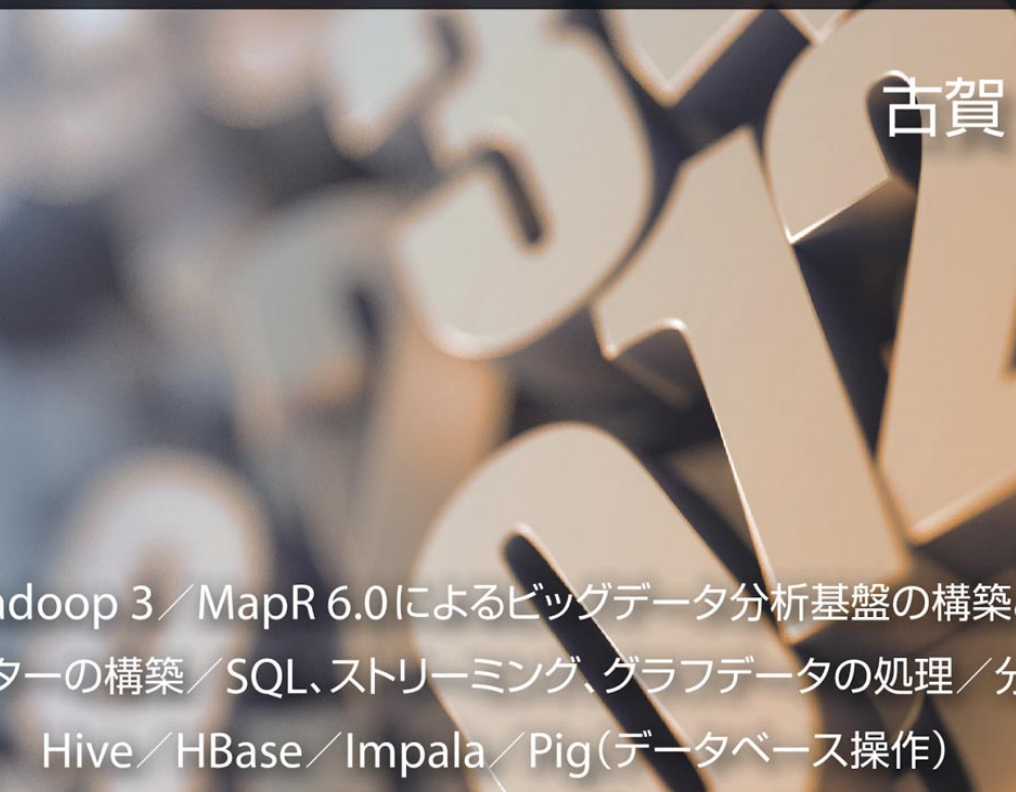
ビッグデータ分析基盤の構築事例集

Hadoo

スター構築実践ガ



古賀

adoop 3／MapR 6.0によるビッグデータ分析基盤の構築
ターの構築／SQL、ストリーミング、グラフデータの処理／分
Hive／HBase／Impala／Pig（データベース操作）

impress
top gear

ビッグデータ分析基盤の構築事例集

Hadoop

クラスター構築実践ガイド

古賀 政純 著

Apache Hadoop 3／MapR 6.0によるビッグデータ分析基盤の構築と運用管理

Spark（クラスターの構築／SQL、ストリーミング、グラフデータの処理／分類器の利用）

Hive／HBase／Impala／Pig（データベース操作）

Sqoop／Flume（データのインポート／エクスポート）

Mahout（クラス分類と協調フィルタリング）

●本書の利用について

◆本書の内容に基づく実施・運用において発生したいかなる損害も、株式会社インプレスと著者は一切の責任を負いません。

◆本書の内容は、2018 年 4 月の執筆時点のものです。本書で紹介した製品／サービスなどの名称や内容は変更される可能性があります。あらかじめご注意ください。

◆Web サイトの画面、URL などは、予告なく変更される場合があります。あらかじめご了承ください。

◆本書に掲載した操作手順は、実行するハードウェア環境や事前のセットアップ状況によって、本書に掲載したとおりにならない場合もあります。あらかじめご了承ください。

●商　標

◆Apache Hadoop、Apache Spark、Apache Sqoop、Apache Flume、Apache Hive、Apache HBase、Apache Impala、Apache Pig、Apache Mahout は、Apache Software Foundation の米国およびその他の国での商標です。

◆MapR は、MapR Technologies, Inc. の米国およびその他の国における商標です。

◆Linux は、Linus Torvalds の米国およびその他の国における商標もしくは登録商標です。

◆Red Hat および Red Hat をベースとしたすべての商標、CentOS マークは、米国およびその他の国における Red Hat,Inc. の商標または登録商標です。

◆その他、本書に登場する会社名、製品名、サービス名は、各社の登録商標または商標です。

◆本文中では、®、©、^{TM} は、表記しておりません。

はじめに

ビッグデータという言葉がクローズアップされたのは2010年以降のことですが、それ以前から、SNSやeコマースを営むインターネット企業では、蓄積された膨大なデータを効率よく処理する手段を模索していました。インターネットに蓄積されたデータは、単に量が多いだけでなく、処理の複雑度も非常に高いため、多くの計算リソースを必要とします。当時のRDBMSを使ったデータ処理では、すでに太刀打ちできない状況にあったためです。

このような課題に対応すべく、米国では、2000年代中頃から、1台のサーバーで稼働するRDBMSにデータを配置するのではなく、大量のコンピューターを使って膨大なデータを保管し、データを分散並列処理するといった手法が本格的に検討されるようになりました。1台のコンピューターのCPUの数やメモリの容量を増やしても、データの処理速度の向上に限界が見えてきたため、大量のコンピューターを並べ、いかに並列処理を高速にできるかという考えが重視されるようになってきたのです。

このように、膨大なデータを高速に処理できるITシステムを実現するには、「並列処理エンジン」を組み込む必要があります。この並列処理のためのフレームワークを提供するのが、オープンソースソフトウェアとして提供されているHadoop（ハドゥープ）です。Hadoopは、そのような大量のデータ保管や処理速度の諸問題を打破するためのソフトウェアとして注目を浴び、当初は米国を中心に利用が広がりました。そして現在では、Hadoopを取り巻くさまざまな分析用のライブラリや人工知能の一分野である機械学習といった周辺ソフトウェア（エコシステム）が登場し、知的情報処理基盤のミドルウェアとして、充実した環境を提供しています。

現在、Hadoopは国内外を問わず、最先端技術をいち早く採用する情報サービス業、製造業、通信事業者、流通業、金融業、そして、航空宇宙産業など、さまざまな分野で採用されていますが、人工知能やIoTの活用が活発化すれば、今後いっそう多くの企業や組織が、ビッグデータ分析基盤の導入に取り組むことが予想されます。しかし、Hadoopによるビッグデータ分析基盤の導入を初めて検討するIT部門やエンドユーザーにとって、その採用に向けた導入前の検討、基盤設計、構築手順、運用管理などの習得は、決してハードルが低いとはいえません。

本書は、このハードルをできるだけ低く抑え、企業における Hadoop の導入を支援するための書籍として企画されました。そのため、Hadoop の技術的な情報だけでなく、Hadoop 基盤の方向性の検討や戦略の立案、および、意思決定を行う企画部門や、実際に基盤構築を行う技術者が、導入前の検討を実践できる内容を盛り込みました。具体的には、ビッグデータ処理システムの経緯、目的、Hadoop とその周辺ソフトウェアの特徴、導入時の検討項目、注意点、システム構成例などの要点を知ることができます。また、大規模なデータ処理基盤の構築の経験がない技術者でも、Hadoop をインストールし、使用できるように、一連の手順をステップバイステップで具体的に記載しています。

本書で対象としたシステムは、2018 年 4 月にリリースされたオープンソースの Apache Hadoop 3.1.0 と、商用版 Hadoop の MapR 6.0.1 を取り上げています。また、日本の企業においても導入が積極的に行われているインメモリ型の分析、および、機械学習のフレームワークを提供する Spark、機械学習エンジンの Mahout、従来の RDBMS の SQL 文のようなデータ操作を実現する Hive、Impala、列指向の分散データベースである HBase、データの加工や抽出を行う Pig、RDBMS と Hadoop 間でのデータ転送を実現する Sqoop、SNS やログなどのさまざまな種類データを Hadoop に取り込む Flume といった、利用頻度の高い Hadoop 周辺ソフトウェアの具体的な構築手順と使用例を網羅しました。これらの多くの事例により、最新のビッグデータ処理基盤で提供される機能や基本的な使用法を理解できます。

Hadoop を中心に据えた知的情報処理基盤は、人工知能、ロボット工学、IoT などの先進技術を駆使する企業や組織体に革新をもたらすものとして、その重要性がますます高まっています。近い将来において、データ量の大小に関係なく、「データを有効活用した知的情報処理基盤」の採用が、企業や組織の運営にとって、欠かすことのできないものとなるでしょう。本書によって、より多くの方々がデータ活用のための次世代 IT 基盤を実現し、大きな成果を出すことを願ってやみません。

最後に、本書の執筆機会をいただいた方々に、この場を借りて厚く御礼を申し上げます。

2018 年 4 月吉日

古賀政純

本書の表記

・注目すべき要素は、太字で表記しています。
・本文中の＊は付帯的な情報としてそのページの欄外に記しています。
・本文中の†は、用語の説明やオペレーション上の説明のために、コメントとして本文中に罫囲みの中に記しています。
・本文中の注目すべき部分は、太字で表記しています。
・コマンドラインのプロンプトは、$、#で示されます。
・実行例のユーザーの入力部分は太字で表記します。
・実行例に関する説明は、←の後に付記しています。
・実行結果の出力を省略している部分は、「...」で表記します。
・紙面の幅に収まらないコマンドラインでは、行末に“\”を入れ、改行しています。

例：

```
# clush -g cl,all -L "sysctl -p | grep swappiness"  太字で表記
n0120: vm.swappiness = 1  ←出力結果
...  省略
$ sshpass -p "password1234" \  ←改行
ssh-copy-id -o StrictHostKeyChecking=no koga@n0120
```

・プログラムや設定ファイルが1行で収まらない場合、右端で折り返し、行末に 1行入力 というマークを入れています。実際に入力する際は、改行せずに1行で入力してください。

例：

```
val maxout = graph.outDegrees.join(airports).sortBy(_._2._1, ascending=false)
maxout.take(N).foreach(println)  1行入力
```

●本書で使用した実行環境

◆ハードウェア

・HPE Apollo 4200 Gen9
・HPE Apollo 4530
・HPE ProLiant SL4540

◆ソフトウェア

・Apache Hadoop 3.1.0
・MapR 6.0.1
・Apache Spark 2.3.0（MapR 版は 2.2.1）
・Apache Hive 2.1
・Apache Impala 2.10.0
・Apache HBase 1.4.3
・Apache Pig 0.17.0
・Apache Sqoop 1.4.6
・Apache Flume 1.8.0
・Apache Mahout 0.13.0
・CentOS 7.4.1708

必要に応じて、インターネットのリポジトリからソフトウェアを取得しています。

●スタッフ
AD ／装丁：岡田 章志＋ GY
本文デザイン／制作／編集：TSUC

目　次

第１章
ビッグデータ分析基盤の概要

インターネットをベースとしたｅコマース市場の拡大やSNSの急激な利用増加、スマートデバイスの普及に伴い、膨大なデータが生成され続けています。こうした生成・蓄積されたデータを経営資源として活用し、新たな価値を見い出すことで、新商品・新サービスの創出、顧客対応の迅速化、生産性向上、開発期間の短縮など、さまざまな分野で役立てようとする動きが活発化しています。そして、こうした動向を支援するITとして注目されているのが、膨大なデータを分析するための基盤ソフトウェア、すなわち、「ビッグデータ分析基盤ソフトウェア」です。

本章では、ビッグデータ分析基盤ソフトウェアとして実績があり、かつ近年の人工知能の進化とともに急速に発展するHadoopとそのエコシステムに焦点を当て、適用分野について解説します。

1-1 ビッグデータ分析ニーズの拡大

長年にわたり、ビジネスに活用する膨大なデータを格納、分析するために、オンライントランザクション処理が可能なリレーショナルデータベース（以下、RDBMS）やデータウェアハウス（以下、DWH）が利用されてきました。これらの RDBMS を軸にしたデータ保管、および、データ処理ソフトウェアは、一般に、**データストア**と呼ばれます。これらの伝統的なタイプのデータストアは、現在でも、勤怠管理システム、顧客管理システム、財務管理システムなど、企業システムの幅広い分野で利用されています。20 世紀までのデータ分析基盤アーキテクチャといえば、RDBMS を指していたといって間違いないでしょう。

しかしこの常識は、21 世紀の初頭に覆されます。その原因となったのは、インターネットにおける膨大なデータの創出です。新たなコミュニケーション手段となった SNS、モバイルデバイスの急速な社会への浸透、e コマース市場の拡大といった社会インフラの変化によって、継続的に膨大なデータが生み出されるようになったためです。当時、RDBMS で扱っていた大規模なデータといえば、金融機関の入出金記録や小売店の POS 集計といったものです。ところが、インターネットによって生み出されたデータは、そのデータ容量が従来に比べて膨大であり、種類も多種多様、更新頻度が高いといった性質を持ち、当時の RDBMS は、もはやビッグデータの適切な分析基盤とはいえなかったのです。

1-1-1 Hadoop の誕生

こうした時代背景の中で、Google は、自社の膨大なデータを効率よく分析するための新しい手法と基盤アーキテクチャを研究しており、その研究結果を論文として発表しました[*1] [*2]。そしてこの Google の論文を参考に、米国 Yahoo!は自社の抱える膨大なデータを処理するためのソフトウェアを開発し、オープンソースとして公開したのが、ビッグデータ分析基盤のためのミドルウェア Hadoop（ハドゥープ）です。

Hadoop によるデータ処理性能が認知されると、eBay や Facebook、VISA などがこぞって Hadoop の利用を開始しました。そして現在では、欧米、日本を問わず、多くのサービスプロバイダー、製造、流通、金融、通信など、幅広い分野で利用されています（図 1-1）。

＊ 1 MapReduce: Simplified Data Processing on Large Clusters ,Jeffrey Dean and Sanjay Ghemawat, Appeared in: OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.

＊ 2 The Google File System ,Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, Appeared in: 19th ACM Symposium on Operating Systems Principles, Lake George, NY, October, 2003.

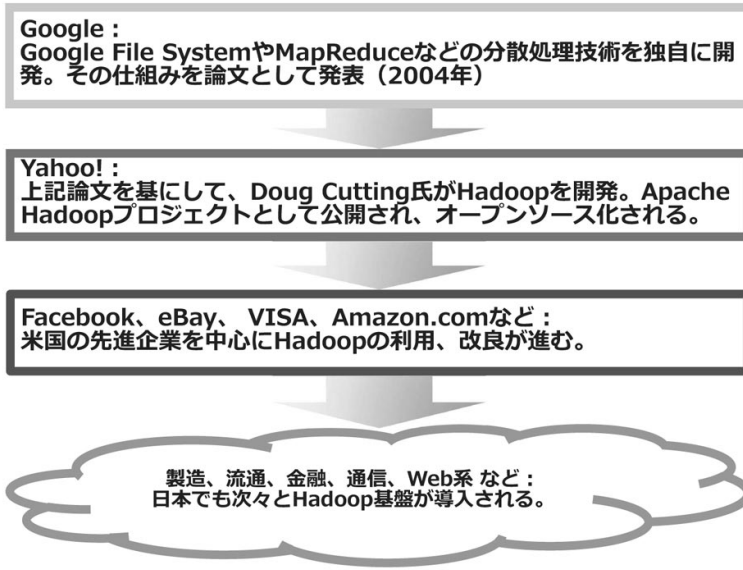


図 1-1　Hadoop の歴史的経緯

1-1-2　スケールアップ型システムの課題と Hadoop のメリット

ビッグデータを扱う従来型の RDBMS や DWH の最大の課題は、そのスケーラビリティにあります。RDBMS は、システム設計時に想定したトランザクション処理を超えると、ユーザーからの問い合わせ処理（クエリー処理）のターン・アラウンド・タイムが極端に長くなる傾向があります。このように処理能力が不足した場合の対処方法としては、サーバーを構成する CPU の高速化、メモリ容量の増設、I/O の高速化（内蔵 HDD の並列化、SSD の採用に加え、高速なファイバーチャネル方式の外部共有ストレージシステムに格納する）といった手法がとられます。しかし、サーバー 1 台を構成するハードウェアコンポーネントをスケールアップさせるのは、非常にコストがかかります。

一方 Hadoop は、システムの拡張が容易に行えるように設計されています。データサイズが大きくなっても、x86 サーバーを追加するだけで、システム全体を停止させることなく、データ保管のための容量をオンラインで拡張できます。従来のような、外部ストレージにおける複雑なファイバーチャネルの接続テストなどは一切不要です。また、x86 サーバーを追加した分だけ、処理性能が向上します。たとえば、Hadoop で構成された計算ノードの数を 3 倍にすると、ジョブの処理時間は、3 分の 1 で完了させることができます。計算ノードの CPU、メモリ、ハードディスク

の増設といったスケールアップではなく、スケールアウトによる性能向上という、非常にシンプルなデザインです。Hadoop を使用すれば、大規模なデータセット内における一部のデータのサンプルに対して、高速なクエリーを実現できるだけでなく、格納されたデータ全体に対して一度にクエリーすることができます。これにより、従来のデータ処理や分析ノウハウを駆使しても時間がかかりすぎて諦めていた**データ分析による新たな洞察の獲得**が可能になったのです。

1-1-3　Hadoop とは何か

Apache Hadoop は、データ分析や分散処理のためのオープンソースのフレームワークであり、複数の x86 サーバーでクラスターを構成し、大量のデータを格納して処理します。並列処理のアルゴリズム、高信頼、かつ、高性能なデータ処理の仕組みにより、エクサバイト級の大量のデータ処理が可能です（図 1-2）。これを実現しているのは、次の 2 つの機能です。

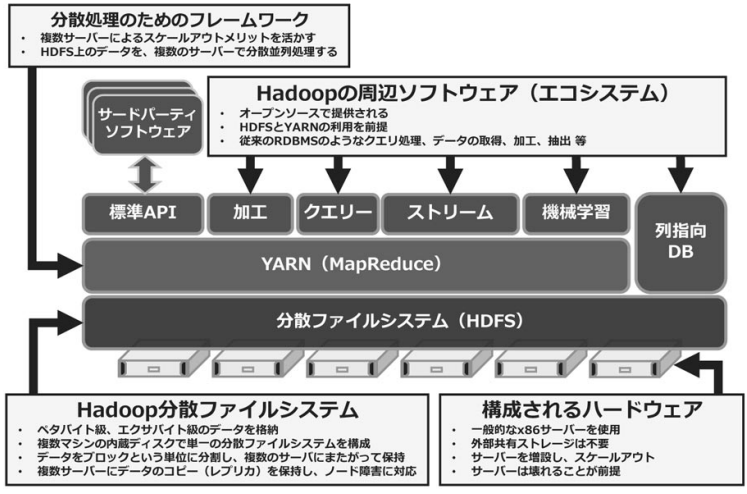


図 1-2　Hadoop の概要

●フレームワーク

1 つ目は、フレームワークの機能です。Hadoop におけるフレームワークは、x86 サーバーのクラスター構成によるスケールアウトメリットを活かし、効率的にデータ処理を行う仕組みです。このフレームワークは、Hadoop において、当初、MapReduce と呼ばれるプログラミングモデルが組み込まれていました。現在は、さらに機能拡張が施された YARN と呼ばれる

分散処理のためのフレームワークが組み込まれています。これらの機能の詳細は後述します（「1-4　Apache Hadoop 3」参照）。

●フォールトトレランス設計

2つ目は、フォールトトレランス（障害許容）設計です。Hadoopは、大量の計算ノードで大規模なデータを管理しますが、そのシステムでは、従来の基幹系業務で利用される耐障害性の高い専用サーバーではなく、一般的なx86サーバーで計算ノードを構成します。そのため、Hadoopでは一部の計算ノードが故障してもデータが消失しないように、データの冗長性が組み込まれています。この冗長性の仕組みは、ファイルシステムとして提供されており、**Hadoop分散ファイルシステム**（Hadoop Distributed File System、以下、HDFS）と呼ばれます。HDFSにおいて、データは**レプリカ**と呼ばれるコピーを持ち、複数の計算ノードに跨がって保存されることで、計算ノードに障害が発生しても、データの消失を防ぐことが可能です。

1-1-4　Hadoopで取り扱うデータ

RDBMSで取り扱っていたデータは、表形式で表されるデータで、これらのデータは、**構造化データ**と呼ばれますが、企業において利用されるデータは、構造化データだけではありません。Webサイトのアクセス記録、通話記録、音声データ、センサから発生するデータ、画像データ、TwitterやFacebookといったSNSの会話データなどさまざまです。これらの表形式では表されないデータは、**非構造化データ**と呼ばれます。

非構造化データにも、企業活動に必要な情報が眠っています。従来のRDBMSで保管されている構造化データに加え、これらの非構造化データを組み合わせて分析することで、より深い洞察を得ることが可能になるかもしれません。しかし、非構造化データは、構造化データに比べ、生成されるデータの量が比較にならないほど膨大になる傾向があります。この比較にならないほど膨大なデータを効率よく保管し、処理できる仕組みが必要です。

このような膨大なデータの取り扱いは、Hadoopが得意とするところです。Hadoopであれば、スキーマ定義が不要なため、非構造化データをHDFS上に保管し、データの増加にも容易に対応できます。また、さまざまなデータ形式を加工するための周辺ソフトウェアも充実しており、Hadoop対応の周辺ソフトウェアを利用することで、データ加工、抽出、分析の一連の処理が可能です。

表1-1は、上で述べたHadoopの特徴をまとめたものです。

表 1-1 Hadoop の特徴

項目	説明
構成される機器	一般的な x86 サーバーを使用し、RDBMS で利用されていたような高価な共有外部ストレージは不要
構成されるソフトウェア	Hadoop およびその周辺ソフトウェアもオープンソースで提供されており、高いコスト効率を実現
取り扱うデータ	従来の RDBMS で取り扱っていた構造化データだけでなく、非構造化データも格納および処理が可能
データの保管	HDFS により、事実上、無制限にデータを保存可能
分散処理	YARN（従来は、MapReduce）と呼ばれるデータ処理のためのフレームワークを装備
障害への耐性	HDFS のフォールトトレランス設計により、一部に障害が発生してもデータを消失することなく、システム全体が稼働可能

1-1-5 Hadoop の適用分野

Hadoop は、サービスプロバイダー以外にも、製造業における製品の解析、鉄道におけるモデリング、電力会社でのバッチ処理基盤、通信事業者における迷惑メールの検出、気象データの分析など、以下に示したようなさまざまな分野で利用されています。

●クリックストリームデータ分析

Web サイトを訪問した利用者のマウスクリックから、顧客の分類（すぐに購買に結び付く顧客なのか、それとも、あまり購買に結び付かない顧客なのか）、顧客行動を分析します。主に、クリックストリームログを格納、分析します。

●迷惑メールのフィルタリング

電子メールには、通常のメールと、迷惑メールに分類できますが、大量に送られてくる迷惑メールを自動的に判断するデータ処理基盤が必要となります。大量の迷惑メールのフィルタリング処理などを行います。

●センチメント分析

Twitter や Facebook といった SNS のデータから、各企業活動の動向、個人の意見や嗜好に関する情報、個人と個人の関係性などを分析し、企業や個人の関係性などを抽出します。

●センサデータの取り込み

産業機械、自動車、航空機などに組み込まれたセンサデータを取り込み、障害予測、自動制御を行います。

●非構造化データのパターン分析

人間の顔の表情を映し出した画像データや、防犯カメラの映像から不審人物の行動を分類し、犯罪行為などのパターンを特定します。

●地理データを使った最適化

巨大建造物のレイアウトの最適化、GPS データなどを組み合わせた配達経路の最適化などに利用されます。

●コールセンターオペレータの返答候補の提示

コールセンターのオペレータから収集された、顧客との過去の会話ログなどを分析し、オペレータが返答すべき文言の候補を提示します。

●ログ解析

大量に存在する Web サーバーへのアクセスログから、Web コンテンツと顧客の興味の動向に関する相関関係を見つけ出します。

1-1-6　機械学習のインフラとしての Hadoop

Hadoop の適用分野の中でも取り上げた、センサデータを取り込んだ障害予測や自動車の自動運転制御に代表されるように、近年、人工知能、機械学習といった知的情報処理技術に大きな注目が集まっています。現在の機械学習ソフトウェアは、単体サーバーで稼働させることもできますが、多くは分散処理基盤で実行できるように設計されており、Hadoop が提供する分散処理の仕組みを利用できます。機械学習では、膨大なデータを学習に利用することが多く、そのデータの保管場所として Hadoop を利用しているものもあり、知的情報処理基盤のインフラとして、Hadoop は、非常に重要な役割を担っています。

1-1-7　RDBMS vs. Hadoop

これまでのHadoopの説明では、Hadoopでどのようなデータでも処理ができそうですが、Hadoopにも不向きな処理があります。前にも述べたように、Hadoopにおける処理対象のデータは、大量のマシンに分散されて保持され、その分散されたデータを並列に処理する**分散型アーキテクチャ**を採用しています。非構造化データや単純な構造化データであれば、ペタバイト級のデータであっても高速に処理できますが、RDBMSが得意とするデータ整合性の完全な保証といったミッションクリティカルシステムで必要とされる、トランザクション管理がサポートされているわけではありません。つまり、ASID特性を有する従来のリレーショナルデータベース（RDBMS）を、完全に置き換えるものではありません。厳密なデータの整合性が保証されなければならないオンラインリアルタイム処理（銀行のオンラインバンキングシステムや座席予約システムなど）のシステムでは、Hadoopではなく、従来どおり、RDBMSで処理しなければなりません。Hadoopは、業務が稼働する本番系のRDBMSとは別のシステムとして構成し、RDBMSの構造化データをコピー、あるいは、外部から取得した非構造化データなどを格納し、大規模な分析基盤として利用されるのが一般的です（図1-3）。

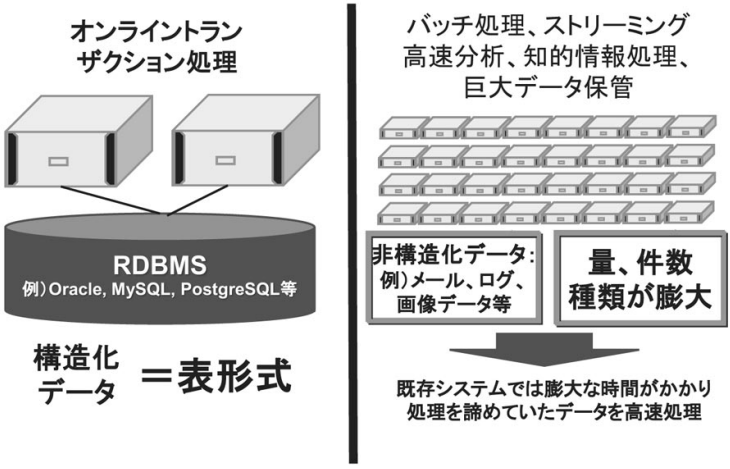


図1-3　RDBMS vs. Hadoop

1-2 Hadoop の種類を知る

Hadoop と一口にいっても、実際は、ディストリビュータが各社の付加価値を付けた形で表 1-2 に示したソフトウェア製品としてリリースしています。

表 1-2 Hadoop ディストリビュータ

提供元	製品
MapR Technologies 社	MapR CDP（MapR Converged Data Platform）
Cloudera 社	CDH（Cloudera's Distribution including Apache Hadoop）
Hortonworks 社	HDP（Hortonworks Data Platform）
Apache コミュニティ	Apache Hadoop

MapR、CDH、HDP は、それぞれの特色があり、インストール手順や、管理ツールの使い勝手、サポートされるソフトウェアの種類やバージョンなどが異なります。

1-2-1 Apache Hadoop

Apache Hadoop は、Apache コミュニティが提供する Hadoop ソフトウェアです。Apache コミュニティのトップレベルプロジェクトに位置付けられており、世界中の技術者によって開発が続いています。2017 年 12 月に、メジャーバージョンアップされた Hadoop 3.0.0 がリリースされました。基本コンポーネントは、Apache コミュニティの Web サイトから入手できますが、MapR、CDH、HDP と異なり、ベンダーの保守サポートは一切ありません*3。

Apache Hadoop は、Java ベースの Hadoop 分散ファイルシステム（HDFS）を提供しています。HDFS により、複数のサーバーを並べてスケールアウト型の 1 つの巨大データ保管庫を実現します。また、HDFS に保管しているデータを分散処理するための仕組み（YARN）が備わっており、スケールアウトメリットを活かした高速分散処理が特徴です。

Apache Hadoop の入手先：

`https://archive.apache.org/dist/hadoop/common/`

＊3　日本ヒューレット・パッカードでは、有償の Hadoop 構築サービスを提供しています。

1-2-2 MapR CDP（MapR Converged Data Platform）

米国 MapR テクノロジーズ社が提供する Hadoop ディストリビューションです。Apache Hadoop が提供する Java ベースの HDFS ではなく、C/C++によってコードを書き直した MapR-FS（MapR ファイルシステム）と呼ばれる超高速分散ファイルシステムを有する点が特徴です。特にサイズの小さいファイルが膨大に存在するようなシステムでは、MapR-FS が威力を発揮するといわれています。また、MapR では、古くから NFS サーバーの機能を提供しており、ユーザーは、Linux クライアントマシンから NFS マウントを行って、MapR-FS へのファイルのコピーや参照などを簡単に行えます。主に、高可用性を意識したアーキテクチャや、スナップショットといったエンタープライズ用途を強く意識した機能を備えたディストリビューションとして、日本でも導入実績が豊富です。MapR 社は、2018 年 1 月、MapR CDP のメジャーバージョンアップを行い、MapR 6.0 をリリースし、管理 GUI などの刷新が行われました。製品としては、無償で利用できる MapR Converged Community Edition と、有償の MapR Converged Enterprise Edition が存在します[*4]。

1-2-3 CDH（Cloudera's Distribution including Apache Hadoop）

米国 Cloudera 社が提供する Hadoop ディストリビューションです。Cloudera 社は、Hadoop の生みの親として知られる Doug Cutting（ダグ・カッティング）氏が在籍していることでも有名です。CDH は、世界中のエンタープライズシステムにおいても導入実績が豊富であり、特に、Cloudera Manager と呼ばれる洗練された GUI 管理ツールに定評があります。

CDH と Cloudera Manager を含む商用版の Cloudera Enterprise においては、保守サポートや、高度な専門知識を有した技術者によるプロフェッショナルサービス、さらには、教育トレーニングも豊富に取り揃えており、すべて日本語に対応しています。また、Cloudera 社では、CDH をベースにしたオープンソースソフトウェアに関するコミュニティ活動も活発であり、インメモリ型クエリーを実現する Impala や、HDFS のファイル群に GUI でアクセスを可能にする HUE への取り組みなどが有名です。Cloudera 社の日本人技術者による講演やイベントも定評があり、国内外を問わず、ハードウェアベンダーやシステムインテグレーターとのアライアンス強化にも精力的に取り組んでいます。

＊4 MapR のディストリビューション比較に関する情報源：
`https://mapr.com/products/mapr-distribution-editions/`

1-2-4 HDP（Hortonworks Data Platform）

米国 Hortonworks 社が提供する Hadoop ディストリビューションです。HDP は、米国 Altaba 社（旧 米国 Yahoo! Inc.）が採用しているディストリビューションとしても有名です。HDP には、オープンソースソフトウェアの Ambari（アンバリ）と呼ばれる多機能な GUI 管理ツールが用意されています。HDP のコアコンポーネントは、Apache Hadoop と完全な互換性を維持しており、米国 Hortonworks 社は、オープンソースソフトウェアのコミュニティとの関係を非常に重視した製品戦略をとっています。そのため、Hadoop コミュニティへの貢献数も非常に多いことで知られています。また、Linux だけでなく、Windows への対応を行っているのも特徴的です。

本書では、最新の機能や管理手法を知ってもらうために、コミュニティ版の Hadoop である「Apache Hadoop 3」を取り上げます。また、MapR 版 Hadoop は、C/C++で書き直されたクローズドソースの MapR-FS を搭載しており、Cloudera 版 Hadoop（CDH）や Hortonworks 版 Hadoop（HDP）に搭載されている HDFS とは異なる特徴や機能があります。特に、日本国内では、HDFS を駆使する Hadoop ユーザーだけでなく、MapR-FS を NFS 対応の高速分散ストレージの NAS として利用するユーザーも少なくありません。HDFS と MapR-FS の両方を取り扱えるようになれば、NAS および Hadoop 分析基盤としての MapR と、オープンソースの HDFS を搭載した Apache Hadoop の両方の構築や管理を実践する際の近道になるため、本書では、Apache Hadoop 3 と MapR の 2 つを取り上げます。

1-3 Hadoop の沿革

本書で取り上げる Apache Hadoop は、Hadoop 3.0、通称 Hadoop 3（Hadoop バージョン 3 系）ですが、初めて Hadoop に取り組む人のために、3.0 に至るまでの、Hadoop 1 と Hadoop 2 のアーキテクチャの変遷について、ここで紹介しておきます。これまでの経緯を理解しておくと、現在の Hadoop の仕組みが理解しやすくなります。

1-3-1 Hadoop 1 と MRv1

Hadoop クラスターは、通常、クラスターの HDFS のメタデータ情報が保管されている**マスターノード**と、実際のデータを保管、処理する**ワーカーノード**から構成されます（図 1-4）。

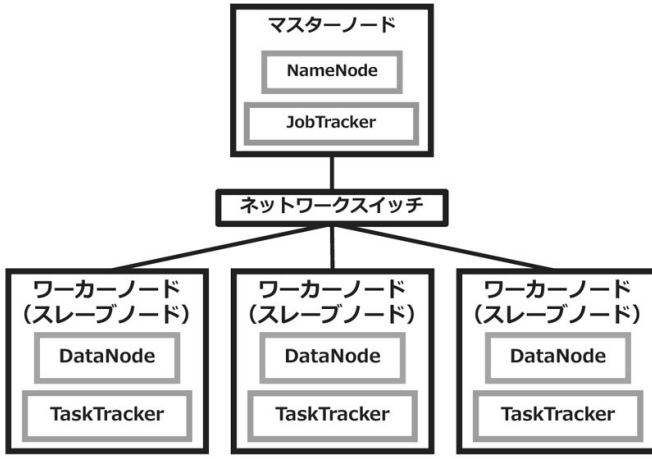


図 1-4　Hadoop 1 の主なコンポーネント

Hadoop のバージョン 1 である Hadoop 1 では、マスターノードが単一障害点となる問題がありました。このため、Hadoop 1 において、マスターノードの可用性を確保するためには、サードパーティ製の HA クラスターソフトウェアなどを導入する必要がありました。Hadoop 1 において、ユーザーのジョブを管理するプロセスは、JobTracker と呼ばれ、ワーカーノード上で実際にユーザーのジョブを実行するプロセスは、TaskTracker と呼ばれます。

Hadoop 1 の JobTracker は、ジョブのスケジューリングとユーザーアプリケーションのタスク管理、その両方を担っています。しかし、Hadoop 1 では、クラスターにおけるリソース管理とジョブのスケジューリングの両方を管理する JobTracker の負荷が高くなる問題を抱えていました。Hadoop 1 自体は、当初の設計思想のとおり、スケールアウトメリットを活かすことが可能でしたが、約 5000 ノードまでしかスケールさせることができませんでした。

Hadoop 1 におけるデータ処理エンジン（分散処理フレームワーク）は、MapReduce でした（Hadoop 2 の YARN と区別するために、MapReduce バージョン 1、あるいは、MRv1 と呼ばれます）。MapReduce の API を使えば、開発者は、数千ノードに及ぶ大規模な Hadoop クラスターに格納されたデータの並列処理が可能なプログラムを作成できます。さらに、データ容量やデータの個数が増えても、計算ノードを追加すれば、リニアにスケールアウトする処理能力を有しています。

MapReduce は、一連のタスクをより小さな単位のタスクに分割し、Map 処理と Reduce 処理の 2 つのフェーズで構成されています（図 1-5）。

Map 処理を行うプログラムは、Mapper と呼ばれ、Reduce 処理を行うプログラムは、Reducer と呼ばれます。具体的には、Map 処理において、入力データをキー（Key）と値（Value）のペアのセッ

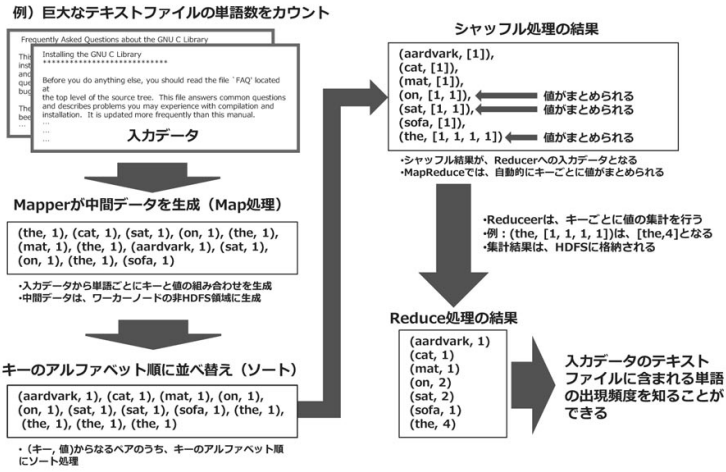


図 1-5　MapReduce の仕組み

トにマップします。たとえば、テキストファイルに含まれる単語の出現頻度を算出する MapReduce プログラムにおける Map 処理では、入力データとなるテキストファイルの文章が「the cat sat on the mat the aardvark sat on the sofa」とすると、Mapper は、以下のような、（キー、値）からなる中間データを生成します。

```
(the, 1), (cat, 1), (sat, 1), (on, 1), (the, 1), (mat, 1), (the, 1), (aardvark, 1), (sat, 1), (on, 1), (the, 1), (sofa, 1)
```

この中間データは、キーのアルファベット順にソートされて、以下のようなデータが生成されます。

```
(aardvark, 1), (cat, 1), (mat, 1), (on, 1), (on, 1), (sat, 1), (sat, 1), (sofa, 1), (the, 1), (the, 1), (the, 1), (the, 1)
```

キーがアルファベット順に並んでいますので、キーごとに値をまとめられます。この一連のまとめる作業は、MapReduce が自動的に処理します。

```
(aardvark, [1]), (cat, [1]), (mat, [1]), (on, [1, 1]), (sat, [1, 1]), (sofa, [1]), (the, [1, 1, 1, 1])
```

最後に、このデータが Reducer に送られ、キーごとに、値が集計され、単語の出現頻度が算出されます。

```
(aardvark, 1), (cat, 1), (mat, 1), (on, 2), (sat, 2), (sofa, 1), (the, 4)
```

この MapReduce 用のプログラムにおいて、非常に重要なのは、開発者がプログラム内にスケジューリングやリソース管理、障害処理などのロジックを組み込む必要がない点です。上で示した分散処理フレームワーク MapReduce の仕組みのおかげで、Hadoop 登場以前に比べ、並列分散アプリケーションの作成が非常に簡単になりました。しかし、MapReduce が Hadoop アプリケーションの処理エンジンとして特化したものであったため、汎用性に欠ける問題もはらんでいました。

1-3-2 Hadoop 2 と YARN

Hadoop 2 では、今まで JobTracker が担っていたリソース管理とジョブスケジューリングの 2 つの主要な機能を分離し、JobTracker の代わりにクラスター全体のスケジューリングを行う ResourceManager（マスターノードで稼働）とリソース管理を行う ApplicationManager が用意されました。また、TaskTracker の代わりに、データ処理のタスクを実行する NodeManager（ワーカーノードで実行）が用意されました。この ResourceManager と NodeManager の仕組みは、Hadoop 3 にも受け継がれており、現在の Hadoop クラスターの主要コンポーネントとなっています。Hadoop 2 のワーカーノードで稼働する NodeManager は、実際にワーカーノードで実行されるプロセスを監視するエージェントとしても機能します。

Hadoop 2 では、MapReduce だけでなく、Apache Spark や、Apache Tez など、さまざまな分散処理のエンジンが稼働し、Hadoop 2 の YARN（Yet Another Resource Negotiator）† と呼ばれる非常に汎用性の高い、分散処理のフレームワークの仕組みが提供されました。

> † MRv1 の次世代バージョンという意味で、MapReduce バージョン 2、MapReduce 2.0、MRv2 とも呼ばれます。

MapReduce が、Hadoop 1 におけるバッチ処理の唯一の分散処理の仕組みだったのに対し、YARN は、バッチ処理、インタラクティブ処理、リアルタイム処理が可能な分散処理の仕組みや、アプリケーションワークロードの管理や監視、さらには、Hadoop-as-a-Service を見据えた、マルチテナント環境のジョブスケジューリング機能を提供しました。

Hadoop 2 の YARN は、その後、Hadoop 3 の分散処理フレームワークとして受け継がれています。Hadoop 1 での MapReduce から Hadoop 2 の YARN になり、柔軟性に富んだ分散処理フレームワークの利用が実現し、Hadoop の利用範囲が格段に広がりました。また、スケール性能も向上し、約 1 万ノードまでスケールできるようになりました。

Hadoop 2 の YARN では、バッチ処理以外のさまざまな処理を提供するフレームワークとして利用できます。**表 1-3** に、MRv1 から YARN に変更することにより、恩恵を受けるデータ処理の例を示します。

表 1-3　YARN によるメリット

処理の種類	YARN の恩恵を受ける処理	Hadoop の周辺ソフトウェアの例
データの取得	データストリーミングにおけるリアルタイムなデータの収集	Kafka、Flume、Spark Streaming など
データの操作	インタラクティブな SQL によるデータ処理	Tez を使った Hive、Impala、Spark SQL など
データの検索	非構造化データと構造化データの両方においけるデータの全文検索	Solr など
反復計算	反復処理が頻繁に実行される機械学習、および深層学習	Mahout、Spark MLlib など

1-3-3　Hadoop のエコシステム

図 1-2「Hadoop の概要」（p.14）に示したように、Hadoop は、HDFS と YARN から構成された、分散処理エンジンを提供する基盤ソフトウェアです。Hadoop 自体は、YARN によってユーザーアプリケーションの分散処理を実現しますが、さまざまなユーザーの分析業務に対応するために、その分析業務ごとに Hadoop で稼働する周辺ソフトウェアが用意されています。この周辺ソフトウェアは、Hadoop エコシステムソフトウェア、または、Hadoop エコシステム、あるいは、単に**エコシステム**と呼ばれます。

Hadoop エコシステムソフトウェアの主要コンポーネントで興味深いのは、そのコンポーネントの多くが HDFS と YARN のフレームワークで稼働するという点です。エコシステムは、分析業務における取り込み（収集）、加工、抽出、学習、結果の推定、そして可視化といった一連のデータ処理の各段階で別々のソフトウェアを駆使します。これらのソフトウェアは、Hadoop の稼働を前提としたエコシステムソフトウェアで構成されます。もちろん、Hadoop に対応したサードパーティ製の優れたソフトウェアも存在するため、通常は、Hadoop ディストリビュータが提供するエコシステムのパッケージとサードパーティ製のソフトウェアを組み合わせて利用するケースがほとんどです。**表 1-4** では、ビッグデータ分析基盤で利用される、Hadoop の基本的なエコシステムのうち、本書で扱うものをピックアップしています。

表 1-4　本書で取り上げるエコシステム

ソフトウェア	主な役割	説明
Sqoop	データの収集	RDBMS 上の構造化データの Hadoop 基盤への取り込み（インポート）、または、その逆（エクスポート）を行う
Flume	データのストリーム処理	複数拠点から絶え間なく生成される非構造化データの Hadoop 基盤への取り込み（データストリーム処理）を行う
Pig	データの加工	分析前のデータの加工を行う ETL（Extract、Transform、Load）ツール、スクリプト言語
Hive	データへのクエリー処理	RDBMS の SQL に似たクエリー言語（HiveQL）でデータの抽出、問い合わせを行う DWH 環境を実現する分散クエリーエンジン
Impala	データへのクエリー処理	RDBMS の SQL に似たクエリー言語（Impala SQL）でデータの抽出、問い合わせを行う高速分散クエリーエンジン
HBase	データの格納	永続性、行単位ロック、冗長化など、複雑なデータ操作が可能なビッグデータ用 KVS（Key Value Store）型の NoSQL（非リレーショナル・データベース）
Mahout	学習、推定	線形代数、統計解析、機械学習のライブラリを提供する
Spark	収集、クエリー処理、学習、推定	データストリーム処理、機械学習、SQL 操作、統計で利用される R 言語、グラフ処理が可能なインメモリ型の分散処理フレームワーク

