペックに応じた最適な設定ファイルを配布する必要がある。具体的には、マシンの CPU 数や搭載メモリを反映した設定ファイルの配布が必要である。

12.4.5 名前解決の方式検討

前述の問題 1-1 に関する方式検討を行う。

上記問題 に対する実現方式としては、二つの方式が考えられる。

一つは Kickstart において付与される IP アドレスを、インストール機器の MAC アドレスで IP アドレスを指定する方式である。この方式では、以下が前提条件として挙げられる。

- 事前にインストール対象となるサーバのネットワークインターフェースカード（以降 NIC）の MAC アドレスが事前にわかっていること。
- MAC アドレスの一覧と、Kickstart 実施時に割り当てる IP アドレスの一覧のリストを設定ファイルとして用意しておくこと。

・ 上記リストに対応した名前解決手法をあらかじめ用意しておくこと（ホストファイルや DNS エントリなど）。

この方式では、あらかじめサーバの MAC アドレスの一覧と付与するべき名前を事前に一覧化する必要がある。

今回はこの方式は採用しない。以下の理由による。

・ 事前に NIC の MAC アドレスの一覧を取得するのは一般的に難しい（ベンダの調達要件としては一般的ではない）また、NIC 故障などの際にこれらの情報を事前に知ることは難しい。
・ MAC アドレスと IP アドレスの対応関係のファイルの維持管理が難しい。これらは DHCP サーバの設定ファイルや、各サーバに配布した hosts ファイルなどリストが分散してしまうため、運用コストが大きくなるとともに、作業ミスを引き起こす可能性が高いためである。

二つめの方式であり、本章で採用する方式は以下の通りである。

Kickstart インストール時に任意に割り当てられた IP アドレスに対して、各サーバにおいてホスト名を自動生成し、そのホスト名と IP アドレスをマッピングするような設定をその場で実施する。具体的には、ホスト名を自動生成し、そのホスト名と IP アドレスの対応づけを動的に DNS レコードとして更新する。この方式を採用する理由は以下の通りである。

・ 第一の方式のように前提となる調達要件が必要ない。
・ サーバの増減に関して自動的にシステムで管理ができるため、運用コストが低く抑えられる。

表 12-8 に上記の二つの方式の比較を行う。便宜的に一つ目の方式を MAC 方式、二つ目の方式を DDNS 方式と記載する。

表 12-8 自動構築における名前解決方式の比較

No.	比較観点	MAC 方式	DDNS 方式
1	名前解決方式	事前に用意した hosts ファイルもしくは DNS の A レコード	インストール時に自動構成される A レコード
2	名前解決の実体	・DHCP サーバの設定ファイル ・各サーバの hosts ファイル	BIND の A レコード
3	前提条件	事前に全てのサーバの MAC アドレスがわかっていること	特になし
4	実現方式	設定ファイルの記述のみ	ホスト名自動生成のためのスクリプトの実装が必要

No.	比較観点	MAC 方式	DDNS 方式
5	構成要素	-	MAC 方式では必要のない DNS サービスが必要

12.4.6 物理配置を反映した命名方式の検討

前述の問題 1-2 に関する方式検討を行う。

IP アドレス及びホスト名を物理配置と対応づける必要がある。Hadoop アプリケーションの動作のためには、論理的にサーバを一意に指定することができれば問題はない。しかし、運用を考えるとサーバ名に物理配置を反映した情報を付与することは必須である。具体的には、故障サーバの物理位置の特定や、特定のサーバの故障の履歴をトラッキングするなどの運用である。

論理配置と物理配置を対応付ける一番明確な方法は、サーバのホスト名に物理的な配置を記載することである。例えば、サーバのホスト名を以下のような命名規則にすることである。

表 12-9　論理配置と物理配置の対応（サーバ名の命名規約）

No.	サーバ名の命名規約例
1	ラック番号とラック位置の組み合わせのホスト名とする 例）rack1-13u.example.net
2	ラック番号とラックに配置されているスイッチのインターフェース番号を組み合わせてホスト名とする 例）rack1switch-port13.example.net

本検証では上記の例２を採用する。例１を採用するためには、事前にどのサーバがどのラックのどの番号にいるのかの対応表及び、それらを管理する必要がありこれらは自動化できないため、追加の運用コストが発生する。一方、例２の方法はインストール時に自動的に構成することが可能であるために、管理コストは少ない。

また、Hadoop 基盤では、スイッチに接続されたサーバ群（これは一般的にはラック単位と同意）を一つの計算単位として捉える。このような観点では、スイッチを主体としたトポロジを反映した例２のようなホスト名を自動生成する方式が適合する。

ホスト名を例２に従って自動的に生成するため、表 12-10 の方式のスクリプトを実装した。

表 12-10 物理的な識別子を意識したホスト名生成のフロー

No.	手順
1	サーバの MAC アドレスを取得する
2	Hadoop 環境の L2 スイッチにログインし mac-address-table(MAC アドレスとスイッチのインターフェースの対応表）を取得する
3	自身の MAC アドレスと対応表を照合して、自身がスイッチのどのポートにつながっているかを判断する
4	接続先のスイッチポートからホスト名を自動生成する
5	自動生成されたホスト名と IP アドレスの対応をダイナミック DNS に登録する

以下にスクリプトの出力の具体例を示す。ラック番号 3 のスイッチのポート番号 1/0/12 に接続されたサーバにおいて、スクリプトを実行すると以下の通りとなり、あらかじめホスト名を決めるのではなく、自動的に生成することが可能となる。この情報をもとに、DNS サーバに自身の IP アドレスとホスト名の対応関係、すなわち A レコードを登録することが可能となる。

```
# /root/scripts/myhostname
r3-1-0-12.example.net
```

なお、このように構成するためには、ラックに配置された各スイッチに対して、ラック名を反映した名前を付与しておくことが必要である。なお、本検証において各サーバの物理的な識別子が接続されるスイッチのポート番号と結びつけるため、表 12-11 のような運用規約をしておくとよいことがわかった。

表 12-11 ラッキングの運用規約

No.	運用規約
1	スイッチの各ポート番号に対応したサーバのラッキングを実施することとする。例えばスイッチのポート番号の若い順につながったサーバのラックの下から順に配置する。
2	故障回復など、新たにサーバをラッキングして使用する際には必ずクリーンインストールを実施し、ホスト名と物理配置を一致させる。

12.4.7 スペックの非均一性に対する方式検討

表 12-7 の問題 2-1 に関する方式検討を行う。

まず、サーバスペックが非均一である際の具体例を記載する。

表 12-12 スペックが均一でないサーバ構成の具体例

No.	具体例
1	ベンダに応じて物理ドライブの見え方が異なる(i.e. /dev/sda, /dev/cciss/c0p0)。
2	搭載 CPU 数に応じて各 Hadoop タスク受け入れ上限数を変更する。
3	搭載メモリ数に応じて MapReduce タスクに割り当てるメモリ量を調節する。
4	配置されたラックごとにクラスタリングするような設定を実施する。

上記例のように、スペックが均一でないために、斉一に設定を配布するのではなく、各マシンの物理構成に応じた設定ファイルを設定することが必要になる。これを実現するために、表 12-13 に示すように、OS のインストール時には Kickstart ファイルを物理ドライブ情報に基づき、動的に生成することとした。また、表 12-14 に示すように、Puppet の facter を新規に開発し実現する。

表 12-13 サーバの非均一性に基づく OS のインストールフロー

No.	方式
1	OS のインストール前にサーバに接続されている物理ディスク名を取得するスクリプトを実行するようにキックスタートファイルに反映(%pre スクリプトの実装)
2	OS のインストール時に上記に基づいてディスクのフォーマットを実行するようにキックスタートファイルを動的に構成する(%include によるディスク情報ファイルのインクルード)

表 12-14 サーバの非均一性に基づく設定ファイル配布フロー

No.	方式
1	クライアント側で自身の情報(CPU 数/搭載メモリ/配置ラック)の情報を収集する。Puppet の facter を拡張する。
2	クライアントは、テンプレートファイルをダウンロードし、構成 Hadoop マスタサーバに自身の情報に応じた値をテンプレートに反映し、設定ファイルを生成する
3	生成された設定ファイルを自身に設定する

12.4.8 自動構築サービスを構成するサービスとその設定方針

以降の章では、表 12-7 にあげた問題 1-1、1-2、2-1 の方式検討結果を受けた自動構築のための詳細な検討結果を報告する。Kickstart を実施するためには DNS サービス、DHCP サービス、TFTP サービス、HTTP サービス、Puppet サービスをあらかじめ構成しておく必要がある。本節ではこれらのサービスの設定方針を記載する。

DNS サーバ

システムを構成する各ノードの名前解決を提供する。全てのノード（NW 機器および、サーバ）は example.net.というドメインに参加する。NW 機器や、運用サーバなどの名前はあらかじめゾーン定義ファイルに記載しておくが、計算ノードに関しては、DHCP サーバによるゾーン更新を許可することで、ダイナミック DNS 機能を提供する。DNS サーバは以下の機能を提供するように構成する。

- Hadoop 基盤を構成する全てのノードの名前解決（正引き・逆引き）
- DHCP サーバによるゾーン情報の更新（DynamicDNS 機能）を受け付ける

耐障害のためにセカンダリを用意する

DHCP サーバ

DHCP サーバは、各計算ノードの初期構築時にはクライアントから要求に対して、一次ローダである TFTP サーバを指定するように設定する。DHCP サーバはオープンソースソフトウェアである dhcpd を使用する。なお、DHCP プロトコルはブロードキャストであるため、DHCP リレーをするスイッチが必要である。Hadoop スレーブサーバが複数のセグメントに分類されているため、L3 スイッチに DHCP リレー設定を実施する必要がある。

なお、初期構築以後は、DHCP サーバはあらかじめ固定的な IP アドレスが構成されていないノード(Hadoop スレーブサーバ)に対して IP アドレスの割当を実施する。また、これらのノードに IP アドレスを割り当てられた際に、DHCP サーバは名前解決サーバと連携してダイナミック DNS 機能を提供するように構成する。DHCP サーバは以下の機能を提供するように構成する。

- 固定 IP アドレスが構成されていないノードに IP アドレスの割当を実施する
- 初期構築（ネットワークブート）時に一次ローダの場所を指定する
- DHCP クライアントからの要求に対して DNS サーバに A レコードを登録する

TFTP サーバ

オープンソースソフトウェアである tftpd を利用することとする。なお、TFTP サーバに配置する一次ローダは RedHat Enterprise Linux に同梱されている syslinux パッケージを利用する。TFTP サーバは以下の要件を満たすように構成する

- 初期構築時に一次ローダを提供する
- スイッチの設定ファイルを保存、バックアップするように構成する

HTTP サーバ

HTTP サーバにはオープンソースソフトウェアである Apache を利用することにす

る。HTTP サーバは自動構築時には以下の機能を提供するように構成する。

- ネットワークインストールのための OS のパッケージを提供する
- ネットワークインストールのための設定ファイルを提供する
- 分散基盤計算に必要なソフトウェアパッケージを配布する

また、HTTP サーバは運用監視サーバとしても利用するために、Ganglia 及び Nagios の Web フロントエンドとして動作するように構成する。

Kickstart ファイル

Kickstart ファイルは Kickstart を実施するための設定ファイルである。インストール時には、サーバに搭載されたディスク数や、デバイス名などの情報を判別して Linux のインストールを実施するように構成する。Kickstart の%pre スクリプトにてディスク情報判別のためのスクリプトを埋め込み動的にディスクパーティション設定を、%post スクリプトでホスト名を自動生成し、puppet サーバと通信し、設定を読み取るように構成する。

Puppet サーバ

Puppet は資材配布及び構成管理を実現するオープンソースソフトウェアである。同じくオープンソースソフトウェアである Ruby 言語で実装されている。Puppet サーバは以下の機能を提供するように構成する。

- Hadoop スレーブサーバに関する設定ファイルを一元管理する
- Hadoop スレーブサーバに関する設定ファイルを一括配布する

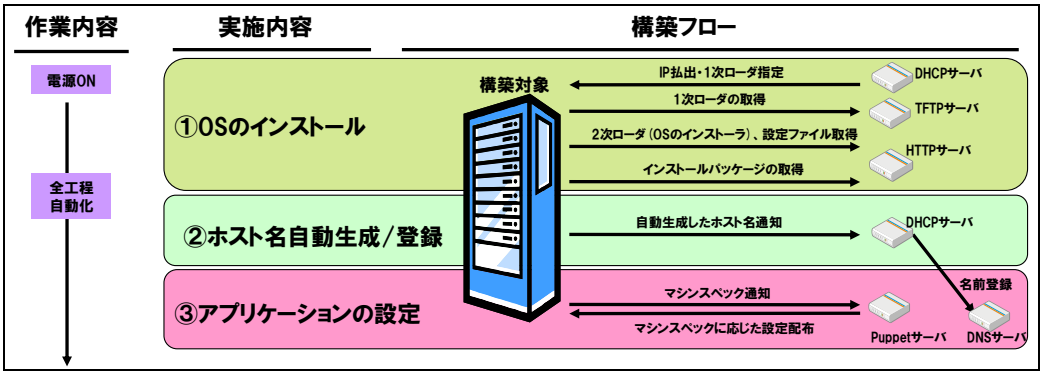
上記各サービスのバージョン及びサービスを提供するサーバを以下に一覧化する。

表 12-15 自動インストールサーバのソフトウェア構成と提供機能

No.	サービス	製品	製品形態	提供サーバ
1	DNS サーバ	BIND	OSS	維持管理サーバ
2	DHCP サーバ	dhcpd	OSS	維持管理サーバ
3	TFTP サーバ	tftpd	OSS	維持管理サーバ
4	HTTP サーバ	Apache	OSS	維持管理サーバ
5	Puppet サーバ	Puppet	OSS	維持管理サーバ

インストールフローは図 12-3 の通りとなる。

図 12-3 Hadoop スレーブサーバ自動構築



12.4.9 Kickstart によるインストールと Puppet による維持管理の役割分担

本節では Kickstart ファイルの詳細手順の設計方針を記載する。OS のインストールは Kickstart で実施し、インストール後の構成管理は Puppet で実施を行う。Kickstart で行うべき項目と、Puppet で実施すべき項目を明確化し、これらの基本方針を策定する。

12.4.9.1 Kickstart と Puppet の役割分担に関する基本方針

Kickstart は OS のインストール時に一度しか実行することがないのに対し、Puppet は随時実行することが可能である。したがって構成変更を柔軟に行うため、以下の基本方針を策定した。

- Puppet が動作できる最低限度の設定を Kickstart で実施する。

上記基本方針に基づき、Kickstart で行うべき項目を以下のように設定した。

表 12-16 Kickstart で実施すべき項目

No.	大項目	小項目	概要
1	ディスクレイアウト	フォーマット	サーバのディスクレイアウトを決定し、フォーマットを行う
2	インストール	OS	OS のインストール
3		レポジトリの追加	Hadoop や、Puppet などデフォルトの OS パッケージでないパッケージのレポジトリの登録

No.	大項目	小項目	概要
4		追加ソフトウェア	上記レポジトリに登録されたソフトウェアのインストールを実施する。
5	ホスト名	ホスト名の決定	ホスト名を自動構成する
6		ホスト名の登録	自動構成されたホストを DNS に登録する
7	Puppet 事前準備	時刻同期	Puppet 通信は時刻同期が必要なため
8		Puppet 設定	Puppet 通信ができる設定を実施
9		初期同期	Puppet による設定ファイルを同期する

12.4.10 Puppet サーバの提供する設定ファイルの雛形に関する詳細検討

本検証では構成管理サービスである Puppet を利用して設定を実施する方式の詳細結果を記載する。表 12-7 にある問題 2-1 の具体的な実装について記載する。

Puppet では、設定ファイルの雛形を用意して、それをノードごとに適用するという方式をとるため、設定ファイルの雛形を定義する必要がある。また、設定ファイルをひとまとめにしてモジュールとして提供する必要がある。本検証において作成した Puppet のモジュールとユーザ定義型を以下に列挙する。

表 12-17 実装した Puppet モジュールとユーザ定義型

No.	モジュール名	ユーザ定義型	概要
1	common	common	OS に関連するファイル群を定義する。不要なサービスの停止や NTP による時刻同期、cron によるログローテーションなど、OS レベルでの設定をひとまとめにしたモジュールを提供する。また、サーバ機器が不均一なことに起因するクライアント固有の facter モジュールは本ファイルで記載することとした。
2	hadoop	hadoop	Hadoop を動作させるために必要な設定ファイル・ディレクトリ群を一まとめにしたモジュールを提供する。引数として namenode/jobtracker などを指定できるように構成し、汎用的なつくりを心がけた。
3	ganglia	gmond	gmond を起動させるために必要な設定を一まとめにしたモジュールである。また、デフォルトで可視化できない項目について追加で可視化するための設定ファイル群を定義する。詳細は可視化項目の章を

No.	モジュール名	ユーザ定義型	概要
			参照のこと。
4		gmetad	gmetad を起動させるために必要な設定を一まとめにしたモジュールである。
5		web	ganglia におけるグラフの見え方の変更をするために利用する。

Puppet はモジュールで定義された設定ファイル群に関して、テンプレートから各ノード固有の設定ファイルを生成し、適用する。テンプレートに適用する変数は、Puppet の manifest ファイルから設定するグローバルな変数と、クライアントごとに値が決まるローカルな変数から構成される。後者のローカルな変数はデフォルトでは facter 変数として定義されているが、本検証では、デフォルトの facter 変数に加え、追加で以下のローカル変数を拡張して作成することで、スペックの異なる環境でも設定ファイルを変更することができるように構成した。

表 12-18 サーバスペックに応じた facter 拡張

No.	拡張 facter 名	概要
1	racknum	サーバがどのラックに配置されているかの値を返却する
2	diskcount	サーバに搭載されている外部記憶装置（ハードディスク）の個数を返却する
3	mygmetad	自身のサーバの metad アドレスを返却する
4	disklist	自身のサーバの外部記憶装置（ハードディスク）の一覧を返却する

12.5 結果の考察と今後の課題

本章では、自動構築の検討結果を記載するとともに残された課題を記載する。

12.5.1 Hadoop スレーブサーバの自動構築における工数の評価

100 台の Hadoop スレーブサーバに関して、50 多重でインストールを二回実施した結果を報告する。いずれの場合にも、本方式による自動構築が有効であることを確認した。

なお、インストール時間に関しては、構築対象の機器ごとに構築時間に大きな差がでた。これは、ディスクの初期化に時間がかかっていたためである。表 12-19 に二回の結果を記載する。