その他にも、本書では取り上げませんが、**表 1-5** に示すようなさまざまなエコシステムが Hadoop で利用されます。

表 1-5　Hadoop を取り巻くソフトウェア

ソフトウェア	主な役割	説明
Accumulo	データの格納	米国国家安全保障局（NSA）が開発し、オープンソース化された分散型 KVS（Key Value Store）
Atlas	統制、監査	Hadoop 内、まはた外部のソフトウェアとメタデータを交換し、企業コンプライアンス要件に対応するためのデータガバナンス（統制）を実現する
Cascading	アプリケーションフレームワーク	Java 開発者、および、データエンジニア向けのデータ処理用 API を提供する
Drill	データへのクエリー処理	大規模データセットに対応し、数千台規模でスケールが可能な低遅延の分散クエリーエンジン

Druid	データの保管	リアルタイム分析に利用される BI（Business Intelligence）向けのクエリー用のデータストア
Falcon	ライスサイクル管理、統制、監査	データライフサイクルを管理するためのフレームワーク、コンプライアンス、監査、データベースの複製を行う
Hue	GUI の提供	HDFS へのアクセスや、Hadoop のエコシステムソフトウェアである Hive、Pig などの GUI を提供
Kafka	データの収集、メトリック収集、監視	データの収集、データストリーム処理を行う。IoT 機器から絶え間なく生成されるセンサーデータなどの取り込みにも利用される
Knox	認証、アクセスの集中管理	ユーザーに集中管理型の認証やアクセスを提供し、Hadoop セキュリティを簡素化する
Oozie	スケジューラー	ユーザーの Hadoop ジョブを管理するためのワークフローをスケジューリングする
Phoenix	データの保管	リアルタイムアクセスが可能な高速リレーショナルデータベースであり、HBase をデータストアとして利用する
Ranger	アクセス制御	HDFS のファイル、フォルダー、データベース、テーブルなどのアクセ制御のフレームワークを提供する
Sentry	アクセス制御	認証済みのユーザーやアプリケーションへのアクセスレベルを提供し、ロールベースの権限を付与するためのフレームワークを提供する
Slider	アプリケーション実行制御、管理	長時間実行される YARN アプリケーションのリソースの割り当て管理、スケールの調整、再起動などを行う
Solr	データの検索	HDFS に格納された非構造化データに対する全文検索を行い、索引を作成する
Tez	分散処理フレームワーク	データフローを定義するための API を提供し、MapReduce よりも効率的な分散処理の仕組みを Hive、Pig、Cascading などに提供する

1-4 Apache Hadoop 3

Apache Hadoop 3 は、複数のサービスが連携して動作することで、クラスターを形成します。以下では、Apache Hadoop 3 を構成する 2 つのコンポーネントについて解説します。

●HDFS

1 つ目のコンポーネントは、分散ファイルシステムである HDFS です。HDFS は、複数のサーバーで構成された、仮想的な 1 つのファイルシステムであり、データは、複数のサーバーにブロックという単位で分割して記録されます。

●YARN

もう1つは、分散処理フレームワークのYARN（Yet Another Resource Negotiator）です。YARNは、HDFS上にあるデータの分散処理を行う汎用のフレームワークです。このYARNにより、HDFS上のデータを加工し高速に処理することができます。Hadoopクラスターにおいて、アプリケーションの実行は、YARNを使って行いますが、その際、アプリケーションは、ジョブとして投入されます。ジョブとして投入されるYARNを使った分散処理のアプリケーションは、**YARNアプリケーション**や、**分散アプリケーション**と呼ばれます。

1-4-1 Apache Hadoopの一般的な構成

一般的なHadoopクラスターのシステム構成は、Hadoopクラスターのメタデータを保管し、クラスター全体の調整役を行う**マスターノード**と、ユーザーデータの保管と分散処理を担当する**ワーカーノード**（データがたまるノードのため、データノード、あるいは、スレーブノードとも呼ばれます）に分けられます。さらに、Hadoopクラスターを構成する物理サーバーの設定、OSのインストール、負荷の監視などを行う管理ノード、ユーザーがHadoopのジョブを投入するクライアントノードが存在します[*5]。

■ マスターノードで稼働するサービス

Hadoopのマスターノードでは、NameNodeサービス（以下、NN）とResourceManagerサービス（以下、RM）が稼働します。

●NNサービス

HDFSのメタデータの保持や、HDFSを提供するDataNodeサービス（以下、DN）の状態を管理します。

●RMサービス

RMサービスは、利用可能な計算リソースや、ワーカーノードで稼働しているNMサービスをトラッキングし、計算リソースの割り当てなどを行います。さらに、RMサービスは、アプリケーションごとに起動するApplicationMaster（以下、AM）と呼ばれるプログラムの監視などを行います。

マスターノードでは、他にも、ジョブの履歴情報を保持するJobHistoryServerサービス（以下、

＊5 管理ノード上でHadoopの管理を行いますが、ジョブの投入を行うクライアントマシンと管理ノードを兼用する場合もあります。

JHS）が存在します。NN サービス、RM サービス、JHS サービスは、マスターノード上で常駐サービスとして稼働します。

■ ワーカーノードで稼働するサービス

ワーカーノードでは、DataNode サービス（以下、DN）と NM サービスが常駐サービスとして稼働します。図 1-6 は、Apache Hadoop 3 クラスターを構成する、各コンポーネントの関係をまとめたものです。

●DN サービス

DN サービスによって、HDFS 上のデータをユーザーに提供します。HDFS には、ユーザーデータの元データと、そのレプリカ（コピー）がノードをまたいでブロックに分割されて記録されます。そのため、ワーカーノードに障害が発生しても、他のワーカーノードに保持されたレプリカが存在する限り、HDFS 上にあるデータにアクセスできます。また、既存の Hadoop クラスターにワーカーノードをオンラインで追加することができ、業務を止めずに HDFS を拡張できます。

●NM サービス

NM サービスは、ワーカーノード上で実際にタスク（アプリケーションやジョブ）を実行します。

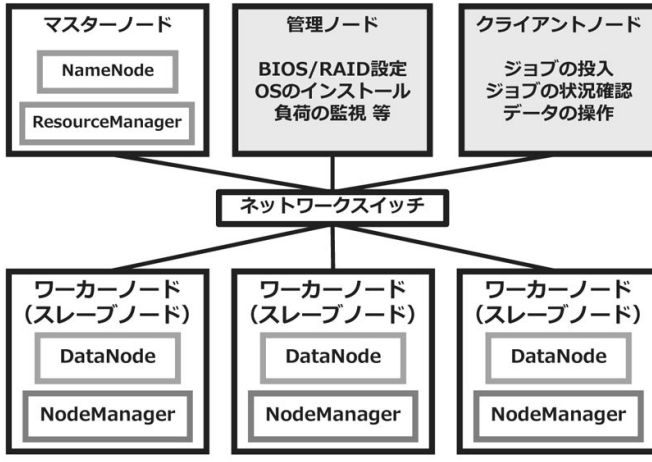


図 1-6　Apache Hadoop 3 クラスターを構成する主なコンポーネント

Column　Apache Hadoop における YARN の処理手順

通常、アプリケーションの実行というと、Windows であれば、exe ファイルをダブルクリックする、Linux であれば、実行ファイルをコマンドラインから入力して実行することを思い浮かべます。しかし、Hadoop の世界では、直接アプリケーションの実行ファイルを起動させるのではなく、アプリケーションを分散処理のためのフレームワーク「YARN」の仕組みを使って、ジョブ投入という形で実行します。そのため、アプリケーションが実際にワーカーノード上で稼働するまでには、さまざまなサービスがかかわります。図 1-7 は、YARN によるアプリケーション処理の流れを示しています。

図 1-7　Apache Hadoop における YARN の処理手順

クライアントノードからアプリケーションの実行依頼が発生すると、その依頼は、マスターノード上の RM サービスが受け取ります。RM サービスは、直接 NM サービスにアプリケーションの起動の依頼を行うのではなく、AM サービスを経由して依頼します。AM サービスは、アプリケーションの起動依頼ごとに生成される非常に軽量なプログラムです。AM サービスは、RM サービスと実際にアプリケーションの実行を手掛ける NM サービスの間に介在します。

・RM サービスは、まず、AM サービスを起動するワーカーノードを選びます（図中の①）。選ばれたワーカーノードで AM サービスを起動させるためには、その起動に必要な計算リソースが空いていることが前提条件です。ここで、計算リソースとは、CPU 資源やメモリ資源を意味し

ます。この計算リソースは、Hadoopにおいて、YARNコンテナ（以下、YC）と呼ばれます。コンテナという名前が付いていますが、コンテナ管理ソフトウェアのDockerとは無関係です。

・RMサービスは、AMサービスを起動するのに必要なYCをNMサービスに提供します（図中の②）。
・RMサービスからYCを受け取ったNMサービスは、AMサービスを起動します（図中の③）。
・AMサービスが起動すると、アプリケーションの実行に必要な計算リソース（YC）をRMサービスに要求します（図中の④）。
・アプリケーションに必要なYCがワーカーノードに割り当てられて、アプリケーションの分散処理が行える環境が整います（図中の⑤）。
・YCが割り当てられたAMサービスは、NMサービスに対して、与えられた計算リソースを使ってアプリケーションの起動を命じます。YCを得たNMサービスは、アプリケーション（図中の「APP」）を実行します（図中の⑥）。ここで稼働するアプリケーションは、タスクという単位で管理されます。アプリケーション実行中においても、AMサービスは、NMサービスに対して、アプリケーションの実行状況を確認し、それに応じて、NMサービスは、AMサービスに対して、アプリケーションの実行状況を逐一報告します。ここで、NMサービスは、YC内の計算リソースの使用量を監視しています。もし、YC内の計算リソースの使用量が当初に割り当てられた量を超過すると、YC自体をkillするなどの処理が行われます。一方、RMサービスも、AMサービスの稼働状況を確認しています。
・RMサービスは、アプリケーションのタスクについては一切監視せず、AMサービスが正常に稼働しているかどうかをチェックしています（⑦）。

AMサービスは、NMサービスでアプリケーションが稼働できるように、YCをRMサービスに随時要求する場合があります。その場合は、RMサービスがYCを追加で付与することで、アプリケーションの実行に必要な計算リソースを拡張させます。アプリケーションが終了すると、AMサービスは、YCを手放し、AMサービス自身も終了します。したがって、AMサービスは、アプリケーションが起動している間、一時的に起動するため、RMサービスやNMサービスと異なり、ワーカーノード上で常駐することはありません。

1-4-2 Apache Hadoop 3の新機能

2017年12月にメジャーバージョンアップされたApache Hadoop 3.0.0（以下、Apache Hadoop 3、あるいは、Hadoop 3系と表記します）には、従来にはない改良や新機能が搭載されています。ここでは、Apache Hadoop 3の新機能について、いくつかピックアップして解説します。

■ 3 台以上の NameNode のサポート

Hadoop クラスターは、大きく分けて、Hadoop クラスター全体のメタデータを処理するマスターノード（NameNode）と HDFS を提供するワーカーノード（DataNode）に分けられますが、DataNode だけでなく、Hadoop クラスター全体の耐障害性を高めるために、NameNode を複数台用意することがあります。Apache Hadoop 2 では、現用系の NameNode が 1 台と待機系の NameNode が 1 台の合計 2 台で構成されていましたが、Apache Hadoop 3 から、3 台以上の NameNode をサポートするようになりました。

■ イレイジャーコーディング

Apache Hadoop 2 で採用されている一般的な HDFS では、データを 3 重化するレプリケーション・スキームが使用されます。しかし、3 重化のレプリケーション・スキームは、元のデータに対して、3 倍のストレージ空間、すなわち、200 %の冗長化のオーバーヘッドが発生してしまいます。元のデータに対して、200 %の冗長化オーバーヘッドは、ストレージ効率が元の 3 分の 1、すなわち、33%しかありませんので、決して効率がよいとはいえません（図 1-8)

。

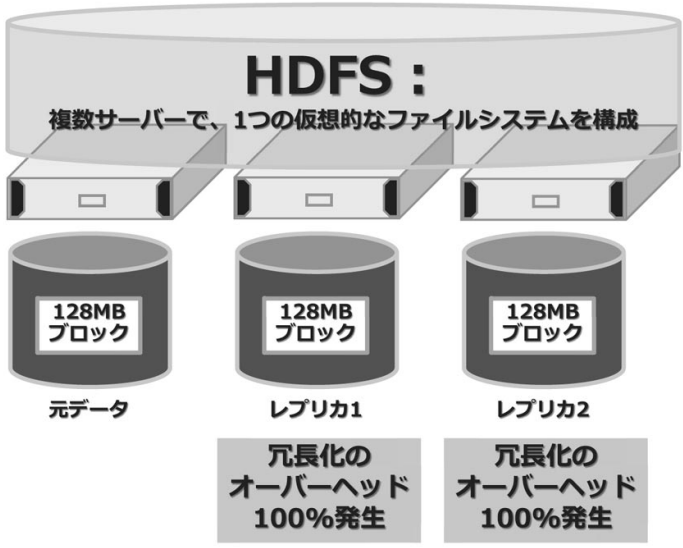


図 1-8　Apache Hadoop 2 で採用された HDFS の一般的なレプリケーション

この冗長化のオーバーヘッドの問題を解決するため、Apache Hadoop 3 では、イレイジャーコーディング（以下、EC）と呼ばれる冗長化の仕組みが実装されました。EC では、一般に冗長化のオーバーヘッドが 40 %～50 %しかなく、レプリケーション・スキームと比較して、約 2 倍のデータ量を格納できます。EC は、RAID を含む他のストレージソリューションでも使用されており、高可用性を提供しつつも、ストレージスペース効率を高めます。

EC は、DataNode に格納されるデータブロックに対して、パリティデータを生成します。元のデータブロックに障害が発生した場合、パリティデータを用いて元データを復元します。どのようにパリティデータを作成するかといった符号化のポリシーが複数存在し、HDFS の特定のディレクトリに対して符号化のポリシーを割り当てることが可能です。Hadoop クラスターでは、リードソロモン（以下、RS）と呼ばれる符号化が有名です。RS は、線形代数演算を使用して複数のパリティブロックを生成し、複数個のデータブロックの障害に対応できる仕組みを提供します。たとえば、6 個のデータブロックを符号化するために 3 個のパリティブロックを生成し、3 個のブロックの故障に対して保護を提供する場合は、RS(6,3) 方式と呼ばれます（**表 1-6**）。

表 1-6　RS のデータブロックと利用効率

	元データのみ	レプリカ（3 重化）	RS(6,3)	RS(10,4)
障害ブロックの最大許容数	0	2	3	4
ストレージ利用効率	100%	33%	67%	71%

RS(6,3) の場合、データブロック 6 個、パリティ 3 個から構成され、この 9 個のブロックをまとめたものを**ブロックグループ**と呼びます。同様に、RS(10,4) の場合は、データブロック 10 個、パリティ 4 個から構成されます。データブロックとパリティの比率によってストレージ効率と障害ブロックの最大許容数が異なります。

データブロックを k 個、パリティブロックを m 個とした場合、リードソロモン採用時におけるストレージ効率は、以下の式で求められます（**図 1-9**）。

k / (k + m)

たとえば、データブロックが 6 個、パリティブロックが 3 個の場合、

6 / (6 + 3) = 6/9 = 0.66666... = 約 67%

となります。

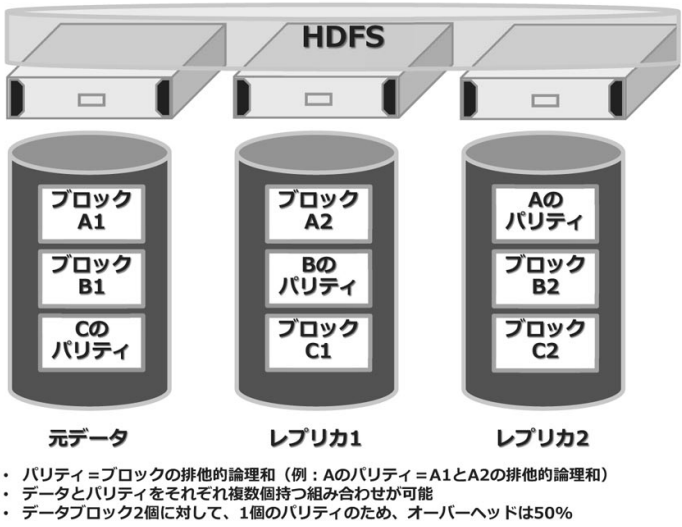


図 1-9　Apache Hadoop 3 における HDFS のイレイジャーコーディング

■ 環境変数の定義

Apache Hadoop 3 では、環境変数の定義場所が、`hadoop-env.sh` スクリプトに集約されました。

■ サービスの起動・停止手順

各種サービスの起動や停止が、`hdfs` コマンドに「`--daemon`」オプションを付与して行うようになりました。

■ ポート番号の刷新

Hadoop クラスターで起動する各種サービスのポート番号も変更されました。**表 1-7** に、Apache Hadoop 2 と Apache Hadoop 3 で利用される、ポート番号の代表的なものをまとめておきます。

表 1-7　Apache Hadoop 2 と Apache Hadoop 3 で利用されるポート番号

デーモンの種類	アプリケーションの種類	Apace Hadoop 2	Apache Hadoop 3
NameNode	HDFS NameNode RPC	8020	8020‡
	HDFS NameNode HTTP UI	50070	9870
	HDFS NameNode HTTPS UI	50470	9871
セカンダリ	セカンダリ NameNode HTTP UI	50090	9868
NameNode	セカンダリ NameNode HTTPS UI	50091	9869
DataNode	HDFS DataNode RPC	50010	9866
	HDFS DataNode IPC	50020	9867
	HDFS DataNode HTTP UI	50075	9864
	HDFS DataNode HTTPS UI	50475	9865

‡　Hadoop 3.0.0 では、NameNode の RPC に関するポート番号は、9820 番でしたが、Hadoop 3.0.1 から再び、8020 番になりました。

■ Java バージョンが 7 から 8 へ変更

Apache Hadoop 2 では、Java 7 で Hadoop のコアコンポーネントと、それを取り巻くエコシステムのアプリケーションが稼働していましたが、Apache Hadoop 3 では、提供されている JAR ファイルが Java 8 を対象にしているため、Java 8 が必要になりました。

■ DataNode 内のディスク間再バランス機能

Hadoop クラスターを構成する DataNode では、通常、内蔵ディスクが複数個で構成されます。しかし、障害によるディスクの交換や、大量の I/O の発生などにより、DataNode 内の内蔵ディスクの使用量が不均一になることがあります。この不均一を解消するため、Apache Hadoop 3 では、DataNode 内の複数の内蔵ディスク間で再バランスする機能が追加されました（図 1-10）。

なお、DataNode 内の複数ディスク間での再バランス† は、Hadoop 3 から搭載された機能ですが、Cloudera 版 Hadoop（CDH）のバージョン 5.8.2 でバックポートされています。

> †　以下の URL に DataNode 内の複数ディスク間での再バランスを行うための HDFS のパッチ情報が掲載されています。
> https://issues.apache.org/jira/browse/HDFS-1312
> CDH 5.8.2 にバックポートされた、DataNode 内の複数ディスク間での再バランスに関する情報：
> http://blog.cloudera.com/blog/2016/10/how-to-use-the-new-hdfs-intra-datanode-disk-balancer-in-apache-hadoop/

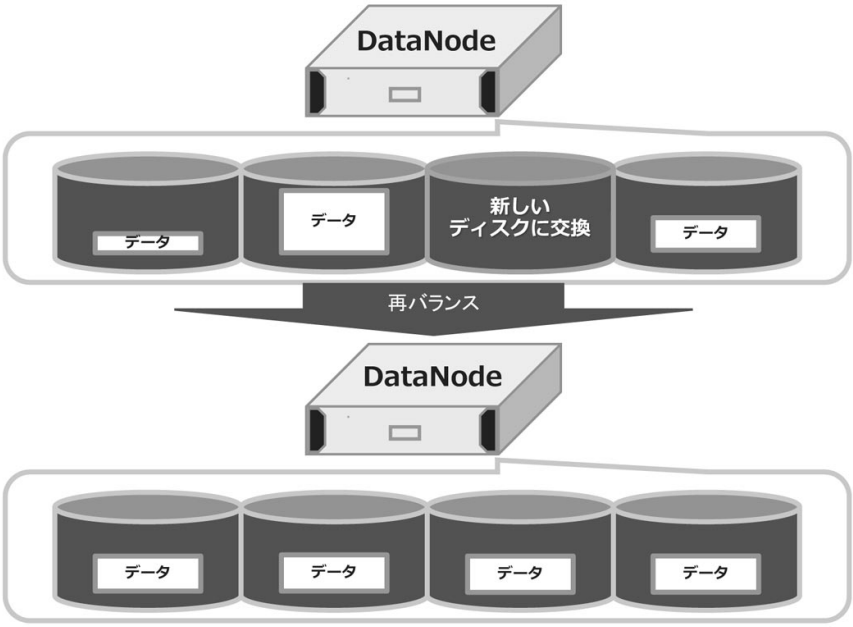


図 1-10　Apache Hadoop 3 における DataNode 内のディスクの再バランス

■ GPU への対応

最近の人工知能、機械学習のアプリケーションは、GPU を利用することで性能を発揮するものが多いため、Hadoop の YARN におけるリソース管理においても GPU への対応が強く求められていました。こうした要望を受け、Hadoop 3 ではスケジューリング可能な YARN のリソース対象として、GPU が追加されています。これにより、従来の CPU を使用するアプリケーションだけでなく、CPU と GPU を両方使用するアプリケーションを Hadoop クラスター上で稼働できるようになりました†（図 1-11）。

† 2018 年 4 月時点の Hadoop 3.1.0 の YARN は、NVIDIA 社製の GPU のみがサポートされています。また、NodeManager サービスが稼働するワーカーノードには、あらかじめ NVIDIA 社製のドライバーをインストールしておく必要があります。

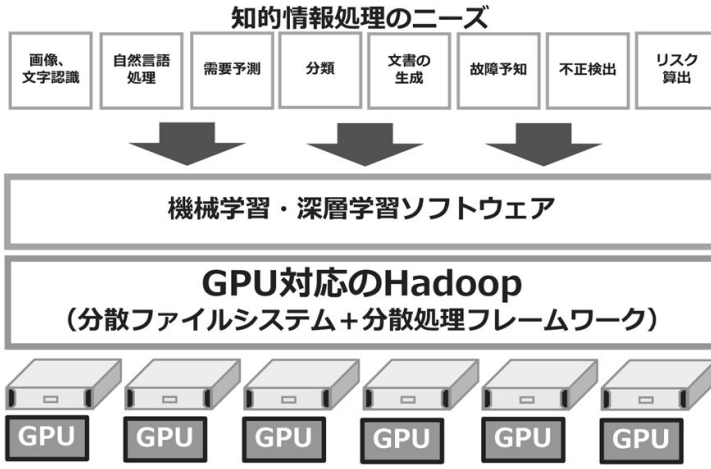


図 1-11　知的情報処理基盤としての Hadoop

Column　機械学習と GPU

過去の機械学習では、主にマルチコアの強力な CPU パワーを使って計算を行っていましたが、近年は、GPU を使った機械学習が処理が増えてきました。もともとは、画像を映し出すグラフィックボードの画像処理エンジンとして、高速に行列演算処理を行うプロセッサです。

一般に、機械学習の計算では行列計算が多用されます。特に、人間の脳の神経細胞を模倣したニューラルネットワークを駆使した「深層学習」と呼ばれる機械学習の計算では、膨大な行列演算が必要になります。GPU の行列計算が得意である特性を活かすことにより、CPU 単体で処理する場合に比べ、格段に高速に機械学習の処理を行うことが可能です。そのため、近年の機械学習基盤では、計算ノードにグラフィックボードを搭載し、GPU 対応の機械学習ソフトウェアを稼働させるようになりました。Hadoop での GPU 対応も、こうしたニーズを反映したものです。

最近では、NVIDIA 社が提供する Pascal P100、Tesla K80、Tesla M40、Tesla K40、AMD 社の FirePro S9150 などが機械学習の計算用 GPU ボードとして選択されます。これらの GPU ボードは、サーバーハードウェアベンダーによって正式に動作サポート情報が提供されています。

表 1-8 に、Apache Hadoop 2 と Apache Hadoop 3 の主な特徴に関する簡単な比較を掲載しておきます[*6]。

＊ 6　Hadoop のディストリビューションによっては、Apache Hadoop 3 の機能がバックポートされているものもあります。

表 1-8　Apache Hadoop 2 と Apache Hadoop 3 の主な特徴比較

比較項目	Apache Hadoop 2	Apache Hadoop 3
ライセンス	Apache 2.0	Apache 2.0
Java のバージョン	Java 6　または、Java 7	Java 8
障害への対応	レプリカ	レプリカ、EC
ディスクの平準化	DataNode 間	DataNode 間、DataNode 内ディスク間
冗長化のオーバーヘッド	3 重化では、200%	EC 使用時は、50%程度
YARN Timeline Service	スケーラビリティに問題あり	スケーラビリティと信頼性が向上
NameNode の数	現用系 1 台、待機系 1 台	3 台以上をサポート
タスクのヒープサイズ指定	複数の設定が必要	設定が簡素になり自動調整が可能
スケーラビリティ	約 1 万台	約 1 万台。スケール性能が向上
資源割り当て、分散処理の仕組み	YARN	YARN
スケジューリング可能な YARN リソース	CPU、メモリ	CPU、メモリ、GPU
アプリを Docker コンテナで実行	対応	対応

1-5　まとめ

以上で、Hadoop の歴史、概要、ディストリビューション、最新の状況などについて簡単に解説しました。Hadoop は、バージョン 1 からバージョン 2 で大きくそのアーキテクチャが変化し、柔軟性の高い YARN の仕組みのおかげで、さまざまな分野で利用が進みました。そして Hadoop のバージョン 3 では、GPU による人工知能・機械学習アプリケーションの超高速処理を見据えた、ビッグデータ分析基盤ソフトウェアとして進化しています。

単なる「巨大データの保管庫」という機能だけでなく、スケールアウトのアーキテクチャを活かした高速分散並列処理のメリットを享受できるのが Hadoop です。従来の RDBMS やデータウェアハウス基盤において時間が掛かりすぎて諦めていた膨大なデータ処理業務は、Hadoop とそれを取り巻くエコシステムによって高速に処理が可能です。ビッグデータから得られる新しい知見・洞察を企業や組織の競争力にするためにも、是非、Hadoop 基盤とそれを取り巻くエコシステムの導入を検討してみてください。

第２章
Hadoop のシステム構成

Hadoop に限らず、一般にビッグデータ分析基盤の導入においては、データ量だけでなく、さまざまな視点での検討が必要です。自社の問題点、課題、適用範囲、導入目的、学習コスト、人的リソース、予算などを明確にする必要があります。そして導入が決まれば、ビッグデータ分析基盤の鳥瞰図だけでなく、具体的なシステム構成も検討しなければなりません。

本章では、最初に Hadoop を使ったビッグデータ分析基盤の導入前の主だった検討項目を洗い出し、その後で、Apache Hadoop 3 と MapR 6.0 を対象にした、小規模から大規模に至る具体的なシステム構成例を紹介します。また、テスト環境や短期的な利用の場合を考えて、パブリッククラウドサービスでの対応状況や、プライベートクラウドでの Hadoop の利用についても簡単に紹介します。

2-1 Hadoop 導入前の検討

Hadoop は、非常に大規模なデータを高速処理し、従来に比べて初期投資を抑えつつも高い価値を得るためのフレームワークとして活用されています。初めて Hadoop を導入する企業では、比較的簡単な集計処理、分類、データ保管庫の基盤として試験的に導入することから始めるケースも多く見られます。こうした企業が小さな知見の獲得とデータ分析ノウハウの蓄積により、さらに Hadoop の活用を推し進めることで、企業の事業推進エンジンとして、より大きな価値をもたらすことも期待できます。しかし、Hadoop 基盤の導入以前に、組織が抱える課題も多く存在しており、その課題を克服しなければ、ビッグデータ分析基盤導入後に得られる効果も薄れてしまいます。Hadoop 基盤を導入する前に、まずは、以下に述べるような、組織の課題を明確にしておく必要があります。

2-1-1 ビッグデータ分析基盤導入前の検討項目

企業にとっての目的は、ビッグデータ分析から知見を得ることであり、Hadoop 基盤や分析技術の導入は、その手段でしかありません。ビッグデータ分析基盤を導入する前に、まず、自社の課題、導入目的を明確にしておく必要があります。

■ Hadoop 導入前の具体的な対応

Hadoop の導入前には、さまざまな視点で自社の問題点を洗い出す必要があります。ビッグデータの導入を検討する場合は、まず、IT ストラテジストや IT アーキテクトとブレーンストーミングを行い、話が発散してもかまわないので、できるかぎり多くの自社の経営上の問題点（あるいは、事業部単位での問題点）、課題などを書き出します。具体的には、**表 2-1** のように、自社の問題点と課題を関係者で共有します。与えられた課題が明確になれば、そこから自社システムにおける「あるべき姿」を描きます。あるべき姿が明確になれば、ベンダーのビッグデータに精通したコンサルタントやデータサイエンティスト、データアナリストと共に、あるべき姿を実現するための現時点で適用可能と思われる技術を書き出すワークショップなどを開催します。この時点で、ハードウェアベンダーやシステムインテグレーター側と慎重に協議し、必要なハードウェア、および、ソフトウェアコンポーネントを明確化します。ここで、検討する各項目は、分析業務における一連の流れとして、データの特定と取り込み（収集）、加工、抽出、学習、結果の推定、そして可視化の順に考えます。

表 2-1　検討事項

自社の問題点	与えられた課題	あるべき姿	適用可能な技術	ソフトウェア例
各拠点で生成されるログが膨大でデータを捨てている	大規模ログ分析基盤の整備	全ログデータをリアルタイムに収集・分析する	データストリーム	Flume Spark Streaming
取り扱うデータが増え、基盤増設で業務が停止する	大容量データ保管庫の整備	データ増加に伴い、無停止で容量を拡張できる	分散ファイルシステム	HDFS、MapR-FS
商品の売り上げの集計に時間がかかる	高性能なデータ検索基盤の整備	高速にデータを抽出・検索する	分散クエリー	Hive、Impala
顧客への適切な回答に時間がかかる	想定問答集の整備	リアルタイムに顧客に妥当な回答を行う	深層学習	Mahout、Spark
機器がいつ故障するのか見当がつかない	障害発生前の予知	故障前に予兆を察知し、故障確率を通知する	深層学習	Mahout、Spark
分析結果が分かりづらく、経営層への説明に時間がかかる	報告書作成の工数削減、見える化	リアルタイムに分析し、結果を簡潔なグラフで表示する	データの可視化	Pentaho

■ データサイエンティストの確保

Hadoop は、開発者やデータサイエンティストなどから支持を得て、スピード感のある開発と高速分析が行える基盤ソフトウェアとして主要な先進企業で利用されてきました。そこでは、散在する多種多様なデータと新しいアイデアを結び付け、競争力のある製品開発やサービスを生み出してきています。しかし、多くの一般企業では、ビッグデータを扱えるエンジニアや統計・数理的なデータ分析が行えるデータサイエンティストの確保ができず、性能実験や単純な業務への適用、小規模で試験的な導入に限られていたのも事実です。本格的なビッグデータ分析基盤の運用には、こうした人材は不可欠です。

しかし、自社でこうした人材を育成するには、相応の時間とコストがかかるため、自社の Hadoop 基盤の運用目的に応じて、データ分析を専門に行うサービスを利用することも選択肢として検討したほうがよいでしょう（Column「ワークショップの活用」「Hadoop のトレーニング」参照）。

Column　ワークショップの活用

Hadoop の情報源の精査や、教育トレーニングの受講などは、ある程度、技術者の学習コストがかかります。自社の技術者だけで Hadoop 基盤とアプリケーションを開発、構成する余力がない場合は、ベンダーが提供する Hadoop 基盤の構築サービスを利用することも検討しなければなりません。ハードウェアベンダーやシステムインテグレーターの多くは、Hadoop 基盤の構築サービスメニューを提供しています。また、ビッグデータを得意としたビジネスコンサルティング企業も Hadoop 基盤上の業務アプリケーション開発のノウハウを有しています。これらのハードウェアベンダー、システムインテグレーター、ビジネスコンサルティングファームの力を借りて、Hadoop 基盤を構築する場合は、ユーザーとベンダーの意思疎通や問題の明確化を目的とした有償ワークショップを開催することが少なくありません。自社内ですべてを解決できる基盤構築能力とアプリケーション開発能力があれば問題ありませんが、そうではない場合は、適宜、ベンダーのこれらのサービスをうまく活用することも検討してください。

■ 低コストで拡張性の高いデータ保管庫の確保

ある調査会社のデータによると、調査対象の企業の 40 %以上が、大量のデータを自社のシステムでは処理しきれないと回答しています。しかしビッグデータ分析基盤において、増え続けるデータの保管は必要不可欠です。初期投資を抑えつつも、拡張性の高い、データの保管庫の確保が必要です。また、さまざまな種類のデータが生成される環境では、それらを比較的低コストで一元的に簡単に管理できなければなりません。

ハードウェアの選定は、自社の分析対象となるデータ容量を見極めたうえで、判断する必要がありますが、最初からストレージのサイジングを考えるのが難しいのであれば、小規模システムから始め、スケールアウトさせていくこともできます。最近は、ビッグデータ分析向けのハードウェアも登場しているので、ノウハウが蓄積された段階で、そうしたラインアップを検討してみてください（2-2　ハードウェアコンポーネントの検討、ディスクのサイジング (p.59) 参照）。

■ 分析基盤の拡張性の確保

Hadoop を導入したいと考えている企業の多くは、自社データの増加に伴い、システムを無停止で拡張したいと考えています。システムの拡張は、無停止で、かつ、容易にできなければなりません。一般に、システムの拡張を続けると、性能向上が期待できますが、大規模システム向けのパラメーター設定（チューニング）の必要性も生じます。しかし、Hadoop のように、システム拡張

に伴い、スケールアウトメリットを活かせるIT基盤は、チューニングに時間をかけることなく、性能向上できなければなりません。Hadoopに限らず、自社に導入しようとしている分析基盤の拡張性、スケールアウト能力に関する実績などの情報を入手し、検討する必要があります。

Column　Hadoopのトレーニング

Hadoopの教育トレーニングを受講することで、ビッグデータ基盤に関する基礎知識を身につけることができます。Hadoopの教育トレーニングは、管理者向け、データ分析者向け、そして開発者向けの3種類に大きく分けられます。

教育トレーニングは、有償のものと無償のものが存在し、有償のトレーニングを受講すると、Hadoopの認定試験を無料で受験できるバウチャーチケットがもらえるものもあります。トレーニングの実施形態は、監督者がいる場合や、動画によって提供されているものもあります。表2-2に主要なHadoopの基礎知識を身につけるためのトレーニングを抜粋しておきます。

表2-2　トレーニング

提供元	有償/無償	URL
MapR	無償	https://mapr.com/training/essentials/
MapR	有償	https://mapr.com/training/academy-pro/
Cloudera	無償	https://www.cloudera.com/more/training/library/hadoop-essentials.html
Cloudera	有償	http://jp.cloudera.com/more/training/courses.html
Hortonworks	無償	http://learn.hortonworks.com//hdp-overview-apache-hadoop-essentials-self-paced-training
Hortonworks	有償	https://jp.hortonworks.com/services/training/

■ Hadoopの活用指針を明確にする

一般に、多くの企業において、Hadoopを最大限に活用できるプロジェクトや適用分野の特定に苦労することが少なくありません。多くの事業部門が、分析基盤で実行すべきワークロードやユースケースを特定し、ビジネスに対する影響や価値を明確に数値化することに悪戦苦闘しています。

このような課題に対しては、**自社のビジネスを理解でき、かつ、複雑なワークロードやユースケースの経験を持つビジネスパートナーの選定**が必要です。特に、複数部門に跨がって利用される従来型アプリケーションとの連携や、複雑なワークロードへの対応となると、当然、Hadoop単

体では完結しません。たとえば、既存の無停止型データベース、ビッグデータ分析環境、ビジネスインテリジェンス（BI）などの意思決定支援システム、統合監視システムなどが連携し、1 つのシステムとして稼働させる必要があります。

事業戦略に即したビジネス課題に対して、どのデータベースシステムのどのデータに対して、どのように分析し、誰に対して何を可視化し、経営や計画の意思決定支援をどこまで行うのかといった、ビッグデータを駆使した戦略策定支援の実績と経験を豊富に持つビジネスコンサルティングパートナーの検討が必要です。

■ Hadoop そのものの課題を知る

ビッグデータ分析市場において、Hadoop は、非常に有用なオープンソースソフトウェアとして急速に普及していますが、その一方で、課題も多数指摘されています。信頼性の高い大規模な Hadoop 基盤を実装するには、適切な導入計画の立案、基盤の設計、配備、長期管理、監査などを実施しなければなりません。にもかかわらず、多くの企業では、それらを実施できる専門スタッフを抱えることができず、Hadoop に精通した人材が不足しているのも事実です。人材不足だけでなく、Hadoop やそれを取り巻く周辺ソフトウェアそのものの技術革新が日進月歩であり、学習コストを押し上げていることも、課題解決のハードルを高くしている一因になっています。

このような課題に対しては、多くの Hadoop のカンファレンスにおける適用事例の情報入手が必要です。Hadoop 基盤の導入での苦労話やベストプラクティス、計画立案の手法などの情報を入手し、Hadoop 基盤の導入の実績が豊富なハードウェアベンダーや SI ベンダー、ユーザー企業の技術者とコネクションを築くことが必要です。

2-2　ハードウェアコンポーネントの検討

Hadoop クラスターで性能を引き出すには、ワークロードに応じたハードウェアを選定する必要があります。Hadoop クラスターを構成するマスターノード、ワーカーノードの両方で適切なハードウェアコンポーネントを構成する必要があります。

一般に、マスターノードは 1U サーバー、あるいは、2U サーバーで構成され、ワーカーノードは、ストレージドライブ数が大量に搭載できる Hadoop や分散ストレージ用途に特化した 2U サーバーや 4U サーバーなどで構成されます。最近では、筐体の高さが 2U で、10TB の容量を持つ LFF（ラージフォームファクタ）の HDD を 28 本搭載できるタイプ（例：HPE Apollo 4200 Gen9）や、4U で 3 ノードが格納でき、3 ノード当たり 10TB の HDD を 45 本搭載できるタイプのサーバー（例：

HPE Apollo 4530）も存在します。このように、Hadoop や分散ストレージに特化した x86 サーバーの登場により、大容量かつ高密度のビッグデータ分析基盤を構築できます（図 2-1）。

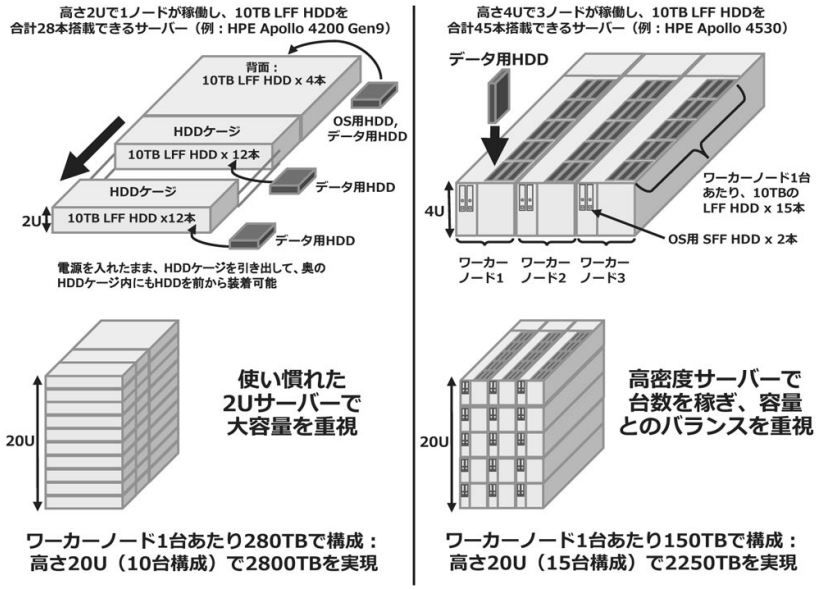


図 2-1　ビッグデータ用に開発された x86 サーバーの例

以下では、ビッグデータ分析基盤用の x86 サーバーにおけるハードウェアコンポーネントの検討項目について解説します。

2-2-1　CPU 構成

Hadoop クラスターでは、インテル社の Xeon プロセッサ、AMD 社の Opteron プロセッサ、あるいは、EPYC プロセッサを搭載したサーバーを用意します。通常、CPU コア数として、8 コア以上のプロセッサを 2 ソケット搭載したものが一般的です。インテル社の Xeon プロセッサが搭載するハイパースレッディングは、有効にすることが推奨されています。

CPU 負荷が高いワークロードの場合、コア数の多いプロセッサを選択することをお勧めします。通常、インメモリ分析で利用される Apache Spark や全文検索エンジンの Apache Solr などのワークロードは、CPU 負荷が高い傾向にあるため、コア数の多いプロセッサの選択を検討してください。表 2-3 に、Hadoop 基盤で利用される CPU 構成例を示します。

表 2-3　CPU 構成

規模	CPU の種類	ソケット数	1 プロセッサあたりのコア数
小規模	Xeon Gold 5118	2	12
中規模	Xeon Gold 6142	2	16
大規模	Xeon Gold 6154	2	18
大規模	AMD EPYC 7000	2	24

2-2-2　メモリ構成

Hadoop では、マスターノード、ワーカーノードともに、十分なメモリ容量が必要です。また、メモリの性能が発揮できるように、サーバー上の全メモリチャネルを使ってデータを格納するメモリ構成が必要です[*1]。特に、マスターノードにおける管理用サービスは、ワーカーノードのデータ量が増えると、メモリを大量に消費する傾向があります。したがって、大量のメモリを搭載しておく必要があります。計算性能が必要な場合は、通常、ワーカーノードにおいて、256GB 以上のメモリを搭載します。特に、インメモリによる集計処理、HDFS キャッシング、全文検索エンジン、機械学習などを使ったワークロードでは、大量のメモリを消費する傾向があるため、搭載する物理メモリの容量には注意が必要です（**表 2-4**）。

表 2-4　メモリ構成

規模	メモリ容量	用途
小規模	128GB 以上	部門での学習用、小規模な実験用、開発用
中規模	256GB～512GB 以上	一般的なディスク I/O 負荷を伴う分析処理
大規模	512GB～数 TB 以上	インメモリ分析、複雑な解析処理、機械学習 等

2-2-3　ディスク構成

ワーカーノードは、SAS または SATA のいずれかを選択できますが、コストとパフォーマンスのトレードオフがあります。SAS ドライブ、および、SATA のいずれも、数テラバイトのドライブが存在しますが、ワーカーノードでは、容量あたりのコストパフォーマンスの観点から、SATA ドライブが選択されることが少なくありません。ただし、I/O 性能が必要な場合は、ディスクの

＊1　最近の Xeon プロセッサでは、1 プロセッサ当たり 6 つのメモリチャネルがあり、2 ソケットサーバーでは、12 チャネル搭載されています。

耐久性と回転数の高い SAS ディスクも検討されます。容量単価を考慮しつつ、ディスクを選択する必要があります。ワーカーノードでは、Apache Hadoop のレプリケーションの仕組みにより、データは、ワーカーノードに跨がって記録されるため、RAID スキームは必要ありません。一般に Hadoop では、物理サーバー 3 台に分散する形で、ユーザーデータを 3 重化して保管します。

Column　ワーカーノードにおける JBOD と RAID 0

Hadoop では、ワーカーノードのディスク構成として、JBOD（Just a Bunch Of Disks）が構成されます。データ保護のための RAID 構成をとらず、単に複数ディスクを連結した構成は、JBOD と呼ばれます。RAID コントローラーではなく、単なるホストバスアダプタ配下のディスクは、JBOD で構成が可能です。また、RAID コントローラー配下のディスクでも、1 台の物理ディスクに対して、RAID 0 の論理ディスク 1 つを割り当てることで、物理ディスク数と同じ個数の論理ディスクからなる JBOD を構成できます。

Hadoop では、JBOD 構成以外に、大幅な性能向上を目的に、複数のディスクを束ねて RAID 0 を構成する場合もあります。しかし、複数の物理ディスクを束ねた RAID 0 構成は、構成された複数の物理ドライブのうち、1 つでも故障すると、RAID 0 配下の論理ドライブ上のデータは失われます。

RAID 0 の場合、ディスクの本数が多ければ多いほど、MTBF（平均故障間隔）は短くなります。Hadoop では、ワーカーノード間でデータのレプリカを 3 重化して保持するため、ワーカーノードのデータ領域を RAID 0 で構成しても Hadoop クラスター全体としてデータをロストしない仕組みになっていますが、RAID 0 による MTBF の低下は、メンテナンス工数の増大を招くおそれがあります。そのため、大幅な性能向上が期待できるとはいえ、RAID 0 の導入は、慎重に検討する必要があります。

表 2-5 は、Hadoop クラスターで利用される、ドライブの種類に関する比較表です。

表 2-5　ドライブの種類

	エントリークラス	ミッドライン	エンタープライズ
アクセス頻度	低	中	高
用途	OS/バックアップ	アーカイブ	トランザクション/DB
インターフェイス	SATA	SATA/SAS	SAS
回転数	5900/7200rpm	7200rpm	10000/15000rpm
想定される使用時間	8 時間/日	24 時間/日	24 時間/日
コスト	低	中	高
一般的な保証期間	1 年	1 年	3 年

マスターノードでは、HDFS にデータを格納するワーカーノードとは異なり、ローカルの管理用サービスと OS 用のストレージで構成しますが、それほど大容量のストレージは必要ありません。しかし、マスターノードは、クラスター全体の構成情報（メタデータ）を保有しており、このメタデータが消失すると、ワーカーノードにあるデータにアクセスできなくなってしまいます。これは、データロストを意味するため、メタデータが保管されるディスクは、耐障害性のある構成にしなければなりません。そのため、マスターノードでは、ストレージの信頼性が重要であり、SAS ドライブが推奨されます。

マスターノード上のメタデータが保管されるハードディスクは、SAS ドライブで RAID を構成し、耐障害性を高める必要があります。また、メタデータの消失を防ぐために、メタデータのバックアップを行う必要があります。通常、Hadoop のマスターノードや管理ノードでは、数百ギガバイトから数テラバイトの 2.5 インチ SAS ドライブを使った耐障害性のある RAID 構成（RAID 5、RAID 6、RAID 1+0 など）にします。また、ホットプラグ対応の SAS ドライブを採用することで、サーバーを再起動せずにハードディスクの交換が可能です。SAS や SATA の種類にかかわらず、サーバーの電源を投入したままハードディスクを引き抜く行為は、サーバーの機種、RAID 構成、ハードディスクドライブのホットプラグ対応の可否に依存しますので、ハードウェアベンダーが提供する、製品仕様に関する情報を入手し、慎重に構成を検討してください。

2-2-4　ネットワーク構成

Hadoop クラスターには、トップ・オブ・ラック・スイッチ（ToR スイッチ）と、それらを束ねる上位スイッチの 2 種類のスイッチがあります。ToR スイッチは、各ラック内のノード間のトラフィックをルーティングします。一方、上位スイッチは、ラック間のトラフィックをルーティングします。Hadoop クラスターは、通常、複数のサーバーとネットワークスイッチから構成され、ラックに搭載されますが、ラックごとに 1 台の ToR スイッチを構成すると、ラックごとに単一障害点（SPOF）が生まれます。複数ラックで構成される場合、1 台の ToR スイッチ構成で障害が発生すると、ストレージデータの複製を再構成するのに膨大な時間を要します。

1 台の ToR スイッチを持つ単一ラック構成の場合は、Hadoop クラスター全体がダウンする可能性があります。したがって、SPOF を排除した耐障害性やポートの枯渇防止の観点から、本番構成では、ラックごとに 2 つ以上の ToR スイッチを構成することが推奨されます。

耐障害性を高める手段[*2]としては、複数の物理ポートを束ねて冗長性を確保するリンクアグリ

＊2　スイッチ自体の耐障害性を高める手段として、冗長電源を搭載することを推奨します。

ゲーションがよく利用されます。サーバー上に搭載されている 1 つ目の NIC ポートは、1 つ目の ToR スイッチに接続し、2 つ目の NIC ポートは、2 つ目の ToR スイッチに接続します。また、スイッチ側では、2 つのスイッチ同士を束ねて 1 つのスイッチのように見せる IRF（Intelligent Resilient Framework）と呼ばれる構成をとることもあります。

ネットワークのレイテンシーも重要な要素です。一般に、Hadoop システムにおいて、ワーカーノードから ToR スイッチへの接続は、10Gbit イーサネット（GbE）で接続されます。

2-2-5　電源装置

大規模な Hadoop クラスターを構成する場合、サーバー、およびネットワーク機器の電源の冗長性を確保する必要があります。また、ラックへの給電システムについても、適切に冗長性が確保されている必要があります。各ラックには、少なくとも 2 つの電源分配装置（PDU）を搭載することを推奨します。 複数の PDU を搭載した場合、サーバーに搭載されている冗長電源の各電源は、異なる PDU に接続することが推奨されています。

2-2-6　Apache Hadoop 3 クラスターの小規模システム構成例

図 2-2 に、Apache Hadoop 3 クラスターのシステム構成例を示します。まずは、小規模システム構成例です。

Apache Hadoop 3 クラスターの小規模システム構成例では、マスターノードを 1 台で構成しています。そのため、マスターノードに障害が発生すると、Hadoop クラスター全体が停止します。また、マスターノードに保管されている Hadoop のメタデータがロストすると、クラスター全体のデータロストに直結するため、メタデータのバックアップが必要です。データは、ワーカーノードをまたいで 3 重化して記録するため、必要最低台数の 3 台のワーカーノードで構成しています。

■ ノード構成

管理ノードは、ハードウェアベンダーが提供する管理ソフトウェア（たとえば、HPE Insight CMU や clush コマンドなど）を導入し、ハードウェアの管理や監視、OS やアプリケーションなどの配備を行います。管理ノードが停止しても、Hadoop クラスターの稼働に影響はありませんが、ソフトウェアの構成変更や日々のメンテナンスを効率よく行うために導入を強く推奨します。クライアントマシンは、Hadoop ジョブ（アプリケーション）の投入、HDFS へのアクセスなどを行いま

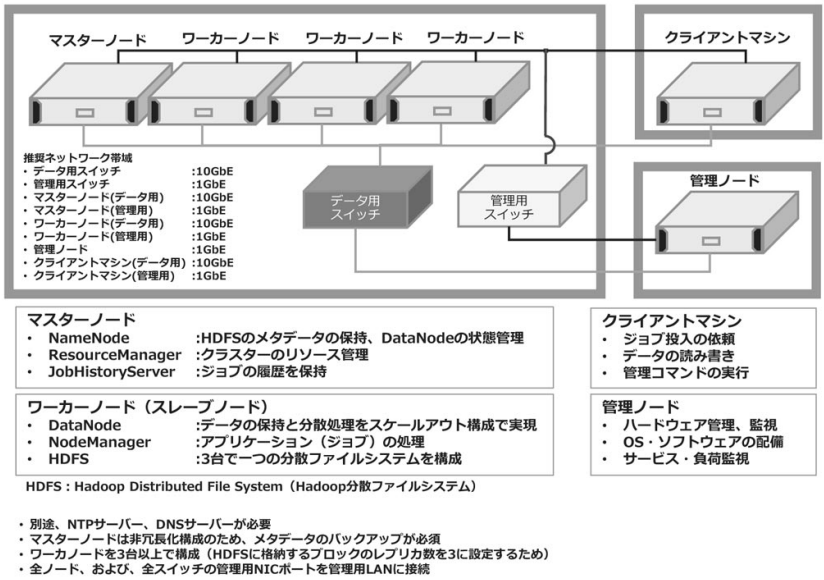


図 2-2　Apache Hadoop 3 クラスターの小規模システム構成例

す。当然、クライアントマシンにおいても、HDFS へのアクセスを行うための Hadoop のコマンド類などが実行できなければなりません。

■ ネットワーク構成

ネットワークスイッチは、データ用のものと管理用のものを 1 つずつ用意しています。データ用のスイッチは、各ワーカーノードの 10GbE NIC と接続します。管理用スイッチは、管理ノードからの OS 配備や管理操作に利用されるため、ワーカーノードの 1GbE NIC とワーカーノードのオンボードに搭載されている遠隔管理用のチップ（たとえば、HPE iLO など）が提供する管理用 NIC のポートに接続します。

■ 時刻同期と名前解決

Hadoop クラスターでは、マスターノード、ワーカーノード、管理ノード、クライアントのすべてにおいて、サーバーが刻む時計が同期して同じ時刻を刻む必要があります。時刻にズレがあると、Hadoop のジョブの実行に失敗するトラブルに見舞われます。そのため、別途、NTP サーバーが必要です。

また、全ノードにおいて、サーバーのホスト名の名前解決を行うDNSサーバーを別途用意します。DNSサーバーが用意できない場合は、全ノードの/etc/hostsファイルなどに、全ノードのホスト名（FQDNとドメイン名を除いたホスト名）とIPアドレスの対応を記述する必要があります。

Column　簡易DNSサーバー「dnsmasqのすすめ」

DNSサーバーは、全社的に利用しているものでもかまいませんが、dnsmasqを使った簡易DNSサーバーであれば、構築や設定も楽に行えます。dnsmasqを使った簡易DNSサーバーは、名前解決したいホストを/etc/hostsファイルのIPアドレスとホスト名の対応の記述で管理できますので、簡単にDNSサービスを提供、管理できます。筆者も、名前解決が必要なソフトウェアの動作検証を行う際は、dnsmasqサーバーを構築し、/etc/hostsにホスト名とIPアドレスを記述し、ホストの名前解決に利用しています。CentOS 7上で稼働するdnsmasqの構築手順は、拙書『Mesos実践ガイド』（インプレス発行）にも掲載しています。BINDに比べて、簡単に簡易DNSサーバーを構築できますので、ぜひ、試してみてください。

2-2-7　Apache Hadoop 3クラスターの中規模システム構成例

図2-3に、Apache Hadoop 3クラスターの中規模システム構成例を示します。

■ ノード構成

Apache Hadoop 3クラスターの中規模システム構成例では、マスターノードを3台で構成しています。そのため、マスターノードでは、NameNodeサービスおよびResourceManagerのHA構成が可能です。HA構成では、NameNodeとResourceManagerが稼働している1台に障害が発生しても、別のノードがNameNodeサービスおよびResourceManagerサービスを引き継ぐため、長時間にわたってHadoopクラスターがまったく使えないといった重大なシステム障害を回避できます。ワーカーノードは、数台規模から数十台規模で構成されますが、データは、ワーカーノードをまたいで3重化で記録するのが一般的です。

■ ネットワーク構成

ネットワークスイッチは、データ用のものを2つ用意し、障害に備え、冗長構成をとります。小規模構成と同様に、データ用のスイッチは、各ワーカーノードの10GbE NICと接続します。ワーカーノードの10GbE NICが複数ポート用意できる、あるいは、10GbE NICを増設できる場合は、

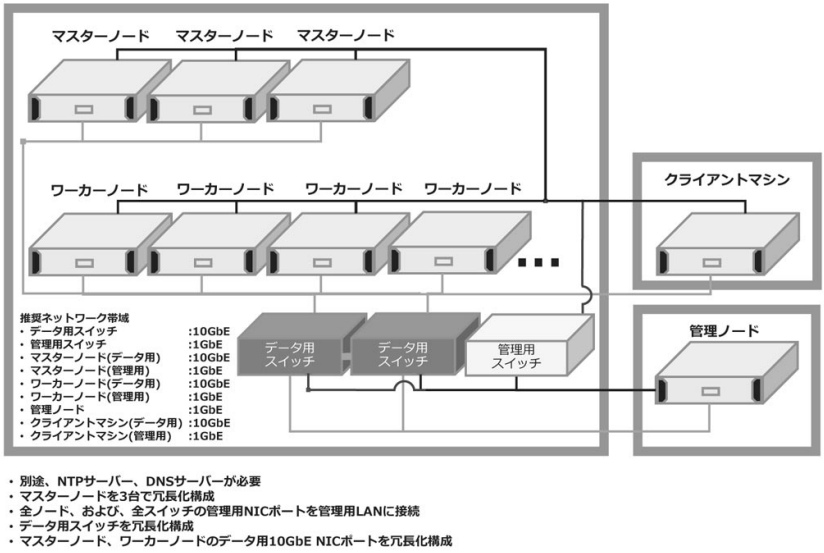


図 2-3　Apache Hadoop 3 クラスターの中規模システム構成例

Linux の Bonding 機能を使って帯域を増やすことも検討します。管理用スイッチは、小規模システム構成と同じです。

2-2-8　Apache Hadoop 3 クラスターのマルチラック大規模システム構成例

最後に、Apache Hadoop 3 クラスターの大規模システム構成例を示します（図 2-4）。

■ ネットワーク構成

大規模なシステムでは、Hadoop クラスターが複数のラックに跨がって構成されます。ラックには、データ用 ToR（トップ・オブ・ラック）スイッチが冗長構成で設置され、各スイッチは、その下に並ぶノード群のデータ用 NIC と接続します。データ用の ToR スイッチは、通常、10GbE ポートを多数搭載し、アップリンクに 40GbE ポートを搭載したものを選択します。

ワーカーノードの台数や通信帯域確保のため、ワーカーノードに NIC を増設した場合は、ToR スイッチのポート不足に注意してください。データ用のネットワークスイッチは、ToR スイッチ以外に、上位スイッチを用意します。上位スイッチも ToR スイッチと同様、冗長構成をとります。上位スイッチは、ワーカーノードから発生する大量のトラフィックをさばくため、40GbE 以上の

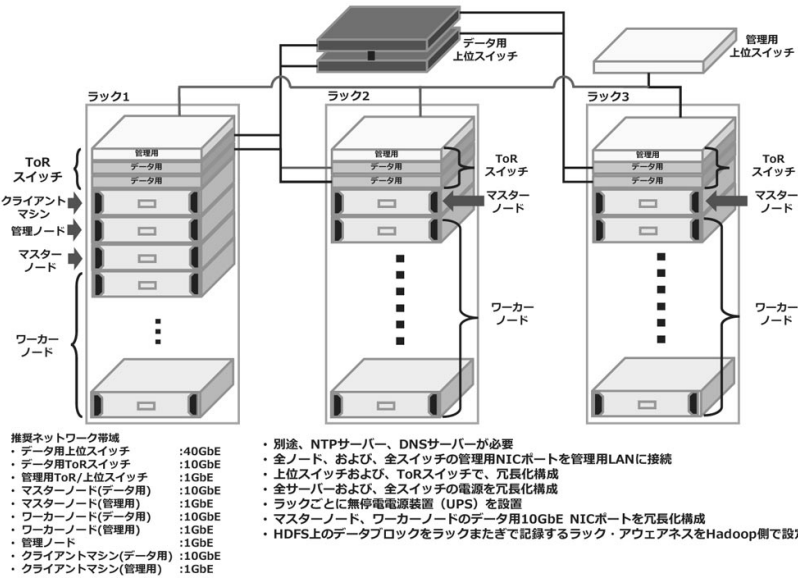


図 2-4　Apache Hadoop 3 クラスターのマルチラック大規模システム構成例

性能が確保できるスイッチを用意します。管理用の ToR スイッチは、1GbE 接続でかまいません。各サーバーの 1GbE の管理用 NIC と、オンボードに搭載されている遠隔管理用チップ（たとえば、HPE iLO など）が提供する管理用 NIC ポートを管理用スイッチに接続します。

■ ノード構成

今回の例では、マスターノードを 3 台で構成しているため、中規模システム構成例と同様に、NameNode サービスおよび ResourceManager の HA 構成が可能です。しかも、マスターノードを複数ラックに分散させて HA を構成した場合は、ラックごとの電源障害に対応できます。大規模システムでは、システム停止が大きな影響を与えるため、NameNode サービスと ResourceManager の HA 構成は必須といえます。ワーカーノードは、数十台以上の規模から数百台、数千台規模で構成されますが、データは、ラックごとの電源障害に対応させるため、ラックをまたいで 3 重化で記録するのが一般的です。ラックをまたいでレプリカを記録するには、Hadoop 側でラックアウェアネスという機能を使って実現できます。所属するラックを IP アドレスの範囲などで区分けした表を Hadoop 側で設定することで、データブロックがラックをまたいで書き込まれます。

2-3 MapR 6.0

本書では、エンタープライズ領域で利用されている商用 Hadoop ソフトウェアとして実績の豊富な MapR 6.0 を取り上げます。MapR 6.0 の Hadoop コアエンジンは、Hadoop 2 系ですが、さまざまなエンタープライズ向けの機能を提供しています。また、MapR 6.x 系では管理 GUI などが大幅に刷新されています。以下では、MapR 6.0 を取り巻くソフトウェア群、基本コンポーネント、システムを構成する際の考慮点、そして、具体的なシステム構成例を取り上げます。

2-3-1 MapR を取り巻くソフトウェア群

MapR 6.0 では、Apache Hadoop と同様に、分散ファイルシステムを提供します。MapR の分散ファイルシステムは、MapR File System（以下、MapR-FS）と呼ばれます。MapR では、MapR-FS 以外に、MapR-DB と呼ばれる NoSQL データベースも利用できます。MapR-DB は、MapR CDP に内蔵されており、高スループット、低遅延の NoSQL データベースを実現できます。また、MapR-ES と呼ばれるストリーミングソフトウェアも存在します。一般に、ストリーミングソフトウェアは、さまざまな IoT 機器からリアルタイムに生成されるデータを取得し、ビッグデータ分析基盤に蓄積する際に利用されます（図 2-5）。

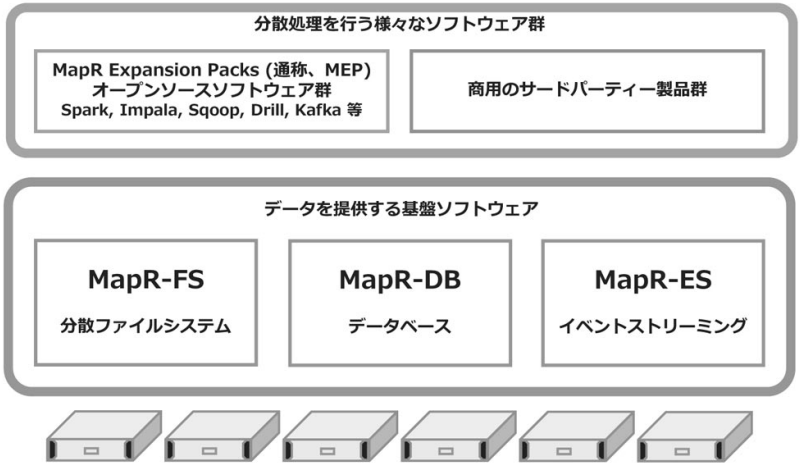


図 2-5 MapR を取り巻くソフトウェア群

これらのMapR-FS、MapR-DB、あるいはMapR-ES上で、分散処理を行うためのさまざまなソフトウェア群が稼働します。MapRでは、ビッグデータ分析基盤で頻繁に利用される有名なオープンソースソフトウェア群をパッケージにまとめたMapR Expansion Packs（以下、MEP）が提供されています。インメモリ分析で利用されるApache Sparkや、データベースとHadoopクラスターのデータのやり取り（コピーなど）を行うSqoop、インメモリクエリーを行うImpalaなど、さまざまなオープンソースソフトウェアを利用できます。また、オープンソースソフトウェアだけでなく、列指向データベースのVerticaや、BIツールとして有名なPentahoなど、商用のサードパーティ製品との連携も可能です。

2-3-2 MapR 6.0の基本コンポーネント

MapRクラスターでは、メタデータを保管し、クラスター全体の調整役を行うノードを**コントロールノード**といい、ユーザーデータの保管と分散処理を担当するノードは、**データノード**（または、ワーカーノード）と呼ばれます。また、データノードとコントロールノードの役割を両方担うノードは、**コントロール・アズ・データノード**（Control-As-Dataノード、以下、C-a-Dノード）と呼ばれます。一般に、大規模なシステムの場合、データノードとコントロールノードを分離することが望ましいとされていますが、予算の都合上、データノードの台数が限られる構成では、C-a-Dノードを導入することが少なくありません。C-a-Dノードを構成[*3]すれば、初期投資を抑えた小規模なHadoopクラスターを構成できます。また、Apache Hadoopクラスターの場合と同様に、クライアントノードが存在し、MapRクラスターへのジョブの投入やMapR-FSに保管されたデータの操作などを行います。MapRクラスターでは、MapR-FS上にデータを保持し、そのデータに対する分散処理は、Apache Hadoop 3と同様に、YARNのフレームワークを使います。

2-3-3 MapRクラスターのサービス

MapRクラスターでは、以下で説明するさまざまなサービスが稼働します。Apache Hadoop 3クラスターとは異なり、MapR特有のサービスも存在します。

● CLDB（Container Location Database）サービス

MapRクラスターのコントロールノードで稼働するサービスです。CLDBサービスは、MapR固有のサービスであり、MapRクラスター内のデータの場所を特定、追跡する重要なサービ

＊3　ただし、C-a-Dノードの構成は、分離した構成に比べると、クラスターの制御する速度が低下します。

スです。MapR では、データ保管の単位を**コンテナ**[*4]と呼びます。

CLDB サービスは、このコンテナの配置に関するデータベースを管理します。また、NFS サービスや、ストレージ機能を提供する FileServer ノードと連携し、ストレージ機能を提供する場合にも必要となるサービスです。CLDB サービスは、Apache Hadoop クラスターにおける NameNode を置き換えたものです。MapR クラスターでは、複数ノードに CLDB サービスをインストールすることが推奨されており、通常は、2 ノード、あるいは、3 ノードで起動[*5]します。メタデータを取り扱うサービスなので、コントロールノード、または、C-a-D ノードで稼働させます。

MapR では、CLDB サービスのフェールオーバー機能が提供されています。MapR の有償のエンタープライズ版では、フェールオーバーが自動的に行われますが、無償のコミュニティ版では、自動的に行われないため、手動でフェールオーバーさせる必要があります。

● ZooKeeper（以下、ZK）サービス

クラスター全体の可用性を維持するためにノード間の調整役を担うサービスです。設定の同期（配布）や、調停などを行います。サービスの競合を防ぎ、クラスターにおける各サービスの一貫性を確認する重要なサービスです。通常、奇数台のノードで実行されます。ZK サービスは、その他のさまざまなサービスが起動する前に起動する必要があります。コントロールノード、あるいは、C-a-D ノードで稼働させます。

● ResourceManager（以下、RM）サービス

RM サービスは、その名のとおり、クラスターのリソースを管理するサービスです。アプリケーションのスケジューリングを行います。RM サービスは、通常、可用性の観点から、アクティブ・スタンバイ型で複数のノードで稼働させることが少なくありません。コントロールノード、あるいは、C-a-D ノードで稼働させます。

● WebServer（以下、WS）サービス

WS サービスは、MapR の管理 GUI である「MapR Control System」（以下、MCS）を提供します。複数ノードで稼働させることが可能です。コントロールノード、あるいは、C-a-D ノードで稼働させます。

＊4　Docker で取り扱う「コンテナ」とは無関係です。

＊5　複数の CLDB サービスのうち、1 つは、マスターとして起動し、その他の CLDB サービスは、スタンバイ用に起動します。

● HistoryServer（以下、HS）サービス

HS サービスは、分散処理のジョブのメトリクス情報やメタデータをアーカイブします。これにより、ユーザーは、実行済みのジョブの履歴を参照できます。コントロールノード、あるいは、C-a-D ノードで稼働させます。

● NodeManager（以下、NM）サービス

NM サービスは、RM サービスと連携し、YARN コンテナ（計算リソース）を管理します。NM サービスは、データノード、あるいは、C-a-D ノードで稼働させます。

● FileServer（以下、FS、あるいは、MFS）サービス

FS サービスは、ディスクストレージを管理します。全データノードで稼働させる必要があります。また、CLDB サービスを実行している全コントロールノードにおいても稼働させる必要があります。FS サービスは、MapR FileServer サービスとも呼ばれ、「MFS」と記される場合もあります。

● NFS サービス

古くからエンタープライズ領域で MapR が採用されていた理由の一つに、Hadoop クラスターにおける NFS の利用が挙げられます。MapR では、クライアントマシンに対して、NFS サービスを提供できるため、Hadoop の専用コマンドを知らなくても、ユーザーは、通常の Linux コマンドで、MapR クラスターに NFS マウントを行い、ファイルのコピーなどの操作が可能です。NFS サービスは、MapR クラスターの全ノードで稼働させることが推奨されています。

● Warden サービス

クラスターで稼働する各種サービス（ZK サービスは除く）のコーディネート、起動、停止、再起動、監視などを行う内部管理用のサービスです。

図 2-6 に、以上のコンポーネントサービスの位置付けをまとめたものを示します。

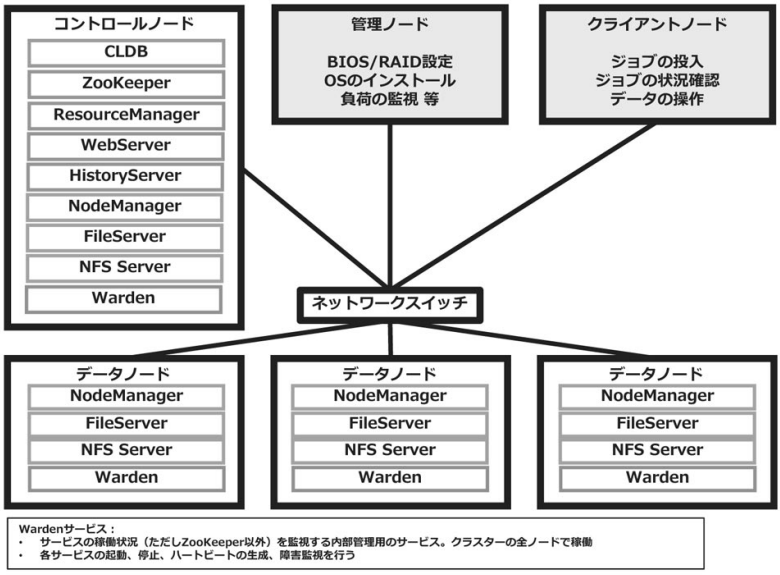


図 2-6　MapR クラスターを構成する主なコンポーネント

2-4　MapR 6.0 におけるハードウェアの検討

MapR は、エンタープライズ用途の Hadoop クラスターという性格上、クラスターの構築計画、推奨ハードウェアスペックなどが明確に記されています。以下では、クラスターの構築前に考慮すべきハードウェアの検討について解説します。

2-4-1　ハードウェアスペック

MapR クラスターを正常に稼働させるためのハードウェアスペックは、その用途によって大きく異なりますが、一般的なハードウェアスペックは、以下のとおりです。

- デュアルソケットの CPU
- 8 コア以上のプロセッサを 2 基搭載し、ハイパースレッディングを有効に設定
- 最低でも 64GB メモリ
- 数テラバイト SATA ドライブ × 12 本以上
- 10GbE NIC

●OS 用のディスクとデータ用ディスクは、物理的に別のディスクドライブで保持

■ CPU

できるだけ多くのコアを搭載した CPU を選択します。動作周波数が高いクロックでもコア数が少ないと、あまり性能を発揮できない場合もあります。MapR では、動作周波数よりも、できるだけコア数の多い CPU が推奨されています。

■ メモリ

Apache Hadoop クラスターの場合にもいえることですが、分析業務では、メモリを大量に消費する場合が少なくありません。ノード数が少ない小規模なものでも 64GB 以上が推奨されています。また、MapR-DB を多用するシステムでは、256GB 以上が推奨されています。

■ OS 用のディスク

MapR クラスターでは、OS 用のディスクとデータ用のディスクは、別の物理ディスクで構成します。OS 用のディスクは、RAID 1 など、耐障害性のある構成にします。また、パーティションは、Linux が提供する LVM を使用してもかまいません。

■ MapR-FS 用のディスク

MapR-FS を構成するためのデータ用のディスクは、**クラスターストレージ**と呼ばれます。クラスターストレージは、RAID 構成にせず、JBOD 構成にします。MapR クラスターでは、複数のディスクからなる JBOD 構成をデータ領域として使用しますが、1 台のデータノードに対して、クラスターストレージは、3 本以上のディスクで構成することが推奨されています。また、MapR-DB では、インデックス処理で I/O 負荷がかかるため、磁気ディスクよりも、SSD が推奨されています。一方、MapR-ES では、SSD ではなく、通常の磁気ディスクが利用されます。クラスターストレージでは、OS 用のディスクと異なり、Linux OS で提供される LVM は使用しません。

■ ディスクのサイジング

MapR クラスターでは、ユーザーデータを 3 重化して MapR-FS に書き込みますが、元のデータに対して、どれくらいの物理ディスクが必要になるのかといったサイジングが必要になります。

MapR クラスターでは、ユーザーデータに対して、以下の手順でサイジングします。

1. ユーザーが将来にわたって使用が予定されているデータサイズを決定
2. レプリケーションファクタをデータサイズに乗じる
3. 25 %のオーバーヘッドを乗じる

最後の 25 %のオーバーヘッドは、MapR クラスターで生成されるログや一時ファイルなどに必要です。具体例を以下に示します。

1. 分析対象のユーザーデータは、100TB
2. 3 重化（レプリケーションファクタは 3）に決定。100TB × 3 ＝ 300TB
3. 25 %のオーバーヘッドを乗じると、300TB × 1.25 ＝ 375TB

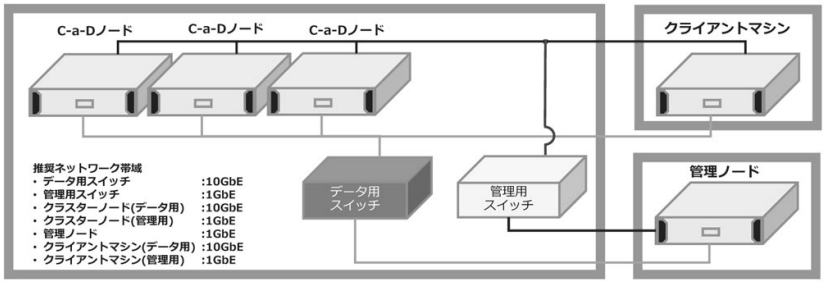
よって、100TB のユーザーデータの場合、最低でも 375TB の容量の実効容量を確保できる物理ディスクが必要です。

2-4-2　MapR クラスターの小規模システム構成例

図 2-7 に、MapR クラスターのシステム構成例を示します。まずは、小規模システム構成例です。

MapR クラスターの小規模システム構成例では、コントロールノードやデータノードを区別せず、C-a-D ノード 3 台で構成しています。そのため、C-a-D ノードの 1 台に障害が発生しても、Hadoop クラスター全体が停止することはありません。ユーザーデータは、データノードをまたいで 3 重化で記録するため、必要最低台数の 3 台のデータノードで構成しています。管理ノードは、ハードウェアベンダーが提供する管理ソフトウェア（たとえば、HPE Insight CMU や clush コマンドなど）を導入し、ハードウェアの管理や監視、OS やアプリケーションなどの配備を行います。管理ノードが停止しても、MapR クラスターの稼働に影響はありませんが、ソフトウェアの構成変更や日々のメンテナンスを効率よく行うために導入を強く推奨します。

クライアントマシンは、Hadoop ジョブ（アプリケーション）の投入や、Hadoop のコマンド類を使って MapR-FS へのアクセスを行います。また、MapR クラスターは、NFS サービスも提供するため、クライアントから NFS マウントを行い、Linux の使い慣れたコマンドで MapR-FS のファイル群にアクセスします。ネットワークスイッチは、データ用のものと管理用のものを 1 つずつ用意しています。データ用のスイッチは、各 C-a-D ノードの 10GbE NIC に接続します。管理用スイッチは、管理ノードからの OS 配備や管理操作に利用されるため、C-a-D ノードの 1GbE NIC



各種サービス	役割	コントロールノードで実行するサービス	データノードで実行するサービス	略称
CLDB	MapR-FS上のメタデータの保持	○		CLDB
ZooKeeper	サービスのコーディネート(設定の同期、配布、調停 等)	○		ZK
ResourceManager	クラスターのリソース管理	○		RM
WebServer	Web管理画面(MCS :MapR Control System) の表示	○		WS
HistoryServer	ジョブの履歴を保持	○		HS
NodeManager	YARNリソースコンテナーの管理	○	○	NM
FileServer	ディスクストレージの管理	○	○	FS
NFS Server	NFSサービスの提供	○	○	NFS
Warden	内部管理用 (サービス起動、停止、障害監視 等)	○	○	-

MapR-FS : MapR File System (MapR社が提供する超高速分散ファイルシステム)

・別途、NTPサーバー、DNSサーバーが必要
・コントロールノードとデータノードが1台に同居するC-a-Dノードで構成
・C-a-Dノードを3台で構成 (MapR-FSに格納されるブロックのレプリカ数を3に設定)
・全ノード、および、全スイッチの管理用NICポートを管理用LANに接続

管理ノード
・ハードウェア管理、監視
・OS・ソフトウェアの配備
・サービス・負荷監視

クライアントマシン
・ジョブ投入の依頼
・データの読み書き
・管理コマンドの実行

図 2-7　MapR クラスターの小規模システム構成例

と C-a-D ノードのオンボードに搭載されている遠隔管理用のチップ（たとえば、HPE iLO など）が提供する管理用 NIC のポートに接続します。

MapR クラスターでは、Apache Hadoop 3 クラスターと同様に、クラスターノード、管理ノード、クライアントのすべてにおいて、サーバーが刻む時計が同期して同じ時刻を刻む必要がありますので、別途、NTP サーバーが必要です。また、全ノードにおいて、サーバーのホスト名の名前解決を行う DNS サーバーを別途用意します。DNS サーバーが用意できない場合は、全ノードの/etc/hosts ファイルなどに、全ノードのホスト名（FQDN とドメイン名を除いたホスト名）と IP アドレスの対応を記述する必要があります。

2-4-3　MapR クラスターの単一ラック構成例

図 2-8 に MapR クラスターの単一ラック構成例を示します。単一ラック構成では、小規模システム構成例と単一ラック中規模システム構成例の 2 種類を示します。

MapR クラスターの単一ラック小規模システム構成例では、C-a-D ノードを 3 台で構成しています。データノードの増設用に、ラックの下側が空きスペース† になっています。

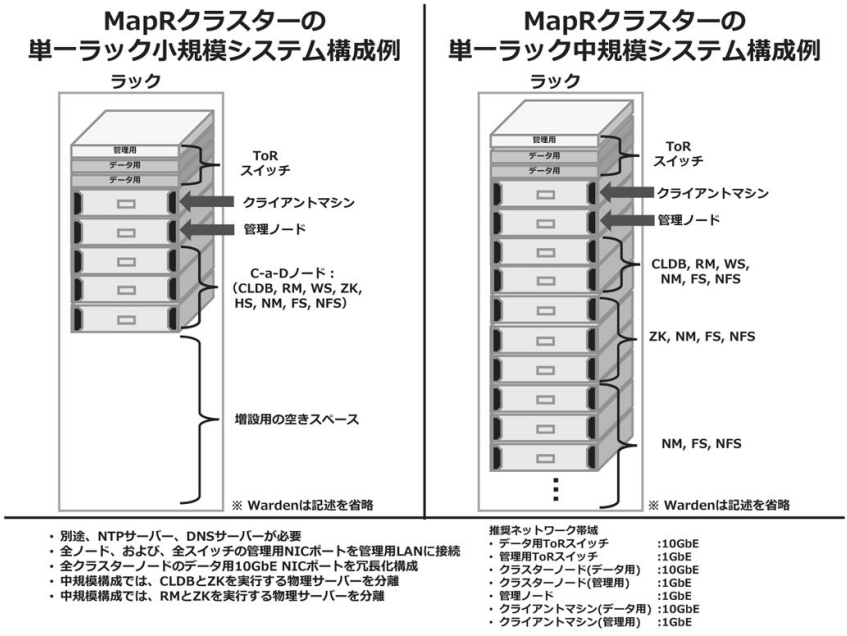


図 2-8　MapR クラスターの単一ラックシステム構成例

† 通常、ラックの最下部には、重量のある UPS（無停電電源装置）を設置します。背丈が高い 42U ラックの場合は、下から UPS やサーバーをマウントし、上側を増設用に空けて重心を下げることでラックの安定性を確保します。

ToR スイッチは、データ用と管理用に分けられます。一方、単一ラック中規模システム構成例では、CLDB サービスと ZK サービスを実行する物理サーバーを分離しています。さらに、RM サービスと ZK サービスを実行する物理サーバーも分離しています。MapR クラスターでは、小規模の場合、CLDB サービスと ZK サービス、および、RM サービスと ZK サービスを同居させる場合が少なくありませんが、中規模以上になると、別々の物理サーバーに配置することが推奨されています。データノードは、数台規模から数十台規模で構成されますが、ユーザーデータは、データノードをまたいで 3 重化で記録するのが一般的です。ネットワークスイッチは、データ用のものを 2 つ用意し、障害に備え、冗長構成をとります。小規模構成と同様に、データ用のスイッチは、各データノードの 10GbE NIC と接続します。データノードの 10GbE NIC が複数ポート用意できる、あるいは、10GbE NIC を増設できる場合は、Linux の Bonding 機能を使って帯域を増やすことも検討します。

2-4-4　MapR クラスターの 2 ラックシステム構成例

MapR クラスターの 2 ラックシステム構成例を示します（図 2-9）。

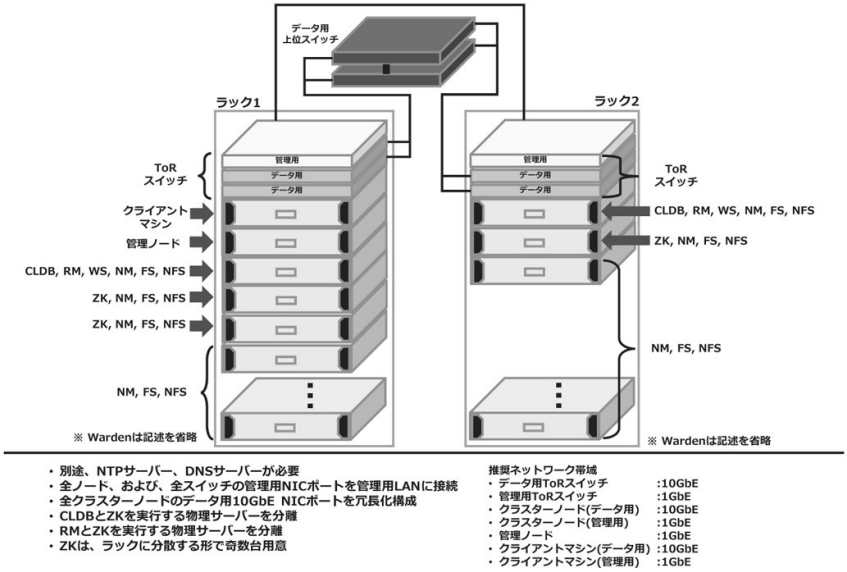


図 2-9　MapR クラスターの 2 ラックシステム構成例

2 ラックシステム構成例では、MapR クラスターが複数のラックにまたがって構成されます。ラックには、データ用の ToR（トップ・オブ・ラック）スイッチが冗長構成で設置され、各スイッチは、その下に並ぶノード群のデータ用 NIC と接続します。データ用 ToR スイッチは、通常、10GbE ポートを多数搭載し、アップリンクに 40GbE ポートを搭載したものを選択します。データノードの台数や通信帯域確保のため、データノードに NIC を増設した場合は、ToR スイッチのポート不足に注意してください。データ用のネットワークスイッチは、ToR スイッチ以外に、上位スイッチを用意します。上位スイッチも ToR スイッチと同様、冗長構成をとります。上位スイッチは、ラック内のデータノードから発生する大量のトラフィックをさばくため、40GbE 以上の性能が確保できるスイッチを用意します。

今回の例では、ラック 1 とラック 2 に CLDB サービス、および、RM サービスを 2 つずつ、分散させて構成しています。また、ZK サービスもラックラック 1 とラック 2 に分散させて構成します。ZK サービスは、奇数個必要ですので、ラック 1 で物理サーバー 2 台に分散させ、残り 1 つ

は、ラック 2 に構成します。また、CLDB サービスと ZK サービス、RM サービスと ZK サービスは、実行する物理サーバーを分離しています。

ラックごとの電源障害に対応させるため、ラックをまたいで 3 重化でデータを記録するラックアウェアネスを構成します。

2-4-5　MapR クラスターのマルチラック大規模システム構成例

MapR クラスターのマルチラック大規模システム構成例を示します（図 2-10）。

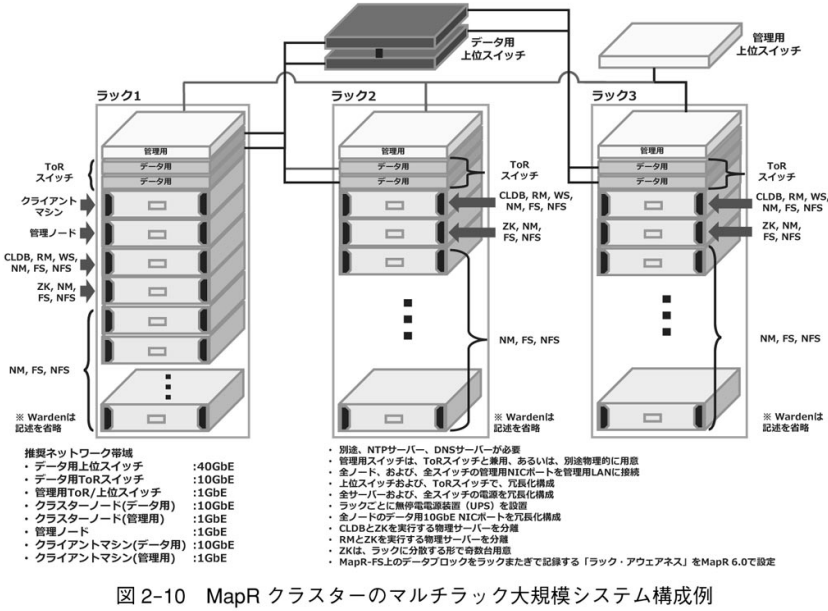


図 2-10　MapR クラスターのマルチラック大規模システム構成例

数百台規模になるような、マルチラック大規模システム構成例では、CLDB サービス、RM サービス、ZK サービスを別々のラックの物理サーバーに分散させます。また、CLDB サービスと ZK サービス、および、RM サービスと ZK サービスは、別々の物理サーバーに分離します。大規模クラスターの場合は、ノード数が膨大になるため、ToR スイッチや上位スイッチの空きポートに注意してください。図では、ラックあたりの ToR スイッチが 2 台で構成されていますが、増設 NIC やノード台数によっては、ポート数の多い ToR スイッチや、データ用の ToR スイッチ自体を 3 台以上にすることも検討しなければなりません。

2-5 Hadoop クラウド基盤の検討

先にオンプレミスにおける Hadoop システムの構成について解説しましたが、Hadoop クラスターを分析の本番機として利用する方法としては、社内のオンプレミス環境で構築する以外に、パブリッククラウドを利用することも可能です。パブリッククラウド、プライベートクラウドに限らず、Hadoop をクラウドサービスとして提供、あるいは、利用する環境を Hadoop-as-a-Service と呼び、Hadoop に限らず、ビッグデータの保管、処理、分析アプリケーションをクラウドサービスとして提供、あるいは、利用する環境を Bigdata-as-a-Service（以下、BaaS）、または、Analytics-as-a-Service（以下、AaaS）と呼びます。