表 12-19 ベンダ別の構築時間

No.	構成要素	多重度	構築時間	【参考】手動構築時間	備考
1	ベンダ A	46	0.75 時間	1 台あたり 4 時間	SAS ハードディスク(72～300GBx2)
2	ベンダ B	50	1.75 時間	1 台あたり 5 時間	SATA ハードディスク(250GBx2)

以下に「表 12-19 ベンダ別の構築時間」の No2 に関する検証結果を記載する。「表 12-20 サーバごとのインストール時間一覧」 では 50 台の各サーバの構築開始時刻と終了時刻を一覧化したものである。インストール開始時刻は、一斉に電源を投入するのではなく、サーバ起動時の突発的な電圧上昇を防止するために、三分程度の時間間隔を置いてインストールを実行した。

表 12-20 サーバごとのインストール時間一覧

ホスト名	開始	終了	時間
r6-1-0-01	11:33	12:42	1:09
r6-1-0-02	11:30	12:42	1:12
r6-1-0-03	11:33	12:42	1:09
r6-1-0-04	11:30	12:42	1:12
r6-1-0-05	11:33	12:42	1:09
r6-1-0-06	11:30	12:42	1:12
r6-1-0-07	11:33	12:42	1:09
r6-1-0-08	11:30	12:42	1:12
r6-1-0-09	11:33	12:42	1:09
r6-1-0-10	11:30	12:42	1:12

ホスト名	開始	終了	時間
r7-1-0-01	11:57	13:03	1:06
r7-1-0-02	11:54	13:03	1:09
r7-1-0-03	11:57	13:03	1:06
r7-1-0-04	11:54	13:02	1:08
r7-1-0-05	11:57	13:03	1:06
r7-1-0-06	11:54	12:57	1:03
r7-1-0-07	11:57	12:57	1:00
r7-1-0-08	11:54	12:57	1:03
r7-1-0-09	11:57	12:57	1:00
r7-1-0-10	11:54	12:56	1:02

ホスト名	開始	終了	時間
r7-1-0-11	11:51	12:54	1:03
r7-1-0-12	11:48	12:50	1:02
r7-1-0-13	11:51	12:54	1:03
r7-1-0-14	11:48	12:51	1:03
r7-1-0-15	11:51	12:55	1:04
r7-1-0-16	11:48	12:52	1:04
r7-1-0-17	11:51	12:55	1:04
r7-1-0-18	11:48	12:51	1:03
r7-1-0-19	11:51	12:55	1:04
r7-1-0-20	11:48	12:52	1:04

ホスト名	開始	終了	時間
r7-2-0-01	11:45	13:06	1:21
r7-2-0-02	11:42	13:05	1:23
r7-2-0-03	11:45	13:06	1:21
r7-2-0-04	11:42	13:05	1:23
r7-2-0-05	11:45	13:05	1:20
r7-2-0-06	11:42	13:00	1:18
r7-2-0-07	11:45	13:00	1:15
r7-2-0-08	11:42	13:00	1:18
r7-2-0-09	11:45	13:00	1:15
r7-2-0-10	11:42	13:00	1:18

ホスト名	開始	終了	時間
r7-2-0-11	11:39	12:48	1:09
r7-2-0-12	11:36	12:46	1:10
r7-2-0-13	11:39	12:48	1:09
r7-2-0-14	11:36	12:45	1:09
r7-2-0-15	11:39	12:48	1:09
r7-2-0-16	11:36	12:46	1:10
r7-2-0-17	11:39	12:49	1:10
r7-2-0-18	11:36	12:45	1:09
r7-2-0-19	11:39	12:49	1:10
r7-2-0-20	11:36	12:46	1:10

上記の各サーバに関して、インストール時間の分布をグラフ化したものを示す。

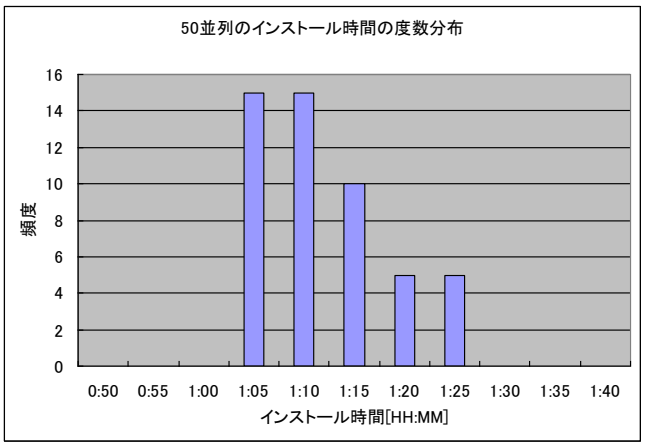


図 12-4 サーバごとの構築時間の度数分布

12.5.2 まとめと今後の課題

まとめ

本章では、クラウド型分散処理基盤における大量の Hadoop スレーブサーバの自動構築手法を検討し実装した。この手法が、Hadoop 基盤における構築のあるべき姿を満たすことを確認する。表 12-21 に表 12-2 で提示した自動構築のあるべき姿と課題を再掲する。

表 12-21 Hadoop スレーブサーバ構築のあるべき姿と課題（再掲）

No.	あるべき姿	課題
1	構築時間を大幅に短縮することができる	手順書に基づく手作業のため、時間がかかる
2	全てのサーバが同一の手順で構築されていることが保障されている	作業漏れや、作業ミスが混在しうる。またその検知が難しい。
3	故障からの回復や、増設時にも同じ手順で Hadoop スレーブサーバが構築できる	初期構築後の設定反映を漏れなく反映させるのが難しい。

Hadoop 基盤の構築に際して、Kickstart インストールと Puppet による構成管理を適切に組み合わせることによって、100 台の Hadoop スレーブサーバを 3 時間程度で構成することを確認した。したがって表 12-21 の No.1「構築時間を大幅に短縮することができる」ことを確認した。

構築の一連の手順は自動化されているために、全てのサーバが同じ手順で構成されていることが保障される。また、再構築も、ネットワークインストールの設定を行うだけであり、非常に平易に行うことができる。表 12-21 の No.2「全てのサーバが同

一の手順で構築されていることが保障されている」ことを確認するとともに、表 12-21 の No.3「故障からの回復や、増設時にも同じ手順で Hadoop スレーブサーバが構築できる」ことを確認し、上記考察により、本自動構築手法が有効であることを確認した。

また、スペックやベンダの異なる機種においても同じ手順でインストールされること、各機種のスペックに応じた意図通りの設定ファイルを配布することも確認した。これらのマシンには自動で物理的な配置を反映したホスト名が付与され、クラスタ全体で名前解決もできている。

以上考察をもって、Kickstart と Puppet を組み合わせた自動基盤構築は適切に実装できたと判断する。

今後の課題

本章では、大量に存在する Hadoop スレーブサーバの構築効率化を想定し、自動構築の対象としたが、Hadoop 基盤全体の自動構築を考えることも重要である。特に Hadoop 基盤の規模に比例して台数が増加する構成要素として、Hadoop スレーブサーバに直結したスイッチが挙げられる。本検証では 100 台の環境であったため、ネットワーク機器の構築は、現実的な時間（3 時間）におさめることができたが、1000 台の環境になるとスイッチの数は 10 倍の工数がかかると推定される。ネットワークの自動構築は今後の課題としたい。

13 Hadoop 基盤における可視化手法検討

本章では、Hadoop 基盤の動作状況を可視化する手法についての検討結果を報告する。

Hadoop 基盤もユーザ・管理者がタスクを遂行するために可視化する必要があるが、構成サーバ台数が膨大であるなどクラウド特有の課題がある。これらの課題について検討した後､抽出した課題は機能追加した可視化ソフトウェア Ganglia と JobTracker の WebUI を利用する方式で解決できることを明らかにする。

本方式は約 100 台の Hadoop スレーブサーバで構成される環境において利用可能であることを確認した。

13.1 あるべき姿と課題

本節では、Hadoop 基盤における可視化のあるべき姿とその課題について論じる。

13.1.1 可視化のあるべき姿

情報システムではユーザ・管理者はアプリケーションの実行、キャパシティプランニングなど様々なタスクを実施するが、タスクを実施した際にシステムの動作が意図通りか確認しなければならない。この目的を円滑に遂行するため、情報システムは可視化する必要がある。ここで可視化とは、システムのユーザ・管理者がタスク実施時に必要な情報をユーザインタフェース上に出力することと定義する。Hadoop 基盤においてもシステムが意図通り動作しているか確認しなければならないため、可視化は必要である。

そこで、Hadoop 基盤における可視化のあるべき姿を以下の通り設定する。

「Hadoop 基盤のユーザ・管理者がタスク実行時に必要な情報を入手できること」

13.1.2 可視化の課題

Hadoop 基盤を可視化する際の課題を 3 つとりあげる。

なお以下では、可視化される個々のサーバを可視化対象サーバ、可視化対象サーバから可視化対象情報を収集し、ユーザインタフェースへ出力するサーバを運用監視サーバと呼ぶ。

可視化するためには、(1)ユーザ・管理者がタスク実施時に必要な情報（以下可視化対象情報）を、(2)可視化対象サーバから運用監視サーバへ伝達し、(3)ユーザインタフェース上に出力する必要があるが、(1)~(3)それぞれについてクラウド上での課題を取り上げる。

(1) 可視化対象情報についての課題

従来のシステムでは、可視化対象情報は過去のノウハウの蓄積から明らかにすることができた。Hadoop 基盤でこれらの動作状況を把握するための可視化対象情報は十分整理されていない。

そこで、Hadoop 基盤の動作状況を把握するための可視化対象情報を洗い出す必要がある。

(2) 可視化対象サーバから運用監視サーバへの伝達についての課題

可視化対象情報を可視化するためには、可視化対象サーバから運用監視サーバへ可視化対象情報を伝達する必要がある。しかし Hadoop 基盤はサーバ数が膨大である上に、業務の拡大に伴い構成サーバ台数が増加するという特徴がある。

したがって可視化対象サーバ・運用監視サーバ間の通信量がボトルネックになることを回避する通信方式を採用する必要がある。

(3) ユーザインタフェースについての課題

従来の情報システムでは、その構成サーバ数は比較的少なかったため 1 台 1 台について可視化すればシステム全体の動作状況が把握できた。

しかし Hadoop 基盤ではサーバ台数が膨大なので、1 台 1 台について可視化するだけではユーザインタフェース上に出力される情報量も膨大になり、視認性が低くなるため、クラウド全体の動作状況を把握することは困難である。

そこで、クラウド全体の動作状況が把握できるユーザインタフェースを決定する必要がある。

13.2 前提条件

本節では可視化手法検討の前提条件を記載する。

13.2.1 適用されるシステム構成

以下に本章で検討対象とするシステム構成の概要を図示する。

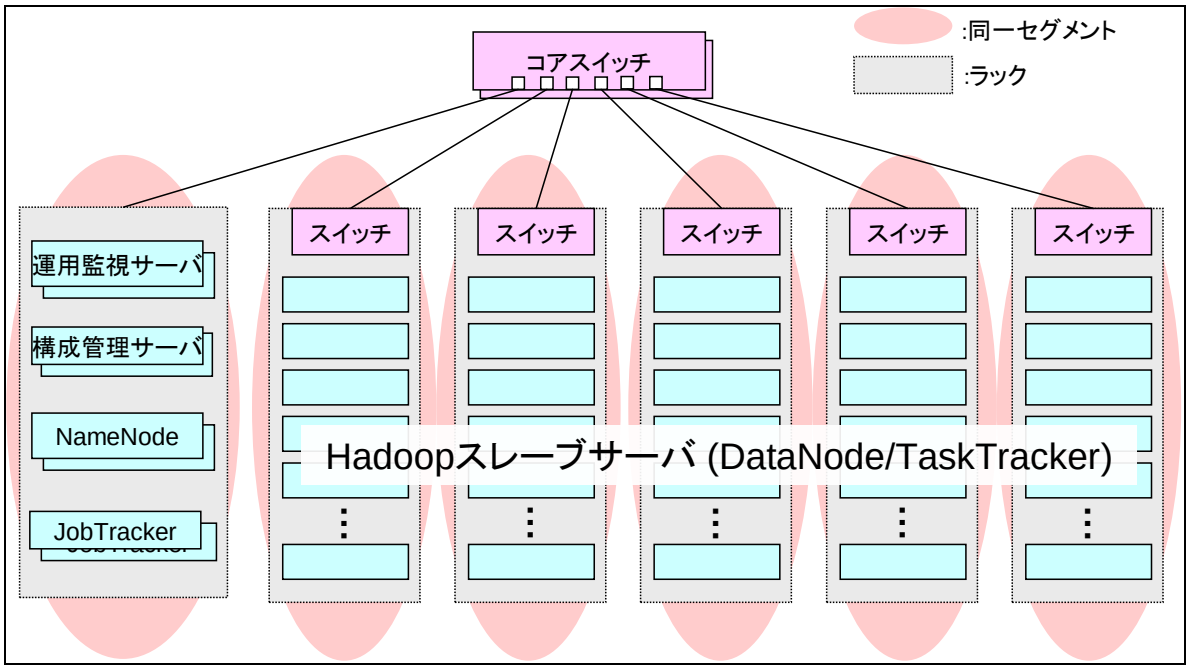


図 13-1 本章で想定するシステムの概要

図 13-1 の構成は以下の特徴がある。

- 可視化対象のリソース情報を収集し、ユーザインタフェースへ出力する運用監視サーバを設置する
- Hadoop スレーブサーバについてはラック 1 台につきスイッチを 1 台設置する
- ラックごとにセグメントは異なる

13.3 課題解決方法の検討

本節の前半では 13.1 で挙げた課題についてその解決方法を検討する。続いてそれらの解決方法を実装するのに最適な可視化ソフトウェアを決定し、選択した可視化ソフトウェアの不足機能について拡張する方法についても検討する。

最後に実装結果について評価する。

13.3.1 課題 1Hadoop 基盤における可視化対象情報の決定

可視化とは前述の通り、システムのユーザ・管理者がタスク実施時に必要な情報をユーザインタフェース上に出力することである。そのため Hadoop 基盤の動作状況を把握するための可視化対象情報を洗い出すには、システムのユーザと管理者が実施するタスクごとに、当該タスクの遂行に必要な情報を明らかにすればよい。

表 13-1 に可視化を必要とするタスクとそのタスクの遂行に必要な情報を整理した。

表 13-1 可視化を必要とするタスク

No.	実施主体	可視化を必要とするタスク	タスクの遂行に必要な情報
1	ユーザ	アプリケーションの実行	Hadoop 計算処理の進捗状況
2		デバッグ チューニング	基本リソース使用状況（アプリケーション）
3	管理者	キャパシティプランニング システムの健全性確認	基本リソース使用状況
4		増設・縮退	動作中の Hadoop スレーブサーバ数
5		回復	故障解析

以下表 13-1 のタスクの遂行に必要な情報を順に詳細化する。

13.3.1.1 Hadoop 計算処理の進捗状況把握に必要な可視化対象情報

Hadoop 計算処理の実行単位は MapReduce ジョブであり、MapReduce ジョブは実行中、完了、失敗の 3 つの状態をとる。このため、MapReduce ジョブの動作状況の概要を把握するため、実行中の MapReduce ジョブ数、完了した MapReduce ジョブ数、失敗した MapReduce ジョブ数について把握する。

また、個々の MapReduce ジョブは複数の Map タスクと Reduce タスクから構成され、これらはそれぞれ実行待ち、実行中、完了、失敗の 4 つの状態をとる。

MapReduce ジョブごとの Map 処理と Reduce 処理の進捗率を把握するため、MapReduce ジョブごとの Map タスク総数、Reduce タスク総数 、実行中の Map タスク数、Reduce タスク数および MapReduce ジョブごとの完了した Map タスク数、MapReduce ジョブごとの完了した Reduce タスク数を把握する。

また、Hadoop は処理の遅い Map・Reduce タスクについては別の Hadoop スレーブサーバでも実行する（speculative execution）ため、少数の Map・Reduce タスクが失敗しても計算処理全体への影響は軽微である。ただし失敗したタスク数が多い場合、なんらかの問題がある可能性が高いため、失敗した Map・Reduce タスク数についても把握する必要がある。

Hadoop 計算処理の進捗状況を把握するための取得リソース情報と、情報取得元を表 13-2 にまとめる。

表 13-2 Hadoop 計算処理の進捗状況を把握するための可視化対象

No.	タスク遂行に必要な作業	可視化の対象	情報取得元
1	MapReduce ジョブの計算進捗状況を把握する	実行中の MapReduce ジョブ数	JobTracker
2		完了した MapReduce ジョブ数	
3		失敗した MapReduce ジョブ数	
4		MapReduce ジョブごとの Map タスク総数	
5		MapReduce ジョブごとの Reduce タスク総数	
6		MapReduce ジョブごとの実行中 Map タスク数	
7		MapReduce ジョブごとの実行中 Reduce タスク数	
8		MapReduce ジョブごとの完了した Map タスク数	
9		MapReduce ジョブごとの完了した Reduce タスク数	
10		MapReduce ジョブごとの失敗した Map タスク数	
11		MapReduce ジョブごとの失敗した Reduce タスク数	

13.3.1.2 基本リソース使用状況把握に必要な可視化対象情報

以下基本リソース情報を、①以下情報システム一般に必要なリソース情報、②Hadoop 固有のリソース情報に分類し、それぞれの可視化対象情報について論じる。

情報システム一般に必要なリソース情報

サーバの主要構成要素は CPU・メモリ・ディスク・ネットワークである。そのため、チューニング時にもこれらの構成要素の利用状況を把握する必要がある。

各種サーバの CPU リソース情報に関しては、ロードアベレージ、CPU 使用率

（system、user、iowait）、割り込み不可能なスリープ状態にあるプロセス数を取得する。ロードアベレージから負荷の大小を把握できる。また負荷が大きい場合、system、user が iowait に比べ高い割合を占めていたならば CPU ネック、逆の場合は I/O ネックであると判断できる。さらにディスク I/O ネックの場合、その程度を確認するため割り込み不可能なスリープ状態にあるプロセス数を取得する。

各種サーバのメモリリソースに関しては、swap-in 発生回数、swap-out 発生回数、物理メモリ使用の内訳（Used、Cached、Buffered、Swapped）を取得する。これにより、メモリが不足しているか判断できる。さらに、物理メモリ使用状況から、スワップが発生した時点のメモリ使用状況を把握することができる。