パブリッククラウドでの運用を検討したほうがよいケースとしては、たとえば、Hadoop の導入以前に、アプリケーション開発を先行して行い、開発者自身でインフラの調達まで行うような場合や、投資効果が予測できないテスト段階で Hadoop を利用するようなケースです。このようなケースでは、パブリッククラウドの Hadoop サービスを利用するのが無難です。また、たとえ本番環境であっても、短期的な利用であれば、クラウドサービスを利用するのが合理的です。以下では、BaaS や AaaS を利用するうえで、利用可能なパブリッククラウドサービスをいくつか紹介します。

2-5-1 仮想マシンベースの Hadoop クラスターが利用可能なパブリッククラウド

仮想マシン（VM）ベースの Hadoop が利用可能なパブリッククラウドとしては、Amazon EMR、Azure HDInsight、Google Cloud Dataproc が代表的なものとして挙げられます。以下、この 3 サービスの特徴をピックアップして掲載します。

Amazon EMR（Elastic MapReduce）の特徴

- ●Apache Spark、HBase、Presto、Flink などのフレームワークが利用可能
- ●EMR File System (以下、EMRFS) により、Hadoop のオブジェクトストアとして Amazon S3 が利用可能
- ●HDFS、Amazon S3、Amazon DynamoDB などの複数のデータストアが利用可能
- ●Pig、Hive、Impala や、機械学習ライブラリの Mahout、MXNet、統計解析用の R 言語などが利用可能
- ●Tableau、MicroStrategy、Datameer などの一般的なビジネスインテリジェンス（BI）ツールが

利用可能

●実行中のクラスターの自動スケーリングや手動によるサイズ変更が可能

サービスに関する情報：

https://aws.amazon.com/jp/emr/details/

料金表：

https://aws.amazon.com/jp/emr/pricing/

Azure HDInsight の特徴

●Hortonworks Data Platform (HDP) のクラウドディストリビューション

●オンプレミスと Azure のハイブリッド型ビッグデータ分析基盤の構築が可能

●Hadoop、Spark、Hive、LLAP（Live Long and Process)、Kafka、Storm、R などのフレームワークが利用可能

●Spark MLlib、Mahout などの機械学習ライブラリが利用可能

●Hortonworks 版 Hadoop で定評のある GUI 管理ツール「Ambari」が利用可能

●Bash 以外に、PowerShell、Windows のコマンド入力もサポート

サービスに関する情報：

https://docs.microsoft.com/ja-jp/azure/hdinsight/hadoop/apache-hadoop-introduction

料金表：

https://azure.microsoft.com/en-us/pricing/details/hdinsight/

Google Cloud Dataproc の特徴

●バッチ処理、クエリー実行、ストリーミング、機械学習が可能な Spark / Hadoop サービス

●クラスターを速やかに作成し、クラスターの起動、スケーリング、シャットダウンが高速

●BigQuery、Cloud Storage、Cloud Bigtable など他の Google Cloud Platform サービスとの統合

●Google Cloud Platform Console や Google Cloud SDK、Cloud Dataproc REST API が使用可能

●Pig、Hive が利用可能

●Hadoop、Spark、ツール類のバージョンの切り替えが可能

サービスに関する情報：

https://cloud.google.com/dataproc/?hl=ja

料金表：

https://cloud.google.com/dataproc/pricing?hl=ja

パブリッククラウドにおける Hadoop サービスは、3 社ともにエコシステムの利用や便利な機能を前面に打ち出しています。上記の 3 社に限らず、パブリッククラウドサービスを利用する場合は、**表 2-6** に挙げられる項目について、十分な検討と確認が必要です。また、サービス利用後には、速やかに、性能ベンチマークを行い、期待する性能が出るかどうかのテストを行うことをおすすめします。

表 2-6　検討事項

項目	検討事項
データ転送速度	社内環境にある大容量のデータをパブリッククラウドにコピーする際の転送速度、転送にかかる料金
データ暗号化	社内環境から、パブリッククラウドにデータをコピーする際の通信の暗号化の仕組みの有無、暗号強度
分散処理性能	ワーカーノードの CPU 速度、コア数、メモリ容量、ディスク性能、ディスク容量、ネットワーク性能、YARN による分散処理の性能
エコシステムソフトウェア	利用可能な Hadoop エコシステムの種類とバージョン（手動による追加インストールの可否）
OS の種類	Hadoop クラスターで利用可能な Linux OS の種類とバージョン
分散ファイルシステム	HDFS 以外のファイルシステムの利用可否
ハイブリッド利用	オンプレミスの Hadoop 基盤とのハイブリッド運用の可否
地理的要素	データセンターの所在国、所在地（データの機密漏えいの観点とデータ転送時間を考慮）

2-5-2　物理マシンベースの Hadoop クラスターが利用可能なパブリッククラウド

Amazon EMR、Azure HDInsight、Google Cloud Dataproc は、いずれも仮想マシンを使った Hadoop クラスターを提供しますが、物理サーバーを提供するビッグデータ専用のクラウドサービスも存在します。物理サーバーを提供するクラウドサービスは、一般に、ベアメタルクラウドと呼ばれます。英国 Bigstep 社は、ビッグデータに特化したベアメタルクラウドサービスを展開しており、

物理サーバーの性能を活かした、高速分析基盤の提供に定評があります。Bigstep 社では、ビッグデータに関連するさまざまなクラウドサービスを展開していますが、特に、Hadoop に関しては、MapR 版 Hadoop、Cloudera 版 Hadoop、Hortonworks 版 Hadoop の 3 つをクラウドサービスとして提供しています（図 2-11）。

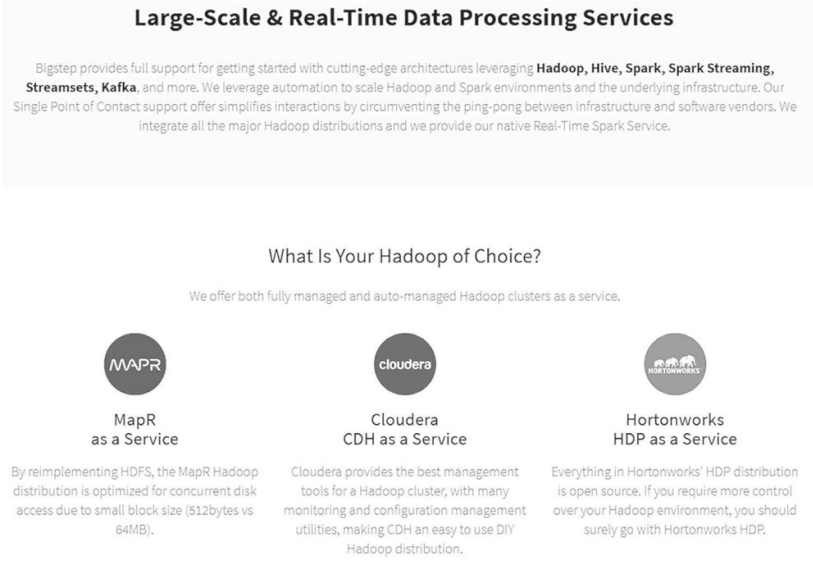


図 2-11　Bigstep 社が提供する Hadoop のベアメタルクラウドサービスでは、MapR 版 Hadoop、Cloudera 版 Hadoop、Hortonworks 版 Hadoop の 3 つを提供

以下、Bigstep 社が掲げる 3 大 Hadoop ディストリビューションをベースとしたベアメタルクラウドサービスの特徴を抜粋して掲載します。

MapR 版 Hadoop のベアメタルクラウドサービスの特徴

- ●大規模にスケールさせた場合、他のディストリビューションより 10 倍高速
- ●NFS サービスの提供、高可用性を実現
- ●1 クリックでベアメタル環境をセットアップ
- ●マルチテナンシーを構成可能

参照：`https://bigstep.com/mapr-on-metal-cloud`

Cloudra 版 Hadoop のベアメタルクラウドサービスの特徴

●高い IOPS が得られる高性能 SSD の採用
●インスタンスあたり 40Gbs の高性能ネットワーク
●クリック操作や API を使ってクラスターを拡張
●Cloudera Manager による GUI 管理

参照：`https://bigstep.com/cloudera-on-bare-metal`

Hortonworks 版 Hadoop のベアメタルクラウドサービス

●ベンダー非依存の 100% オープンソースソフトウェアで構成
●Bigstep 社が提供するドラッグ＆ドロップの GUI で基盤を配備
●ほとんどの Hadoop アプリケーションをサポート

参照：`https://bigstep.com/hortonworks-on-bare-metal-cloud`

2-5-3 Hadoop サービスを提供する社内プライベートクラウド基盤

オンプレミスのプライベートクラウド基盤で Hadoop の環境を構築する場合、仮想化を使わない物理サーバーベースの**ベアメタル配備**によるものと、仮想マシンを使ったものに分けられます。ベアメタル配備の場合は、物理サーバー構成なので、性能検証まで含めた動作確認を行いますが、仮想マシンの場合は、KVM（Linux カーネルが提供する仮想化機能）による性能劣化を許容しなければならないため、事前の機能検証やアプリケーション開発者向けの開発基盤提供などに留めておくなどの Hadoop の利用範囲の検討が必要です。

社内環境に Hadoop を提供するには、Hadoop を自動配備するソフトウェアや OpenStack などのクラウド基盤ソフトウェアを活用します。Hadoop に限らずオープンソースソフトウェアを自動配備するソフトウェアとしては、Mesosphere 社が提供する Mesosphere DC/OS や、Canonical 社が提供する「MaaS と Juju」などがあります。

Mesosphere DC/OS は、データセンターにおける大量のマシンで構成されたシステムにおいて、アプリケーションの自動配備と資源管理を行います。ユーザーは、Mesosphere DC/OS の画面において、Hadoop（CDH）のベアメタル配備が可能です。ただし、Mesosphere DC/OS の場合は、事前に、Hadoop をインストールする全ノードに、OS と Mesosphere DC/OS の管理エージェントをイン

ストールし、Mesosphere DC/OS 管理配下に全ノードを参加させておくことが必要です。

MaaS（Metal-as-a-Service）は、管理対象となる物理サーバーノードに Ubuntu Server を自動インストールし、さらに、Juju が提供する GUI により、Hadoop をはじめとするオープンソースソフトウェアを簡単にインストール可能です（**表 2-7**）。ユーザーは、マスターノードやワーカーノードの連携動作のためのクラスター化（Juju では、オーケストレーションと呼びます）を Juju の GUI で簡単に行えます。

IaaS 基盤ソフトウェアの OpenStack を使う場合は、OpenStack Sahara と呼ばれる Hadoop の配備機能を利用します。OpenStack Sahara は、ベアメタル配備と仮想マシンでの配備の両方をサポートしています。

表 2-7　MaaS

	Mesosphere DC/OS	MaaS と Juju の組み合わせ	OpenStack Sahara	
OS やアプリケーションの配備形態	ベアメタル	ベアメタル	ベアメタル	仮想マシン
OS の事前インストール	別途必要	MaaS で可能	Ironic で可能	Hadoop 入り VM イメージを利用
Hadoop のインストール	Mesosphere DC/OS で可能	Juju で可能	Sahara で可能	Hadoop 入り VM イメージを利用
インストール対象のサーバー OS の種類	RHEL、CentOS	Ubuntu Server	RHEL、CentOS、Ubuntu Server	RHEL、CentOS、Ubuntu Server

以上で、仮想マシンベースで Hadoop を利用する Amazon、Azure、Google のパブリッククラウドでの検討、ベアメタルクラウドでの検討、そして社内プライベートクラウド基盤の検討内容について簡単に紹介しました。

2-6　まとめ

本章では、導入前の検討事項、簡単なシステム構成例を示しました。Hadoop クラスターは、非常に大規模なシステムというイメージがありますが、まずは、小規模なシステムからスタートし、性能が必要であれば、徐々に拡張させるとよいでしょう。Hadoop によるビッグデータ分析基盤導入の際には、ぜひこれらのクラスター構成例を参考にし、具体的な検討を行ってみてください。

Column　Hadoop の情報源

Hadoop の情報入手先は、非常に多岐に渡りますが、以下、Hadoop の導入検討前に知っておくべき情報源をピックアップします。Hadoop は、商用版のディストリビューションとして、MapR 社が提供する MapR CDP（MapR Converged Data Platform）、Cloudera 社が提供する CDH（Cloudera's Distribution including Apache Hadoop）、Hortonworks 社が提供する HDP（Hortonworks Data Platform）があるため、すべてのディストリビューションに関するリリース情報を確認し、サポートされている Hadoop 周辺ソフトウェアや新機能の特徴に目を通しておくとよいでしょう。

また、Hadoop のイベントには、さまざまなベンダーの事例情報がスライドで掲載されています。自社とまったく同じ課題を抱えた事例は、なかなか見つらないかもしれませんが、ベストプラクティスや、適用の勘所を知ることができるため、イベントの Web サイトで公開されているスライドやビデオ、そして、Hadoop ディストリビューターのブログ記事にも目を通しておきます。

一通りの最新情報を得ることができたら、ハードウェアベンダーが出しているリファレンスアーキテクチャの公開資料を参照すると、ハードウェア、ミドルウェア、アプリケーションを含む Hadoop 基盤の全体像を理解できます。特に、Apache Spark のリファレンスアーキテクチャは、Hadoop と Spark を組み合わせたインメモリ型の高速処理基盤の全体像を知ることができるため、必ず目を通しておきましょう。

表 2-8　Hadoop の情報源

Hadoop 導入前の確認項目	情報入手先
コミュニティ提供の最新リリース情報	http://hadoop.apache.org/releases.html
ロードマップ、開発スケジュール	https://cwiki.apache.org/confluence/display/HADOOP/Roadmap
ディストリビューター製品情報（MapR）	https://maprdocs.mapr.com/home/
ディストリビューター製品情報（Cloudera）	https://www.cloudera.com/documentation/enterprise/release-notes/topics/rg_release_notes.html
ディストリビューター製品情報（Hortonworks）	https://docs.hortonworks.com/
イベントのスライド集（Dataworks Summit）	https://www.slideshare.net/HadoopSummit/presentations
MapR 社提供のスライド集	https://www.slideshare.net/search/slideshow?searchfrom=header&q=MapR
MapR 社のブログ記事	https://community.mapr.jp/
MapR 社の動画集	https://www.youtube.com/user/maprtech/videos
Cloudera 社提供のスライド集	https://www.slideshare.net/Cloudera/presentations
Cloudera 社のブログ記事（CDH 関連）	http://blog.cloudera.com/blog/category/cdh/
Cloudera 社の動画集	https://www.youtube.com/user/clouderahadoop/videos

Hortonworks 社提供のスライド集	https://www.slideshare.net/search/slideshow?searchfrom=header&q=Hortonworks
Hortonworks 社のブログ記事	https://jp.hortonworks.com/blog/
Hortonworks 社の動画集	https://www.youtube.com/user/Hortonworks/videos
リファレンスアーキテクチャ（MapR）	https://mapr.com/resources/hp-reference-architecture-mapr-m7/assets/hp-reference-architecture-for-mapr-m7-on-hp-proliant-1-18.pdf
リファレンスアーキテクチャ（Cloudera）	https://www.cloudera.com/content/dam/www/marketing/resources/whitepapers/hpe-big-data-reference-architecture-for-apollo-2000-and-4200.pdf.landing.html
リファレンスアーキテクチャ（Hortonworks）	https://h20195.www2.hpe.com/V2/getpdf.aspx/4AA6-7444ENW.pdf
エコシステムソフトウェア（MapR）	https://maprdocs.mapr.com/home/c_ecosystem_intro.html
エコシステムソフトウェア（Cloudera）	https://www.cloudera.com/products/open-source/apache-hadoop/key-cdh-components.html
エコシステムソフトウェア（Hortonworks）	https://jp.hortonworks.com/ecosystems/
Spark を使ったリファレンスアーキテクチャ	https://h20195.www2.hpe.com/V2/getpdf.aspx/4AA6-8143ENW.pdf
Spark とインメモリ DB のリファレンスアーキテクチャ	https://h20195.www2.hpe.com/V2/getpdf.aspx/4AA6-7739ENW.pdf

第３章

ハードウェアの事前設定とHadoopのインストール

ビッグデータ分析基盤の構築に限らず、どのようなIT基盤であっても、ハードウェアとOSが適切に設定されていなければ、システムの性能を引き出すことはできません。特にHadoopクラスターでは、前提となるハードウェアとOSの設定チューニングを行うか否かで、Hadoopの処理能力を大きく左右します。

本章では、新たにHadoopクラスターの構築をするに当たって必要となる、ハードウェアの設定、OSの構成、チューニングポイントなどを押さえるとともに、Apache Hadoop 3とMapRのクラスター構築の具体的な構築手順について、注意点を交えながら解説します。

3-1 Hadoop クラスターハードウェアの設定

Hadoop クラスターは、汎用の x86 サーバーで構成可能ですが、サーバーの BIOS 設定を適切に行っておく必要があります。まず、BIOS やファームウェアの不安定な要素をなるべく排除するために、Hadoop クラスターのすべてのノードにおける BIOS およびファームウェアを最新バージョンにアップグレードしてください。

パフォーマンスが要求される Hadoop クラスターでは、ハードウェアの性能を最大限に引き出す BIOS 設定が必要です。ハードウェアの設定項目は、機種によって異なりますが、最近のサーバーでは、BIOS 設定において、性能に関するプロファイル設定（Power Profile）のメニューが用意されています。Hadoop の場合は、最大性能が出る「Maximum Performance」を設定します。また、動的に消費電力の管理が行えるパワーレギュレーターも搭載されていますが、Hadoop の場合は、性能を重視するため、静的な設定（Static High Performance mode）にします。メモリに関しても、省電力機能を利用せず、消費電力の上限を設けるパワーキャッピングも無効に設定します。CPU については、仮想化機能を無効にすることが推奨されています。

表 3-1 は、Xeon プロセッサを搭載した x86 サーバー（ビッグデータ用の HPE Apollo サーバーなど）の Hadoop クラスター用に必要なサーバーの BIOS 設定項目とパラメーター例の抜粋です。この設定を参考にして、使用するサーバーの BIOS 設定を行ってください。

表 3-1 サーバーの BIOS 設定項目とパラメーター例

BIOS 設定項目	パラメーター例 ‡
HPE Power Profile	Maximum Performance
HPE Power Regulator	Static High Performance mode
Intel_QPI_Link_Mgt	Disabled
Mem_Power_Saving	Max Perf
Thermal Configuration	Increased Cooling
Min_Proc_Idle Power Package state	No Package state
Energy/Performance Bios	Disabled
Collaborative Power Control	Disabled
Dynamic Power Capping Functionality	Disabled
DIMM Voltage Preference	Optimized for Performance
Intel Virtualization Technology	Disabled
Intel VT-d	Disabled

‡ 機種によって項目とパラメーターは異なります。上記は、代表的な HPE Apollo サーバーの例です。

3-1-1　RAID コントローラーの設定

RAID コントローラーでは、ストライプサイズ、アレイアクセラレータなどを調整します。Hadoop クラスターのワーカーノードのドライブは、RAID 0、あるいは、JBOD で構成しますが、それぞれのドライブのストライプサイズは、1024KB に設定します。また、RAID コントローラーの種類によって設定項目が異なりますが、データ用の論理ディスクについては、RAID に関するキャッシュ機能（HPE SmartArray コントローラーの場合は、「Array Acceleration/Caching」）を無効にします。一方、RAID コントローラー配下の OS 用の論理ディスクについては、RAID のキャッシュ機能を有効にします（表 3-2）。

表 3-2　RAID コントローラーの設定

設定項目	パラメーター例
Stripe Size	1024
Array Acceleration/Caching	off
RAID キャッシュ（データ領域）	disable
RAID キャッシュ（OS 領域）	enable

■ 擬似 RAID コントローラーの場合（HPE SmartArray B140i など）

サーバーのマザーボードに搭載されたチップセット内蔵の AHCI コントローラーを拡張利用する形式の擬似 RAID コントローラー（たとえば、HPE Dynamic SmartArray B140i コントローラーなど）は、必要に応じて内蔵 SATA ドライブの設定を有効にする必要があります。また、OS 向けの論理ディスクのキャッシュを有効にします（表 3-3）。

表 3-3　擬似 RAID コントローラーの設定

設定項目	パラメーター例
Embedded SATA Configuration	SATA_RAID_ENABLED
Enable caching on the OS logical drive	enable

3-2 Apache Hadoop 3 基盤の構築手順

ハードウェアの設定が終わったら、Linux ディストリビューションの選定、対応している OS バージョンの確認、OS のインストール、OS 関連のパラメーター設定、clush コマンドのインストールなどを行います。

3-2-1 Linux のインストールとファイルシステム

Apache Hadoop 3 を構成する OS としては、代表的な RHEL、CentOS、Ubuntu、SLES などが利用可能です。OS は、マスターノード（コントロールノード）とワーカーノード（データノード）でバージョンの種類とバージョンを統一します。ここでは、CentOS 7.4 を使用します。

Hadoop クラスターでは、性能の観点から、OS のパーティションとユーザーデータのパーティションを別々の物理ディスクで構成します。また、CentOS 7 の場合、ファイルシステムは、標準でサポートしている XFS で構成します。表 3-4 と表 3-5 に、CentOS 7.4 における Hadoop のパーティション例を示します。

表 3-4 マスターノードのパーティション例

パーティション	マウントポイント	ファイルシステムの種類	割り当てた容量
/dev/sda1	/boot	XFS	1GB
/dev/sda2	無し	swap	8GB
/dev/sda3	/	XFS	残りすべて

表 3-5 ワーカーノードのパーティション例

パーティション	マウントポイント	ファイルシステムの種類	割り当てた容量
/dev/sda1	/boot	XFS	1GB
/dev/sda2	無し	swap	8GB
/dev/sda3	/	XFS	残りすべて
/dev/sdb1	/data1	XFS	3TB
/dev/sdc1	/data2	XFS	3TB
/dev/sdd1	/data3	XFS	3TB
/dev/sde1	/data4	XFS	3TB

OS 用のディスクは、/dev/sda で構成し、HDFS 用のディスクは、/dev/sda 以外のディスク

（/dev/sdb、/dev/sdc、/dev/sdd など）で構成しています。OS 用のディスクの/dev/sda は、LVM で構成してもかまいませんが、データ用ディスクは、非 LVM で構成してください。本書では、OS として CentOS 7.4.1708（本書では、CentOS 7.4 と呼ぶことにします）をベースで解説します。表 3-6 に、マスターノード、および、ワーカーノードの CentOS 7.4 での設定項目を示します。

表 3-6　CentOS 7.4 での設定項目

CentOS 7.4 での設定項目	設定例
RPM パッケージ	最小構成
言語設定	英語
タイムゾーン	Hadoop 基盤を設置するデータセンターの所在地域（例：Asia/Tokyo）
root パスワード	設定
一般ユーザー	OS インストール後に useradd コマンドで作成
ディスクパーティション	表 2-4、表 2-5 を参照（データ用ディスクは、OS インストール後に設定）
Kdump の設定	有効
ネットワーク	固定 IP を付与（データ用、管理用の最低 2 つが必要）
ホスト名	FQDN で設定。ドメイン名を付けること（例：n0121.jpn.linux.hpe.com）

Column　Hadoop インストール前の OS のイメージバックアップ

Hadoop を構築する場合、複数ノードに同じ構成の Linux をインストールしますが、方法としては、大きく分けて以下の 2 通りがあります。

方法 1. 全ノードに Linux を自動インストールする
方法 2. イメージバックアップを他のノードにリストアする

方法 1 は、管理ノードに OS の自動インストールサーバーを構築しておき、他のワーカーノードを PXE ブートさせて Linux を全自動インストールする方法です。自動インストールの手法としては、Red Hat 系 OS であれば、Kickstart、SUSE 系 OS であれば AutoYast、Ubuntu 系であれば Preseed を使います。大規模クラスターなどで、OS の新規インストールを行う際によく用いられる方法です。

もう一方の方法 2 は、バックアップツールなどを使って、ワーカーノードの 1 台にインストールした OS のハードディスク全体をイメージファイルとして取得し、他のワーカーノードに OS

イメージをリストアする方法です。障害対応だけでなく、Hadoop 構築済みの OS イメージを同様の手順で取得すれば、ノード増設時の Hadoop 構築の手間を大幅に削減できるため、非常に有用です。一般に、Kickstart インストールサーバーやイメージバックアップサーバーは、管理ノードに構成します。

管理ツールとしては、さまざまなものが存在しますが、たとえば、HPE Insight CMU のように、Kickstart インストールサーバー、イメージバックアップ、イメージリストアのすべての機能が搭載されているソフトウェアを選択する場合（図 3-1）もあれば、オープンソースのソフトウェアで構成する場合もあります。

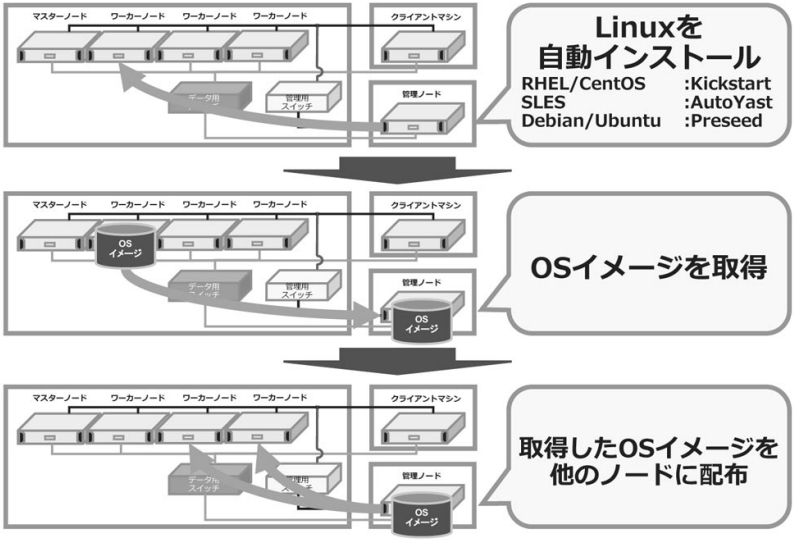


図 3-1　HPE Insight CMU による OS 配備、イメージバックアップ、リストア

3-2-2　Apache Hadoop 3 に必要な OS の事前設定

OS 側の基本的な設定が済んだら、Apache プロジェクトが提供している「Apache Hadoop 3」を CentOS 7.4 にインストールします。実際の本番環境においては、マスターノードのデータ保全性とサービスの可用性を考慮する必要がありますが、今回は、マスターノードの可用性を考慮しない構成（マスターノード 1 台の構成）で説明します。また、管理ノードとクライアントノードを

1 台で兼用します。

構築する Apache Hadoop 3 クラスターのシステム構成を図 3-2 に示します。

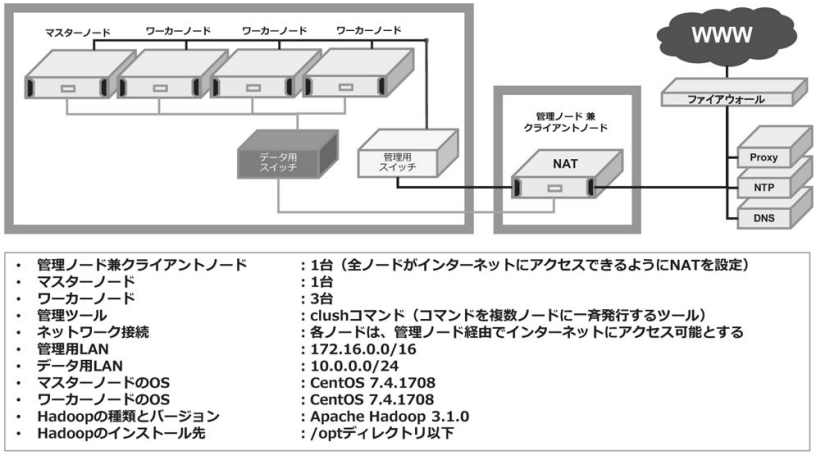


図 3-2　Apache Hadoop 3 クラスターのシステム構成

■ 固定 IP アドレスの設定

クライアントノードを含めた全ノードにおいて、管理用 NIC とデータ用 NIC に固定 IP アドレスを設定してください。今回は、172.16.1.X/16 が管理用ネットワーク、10.0.0.0/24 が Hadoop クラスターのデータ用ネットワークとします。

クライアントノードと Hadoop クラスターの全ノードが管理用ネットワーク、および、データ用ネットワークで TCP/IP 通信できなければなりません。ホスト名と IP アドレスの対応は、表 3-7（1）と表 3-7（2）に示したとおりです。

表 3-7（1） 管理用ホスト名と IP アドレスの対応

ノードの種類	管理用ホスト名	IP アドレス
クライアントノード	n0120-mgm.jpn.linux.hpe.com	172.16.1.120/16
マスターノード	n0121-mgm.jpn.linux.hpe.com	172.16.1.121/16
ワーカーノード	n0122-mgm.jpn.linux.hpe.com	172.16.1.122/16
ワーカーノード	n0123-mgm.jpn.linux.hpe.com	172.16.1.123/16
ワーカーノード	n0124-mgm.jpn.linux.hpe.com	172.16.1.124/16

表 3-7（2） データ用ホスト名と IP アドレスの対応

ノードの種類	データ用ホスト名	IP アドレス
クライアントノード	n0120.jpn.linux.hpe.com	10.0.0.120/24
マスターノード	n0121.jpn.linux.hpe.com	10.0.0.121/24
ワーカーノード	n0122.jpn.linux.hpe.com	10.0.0.122/24
ワーカーノード	n0123.jpn.linux.hpe.com	10.0.0.123/24
ワーカーノード	n0124.jpn.linux.hpe.com	10.0.0.124/24

■ ホスト名の設定

クライアントノードのホスト名を設定しておきます。ホスト名は、ドメイン名を含む FQDN で設定します。

```
# ip -4 addr show dev eth0
3: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    inet 172.16.1.120/16 brd 172.16.255.255 scope global eth0
       valid_lft forever preferred_lft forever

# hostnamectl set-hostname n0120.jpn.linux.hpe.com; hostname
n0120.jpn.linux.hpe.com
```

クライアントノードから全ノードに SSH 接続し、ホスト名を設定します。ホスト名は、ドメイン名を含む FQDN で設定します。

```
# ssh -l root 172.16.1.121 \
"hostnamectl set-hostname n0121.jpn.linux.hpe.com; hostname"
root@172.16.1.121's password: xxxxxxxxxxxx
n0121.jpn.linux.hpe.com

# ssh -l root 172.16.1.122 \
"hostnamectl set-hostname n0122.jpn.linux.hpe.com; hostname"
root@172.16.1.122's password: xxxxxxxxxxxx
n0122.jpn.linux.hpe.com

# ssh -l root 172.16.1.123 \
"hostnamectl set-hostname n0123.jpn.linux.hpe.com; hostname"
root@172.16.1.123's password: xxxxxxxxxxxx
n0123.jpn.linux.hpe.com

# ssh -l root 172.16.1.124 \
```

```
"hostnamectl set-hostname n0124.jpn.linux.hpe.com; hostname"
root@172.16.1.124's password: xxxxxxxxxxxx
n0124.jpn.linux.hpe.com
```

■ hosts ファイルの編集

クライアントノードの/etc/hosts ファイルを編集します†。以下、root ユーザーのコマンドプロンプトを「#」、一般ユーザーのコマンドプロンプトを「$」で表します。

```
# hostname
n0120.jpn.linux.hpe.com

# vi /etc/hosts
127.0.0.1   localhost localhost.localdomain localhost4 localhost4.localdomain4
::1         localhost localhost.localdomain localhost6 localhost6.localdomain6
172.16.1.120    n0120-mgm.jpn.linux.hpe.com     n0120-mgm
172.16.1.121    n0121-mgm.jpn.linux.hpe.com     n0121-mgm
172.16.1.122    n0122-mgm.jpn.linux.hpe.com     n0122-mgm
172.16.1.123    n0123-mgm.jpn.linux.hpe.com     n0123-mgm
172.16.1.124    n0124-mgm.jpn.linux.hpe.com     n0124-mgm
10.0.0.120      n0120.jpn.linux.hpe.com         n0120
10.0.0.121      n0121.jpn.linux.hpe.com         n0121
10.0.0.122      n0122.jpn.linux.hpe.com         n0122
10.0.0.123      n0123.jpn.linux.hpe.com         n0123
10.0.0.124      n0124.jpn.linux.hpe.com         n0124
```

† Hadoop クラスターの全ノードの/etc/resolv.conf ファイルで指定された DNS サーバーによる名前解決ができる場合は、/etc/hosts ファイルの編集は不要です。DNS サーバーによる名前解決ができない場合は、/etc/hosts ファイルの設定は必須です。

■ SSH 接続の設定

クライアントノードからパスワード入力なしで全ノードに SSH 接続ができるように設定します。

```
# rm -rf /root/.ssh/*
# ssh-keygen -f $HOME/.ssh/id_rsa -t rsa -N ''
# ls -a /root/.ssh
  .  ..  id_rsa  id_rsa.pub
```

鍵を各ノードにコピーし、パスワード入力なしで、クライアントノードから全ノードに SSH 接続できるように設定します。root ユーザーには、パスワードとして「password1234」が事前に設定されているとします。

```
# yum makecache fast && yum install -y sshpass
# sshpass -p "password1234" \
ssh-copy-id -f -o StrictHostKeyChecking=no root@n0120
# sshpass -p "password1234" \
ssh-copy-id -f -o StrictHostKeyChecking=no root@n0121
# sshpass -p "password1234" \
ssh-copy-id -f -o StrictHostKeyChecking=no root@n0122
# sshpass -p "password1234" \
ssh-copy-id -f -o StrictHostKeyChecking=no root@n0123
# sshpass -p "password1234" \
ssh-copy-id -f -o StrictHostKeyChecking=no root@n0124
# sshpass -p "password1234" \
ssh-copy-id -f -o StrictHostKeyChecking=no root@n0120-mgm
# sshpass -p "password1234" \
ssh-copy-id -f -o StrictHostKeyChecking=no root@n0121-mgm
# sshpass -p "password1234" \
ssh-copy-id -f -o StrictHostKeyChecking=no root@n0122-mgm
# sshpass -p "password1234" \
ssh-copy-id -f -o StrictHostKeyChecking=no root@n0123-mgm
# sshpass -p "password1234" \
ssh-copy-id -f -o StrictHostKeyChecking=no root@n0124-mgm
```

■ clustershell のインストール

クラスターでの設定作業の効率化を図るため、clustershell をクライアントノードにインストールします。管理対象ノードにコマンドを一斉発行するには、clush コマンドを利用します。

clush コマンドは、オプションでノードを指定できます。マスターノード 1 台のみ、複数ワーカーノード、全ノードなどというように、細かく管理対象のノードを指定し、それらに対して一斉にコマンドを発行できるため、大規模クラスターでは、非常に有用なツールです。

```
# yum clean all && yum makecache fast && yum install -y epel-release
# yum install -y clustershell
```

clush コマンドの管理対象ノードのグループを作成します。clush コマンドは、事前に設定した管理対象ノードのグループを設定しておけば、clush コマンドのオプションにそのグループ名を付与することで、一斉にコマンド発行する SSH 接続先の管理対象ノードを変更できます。たとえば、グループ名「all」を定義し、全クラスターノードをグループ「all」に所属させると、clush

コマンドでグループ名「all」を指定すれば、全クラスターノードに一斉に SSH 接続し、コマンドを実行できます。今回は、clush コマンドのグループを以下のように定義します。

表 3-8　clush コマンドのグループ

clush コマンドで使用するグループ名	管理対象ノードのホスト名
all	n0121、n0122、n0123、 n0124
cl	n0120
nn	n0121
dn	n0122、n0123、n0124
all-mgm	n0121-mgm、n0122-mgm、n0123-mgm、n0124-mgm
cl-mgm	n0120-mgm
nn-mgm	n0121-mgm
dn-mgm	n0122-mgm、n0123-mgm、n0124-mgm

■ clush コマンドで使用するグループの定義

Apache Hadoop 3 クラスターは、マスターノードとワーカーノードに分かれるため、clush コマンドで使用するグループを以下のように定義します。以下の clush コマンドの設定ファイル/etc/clustershell/groups 内では、ドメイン名を省略したホスト名を指定していますが、/etc/hosts ファイル、または、DNS サーバーでドメイン名を省略したホスト名の名前解決ができることが前提です。

```
# cat > /etc/clustershell/groups << __EOF__
all:           n0121,n0122,n0123,n0124
cl:      n0120
nn:            n0121
dn:                  n0122,n0123,n0124

all-mgm:           n0121-mgm,n0122-mgm,n0123-mgm,n0124-mgm
cl-mgm:  n0120-mgm
nn-mgm:            n0121-mgm
dn-mgm:                      n0122-mgm,n0123-mgm,n0124-mgm
__EOF__
```

■ clush コマンドによる一斉 SSH 接続の確認

パスワードを入力せずに SSH 接続ができるかどうかをテストします。また、全ノードのホスト名が、FQDN で表示されることを確認します。まずは、データ用 LAN 経由で全ノードに対して、一斉に SSH 接続し、clush コマンドに、-g オプションでグループ名を指定し、ホスト名を表示できるかを確認します。

表 3-9 に Hadoop クラスターでよく利用される clush コマンドの主なオプションとその使用例を掲載しておきます。

表 3-9　clush コマンドの主なオプションとその使用例

clush コマンドのオプション	オプションの意味	使用例
-g グループ名	グループ名を明示的に指定	clush -g cl "ls"
-g all	all グループを指定	clush -g all "ls"
-a	all グループを指定	clush -a "ls"
-g グループ 1, グループ 2	グループ名を複数指定	clush -g nn,dn "ls"
-w ホスト名	ホストを明示的に指定	clush -w n0131-mgm "ls"
-g グループ名 -L	グループでソート	clush -g dn -L "ls"
-aL	all グループでソート	clush -aL "ls"
-c ファイル --dest=ディレクトリ	ファイルをディレクトリにコピー	clush -a -c file --dest=/root/

クライアントノードに対しては、「-g cl」、全クラスターノードには、「-g all」を指定します。また、「-g cl,all」と指定すれば、クライアントノードと全クラスターノードを一度に指定できます。-L オプションにより、ノード名でソートします。

```
# clush -g cl,all -L hostname
n0120: n0120.jpn.linux.hpe.com
n0121: n0121.jpn.linux.hpe.com
n0122: n0122.jpn.linux.hpe.com
n0123: n0123.jpn.linux.hpe.com
n0124: n0124.jpn.linux.hpe.com
```

次に、管理用 LAN 経由で全ノードに対して一斉に SSH 接続し、ホスト名を表示できるかを確認します。

```
# clush -g cl-mgm,all-mgm -L hostname
n0120-mgm: n0120.jpn.linux.hpe.com
```

```
n0121-mgm: n0121.jpn.linux.hpe.com
n0122-mgm: n0122.jpn.linux.hpe.com
n0123-mgm: n0123.jpn.linux.hpe.com
n0124-mgm: n0124.jpn.linux.hpe.com
```

全ノードにSSH接続できることが確認できました。グループ名を使って、マスターノードのみ、および、ワーカーノードのみにSSH接続できるかも確認してください。

```
# clush -g nn hostname
n0121: n0121.jpn.linux.hpe.com

# clush -g nn-mgm hostname
n0121-mgm: n0121.jpn.linux.hpe.com

# clush -g dn -L hostname
n0122: n0122.jpn.linux.hpe.com
n0123: n0123.jpn.linux.hpe.com
n0124: n0124.jpn.linux.hpe.com

# clush -g dn-mgm -L hostname
n0122-mgm: n0122.jpn.linux.hpe.com
n0123-mgm: n0123.jpn.linux.hpe.com
n0124-mgm: n0124.jpn.linux.hpe.com
```

■ yumにおけるプロキシーサーバーの設定

プロキシーサーバー経由でパッケージを入手する場合は、Hadoopクラスターノードの`yum.conf`ファイルにプロキシーサーバーの設定が必要です。そこで、プロキシーサーバーの記述を設定した`/etc/yum.conf`ファイルを全ノードにコピーします。clushコマンドを使って、ファイルをコピーするには、`-c`オプションを付与し、コピー先のディレクトリ名を「`--dest=`」の後に指定します。

```
# echo "proxy=http://proxy.your.site.com:8080" >> /etc/yum.conf
# clush -g cl,all -c /etc/yum.conf --dest=/etc/
# clush -g cl,all -L "grep proxy /etc/yum.conf"
n0120: proxy=http://proxy.your.site.com:8080
...
```

■ /etc/hosts ファイルのコピー

/etc/hosts ファイルを全ノードにコピーします。

```
# clush -a -c /etc/hosts --dest=/etc/
```

■ カーネルパラメーターの設定

Hadoop クラスター向けのカーネルパラメーター† を設定します。スワップに関するカーネルパラメーター「vm.swappiness」を「1」に設定します。

```
# cat >> /etc/sysctl.conf << __EOF__
vm.swappiness=1
__EOF__

# clush -a -c /etc/sysctl.conf --dest=/etc/
# clush -g cl,all -L "sysctl -p | grep swappiness"
n0120: vm.swappiness = 1
...
```

† 今回は、クライアントノードとクラスターノードとのカーネルパラメーターを同一に設定します。

次に、Transparent Hugepage（以下、THP）† を無効にします。

† THP によって OS におけるメモリの管理単位となるページテーブルのサイズを変更できますが、Hadoop では、THP を無効にすることが推奨されています。

OS が提供する rc.local ファイルにパラメーターを設定するコマンドを記述することで、OS 起動時にパラメーターが自動的に設定されます†。

† rc.local ファイルの 1 行目には、「#!/bin/bash」の記述があることを確認してください。この 1 行がないと、OS 起動時の rc.local ファイルの実行に失敗します。

```
# cat >> /etc/rc.d/rc.local << __EOF__
echo never > /sys/kernel/mm/transparent_hugepage/defrag
echo never > /sys/kernel/mm/transparent_hugepage/enabled
__EOF__
```

```
# clush -a -c /etc/rc.d/rc.local --dest=/etc/rc.d/
# clush -g cl,all "chmod 755 /etc/rc.d/rc.local"
# clush -g cl,all -L "ls -l /etc/rc.d/rc.local"
n0120: -rwxr-xr-x 1 root root 586 Feb  9 15:53 /etc/rc.d/rc.local
...
```

プロセス数のソフトリミットとハードリミットに関する設定ファイル「90-nproc.conf」を作成します。

```
# cat > /etc/security/limits.d/90-nproc.conf << __EOF__
* soft nproc 65536
* hard nproc 65536
__EOF__
```

オープンできるファイル数のソフトリミットとハードリミットに関する設定ファイル「91-nofile.conf」を作成します。

```
# cat > /etc/security/limits.d/91-nofile.conf << __EOF__
* soft nofile 65536
* hard nofile 65536
__EOF__
```

作成した2つの設定ファイルを全ノードにコピーします。

```
# clush -a -c \
/etc/security/limits.d/90-nproc.conf \
--dest=/etc/security/limits.d/

# clush -a -c \
/etc/security/limits.d/91-nofile.conf \
--dest=/etc/security/limits.d/
```

■ ファイアウォールと SELinux の設定

今回は、ファイアウォールを OFF、SELinux を disabled にしておきます。

```
# clush -g cl,all "systemctl disable firewalld"
# clush -g cl,all "systemctl stop firewalld"

# clush -g cl,all \
sed -i \
```

```
"'s/SELINUX=enforcing/SELINUX=disabled/g' /etc/sysconfig/selinux"
```

SELinux の設定ファイル内に「SELINUX=disabled」が記述されているかを確認します。

```
# clush -g cl,all -L "grep ^SELINUX= /etc/sysconfig/selinux"
n0120: SELINUX=disabled
...
```

■ OS の再起動

全ノードを再起動します。

```
# hostname
n0120.jpn.linux.hpe.com

# clush -a reboot
# reboot
```

■ OS 関連のパラメーターの確認

OS 再起動後、引き続き、クライアントノードで作業します。設定した OS 関連のパラメーターを確認します。

```
# hostname
n0120.jpn.linux.hpe.com

# clush -g cl,all -L "cat /proc/sys/vm/swappiness"
n0120: 1
...

# clush -g cl,all -L "cat /sys/kernel/mm/transparent_hugepage/defrag"
n0120: always madvise [never]
...

# clush -g cl,all -L "cat /sys/kernel/mm/transparent_hugepage/enabled"
n0120: always madvise [never]
...
```

プロセス数に関するソフトリミットとハードリミットの設定が有効になっているかを確認します。

```
# clush -g cl,all -L "ulimit -Su"
n0120: 65536
...

# clush -g cl,all -L "ulimit -Hu"
n0120: 65536
...
```

オープンできるファイル数に関するソフトリミットとハードリミットの設定が有効になっているかを確認します。

```
# clush -g cl,all -L "ulimit -Sn"
n0120: 65536
...

# clush -g cl,all -L "ulimit -Hn"
n0120: 65536
...
```

全ノードの SELinux とファイアウォールが無効になっているかを確認します。

```
# clush -g cl,all -L "getenforce"
n0120: Disabled
...

# clush -g cl,all -L "systemctl status firewalld | grep Active"
n0120:    Active: inactive (dead)
...
```

■ 時刻同期の設定

Hadoop クラスターでは、全ノードで同じ時刻を刻むことが必要です。複数のサーバーが同じ時刻を刻むには、時刻同期の仕組みが必要です。時刻同期には、ntp や chrony が利用可能です。ここでは、社内 NTP サーバーのホスト名を `ntp.jpn.linux.hpe.com`[†] とします。まず、全ノードに chrony をインストールします。

```
# clush -g cl,all "yum clean all && yum makecache fast && yum install -y chrony"
```

> † DNS サーバーに ntp.jpn.linux.hpe.com が登録され、全ノードから DNS サーバーへの問い合わせによる名前解決ができなければなりません。

chrony.conf ファイルの「server X.centos.pool.ntp.org iburst」(X には、0 から 3 が入る)と記述されている 4 行の先頭に「#」を入れ、コメントアウトします。

```
# cp -a /etc/chrony.conf /etc/chrony.conf.org
# sed -i '/server /s/^/#/g' /etc/chrony.conf
```

さらに、設定ファイルの最下行に、同期する社内 NTP サーバーと、同期を許可する下位ネットワーク（172.16.0.0/16 と 10.0.0.0/24）を追記します。

```
# echo "server ntp.jpn.linux.hpe.com iburst " >> /etc/chrony.conf
# echo "allow 172.16.0.0/16" >> /etc/chrony.conf
# echo "allow 10.0.0.0/24" >> /etc/chrony.conf
```

設定ファイルを確認します。

```
# cat /etc/chrony.conf | grep -v ^# | grep -v ^$
driftfile /var/lib/chrony/drift
makestep 1.0 3
rtcsync
logdir /var/log/chrony
server ntp.jpn.linux.hpe.com iburst
allow 172.16.0.0/16
allow 10.0.0.0/24
```

設定ファイルを全ノードにコピーし、chronyd を起動します。

```
# clush -a -c /etc/chrony.conf --dest=/etc/

# clush -g cl,all "systemctl restart chronyd"
# clush -g cl,all -L "systemctl status chronyd | grep Active"
n0120:    Active: active (running) since Wed 2018-04-04 04:45:38 JST; 30min ago

# clush -g cl,all "systemctl enable chronyd"

# clush -g cl,all "chronyc sources"
n0120: 210 Number of sources = 1
n0120: MS Name/IP address         Stratum Poll Reach LastRx Last sample
n0120: ===============================================================================...
n0120: ^* ntp.jpn.linux.hpe.com   10      6    377   51    -119ns[-7401ns] +/-
```

```
    12us
...
```

上記の実行結果で、同期先の NTP サーバー「ntp.jpn.linux.hpe.com」の左側に「^*」が表示されていると NTP サーバーと時刻が同期できています。

ノード間で日付に大きな差異がある場合は、すべてのノードの時刻を強制的に設定します。以下は、2018 年 12 月 11 日 23 時 45 分に設定する例です。

```
# clush -g cl,all -L "date 121123452018; hwclock --systohc"
n0120: Tue Dec 11 23:45:00 JST 2018
...
```

Column　社内 NTP サーバーがうまく動かない

chrony によって社内 NTP サーバーを構築したにもかかわらず、時刻同期に失敗する原因としては、NTP サーバー側の問題も考えられます。NTP を使う分散システムは、一般に、時刻の提供側（NTP サーバー）と、同期する NTP クライアント側からなる階層構造をとります。この階層は、15 段からなり、最下位の 15 段を越えるマシンは、時刻同期において信頼度が低いとみなされ、時刻同期から除外（非同期状態）されます。

上位の NTP サーバーと同期ができない環境やインターネットから隔離された LAN 環境の場合、段数が 16 段とみなされるため、NTP クライアント側で chrony の設定を正しく行っても時刻が同期しないという状況に陥ってしまいます。

これに対処するには、NTP サーバーにおいて階層の段数を明示的に変更（偽装）します。具体的には、NTP サーバーの/etc/chrony.conf ファイルに以下のような設定を追記します。

```
local stratum 10
```

上記の場合、この設定を施した NTP サーバーは、階層の 10 段目とみなされ、下位のネットワークに時刻同期のサービスを提供し、NTP クライアントは、時刻同期を行えます。NTP サーバーは、上位の NTP サーバーと同期ができないため、ローカルの時刻を使って時刻同期のサービスを提供します。

3-2-3 Apache Hadoop 3 のインストール

OS の事前設定が完了したら、Hadoop クラスターの稼働に必要なパッケージをインストールします。

■ Java パッケージのインストール

マスターノードとワーカーノードすべてに Java をインストールする必要があります。Apache Hadoop 3 では、Java の 1.8 系以上がサポートされています。Java にはいくつか種類がありますが、今回は、CentOS 7 のリポジトリで提供されている Java を使用します。

```
# hostname
n0120.jpn.linux.hpe.com

# clush -g cl,all "yum makecache fast && yum install -y \
wget \
java-1.8.0-openjdk \
java-1.8.0-openjdk-devel"
```

■ ユーザーの追加

Hadoop クラスターを利用するユーザーを追加します。今回は、Hadoop のジョブを一般ユーザー koga で投入するシステムを想定します。ユーザーが複数必要な場合は、ここで必要なすべてのユーザーを追加しておきます。以下の例では、Hadoop クラスターを利用する「hadoop」というユーザーグループを作成し、hadoop グループに所属するユーザー koga、ユーザー yarn、ユーザー hdfs、ユーザー mapred を作成しています。パスワードは初期設定で「password1234」にしました。また、全ノードで UID と GID を一致させる必要があります。

```
# clush -g cl,all "groupadd -g 5000 hadoop"
# clush -g cl,all "useradd -g hadoop -u 5000 koga"
# clush -g cl,all "useradd -g hadoop -u 6001 yarn"
# clush -g cl,all "useradd -g hadoop -u 6002 hdfs"
# clush -g cl,all "useradd -g hadoop -u 6003 mapred"
# clush -g cl,all "echo 'password1234' | passwd koga --stdin"
```

利便性向上のため、一般ユーザーにおいても、公開鍵（`id_dsa.pub` ファイル）を使って、パスワード入力なしで全ノードに SSH 接続ができるように設定しておきます。

```
# hostname
n0120.jpn.linux.hpe.com

# su - koga
$ whoami
koga

$ rm -rf $HOME/.ssh/id_rsa
$ ssh-keygen -f $HOME/.ssh/id_rsa -t rsa -N ''
```

鍵を各ノードにコピーし、パスワード入力なしで、n0120 から全ノードに SSH 接続できるように設定します†。

```
$ sshpass -p "password1234" \
ssh-copy-id -o StrictHostKeyChecking=no koga@n0120
$ sshpass -p "password1234" \
ssh-copy-id -o StrictHostKeyChecking=no koga@n0121
$ sshpass -p "password1234" \
ssh-copy-id -o StrictHostKeyChecking=no koga@n0122
$ sshpass -p "password1234" \
ssh-copy-id -o StrictHostKeyChecking=no koga@n0123
$ sshpass -p "password1234" \
ssh-copy-id -o StrictHostKeyChecking=no koga@n0124
$ sshpass -p "password1234" \
ssh-copy-id -o StrictHostKeyChecking=no koga@n0120-mgm
$ sshpass -p "password1234" \
ssh-copy-id -o StrictHostKeyChecking=no koga@n0121-mgm
$ sshpass -p "password1234" \
ssh-copy-id -o StrictHostKeyChecking=no koga@n0122-mgm
$ sshpass -p "password1234" \
ssh-copy-id -o StrictHostKeyChecking=no koga@n0123-mgm
$ sshpass -p "password1234" \
ssh-copy-id -o StrictHostKeyChecking=no koga@n0124-mgm
$ exit
#
```

† 公開鍵を各ノードにコピーしてもパスワード入力のプロンプトが表示されてしまう場合は、/home/koga ディレクトリのパーミッションが 700 になっているかどうかを確認してください。また、sshpass コマンドによる設定を行った後、初回の SSH 接続時に、「Are you sure you want to continue connecting (yes/no)? 」と入力を求められる場合は、「yes」と入力してください。2 回目以降は、パスワード入力なしによる SSH 接続が可能になります。

■ 環境変数の設定

Hadoop に関する環境変数を設定します。今回、インストールするのは、Apache 版 Hadoop 3.1.0 の tarball です。Hadoop の tarball は、/opt ディレクトリに展開します。このため、環境変数の「HADOOP_HOME」は、/opt/hadoop-3.1.0 に設定します。環境変数は、Hadoop クラスターを管理する root アカウントと、Hadoop クラスターを利用するユーザーすべてに設定します。

CentOS 7 におけるユーザーの bash 環境変数は、$HOME/.bash_profile に記述します。以下のように、.bash_profile に Hadoop 関連の環境変数を記述します。プロキシーサーバー経由でインターネットにアクセスする場合は、「.bash_profile」に環境変数「http_proxy」と「https_proxy」を記述します。

```
# whoami
root

# cat > $HOME/.bash_profile << '__EOF__'
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi
export http_proxy=http://proxy.your.site.com:8080
export https_proxy=http://proxy.your.site.com:8080
export PATH=$PATH:$HOME/.local/bin:$HOME/bin
export LANG=en_US.utf8
export JAVA_HOME=/usr/lib/jvm/jre
export HADOOP_HOME=/opt/hadoop-3.1.0
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
export HADOOP_INSTALL=$HADOOP_HOME
export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_COMMON_HOME=$HADOOP_HOME
export HADOOP_HDFS_HOME=$HADOOP_HOME
export HADOOP_YARN_HOME=$HADOOP_HOME
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native
export HADOOP_OPTS="-Djava.library.path=$HADOOP_COMMON_LIB_NATIVE_DIR"
export JAVA_LIBRARY_PATH=$HADOOP_HOME/lib/native:$JAVA_LIBRARY_PATH
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HADOOP_HOME/lib/native
export PATH=$HADOOP_HOME/sbin:$HADOOP_HOME/bin:$PATH
__EOF__
```

作成した .bash_profile を全ノードのユーザーのホームディレクトリにコピーします。

```
# . $HOME/.bash_profile
# clush -a -c $HOME/.bash_profile --dest=$HOME/
# clush -g cl,all -c $HOME/.bash_profile --dest=/home/koga/
```

```
# clush -g cl,all "chown koga:hadoop /home/koga/.bash_profile"
# clush -g cl,all -L "ls -l /home/koga/.bash_profile"
n0120: -rw-r--r-- 1 koga hadoop 841 Apr 4 05:55 /home/koga/.bash_profile
...
```

$HOME/.bash_profile において、環境変数の JAVA_HOME には、インストールした Java のディレクトリ/usr/lib/jvm/jre を指定しています。/usr/lib/jvm/jre のシンボリックリンクがバイナリの実体にリンクされているかどうかを確認します。

```
# clush -g cl,all -L "ls -l /usr/lib/jvm/jre"
n0120: lrwxrwxrwx 1 root root 21 Sep 28 04:51 /usr/lib/jvm/jre -> /etc/alternat
ives/jre
...

# clush -g cl,all -L "ls -l /etc/alternatives/jre"
n0120: lrwxrwxrwx 1 root root 64 Jan 16 06:54 /etc/alternatives/jre -> /usr/lib
/jvm/java-1.8.0-openjdk-1.8.0.151-5.b12.el7_4.x86_64/jre
...
```

■ ワーカーノードのデータ用ディスクのフォーマット

ワーカーノードのデータ用ディスク/dev/sdb をフォーマットします。root アカウントでフォーマットします。まず、ワーカーノードの/dev/sdb の状態を確認します。

```
# hostname
n0120.jpn.linux.hpe.com

# whoami
root

# clush -g dn -L "export LANG=en_US.utf8; parted -ms /dev/sdb print"
n0122: BYT;
n0122: /dev/sdb:3001GB:scsi:512:512:gpt:ATA MB3000GCWDB:;
n0122: 1:1049kB:3001GB:3001GB:xfs:primary:;
...
```

ワーカーノードの/dev/sdb 内のマスターブートレコード、および、GPT パーティションに関する情報を削除するために、ワーカーノードに sgdisk コマンドをインストールします。sgdisk コマンドは、gdisk RPM パッケージに含まれています。

```
# clush -g dn "yum install -y gdisk"
```

ワーカーノードの/dev/sdb 内のマスターブートレコード、および、GPT パーティションに関する情報を削除します。

```
# clush -g dn "sgdisk -Z /dev/sdb"
```

ワーカーノードを再起動します。

```
# clush -g dn "reboot"
```

ワーカーノードの/dev/sdb に GPT パーティションのラベルを付与します。

```
# clush -g dn "parted -s /dev/sdb mklabel gpt"
```

ワーカーノードの/dev/sdb に XFS 用のプライマリパーティションを作成します。

```
# clush -g dn "parted -s /dev/sdb -- mkpart primary xfs 1 -1"
```

ワーカーノードの/dev/sdb に作成したパーティション/dev/sdb1 を XFS でフォーマットします†。

```
# clush -g dn "mkfs.xfs -f -i size=512 /dev/sdb1"
```

† 事前に/data がアンマウントされている必要があります。

■ /etc/fstab への追記

ワーカーノードの OS 起動時に/dev/sdb1 が/data ディレクトリに自動的にマウントされるように、ワーカーノード/etc/fstab ファイルを記述します。

```
# clush -g dn "mkdir /data"
# clush -g dn "echo '/dev/sdb1 /data xfs defaults 0 0' >> /etc/fstab"
# clush -g dn "grep /dev/sdb /etc/fstab"
n0122: /dev/sdb1 /data xfs defaults 0 0
...
```

Column　/etc/fstab の記述

/etc/fstab ファイルは、ワーカーノードによって OS の/boot パーティション、/パーティション、swap パーティションなどに割り当てられたデバイスの UUID が異なります。

```
# clush -g dn -L "cat /etc/fstab | grep -v ^# | grep -v ^$"
n0122: UUID=24aa604b-76d6-4b78-86ce-a9de2b7034b1 /     xfs    defaults   0 0
n0122: UUID=7a41373c-3017-4448-87a8-33aae1b99619 /boot xfs    defaults   0 0
n0122: UUID=219732bf-d477-455d-816b-6ae09eeb20bf swap  swap   defaults   0 0
n0122: /dev/sdb1 /data xfs defaults 0 0
...
```

/etc/fstab ファイルを全ワーカーノードで統一したい場合は、UUID ではなく、デバイス名で記述します。当然、論理デバイスのパーティション番号（/dev/sdaX）とパーティション名（/、/boot、swap など）の対応は、OS インストール時に全ノードで統一しておく必要があります。

```
# cat > /tmp/fstab << __EOF__
/dev/sda3 /      xfs    defaults    0 0
/dev/sda1 /boot  xfs    defaults    0 0
/dev/sda2 swap   swap   defaults    0 0
/dev/sdb1 /data  xfs    defaults    0 0
__EOF__
```

/etc/fstab ファイルは、記述ミスがあると OS の起動に失敗するため注意が必要です。まずは、テスト用のワーカーノード 1 台（以下の例では、n0122）で正常に OS が起動できることが確認できたら、残りの全ワーカーノードに fstab ファイルをコピーするという手順を踏むとよいでしょう。

```
# clush -g dn "cp /etc/fstab /etc/fstab.old"
# clush -w n0122 -c /tmp/fstab --dest=/etc/
# clush -w n0122 reboot
# clush -w n0122 "df -HT"
# clush -g dn -c /tmp/fstab --dest=/etc/
# clush -g dn reboot
# clush -g dn -L "df -HT"
```

全ワーカーノードの/data がマウントできるかを確認します。

```
# clush -g dn "mount /data"
# clush -g dn -L "df -HT | grep /data"
n0122: /dev/sdb1      xfs         3.0T   34M   3.0T    1% /data
```

```
...
```

以上で、データ用ディレクトリを全ワーカーノードにおいてマウントできました。全ワーカーノードの OS を再起動し、自動的に/data ディレクトリがマウントされていることを確認します。

```
# clush -g dn reboot
# clush -g dn -L "df -HT | grep /data"
n0122: /dev/sdb1      xfs          3.0T   34M   3.0T    1% /data
```

■ HDFS 用ディレクトリの作成

マスターノードおよび、ワーカーノードに/data/hadoop/hdfs ディレクトリを作成し、所有者を hdfs、所有グループを hadoop に変更します。また、hadoop グループに所属するユーザーに対して書き込み許可を与えるため、アクセス権限を 775 に設定します。

```
# clush -g nn,dn "mkdir -p /data/hadoop/hdfs"
# clush -g nn,dn "chown hdfs:hadoop /data/hadoop/hdfs"
# clush -g nn,dn "chmod 775 /data/hadoop/hdfs"
```

ファイルの所有者、所有グループ、所有権が正しく設定されているかを確認します。

```
# clush -g nn,dn -L "export LANG=en_US.utf8;ls -ld /data/hadoop/hdfs"
n0121: drwxrwxr-x 4 hdfs hadoop 27 Apr 4 19:37 /data/hadoop/hdfs
n0122: drwxrwxr-x 2 hdfs hadoop 6 Apr 4 20:33 /data/hadoop/hdfs
n0123: drwxrwxr-x 2 hdfs hadoop 6 Apr 4 20:33 /data/hadoop/hdfs
n0124: drwxrwxr-x 2 hdfs hadoop 6 Apr 4 20:33 /data/hadoop/hdfs
```

■ Hadoop の tarball の入手と展開

Hadoop の最新版の tarball を入手し、/opt に展開します。$HOME/.bash_profile 内に記述した「HADOOP_HOME=/opt/hadoop-3.1.0」と整合性がとれるように tarball を展開します。

```
# cd
# pwd
root
# wget https://www.apache.org/dist/hadoop/core/hadoop-3.1.0/hadoop-3.1.0.tar.gz
# md5sum ./hadoop-3.1.0.tar.gz
f036ebd3fa0ef66ee1819e351d15b6cb ./hadoop-3.1.0.tar.gz
```

```
# clush -g all -c ./hadoop-3.1.0.tar.gz --dest=$HOME/
# clush -g cl,all "tar xzvf $HOME/hadoop-3.1.0.tar.gz -C /opt/"
```

3-2-4 Hadoop の設定ファイルの作成

Apache Hadoop 3 の XML 形式の設定ファイルを作成します。主な XML 形式の設定ファイルは、表 3-10 のとおりです。

表 3-10 主な XML 形式の設定ファイル

XML ファイル	内容
core-site.xml	マスターノード、テンポラリディレクトリ、バッファサイズの指定など
yarn-site.xml	リソースマネージャの指定やメモリ割り当て容量など
hdfs-site.xml	レプリカ数の指定、HDFS のディレクトリの指定など
mapred-site.xml	分散処理の仕組みである「MapReduce」の各種パラメータの設定など

■ core-site.xml ファイルの作成

core-site.xml ファイルを作成します。設定ファイル内の fs.default.name の値に、「hdfs://マスターノード名:9000」を指定します。今回は、マスターノードが n0121 ですので、「hdfs://n0121.jpn.linux.hpe.com:9000」を指定します。環境変数の「$HADOOP_HOME」に設定されている値を確認します。

```
# clush -g cl,all -L ". $HOME/.bash_profile; echo $HADOOP_HOME"
n0120: /opt/hadoop-3.1.0
...
```

core-site.xml ファイルを core-site.xml.org としてコピーしておきます。

```
# clush -g cl,all ". $HOME/.bash_profile; \
cp $HADOOP_HOME/etc/hadoop/core-site.xml \
$HADOOP_HOME/etc/hadoop/core-site.xml.org"
```

オリジナルのファイルをコピーしたら、core-site.xml ファイルを作成します。

```
# cat > $HADOOP_HOME/etc/hadoop/core-site.xml << "__EOF__"
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
<property>
  <name>fs.defaultFS</name>
    <value>hdfs://n0121.jpn.linux.hpe.com:9000</value>
</property>
<property>
  <name>hadoop.tmp.dir</name>
    <value>/tmp/hadoop-${user.name}</value>
</property>
</configuration>
__EOF__
```

作成した core-site.xml ファイルを全ノードにコピーします。

```
# clush -a -c \
$HADOOP_HOME/etc/hadoop/core-site.xml \
--dest=$HADOOP_HOME/etc/hadoop/
```

全ノードの core-site.xml ファイル内に「hdfs://n0121.jpn.linux.hpe.com:9000」が記述されているかどうかを確認します。

```
# clush -g cl,all -L ". $HOME/.bash_profile; \
grep n0121.jpn.linux.hpe.com \
$HADOOP_HOME/etc/hadoop/core-site.xml"
n0120:     <value>hdfs://n0121.jpn.linux.hpe.com:9000</value>
...
```

■ yarn-site.xml ファイルの作成

次に、yarn-site.xml ファイルを作成します。Hadoop クラスター全体の資源管理を行う Resource Manager のノードを n0121.jpn.linux.hpe.com に指定するため、設定ファイル内のパラメーター「yarn.resourcemanager.hostname」に n0121.jpn.linux.hpe.com を記述します。

```
# clush -g cl,all ". $HOME/.bash_profile; \
cp $HADOOP_HOME/etc/hadoop/yarn-site.xml \
$HADOOP_HOME/etc/hadoop/yarn-site.xml.org"

# cat > $HADOOP_HOME/etc/hadoop/yarn-site.xml << "__EOF__"
```

```
<?xml version="1.0"?>
<configuration>
<property>
   <name>yarn.resourcemanager.hostname</name>
   <value>n0121.jpn.linux.hpe.com</value>
</property>
<property>
   <name>yarn.resourcemanager.address</name>
   <value>n0121.jpn.linux.hpe.com:8032</value>
</property>
<property>
  <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
</property>
<property>
  <name>yarn.nodemanager.env-whitelist</name>
<value>JAVA_HOME,HADOOP_COMMON_HOME,HADOOP_HDFS_HOME,HADOOP_CONF_DIR,CLASSPATH_
PREPEND_DISTCACHE,HADOOP_YARN_HOME,HADOOP_MAPRED_HOME</value>
</property>
<property>
  <name>yarn.nodemanager.pmem-check-enabled</name>
    <value>false</value>
</property>
<property>
  <name>yarn.nodemanager.vmem-check-enabled</name>
    <value>false</value>
</property>
<property>
  <name>yarn.log-aggregation-enable</name>
    <value>true</value>
</property>
<property>
  <name>yarn.nodemanager.remote-app-log-dir</name>
    <value>/tmp/logs</value>
</property>
<property>
  <name>yarn.resourcemanager.nodes.exclude-path</name>
    <value>/opt/hadoop-3.1.0/etc/hadoop/yarn.exclude</value>
</property>
</configuration>
__EOF__
```

作成した yarn-site.xml ファイルを全ノードにコピーします。

```
# clush -a -c \
```

```
$HADOOP_HOME/etc/hadoop/yarn-site.xml \
--dest=$HADOOP_HOME/etc/hadoop/
```

■ hdfs-site.xml ファイルの作成

hdfs-site.xml ファイルを作成します。Hadoop クラスターでは、通常 HDFS のレプリケーション数を 3 にします。レプリケーション数が 3 の場合、データはワーカーノードをまたいで 3 重化されて記録されます。レプリケーション数を 3 にするには、「dfs.replication」の値を 3 に指定します。さらに、HDFS を構成するマスターノードのディレクトリパスを dfs.name.dir に、ワーカーノードのパスを dfs.data.dir に指定します。今回は、マスターノードとワーカーノードの HDFS のディレクトリを表 3-11 のようにします。

表 3-11　HDFS のディレクトリ

ノードの種類	パス
マスターノード	/data/hadoop/hdfs/nn
ワーカーノード	/data/hadoop/hdfs/dn

```
# clush -g cl,all ". $HOME/.bash_profile; \
cp $HADOOP_HOME/etc/hadoop/hdfs-site.xml \
$HADOOP_HOME/etc/hadoop/hdfs-site.xml.org"

# cat > $HADOOP_HOME/etc/hadoop/hdfs-site.xml << "__EOF__"
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
<property>
 <name>dfs.replication</name>
 <value>3</value>
</property>
<property>
  <name>dfs.namenode.name.dir</name>
    <value>/data/hadoop/hdfs/nn</value>
</property>
<property>
  <name>dfs.datanode.data.dir</name>
    <value>/data/hadoop/hdfs/dn</value>
```

```
</property>
<property>
  <name>dfs.namenode.checkpoint.dir</name>
    <value>/data/hadoop/hdfs/snn</value>
</property>
<property>
  <name>dfs.namenode.checkpoint.edits.dir</name>
    <value>/data/hadoop/hdfs/snn</value>
</property>
<property>
  <name>dfs.hosts.exclude</name>
    <value>/opt/hadoop-3.1.0/etc/hadoop/dfs.exclude</value>
  </property>
</configuration>
__EOF__
```

作成した hdfs-site.xml ファイルを全ノードにコピーします。

```
# clush -a -c \
$HADOOP_HOME/etc/hadoop/hdfs-site.xml \
--dest=$HADOOP_HOME/etc/hadoop/
```

■ mapred-site.xml ファイルの作成

mapred-site.xml ファイルを作成します。Apache Hadoop 3 では、分散処理のフレームワークとして YARN を指定するため（YARN は、CPU やメモリなどのハードウェア資源のスケジューリングや分散処理基盤向けのアプリケーション開発のためのフレームワークです）、以下の設定を行います。今回は、JobHistoryServer サービスを n0121 で稼働させる設定ファイルの例です。

```
# clush -g cl,all ". $HOME/.bash_profile; \
cp $HADOOP_HOME/etc/hadoop/mapred-site.xml \
$HADOOP_HOME/etc/hadoop/mapred-site.xml.org"

# cat > $HADOOP_HOME/etc/hadoop/mapred-site.xml << "__EOF__"
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
<property>
  <name>mapreduce.framework.name</name>
   <value>yarn</value>
</property>
```

```
<property>
  <name>mapreduce.admin.user.env</name>
  <value>HADOOP_MAPRED_HOME=$HADOOP_COMMON_HOME</value>
</property>
<property>
  <name>yarn.app.mapreduce.am.env</name>
  <value>HADOOP_MAPRED_HOME=$HADOOP_COMMON_HOME</value>
</property>
<property>
  <name>mapreduce.map.java.opts</name>
  <value>-Xmx2560M</value>
</property>
<property>
  <name>mapreduce.reduce.java.opts</name>
  <value>-Xmx2560M</value>
</property>
<property>
 <name>mapreduce.jobhistory.address</name>
 <value>n0121.jpn.linux.hpe.com:10020</value>
</property>
</configuration>
__EOF__
```

作成した mapred-site.xml ファイルを全ノードにコピーします。

```
# clush -a -c \
$HADOOP_HOME/etc/hadoop/mapred-site.xml \
--dest=$HADOOP_HOME/etc/hadoop/
```

ワーカーノードのホスト名を記述したファイル「workers」を作成し、全ノードにコピーします。

```
# cat > $HADOOP_HOME/etc/hadoop/workers << __EOF__
n0122.jpn.linux.hpe.com
n0123.jpn.linux.hpe.com
n0124.jpn.linux.hpe.com
__EOF__

# clush -a -c \
$HADOOP_HOME/etc/hadoop/workers \
--dest=$HADOOP_HOME/etc/hadoop/
```

以上で、Apache Hadoop 3 の各種設定ファイルが準備できました。

3-2-5 HDFS の整備

HDFS を利用するための操作を行います。HDFS のフォーマットを行い、HDFS の稼働に必要な各種サービスを起動し、動作確認を行います。

■ HDFS のフォーマット

マスターノード上の hdfs コマンドを使って HDFS のフォーマットを行います。$HOME/.bash_profile に実行ファイルのパスが記述されているため、hdfs コマンドが利用可能です。hdfs コマンドに-format オプションを付与しフォーマットします。

```
# hostname
n0120.jpn.linux.hpe.com

# whoami
root

# clush -g nn ". $HOME/.bash_profile; which hdfs"
n0121: /home/koga/hadoop-3.1.0/bin/hdfs

# clush -g nn \
". $HOME/.bash_profile; \
hdfs namenode -format -force"
...
n0121: 2018-04-04 08:17:39,723 INFO common.Storage: Storage directory /data/had
oop/hdfs/nn has been successfully formatted.
...
n0121: /************************************************************
n0121: SHUTDOWN_MSG: Shutting down NameNode at n0121.jpn.linux.hpe.com/10.0.0.1
21
n0121: ************************************************************/
```

エラーがなければメッセージの途中に「Storage directory /data/hadoop/hdfs/nn has been successfully formatted.」と表示されて無事フォーマットが完了します。

Column　HDFS のフォーマットができない

HDFS のフォーマットに失敗する原因としては、Hadoop 用の XML ファイルの設定ミス、セキュリティの設定ミス、HDFS 用に作成したディレクトリの所有権、書き込み権限、所有グループの設定ミスなどが挙げられます。HDFS をフォーマットする場合は、そのユーザーが HDFS

用のディレクトリへの書き込み権限が必要です。/data ディレクトリは、hadoop グループに対して書き込み可能なパーミッション設定（chmod 775）が必要です。

```
# clush -a "ls -ld /data/hadoop/hdfs"
n0121: drwxrwxr-x 3 hdfs hadoop 16 Jan 18 20:37 /data/hadoop/hdfs
...
```

HDFS をフォーマットすると、hdfs-site.xml ファイルで HDFS として指定したディレクトリ配下に NameNode 用のディレクトリが生成されます。パーミッションは、所有者に対して書き込み可能なパーミッション設定（chmod 755）になっているかを確認してください。

■ NameNode サービスと DataNode サービスの起動

マスターノードでは、NameNode サービスと ResourceManager サービスを稼働させます。一方、ワーカーノードでは、DataNode サービスと NodeManager サービスを稼働させます。Hadoop 3.1.0 におけるサービスの起動は、hdfs コマンドに「--daemon」オプションを付与する方法が推奨されています。まずは、NameNode サービスと DataNode サービスを起動します。

```
# clush -g nn ". $HOME/.bash_profile; hdfs --daemon start namenode"
# clush -g dn ". $HOME/.bash_profile; hdfs --daemon start datanode"
```

NameNode における Java のプロセスが稼働しているかを確認します†。

```
# clush -g nn "jps | grep -vi jps"
n0121: 6507 NameNode

# clush -g dn -L "jps | grep -vi jps"
n0122: 4748 DataNode
n0123: 4376 DataNode
n0124: 5618 DataNode
```

† この時点で、Java のプロセス「NameNode」、あるいは「DataNode」が起動できない場合、/data/hadoop/hdfs ディレクトリ以下のファイルをすべて削除したうえで、再度、HDFS のフォーマットを行ってください。また、/data ディレクトリに対して、適切な所有者やパーミッションが設定されているかを確認してください。

マスターノード上に HDFS のメタデータ情報に関連するファイル群が格納されているかを確認します。

```
# clush -g nn "ls -1 /data/hadoop/hdfs/nn/current/"
n0121: edits_inprogress_0000000000000000001
n0121: fsimage_0000000000000000000
n0121: fsimage_0000000000000000000.md5
n0121: seen_txid
n0121: VERSION
```

■ ResourceManager サービス、NodeManager サービス、JobHistory Server サービスの起動

次に、ResourceManager サービス、NodeManager サービス、JobHistoryServer サービスを起動します。

```
# clush -g nn ". $HOME/.bash_profile; yarn --daemon start resourcemanager"
# clush -g dn ". $HOME/.bash_profile; yarn --daemon start nodemanager"
# clush -g nn ". $HOME/.bash_profile; mapred --daemon start historyserver"
```

マスターノードにおいて、ResourceManager と JobHistoryServer に関する Java のプロセスが稼働しているかを確認します。

```
# clush -g nn "jps | grep -vi jps"
n0121: 6743 ResourceManager
n0121: 7017 JobHistoryServer
n0121: 6507 NameNode
```

ワーカーノードにおいて、NodeManager の Java のプロセスが稼働しているかを確認します。

```
# clush -g dn -L "jps | grep -vi jps"
n0122: 1733 NodeManager
n0122: 1560 DataNode
n0123: 1714 NodeManager
n0123: 1541 DataNode
n0124: 1703 NodeManager
n0124: 1531 DataNode
```

■ HDFS の状態確認

マスターノード上で HDFS の状態を確認します。ワーカーノードが 3 ノード構成の場合は、「`Live datanodes (3)`」と表示されているかどうかを確認してください。

```
# clush -g nn ". $HOME/.bash_profile; hdfs dfsadmin -report"
n0121: Configured Capacity: 8997377544192 (8.18 TB)
n0121: Present Capacity: 8997276008448 (8.18 TB)
n0121: DFS Remaining: 8997275996160 (8.18 TB)
n0121: DFS Used: 12288 (12 KB)
n0121: DFS Used%: 0.00%
n0121: Under replicated blocks: 0
n0121: Blocks with corrupt replicas: 0
n0121: Missing blocks: 0
n0121: Missing blocks (with replication factor 1): 0
n0121: Pending deletion blocks: 0
n0121:
n0121: -------------------------------------------------
n0121: Live datanodes (3):
n0121:
...
...
```

■ HDFS の/tmp ディレクトリの設定

Hadoop クラスターでは、YARN アプリケーションの実行時に HDFS の/tmp に書き込み権限が必要です。一般ユーザーでも/tmp に書き込みができるようにアクセス権を設定します。

```
# hdfs dfs -chmod -R 1777 /tmp
# hdfs dfs -ls /
Found 1 items
drwxrwxrwt   - root supergroup          0 2018-04-04 09:10 /tmp
```

■ HDFS の容量確認

Hadoop クラスターの HDFS の容量を確認します。

```
# hdfs dfs -df -h
Filesystem                              Size    Used  Available  Use%
hdfs://n0121.jpn.linux.hpe.com:9000  8.2 T  24.8 M      8.2 T    0%
```

上記より、容量が 8.2TB の HDFS が確保されていることがわかります。

■ HDFS へのファイルのコピー

テスト用のファイル「file01」を作成し、Hadoop クラスターが提供する HDFS にコピーできるかテストします。

```
# dd if=/dev/zero of=$HOME/file01 bs=1024k count=10
# hdfs dfs -mkdir /dir01
# hdfs dfs -put $HOME/file01 /dir01/
```

HDFS 上に file01 が存在するかを確認します。

```
# hdfs dfs -ls /dir01/
Found 1 items
-rw-r--r--   3 root supergroup   10485760 2018-04-14 08:26 /dir01/file01
```

HDFS の容量が消費されているかを確認します。

```
# hdfs dfs -df -h
Filesystem                                Size    Used  Available  Use%
hdfs://n0121.jpn.linux.hpe.com:9000  8.2 T  55.1 M      8.2 T    0%
```

先ほどに比べて「Used」の値が増えていることがわかります。以上で、HDFS へのファイルの保管が確認できました。さらに、Hadoop マスターノードが提供する GUI 画面（http://n0121-mgm:9870）にアクセスし、「DFS Used」の項目に消費されているファイルサイズが変化しているかを確認してください (図 3-3)。

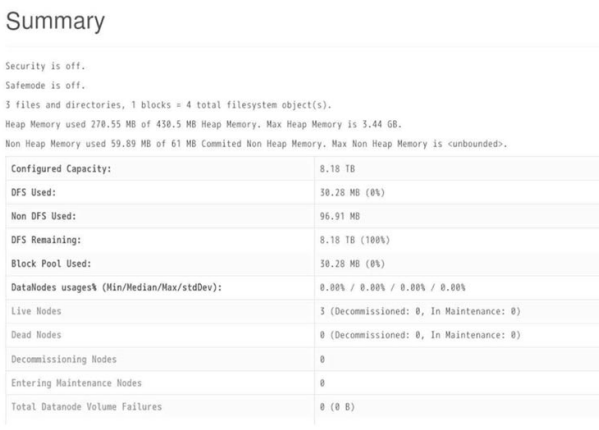


図 3-3　Hadoop の GUI 管理画面

■ 一般ユーザーによる Hadoop クラスターの利用

クライアントノードの一般ユーザー koga から hdfs コマンドを使って Hadoop クラスターを利用します。HDFS 上に一般ユーザー用のディレクトリを作成し、所有権とアクセス権限を設定します。

```
# hostname
n0120.jpn.linux.hpe.com

# hdfs dfs -mkdir -p /user/koga
# hdfs dfs -chown koga:supergroup /user/koga
# hdfs dfs -chmod 700 /user/koga
# hdfs dfs -ls /user/
Found 1 items
drwx------   - koga supergroup          0 2018-04-14 08:33 /user/koga
```

一般ユーザー koga で HDFS の容量確認とファイルの作成を行います。すでに、一般ユーザー koga の$HOME/.bash_profile に Hadoop に関連する環境変数が設定されているため、一般ユーザーでも hdfs コマンドが利用可能です。

```
# su - koga
$ whoami
koga

$ which hdfs
/opt/hadoop-3.1.0/bin/hdfs

$ hdfs dfs -df -h
Filesystem                              Size  Used  Available  Use%
hdfs://n0121.jpn.linux.hpe.com:9000  8.2 T  12 K      8.2 T    0%
```

テスト用のファイル「file01」を作成し、Hadoop クラスターが提供する HDFS にコピーできるかテストします。

```
$ dd if=/dev/zero of=$HOME/file01 bs=1024k count=10
$ hdfs dfs -mkdir -p /user/koga/dir01
$ hdfs dfs -put $HOME/file01 /user/koga/dir01/
```

HDFS 上にテスト用のファイルが存在するかを確認します。

```
$ hdfs dfs -ls /user/koga/dir01/
Found 1 items
```

```
-rw-r--r--   3 koga supergroup   10485760 2018-04-14 08:39 /user/koga/dir01/file01

$ exit
#
```

以上で、一般ユーザーでも HDFS が利用できることが確認できました。

3-2-6　YARN アプリケーションの実行確認

次に、YARN アプリケーションが実行できるかをテストします。Hadoop クラスターでは、ノードの時刻に大きな差があると、アプリケーションの実行に失敗します。念のため、ジョブ投入前に、ノードの時刻に大きなズレがないかを確認します。

```
# whoami
root

# clush -g cl,all -L date
n0120: Sat Apr 14 08:43:24 JST 2018
n0121: Sat Apr 14 08:43:24 JST 2018
...
```

さらに、Hadoop クラスターの HDFS がセーフモードになっていないことを確認します†。

```
# hdfs dfsadmin -safemode get
Safe mode is OFF
```

> †　ワーカーノードにおいて、DataNode サービスと NodeManager サービスの起動直後は、HDFS のセーフモードが ON になっていますが、時間が経過すると自動的に OFF になります。

時刻が同期されており、HDFS がセーフモードになっていないことが確認できたので、YARN アプリケーションを実行します。今回は、モンテカルロシミュレーションを使って円周率の近似値を求めるデモプログラムを実行します。ユーザー koga で、モンテカルロシミュレーションのサンプルプログラムをクライアントノード（n0120.jpn.linux.hpe.com）から実行します。

```
# hostname
n0120.jpn.linux.hpe.com

# su  - koga
```

```
$ whoami
koga
```

Apache Hadoop 3 では、モンテカルロシミュレーションによる円周率の近似値を計算するサンプルプログラムが用意されていますので、hadoop コマンドを使ってアプリケーションを実行します。

```
$ which hadoop
/opt/hadoop-3.1.0/bin/hadoop

$ hadoop jar \
$HADOOP_HOME/share/hadoop/mapreduce/\
hadoop-mapreduce-examples-3.1.0.jar pi 5 1000
...
Job Finished in 23.661 seconds
Estimated value of Pi is 3.14160000000000000000
```

モンテカルロシミュレーションによる円周率（近似値）の計算結果は、「Estimated value of Pi is 3.14...」と表示されます。以上で、Apache Hadoop 3 のクラスター上で YARN アプリケーションが実行できることが確認できました。

Note　YARN ジョブ実行時のメモリ不足のエラー

Hadoop のジョブを実行中に「Container [pid=...] is running beyond virtual memory limits.」という警告メッセージが表示される場合は、mapred-site.xml ファイルの「mapreduce.map.java.opts」パラメーターと「mapreduce.reduce.java.opts」パラメーターの値に指定しているメモリ容量の指定に誤りがあるか、あるいは、パラメーターそのものの指定が欠落しているなどが考えられます。これら 2 つのパラメーターは、ジョブの実行時に必要となりますので、本書に記載したパラメーターの記述例を参考に、適切な値をセットしてください。また実行するアプリケーションによっては、使用するメモリサイズを増やす必要がありますので、アプリケーションの稼働を正常に行うためにも、サーバーの物理メモリは、余裕を持って搭載してください。

Column　時刻のズレ

Hadoop に限らず、複数サーバーで協調して稼働するソフトウェアでは、サーバーが刻む時刻のズレが許されないものが少なくありません。Hadoop クラスターにおいても、全ノードは、NTP サーバーによる時刻同期により、同じ時間を刻まなければなりません。もしノード間の時刻に差がある場合は、Hadoop クラスターのジョブの投入に失敗し、以下のようなエラーメッ

セージが表示されます。

```
2017-12-27 03:43:21,410 INFO mapreduce.Job: Job job_1514311606815_0001 fai
led with state FAILED due to: Application application_1514311606815_0001
failed 2 times due to Error launching appattempt_1514311606815_0001_000002.
 Got exception: org.apache.hadoop.yarn.exceptions.YarnException: Unauthori
zed request to start container.This token is expired. current time is 1514
314072220 found 1514312682564
```

NTP サーバーが存在しない場合は、`date` コマンドで時刻を合わせても徐々にズレが生じますので、必ず NTP サーバーを構築し、Hadoop クラスターの全ノードで時刻が同期できるように設定してください。

3-3 MapR 版 Hadoop 基盤の構築手順

MapR 社が提供している「MapR 6.0」を CentOS 7.4 にインストールします。Linux の選定、対応している OS バージョンの調査、OS のインストール、OS 関連のパラメーター設定、`clush` コマンドのインストールを行います。その後、MapR のインストールを行い、簡単なジョブを投入し、動作確認を行います。

3-3-1 Linux の選定と OS バージョン

MapR クラスターを構成する OS としては、代表的な RHEL、CentOS、Ubuntu、SLES などが利用可能です。MapR 6.0 では、**表 3-12** に示す OS がサポートされています[*1]。

表 3-12 MapR のサポート OS

OS の種類	MapR 6.0 でサポートされる OS バージョン
Red Hat Enterprise Linux（64 ビット）	7.2, 7.3, 7.4
CentOS（64 ビット）	7.2.1511, 7.3.1611, 7.4.1708
Ubuntu Server（64 ビット）	14.04, 16.04
SUSE Linux Enterprise Server（64 ビット）	12 SP2
Oracle Enterprise Linux（64 ビット）‡	7.2, 7.3, 7.4

＊1 MapR のサポート OS バージョンについては、以下の URL に情報が掲載されています。
`https://maprdocs.mapr.com/home/InteropMatrix/r_os_matrix_6.x.html?hl=system,matrix`