ディスクについてはディスク I/O の滞留状況を把握するために、ディスクデバイスのキューに滞留しているリクエストの平均サイズを取得する。

各種サーバの NIC については、単位時間当たりに受信した情報量、単位時間当たりに送信した情報量を可視化する。これにより、NIC の使用状況が把握できる。

基本リソース情報のうち、情報システム一般に必要なリソース情報を表 13-3 にまとめる。

表 13-3 情報システム一般に必要なリソースについての可視化対象

No.	タスク遂行に必要な作業	取得リソース情報	情報取得元
1	各種サーバの CPU リソースの使用状況を把握する	ロードアベレージ CPU 使用率（system、user、iowait） 割り込み不可能なスリープ状態にあるプロセス数	NameNode JobTracker Hadoop スレーブサーバ MapReduce ジョブ Client 構成管理 サーバ 運用監視 サーバ
2	各種サーバのメモリリソースの使用状況を把握する	物理メモリ使用率（Used、Cached、Buffered、Swapped） swap-in 発生回数 swap-out 発生回数	NameNode JobTracker Hadoop スレーブサーバ MapReduce ジョブ Client 構成管理 サーバ

No.	タスク遂行に必要な作業	取得リソース情報	情報取得元
			運用監視 サーバ
3	ディスクの滞留状況を把握する	デバイスのキューに滞留しているリクエストの平均サイズ	NameNode JobTracker Hadoop スレーブサーバ MapReduce ジョブ Client 構成管理 サーバ 運用監視 サーバ
4	各サーバのネットワーク使用状況を把握する	各サーバのネットワーク使用量（bytes received） 各サーバのネットワーク使用量（bytes sent）	NameNode JobTracker Hadoop スレーブサーバ MapReduce ジョブ Client 構成管理 サーバ 運用監視 サーバ
5	ネットワーク装置の利用状況を把握する	ラック外からスイッチへの入力パケット量 スイッチからラック外への出力パケット数	スイッチ

Hadoop 固有のリソース情報

管理系サーバである NameNode、JobTracker では、JVM のメモリ利用状況が適切か、メモリリークの兆候があるか判断するために、ヒープサイズと FullGC の実行頻度について可視化する。

また HDFS については、十分な空き容量がない場合 HDFS は正常に動作しないため、空き容量を確認する必要がある。

HDFS はブロックが破損した場合、他サーバに存在するレプリカで自動修復する。

破損したブロックがないか、また修復が正常に実施できているかを確認するために、以下の項目についてリソース情報を取得する。

また、Hadoop 基盤の構成ではスイッチがボトルネックになりうるため、スイッチの負荷状況を把握する必要がある（図 13-1）。

スイッチの負荷状況を把握するために、各ラックのスイッチとコアスイッチ間の入力通信量、出力通信量を可視化する（図 13-2）。

特定のスイッチの入力通信量、出力通信量が他のネットワーク機器に比べて高い場合、通信量がボトルネックになっていることがわかる。

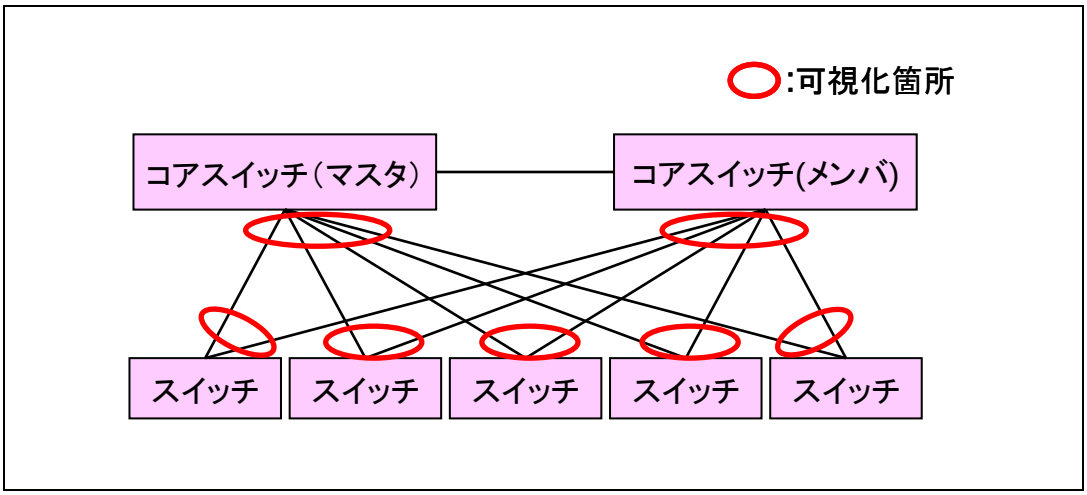


図 13-2 スイッチ可視化個所

基本リソース情報のうち、Hadoop 固有のリソース情報を表 13-4 にまとめる。

表 13-4 Hadoop 固有のリソースについての可視化対象

No.	タスク遂行に必要な作業	取得リソース情報	情報取得元
1	JVM のメモリ使用状況が適切か把握する	・Heap 領域（New） ・Heap 領域（Old） ・Heap 領域（Permanent） ・FullGC の実行頻度	NameNode JobTracker
2	HDFS に十分な空き容量があるか確認する	・HDFS の利用率	NameNode
3	HDFS のレプリケーションが正常に動作しているか確認する	・UnderReplicatedBlocks (指定したレプリケーション数を下回っているブロック)	NameNode
4		MissingBlocks (レプリカが 1 つしかなく、唯一のブロックが破損すると修復不可能になるブロック数)	NameNode
5		CorruptBlocks (全てのレプリカが破損し、修復不可能なブロック数)	NameNode

13.3.1.3 動作中の Hadoop スレーブサーバ台数の確認

Hadoop スレーブサーバを増設・縮退した場合、動作中の Hadoop スレーブサーバ台数を確認する必要がある。

表 13-5 増設・縮退した Hadoop スレーブサーバの動作確認

No.	タスク遂行に必要な作業	可視化の対象	情報取得元
1	動作中の Hadoop スレーブサーバ台数	動作中の Hadoop スレーブサーバ台数	Hadoop スレーブサーバ

13.3.1.4 故障解析に必要な可視化対象情報

故障解析時には①MapReduce ジョブ・Map・Reduce の失敗した数、②サーバの構成要素ごとの基本リソース情報を取得する必要がある。①については表 13-2、②については表 13-3~ 表 13-4 で同様項目について可視化する必要がある。

13.3.1.5 課題 1 Hadoop 基盤における可視化対象情報の決定の結論

以上の検討により、Hadoop 基盤の動作状況を把握するためには、表 13-2 から表 13-5 の各項目について可視化する必要があるといえる。

13.3.2 課題 2 ボトルネックを回避する通信方式決定

本節では、課題 1 で洗い出した可視化対象情報を可視化対象サーバから運用監視サーバへ伝達する際の通信方式の要件について検討する。

13.3.2.1 方式の検討

Hadoop 基盤はサーバ数が膨大である上に業務の拡大に伴い構成サーバ台数が増加するという特徴がある。

このため、運用監視サーバ・可視化対象サーバ間の通信方式は、可視化対象サーバ台数についてスケーラブルな方式であることが望ましい。

この点、単純に個々の可視化対象サーバから運用監視サーバへ可視化対象情報を伝達する方式は可視化対象サーバ数の増加に応じて通信量が線形に増加するため、スケーラブルとはいえない。

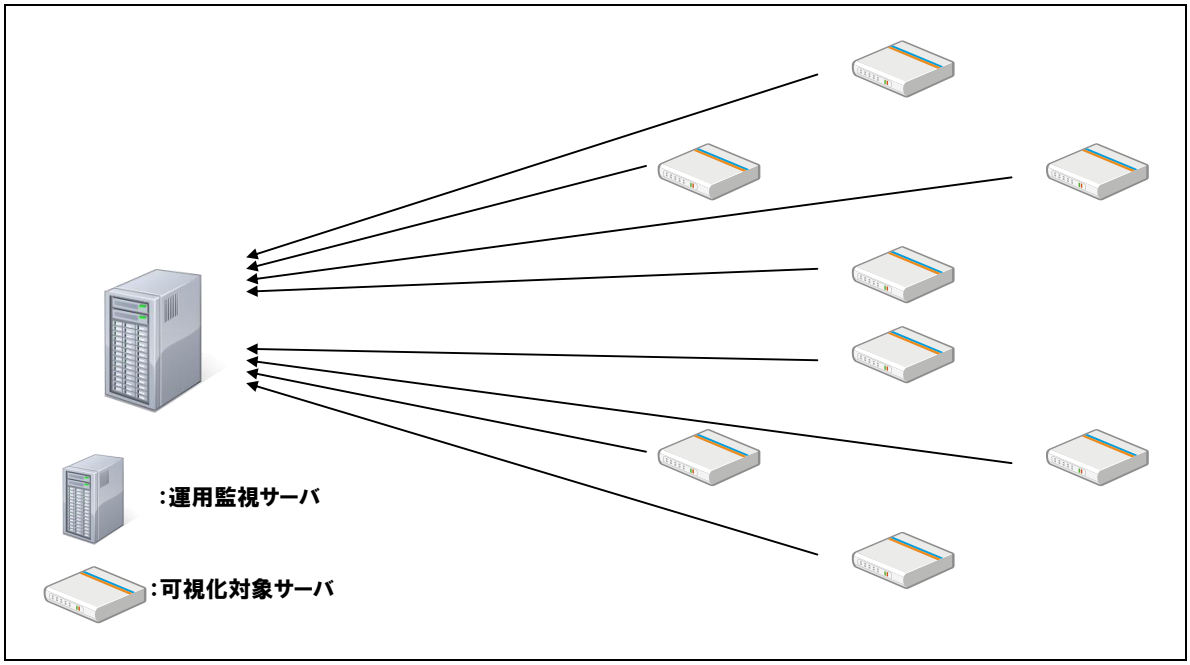


図 13-3 サーバ台数に比例して通信量が増加する通信方式の例

そこで、何らかの形で可視化対象情報を集約し、運用監視サーバへの通信量が線形に増加しないよう工夫する必要がある。

具体的には、以下の 2 方式が考えられる。

運用監視サーバの階層化

この方式では、運用監視サーバを複数台設置し、階層構造化する。全ての可視化対象情報が収集される運用監視サーバは、下位にある少数の運用監視サーバとのみ通信すればよいため、可視化対象サーバ数の増加に応じて通信量が線形に増加しない。このため本方式はスケーラブルといえる。

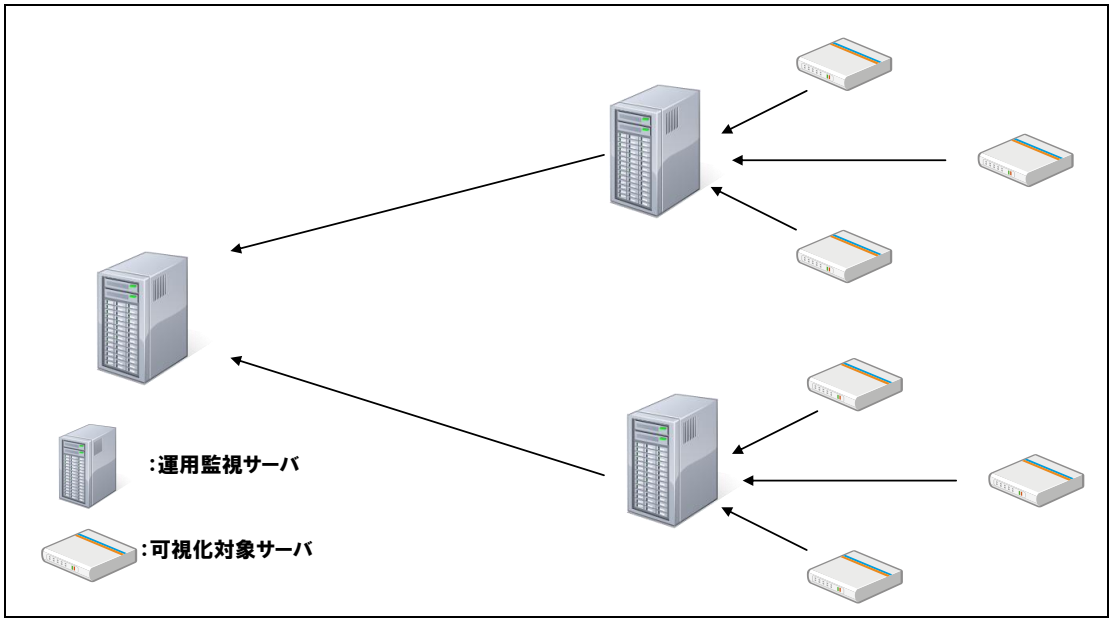


図 13-4 運用監視サーバ階層化のイメージ

可視化対象サーバ間での可視化対象情報共有

この方式では、可視化対象サーバをいくつかのグループに分け、グループ内で可視化対象情報を共有する。運用監視サーバは、1 グループにつき 1 台の可視化対象サーバとのみ通信すれば当該グループの全ての可視化対象サーバの可視化対象情報を入手すればよいため、可視化対象サーバ数の増加に応じて通信量が線形に増加しない。このため本方式もスケーラブルといえる。

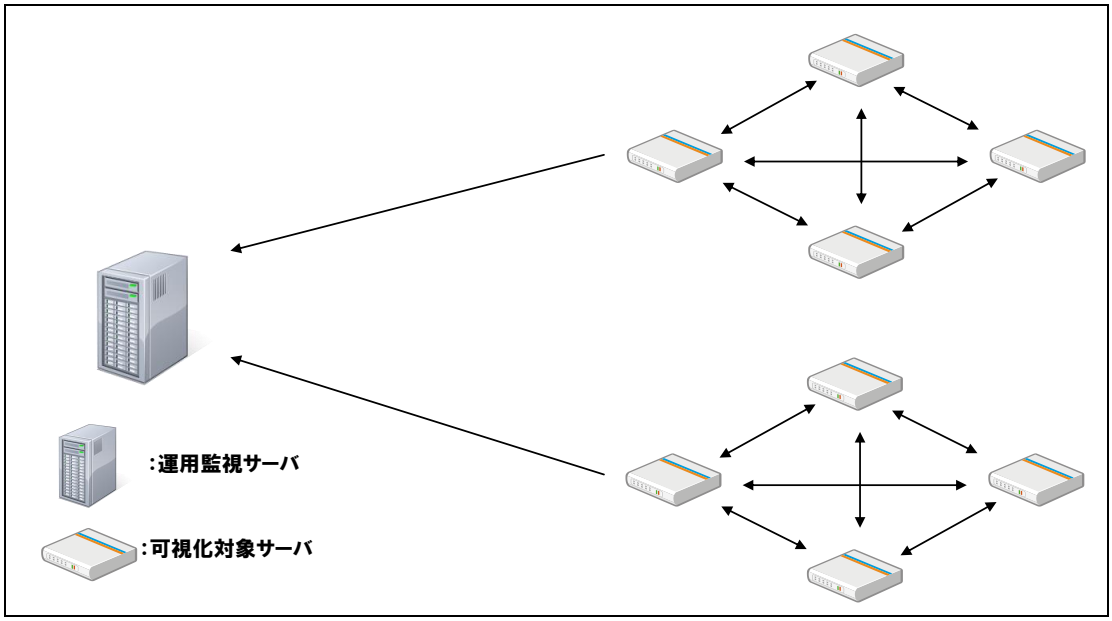


図 13-5 可視化対象サーバ間での可視化対象情報共有のイメージ

13.3.2.2 課題 2 ボトルネックを回避する通信方式決定の結論

以上の検討より、ボトルネックを回避する運用監視サーバ・可視化対象サーバ間の通信方式は、運用監視サーバの階層構造化が可能であること、ないし可視化対象サーバ間で可視化対象情報を共有できることを要件とする。

13.3.3 課題 3 クラウド全体の動作状況が把握できるユーザインタフェースの決定

本節では Hadoop を構成するサーバの動作状況が把握できるユーザインタフェースの要件を導出し、可視化の定義における『高い視認性を持つユーザインタフェース』を具体化する。台数の多寡から、Hadoop スレーブサーバとそれ以外のサーバに分けて議論する。

13.3.3.1 Hadoop スレーブサーバについてのユーザインタフェース

集計の必要性

Hadoop 基盤環境は Hadoop スレーブサーバ数が群を抜いて多く、また増設する可能性がある。このため、単純に全ての Hadoop スレーブサーバについて 1 台ずつ可視化するだけでは、大量のグラフが出力され、Hadoop 全体の状態を把握することが困難になる。

そこで、Hadoop スレーブサーバについては個別にサーバの状態を可視化することに加え、Hadoop スレーブサーバ全体について課題 1 で検討した可視化対象情報の平均や合計など統計情報を集計し可視化することとする。

ただし、Hadoop スレーブサーバ全体について可視化しただけでは、スイッチ故障時に当該スイッチに接続されているサーバがネットワーク的に孤立した場合などに、単に数十台のサーバがダウンしているように見え、その原因の究明が難しくなる。

そこで、共通の特徴を持つ Hadoop スレーブサーバについて、Hadoop スレーブサーバ全体より小規模な Hadoop スレーブサーバのグループを作成し、そのグループ当たりの統計情報も集計し可視化することとする（図 13-6)。

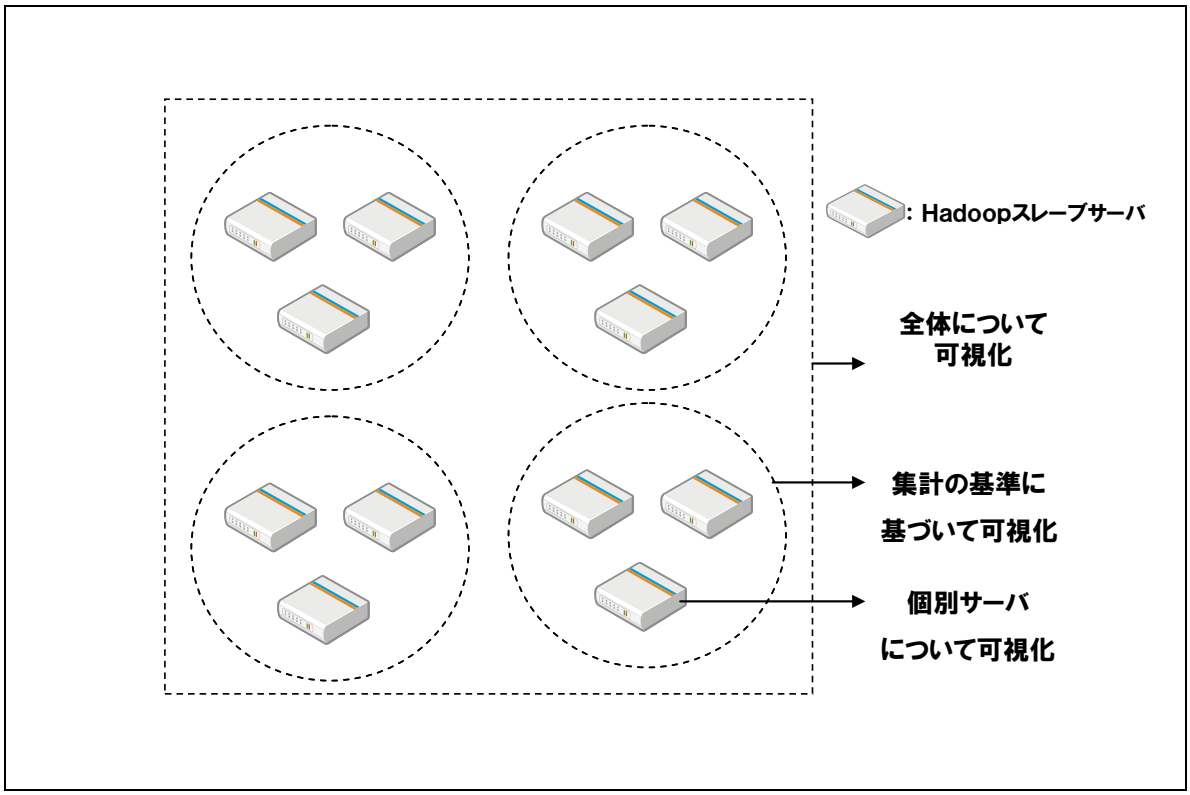


図 13-6 集計の概念図

集計の基準

次にグループを作成する際の基準を検討する（図 13-6 破線の円）。

Hadoop はラックを 1 つの単位として動作する。また 13.2 で述べたようにラックごとにセグメントが異なる。このためネットワークトポロジという共通の特徴に基づいて集計することが考えられる。