‡ MapR において、Oracle Enterprise Linux は、MapR 6.0 のコアのみがサポートされています。さまざまな周辺ソフトウェアで構成されるエコシステムコンポーネントはサポートされていません。

OS の種類とバージョンは、全 C-a-D ノードで統一します。

3-3-2 Hadoop を考慮した CentOS 7 のインストールとファイルシステム

MapR クラスターでは、OS のパーティションとユーザーデータのパーティションを別々の物理ディスクで構成します。また、CentOS 7 の場合、標準でサポートしている XFS で構成します。表 3-13 は、CentOS 7.4 における Hadoop のパーティション例です。

表 3-13 C-a-D ノードのパーティション例

パーティション	マウントポイント	ファイルシステムの種類	割り当てた容量
/dev/sda1	/boot	XFS	1GB
/dev/sda2	無し	swap	8GB
/dev/sda3	/	XFS	残りすべて
/dev/sdb1	/data1	XFS	3TB
/dev/sdc1	/data2	XFS	3TB
/dev/sdd1	/data3	XFS	3TB

OS 用のディスクは、/dev/sda、MapR-FS 用のディスクは、/dev/sda 以外のディスク（/dev/sdb、/dev/sdc、...）で構成しています。OS 用のディスクの/dev/sda は、LVM で構成してもかまいませんが、データ用ディスクは、非 LVM で構成してください。なお、/tmp ディレクトリ、/opt ディレクトリ、スワップ領域の容量には注意が必要です。

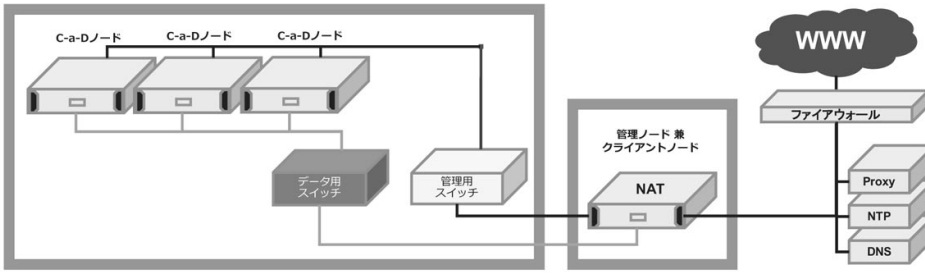
/tmp ディレクトリは、最低 10GB、/opt ディレクトリは、最低 128GB が必要です。これを下回ると、クラスターの動作やジョブの実行に影響が出るおそれがあります。今回は、C-a-D ノードの非常に簡素なパーティション構成例のため、/tmp と/opt を/パーティションに含めていますが、/パーティションの空き容量がなくなると、OS の動作に影響が出るため、これを防ぐために、/tmp と/opt を個別のパーティションに分ける場合もあります。

スワップ領域は、最低 4GB 以上で、かつ、物理メモリの 10%を下回らないようにします。たとえば、物理メモリが 128GB であれば、スワップ領域は、12.8GB を下回らないようにしなければなりません。さらに、別途、ZooKeeper ノードを設ける場合は、/opt/mapr/zkdata パーティショ

ンを設けます。/opt/mapr/zkdata パーティションは、少なくとも 500MB 必要です。

3-3-3 MapR に必要な OS の事前設定

今回、構築する MapR クラスターのシステム構成は、図 3-4 のようになります。Apache Hadoop 3 の構成と同様に、マスターノードの可用性は考慮せず、管理ノードとクライアントノードを 1 台で兼用します。また、作業の高効率化を図るため、管理ノードにコマンドを一斉発行できる clush コマンドをインストールします。



- 管理ノード兼クライアントノード　：1台（全ノードがインターネットにアクセスできるようにNATを設定）
- C-a-Dノード　：3台
- 管理ツール　：clushコマンド（コマンドを複数ノードに一斉発行するツール）
- ネットワーク接続　：各ノードは、管理ノード経由でインターネットにアクセス可能とする
- 管理用LAN　：172.16.0.0/16
- データ用LAN　：10.0.0.0/24
- C-a-DノードのOS　：CentOS 7.4.1708
- Hadoopの種類とバージョン　：MapR 6.0.1
- Hadoopのインストール先　：/optディレクトリ以下

図 3-4　MapR クラスターのシステム構成

クラスターノードは、コントロールノードとデータノードを兼用したコントロール・アズ・データノード（C-a-D ノード）で構成します。

■ 固定 IP アドレスの設定

クライアントノードを含めた全ノードにおいて、管理用 NIC とデータ用 NIC に固定 IP アドレスを設定してください。今回は、172.16.1.X/16 が管理用ネットワーク、10.0.0.0/24 が Hadoop クラスターのデータ用ネットワークとします。クライアントノードと Hadoop クラスターの全ノードが管理用ネットワーク、および、データ用ネットワークで TCP/IP 通信できることが前提条件です。ホスト名と IP アドレスの対応は、表 3-14（1）と表 3-14（2）のとおりです。

表 3-14（1） 管理用ホスト名と IP アドレスの対応

ノードの種類	管理用ホスト名	IP アドレス
クライアントノード	n0130-mgm.jpn.linux.hpe.com	172.16.1.130/16
C-a-D ノード	n0131-mgm.jpn.linux.hpe.com	172.16.1.131/16
C-a-D ノード	n0132-mgm.jpn.linux.hpe.com	172.16.1.132/16
C-a-D ノード	n0133-mgm.jpn.linux.hpe.com	172.16.1.133/16

表 3-14（2） データ用ホスト名と IP アドレスの対応

ノードの種類	データ用ホスト名	IP アドレス
クライアントノード	n0130.jpn.linux.hpe.com	10.0.0.130/24
C-a-D ノード	n0131.jpn.linux.hpe.com	10.0.0.131/24
C-a-D ノード	n0132.jpn.linux.hpe.com	10.0.0.132/24
C-a-D ノード	n0133.jpn.linux.hpe.com	10.0.0.133/24

■ ホスト名の設定

クライアントノードのホスト名を設定しておきます。ホスト名は、ドメイン名を含む FQDN で設定します。

```
# hostnamectl set-hostname n0130.jpn.linux.hpe.com; hostname
n0130.jpn.linux.hpe.com
```

クライアントノードから全ノードに SSH 接続し、ホスト名を設定します。ホスト名は、ドメイン名を含む FQDN で設定します。

```
# ssh -l root 172.16.1.131 \
"hostnamectl set-hostname n0131.jpn.linux.hpe.com; hostname"
root@172.16.1.131's password: xxxxxxxxxxxx
n0131.jpn.linux.hpe.com

# ssh -l root 172.16.1.132 \
"hostnamectl set-hostname n0132.jpn.linux.hpe.com; hostname"
root@172.16.1.132's password: xxxxxxxxxxxx
n0132.jpn.linux.hpe.com

# ssh -l root 172.16.1.133 \
"hostnamectl set-hostname n0133.jpn.linux.hpe.com; hostname"
root@172.16.1.133's password: xxxxxxxxxxxx
n0133.jpn.linux.hpe.com
```

■ hosts ファイルの編集

全ノードの/etc/hosts ファイルを編集します†。

> † Hadoop クラスターの全ノードの/etc/resolv.conf ファイルで指定された DNS サーバーによる名前解決ができる場合は、/etc/hosts ファイルの編集は不要です。DNS サーバーによる名前解決ができない場合は、/etc/hosts ファイルの設定は必須です。

```
# hostname
n0130.jpn.linux.hpe.com

# vi /etc/hosts
127.0.0.1   localhost localhost.localdomain localhost4 localhost4.localdomain4
::1         localhost localhost.localdomain localhost6 localhost6.localdomain6
172.16.1.130    n0130-mgm.jpn.linux.hpe.com     n0130-mgm
172.16.1.131    n0131-mgm.jpn.linux.hpe.com     n0131-mgm
172.16.1.132    n0132-mgm.jpn.linux.hpe.com     n0132-mgm
172.16.1.133    n0133-mgm.jpn.linux.hpe.com     n0133-mgm
10.0.0.130      n0130.jpn.linux.hpe.com         n0130
10.0.0.131      n0131.jpn.linux.hpe.com         n0131
10.0.0.132      n0132.jpn.linux.hpe.com         n0132
10.0.0.133      n0133.jpn.linux.hpe.com         n0133
```

■ SSH 接続の設定

クライアントノードからパスワード入力なしで全ノードに SSH 接続ができるように設定します。

```
# rm -rf /root/.ssh/*
# ssh-keygen -f $HOME/.ssh/id_rsa -t rsa -N ''
# ls -a /root/.ssh
  .  ..  id_rsa  id_rsa.pub
```

クライアントノードから鍵を各ノードにコピーし、パスワード入力なしで全ノードに SSH 接続できるように設定します。root ユーザーには、パスワードとして「password1234」が事前に設定されているとします。クライアントノードの n0130 は、自分自身にも鍵をコピーし、パスワードなしでローカルに SSH 接続できるようにしておきます。

```
# yum makecache fast && yum install -y sshpass
# sshpass -p "password1234" \
ssh-copy-id -f -o StrictHostKeyChecking=no root@n0130
```

```
# sshpass -p "password1234" \
ssh-copy-id -f -o StrictHostKeyChecking=no root@n0131
# sshpass -p "password1234" \
ssh-copy-id -f -o StrictHostKeyChecking=no root@n0132
# sshpass -p "password1234" \
ssh-copy-id -f -o StrictHostKeyChecking=no root@n0133
# sshpass -p "password1234" \
ssh-copy-id -f -o StrictHostKeyChecking=no root@n0130-mgm
# sshpass -p "password1234" \
ssh-copy-id -f -o StrictHostKeyChecking=no root@n0131-mgm
# sshpass -p "password1234" \
ssh-copy-id -f -o StrictHostKeyChecking=no root@n0132-mgm
# sshpass -p "password1234" \
ssh-copy-id -f -o StrictHostKeyChecking=no root@n0133-mgm
```

■ clustershell のインストール

全ノードに一斉にコマンドを発行できるツール「clustershell」をクライアントノードにインストールします。

```
# yum install -y epel-release
# yum install -y clustershell
```

clush コマンドの管理対象ノードのグループを作成します。今回は、clush コマンドのグループを以下のように定義します。

表 3-15　clush コマンドのグループ

clush コマンドで使用するグループ名	管理対象ノードのホスト名
all	n0131, n0132, n0133
cl	n0130
cldb	n0131, n0132, n0133
zk	n0131, n0132, n0133
rm	n0131, n0132, n0133
nfs	n0131, n0132, n0133
web	n0133
hs	n0131
all-mgm	n0131-mgm, n0132-mgm, n0133-mgm
cl-mgm	n0130-mgm
cldb-mgm	n0131-mgm, n0132-mgm, n0133-mgm

zk-mgm	n0131-mgm, n0132-mgm, n0133-mgm
rm-mgm	n0131-mgm, n0132-mgm, n0133-mgm
nfs-mgm	n0131-mgm, n0132-mgm, n0133-mgm
web-mgm	n0133-mgm
hs-mgm	n0131-mgm

■ clush コマンドで使用するグループの定義

clush コマンドで使用するグループを以下のように定義します。以下の clush コマンドの設定ファイル/etc/clustershell/groups 内では、ドメイン名を省略したホスト名を指定していますが、/etc/hosts ファイル、または、DNS サーバーでドメイン名を省略したホスト名でも名前解決できることが前提です。

```
# cat > /etc/clustershell/groups << __EOF__
all:        n0131,n0132,n0133
cl:   n0130
cldb:       n0131,n0132,n0133
zk:         n0131,n0132,n0133
rm:         n0131,n0132,n0133
nfs:        n0131,n0132,n0133
web:                  n0133
hs:         n0131

all-mgm:            n0131-mgm,n0132-mgm,n0133-mgm
cl-mgm:   n0130-mgm
cldb-mgm:           n0131-mgm,n0132-mgm,n0133-mgm
zk-mgm:             n0131-mgm,n0132-mgm,n0133-mgm
rm-mgm:             n0131-mgm,n0132-mgm,n0133-mgm
nfs-mgm:            n0131-mgm,n0132-mgm,n0133-mgm
web-mgm:                              n0133-mgm
hs-mgm:             n0131-mgm
__EOF__
```

■ clush コマンドによる一斉 SSH 接続の確認

パスワード入力せずに SSH 接続ができるかどうかをテストします。また、全ノードのホスト名が、FQDN で表示されることを確認します。

```
# clush -g cl,all -L hostname
n0130: n0130.jpn.linux.hpe.com
n0131: n0131.jpn.linux.hpe.com
n0132: n0132.jpn.linux.hpe.com
n0133: n0133.jpn.linux.hpe.com

# clush -g cl-mgm,all-mgm -L hostname
n0130-mgm: n0130.jpn.linux.hpe.com
n0131-mgm: n0131.jpn.linux.hpe.com
n0132-mgm: n0132.jpn.linux.hpe.com
n0133-mgm: n0133.jpn.linux.hpe.com
```

全ノードに SSH 接続できることが確認できました。グループ名を使って、SSH 接続できるかも確認してください。

この後は、yum におけるプロキシーサーバーの設定、/etc/hosts ファイルのコピー、カーネルパラメーターの設定、ファイアウォールと SELinux の無効化、時刻同期と日付の設定を行います。これらの作業は、Apache Hadoop 3 クラスターの構築手順と同じ方法で設定します。ただし、MapR クラスターでは、各ノードの OS のカーネルパラメーター「vm.swappiness」を 10 に設定することが推奨[*2]されていますので注意してください。

3-3-4 MapR 6.0 のインストール

OS の事前設定が完了したら、MapR をインストールします。MapR クラスターの稼働に必要な RPM パッケージをインストールしておきます。

```
# clush -g cl,all "yum install -y \
curl device-mapper iputils libsysfs lsof lvm2 nc nfs-utils openssl \
perl python-devel sdparm sudo syslinux sysstat wget which yum-utils\
java-1.8.0-openjdk java-1.8.0-openjdk-devel"
```

＊2 MapR クラスターにおけるカーネルパラメーターの「vm.swappiness」の推奨値に関する情報：
https://maprdocs.mapr.com/home/AdvancedInstallation/PreparingEachNode-memory.html

■ ユーザーの追加

mapr ユーザーと mapr グループを作成します。パスワードは初期設定で「password1234」にしました。また、全ノードで UID と GID を一致させる必要があります。クラスターノードだけでなく、クライアントノードにもユーザーを作成します。

```
# clush -g cl,all "groupadd -g 5000 mapr"
# clush -g cl,all "useradd -g 5000 -u 5000 mapr"
# clush -g cl,all "echo 'password1234' | passwd mapr --stdin"
```

■ 環境変数の設定

プロキシーサーバーと言語の環境変数を$HOME/.bash_profile に記述します。

```
# cat >> $HOME/.bash_profile << __EOF__
export http_proxy=http://proxy.your.site.com:8080
export https_proxy=http://proxy.your.site.com:8080
export LANG=en_US.UTF-8
__EOF__

# clush -a -c $HOME/.bash_profile --dest=$HOME/
```

■ GPG キーの登録

MapR 社が提供する、RPM パッケージのリポジトリに関する GPG キーを登録します。

```
# clush -g cl,all \
" . $HOME/.bash_profile; \
rpm --import http://package.mapr.com/releases/pub/maprgpg.key"
```

■ リポジトリの設定

RPM パッケージのリポジトリ設定ファイル「maprtech.repo」を作成します。以下は、MapR 6.0.1 および、MEP 5.0.0 に対応した、リポジトリの設定ファイル例です。

```
# cat > maprtech.repo << __EOF__
[maprtech]
```

```
name=MapR Technologies
baseurl=http://package.mapr.com/releases/v6.0.1/redhat/
enabled=1
gpgcheck=0
protect=1

[maprecosystem]
name=MapR Technologies
baseurl=http://package.mapr.com/releases/MEP/MEP-5.0.0/redhat
enabled=1
gpgcheck=0
protect=1
__EOF__
```

パッケージのリポジトリを登録します。

```
# clush -g cl,all -c ./maprtech.repo --dest /etc/yum.repos.d/
# clush -g cl,all -L "ls -l /etc/yum.repos.d/maprtech.repo"
n0130: -rw-r--r-- 1 root root 253 Apr 05 16:33 /etc/yum.repos.d/maprtech.repo
...
```

■ RPM パッケージのインストール

MapR クラスターでは、クラスターの規模によってノードにインストールする RPM パッケージ*3が異なりますので、clush コマンドにオプションを付与して、インストール対象ノードを指定します。CLDB、Zookeeper、ResourceManager、WebServer、HistoryServer をインストールします。

```
# clush -g cldb "yum install -y mapr-cldb"
# clush -g zk "yum install -y mapr-zookeeper"
# clush -g rm "yum install -y mapr-resourcemanager"
# clush -g web "yum install -y mapr-webserver"
# clush -g hs "yum install -y mapr-historyserver"
```

全クラスターノードにコアサービス、NodeManager サービス、ファイルサーバー、NFS サービス、Gateway サービスに関するパッケージをインストールします。

＊3　パッケージのリストは以下の URL に記載されています。
http://maprdocs.mapr.com/home/AdvancedInstallation/InstallingMapRSoftware-service-packages.html

```
# clush -a yum install -y \
mapr-core \
mapr-nodemanager \
mapr-fileserver \
mapr-nfs \
mapr-gateway
```

■ 設定ファイルのコピー

MapR の設定ファイルが保存されている conf.new ディレクトリ配下のファイルを conf ディレクトリコピーします。また、conf.d.new ディレクトリ配下のファイルを conf.d ディレクトリコピーします。まずは、MapR クラスター全体に関する /opt/mapr/conf ディレクトリに設定ファイルをコピーします。

```
# clush -a "cp -a \
/opt/mapr/conf.new/* \
/opt/mapr/conf/"
```

ZooKeeper に関する設定ファイルをコピーします。

```
# clush -g zk "cp -a \
/opt/mapr/zookeeper/zookeeper-3.4.5/conf.new/* \
/opt/mapr/zookeeper/zookeeper-3.4.5/conf/"
```

API サーバーに関する設定ファイルをコピーします。

```
# clush -g web "cp -a \
/opt/mapr/apiserver/conf.new/* \
/opt/mapr/apiserver/conf/"
```

ResourceManager および NodeManager サービスに関する設定ファイルをコピーします。

```
# clush -a "cp -a \
/opt/mapr/conf/conf.d.new/*  \
/opt/mapr/conf/conf.d/"
```

■ スクリプトによる MapR クラスターの設定

configure.sh スクリプトにより、MapR クラスターの設定を行います†。configure.sh スクリプトに指定するオプションで、ノードの役割を決定します。表 3-16 は、configure.sh スクリプトに指定する主なオプションとその値です。

> † CLDB、Zookeeper、ResourceManager、WebServer、HistoryServer の RPM パッケージをインストールしたホストと、オプションで指定したホスト名が一致しているかを注意深く確認し、慎重に実行してください。

表 3-16 configure.sh スクリプトのオプション

オプション	指定する値
-C	CLDB ノードのリスト
-Z	ZooKeeper ノードのリスト
-RM	ResourceManager ノードのリスト
-HS	HistoryServer ノードのリスト
-N	クラスター名
-M7	ライセンスタイプ
-no-autostart	configure.sh 実行時に Zookeeper または Warden を起動しない

今回、クラスター名は、「hpe-mapr-cluster01」にしました。

```
# clush -a \
"/opt/mapr/server/configure.sh \
-C n0131.jpn.linux.hpe.com,n0132.jpn.linux.hpe.com,n0133.jpn.linux.hpe.com \
-Z n0131.jpn.linux.hpe.com,n0132.jpn.linux.hpe.com,n0133.jpn.linux.hpe.com \
-RM n0131.jpn.linux.hpe.com,n0132.jpn.linux.hpe.com,n0133.jpn.linux.hpe.com \
-HS n0131.jpn.linux.hpe.com \
-N hpe-mapr-cluster01 \
-M7 \
-no-autostart"
```

■ サービスの起動

ZooKeeper サービスと Warden サービスを起動します。Warden サービスは、ZooKeeper サービスが稼働している全ノードで起動します。

```
# clush -g zk "systemctl daemon-reload"
# clush -g zk "systemctl restart mapr-zookeeper"
# clush -g zk -L "systemctl status mapr-zookeeper | grep Active"
n0131: Active: active (running) since Thu 2018-04-05 14:54:54 JST; 1min 30s ago
...

# clush -a "systemctl restart mapr-warden"
# clush -aL "systemctl status mapr-warden | grep Active"
n0131: Active: active (running) since Thu 2018-04-05 14:57:11 JST; 31s ago
...
```

各ノードにインストールされているサービスを確認します。これにより、各ノードで起動するサービスがわかります。

```
# clush -aL "ls -l /opt/mapr/roles"
n0131: total 0
n0131: -rw-r--r-- 1 root root 0 Apr  5 16:00 cldb
n0131: -rw-r--r-- 1 root root 0 Apr  5 16:00 fileserver
n0131: -rw-r--r-- 1 root root 0 Apr  5 16:00 gateway
n0131: -rw-r--r-- 1 root root 0 Apr  5 15:26 historyserver
n0131: -rw-r--r-- 1 root root 0 Apr  5 16:00 nfs
n0131: -rw-r--r-- 1 root root 0 Apr  5 15:26 nodemanager
n0131: -rw-r--r-- 1 root root 0 Apr  5 15:26 resourcemanager
n0131: -rw-r--r-- 1 root root 0 Apr  5 16:00 zookeeper
n0132: total 0
n0132: -rw-r--r-- 1 root root 0 Apr  5 16:00 cldb
n0132: -rw-r--r-- 1 root root 0 Apr  5 16:00 fileserver
n0132: -rw-r--r-- 1 root root 0 Apr  5 16:00 gateway
n0132: -rw-r--r-- 1 root root 0 Apr  5 16:00 nfs
n0132: -rw-r--r-- 1 root root 0 Apr  5 15:26 nodemanager
n0132: -rw-r--r-- 1 root root 0 Apr  5 15:26 resourcemanager
n0132: -rw-r--r-- 1 root root 0 Apr  5 16:00 zookeeper
n0133: total 4
n0133: -rw-r--r-- 1 root root 90 Mar 24 03:52 apiserver
n0133: -rw-r--r-- 1 root root  0 Apr  5 16:00 cldb
n0133: -rw-r--r-- 1 root root  0 Apr  5 16:00 fileserver
n0133: -rw-r--r-- 1 root root  0 Apr  5 16:00 gateway
n0133: -rw-r--r-- 1 root root  0 Apr  5 16:00 nfs
n0133: -rw-r--r-- 1 root root  0 Apr  5 15:26 nodemanager
n0133: -rw-r--r-- 1 root root  0 Apr  5 15:26 resourcemanager
```

```
n0133: -rw-r--r-- 1 root root  0 Apr  5 16:00 zookeeper
```

■ ローカルディスクの設定

各ノードに搭載されているローカルストレージのデバイスファイルと容量を確認します。

```
# clush -aL '(export LANG=en_US.utf8; parted -l | grep "Disk /dev/")'
...
n0131: Disk /dev/sda: 500GB
n0131: Disk /dev/sdb: 3221GB
n0131: Disk /dev/sdc: 3221GB
n0131: Disk /dev/sdd: 3221GB
...
```

データ用ディスクにおけるデバイスファイルのリストをファイルに記述します。データ用は、/dev/sdb、/dev/sdc、/dev/sdd とします。

```
# clush -a "cat > /root/disks.txt << EOL
/dev/sdb
/dev/sdc
/dev/sdd
EOL"
```

データ用ディスクにおけるデバイスファイルのリストが存在しているかを確認します。

```
# clush -aL "ls /root/disks.txt"
n0131: /root/disks.txt
...
```

データ用ディスクのデバイスファイルを記述した disks.txt を使って、MapR クラスターノードにディスクを登録します。ディスク容量によっては、登録にしばらく時間がかかる場合があります。

```
# clush -a "/opt/mapr/server/disksetup -F /root/disks.txt"
n0131: /dev/sdb added.
n0131: /dev/sdc added.
n0131: /dev/sdd added.
...
```

■ 環境変数の設定

env.sh ファイルに Java に関する環境変数を設定します。

```
# clush -g all \
sed -i \
"'/#set JAVA_HOME to override default search\
/a export JAVA_HOME=\/usr\/lib\/jvm\/jre/' \
/opt/mapr/conf/env.sh"
```

env.sh ファイルに記述した環境変数「JAVA_HOME」を確認します。

```
# clush -aL "grep '^export JAVA_HOME=' /opt/mapr/conf/env.sh"
n0131: export JAVA_HOME=/usr/lib/jvm/jre
...
```

Column　MapR クラスターの環境変数

MapR では、利用する Java のパス、クラスターのネットワーク設定や CLDB が使用するポート、ユーザーなどを環境変数として設定が可能です。環境変数は、全ノードの/opt/mapr/conf/env.sh スクリプトに記述します。MapR クラスターノードに複数の NIC が搭載されている場合に、クライアントマシンからアクセスするネットワークを明示的に指定する場合は、env.sh スクリプトに環境変数として設定します。env.sh スクリプトで設定する環境変数の例は、以下の URL に記載がありますので、一読されることをお勧めします。

https://maprdocs.mapr.com/60/ReferenceGuide/EnvironmentVariables.html

CLDB のマスターとなるサービス（CLDB マスター）が起動するノードを確認します†。

† Zookeeper と Warden サービス起動後、CLDB マスターのアサインが完了するまでに、しばらく時間がかかる場合があります。

```
# clush -aL "maprcli node cldbmaster"
n0131: cldbmaster
n0131: ServerID: 449629769426475027 HostName: n0131.jpn.linux.hpe.com
n0132: cldbmaster
n0132: ServerID: 449629769426475027 HostName: n0131.jpn.linux.hpe.com
n0133: cldbmaster
```

```
n0133: ServerID: 449629769426475027 HostName: n0131.jpn.linux.hpe.com
```

全クラスターノードの OS を再起動します。

```
# clush -a reboot
```

OS 再起動後、ZooKeeper、Warden サービスの起動、CLDB マスターのアサインが完了しているかを確認してください。

```
# clush -g zk -L "systemctl status mapr-zookeeper | grep Active"
# clush -aL "systemctl status mapr-warden | grep Active"
# clush -aL "maprcli node cldbmaster"
```

3-3-5 MapR Control System による設定

MapR クラスターの管理画面（MapR Control System、以下、MCS）は 8443 番ポートで提供されます。WebServer サービスが稼働するノーにおいて、8443 番ポートでリッスンしているかどうかを確認します。

```
# clush -g web "netstat -tunlp | grep 8443"
n0133: tcp6    0    0 :::8443        :::*        LISTEN      5818/java
```

■ MapR の評価版ライセンスの入手

クライアントノードから Web ブラウザで MCS にアクセスします。MCS の URL は、「https://Web サーバーの IP アドレス:8443」です。管理用 LAN に割り当てられているホスト名（n0133-mgm）か、IP アドレスでアクセス† します。

> † MCS が稼働するノードに対して、ホスト名でアクセスする場合は、Web ブラウザを起動するクライアントノードから DNS サーバーへ MCS のノード名の名前解決が必要です。DNS サーバーによる名前解決ができない場合は、クライアントノード側の/etc/hosts ファイルにおいて、MCS のホスト名と IP アドレスの対応の記述が必要です。

```
$ firefox https://n0133-mgm:8443 &
```

root アカウントで MCS にログインし (図 3-5)、MapR のエンドユーザーライセンスの許諾に同意します (図 3-6)。

図 3-5　MCS のログイン画面

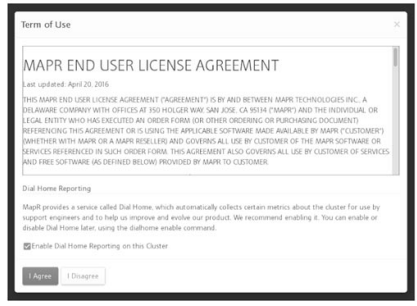


図 3-6　MapR のエンドユーザーライセンス許諾画面

MCS の GUI 画面が表示されます。この時点では、ライセンスを適用していないため、NFS Gateway サービスを示す「nfs」と CLDB サービスを示す「cldb」がデグレード状態になっている旨の警告が表示されます（図 3-7)。サービスがデグレード状態になると GUI 画面左の「Node Health」内のサービスの四角が黄色になります。

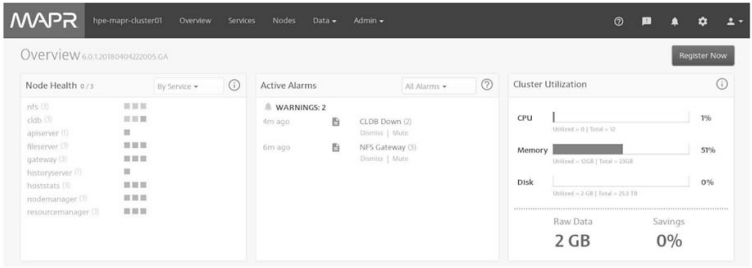


図 3-7　ライセンス適用前の MCS の GUI 画面

MCS の管理画面上部の［Admin］から［Cluster Settings］をクリックします (図 3-8)。

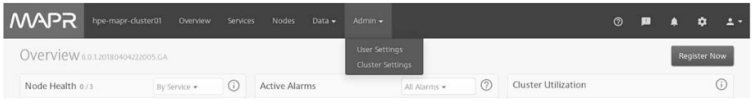


図 3-8　MCS の管理画面上部の「Admin」から「Cluster Settings」をクリック

「Admin / Cluster Settings」の画面内の［Licenses］タブをクリックします。すると、ライセンスの設定画面が表示されます。現在のライセンスの状態を確認することができます (図 3-9)。

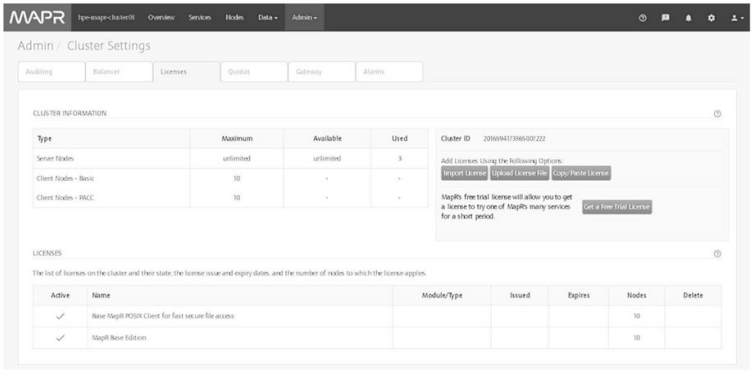


図 3-9　ライセンスの設定画面

Web ブラウザの別のタブを開き、`https://mapr.com/user` にアクセスします。MapR 社が提供する「My Clusters」と書かれた Web サイトで、MapR クラスターのライセンスを管理します。MapR のユーザーアカウントを使って My Clusters の Web サイトにログイン† します (図 3-10)。

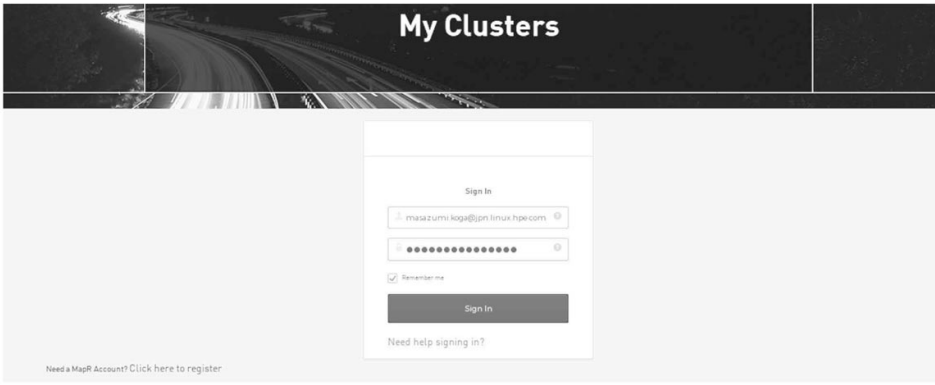


図 3-10　「My Clusters」のログイン画面

† MapR のユーザーアカウントを作成していない場合は、同ページ内の左下にある［Need a MapR Account? Click here to register］をクリックし、事前にユーザーアカウントを作成しておいてください。

MapR のユーザーアカウントで「My Clusters」の Web サイトにログインしたら、画面左下の［ADD NEW CLUSTER］をクリックします (図 3-11)。

図 3-11　［ADD NEW CLUSTER］をクリック

クラスターの追加に関する設定画面が表示されますので、「Cluster ID」の欄に、MCS のライセンス設定画面の右側に表示されている、「Cluster ID」の右側に続く、先に示した ID 番号を入力します。「Cluster Name」には、作成したクラスター名（今回の場合は、hpe-mapr-cluster01）を入力します。今回は、Converged Enterprise Edition の評価ライセンスを適用しますので、「Converged Enterprise Edition (formerly M5, M7) Trial」を選択し、［ADD NEW CLUSTER］をクリックします (図 3-12)。

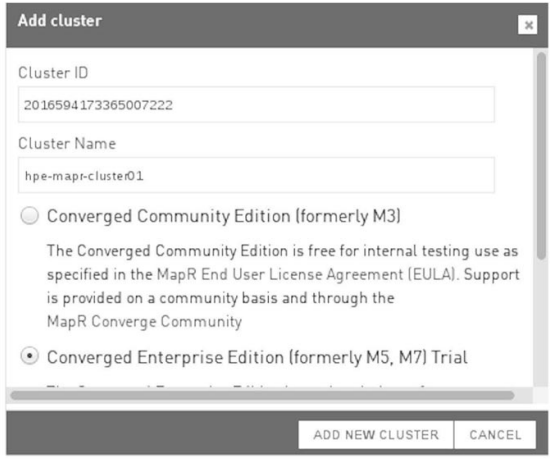


図 3-12　Cluster ID と Cluster Name の入力、エディションの選択

クラスター「hpe-mapr-cluster01」の登録を「My Clusters」の画面で確認します (図 3-13)。

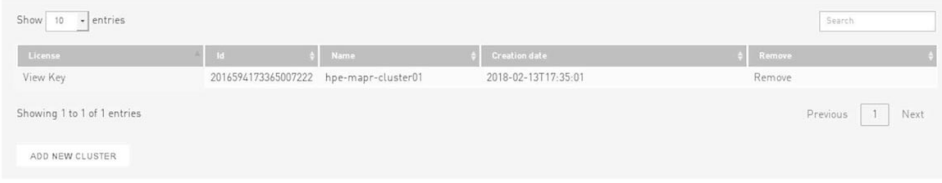


図 3-13　登録されたクラスター「hpe-mapr-cluster01」

MCS のライセンス設定画面に戻り、「Cluster ID」の下にある［Import License］をクリックします (図 3-14)。

図 3-14　MCS のライセンス設定画面上の「Cluster ID」の下にある［Import License］をクリック

「Import License」のウィンドウが表示されるので、MapR のユーザーアカウントでログインし(図 3-15)、［Submit］をクリックします（図 3-16）。

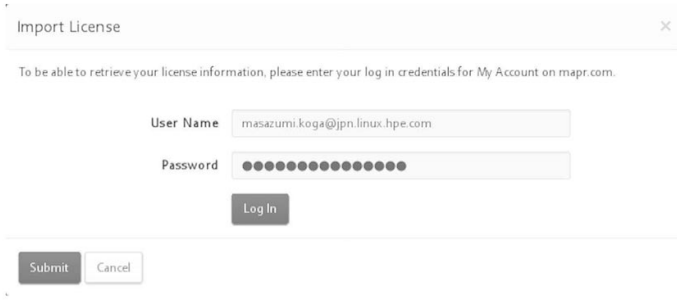


図 3-15　「Import License」のウィンドウ

図 3-16　［Submit］をクリック

† ［Submit］ボタンをクリックしても画面が遷移しない場合は、MCS への接続がタイムアウトしている可能性があります。その場合は、再度、MCS にログインし［Submit］ボタンを押してください。

MCS のライセンス設定画面に戻り、画面下の「LICENSES」の「Name」列に「MapR Enterprise Trial Edition」[*4]が表示され、「Active」列にチェックが入っていることを確認します (図 3-17)。

図 3-17　「MapR Enterprise Trial Edition」のライセンスがアクティブになっていることを確認

全クラスターノードを再起動します。

＊4　「MapR Enterprise Trial Edition」は、30 日間の評価用ライセンスです。

```
# hostname
n0130.jpn.linux.hpe.com

# clush -a "reboot"
```

再び、MCS の Web 管理画面にアクセスします。

```
$ firefox https://n0133-mgm:8443 &
```

MCS の Web 管理画面の上側にある［Nodes］をクリックし、ブラウザをスクロールダウンさせると、MapR クラスターを構成しているノード一覧が表示されます。全ノードの「Health」列に「■」が表示されているかを確認してください (図 3-18)。

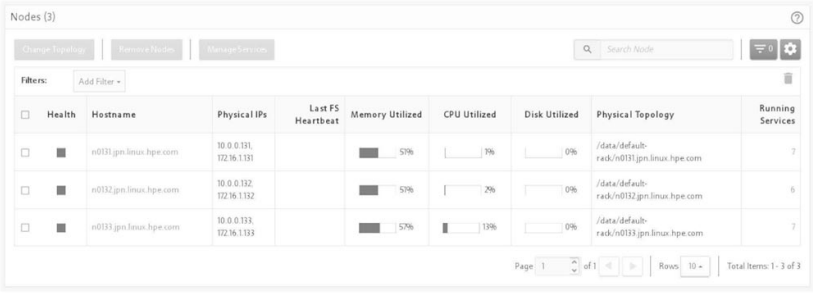


図 3-18　MapR クラスターノードの状態を確認

MapR Enterprise Trial Edition のライセンスが適用されると、デグレード状態だった CLDB サービスと NFS サービスが正常な状態になります。MCS の Web 管理画面上の［Overview］をクリックし、CLDB サービスと NFS サービスの状態を確認してください (図 3-19)。

図 3-19　ライセンス適用後の MCS の GUI 管理画面

3-3-6　MapR クラスターの動作確認

データを生成する MapReduce プログラムの「TeraGen」を実行し、MapR クラスターでジョブが実行できるかを確認します。さらに、TeraGen で生成したデータを TeraSort によりサンプリングし、データを並べ替えます。

■ TeraGen と TeraSort の動作確認

まず、ユーザー mapr としてクラスターノードにログインし、ベンチマークデータを格納するボリュームを作成します。ユーザー mapr のコマンドプロンプトを「$」で表します。

```
# ssh n0131-mgm
# su - mapr
$ whoami
mapr
```

MapR-FS 上に/test01 ディレクトリを作成します。

```
$ maprcli volume create -name test01 -mount 1 -path /test01
$ hadoop fs -ls /
Found 6 items
drwxr-xr-x - mapr mapr                0 2018-04-07 22:35 /apps
drwxr-xr-x - mapr mapr                0 2018-04-07 22:37 /opt
drwxr-xr-x - mapr mapr                0 2018-04-07 21:12 /test01
drwxrwxrwx - mapr mapr                0 2018-04-07 22:35 /tmp
drwxr-xr-x - mapr mapr                0 2018-04-07 22:37 /user
drwxr-xr-x - mapr mapr                1 2018-04-07 22:35 /var
```

TeraGen によりデータを生成します。TeraGen によるデータ生成は、yarn コマンドで行います。

```
$ yarn jar \
/opt/mapr/hadoop/hadoop-2.7.0/share/hadoop/mapreduce/\
hadoop-mapreduce-examples-2.7.0-mapr-1803.jar \
teragen 500000 /test01/teragen01
...
18/04/05 15:23:56 INFO mapreduce.Job: Job job_1506147309785_0001 completed succ
essfully
...
```

ジョブの処理に関するメッセージが表示されます。メッセージの途中に「completed successfully」が表示されているかを確認してください†。

† 「completed successfully」が出力されない場合は、TeraGen の実行が失敗していますので、クラスターが適切に構成されていません。この場合は、OS の設定（ホスト名、ファイアウォールなど）、サービス、ポート番号などをチェックする必要があります。

データが MapR-FS 上の/test01 ディレクトリ以下に生成されているかを確認します。

```
$ hadoop fs -ls /test01/teragen01
Found 3 items
-rwxr-xr-x 3 mapr mapr        0 2018-04-05 19:06 /test01/teragen01/_SUCCESS
-rwxr-xr-x 3 mapr mapr 25000000 2018-04-05 19:06 /test01/teragen01/part-m-00000
-rwxr-xr-x 3 mapr mapr 25000000 2018-04-05 19:06 /test01/teragen01/part-m-00001
```

Column　Connection refused の呪縛

Hadoop のジョブの実行に失敗する原因には、さまざまなものが存在します。ジョブが失敗する場合のエラーメッセージとして有名なものが、「failed on connection exception: java.net.ConnectException: Connection refused」です。「Connection refused」という英語表記からセキュリティのエラーを想像するかもしれませんが、実は、設定ファイルを記述したディレクトリが見つからない場合にもこのようなメッセージが表示されます。MapR の場合、Hadoop 関連の RPM パッケージをインストールすると、/opt/mapr ディレクトリ以下に存在する conf.new ディレクトリや conf.d.new ディレクトリなどに多くの設定ファイルが配置されます。しかし、MapR では、クラスターの稼働に必要な設定ファイル一式を適切なディレクトリ（conf ディレクトリや conf.d ディレクトリなど）に事前にコピーしなければなりません。パッケージのアンインストールを行って、再インストールを行う場合や MapR 自体のアップグレードを行う場合、ファイルのコピー作業を忘れると「Connection refused」のエラーに遭遇するおそれがあります。

Hadoop クラスターにおける「Connection refused」のエラーの発生には、さまざまな原因が考えられますが、まずは、MapR が生成するログを確認し、エラーの発生原因を調べます。

MapR におけるログの保管先ディレクトリ：
/opt/mapr/hadoop/hadoop-2.7.0/logs

以下に、ジョブが実行できない場合の主な確認事項を挙げておきます。

- /etc/hosts ファイルの記述（localhost の行、各ノードの行、FQDN、ホスト名の別名）
- configure.sh で指定したノード役割と RPM パッケージをインストールしたノードの整合性
- SELinux
- ポート番号（iptables や firewalld）
- 時刻同期
- サービス稼働状態（ResourceManager、NodeManager など）

- 設定ファイルが保管されているディレクトリパス
- 各ノードの設定ファイルの読み取り、書き込み権限、所有者
- 分散ファイルシステムのディレクトリの読み取り、書き込み権限、所有者
- XML ファイルの記述、ファイルフォーマット、改行の有無

MapR-FS 上の/test01 ディレクトリに生成されたデータを使って、ソート処理のジョブを実行します。

```
$ yarn jar \
/opt/mapr/hadoop/hadoop-2.7.0/share/hadoop/mapreduce/\
hadoop-mapreduce-examples-2.7.0-mapr-1803.jar \
terasort /test01/teragen01 /test01/terasort01
...
18/04/05 15:24:51 INFO mapreduce.Job: Job job_1506147309785_0002 completed succ
essfully
...
```

JobHistory サーバー（n0131）にアクセスします (図 3-20)。

```
$ firefox http://n0131-mgm:19888 &
```

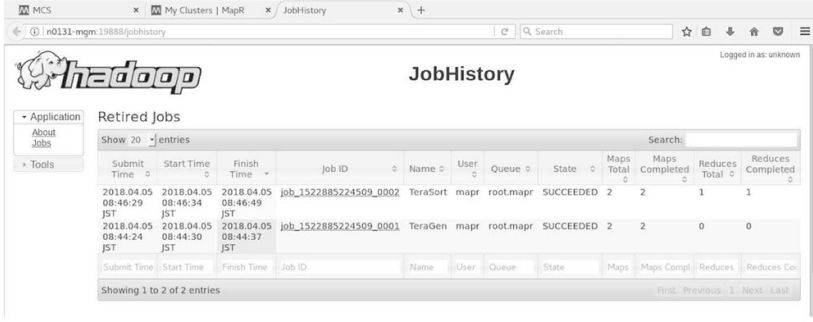


図 3-20　JobHistory サーバーが提供するジョブの履歴管理画面

JobHistory サーバーが提供するジョブの履歴管理画面において、実行したジョブが一覧表示されます。ジョブの詳細を表示するには、「Job ID」列に表示されている「job_」で始まるリンクをクリックします (図 3-21)。