また Hadoop の計算処理は MapReduce ジョブ単位で実行される。このため、同一 MapReduce ジョブの実行時に利用された Hadoop スレーブサーバという特徴でグループを作成することが考えられる。このグルーピングは、MapReduce ジョブは全ての Hadoop スレーブサーバを利用するため、時間に基づいて Hadoop スレーブサーバをグルーピングすることと同義になる。

そこでネットワークトポロジと MapReduce ジョブ単位の 2 つを集計の基準として利用することにする。

13.3.3.2 Hadoop スレーブサーバ以外のサーバについてのユーザインタフェース

Hadoop スレーブサーバ以外の JobTracker、NameNode、運用監視サーバ、構成管理サーバ、Domain-0 については台数が 1～4 台と少なく、また拡張されることも通常ない（12.1.1）。そこで、これらのサーバについては、従来通り 1 台ずつ可視化すれ

ばその動作状況は把握できる。

13.3.3.3 課題 3 の結論

以上の検討を表 13-6 にまとめる。

表 13-6 クラウド全体の動作状況が把握できるユーザインタフェースの要件

No.	ユーザインタフェースの要件
1	Hadoop スレーブサーバのグループについての集計情報を表示できること
2	Hadoop スレーブサーバを①ネットワークトポロジ、ないし②MapReduce ジョブ単位に基づいてグルーピングできること

13.3.4 基本となる可視化ソフトウェアの決定と機能追加の検討

以下 13.3.1 から 13.3.3 で検討した課題解決方法を実装する際の基本となる可視化ソフトウェアに求められる要件を検討する。

13.3.4.1 可視化ソフトウェアの要件

「課題 1Hadoop 基盤における可視化対象情報の決定」の検討結果から、表 13-2 から表 13-5 の各項目について可視化する必要がある。

「課題 2 ボトルネックを回避する通信方式決定」の検討結果から、運用監視サーバの階層構造化が可能であること、ないし可視化対象サーバ間で可視化対象情報を共有できる必要がある。

「課題 3 クラウド全体の動作状況が把握できるユーザインタフェースの決定」の検討結果から、Hadoop スレーブサーバを①ネットワークトポロジ、ないし② MapReduce ジョブ単位に基づいてグルーピングできること、及び、Hadoop スレーブサーバのグループについての集計情報を表示できることの 2 つが要件となる。

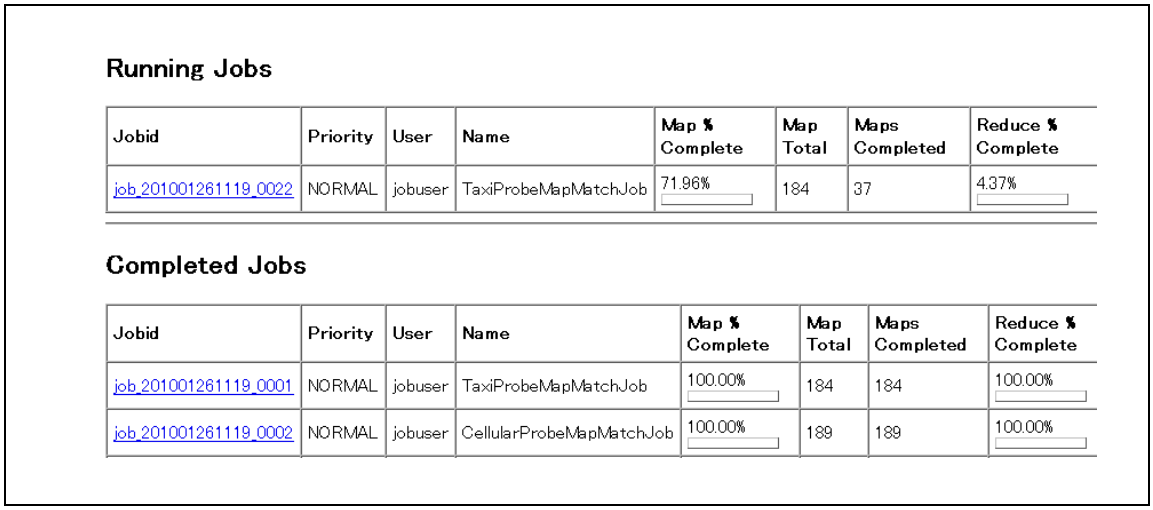
以上の要件を表 13-7 にまとめる。

表 13-7 可視化ソフトウェア選定の要件

No.	要件
1	表 13-2 から表 13-5 の各項目について可視化できること
2	運用監視サーバの階層構造化が可能であること、ないし可視化対象サーバ間で可視化対象情報を共有できること
3	Hadoop スレーブサーバを①ネットワークトポロジ、ないし②MapReduce ジョブ単位に基づいてグルーピングできること

No.	要件
4	Hadoop スレーブサーバのグループについての集計情報を表示できること

なお、計算処理の進捗状況（表 13-2）については、図 13-7 の通り JobTracker の WebUI から可視化できる。JobTracker の WebUI の情報は、初期状態で各 Hadoop スレーブサーバから収集されるため、可視化対象情報の伝達方法に関する要件である No.2 の要件については問題にならない。No.3 の要件については MapReduce ジョブ単位に基づいて可視化できている。No.4 の要件については MapReduce ジョブ単位での集計情報を表示できる。



Running Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete
job_201001261119_0022	NORMAL	jobuser	TaxiProbeMapMatchJob	71.96%	184	37	4.37%

Completed Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete
job_201001261119_0001	NORMAL	jobuser	TaxiProbeMapMatchJob	100.00%	184	184	100.00%
job_201001261119_0002	NORMAL	jobuser	CellularProbeMapMatchJob	100.00%	189	189	100.00%

図 13-7JobTracker の WebUI の出力例

このため以下では基本リソース使用状況を把握するための可視化対象情報(表 13-3 から表 13-4)、Hadoop スレーブサーバの台数（表 13-5）を可視化するためのソフトウェアについて検討する。

13.3.4.2 可視化ソフトウェアの比較

以下、有力なオープンソースの可視化ソフトウェアである Ganglia、Munin、Cacti について、表 13-7 の要件 1 から 4 の基準で比較する。

なお、以降の検討では、可視化対象サーバ上で可視化対象情報を収集するデーモンをエージェント、各エージェントの収集した可視化対象情報を集約するデーモンをマネージャと呼ぶ。

要件 1 については、Ganglia、Munin、Cacti のいずれにおいても、任意の可視化対象情報を追加できる。

要件 2 については、Ganglia のエージェントは、マルチキャスト通信を利用して同

一ネットワーク上にある他の Ganglia エージェントに自身が収集した情報を送信することができる。このため、どのエージェントも当該ネットワーク上にある全てのサーバのリソース情報を保持しているため、マネージャは 1 ネットワークにつき 1 つのエージェントと通信すればよい。したがって Ganglia は、可視化対象サーバ間で可視化対象情報を共有できる。

要件 3 については、Ganglia、Munin、Cacti のいずれにおいても、サーバをネットワークトポロジに基づいてグルーピングできる。一方 MapReduce ジョブ単位でグルーピングすることは、新しい MapReduce ジョブを実行する度に動的に新しいグループを作成し、これは Ganglia、Munin、Cacti のいずれにおいてもできない。

要件 4 については、Ganglia はグループごとに集計情報を表示することができる。一方 Cacti、Munin ではグループ化は可能なものの、その集計情報を表示することはできない。

以上の検討を表 13-8 にまとめる。

表 13-8 可視化ソフトウェアの比較

No.	要件	Ganglia	Munin	Cacti
1	表 13-2 から表 13-5 の各項目について可視化できること	○ 任意の可視化対象情報を追加できる	○ 任意の可視化対象情報を追加できる	○ 任意の可視化対象情報を追加できる
2	運用監視サーバの階層構造化が可能であること、ないし可視化対象サーバ間で可視化対象情報を共有できること	○ マルチキャストを利用して可視化対象情報を共有できる	× 特別な仕組みはない	× 特別な仕組みはない
3	サーバを①ネットワークトポロジ、ないし②MapReduce ジョブ単位に基づいてグルーピングできること	△ ①は可能 ②は不可	△ ①は可能 ②は不可	△ ①は可能 ②は不可
4	サーバのグループについて集計情報を表示できること	○ 可能	× 不可	× 不可

この結果から、最も多くの要件を満たす Ganglia を利用することとする。

13.3.4.3 Ganglia の概要

以下に Ganglia の概要を示す。

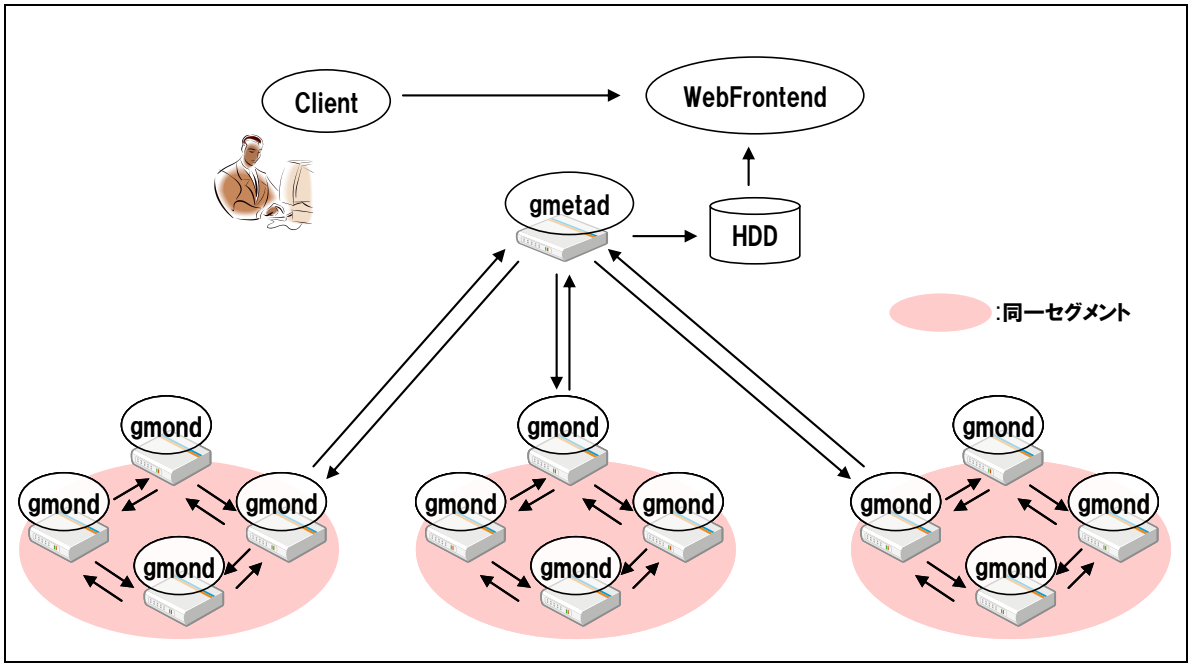


図 13-8Ganglia 概要

図 13-8 の Ganglia の構成要素の基本的な動作について解説する。

- gmond … エージェント上に常駐し、各種リソース情報を収集する。マルチキャストが到達する範囲で他エージェントが収集したリソース情報を共有する。
- gmetad … 指定した gmond の持つリソース情報を収集し、ディスクに保存する。
- WebFrontend … gmetad がディスクに保存したリソース情報を可視化する。

13.3.4.4 Ganglia の可視化対象情報追加方法

Ganglia は初期状態で、ロードアベレージ・CPU 使用状況・メモリリソース利用状況・ネットワーク利用状況について可視化しているが、これだけでは課題 1 で洗い出した可視化対象情報は網羅できない。そのため、可視化対象情報の追加方法について検討する。

Ganglia への可視化対象情報の追加は以下の 2 つの方法で実施できる。

- Hadoop Ganglia メトリクス
- gmetric

以下順に説明する。

Hadoop Ganglia メトリクス

Hadoop Ganglia メトリクスは MapReduce 処理、HDFS の状況など Hadoop 基盤で必要な可視化対象情報を取得するためメトリクスである。これを適用するためには、Hadoop Ganglia メトリクスのパッチ(https://issues.apache.org/jira/browse/HADOOP-4675)を適用した Hadoop をインストールすればよい。

gmetric

gmetric とは ganglia-gmond rpm パッケージをインストールすることで利用できるコマンドであり、このコマンドの引数に取得したいリソース名と、リソース情報の値を戻り値として返すコマンドないしスクリプトを指定し実行すれば、gmetric コマンドを実行したノードの gmond を含めた同一ネットワーク上の全ての gmond がこの結果を受け取ることができる（図 13-9)。

可視化対象情報の値を返すスクリプトを作成し、gmetric の引数に指定すれば任意の可視化対象情報を追加できる。定期的にリソース情報を取得するためには、cron に登録すればよい。

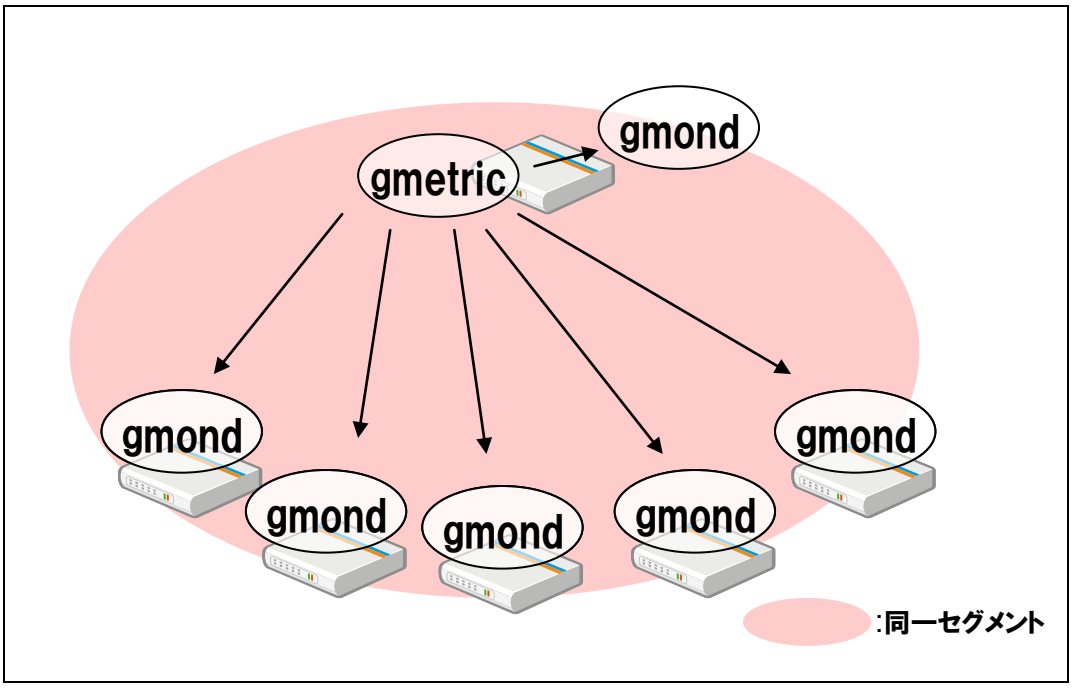


図 13-9 gmond の概念図

13.3.4.5 Ganglia のユーザインタフェース変更

13.3.4.4 の方法で可視化対象情報を追加した場合、1 つのグラフ上には単一のリソース情報についてのみ可視化される。可視化対象情報によっては複数のリソース情報を同一のグラフ上に可視化した方がシステム動作状況を把握しやすくなるため、その

方法について検討する。

オリジナルのグラフ出力定義

Ganglia で複数リソース情報を同一グラフ上に出力するためには、グラフ出力を php ファイルで定義したものを conf.php ファイルの$optional_graphs で指定する。したがって、相互に関連するリソース情報を同一グラフ上に出力するよう定義した php ファイルを作成し、conf.php から読み出せばよい。

図 13-10 に独自に定義したグラフの例として、ラック外からスイッチへの入力パケット量を可視化したグラフを掲載する。

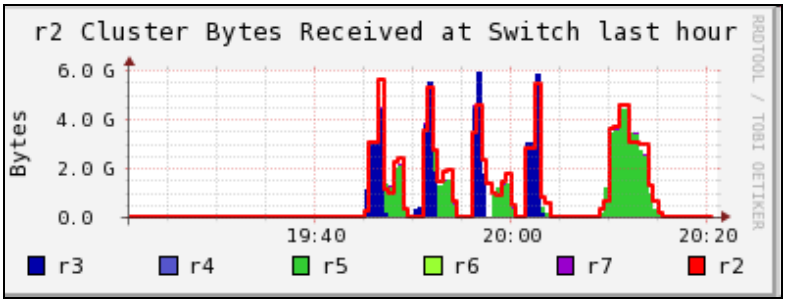


図 13-10 オリジナルグラフの例

システム全体・サーバ単体のビューでのオリジナルグラフ出力方法

上の手順でオリジナルのグラフ出力を定義した場合、システム全体のビューとサーバごとのビューではグラフは出力されない。システム全体のビューで当該グラフを出力するためには host_extra.tpl に、サーバ単体のビューで当該グラフを出力するためには、host_extra.tpl にオリジナルのグラフ出力名を指定する必要がある。