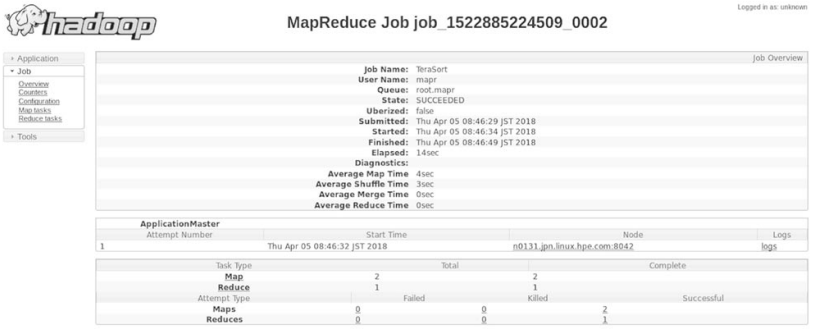


図 3-21　ジョブの詳細を表示

以上で、MapR クラスターで TeraGen と TeraSort による MapReduce ジョブが実行できました。

3-3-7　MapR クライアントのインストール

MapR クラスター外のクライアントマシンから、hadoop コマンドなどを使って MapR クラスターを管理したい場合があります。そのような場合は、MapR クライアントをクラスター外のマシンにインストールします。以下では、CentOS 7.4 が稼働するクライアントマシンに、MapR クライアントをインストールする手順を紹介します[*5]。

■ ユーザーの追加

クライアントノードに mapr ユーザーと mapr グループが存在しない場合は、作成しておきます。

```
# hostname
n0130.jpn.linux.hpe.com

# groupadd -g 5000 mapr
# useradd -g 5000 -u 5000 mapr
# echo 'password1234' | passwd mapr --stdin
# echo 'export LANG=en_US.UTF-8' >> /home/mapr/.bash_profile
# chown mapr:mapr /home/mapr/.bash_profile
# ls -l /home/mapr/.bash_profile
-rw-r--r-- 1 mapr mapr 216 Apr 5 11:49 /home/mapr/.bash_profile
```

＊5　MapR クライアントのインストールについては、以下の URL が参考になります。
https://maprdocs.mapr.com/60/AdvancedInstallation/SettingUptheClient-redhat.html

クライアントマシン上で、MapR 社が提供するパッケージのリポジトリを追加します。MapR クラスターノードですでに設定されているリポジトリの設定ファイルをコピーします。

```
# scp n0131-mgm:/etc/yum.repos.d/maprtech.repo /etc/yum.repos.d/
# cat /etc/yum.repos.d/maprtech.repo
[maprtech]
name=MapR Technologies
baseurl=http://package.mapr.com/releases/v6.0.1/redhat/
enabled=1
gpgcheck=0
protect=1

[maprecosystem]
name=MapR Technologies
baseurl=http://package.mapr.com/releases/MEP/MEP-5.0.0/redhat
enabled=1
gpgcheck=0
protect=1
```

クライアントマシンに、MapR 関連のパッケージがないことを確認します。

```
# rpm -qa | grep mapr
```

サードパーティ製のパッケージを提供する EPEL リポジトリの設定ファイル、Java、MapR クライアント用のパッケージをインストールします。

```
# yum install -y epel-release
# yum install -y java-1.8.0-openjdk mapr-client.x86_64
```

クライアントマシン上の/opt/mapr ディレクトリ配下の構成を確認します。

```
# ls -F /opt/mapr/
bin/       contrib/   include/  LICENSE.doc       NOTICE.txt
conf/      examples/  lib/      logs/             pid/
conf.new/  hadoop/    libexp/   MapRBuildVersion  server/
```

クライアントマシン上の/opt/mapr/conf.new ディレクトリ以下のすべてのファイルを/opt/mapr/conf ディレクトリにコピーします。

```
# cp -a /opt/mapr/conf.new/* /opt/mapr/conf/
```

クライアントを MapR クラスターに登録します。configure.sh スクリプトに「-c」オプション

を付与することにより、クライアントの登録が可能です。それ以外のオプションは、MapR クラスターを作成した際のオプションと同じです。

```
# /opt/mapr/server/configure.sh \
-c \
-C n0131.jpn.linux.hpe.com,n0132.jpn.linux.hpe.com,n0133.jpn.linux.hpe.com \
-Z n0131.jpn.linux.hpe.com,n0132.jpn.linux.hpe.com,n0133.jpn.linux.hpe.com \
-RM n0131.jpn.linux.hpe.com,n0132.jpn.linux.hpe.com,n0133.jpn.linux.hpe.com \
-HS n0131.jpn.linux.hpe.com \
-N hpe-mapr-cluster01 \
-M7 \
-no-autostart
```

以上で MapR クライアントのインストールができました。

■ MapR クライアントから MapR クラスターへの接続確認

MapR クライアントに含まれる hadoop コマンドで MapR クラスターの MapR-FS にアクセスできるかを確認します。

```
# hostname
n0130.jpn.linux.hpe.com

# su - mapr
$ which hadoop
/bin/hadoop

$ ls -l /bin/hadoop
lrwxrwxrwx 1 root root 20 Apr 5 11:58 /bin/hadoop -> /opt/mapr/bin/hadoop

$ hadoop fs -ls /
...
SLF4J: Class path contains multiple SLF4J bindings.
...
Found 6 items
drwxr-xr-x - mapr mapr                0 2018-04-04 09:59 /apps
drwxr-xr-x - mapr mapr                0 2018-04-05 10:01 /opt
drwxr-xr-x - mapr mapr                2 2018-04-04 11:36 /test01
drwxrwxrwx - mapr mapr                0 2018-04-04 09:54 /tmp
drwxr-xr-x - mapr mapr                0 2018-04-04 10:00 /user
drwxr-xr-x - mapr mapr                1 2018-04-04 09:59 /var
```

クライアントマシンに存在する/etc/hosts ファイルを MapR-FS 上の/test01 ディレクトリに

コピーし、ファイルの中身を確認します。

```
$ hadoop fs -put /etc/hosts /test01/
$ hadoop fs -cat /test01/hosts
...
127.0.0.1   localhost localhost.localdomain localhost4 localhost4.localdomain4
::1         localhost localhost.localdomain localhost6 localhost6.localdomain6
...
```

さらに、MapR クライアントマシンから、TeraGen、TeraSort のジョブの投入が可能かどうかを確認します。今回は、MapR-FS 上の/test01/teragen02 ディレクトリ以下にデータを生成し、ソート結果は、MapR-FS 上の/test01/terasort02 ディレクトリ上に出力します。

```
$ yarn jar \
/opt/mapr/hadoop/hadoop-2.7.0/share/hadoop/mapreduce/\
hadoop-mapreduce-examples-2.7.0-mapr-1803.jar \
teragen 500000 /test01/teragen02

$ yarn jar \
/opt/mapr/hadoop/hadoop-2.7.0/share/hadoop/mapreduce/\
hadoop-mapreduce-examples-2.7.0-mapr-1803.jar \
terasort /test01/teragen02 /test01/terasort02
```

ジョブが正常に実行できたら、MapR クライアントの設定は完了です。

3-4　まとめ

本章では、Apache Hadoop 3 クラスターと MapR 6.0 クラスターの構築手順を紹介しました。コミュニティ版の Hadoop と商用版 Hadoop では、インストール手順が大きく異なります。まずは、数ノードからなる小規模なクラスターの構築からスタートし、分散ファイルシステムへのファイルのコピーやジョブの実行を体験してみてください。

Column　MapR インストーラー「mapr-setup.sh」のススメ

本章では、Apache Hadoop と MapR 版 Hadoop のインストール手順を紹介しましたが、MapR 版 Hadoop には、クラスターのセットアップを行うスクリプト「`mapr-setup.sh`」が提供されています。この `mapr-setup.sh` は、「MapR インストーラ」と呼ばれます。

`mapr-setup.sh` スクリプトを使えば、Web ブラウザを使って MapR クラスターの構築が可能です。MapR 社が提供している管理者向けの教育トレーニングのカリキュラムにおいても、`mapr-setup.sh` スクリプトを使った実習が含まれています。MapR クラスターの管理に不慣れな入門者は、オペレーションミスやクラスターの構成ミスを回避するためにも、`mapr-setup.sh` スクリプトを使用するとよいでしょう。

【参考情報】

mapr-setup.sh スクリプトの入手先：

`http://package.mapr.com/releases/installer/`

mapr-setup.sh スクリプトの使用法：

`https://maprdocs.mapr.com/home/MapRInstaller.html`

第４章
Hadoop クラスターの運用管理

本章では、ビッグデータ分析基盤の管理者が知っておくべき、Hadoop 基盤の基本的な管理手法を紹介します。また、Apache Hadoop 3 から採用されている新機能に関する運用管理手法についても具体的な手順を交えて解説します。さらに、後半では、多くの企業システムでも利用されている MapR クラスターの NFS サーバーの運用管理手法、スナップショットによるデータの管理手法、GUI 管理について、構築手順を交えて簡単に紹介します。

4-1 Apache Hadoop 3 クラスターの管理

Hadoop システムでは、ユーザーのジョブの投入を問題なく行うために、Hadoop 基盤の管理者は、クラスターで稼働するサービスの状態を日々確認しておくことが必要です。日々のメンテナンスを怠ると、性能面だけでなく、システム全体の障害につながり、ビッグデータ分析基盤を活用した分析業務に大きな影響が出ます。Hadoop 基盤の管理者は、クラスターで稼働するサービスの死活監視、負荷状況のチェック、データの使用状況など、管理項目は少なくありませんが、まずは、Hadoop クラスターを構成する最低限のサービスが稼働しているかどうかをチェックすることから始めます。

4-1-1 サービスの状態確認

Apache Hadoop 3 クラスターでは、マスターノードとデータノードで Java のプロセスが稼働します。この Java のプロセスの起動、停止によってクラスターの稼働状態が変わります。主に、サービスの稼働状態は、jps コマンド† で確認します。

† jps コマンドは、java-1.8.0-openjdk-devel パッケージに含まれています。

```
# hostname
n0120.jpn.linux.hpe.com

# clush -g nn "jps | grep -vi jps"
n0121: 4152 NameNode
n0121: 4554 JobHistoryServer
n0121: 4286 ResourceManager

# clush -g dn -L "jps | grep -vi jps"
n0122: 1269 DataNode
n0122: 1400 NodeManager
...
```

上記のように、jps コマンドを使うと、稼働しているサービスのプロセス番号も表示されます。jps コマンドではなく、pgrep コマンドでもサービスのプロセス番号を表示できます。以下は、NameNode サービスのプロセス番号を取得する例です。

```
# clush -g nn "pgrep -u root -f namenode"
n0121: 4152
```

4-1-2　クラスターの停止

第 2 章のシステム構築でも説明したように、Apache Hadoop 3 クラスターでは、クラスターの起動を hdfs コマンド、yarn コマンド、mapred コマンドで行いますが、停止についても同様に、hdfs コマンド、yarn コマンド、mapred コマンドを使います。

■ Apache Hadoop 3 クラスターの停止手順：

```
# clush -g nn ". $HOME/.bash_profile; hdfs --daemon stop namenode"
# clush -g dn ". $HOME/.bash_profile; hdfs --daemon stop datanode"
# clush -g dn ". $HOME/.bash_profile; yarn --daemon stop nodemanager"
# clush -g nn ". $HOME/.bash_profile; yarn --daemon stop resourcemanager"
# clush -g nn ". $HOME/.bash_profile; mapred --daemon stop historyserver"
```

Note　各ノードの停止順序

Hadoop では、NameNode からメタデータを取得して保持しておくセカンダリー NameNode を設ける場合がありますが、セカンダリー NameNode が存在する場合は、DataNode の停止の後にセカンダリー NameNode を停止します。また、ZooKeeper を使った HA クラスター構成の場合は、クォーラムジャーナルノードと呼ばれる HA クラスターの調停役を担うノードも存在します。この場合は、セカンダリー NameNode を停止した後にクォーラムジャーナルノードのサービス（ジャーナルノードサービスといいます）を停止し、最後に ZooKeeper サービスを停止します。

サービスの停止後、jps コマンドでプロセスが存在しないことを確認します。

4-1-3　HDFS を提供しているノード情報の表示

Hadoop クラスターでは、1 台のワーカーノードに障害が発生しても、分散ファイルシステムのレプリカの機能により、データのブロックの複製が別のワーカーノードに存在するため、ユーザー

は、データにアクセスできますが、当然、ワーカーノードの数が減るため、分散処理のターンアラウンドタイムは長くなります。また、障害が発生すると、その障害ノードに存在したデータについては、レプリカの数が減少するため、レプリカの数を維持するためにレプリカの再作成の処理が発生します。レプリカの再作成の処理は負荷がかかるため、分散処理性能に影響します。1 台のワーカーノード障害がシステムの全停止につながるわけではありませんが、ユーザーが期待する分散処理性能の確保、迅速な復旧、システム全体の安定稼働の維持という観点から、Hadoop の管理者は、ワーカーノードの死活状態を把握しておく必要があります。

クラスターに参加し、正常に HDFS を提供できるノード（ライブノード）と、HDFS を提供できていないノード（デッドノード）の数を把握するには、`hdfs` コマンドを使います。正常な DataNode を表示する場合は、「`hdfs dfsadmin`」コマンドに、「`-report -live`」を付与して実行します。

```
# hdfs dfsadmin -report -live
Configured Capacity: 9658951544832 (8.78 TB)
Present Capacity: 9658850021376 (8.78 TB)
DFS Remaining: 9658784931840 (8.78 TB)
DFS Used: 65089536 (62.07 MB)
DFS Used%: 0.00%
...
...
  -------------------------------------------------
Live datanodes (3):

Name: 10.0.0.122:9866 (n0122.jpn.linux.hpe.com)
Hostname: n0122.jpn.linux.hpe.com
Decommission Status : Normal
Configured Capacity: 3219650514944 (2.93 TB)
DFS Used: 21696512 (20.69 MB)
Non DFS Used: 33841152 (32.27 MB)
DFS Remaining: 3219594977280 (2.93 TB)
DFS Used%: 0.00%
DFS Remaining%: 100.00%
...
```

HDFS 全体の容量などが表示され、その下に「`Live datanodes (3):`」と表示されています。この場合は、ライブノードが 3 台であることを意味しています。ライブノードが存在すれば、その下に全ライブノードの状態が表示されます。一方、デッドノードを表示する場合は、「`hdfs dfsadmin`」コマンドに、「`-report -dead`」を付与して実行します。

```
# hdfs dfsadmin -report -dead
Configured Capacity: 9658951544832 (8.78 TB)
```

```
Present Capacity: 9658850021376 (8.78 TB)
DFS Remaining: 9658784931840 (8.78 TB)
DFS Used: 65089536 (62.07 MB)
DFS Used%: 0.00%
...
...
  -------------------------------------------------
Dead datanodes (0):
```

コマンドを実行すると、HDFS 全体の容量などが表示され、その下にデッドノード数が表示されます。「`Dead datanodes (0):`」と表示された場合は、デッドノード数が 0 台であることを意味しています。

デッドノードがある場合は、「`Dead datanodes`」にその台数が表示されます。デッドノードは、Hadoop クラスターにワーカーノードとして登録されているにもかかわらず、マスターノードからの応答に反応しないノードです。NIC に障害が発生した場合や、サーバー自体が停止した場合、さらには、正常にシャットダウンされ、電源が OFF になっているノードもデッドノードとしてカウントされます。

正常にシャットダウンしたワーカーノードの場合は、再び電源を投入し、OS が正常に起動しても、Hadoop クラスターの各サービスが正常に稼働しなければ、デッドノードとしてカウントされたままなので、Hadoop の DataNode サービスを `hdfs` コマンドを使って、手動で起動させる必要があります。たとえば、ワーカーノードの n0124 をシャットダウンすると、n0124 は、デッドノードとしてカウントされますが、電源投入後、再び正常ノードに復帰させるには、以下のように、DataNode サービスを起動させる必要があります。

```
# clush -w n0124 ". $HOME/.bash_profile; hdfs --daemon start datanode"
```

4-1-4 Hadoop バージョンの表示

Hadoop では、1 つのクラスターのマスターノードとワーカーノードの Hadoop のバージョンを統一しますが、部門ごとに Hadoop クラスターを購入している場合は、部門で利用するアプリケーションの都合により、Hadoop のバージョンが異なることがあります。Hadoop のバージョンによって、アプリケーションの挙動や指定するパラメーターが異なる場合があるため、Hadoop クラスターが複数存在する場合は、Hadoop のバージョンを事前に把握しておくことが必要です。また、Hadoop 自体をアップグレードする場合、旧バージョンの Hadoop と新バージョンの Hadoop を 2

つ保存しておき、新バージョンの Hadoop でアプリケーションが正常に稼働するかどうかを見極めたうえで、新バージョンにアップグレードするのが普通です。そのような場合、新バージョンの Hadoop 関連のコマンドが環境変数によって正しく設定されているかどうかを確認しなければなりません。

「hadoop version」コマンドを実行すれば、Hadoop のバージョン番号が表示され、Hadoop の環境変数が正しくセットされているかどうかがすぐにわかります。

```
# clush -g cl,all -L ". .bash_profile; hadoop version" | grep Hadoop
n0120: Hadoop 3.1.0
n0121: Hadoop 3.1.0
n0122: Hadoop 3.1.0
n0123: Hadoop 3.1.0
n0124: Hadoop 3.1.0
```

4-1-5　DataNode サービスの稼働時間の確認

Hadoop クラスターでは、ワーカーノードが停止しても、HDFS のレプリカの機能により、ユーザーはデータにアクセスできるため、DataNode の停止が即システム全体の停止に至ることはありません。しかし、システム運用の観点では、大量のワーカーノードが存在する場合、障害が発生したワーカーノードを素早く見つけ、障害ノードの保守を行う必要があります。たとえば、ワーカーノードの OS 起動時に Hadoop 関連のサービス（DataNode サービスや NodeManager サービスなど）を自動的に起動するといった運用の場合を考えます。このようなシステムでは、障害によってワーカーノードの電源リセットや、OS の再起動、強制リセットが発生すると、ワーカーノードが Hadoop クラスターに再び自動的に参加します。ユーザーや IT 運用部門は、HDFS のレプリカの機能により、ユーザーデータに引き続きアクセスできるため、ノード障害の発生に気付かないといった「障害の見過ごし」が起こりえます。

このような「障害の見過ごし」を防止するには、ワーカーノードの連続稼働時間を監視します。連続稼働時間が極端に短いワーカーノードは、なんらかの原因により過去にサービスが停止したことを暗示します。このように、現時点で正常に稼働しているように見えても、過去に障害が発生し、サービスの停止があったかどうかを知るために、IT 部門は、ワーカーノードの連続稼働時間を把握しておくことが重要です。

```
# clush -g dn -L \
```

```
'. $HOME/.bash_profile;hdfs dfsadmin -getDatanodeInfo 'hostname':9867'
n0122: Uptime: 21, Software version: 3.1.0, Config version: core-3.0.0,hdfs-1
n0123: Uptime: 89837, Software version: 3.1.0, Config version: core-3.0.0,hdfs-1
n0124: Uptime: 89837, Software version: 3.1.0, Config version: core-3.0.0,hdfs-1
```

上記の例では、ワーカーノード n0122 の DataNode サービスの連続稼働時間が 21 秒、n0123 と n0124 の DataNode サービスの連続稼働時間が 89837 秒となっています。n0123 と n0124 の DataNode サービスは、長時間稼働し続けているのに対して、n0122 の DataNode サービスは、起動したばかりであることが読み取れます。

4-1-6　データの書き込み・削除の禁止

Hadoop クラスターをメンテナンスする場合、HDFS へのデータの書き込みや既存のデータの削除などを禁止したい場合があります。このような場合、HDFS をセーフモードにすることで、データの書き込みや削除を禁止することができます。セーフモードを ON にするには、「`hdfs dfsadmin`」コマンドに`-safemode` オプションを付与し、`enter` を指定します。

```
# whoami
root

# hdfs dfsadmin -safemode enter
Safe mode is ON
```

HDFS がセーフモードになると、HDFS へのファイルの参照や読み出しは可能ですが、書き込みや削除などのオペレーションを禁止できます。以下は、クライアントノード上にテスト用のファイル「`testfile`」を作成し、セーフモードになった HDFS に書き込みができないことを確認する例です。

```
# su - koga
$ echo "Hello Hadoop 3." > /home/koga/testfile

$ hdfs dfs -ls /user
Found 1 items
drwx------   - koga supergroup          0 2018-04-14 15:16 /user/koga

$ hdfs dfs -put /home/koga/testfile /user/koga/
put: Cannot create file/user/koga/testfile._COPYING_. Name node is in safe mode.
```

セーフモードでは、ファイルの書き込みが禁止されていることがわかります。同様に、ファイ

ルの削除も禁止されているかを確認します。

```
$ hdfs dfs -rm  /user/koga/dir01/file01
rm: Cannot delete /user/koga/dir01/file01. Name node is in safe mode.
```

■ セーフモードの解除

セーフモードを解除するには、「hdfs dfsadmin」コマンドの「-safemode」オプションに leave を指定します。

```
# whoami
root

# hdfs dfsadmin -safemode leave
Safe mode is OFF

# su - koga
$ echo "Hello Hadoop 3." > /home/koga/testfile2
$ hdfs dfs -put /home/koga/testfile2 /user/koga/
$ hdfs dfs -cat /user/koga/testfile2
Hello Hadoop 3.
```

4-1-7　クォータの設定

HDFS には、ディレクトリにクォータを設定することで、ユーザー対して、HDFS の使用量を制限できます。Hadoop では、**表 4-1** の 2 種類のクォータを設定できます。

表 4-1　クォータ

HDFS のクォータの種類	説明
ネームクォータ	ファイルとディレクトリの数を制限
スペースクォータ	ディレクトリに対して容量を制限

■ ネームクォータによるファイル数の制限

ネームクォータは、指定したディレクトリに対して、ファイルとディレクトリの数を制限します。ネームクォータを設定するには、「hdfs dfsadmin」コマンドに「-setQuota」オプションを

付与し、許容したい最大数とディレクトリ名を指定します。以下は、HDFS の/user/koga ディレクトリに対して、作成できるファイルとディレクトリの合計の数を 1000 個に制限する例です。

```
# whoami
root

# hdfs dfsadmin -setQuota 1000 /user/koga
```

HDFS の/user/koga ディレクトリにファイルやディレクトリを作成してみます。例として、クライアントノードの/usr/share/doc ディレクトリ以下にあるドキュメント類を HDFS の/user/koga ディレクトリにコピーします。

```
# su - koga
$ hdfs dfs -put /usr/share/doc/* /user/koga/
put: The NameSpace quota (directories and files) of directory /user/koga is exc
eeded: quota=1000 file count=1001
...
```

HDFS 上の/user/koga ディレクトリにファイルとディレクトリの数の合計数が 1000 を超えると、メッセージが表示され、書き込みが制限されます。

■ ネームクォータの設定の解除

ネームクォータの設定を解除するには、「hdfs dfsadmin」コマンドに「-clrQuota」オプションを付与し、ディレクトリ名を指定します。

```
# whoami
root

# hdfs dfsadmin -clrQuota /user/koga
```

■ スペースクォータによる容量制限

スペースクォータは、指定したディレクトリに対して使用量を制限します。使用量は、データのレプリカを掛けた値で計算します。たとえば、レプリケーション・ファクタを 3 で設定した環境において、300GB のスペースクォータを HDFS のディレクトリに設定した場合、ユーザーは、100GB のデータを HDFS ディレクトリに保存するとスペースクォータを使い果たします。

スペースクォータを設定するには、「hdfs dfsadmin」コマンドに「-setSpaceQuota」オプションを

付与し、許容したい最大容量とディレクトリ名を指定します。以下は、HDFS の/user/koga/testdir01 ディレクトリに対して、作成できる容量を 1GB に制限する例です。

```
# whoami
root

# hdfs dfs -mkdir /user/koga/testdir01
# hdfs dfs -chown koga:supergroup /user/koga/testdir01
# hdfs dfsadmin -setSpaceQuota 1g /user/koga/testdir01
```

クライアントノードにおいて、テスト用の 200MB のファイルを 2 つ作成します。

```
# su - koga
$ dd if=/dev/zero of=/home/koga/file200MB-01 bs=1M count=200
$ dd if=/dev/zero of=/home/koga/file200MB-02 bs=1M count=200
```

最初の 200MB のファイルを HDFS の/user/koga/testdir01 ディレクトリにコピーします。

```
$ hdfs dfs -put /home/koga/file200MB-01 /user/koga/testdir01
```

もう 1 つの 200MB のファイルも HDFS 上の/user/koga/testdir01 ディレクトリにコピーします。

```
$ hdfs dfs -put /home/koga/file200MB-02 /user/koga/testdir01
put: The DiskSpace quota of /user/koga/testdir01 is exceeded: quota = 107374182
4 B = 1 GB but diskspace consumed = 1434451968 B = 1.34 GB
```

HDFS 上の/user/koga/testdir01 ディレクトリに設定したスペースクォータの容量を超えると、メッセージが表示され、書き込みができないことがわかります。

■ スペースクォータの設定の解除

スペースクォータの設定を解除するには、「hdfs dfsadmin」コマンドに「-clrSpaceQuota」オプションを付与し、ディレクトリ名を指定します。

```
# whoami
root

# hdfs dfsadmin -clrSpaceQuota /user/koga/testdir01
# su - koga
$ hdfs dfs -put /home/koga/file200MB-0? /user/koga/testdir01
```

```
$ hdfs dfs -ls /user/koga/testdir01/
Found 2 items
-rw-r--r--   3 koga supergroup  ... /user/koga/testdir01/file200MB-01
-rw-r--r--   3 koga supergroup  ... /user/koga/testdir01/file200MB-02
```

4-1-8　レプリケーション・ファクタの変更

hdfs-site.xml ファイルに dfs.replication パラメーターを設定することによって、クラスター全体のレプリケーション・ファクタを構成できますが、特定のディレクトリ配下のファイルに対してのみにレプリケーション・ファクタを変更したい場合があります。Hadoop 基盤では、データの重要度に基づいてレプリケーション・ファクタを異なる設定にする運用も見られます。たとえば、分析などで頻繁に利用するデータは、ホットデータと呼ばれ、あまり重要度の高くないデータや頻繁にアクセスしないデータは、**コールドデータ**と呼ばれますが、ホットデータについては、レプリケーション・ファクタの値を 3 に設定し、あまり使用頻度が高くない履歴データや重要度の低いデータについては、レプリケーション・ファクタを小さくするといった運用も見られます。以下は、HDFS の/user/koga ディレクトリのレプリケーション・ファクタを 2 に設定する例です。まず、HDFS の/user/koga ディレクトリにテスト用のファイル「testfile」をコピーし、そのファイルのレプリケーション・ファクタを確認します。

```
$ echo "Hello Hadoop 3." > testfile
$ hdfs dfs -rm -f -R /user/koga/*
$ hdfs dfs -put ./testfile /user/koga/
$ hdfs dfs -ls /user/koga/
Found 1 items
-rw-r--r--   3 koga supergroup         16 2018-04-14 22:55 /user/koga/testfile
```

アクセス権の「-rw-r--r--」の右側にある数字がレプリケーション・ファクタを表します。現在、レプリケーション・ファクタは 3 に設定されていることがわかります。HDFS の/user/koga ディレクトリのレプリケーション・ファクタを 2 に設定します。レプリケーション・ファクタの変更は、「hdfs dfs」コマンドに「-setrep -w」オプション†を付与し、その後にレプリケーション・ファクタの数値と対象となる HDFS のディレクトリを指定します。

```
$ hdfs dfs -setrep -w 2 /user/koga
Replication 2 set: /user/koga/testfile
Waiting for /user/koga/testfile ...
```

```
WARNING: the waiting time may be long for DECREASING the number of replications.
. done
```

† -w オプションは、レプリケーションが完了するまで待機することを意味します。

HDFS の/user/koga ディレクトリに保管されているファイル「testfile」のレプリケーション・ファクタを確認します。

```
$ hdfs dfs -ls /user/koga/
Found 1 items
-rw-r--r--   2 koga supergroup         16 2018-04-04 22:55 /user/koga/testfile
```

HDFS の/user/koga ディレクトリに存在するファイル「testfile」のレプリケーション・ファクタが 2 になりました。

4-1-9 HDFS のブロックの状態確認

HDFS に保存されているデータ（ファイルやディレクトリなど）は、ブロックという単位に分割されて複数ノードに記録されます。Hadoop では、ブロックのレプリカが複数ノードに記録されるため、ノード障害が発生しても残りのレプリカを使ってデータを維持することができます。また、障害ノードを交換し、正常なノードが追加されれば、レプリカが生成されます。Hadoop クラスターの管理では、このブロックのレプリカの状況を把握しておくことが重要です。一般に、Hadoop クラスターにおいて、レプリカの数は 3 以上に設定しますが、ディスク障害が発生したワーカーノードのデータブロックが失われると、当然、そのデータに関するレプリカの数が減少します。障害ノードを復旧させた後に、減少したレプリカ数が元に戻っていれば、ワーカーノードのディスク障害に関するブロック損失から復旧したとみなすことができます。

ブロックの状態を確認するには、「hdfs dfsadmin」コマンドに、「-metasave」オプションを付与し、ブロックの状態に関する情報を記録するログファイル名を指定します。

```
# hdfs dfsadmin -metasave hdfs-report-0001.log
Created metasave file hdfs-report-0001.log in the log directory of namenode hdf
s://n0121.jpn.linux.hpe.com:9000
```

ログファイル「hdfs-report-0001.log」は、マスターノードの$HADOOP_HOME/logs ディレクトリに記録されます。

```
# clush -g nn ". .bash_profile; cat $HADOOP_HOME/logs/hdfs-report-0001.log"
n0121: 40 files and directories, 13 blocks = 53 total filesystem objects
n0121: Live Datanodes: 3
n0121: Dead Datanodes: 0
n0121: Metasave: Blocks waiting for reconstruction: 0
n0121: Metasave: Blocks currently missing: 0
n0121: Mis-replicated blocks that have been postponed:
n0121: Metasave: Blocks being reconstructed: 0
n0121: Metasave: Blocks 0 waiting deletion from 0 datanodes.
n0121: Corrupt Blocks:
n0121: Metasave: Number of datanodes: 3
n0121: 10.0.0.122:9866 IN 3219650514944(2.93 TB) 22249472(21.22 MB) 0.00% 32195
94424320(2.93 TB) 0(0 B) 0(0 B) 100.00% 0(0 B) Sat Apr 14 16:21:27 JST 2018
n0121: 10.0.0.123:9866 IN 3219650514944(2.93 TB) 22249472(21.22 MB) 0.00% 32195
94424320(2.93 TB) 0(0 B) 0(0 B) 100.00% 0(0 B) Sat Apr 14 16:21:27 JST 2018
n0121: 10.0.0.124:9866 IN 3219650514944(2.93 TB) 22249472(21.22 MB) 0.00% 32195
94424320(2.93 TB) 0(0 B) 0(0 B) 100.00% 0(0 B) Sat Apr 14 16:21:28 JST 2018
```

ファイルとディレクトリの総数、ライブノード、デッドノードの数に加え、再構成を待機しているブロックの数、欠損しているブロック数、再構成中のブロック数などが表示されます。デッドノードが存在する場合は、OS の稼働状態に加え、Hadoop のサービス（DataNode サービス）の状態を jps コマンドで確認します。

4-1-10 ワーカーノード（DataNode）の増設

ここでは、Apache Hadoop 3 クラスター全体を停止させることなく、ワーカーノード（DataNode）を増設する手順を説明します。 まずは、増設する DataNode となるマシン（ホスト名は、n0125.jpn.linux.hpe.com とします）に、既存の Hadoop クラスターと同じ Linux をインストールし、以下に挙げる OS の事前設定† はすべて済ませておいてください。

> † 大規模環境では、事前設定を施した OS イメージを取得し、増設ノードにイメージを展開する、あるいは、Kickstart やスクリプトなどを駆使し、事前設定を自動化する運用が見られます。

1. 増設ノードでパッケージの更新、java-1.8.0-openjdk と java-1.8.0-openjdk-devel のインストール
2. DNS サーバーによる名前解決がない場合は、全ノードの/etc/hosts ファイルに増設ノードを追記
3. DNS サーバーによる名前解決を行う場合は、DNS サーバーに増設ノードのエントリを追加

4. clush コマンドを実行するクライアントノードから増設ノードに対してパスワード入力なしの SSH 接続を設定（root アカウント用）
5. クライアントノードの clush コマンドの設定ファイル/etc/clustershell/groups に増設ノードを追記
6. 増設ノードの Linux カーネルパラメーター、ファイアウォール、SELinux、時刻同期の設定
7. 既存の Hadoop クラスターに存在するユーザーと同じ UID、GID を持つユーザーを増設ノードに作成
8. Hadoop ユーザーに、既存の Hadoop クラスターに存在するユーザーと同じパスワードを付与
9. クライアントノードから増設ノードに対してパスワード入力なしの SSH 接続を設定（一般ユーザー用）
10. クラスターノードの/root/.bash_profile ファイルを増設ノードの/root ディレクトリにコピー
11. クラスターノードの一般ユーザーの$HOME/.bash_profile ファイルを増設ノードのユーザーのホームディレクトリにコピー

■ 増設ノードのデータ用ディスクのフォーマット

増設ノードのデータ用ディスクをフォーマットしますが、オペレーションミスによって既存の Hadoop クラスターのディスクをフォーマットしてしまう事故を防止するため、今回は、clush コマンドを使用せずに増設ノードに SSH 接続して作業します。物理ディスクのパーティション作成やフォーマット作業を行う際は、必ず、ホスト名やユーザー名を確認して作業してください。まず、増設ノードに SSH 接続します。

```
# ssh n0125-mgm
# hostname
n0125.jpn.linux.hpe.com

# whoami
root
```

以下の手順で、増設ノードのデータ用ディスク/dev/sdb をフォーマットし、/data ディレクトリにマウントします。

```
# yum install -y gdisk
# sgdisk -Z /dev/sdb
# reboot
# parted -s /dev/sdb mklabel gpt
```

```
# parted -s /dev/sdb -- mkpart primary xfs 1 -1
# partprobe /dev/sdb
# mkfs.xfs -f -i size=512 /dev/sdb1
# mkdir /data
# echo '/dev/sdb1 /data xfs defaults 0 0' >> /etc/fstab
# mount /data
```

データ用のパーティション/dev/sdb1 を/data ディレクトリにマウントできたら、増設ノードを再起動します。

```
# hostname
n0125.jpn.linux.hpe.com

# reboot
```

OS 再起動後、/data ディレクトリが自動的にマウントされていることを確認します。

```
# hostname
n0125.jpn.linux.hpe.com

# df -HT | grep "/data"
/dev/sdb1      xfs       3.0T  2.7G  3.0T   1% /data
```

■ 増設ノードの HDFS 用ディレクトリの作成

既存の DataNode とまったく同じディレクトリ構成、同じ所有者、所有グループ、所有権で、/data/hadoop/hdfs ディレクトリを作成します。データ用ディスクのパーティションが/data ディレクトリにマウントされていることが前提です。

```
# hostname
n0125.jpn.linux.hpe.com

# mkdir -p /data/hadoop/hdfs
# chown hdfs:hadoop /data/hadoop/hdfs
# chmod 775 /data/hadoop/hdfs
```

■ Hadoop のディレクトリのコピー

以下は、クライアントノードで作業します。/opt/hadoop-3.1.0 ディレクトリ以下すべてを増設ノードにコピーします。

```
# hostname
n0120.jpn.linux.hpe.com

# clush -w n0125 -c /opt/hadoop-3.1.0 --dest=/opt/
```

■ ワーカーノードのホスト名の追記

増設ノードのホスト名を$HADOOP_HOME/etc/hadoop/workers ファイルに追記し、全ノードにコピーします。

```
# hostname
n0120.jpn.linux.hpe.com

# . /root/.bash_profile
# echo $HADOOP_HOME
/opt/hadoop-3.1.0

# cat $HADOOP_HOME/etc/hadoop/workers
n0122.jpn.linux.hpe.com
n0123.jpn.linux.hpe.com
n0124.jpn.linux.hpe.com

# echo "n0125.jpn.linux.hpe.com" >> $HADOOP_HOME/etc/hadoop/workers
# cat $HADOOP_HOME/etc/hadoop/workers
n0122.jpn.linux.hpe.com
n0123.jpn.linux.hpe.com
n0124.jpn.linux.hpe.com
n0125.jpn.linux.hpe.com

# clush -a -c \
$HADOOP_HOME/etc/hadoop/workers --dest=$HADOOP_HOME/etc/hadoop/
```

■ 増設ノードの既存 Hadoop クラスターへの参加

増設ノードを Hadoop クラスターに参加させます。増設ノードにログインし、サービスを起動します。

```
# clush -w n0125 ". $HOME/.bash_profile; which hdfs"
n0125: /opt/hadoop-3.1.0/bin/hdfs
```

```
# clush -w n0125 ". $HOME/.bash_profile; hdfs --daemon start datanode"
# clush -w n0125 "jps | grep -v Jps"
n0125: 1284 DataNode
```

増設した DataNode ノードに DataNode サービスに関する致命的なエラーがないかログを確認します。

```
# clush -w n0125 \
". $HOME/.bash_profile; less $HADOOP_HOME/logs/hadoop-*-datanode-*.log" \
| less
```

この時点で、増設ノードが DataNode として Hadoop クラスターに参加しているかを確認します。以下の例では、3 ノードの Hadoop クラスターに増設ノード n0125.jpn.linux.hpe.com が 1 台追加され、合計 4 台の DataNode で Hadoop クラスターが拡張されていることがわかります。

```
# hdfs dfsadmin -report |egrep 'Live|Name'
Live datanodes (4):
Name: 10.0.0.122:9866 (n0122.jpn.linux.hpe.com)
Name: 10.0.0.123:9866 (n0123.jpn.linux.hpe.com)
Name: 10.0.0.124:9866 (n0124.jpn.linux.hpe.com)
Name: 10.0.0.125:9866 (n0125.jpn.linux.hpe.com)
```

次に、NodeMamager サービスを起動します。

```
# clush -w n0125 ". $HOME/.bash_profile; yarn --daemon start nodemanager"
# clush -w n0125 "jps | grep -v Jps"
n0125: 1284 DataNode
n0125: 1439 NodeManager
```

増設ノードの NodeManager サービスに関する致命的なエラーがないかログを確認します。

```
# clush -w n0125 \
". $HOME/.bash_profile; less $HADOOP_HOME/logs/hadoop-*-nodemanager*.log" | less
```

■ アプリケーションの動作確認

増設ノードにおいて DataNode サービスと NodeManager サービスが起動したら、モンテカルロシミュレーションによる円周率の近似値を求めるプログラムを実行し、増設ノードを含む全ワーカーノードで CPU 負荷が高くなるかどうかを確認してください。

```
# hostname
n0120.jpn.linux.hpe.com

# su - koga
$ which hadoop
/opt/hadoop-3.1.0/bin/hadoop

$ hadoop jar \
/opt/hadoop-3.1.0/share/hadoop/mapreduce/\
hadoop-mapreduce-examples-3.1.0.jar pi 8 10000
...
Job Finished in 19.404 seconds
Estimated value of Pi is 3.14145000000000000000
```

Column　コマンドライン環境で CPU 負荷を棒グラフ表示するツール

Hadoop クラスターのアプリケーションの動作確認では、全ワーカーノードの CPU やメモリを使って計算しているかどうかのチェックを行うことが少なくありません。VNC ビューワーなどのデスクトップ GUI を表示できる環境がなく、コマンドラインのみが表示できるターミナル・エミュレータ・ソフトウェアなどを使って、遠隔から Hadoop クラスターをメンテナンスしている場合は、コマンドラインで管理対象ノードの負荷を確認したい場合があります。

Linux では、システム負荷を見るコマンドラインのツールとして top コマンドがありますが、top コマンドよりも視覚的に CPU やメモリの負荷を棒グラフで確認できる htop コマンドがあります。コマンドラインのみの環境でも棒グラフで確認できるため、導入をおすすめします。htop コマンドは、CentOS 環境の場合、負荷を表示させたい管理対象となるノードに EPEL リポジトリを追加し、yum コマンドでインストール可能です。

```
# clush -g cl,all "yum makecache fast && yum install -y epel-release"
# clush -g cl,all "yum install -y htop"
```

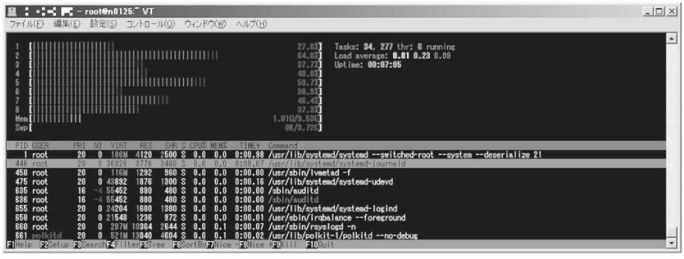


図 4-1　Windows 端末からワーカーノードに接続

図 4-1 では、Windows のターミナル・エミュレータ・ソフトウェアで遠隔にある CPU が 8 コアのワーカーノードに接続し、ワーカーノードのコマンドラインから htop を起動し、8 コアの CPU 負荷を棒グラフで表示しています。

4-1-11 ワーカーノード（DataNode）の離脱

日常のメンテナンスや古くなったワーカーノードは、稼働中の Hadoop クラスターから離脱させる必要があります。Hadoop クラスターにおいて、ワーカーノードを離脱させるには、ワーカーノードの DataNode サービスおよび NameNode サービスを無効にする処理が必要です。この無効にする処理は、一般に**デコミッション**と呼ばれます。以下では、ワーカーノード（DataNode サービスと NameNode サービスが稼働するホスト）をデコミッションさせる手順を紹介します。

■ パラメーターの確認

ワーカーノードのデコミッションには、`hdfs-site.xml` ファイルと `yarn-site.xml` ファイル内に無効化したいワーカーノードのホスト名を記述するファイル名がフルパスで指定されている必要があります。全クラスターノードの `hdfs-site.xml` ファイル内のパラメーター「`dfs.hosts.exclude`」と、`yarn-site.xml` ファイル内のパラメーター「`yarn.resourcemanager.nodes.exclude-path`」が適切に設定されているかを確認します。

```
# hostname
n0120.jpn.linux.hpe.com

# clush -g cl,all -L \
". $HOME/.bash_profile; grep -A 1 'dfs.hosts.exclude' \
$HADOOP_HOME/etc/hadoop/hdfs-site.xml"
n0120:   <name>dfs.hosts.exclude</name>
n0120:   <value>/opt/hadoop-3.1.0/etc/hadoop/dfs.exclude</value>
...

# clush -g cl,all -L \
". $HOME/.bash_profile; grep -A 1 'exclude-path' \
$HADOOP_HOME/etc/hadoop/yarn-site.xml"
n0120:   <name>yarn.resourcemanager.nodes.exclude-path</name>
n0120:     <value>/opt/hadoop-3.1.0/etc/hadoop/yarn.exclude</value>
...
```

■ DataNode のデコミッション

NameNode（今回の場合は、n0121.jpn.linux.hpe.com）上の$HADOOP_CONF_DIR/dfs.exclude ファイルに停止させたい DataNode のホスト名を記述します。今回は、ワーカーノードの n0125.jpn.linux.hpe.com をデコミッションします。複数の DataNode をデコミッションする場合は、dfs.exclude ファイル内にホスト名を複数行にわたって記述します。以下は、NameNode の dfs.exclude ファイルに n0125.jpn.linux.hpe.com を記述する例です。

```
# clush -g nn \
". $HOME/.bash_profile; \
cat > $HADOOP_CONF_DIR/dfs.exclude << __EOF__
n0125.jpn.linux.hpe.com
__EOF__"
```

dfs.exclude ファイルの記述を確認します。

```
# clush -g nn \
". $HOME/.bash_profile; cat $HADOOP_CONF_DIR/dfs.exclude"
n0121: n0125.jpn.linux.hpe.com
```

dfs.exclude ファイルにワーカーノードのホスト名が記述されていることが確認できたら、DataNode サービスが稼働するワーカーノードのデコミッション処理を開始します。

```
# hdfs dfsadmin -refreshNodes
Refresh nodes successful
```