オリジナルのグラフ作成項目

本章の環境では、表 13-9 の項目についてオリジナルのグラフを作成する。

表 13-9 オリジナルグラフ作成項目

No.	作成グラフ名	作成元データ	追加するビュー
1	コンテキストスイッチ回数の折れ線グラフ	コンテキストスイッチ回数	システム全体
2	待ちプロセス数の折れ線グラフ	待ちプロセス数	システム全体
3	Heap の各領域の積み上げ	Heap 領域（New）	システム全体

No.	作成グラフ名	作成元データ	追加するビュー
	グラフ	Heap 領域（Old） Heap 領域（Permanent）	
4	swap-in、out の発生回数 両方の折れ線グラフ	swap-in 発生回数 swap-out 発生回数	システム全体 サーバ単体
5	ラック外からの各スイッチへの入力パケット量の積み上げグラフ	ラック外からのスイッチへの入力パケット量	システム全体
6	スイッチからラック外への出力パケット量の積み上げグラフ	スイッチからラック外への出力パケット量	システム全体

13.3.5 実装結果と評価

本節では、実装結果について、①課題 1 から 3 及び Ganglia の機能拡張結果と、②性能限界の 2 点から評価する。

13.3.5.1 課題 1 から 3 及び Ganglia の機能拡張結果

Hadoop 基盤の可視化対象情報についての結果

13.3.1 の検討で洗い出した取得リソース情報について、集計の単位、集計方法、利用ソフトウェアとともに、実装できた項目を表 13-10 にまとめた。Hadoop スレーブサーバ以外から取得するリソース情報については集計を行わないので、集計の単位と集計方法は記載していない。

集計方法については、パーセントなど比率を示すものについては平均を、それ以外のものについては総和を求めることとした。

Hadoop スレーブサーバの集計（13.3.3）を実施した場合、集計の単位は Ganglia であればネットワークトポロジ、JobTracker の WebUI であれば MapReduce ジョブである。

表 13-10 可視化対象情報と実装

No.	取得リソース情報	集計の単位	集計方法	利用ソフトウェア
1	実行中の MapReduce ジョブ数	MapReduce ジョブ	総和	JobTracker の WebUI
2	完了した MapReduce ジョブ数	MapReduce ジョブ	総和	JobTracker の WebUI

No.	取得リソース情報	集計の単位	集計方法	利用ソフトウェア
3	失敗した MapReduce ジョブ数	MapReduce ジョブ	総和	JobTracker の WebUI
4	MapReduce ジョブごとの Map タスク総数	MapReduce ジョブ	総和	JobTracker の WebUI
5	MapReduce ジョブごとの Reduce タスク総数	MapReduce ジョブ	総和	JobTracker の WebUI
6	MapReduce ジョブごとの実行中 Map タスク数	MapReduce ジョブ	総和	JobTracker の WebUI
7	MapReduce ジョブごとの実行中 Reduce タスク数	MapReduce ジョブ	総和	JobTracker の WebUI
8	MapReduce ジョブごとの完了した Map タスク数	MapReduce ジョブ	総和	JobTracker の WebUI
9	MapReduce ジョブごとの完了した Reduce タスク数	MapReduce ジョブ	総和	JobTracker の WebUI
10	MapReduce ジョブごとの失敗した Map タスク数	MapReduce ジョブ	総和	JobTracker の WebUI
11	MapReduce ジョブごとの失敗した Reduce タスク数	MapReduce ジョブ	総和	JobTracker の WebUI
12	ロードアベレージ	ネットワークトポロジ	平均	Ganglia
13	CPU 使用状況（System、User、iowait）	ネットワークトポロジ	平均	Ganglia
14	割り込み不可能なスリープ状態にあるプロセス数	ネットワークトポロジ	総和	Ganglia
15	待ちプロセス数	ネットワークトポロジ	平均	Ganglia
16	コンテキストスイッチ回数	ネットワークトポロジ	総和	Ganglia
17	物理メモリ使用率（Used、Cached、Buffered、Swapped）	ネットワークトポロジ	総和	Ganglia
18	swap-in 発生回数	ネットワークトポロジ	総和	Gangli a
19	swap-out 発生回数	ネットワークトポロ	総和	Ganglia

No.	取得リソース情報	集計の単位	集計方法	利用ソフトウェア
		ジ		
20	デバイスのキューに滞留しているリクエストの平均サイズ	ネットワークトポロジ	総和	Ganglia
21	各サーバのネットワーク使用量（bytes received）	ネットワークトポロジ	総和	Ganglia
22	各サーバのネットワーク使用量（bytes sent）	ネットワークトポロジ	総和	Ganglia
23	Heap 領域（New）	—	—	Ganglia
24	Heap 領域（Old）	—	—	Ganglia
25	Heap 領域（Permanent）	—	—	Ganglia
26	FullGC の実行頻度	—	—	Ganglia
27	ラック外からスイッチへの入力パケット量	—	—	Ganglia
28	スイッチからラック外への出力パケット量	—	—	Ganglia
29	HDFS の利用率	—	—	Ganglia
30	UnderRepulicatedBlocks	—	—	Ganglia
31	MissingBlocks	—	—	Ganglia
32	CorruptBlocks	—	—	Ganglia
33	計算処理に加わったサーバ数	ネットワークトポロジ	総和	JobTracker の WebUI

以上の結果から、基本リソース情報(表 13-3~表 13-4)、Hadoop スレーブサーバの台数（表 13-5）の全ての項目について可視化といえる。

ボトルネックを回避するマネージャ・可視化対象サーバ間通信方式についての結果

Hadoop MapReduce ジョブ実行時の運用管理サーバのネットワーク通信量を図 13-11 に掲載する。

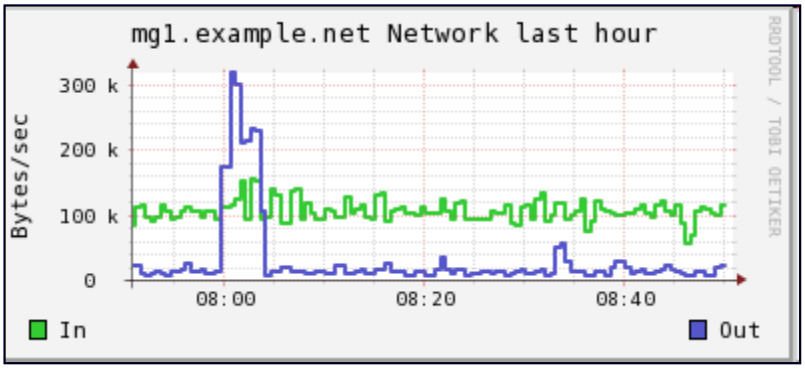


図 13-11 Hadoop MapReduce ジョブ実行時の運用管理サーバのネットワーク通信量

この結果から最大で 320KBytes/sec(2.56MBytes/sec)程度の通信量であるため、マネージャ・可視化対象サーバ間の通信はボトルネックになっていないことがわかる。

クラウド全体の動作状況が把握できるユーザインタフェースについての結果

13.3.3 で検討したユーザインタフェースについては、個別サーバに加え、Hadoop スレーブサーバ全体について集計したユーザインタフェース、およびネットワークトポロジに基づいて集計したユーザインタフェースを実装した。図 13-12 は、実装したユーザインタフェースから、ネットワークの使用状況について抜粋したものである。

また JobTracker の WebUI 上で MapReduce ジョブ単位の集計が実装済みである（図 13-7）。

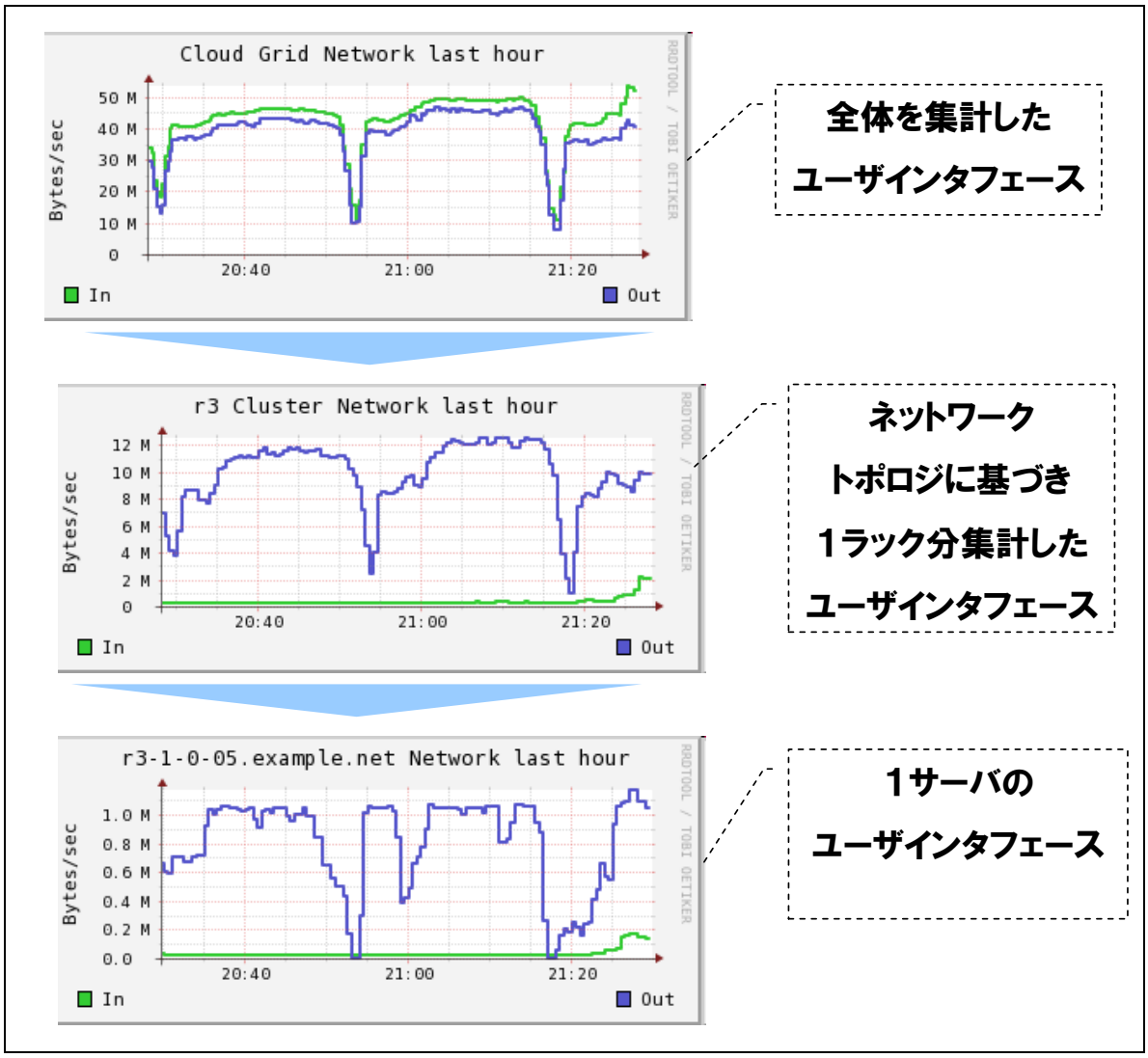


図 13-12 Hadoop 基盤向けに実装された Ganglia のユーザインタフェース

表 13-11 ユーザインタフェースの要件

No.	ユーザインタフェースの要件	実装
1	Hadoop スレーブサーバのグループについての集計情報を表示できること	○
2	Hadoop スレーブサーバを①ネットワークトポロジ、ないし②MapReduce ジョブ単位に基づいてグルーピングできること	○

Ganglia への機能追加の結果

表 13-9 にて検討した Ganglia へ追加するオリジナルグラフについては、図 13-10 の通り全て追加することができた。

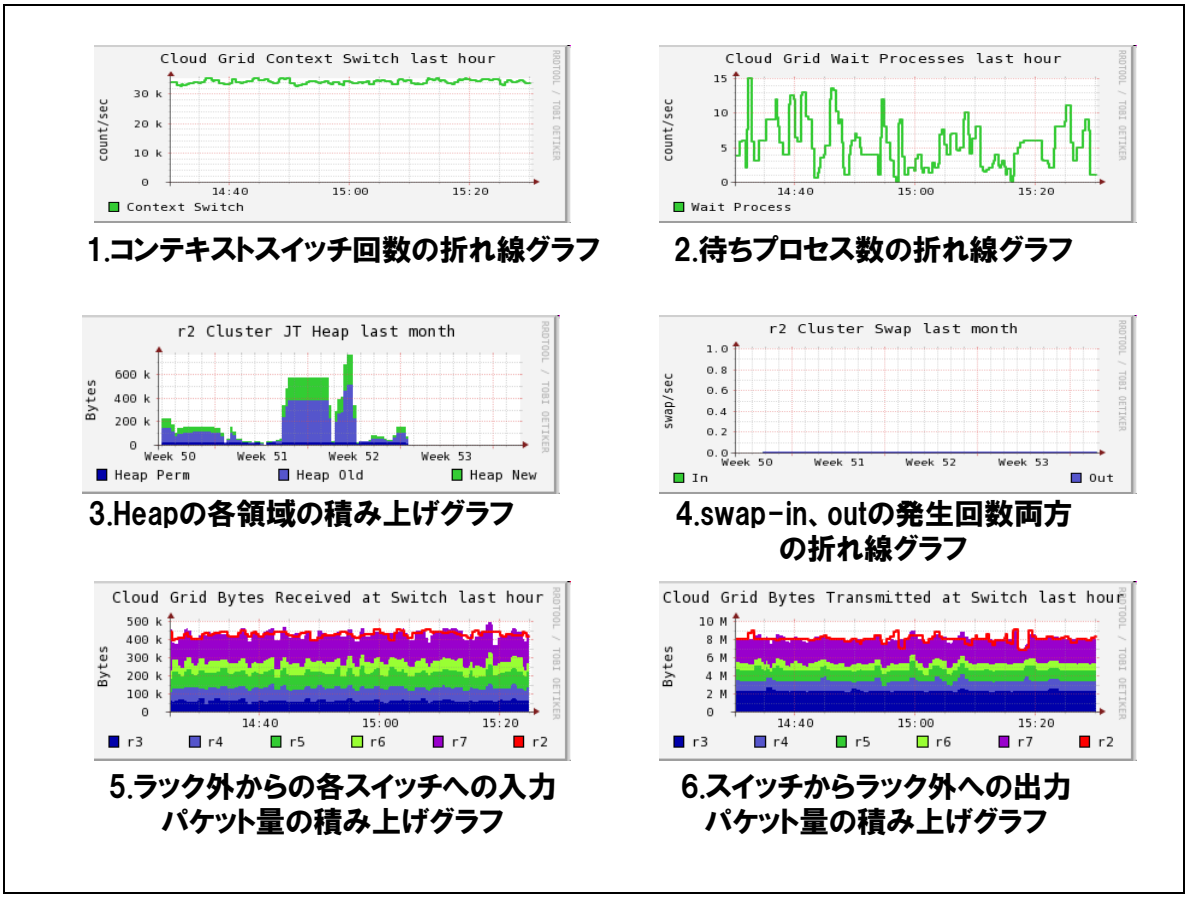


図 13-13 Ganglia へ追加したオリジナルグラフ一覧

以上より、JobTracker の WebUI 及び機能追加した Ganglia を利用して、本 Hadoop 基盤における可視化対象情報（課題 1）を収集し、マネージャ・可視化対象サーバ間の通信がボトルネックを迎えることなく（課題 2)、その情報を高い視認性を持つユーザインタフェース上に（課題 3）出力することができた。したがって、ユーザ・管理者が、タスク実行時に必要な情報を入手できるよう Hadoop 基盤を可視化できたといえる。

13.3.5.2 性能限界

Hadoop 基盤は業務規模拡大に伴いその Hadoop スレーブサーバ数が増加する可能性がある。そのため、本方式で可視化できる Hadoop スレーブサーバの上限を推定する。

100 台環境でのリソース利用状況を観察した結果、”WAIT CPU”が 25%程度と高い割合を占めていることが分かった(図 13-14)。これは、運用管理サーバ上の gmetad が gmond から取得したリソース情報をディスクに書き込む際の I/O によるものである。Hadoop スレーブサーバを増設した場合 gmond も増加することから、このディス

ク I/O がボトルネックになるおそれがある。

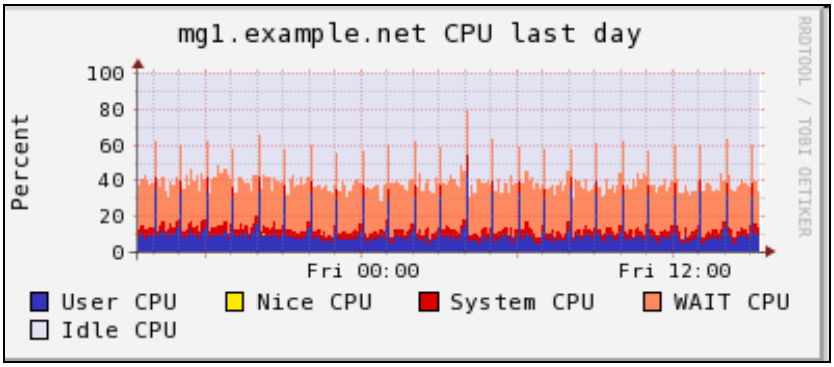


図 13-14 運用管理サーバの CPU 使用率の内訳

そこで、可視化対象サーバの台数を 0、10、26、48、66、88、96、105 台に変化させた際の運用管理サーバの CPU の利用率の内訳を計測し、性能限界に達する Hadoop スレーブサーバの台数を推定した。各サーバ台数において約 200 個のサンプルを取得し、その平均を利用した。

結果を図 13-15 及び表 13-12 に示す。

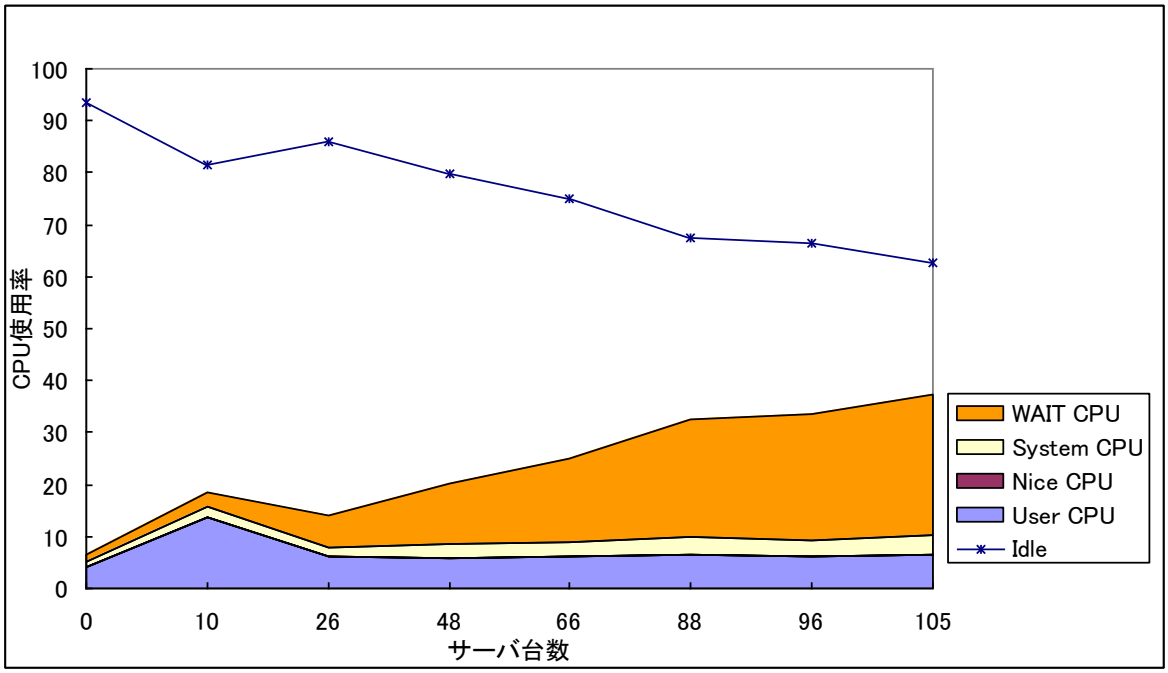


図 13-15 サーバ台数と CPU の利用率内訳

表 13-12 サーバ台数と CPU の利用率内訳

No.	サーバ台数	WAIT CPU	System CPU	Nice CPU	User CPU	Idle	サーバ 1 台当たりの WAITCPU
1	0	1.160761	1.041576	0.000054	4.151413	93.64630	-
2	10	2.679405	2.284270	0.037351	13.59621	81.40313	0.267941
3	26	6.125225	1.852921	0.000618	6.108033	85.91297	0.235586
4	48	11.54683	2.636881	0.000990	5.971237	79.84410	0.240559
5	66	15.88108	2.928797	0.000633	6.108101	75.08151	0.240622
6	88	22.68472	3.345284	0.001136	6.439261	67.52965	0.257781
7	96	24.52599	3.175769	0.001099	6.010329	66.28692	0.255479
8	105	26.91957	3.889626	0.038984	6.484171	62.66791	0.256377

この結果から、サーバ台数に比例して”WAIT CPU”のみが増加し、1 サーバ追加するごとに約 0.25%”WAIT CPU”が増加することが読み取れる。”WAIT CPU”以外に平均して 10%程度 CPU を利用していること考慮すると、90 / 0.25 = 360 台程度で CPU の Idle が枯渇し、ボトルネックを迎えると推定できる。

13.4 まとめと今後の課題

以下本章の検討結果のまとめと今後の課題について述べる。

13.4.1 まとめ

本章では、Hadoop 基盤の動作状況を可視化する手法について検討し、その実現性について検証した。

Hadoop 基盤の可視化対象情報をユースケースから洗い出した。大量のサーバから

構成される Hadoop 基盤の動作状況を可視化するために、スケーラビリティの高い可視化ソフトウェアとして Ganglia を選定した。また、洗い出した可視化対象情報のうち、Ganglia で実装されていない可視化対象情報を追加で実装した。

Hadoop 基盤の動作状況を把握できる Ganglia のユーザインタフェースの要件は、収集した可視化対象情報を Hadoop 全体、ネットワークトポロジ、サーバ単位の 3 つの単位で集計することであることを明らかにし、Ganglia 上に本要件を実装した。

今回採用した可視化手法は、約 100 台の Hadoop スレーブサーバで構成される環境においても利用可能であることを検証した。

13.4.2 今後の課題

以下今後の課題を 3 点挙げる。

13.4.2.1 集計の単位の変更

表 13-10 が示す通り、本環境では CPU 使用率など各サーバの基本リソース情報についてネットワークトポロジに基づく集計は実装できたものの、MapReduce ジョブ単位での集計は Ganglia 上に実装できていない。MapReduce ジョブを実行するユーザにとっては MapReduce ジョブ単位での集計が必要になる場合が考えられるため、集計の単位を変更できることが望ましい。

13.4.2.2 I/O ネック解消

13.3.5 にて検討した通り、本方式は gmetad のディスク I/O がボトルネックになる。Hadoop 基盤は、Hadoop スレーブサーバを追加することで性能を拡張できる点を強みとしており、本環境より多数の Hadoop スレーブサーバを配置する可能性がある。

このため gmetad のディスク I/O がボトルネックを解決する方法について検討する必要がある。

gmetad のディスク I/O がボトルネックを解決する方法には以下のものが考えられる。

- SSD など高性能なディスクの利用
- 複数ディスクの利用
- RAM ディスクへの保存

今回の環境では一般の PC にも利用されるハードディスクドライブを利用したが、高性能なディスクや SSD を利用することで、より多くの Hadoop スレーブサーバを可視化できると考えられる。また、書き込むディスクを複数設けることで I/O を分散させることも有効と考える。

RAM ディスクへの保存については、gmetad の書き込み先をハードディスクではなく、I/O 性能の高い RAM 上に行うことでボトルネックを回避する。ただし、RAM ディスクは揮発性のメディアであるため、定期的にハードディスクに保存するなどの工夫が必要になる。

本環境ではボトルネックに達しなかったため、これらの効果の比較は今後の検討課題としたい。

13.4.2.3 描画期間指定

Ganglia では描画期間を過去 1 時間、1 日、1 週間、1 か月、1 年の 5 つから出力期間を選択できる。しかし任意の 2 時点間を可視化することはできない。詳細な故障解析を実施する際には、1 時間未満の期間を可視化する必要性もあると考えられるため描画期間の指定も今後の検討課題としたい。

第３編　付録

I　実証実験環境

本章では、6 章で使用する実証実験環境の全体構成について述べる。アプリケーションと基盤の 2 側面から論じる。

I.1 渋滞解析アプリケーションの構成

本節では渋滞解析アプリケーションの実証実験環境についてまとめる。

渋滞解析アプリケーションの実証実験環境の全体像を図 I-1 に示す。

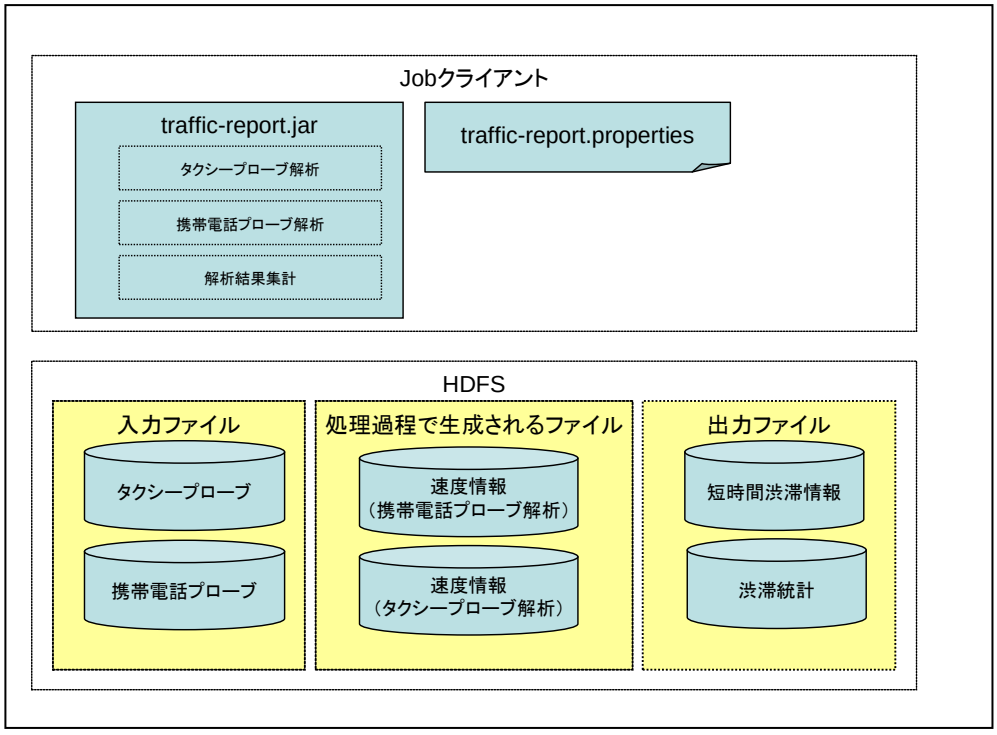


図 I-1 渋滞解析アプリケーションの実証実験における動作環境

traffic-report.jar

渋滞解析アプリケーションの実行モジュールを指す。traffic-report.properties の設定に応じて、「短時間渋滞情報生成処理」と「渋滞統計生成処理」を行う。各処理は「タクシープローブ解析」、「携帯電話プローブ解析」、「解析結果集計」の 3 つの MapReduce ジョブで構成される。各処理の詳細については 2 章を参照されたい。

本アプリケーションでは、解析対象のタクシープローブ、携帯電話プローブと、その結果の格納先をプログラム実行時に指定する。

traffic-report.properties

渋滞解析アプリケーションの各種設定を行うためのファイルを指す。渋滞解析アプリケーションの起動時に読込まれるため、渋滞解析アプリケーションと同ディレクトリに配置する。本ファイルの設定項目を表 I-1 に示す。

表 I-1 アプリケーション設定ファイルの設定項目一覧

No.	設定項目
1	タクシープローブ解析の速度情報出力パス
2	携帯電話プローブ解析の速度情報出力パス
3	道路の選択で利用する道路情報
4	道路を区切る間隔
5	携帯電話プローブの間引きの間隔
6	タクシープローブ解析の入力データの InputSplit サイズ
7	携帯電話プローブ解析の入力データの InputSplit サイズ
8	タクシープローブ解析の Reduce 処理のタスク数
9	携帯電話プローブ解析の Reduce 処理のタスク数
10	解析結果集計の Reduce 処理のタスク数

入力ファイル

渋滞解析アプリケーションの解析対象とするタクシープローブ、携帯電話プローブを指す。実証実験のシナリオに応じたタクシープローブ、携帯電話プローブを HDFS へ配置する。タクシープローブ、携帯電話プローブのファイル仕様については 2 章を参照されたい。

処理過程で生成されるファイル

渋滞解析アプリケーションの「タクシープローブ解析」、「携帯電話プローブ解析」で出力する速度情報を指す。

出力ファイル

渋滞解析アプリケーションを実効した際に出力される短時間渋滞情報ファイル、渋滞統計ファイルを指す。短時間渋滞情報ファイル、渋滞統計ファイルの仕様については 2 章を参照されたい。

I.2 クラウド型分散処理基盤の構成

本節ではクラウド型分散処理基盤の構成についてまとめる。

I.2.1 システム構成

本節では、システム構成図、構成要素、利用ソフトウェアについてまとめる。

I.2.1.1 システム構成図

クラウド型分散処理基盤のシステム構成図を図 I-2 に示す。

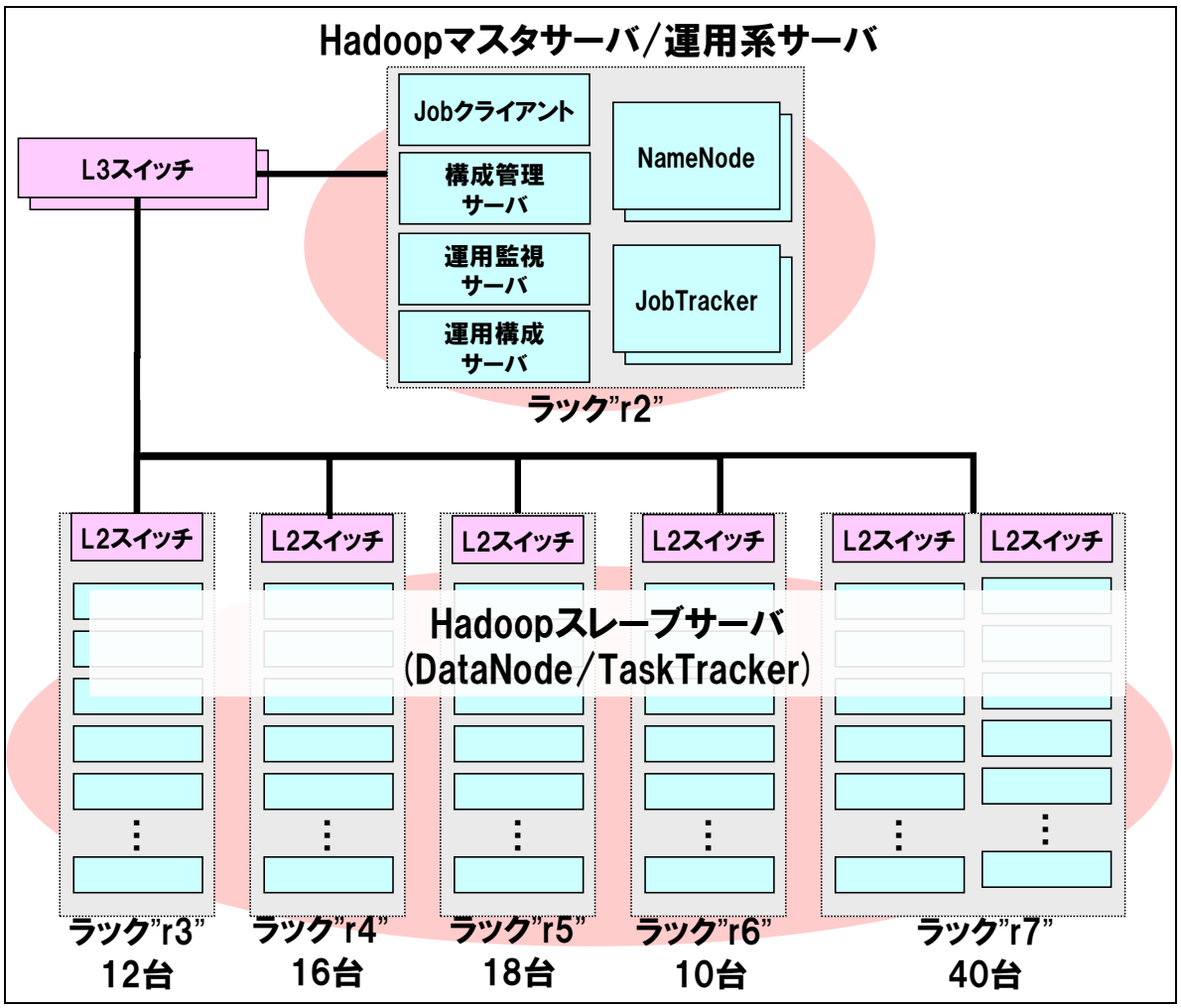


図 I-2 クラウド型分散処理基盤のシステム構成図

I.2.1.2 システムの構成要素

クラウド型分散処理基盤の構成要素を Hadoop マスタ系サーバ、Hadoop スレーブ系サーバ、運用系サーバ及びネットワーク機器の４つに分類してまとめる。

Hadoop マスタサーバ

Hadoop マスタサーバのホスト名、用途と機能及び動作機器について表 I-2 にまとめる。なお、いずれの構成要素も 1 つずつ存在し、全てラック r2 に格納する。

表 I-2 Hadoop マスタサーバ

No.	構成要素	ホスト名	用途と機能	機器
1	JobTracker	jt	JobTracker サーバ。Kemari により FT が実現されている。仮想サーバとして動作する。	仮想化されたサーバ。JobTracker 用 Kemari ホスト（正）（副）上で動作する。
2	JobTracker 用 Kemari ホスト（正）	hjt1	仮想サーバ jt を動作させるためのホストサーバ。通常時に jt が動作する。	DL380G5 QC XE5345 QuadCore/2.33GHz x2 メモリ 32GB HDD 146GB x2
3	JobTracker 用 Kemari ホスト（副）	hjt2	仮想サーバ jt を動作させるためのホストサーバ。hjt1 故障時に jt が動作する。	DL380G5 QC XE5345 QuadCore/2.33GHz x2 メモリ 32GB HDD 146GB x2
4	NameNode	nn	NameNode サーバ。Kemari により FT が実現されている。仮想サーバとして動作する。	仮想化されたサーバ。NameNode 用 Kemari ホスト（正）（副）上で動作する。
5	NameNode 用 Kemari ホスト（正）	hnn1	仮想サーバ nn を動作させるためのホストサ	DL380G5 QC XE5345

No.	構成要素	ホスト名	用途と機能	機器
			ーバ。通常時に nn が動作する。	QuadCore/2.33GHz x2 メモリ 32GB HDD 146GB x2
6	NameNode 用 Kemari ホスト（副）	hnn2	仮想サーバ jt を動作させるためのホストサーバ。hnn 故障時に nn が動作する。	DL380G5 QC XE5345
				QuadCore/2.33GHz x2 メモリ 32GB HDD 146GB x2
7	JobClient	job	JobClient サーバ。	DELL R410
				Intel(R)Xeon(R)CPUE5506 2.13GHZ x 8 メモリ 8GB HDD 13GB

Hadoop マスタサーバの選定理由を以下に記載する。

・ CPU

Hadoop マスタサーバに関しては、CPU リソースを消費することは稀である。サーバ機器の他の構成とのバランスの取れた CPU でよいと考えられる。

・ メモリ

NameNode は HDFS のファイル及びブロックのメタ情報全体を NameNode のメモリ内に保持する。そのため、Hadoop 基盤の規模に応じたメモリを搭載することが必須である。JobTracker に関しても多数の Job が走ることで、大量の CPU リソース及びメモリを消費するため、拡張性を意識したメモリを搭載する必要がある。下記に Hadoop マスタサーバのメモリ量を見積もるための基礎数値を記載する。

表 I-3 NameNode の使用メモリ量の見積もり基礎数値

No.	構成	1 単位当たりのサイズ
1	ファイル	250Byte
2	ディレクトリ	150Byte
3	ブロック	180Byte

・ ディスク

NameNode は管理情報を全てメモリに蓄えるため、また、JobTracker に関しても格納情報が少ないため、Hadoop スレーブサーバと異なり、大容量のディスクは必要ない。しかし、ディスクの故障によるメタデータの破損を防ぐために、内蔵ディスクの冗長化は必要である。従って SAS ディスクの RAID1 構成が妥当である。

・ ネットワークインタフェース

Hadoop マスタサーバは単一故障点となりうるため、サービスを提供する LAN の二重化は必須である。特に、クラスタ構成やソフトウェア FT 構成等の冗長化構成のために死活監視のための通信（ハートビート通信）を行うための専用の LAN を構成する必要がある。これらに関しても多重化する必要がある。ソフトウェア FT の場合は、さらにデータ同期用の専用 LAN が必要となる。このデータ同期用のセグメントには大量の同期データが必要なため、10GBps のネットワークインターフェースカードが必要となる。上記を考え 4 つのネットワークインターフェースを持つ機器を選択した。本実証実験はソフトウェア FT を実現するために、さらに 10G の NIC を追加している。

・ リモート管理モジュール

Hadoop マスタサーバは故障時の対応が必要であるため、リモートから迅速に復旧処理が可能である必要がある。従って、管理モジュールが必要である。管理モジュールは、機器のパワーサイクルを管理できるだけでなく、OS と連携して OS のコンソール出力が閲覧できる機器を選定するべきである。本実証実験では iLO を搭載したサーバを選択した。

Hadoop スレーブサーバ

Hadoop スレーブサーバである計算サーバは、異なるスペックのハードウェアが混在している。計算サーバとして利用する機器及びその CPU、メモリ、ハードディスク、台数、格納ラックについて表 I-4 にまとめる。

表 I-4 Hadoop スレーブサーバ

No.	機器	CPU	メモリ	HDD	台数	ラック
1	DL380G5 XE5345	Xeon QuadCore/	8GB	SAS 146GB x 2	4	r3

		2.33GHz x2				
2	DL360G5 XX5460	Xeon QuadCore/ 3.16G	6GB	SAS 146GB x 2	8	r3
3	DL360G5 LV DC X5148	Xeon DualCore/ 2.33G	2GB	SAS 72GB x 2	16	r4
4	DL360G6 XE 5504 1P4C	Xeon QuadCore/2 G	6GB	SAS300GB x 2	18	r5
5	Express 5800 iR110a-1	Core2 Duo T9400	2GB	SATA 250GB x 2	10	r6
					40	r7

Hadoop スレーブサーバの選定理由を以下に記載する。

・ CPU

Hadoop は大量のサーバで計算処理を行うため、購入時点でもっとも価格性能比が高い構成を採用した。本報告書の執筆時点(2010 年 1 月時点)では、Intel Xeon 5500 番台 2GHz 4 コア程度が、価格性能比が高い構成になるサーバ製品が多い（注)。そのため、該当 CPU を含めることとした。また、混在性を検証対象とするために、価格性能比の高い CPU を複数選択した。

（注）CPU に関する性能比較は、標準化されたベンチマークのスコア値に基づいた値を指標にとることが可能である。標準化されたベンチマーク及びその検証結果に関する資料は、spec.org や TPC 等のサイトで一覧化することができる。TPC のサイトでは、価格性能比を定義しており、現時点での価格性能比が一番よいサーバの一覧化を行っている。

・ メモリ

Hadoop では、計算をタスク(Map タスク、Reduce タスク)という単位に分割して実行する仕組みになっている。タスク 1 つ当たりの Java VM ヒープサイズは 200MB(デフォルト)、CPU 1 コア当たり同時実行 Map タスク 1 個、同時実行 Reduce タスク 1 個の設定が基準値となっている。上記 1 コア当たりのメモリ量を計算すると、

(Map タスク 2 個+Reduce タスク 1 個)×200MB = 600MB

となる。その他、JavaVM ヒープサイズを増やす可能性、Hadoop のデーモン(TaskTracker・DataNode 等)、OS 自体等のメモリ消費を考えて、サーバ搭載の CPU1 コア当たり最低 1GB のメモリをメモリ選定の基準として採用し、データセンタ業者保有のサーバの中から選択した。

・ ネットワークインタフェース

Hadoop では Shuffle 処理と Reduce 処理で多くのネットワーク通信が発生するため、高速な 1Gbps のネットワークインタフェースを採用する必要がある。直接接続するネットワークスイッチの冗長化は行わず、運用 LAN を設置しないため、ネットワークインタフェースは 1 個でよい。また、Hadoop 環境自動構築・増設を行うため、ネットワークブートの規格である PXE ブートが行える必要がある。BIOS 設定の手間を省くため、初期購入状態で BIOS のデフォルト値で、PXE ブート機能が有効になっていること、及びブート順序は、ハードディスクの後に PXE ブートとなっていることが望ましい。

これらを満たすように機器選定を行い、ハードディスクに OS がインストールされている際はハードディスクより機器が起動し、そうでない場合には、ネットワーク経由で機器の自動インストールを行うことができる。

運用系サーバ

運用系サーバについて表 I-2 にまとめる。なお、運用管理・構成管理系サーバは全てラック r2 に格納する。

表 I-5 運用系サーバ

No.	構成要素	ホスト名	用途と機能	機器
1	運用管理サーバ	mg1	可視化、監視サービスを提供する。 Nagios、Ganglia サーバが動作する。	HP Compaq dc7800 SFF E8400
				Intel(R)Pentium(R)4CPU 3.00GHZ x 2 メモリ 2G HDD 120GB
2	構成管理サーバ	pp1	Puppet サーバ、DNS、DHCP、TFTP が動作する。主に初期構築時に利用する。	HP Compaq dc7800 SFF E8400
				Intel(R)Core(TM)2 Duo CPU E8400 3.00GHZ x 2 メモリ:2GB

No.	構成要素	ホスト名	用途と機能	機器
				HDD:42GB x1
3	運用構成サーバ	pp2	運用管理サーバ、構成管理サーバの待機系として動作する。	NEC MATE ME-8 MY30A/E-8
				Intel(R) Core(TM)2 Duo CPU E8400 3.00GHz x2 メモリ 2 GB HDD 120GB

ネットワーク機器

本実証実験で利用したサーバ機器の製品名・モデル・主要スペック・台数について表 I-6 にまとめる。

表 I-6 ネットワーク機器一覧

No.	構成要素	機種名	主要機能	インターフェース数	台数
1	L3スイッチ	WS-C3750G-24TS-E	・スタック ・リンクアグリゲーション (EtherChannel)	・イーサネット 10/100/1000 ポート × 24 ・ SFP ベース ギガビット イーサネット ポート× 4	2
2	L2スイッチ	WS-C3750E-24TD-S	・リンクアグリゲーション (EtherChannel)	・10/100/1000 ポート × 24、X2 ベース 10 ギガビット イーサネット ポート × 2	4
3		WS-C3750G-24TS-E	・リンクアグリゲーション (EtherChannel)	・イーサネット 10/100/1000 ポート × 24 ・ SFP ベース ギガビット イーサネット ポート× 4	2

ネットワーク機器の選定の際に必要となった要件を以下に記載する。

表 I-7 スイッチ-1 選定要件

No.	要件	内容
1	リンクアグリゲーション機能	Hadoop 計算においては Hadoop スレーブサーバ同士が計算データの交換が高頻度、高負荷で起こりうる。従ってラックをまたがる通信量は Gigabit の帯域だけでは足りなくなる。従って複数のインターフェースを束ねて実効的な帯域を増加させる機能であるリンクアグリゲーション機能を有している機器を選定した。
2	ネットワーク経由での管理機能	ラック内スイッチは Hadoop 基盤の規模に比例して増大するために、管理が煩雑となる。従ってネットワーク経由で管理できる機能が求められる。多くのスイッチは Telnet や SSH などにより管理できる機器を選定した。
3	冗長化機能 / ポート数拡張機能	規模に応じて増大するラック内スイッチが全て接続されるため、ラック増設等に応じた、ポート数を拡張できる機能が必要である。一般的にこれは L3 スイッチを専用のケーブルで接続することで実現できる（スタック機能）。スタック機能とリンクアグリゲーション機能を組み合わせることにより、コア L3 スイッチ一つが故障しても、残りのコア L3 スイッチと接続された通信路で通信ができるように構成することが可能な機種が多い。本実証実験では、冗長化機能とポート数拡張機能を満たすために、スタック接続ができる機器を選定した。
4	DHCP リレーエージェント機能（IP ヘルパー機能）	自動構築（初期構築・増設・復旧）のためにセグメントの異なる IP に対して DHCP サーバは IP の割当を実施することが必要である。DHCP は通信プロトコルとしてブロードキャストであり、異なるセグメントに対して通信が不可能である。そのため、L3 スイッチには、DHCP のパケットを中継する機能を有する製品が存在する。この機能を利用できる機器を選定した。

表 I-8 スイッチ-2 選定要件

No.	要件	内容
1	リンクアグリゲーション機能	表 I-7 のリンクアグリゲーション機能に同じ。
2	ネットワーク経由での管理機能	表 I-7 のネットワーク経由での管理機能に同じ。
3	ポート数	Hadoop 基盤では、Hadoop スレーブサーバとスイッチ間の接続は 1 本の LAN ケーブルで接続される。従ってラック内スイッチはラックに存在する全てのサーバの数よりも多くのポートを持つ必要がある。それに加え、コア L3 スイッチとリンクアグリゲーション機能で接続するための複数（本実証実験では 4 本）のポートを利用する。また、ポート故障を想定した代替ポート、緊急時のメンテナンス用のポートが必要となる。これらのポート数を満たすような機器を選定した。

I.2.1.3 利用ソフトウェア

構成要素ごとに利用する主要ソフトウェアとそのバージョンを表 I-9 に示す。OS は全て CentOS 5.3 を利用している。

表 I-9 主要ソフトウェア一覧

No.	構成要素	主要ソフトウェア（括弧内はバージョン）
1	JobTracker	・hadoop（0.20.1） ・ganglia-gmond（3.1.2） ・nagios-plugin（1.4.14） ・puppet（0.24.8）
2	NameNode	・hadoop（0.20.1） ・ganglia-gmond（3.1.2） ・nagios-plugin（1.4.14） ・puppet（0.24.8）
3	JobTracker ホスト(正) JobTracker ホスト(副) NameNode ホスト(正)	・xen（3.0.3） ・drbd（8.3.2） ・heartbeat(2.1.4)

No.	構成要素	主要ソフトウェア（括弧内はバージョン）
	NameNode ホスト（副）	・kemari(v1) ・ganglia-gmond（3.1.2） ・nagios-plugin（1.4.14） ・puppet (0.24.8)
4	Job クライアント	・hadoop（0.20.1） ・ganglia-gmond（3.1.2） ・nagios-plugin（1.4.14） ・puppet（0.24.8）
5	構成管理サーバ（正） 構成管理サーバ（副）	・puppet-server（0.24.8） ・bind-chroot(9.3.4) ・bind-libs(9.3.4) ・bind-utils(9.3.4) ・bind(9.3.4) ・caching-nameserver(9.3.4) ・ganglia-gmond（3.1.2） ・nagios-plugin（1.4.14） ・puppet（0.24.8） ・ypbind(1.19)
6	運用管理サーバ	・nagios-3.2.0-1 ・ganglia-gmetad(3.1.2) ・ganglia-web(3.1.1) ・ganglia-gmond（3.1.2） ・libganglia（3_1_0-3.1.2） ・nagios-plugin（1.4.14） ・net-snmp(5.3.2.2) ・puppet（0.24.8）
7	運用・構成サーバ（副）	・puppet-server（0.24.8-1） ・bind-chroot(9.3.4) ・bind-libs(9.3.4) ・bind-utils(9.3.4) ・bind(9.3.4) ・caching-nameserver(9.3.4) ・bind （9.3.4） ・ganglia-gmond（3.1.2） ・nagios-plugin（1.4.14）

No.	構成要素	主要ソフトウェア（括弧内はバージョン）
		・puppet（0.24.8） ・ypbind(1.19-11)
8	Hadoop スレーブサーバ	・hadoop（0.20.1） ・ganglia-gmond（3.1.2） ・nagios-plugin（1.4.14） ・puppet（0.24.8）

各ソフトウェアの用途・機能及び入手先を表 I-10 にまとめる。

表 I-10 主要ソフトウェアの構成パッケージと入手先

No.	ソフトウェア名	パッケージ名	入手先
1	Hadoop	hadoop-0.20.1	http://www.apache.org/dyn/closer.cgi/hadoop/core/ 以下のパッチを適用している。 http://issues.apache.org/jira/browse/MAPREDUCE-112 http://issues.apache.org/jira/browse/MAPREDUCE-118 http://issues.apache.org/jira/browse/MAPREDUCE-1182 http://issues.apache.org/jira/browse/HADOOP-5759 https://issues.apache.org/jira/browse/HADOOP-4675
2	BIND	ypbind-1.19-11.el5.x86_64.rpm bind-chroot-9.3.4-10.P1.el5x86_64.rpm bind-libs-9.3.4-10.P1.el5x86_64.rpm bind-utils-9.3.4-10.P1.el5.x86_64.rpm bind-9.3.4-10.P1.el5.x86_64.rpm	CentOS5.3 同梱版を利用。

No.	ソフトウェア名	パッケージ名	入手先
		caching-nameserver-9.3.4-10.P1.el5.x86_64.rpm	
3	DRBD	drbd-8.3.2.tar.gz	http://oss.linbit.com/drbd/
4	Ganglia	ganglia-3.1.2.tar.gz	http://sourceforge.net/projects/ganglia/files/ganglia%20monitoring%20core/
5	Heartbeat	・Heartbeat 本体 heartbeat-2.1.4-1.rhel5.x86_64.RPMS.tar.gz ・追加パッケージ hb-monitor-1.02-1.hb214.x86_64.rpm	・Heartbeat 本体 http://www.linux-ha.org/wiki/Download/ja ・追加パッケージ http://www.linux-ha.org/wiki/Contrib/ja
6	Kemari	・Kemari 本体 kemari-xen-testing.tar.bz2 ・Kemari RA ha-tools.tar.bz2	http://sourceforge.net/projects/kemari/files/-kemari-v1
7	Nagios	nagios-3.2.0.tar.gz	http://www.nagios.org/download/core
8	Net-SNMP	net-snmp-5.3.2.2-5.el5 net-snmp-utils-5.3.2.2-5.el5 net-snmp-perl-5.3.2.2-5.el5 net-snmp-libs-5.3.2.2-5.el5	CentOS5.3 同梱版を利用。
9	Puppet	puppet-server-0.24.8-1.el5.1.noarch.rpm puppet-0.24.8-1.el5.1.n	http://download.fedora.redhat.com/pub/epel/5/x86_64/repoview/letter_p.group.html

No.	ソフトウェア名	パッケージ名	入手先
		oarch.rpm facter-1.5.2-2.el5.noarch.rpm	
10	Xen	kemari-xen-testing.tar.bz2	http://sourceforge.net/projects/kemari/files/

I.2.2 システム詳細構成

本節では、システムの詳細な構成についてまとめる。

I.2.2.1 システム詳細構成図

図 I-3 にクラウド型分散処理基盤のシステム詳細構成図を示す。

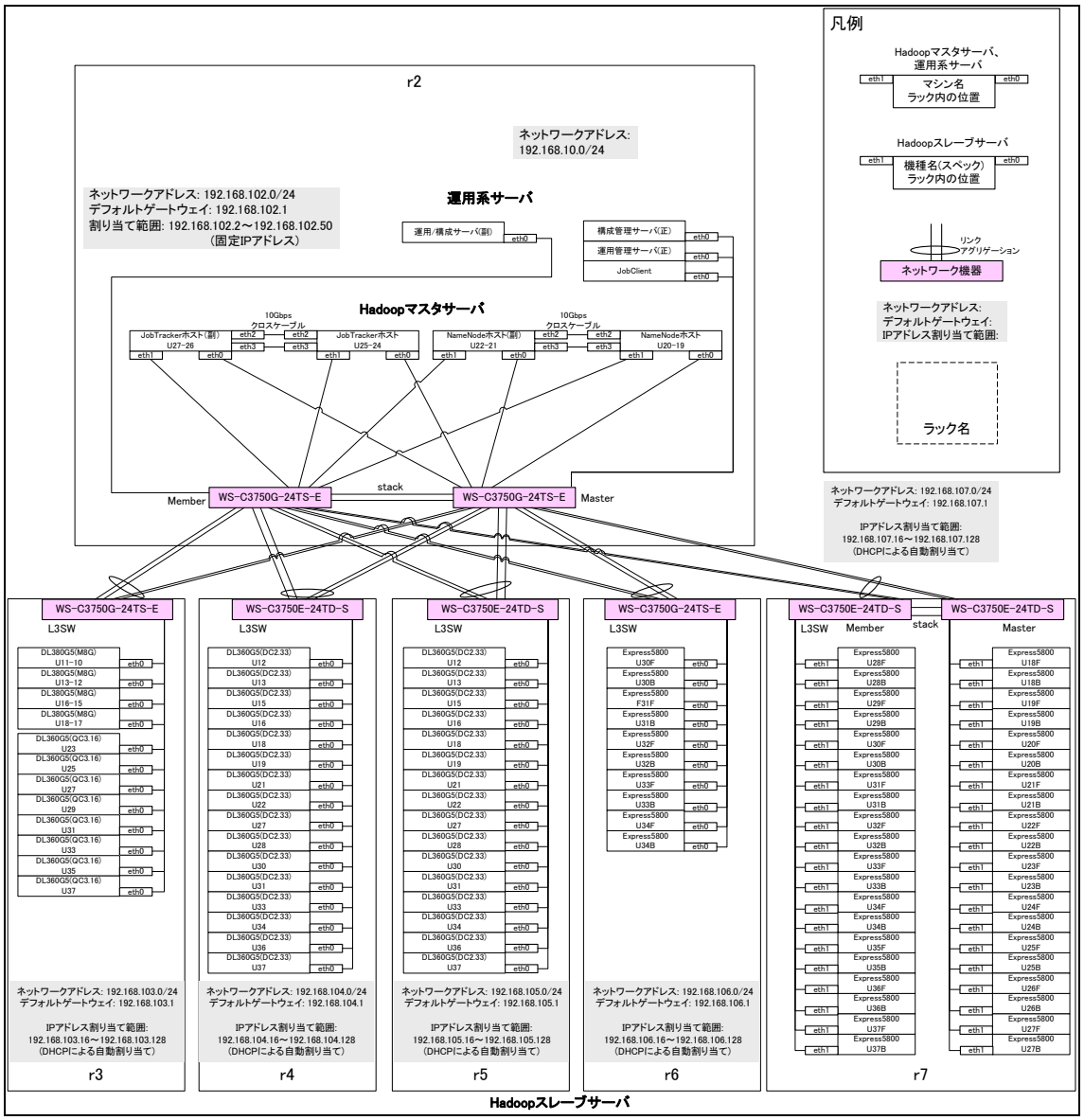


図 I-3 クラウド型分散処理基盤のシステム詳細構成図

I.2.2.2 構成要素詳細

表 I-11 にクラウド型分散処理基盤のシステム構成要素の詳細をまとめる。

表 I-11 クラウド型分散処理基盤のシステム構成要素詳細

ラック名	位置	SWポートNo	機器	ホスト名	機能	IPアドレス	備考	ベンダ	コア数	MEM(GB)
r2	-	-	WS-C3750G-24TS-E x2	r2	コアL3スイッチ	192.168.102.1 192.168.103.1 192.168.104.1 192.168.105.1 192.168.106.1 192.168.107.1	・r2.example.netはvlan102に付与された名前。 ・vlan103/vlan104/vlan105/vlan107のVLANのGWとしてのIPアドレスを持ち、このIPがDHCPリレーエージェントとなる。	CISCO	-	-
r2	仮想			nn	NameNodeサーバ	192.168.102.10			仮想	
r2	U19-20	Gi1/0/21(eth0) Gi2/0/21(eth1)	DL380G5(M32G)	hnn1	NameNodeホスト	192.168.102.11	ILO: 192.168.102.201(Gi1/0/19)	HP	8	32
r2	U21-22	Gi1/0/22(eth0) Gi2/0/22(eth1)	DL380G5(M32G)	hnn2	NameNodeホスト(副)	192.168.102.12	ILO: 192.168.102.202(Gi1/0/20)	HP	8	32
r2	仮想			jt	JobTrackerサーバ	192.168.102.20			仮想	
r2	U24-25	Gi1/0/23(eth0) Gi2/0/23(eth1)	DL380G5(M32G)	hjt1	JobTrackerホスト	192.168.102.21	ILO: 192.168.102.203(Gi2/0/19)	HP	8	32
r2	U26-27	Gi1/0/24(eth0) Gi2/0/24(eth1)	DL380G5(M32G)	hjt2	JobTrackerホスト(副)	192.168.102.22	ILO: 192.168.102.204(Gi2/0/20)	HP	8	32
r2	-	Gi1/0/12	Compaq dc7800 SFF	pp1	構成管理サーバ	192.168.102.2	puppetサーバ/DNS/DHCP/TFTP	HP	4	2
r2	-	Gi2/0/12	MATE ME-8 MY30A/E-8	pp2	運用/構成サーバ(副)	192.168.102.3	puppetサーバ/DNS/DHCP/TFTP	NEC	4	2
r2	-	Gi1/0/13	Compaq dc7800 SFF	mg1	運用管理サーバ	192.168.102.5	Nagios/Ganglia	HP	4	2
r2	-	Gi1/0/17(eth0) Gi1/0/17(eth1)	DELL R410	job2	Jobクライアント	自動割当		DELL	8	8
r3	-	-	WS-C3750G-24TS-E	r3	スイッチ	192.168.103.254	スイッチの管理IP	CISCO	-	-
r3	U11-10	Gi0/1	DL380G5 XE5345	r3-1-0-01				HP	8	8
r3	U13-12	Gi0/2	DL380G5 XE5345	r3-1-0-02				HP	8	8
r3	U16-15	Gi0/3	DL380G5 XE5345	r3-1-0-03				HP	8	8
r3	U18-17	Gi0/4	DL380G5 XE5345	r3-1-0-04				HP	8	8
r3	U23	Gi0/5	DL360G5 XX5460	r3-1-0-05				HP	4	6
r3	U25	Gi0/6	DL360G5 XX5460	r3-1-0-06	計算サーバ	自動割当		HP	4	6
r3	U27	Gi0/7	DL360G5 XX5460	r3-1-0-07				HP	4	6
r3	U29	Gi0/8	DL360G5 XX5460	r3-1-0-08				HP	4	6
r3	U31	Gi0/9	DL360G5 XX5460	r3-1-0-09				HP	4	6
r3	U33	Gi0/10	DL360G5 XX5460	r3-1-0-10				HP	4	6
r3	U35	Gi0/11	DL360G5 XX5460	r3-1-0-11				HP	4	6
r3	U37	Gi0/12	DL360G5 XX5460	r3-1-0-12				HP	4	6
r4	-	-	WS-C3750E-24TD-S	r4	スイッチ	192.168.104.254	スイッチの管理IP	CISCO		
r4	U12	Gi0/1	DL360G5 LV DC X5148	r4-1-0-01				HP	2	2
r4	U13	Gi0/2	DL360G5 LV DC X5148	r4-1-0-02				HP	2	2
r4	U15	Gi0/3	DL360G5 LV DC X5148	r4-1-0-03				HP	2	2
r4	U16	Gi0/4	DL360G5 LV DC X5148	r4-1-0-04				HP	2	2
r4	U18	Gi0/5	DL360G5 LV DC X5148	r4-1-0-05				HP	2	2
r4	U19	Gi0/6	DL360G5 LV DC X5148	r4-1-0-06				HP	2	2
r4	U21	Gi0/7	DL360G5 LV DC X5148	r4-1-0-07				HP	2	2
r4	U22	Gi0/8	DL360G5 LV DC X5148	r4-1-0-08	計算サーバ	自動割当		HP	2	2
r4	U27	Gi0/9	DL360G5 LV DC X5148	r4-1-0-09				HP	2	2
r4	U28	Gi0/10	DL360G5 LV DC X5148	r4-1-0-10				HP	2	2
r4	U30	Gi0/11	DL360G5 LV DC X5148	r4-1-0-11				HP	2	2
r4	U31	Gi0/12	DL360G5 LV DC X5148	r4-1-0-12				HP	2	2
r4	U33	Gi0/13	DL360G5 LV DC X5148	r4-1-0-13				HP	2	2
r4	U34	Gi0/14	DL360G5 LV DC X5148	r4-1-0-14				HP	2	2
r4	U36	Gi0/15	DL360G5 LV DC X5148	r4-1-0-15				HP	2	2
r4	U37	Gi0/16	DL360G5 LV DC X5148	r4-1-0-16				HP	2	2
r5	-	-	WS-C3750E-24TD-S	r5	スイッチ	192.168.105.254	スイッチの管理IP	CISCO		
r5	U10	Gi0/1	DL360G6 XE 5504 1P4C	r5-1-0-01				HP	4	6
r5	U11	Gi0/2	DL360G6 XE 5504 1P4C	r5-1-0-02				HP	4	6
r5	U13	Gi0/3	DL360G6 XE 5504 1P4C	r5-1-0-03				HP	4	6
r5	U14	Gi0/4	DL360G6 XE 5504 1P4C	r5-1-0-04				HP	4	6
r5	U16	Gi0/5	DL360G6 XE 5504 1P4C	r5-1-0-05				HP	4	6
r5	U17	Gi0/6	DL360G6 XE 5504 1P4C	r5-1-0-06				HP	4	6
r5	U20	Gi0/7	DL360G6 XE 5504 1P4C	r5-1-0-07				HP	4	6
r5	U21	Gi0/8	DL360G6 XE 5504 1P4C	r5-1-0-08				HP	4	6
r5	U23	Gi0/9	DL360G6 XE 5504 1P4C	r5-1-0-09	計算サーバ	自動割当		HP	4	6
r5	U24	Gi0/10	DL360G6 XE 5504 1P4C	r5-1-0-10				HP	4	6
r5	U27	Gi0/11	DL360G6 XE 5504 1P4C	r5-1-0-11				HP	4	6
r5	U30	Gi0/12	DL360G6 XE 5504 1P4C	r5-1-0-12				HP	4	6
r5	U31	Gi0/13	DL360G6 XE 5504 1P4C	r5-1-0-13				HP	4	6
r5	U33	Gi0/14	DL360G6 XE 5504 1P4C	r5-1-0-14				HP	4	6
r5	U34	Gi0/15	DL360G6 XE 5504 1P4C	r5-1-0-15				HP	4	6
r5	U36	Gi0/16	DL360G6 XE 5504 1P4C	r5-1-0-16				HP	4	6
r5	U37	Gi0/17	DL360G6 XE 5504 1P4C	r5-1-0-17				HP	4	6
r5	U37	Gi0/18	DL360G6 XE 5504 1P4C	r5-1-0-18				HP	4	6

ラック名	位置	SWポートNo	機器	ホスト名	機能	IPアドレス	備考	ベンダ	コア数	MEM（GB）
r6	-	-	WS-C3750G-24TS-E	r6	スイッチ	192.168.106.254	スイッチの管理IP	CISCO		
r6	U30F	Gi0/1	Express5800	r6-1-0-01				NEC	2	2
r6	U30B	Gi0/2	Express5800	r6-1-0-02				NEC	2	2
r6	U31F	Gi0/3	Express5800	r6-1-0-03				NEC	2	2
r6	U31B	Gi0/4	Express5800	r6-1-0-04				NEC	2	2
r6	U32F	Gi0/5	Express5800	r6-1-0-05	計算サーバ	自動割当		NEC	2	2
r6	U32B	Gi0/6	Express5800	r6-1-0-06				NEC	2	2
r6	U33F	Gi0/7	Express5800	r6-1-0-07				NEC	2	2
r6	U33B	Gi0/8	Express5800	r6-1-0-08				NEC	2	2
r6	U34F	Gi0/9	Express5800	r6-1-0-09				NEC	2	2
r6	U34B	Gi0/10	Express5800	r6-1-0-10				NEC	2	2
r7	-	-	WS-C3750E-24TD-S x2	r7	スイッチ	192.168.107.254	スイッチの管理IP	CISCO		
r7	U18F	Gi1/0/1	Express5800	r7-1-0-01				NEC	2	2
r7	U18B	Gi1/0/2	Express5800	r7-1-0-02				NEC	2	2
r7	U19F	Gi1/0/3	Express5800	r7-1-0-03				NEC	2	2
r7	U19B	Gi1/0/4	Express5800	r7-1-0-04				NEC	2	2
r7	U20F	Gi1/0/5	Express5800	r7-1-0-05				NEC	2	2
r7	U20B	Gi1/0/6	Express5800	r7-1-0-06				NEC	2	2
r7	U21F	Gi1/0/7	Express5800	r7-1-0-07				NEC	2	2
r7	U21B	Gi1/0/8	Express5800	r7-1-0-08				NEC	2	2
r7	U22F	Gi1/0/9	Express5800	r7-1-0-09				NEC	2	2
r7	U22B	Gi1/0/10	Express5800	r7-1-0-10				NEC	2	2
r7	U23F	Gi1/0/11	Express5800	r7-1-0-11				NEC	2	2
r7	U23B	Gi1/0/12	Express5800	r7-1-0-12				NEC	2	2
r7	U24F	Gi1/0/13	Express5800	r7-1-0-13				NEC	2	2
r7	U24B	Gi1/0/14	Express5800	r7-1-0-14				NEC	2	2
r7	U25F	Gi1/0/15	Express5800	r7-1-0-15				NEC	2	2
r7	U25B	Gi1/0/16	Express5800	r7-1-0-16				NEC	2	2
r7	U26F	Gi1/0/17	Express5800	r7-1-0-17				NEC	2	2
r7	U26B	Gi1/0/18	Express5800	r7-1-0-18				NEC	2	2
r7	U27F	Gi1/0/19	Express5800	r7-1-0-19				NEC	2	2
r7	U27B	Gi1/0/20	Express5800	r7-1-0-20				NEC	2	2
r7	U28F	Gi2/0/1	Express5800	r7-2-0-01	計算サーバ	自動割当		NEC	2	2
r7	U28B	Gi2/0/2	Express5800	r7-2-0-02				NEC	2	2
r7	U29F	Gi2/0/3	Express5800	r7-2-0-03				NEC	2	2
r7	U29B	Gi2/0/4	Express5800	r7-2-0-04				NEC	2	2
r7	U30F	Gi2/0/5	Express5800	r7-2-0-05				NEC	2	2
r7	U30B	Gi2/0/6	Express5800	r7-2-0-06				NEC	2	2
r7	U31F	Gi2/0/7	Express5800	r7-2-0-07				NEC	2	2
r7	U31B	Gi2/0/8	Express5800	r7-2-0-08				NEC	2	2
r7	U32F	Gi2/0/9	Express5800	r7-2-0-09				NEC	2	2
r7	U32B	Gi2/0/10	Express5800	r7-2-0-10				NEC	2	2
r7	U33F	Gi2/0/11	Express5800	r7-2-0-11				NEC	2	2
r7	U33B	Gi2/0/12	Express5800	r7-2-0-12				NEC	2	2
r7	U34F	Gi2/0/13	Express5800	r7-2-0-13				NEC	2	2
r7	U34B	Gi2/0/14	Express5800	r7-2-0-14				NEC	2	2
r7	U35F	Gi2/0/15	Express5800	r7-2-0-15				NEC	2	2
r7	U35B	Gi2/0/16	Express5800	r7-2-0-16				NEC	2	2
r7	U36F	Gi2/0/17	Express5800	r7-2-0-17				NEC	2	2
r7	U36B	Gi2/0/18	Express5800	r7-2-0-18				NEC	2	2
r7	U37F	Gi2/0/19	Express5800	r7-2-0-19				NEC	2	2
r7	U37B	Gi2/0/20	Express5800	r7-2-0-20				NEC	2	2

II　用語集

No.	用語	意味
	クラウド型分散処理基盤	クラウドコンピューティングを実現するための基盤技術の 1 つで、大量のサーバで分散処理を行い大規模な計算能力を提供する技術のことである。実装方式としては、Google が考案し実用化した GFS /MapReduce、GFS/MapReduce のオープンソース実装である Hadoop、MPI(Message Passing Interface)に基づく MPICH、Open MPI 等が相当する。
	プローブ情報	車載機器や携帯電話などの GPS 機能を搭載した移動体をセンサーとみなし、そこから発信される位置情報や取得時刻などを含んだデータを指す。
	管理ノード	クラウド型分散処理基盤において、基盤全体での進捗状況・データ格納場所等の管理を行うサーバのことである。Hadoop では「Hadoop マスタサーバ」が相当する。
	処理ノード	クラウド型分散処理基盤において、大量のサーバで構成され計算処理とデータ格納の役目を行うサーバのことである。Hadoop では「Hadoop スレーブサーバ」が相当する。
	コモディティ製品	市場で広く売られており、製造するメーカごとの機能差/品質差が小さくなっている製品のこと。IT 分野では PC や IA サーバがコモディティ製品の例として挙げられる。
	オープンソースソフトウェア	ソースコードが公開されており、独自のライセンスに基づいて作成されるソフトウェアのことである。Open Source Initiative がオープンソースソフトウェアのライセンスに関して 10 個の条件を定義しており、適合しているライセンスの代表例として GPL, Apache ライセンス等が挙げられる。
	Hadoop	大規模データの分散処理を実現するオープンソースソフトウェアで、Apache Software Foundations のトッププロジェクトの 1 つである。Hadoop プロジェクトの主要な構成要素として、分散ファイルシステムである HDFS と、分散処理プログラムモデル/フレームワークである MapReduce の 2 つのソフトウェアが挙げられる。

No.	用語	意味
	MapReduce	分散処理プログラムモデルである。大量データの処理を Map と Reduce の 2 つのフェーズに分けて行う仕組みである。Map フェーズでは、処理対象の入力データを細かいブロックに分割し、多数のサーバに分散して処理を実施する。Reduce フェーズでは、Map フェーズでの処理結果に対して、同じキーを持つデータを同じサーバに集め集計を行う。Map フェーズは HDFS から処理内容を読み込み、Reduce フェーズは HDFS に処理結果を出力する。 MapReduce の処理全体をジョブと呼び、ジョブは多数の Map タスクと Reduce タスクに分割して実行される。 MapReduce は、ジョブとタスクの実行状況を管理する JobTracker と、タスクの実行を行う TaskTracker の 2 種類で構成される。
	HDFS	Hadoop の分散ファイルシステムであり、サイズが非常に大きいファイルを 64MB 程度のブロックに分割して多数のサーバに分散して格納する仕組みである。1 つのファイルを多数のサーバに格納しているため、高い読み込み性能を持っている。また、1 個のブロックを 3 個程度のサーバにコピーして格納する仕組みとなっているため、サーバに故障が発生した場合でもデータが失われることなく動作を続けることが可能である。HDFS はファイルのメタ情報を管理する NameNode と、データブロックを格納する多数のサーバである DataNode の 2 種類で構成される。
	Hadoop マスタサーバ	Hadoop において全体を管理する役目を果たすサーバのことで、JobTracker と NameNode の総称である。
	Hadoop スレーブサーバ	Hadoop において大量のサーバで構成され計算処理とデータブロックの格納を行うサーバのことで、TaskTracker と DataNode の総称である。
	ソフトウェア FT	サーバの可用性を向上させる技術である。通常はハードウェアで実施されているサーバ間の同期処理を、ソフトウェアで実現する。サーバの実行状態を常時同期させているため、ハードウェア故障時にもサービスに影響を与えず処理を継続することができる。 一般的に仮想化技術を利用して実装されており、仮想マシン単位で同期、処理の継続を行う。

No.	用語	意味
	Kemari	ソフトウェア FT を実現するオープンソースソフトウェアである。ネットワーク I/O、ディスク I/O を契機としてメモリの同期を行うことにより、故障時に実行されていた処理を継続する。Kemari 自身はメモリ同期の機能のみを保持するため、故障時に切り替えを行うには HA クラスタソフトウェアの併用が必要である。 http://www.osrg.net/kemari/
	HA クラスタ	一般的に 2 台のサーバを用いて、サービスの可用性を向上させる技術である。現用系、待機系の 2 種類のサーバから構成され、現用系のサーバに故障が発生した場合にサービスを待機系に切り替える。サービスの停止時間を短くすることができる。
	Heartbeat	HA クラスタを実現するオープンソースソフトウェアである。各種ハードウェア・ソフトウェアを監視対象として、一定間隔で生存確認を行う。これにより、監視対象に故障が発生したことを検知することができる。故障を検知したあとは、待機系への自動切り替えを行う。 http://linux-ha.org
	Kickstart	Red Hat Linux 及び互換 Linux ディストリビューションにおけるインストールの仕組みの一つ。一般的な Linux のインストールでは、対話的な Linux インストーラーにおいて、ディスク初期化方法や、インストールパッケージの選択等を行う。一方 Kickstart では、これらの入力項目をあらかじめ記述した設定ファイル（Kickstart ファイル）を入力とすることで、非対話的にインストールを実施することができる。
	Puppet	大量のサーバに対する設定を一括して行うためのオープンソースソフトウェア。設定ファイルの配布や、ファイルの権限情報の反映、サービスの起動や停止、パッケージのインストールなどを行うことができる。設定を一括管理する Puppet マスタサーバと、実際に設定を反映する Puppet サーバから構成される。 http://reductivelabs.com/products/puppet/

No.	用語	意味
	Ganglia	クラスタ等の大量にサーバが存在する環境において、サーバのシステムリソースをグラフ化するためのオープンソースソフトウェア。CPU 使用率やネットワーク使用量などのリソース情報を取得し、それらを集計したグラフを Web 画面で提供する。ハイパフォーマンスコンピューティング分野での導入実績が多い。 http://ganglia.sourceforge.net/