コマンドが成功すると、DataNode サービスが稼働しているワーカーノードのデコミッション処理が開始されます。

■ デコミッション処理の確認

dfs.exclude で指定したワーカーノードの状態を確認します。

```
# hdfs dfsadmin -report | grep -B 1 Decommission
...
Hostname: n0125.jpn.linux.hpe.com
Decommission Status : Decommission in progress
...
```

上記のように、dfs.exclude で指定したワーカーノードにおいて「Decommission in progress」

が表示されていれば、DataNode のデコミッション処理が進行中です。しばらく時間が経過すると、「Decommission in progress」が、「Decommissioned」になります。

```
# hdfs dfsadmin -report | grep -B 1 Decommission
...
Hostname: n0125.jpn.linux.hpe.com
Decommission Status : Decommissioned
...
```

「Decommissioned」になれば、DataNode サービスに関しては、デコミッションが完了したので、DataNode サービスを停止させます。

```
# clush -w n0125-mgm \
'. $HOME/.bash_profile; hdfs --daemon stop datanode'
```

この時点で、ワーカーノード上の DataNode サービスのデコミッションが完了し、DataNode サービスのみが停止しましたが、NodeManager サービスは、デコミッションされていない点に注意してください。現時点において、ワーカーノード n0125 で稼働しているサービスを確認します。

```
# clush -w n0125 jps
n0125-mgm: 8503 NodeManager
n0125-mgm: 8879 Jps
```

Note　DataNode の再有効化

無効化した DataNode を再び有効化するには、dfs.exclude ファイルの当該ノードの記述を削除し、hdfs dfsadmin コマンドで DataNode を起動します。

```
# ssh n0121-mgm
# vi $HADOOP_CONF_DIR/dfs.exclude  ← 無効化したノードのホスト名の記述を削除
# hdfs dfsadmin -refreshNodes
# exit
# clush -w n0125-mgm ". $HOME/.bash_profile; hdfs --daemon start datanode"
```

■ NodeManager のデコミッション

DataNode サービスの停止手順と同様に、NameNode 上の$HADOOP_CONF_DIR/yarn.exclude ファイルに NodeManager のホスト名を記述し、NodeManager サービスが稼働するワーカーノードをデコミッションします。Hadoop クラスターから離脱させるワーカーノード「n0125.jpn.linux.hpe.com」を yarn.exclude ファイルに記述します。

```
# hostname
n0120.jpn.linux.hpe.com

# clush -g nn \
". $HOME/.bash_profile; \
cat > $HADOOP_CONF_DIR/yarn.exclude << __EOF__
n0125.jpn.linux.hpe.com
__EOF__"
```

NameNode 上の$HADOOP_CONF_DIR/yarn.exclude ファイルの記述を確認します。

```
# clush -g nn \
". $HOME/.bash_profile; cat $HADOOP_CONF_DIR/yarn.exclude"
n0121: n0125.jpn.linux.hpe.com
```

yarn.exclude ファイルにワーカーノードホスト名が記述されていることが確認できたら、NodeManager サービスが稼働するワーカーノードをデコミッションします。

```
# yarn rmadmin -refreshNodes
```

コマンドが成功すると、NodeManager サービスが稼働しているワーカーノードのデコミッション処理が開始されます。

■ NodeManager の無効化処理の確認

yarn.exclude で指定したワーカーノードの状態を確認します。

```
# yarn node -list
...
n0123.jpn.linux.hpe.com:42455   RUNNING n0123.jpn.linux.hpe.com:8042 ...
n0122.jpn.linux.hpe.com:37854   RUNNING n0122.jpn.linux.hpe.com:8042 ...
n0124.jpn.linux.hpe.com:39522   RUNNING n0124.jpn.linux.hpe.com:8042 ...
```

上記のように、yarn.exclude で指定したワーカーノード（今回の場合は、n0125.jpn.linux.hpe.com）

が表示されなければ、NodeManager サービスが稼働するワーカーノードのデコミッションは完了です。この時点で、即座に NodeManager サービスも停止します。ワーカーノードのサービスの状況を確認します。

```
# clush -w n0125-mgm jps
n0125-mgm: 6313 Jps
```

以上で、DataNode および、NodeManager 両方に関する無効化が完了し、ワーカーノードを Hadoop クラスターから切り離すことができました。

Note　NodeManager の再有効化

無効化した NodeManager を再び有効化するには、`yarn.exclude` ファイルの当該ノードの記述を削除し、`yarn` コマンドで NodeManager を起動します。

```
# ssh n0121-mgm
# vi $HADOOP_CONF_DIR/yarn.exclude   ← 無効化したノードのホスト名の記述を削除
# exit
# yarn rmadmin -refreshNodes
# clush -w n0125-mgm ". $HOME/.bash_profile; yarn --daemon start nodemanager"
```

4-1-12 HDFS の再バランス

新しいワーカーノードを追加すると、追加したワーカーノードと既存の Hadoop クラスターにおけるワーカーノード群の HDFS の使用量に大きな差が生まれます。Hadoop クラスターでは、ワーカーノードを追加しても、追加ノードと既存ノード群で自動的に HDFS の使用量の差を埋めることはありません。そのため、HDFS の再バランスは、管理者が手動で行う必要があります。

今回は、ワーカーノードの n0125.jpn.linux.hpe.com を追加し、ワーカーノードが 5 台になった Hadoop クラスターの HDFS の再バランスを行います。まずは、再バランス前の HDFS の使用率を確認しておきます。

```
# hostname
n0120.jpn.linux.hpe.com

# hdfs dfsadmin -report | grep "DFS Used%"
```

```
DFS Used%: 18.00%
DFS Used%: 23.96%
DFS Used%: 23.96%
DFS Used%: 23.96%
DFS Used%: 0.04%
```

上記より、追加したワーカーノード（n0125.jpn.linux.hpe.com）は、他のワーカーノード群に比べて、HDFS の使用率が極端に低い状態になっていることがわかります。では、実際に再バランスを行いますが、Apache Hadoop では、HDFS の再バランスの際に、ワーカーノード間のブロック移動の帯域幅を設定できます。今回は、帯域幅を 1GB/秒に設定します。

```
# hdfs dfsadmin -setBalancerBandwidth 1073741824
Balancer bandwidth is set to 1073741824
```

HDFS の再バランスを行います。

```
# hdfs balancer -threshold 5
```

上記の場合、各ワーカーノードのディスク使用量がクラスター全体の平均ディスク使用量の±5 パーセントの範囲で、使用率の高いワーカーノードから使用率の低いワーカーノードにブロックが移動します。

4-1-13 イレイジャーコーディング

第 1 章でも述べたように、Apache Hadoop 3 からイレイジャーコーディング（以下 EC）がサポートされています。以下では、特定のディレクトリに対して EC を設定します。

■ ポリシーの一覧を表示

まず、HDFS で提供される EC に関するポリシーの一覧を表示します。EC のポリシーの一覧を表示するには、「`hdfs ec`」コマンドに「`-listPolicies`」オプションを付与して実行します。

```
# hostname
 n0120.jpn.linux.hpe.com

# hdfs ec -listPolicies
Erasure Coding Policies:
ErasureCodingPolicy=[Name=RS-10-4-1024k, ... , State=DISABLED
```

```
ErasureCodingPolicy=[Name=RS-3-2-1024k, ... , State=DISABLED
ErasureCodingPolicy=[Name=RS-6-3-1024k, ... , State=ENABLED
ErasureCodingPolicy=[Name=RS-LEGACY-6-3-1024k, ... , State=DISABLED
ErasureCodingPolicy=[Name=XOR-2-1-1024k, ... , State=DISABLED
```

デフォルトでは、ポリシー「RS-6-3-1024k」が ENABLED になっています。このポリシーは、データ用に 6 台、パリティ用に 3 台、合計 9 台のワーカーノードが最低必要です。今回は、データ用 3 台、パリティ用 2 台で構成できるポリシー「RS-3-2-1024k」を使います。

■ ポリシーの有効化

ポリシー「RS-3-2-1024k」に最低限必要な 5 台のワーカーノードが Hadoop クラスターとして稼働していることが確認できたら、ポリシーを ENABLED に変更します。

```
# hdfs ec -enablePolicy -policy RS-3-2-1024k
Erasure coding policy RS-3-2-1024k is enabled

# hdfs ec -listPolicies
Erasure Coding Policies:
ErasureCodingPolicy=[Name=RS-10-4-1024k, ... , State=DISABLED
ErasureCodingPolicy=[Name=RS-3-2-1024k, ... , State=ENABLED
ErasureCodingPolicy=[Name=RS-6-3-1024k, ... , State=ENABLED
ErasureCodingPolicy=[Name=RS-LEGACY-6-3-1024k, ... , State=DISABLED
ErasureCodingPolicy=[Name=XOR-2-1-1024k, ... , State=DISABLED
```

■ ポリシーの適用

HDFS に/ec01 ディレクトリを作成し、ポリシー「RS-3-2-1024k」を適用します。

```
# hdfs dfs -mkdir /ec01
# hdfs ec -setPolicy -path /ec01 -policy RS-3-2-1024k
Set erasure coding policy RS-3-2-1024k on /ec01
```

HDFS の/ec01 ディレクトリにポリシー「RS-3-2-1024k」が適用されているかを確認します。

```
# hdfs ec -getPolicy -path /ec01
RS-3-2-1024k
```

■ ブロックグループの確認

HDFS 上の/ec01 ディレクトリにテスト用のファイル「file2GB」を書き込みます。

```
# dd if=/dev/zero of=$HOME/file2GB bs=1024M count=2
# hdfs dfs -put $HOME/file2GB /ec01/
```

EC のブロックグループに関する情報を表示します。

```
# hdfs fsck / | grep -A 15 "Erasure Coded Block Groups"
Connecting to namenode via http://n0121.jpn.linux.hpe.com:9870/fsck?ugi=koga&pa
th=%2F
Erasure Coded Block Groups:
 Total size:    2147483648 B
 Total files:   1
 Total block groups (validated):        6 (avg. block group size 357913941 B)
 Minimally erasure-coded block groups:  6 (100.0 %)
 Over-erasure-coded block groups:       0 (0.0 %)
 Under-erasure-coded block groups:      0 (0.0 %)
 Unsatisfactory placement block groups: 0 (0.0 %)
 Average block group size:      5.0
 Missing block groups:          0
 Corrupt block groups:          0
 Missing internal blocks:       0 (0.0 %)
FSCK ended at Mon Jan 15 02:57:54 JST 2018 in 763 milliseconds


The filesystem under path '/' is HEALTHY
```

「Total block groups (validated):」の値から、保管したデータ「file2GB」は、6 個のブロックグループから構成されていることがわかります。

ポリシーの適用を解除するには、「hdfs ec」コマンドに「-unsetPolicy」オプションを付与します。

```
# hdfs ec -unsetPolicy -path /ec01
Unset erasure coding policy from /ec01
```

4-1-14 HDFS のファイルシステムチェック

HDFS は、イレイジャーコーディングや、レプリケーションなどのファイルの保護機能がありますが、ディスク障害等、なんらかの理由により、ブロックに欠損が見つかった場合は、GUI 管理画面に欠損ブロックに関する情報が表示されます。図 4-2 は、マスターノードの 9870 番ポートで提供される GUI 管理画面で表示されている、HDFS の欠損ブロックに関する情報の例です。

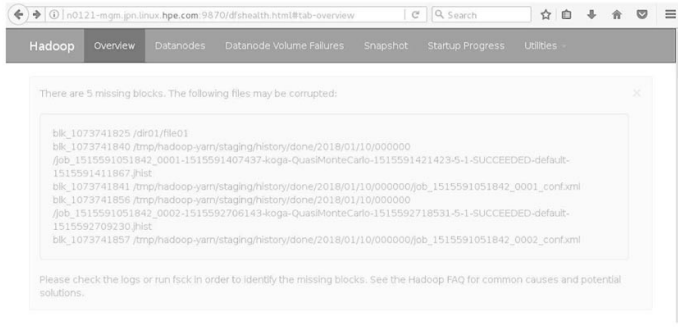


図 4-2　HDFS の欠損ブロックに関する情報

HDFS 上に保管されているブロックが読み込めるかどうかをファイルごとに確認するには、以下のように、`hdfs fsck` コマンドを入力します。

```
# hostname
n0120.jpn.linux.hpe.com

# hdfs fsck / -files -blocks > /tmp/hdfs_status0001.log
# grep MISSING /tmp/hdfs_status0001.log
...
/dir01/file01 10485760 bytes, replicated: replication=3, 1 block(s):  MISSING 1
 blocks of total size 10485760 B
...
  MISSING BLOCKS:       5
  MISSING SIZE:         10966610 B
```

「`Missing blocks`」の値が 1 以上の場合は、HDFS 上に保管したファイルの一部が読み込みできない状態になっている可能性があります。しかし、読み込めなくなったファイルは、別のファイルサーバーなどから正常なファイルをコピーするなどの復旧措置が必要になります。読み込めなくなったファイルを削除しておきます。

```
# hdfs fsck / -delete
```

しばらくしてから、再度、HDFS に格納されているファイル群のブロック状態を確認します。

```
# hdfs fsck / -files -blocks > /tmp/hdfs_status0002.log
# grep Missing /tmp/hdfs_status0002.log
 Missing blocks:                0
 Missing replicas:              0 (0.0 %)
 Missing block groups:          0
 Missing internal blocks:       0
```

「Missing blocks」が 0 であれば、HDFS に欠損ブロックが存在しないと判断できます。次に、壊れたブロックがないかも確認しておきます。

```
# hdfs fsck -list-corruptfileblocks
...
The filesystem under path '/' has 0 CORRUPT files
```

「The filesystem under path '/' has 0 CORRUPT files」と表示されれば、HDFS に壊れたブロックは存在しないと判断できます。最後に、HDFS の状態を確認します。

```
# hdfs fsck / -blocks -files | tail -1
...
The filesystem under path '/' is HEALTHY
```

4-1-15 HDFS におけるストレージ階層化

一般に、時間の経過に伴って、データはそのアクセス頻度が推移します。新しいデータを入手、あるいは、生成されて間もない頃は、頻繁にアクセスされる傾向にあります。頻繁にアクセスされるデータは、**ホットデータ**と呼ばれます。しかし、数週間以上経過すると、アクセス頻度は徐々に低下していきます。ホットデータに比べて、アクセス頻度がそれほど高くなくなったデータは、**ウォームデータ**と呼ばれ、数か月が経過すると、データ使用率はさらに低下し、ほとんどアクセスしなくなるデータが出てきます。

Hadoop クラスターで高い性能を引き出すには、SSD などの高速ストレージを大量に搭載すれば実現できますが、初期投資がかさむため、限られた予算でストレージの利用効率を高めるためには、どうしても比較的安価で大容量の SATA ディスクなどを組み合わせる必要があります。たとえば、緊急を要する分析処理や、頻繁にアクセスするホットデータは、SSD などの高速半導体ディ

スク上に保管し、それほど頻繁にアクセスはしないものの、通常の分析業務で利用するウォームデータは、SATA ディスクなどに保管します。あまりアクセスしなくなった古いコールドデータは、別の SATA ディスクにアーカイブ用として保管しておくといった利用形態が考えられます。このようなデータのアクセス頻度に応じて、性能の異なるディスクの使い分けは、一般に**ストレージの階層化**と呼ばれます (図 4-3)。Hadoop では、クラスター内にストレージ階層を複数持つことができ、データの利用頻度に応じたストレージ層にデータブロックを配置する機能を備えています。

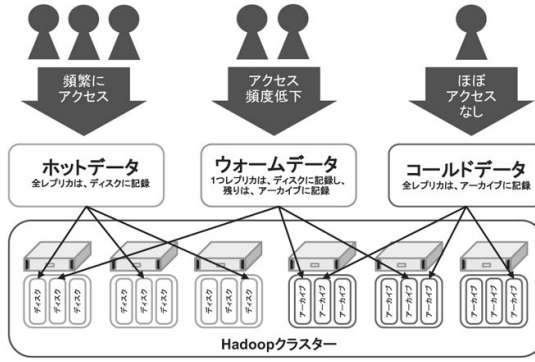


図 4-3　Hadoop におけるストレージの階層化

■ ストレージタイプ

Hadoop では、ストレージの階層化を実現するに当たって、**ストレージタイプ**と**ストレージポリシー**という 2 つの概念が存在します。ストレージタイプは、ディスク性能で分類したもので、**表 4-2** に示す 4 つが存在します。

表 4-2　ストレージタイプ

ストレージタイプ	説明
DISK	HDFS で使用される標準のディスクベースストレージ
ARCHIVE	あまり使用されないデータを格納するアーカイブディスクベースのストレージ
SSD	低遅延アプリ用のデータを格納するフラッシュストレージ
RAM_DISK	物理メモリ上に記憶させるインメモリストレージ

これらのストレージタイプは、hdfs-site.xml ファイル内における DataNode 用ディレクトリパスの先頭に括弧で括ったタグとして記述します（表 4-3）。本書では、ストレージタイプが「DISK」に設定されたストレージ層を DISK タイプのストレージ層または「DISK 層」と呼びます。

表 4-3　ストレージタイプの記述方法

ストレージタイプ	ディレクトリの例	hdfs-site.xml ファイル内での記述例
DISK	/data/disk01	[DISK]file:///data/disk01

ストレージタイプの利用ケースはいろいろありますが、ここで 1 つ例を挙げておきます。たとえば、DataNode に 120TB の内蔵ストレージが搭載されている 100 ノードの Hadoop クラスターが存在するとします。この Hadoop クラスターのストレージ容量は、以下のとおりです。

120TB × 100 = 12000TB = 12PB

12PB のディスク領域には、ストレージタイプとして DISK を設定しておきます。この Hadoop クラスターに対して、さらに 120TB の内蔵ストレージが搭載された DataNode を 50 台（ストレージ容量は、120TB × 50 = 6PB）追加したとします。この 50 台の DataNode では、内蔵ディスクのストレージタイプを ARCHIVE に設定します。これにより、1 つの Hadoop クラスター内に 12PB の DISK タイプのストレージ層と 6PB の ARCHIVE タイプのストレージ層を混在できます。

■ ストレージポリシー

ストレージポリシーは、表 4-4 に示す 6 種類が存在します。主にデータの利用頻度によって、レプリカの格納場所を決めます。

表 4-4　ストレージポリシー

ポリシーの種類	説明
HOT	保管と計算の両方で使用。頻繁にアクセスされるブロックの全レプリカは、DISK 層に格納される
COLD	使用頻度が低くなったデータは、HOT ストレージから COLD ストレージに移動される。頻繁にアクセスされなくなったブロックの全レプリカは、ARCHIVE 層に保存される
WARM	レプリカのいくつかは DISK 層に格納され、残りのレプリカは ARCHIVE 層に格納される
All_SSD	全レプリカが SSD 層に格納される

One_SSD	1 つのレプリカは、SSD 層に格納され、残りのレプリカは DISK 層に格納される
Lazy_Persist	単一のレプリカを持つブロックは、メモリに格納される。レプリカは、最初に RAM_DISK 層に格納され、その後、DISK 層に格納される

■ HDFS のストレージ階層化の設定手順

以下では、HDFS のストレージ階層化の設定手順を紹介します。今回は、通常利用する DISK 層と ARCHIVE 層を作成し、DISK 層にあるブロックを ARCHIVE 層に移動させる例です。全ノードの hdfs-site.xml ファイルを編集するため、全 DataNode サービスを停止します。

```
# hostname
n0120.jpn.linux.hpe.com

# clush -g dn '. $HOME/.bash_profile; hdfs --daemon stop datanode'
```

■ hdfs-site.xml ファイルを編集

hdfs-site.xml ファイルを編集する前に、hdfs-site.xml ファイルのバックアップをとっておきます。

```
# cp -a \
$HADOOP_HOME/etc/hadoop/hdfs-site.xml \
$HADOOP_HOME/etc/hadoop/hdfs-site.xml.bak
```

hdfs-site.xml ファイル内のパラメーター「dfs.datanode.data.dir」に DISK タイプと ARCHIVE タイプを指定します。DISK ストレージタイプの場合は、HDFS 用ディレクトリのフルパス名の前に [DISK] を付与し、ARCHIVE ストレージタイプの場合は、[ARCHIVE] を付与します。

```
# vi $HADOOP_HOME/etc/hadoop/hdfs-site.xml
...
<property>
  <name>dfs.datanode.data.dir</name>
    <value>[DISK]file:///data/hadoop/hdfs/dn,[ARCHIVE]file:///cold/dn</value>
</property>
<property>
...
```

上記の例では、ワーカーノードの HDFS 用のディレクトリ /data/hadoop/hdfs/dn に DISK ストレージタイプを指定し、/cold/dn ディレクトリを ARCHIVE ストレージタイプに指定しています。hdfs-site.xml ファイルを編集したら、全ワーカーノードにコピーします。

```
# clush -a -c \
$HADOOP_HOME/etc/hadoop/hdfs-site.xml \
--dest=$HADOOP_HOME/etc/hadoop/
```

■ ARCHIVE 層の HDFS 用ディレクトリの作成

ARCHIVE ストレージタイプに指定した /cold ディレクトリを作成し、所有者を hdfs、所有グループを hadoop、/cold ディレクトリのアクセス権限を 775 に設定します。

```
# clush -g dn "rm -rf /cold; mkdir /cold"
# clush -g dn "chown hdfs:hadoop /cold"
# clush -g dn "chmod 775 /cold"
```

■ DataNode の起動

hdfs-site.xml ファイルを編集し、ARCHIVE 層の HDFS 用ディレクトリが作成できたので、DataNode サービスを再起動します。

```
# clush -g dn '. $HOME/.bash_profile; hdfs --daemon stop datanode'
# clush -g dn '. $HOME/.bash_profile; hdfs --daemon start datanode'
```

この時点で、DataNode サービスが起動できない場合は、hdfs-site.xml ファイルのパラメーターの設定を再確認してください。

■ ストレージポリシーの表示

利用可能なストレージポリシーを確認します。

```
# hdfs storagepolicies -listPolicies
Block Storage Policies:
        BlockStoragePolicy{COLD:2, storageTypes=[ARCHIVE], ...
        BlockStoragePolicy{WARM:5, storageTypes=[DISK, ARCHIVE], ...
        BlockStoragePolicy{HOT:7, storageTypes=[DISK], ...
```

```
        BlockStoragePolicy{ONE_SSD:10, storageTypes=[SSD, DISK], ...
        BlockStoragePolicy{ALL_SSD:12, storageTypes=[SSD], ...
        BlockStoragePolicy{LAZY_PERSIST:15, storageTypes=[RAM_DISK, DISK], ...
```

■ アーカイブ用のディレクトリの作成とファイルの格納

HDFS 上にアーカイブの/archive01 ディレクトリを作成し、ストレージポリシーをアーカイブ用途の「COLD」に設定します。HDFS の/archive01 ディレクトリには、テスト用のファイル群をコピーします。今回は、クライアントノードの/usr/share/doc ディレクトリ以下で名前の末尾が「.jpg」で終わるファイルを保存します。

```
# hdfs dfs -mkdir /archive01
# find /usr/share -name "*.jpg" | xargs -I{} hdfs dfs -put -f {} /archive01/
```

■ 現在のストレージポリシーの確認

HDFS の/archive01 ディレクトリのストレージポリシーを確認しておきます。

```
# hdfs storagepolicies -getStoragePolicy -path /archive01
The storage policy of /archive01 is unspecified
```

現時点では、まだストレージポリシーが設定されていないことがわかります。

■ ストレージポリシーの設定と確認

HDFS の/archive01 ディレクトリに対して、アーカイブ用途を意味するストレージポリシーの「COLD」を設定します。

```
# hdfs storagepolicies -setStoragePolicy -path /archive01 -policy COLD
Set storage policy COLD on /archive01
```

HDFS の/archive01 ディレクトリのストレージポリシーが「COLD」に変更されているかを確認します。

```
# hdfs storagepolicies -getStoragePolicy -path /archive01
The storage policy of /archive01:
```

```
BlockStoragePolicy{COLD:2, storageTypes=[ARCHIVE], ...
```

■ ブロックの移動

設定した新しいストレージポリシーに対応したアーカイブ用のディスクにブロックを移動させます。

```
# hdfs mover -p /archive01
...
Mover Successful: all blocks satisfy the specified storage policy. Exiting...
Apr 7, 2018 3:44:28 AM  Mover took 1mins, 6sec
```

hdfs mover を定期的に実行し、HDFS の特定ディレクトリに付与したストレージポリシーに基づいて、ストレージタイプのディスク上にブロックを移行することが可能です。あまり利用しないデータは、COLD ポリシーを付与した HDFS のディレクトリに保管し、ブロックを移動させておくことをお勧めします。

4-1-16 Apache Hadoop 3 における管理コマンド例

Apache Hadoop では、管理者向けのコマンドやユーザーが利用するコマンドなど、非常に多くの管理コマンドが用意されています。紙面の都合上、本書ですべてを網羅することはできませんが、以下では、運用管理で利用される、主なコマンドの使用例をまとめておきます。管理者権限で実行する場合は、コマンドプロンプトを「#」で表し、一般ユーザーで実行する場合は、コマンドプロンプトを「$」で表します。

例）一般ユーザー用のディレクトリ/user/koga を HDFS 上に作成

```
# hdfs dfs -mkdir -p /user/koga; hdfs dfs -chown -R koga:supergroup /user/koga
```

例）echo コマンドで指定した文字列のテキストのみを含む testfile を HDFS 上に作成

```
$ echo "Hello Hadoop 3." | hdfs dfs -put -f - /user/koga/testfile
```

例）echo コマンドで指定した文字列のテキストを HDFS 上の testfile に追記

```
$ echo "Hello Bigdata." | hdfs dfs -appendToFile - /user/koga/testfile
```

例）ローカルのファイル「testfile」を HDFS の/user/koga ディレクトリにコピー

```
$ echo "Hello." > ./testfile && hdfs dfs -put -f ./testfile /user/koga/
```

例）HDFS 上にファイル「testfile」が存在すれば、中身を確認

```
$ hdfs dfs -test -f /user/koga/testfile && hdfs dfs -cat /user/koga/testfile
```

例）HDFS のディレクトリ/user/koga 以下のサブディレクトリも含めて再帰的に内容を確認

```
$ hdfs dfs -ls -R /user/koga/
drwxr-xr-x   - koga supergroup   0 2018-02-20 09:42 /user/koga/dir01
-rw-r--r--   3 koga supergroup   7 2018-02-20 09:42 /user/koga/dir01/file01
-rw-r--r--   3 koga supergroup   9 2018-02-20 09:35 /user/koga/testfile
```

例）HDFS に保管されているファイル「testfile」の削除

```
$ hdfs dfs -rm /user/koga/testfile
```

例）HDFS 上のディレクトリ「/user/koga/dir01」以下を再帰的に削除

```
$ hdfs dfs -rm -r /user/koga/dir01
```

例）ローカルのファイル「testfile」を HDFS の/user/koga ディレクトリにコピー

```
$ hdfs dfs -copyFromLocal ./testfile /user/koga/
```

例）ローカルのファイル「testfile」を HDFS 上に強制的にコピー

```
$ hdfs dfs -copyFromLocal -f ./testfile /user/koga/
```

例）HDFS 上のファイル「/user/koga/testfile」をローカルの/tmp にコピー

```
$ hdfs dfs -get /user/koga/testfile /tmp/
```

例）HDFS 上のファイル「/user/koga/testfile」をローカルの/tmp にコピー

```
$ hdfs dfs -copyToLocal /user/koga/testfile /tmp/
```

例）HDFS 上のファイル「/user/koga/testfile」をローカルの/tmp に強制的にコピー

```
$ hdfs dfs -copyToLocal -f /user/koga/testfile /tmp/
```

例）HDFS 上の/user/koga ディレクトリ以下のディレクトリ数、ファイル数、サイズを表示（左からディレクトリ数、ファイル数、サイズ、対象ディレクトリ）

```
$ hdfs dfs -count /user/koga/*
           1            0                  0 /user/koga/dir01
           0            1                  9 /user/koga/testfile
           0            1                 10 /user/koga/testfile2
```

例）HDFS 上の/user/koga/testfile を HDFS 上の/user/koga/testfile.bak としてコピー

```
$ hdfs dfs -cp /user/koga/testfile /user/koga/testfile.bak
```

例）HDFS の容量、使用量を表示

```
$ hdfs dfs -df -h
Filesystem                                Size    Used  Available  Use%
hdfs://n0121.jpn.linux.hpe.com:9000  11.9 T  19.3 M     11.9 T    0%
```

例）HDFS のディレクトリ「/user/koga」のサイズを表示（左からレプリカを含まないサイズ、全レプリカを含んだ合計サイズ、対象となるディレクトリ）

```
$ hdfs dfs -du -h -s /user/koga/
1.1 G  3.3 G  /user/koga
```

例）HDFS のディレクトリ「/user/koga」やファイルのサイズを表示（左からファイルやディレクトリのサイズ、全レプリカを含んだ合計サイズ、対象となるファイルやディレクトリ）

```
$ hdfs dfs -du -h /user/koga/
0      0      /user/koga/dir01
100 M  300 M  /user/koga/file100MB
1 G    3 G    /user/koga/file1GB
9      27     /user/koga/testfile
```

例）HDFS のディレクトリ「/user/koga」配下にファイル「testfile」が存在するかどうかを検索

```
$ hdfs dfs -find /user/koga -name testfile -print
/user/koga/testfile
```

例）HDFS 上のファイル「testfile」のファイル名を「testfile2」に変更

```
$ hdfs dfs -mv /user/koga/testfile /user/koga/testfile2
```

例）HDFS 上のファイル「testfile」を HDFS の/tmp ディレクトリに移動

```
$ hdfs dfs -mv /user/koga/testfile /tmp/
```

例）HDFS 上のファイル「testfile」のパーミッションを 664 に変更

```
$ hdfs dfs -chmod 664 /user/koga/testfile
$ hdfs dfs -ls /user/koga/testfile
  -rw-rw-r--   3 koga supergroup     18 2018-04-07 20:57 /user/koga/testfile
```

例）HDFS のディレクトリ「/user/koga/testdir」のパーミッションを 775 に変更

```
$ hdfs dfs -chmod 775 /user/koga/testdir
$ hdfs dfs -ls -d /user/koga/testdir
drwxrwxr-x   - koga supergroup      0 2018-04-07 21:08 /user/koga/testdir
```

例）HDFS 上のファイル「testfile」の所有者を tanaka に、グループを sales 変更

```
# hdfs dfs -chown tanaka:sales /user/koga/testfile
# hdfs dfs -ls /user/koga/testfile
  -rw-r--r--   3 tanaka sales      18 2018-04-07 20:57 /user/koga/testfile
```

例）ローカルの複数のファイルを HDFS 上のファイルに追記

```
$ echo "file1" > file1; echo "file2" > file2
$ hdfs dfs -appendToFile file1 file2 /user/koga/file1and2
$ hdfs dfs -cat /user/koga/file1and2
file1
file2
```

例）HDFS 上のファイル「testfile」に追記されている内容をリアルタイムで表示

```
$ hdfs dfs -tail -f /user/koga/testfile
```

例）サイズが 0 のファイル「nullfile」を HDFS の/user/koga ディレクトリに作成

```
$ hdfs dfs -touchz /user/koga/nullfile
$ hdfs dfs -ls /user/koga/nullfile
```

```
-rw-r--r--   3 koga supergroup      0 2018-04-07 21:03 /user/koga/nullfile
```

例）HDFS の/user/koga ディレクトリ以下の全ファイルを Hadoop アーカイブ「test01.har」として HDFS 上の/tmp ディレクトリ以下に作成

```
# hadoop archive -archiveName test01.har -p /user/koga /tmp
```

例）HDFS 上の/tmp ディレクトリに作成した Hadoop アーカイブ「test01.har」の中身を確認

```
# hdfs dfs -ls -R har:///tmp/test01.har/
```

例）Hadoop アーカイブ「test01.har」に含まれる testfile を HDFS 上の/tmp ディレクトリに展開

```
# hdfs dfs -cp har:///tmp/test01.har/testfile hdfs:/tmp/
```

例）HDFS の/user/koga ディレクトリ以下のスナップショット snap0001 の取得

```
# hdfs dfsadmin -allowSnapshot /user/koga
$ su - koga
$ hdfs dfs -createSnapshot /user/koga snap0001
$ hdfs dfs -ls /user/koga/.snapshot/snap0001/
Found 1 items
-rw-r--r--   3 koga supergroup ... /user/koga/.snapshot/snap0001/testfile
```

例）HDFS の/user/koga ディレクトリ以下のスナップショット snap0001 からファイルを復元

```
$ hdfs dfs -cp -ptopax /user/koga/.snapshot/snap0001/* /user/koga/
```

例）HDFS の/user/koga ディレクトリ以下のスナップショット snap0001 を削除

```
$ hdfs dfs -deleteSnapshot /user/koga snap0001
```

4-1-17 Apache Hadoop 3 が提供する GUI 管理画面

Hadoop の Web 管理画面では、クラスターノードの死活状態やジョブの実行状態の概要を確認できます。以下では、管理者が知っておくべき Apache Hadoop 3 の主な Web 管理画面を紹介します。

■ クラスターの概要の表示

HTTP プロトコルでマスターノードの 9870 番ポートにアクセス† します。「Overview」と「Summary」にクラスターの概要が表示されます (図 4-4)。

```
$ firefox http://n0121-mgm.jpn.linux.hpe.com:9870 &
```

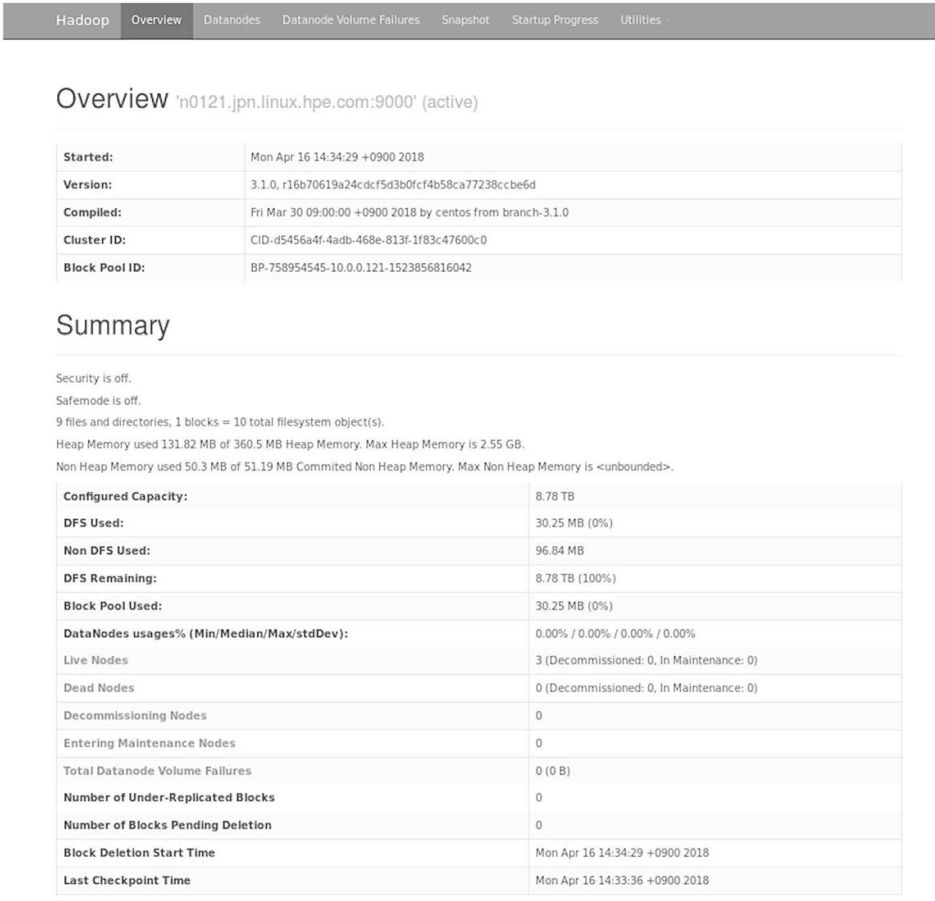


図 4-4　Hadoop クラスターの概要を表示

† Web ブラウザを起動するクライアントマシン側で、アクセス先となる「n0121-mgm.jpn.linux.hpe.com」の名前解決ができることが前提です。名前解決のためには、クライアントマシン側から DNS サーバーへの問い合わせを行う設定が必要です。クライアントマシンから DNS サーバーによる名前解決ができない場合は、クライアントマシン上の/etc/hosts ファイルにアクセス先の IP アドレスとホスト名の対応を記述する必要があります。

さらに、画面の下に遷移すると、NameNode のメタデータ情報が格納されているディレクトリや現在の HDFS の使用量、提供ノード数などが表示されます（図 4-5）。

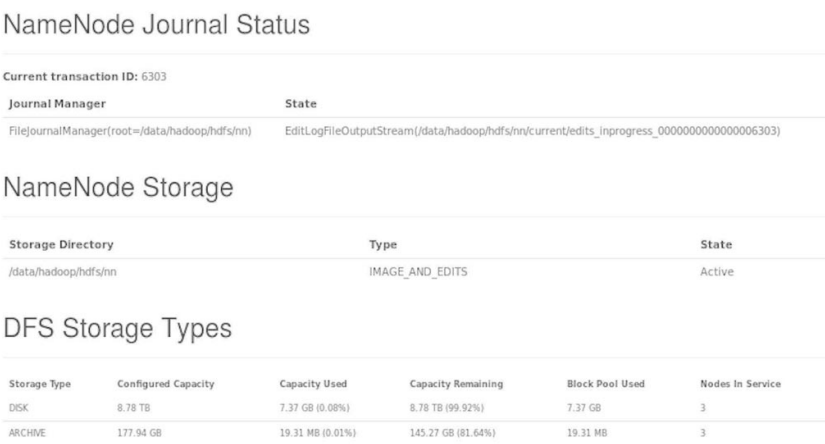


図 4-5 NameNode のメタデータ情報が格納されているディレクトリや、HDFS の容量、使用量、提供ノード数などを表示

■ DataNode の概要を表示

画面上のクラスター概要の［DataNode］をクリックすると、DataNode の概要に関する画面になります。稼働中の DataNode の一覧、提供するディスク容量、消費しているブロック数、Hadoop のバージョンなどが表示されます（図 4-6）。

ワーカーノードの 9864 番ポートをアクセスすると、ワーカーノードが提供している HDFS のボリューム情報が表示されます。図 4-7 は、ワーカーノードの n0122.jpn.linux.hpe.com が提供する Web 管理画面の例です。

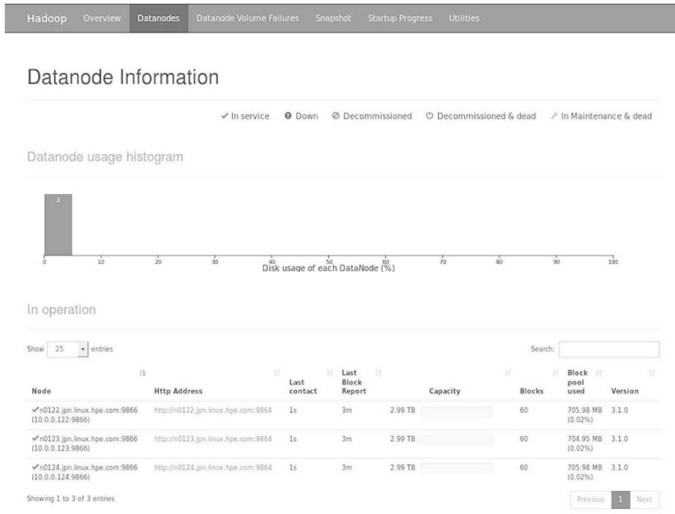


図 4-6　DataNode の概要を表示

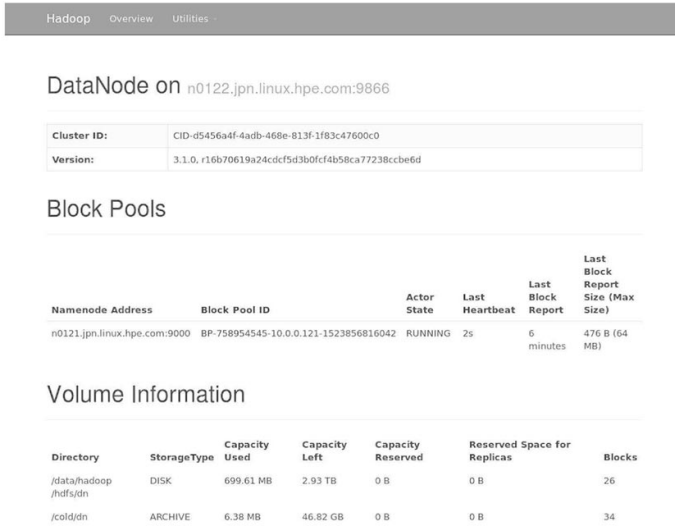


図 4-7　DataNode が提供する HDFS ボリュームに関する情報を表示

■ アプリケーションの状態確認

Hadoop クラスターで実行したアプリケーションの状態を Web 画面で管理するには、ResourceM

anager が提供する管理画面にアクセスする方法と、ジョブの履歴画面にアクセスする方法があります。ResourceManager が提供する管理画面は、ResourceManager サービスが稼働するノード（今回の場合は、マスターノード）の 8088 番ポートで提供されます（図 4-8）。

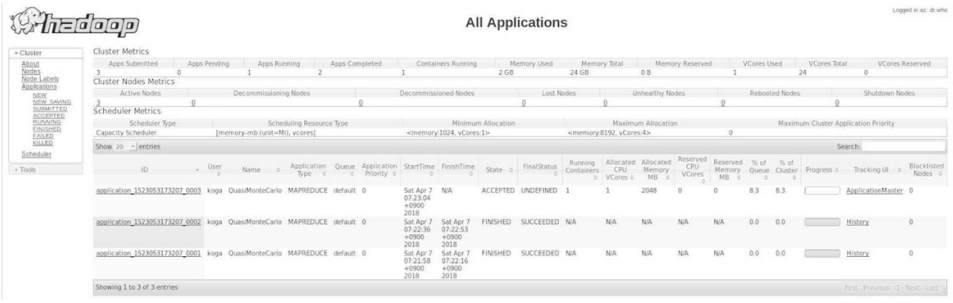


図 4-8　ResourceManager が提供する管理画面を表示

Note　ResourceManager の管理画面が表示されない場合

ResourceManager の管理用 LAN のホスト名（今回の場合は、n0121-mgm.jpn.linux.hpe.com）の 8088 番ポートにアクセスしても ResourceManager の管理画面が表示されない場合は、データ用 LAN のホスト名（n0121.jpn.linux.hpe.com）の 8088 番ポートで ResourceManager の管理画面が提供されている可能性があります。ResourceManager の管理画面の URL を管理用 LAN 側に固定するには、以下のように、`$HADOOP_HOME/etc/hadoop/yarn-site.xml` ファイルにパラメーター「`yarn.resourcemanager.webapp.address`」を記述し、値として、ResourceManager が稼働する管理用 LAN 側のホスト名とポート番号（8088）を指定します。

```
# vi $HADOOP_HOME/etc/hadoop/yarn-site.xml
...
<property>
<name>yarn.resourcemanager.webapp.address</name>
<value>n0121-mgm.jpn.linux.hpe.com:8088</value>
</property>
...
```

`yarn-site.xml` ファイルの編集が完了したら、ファイルを全ノードにコピーし、ResourceManager サービスを再起動します。

```
# clush -a -c $HADOOP_HOME/etc/hadoop/yarn-site.xml \
--dest=$HADOOP_HOME/etc/hadoop

# clush -g nn ". $HOME/.bash_profile; yarn --daemon stop resourcemanager"
```

```
# clush -g nn ". $HOME/.bash_profile; yarn --daemon start resourcemanager"
```

ジョブの履歴を確認するには、マスターノードの 19888 番ポートにアクセスします（図 4-9）。

```
$ firefox http://n0121-mgm.jpn.linux.hpe.comm:19888 &
```

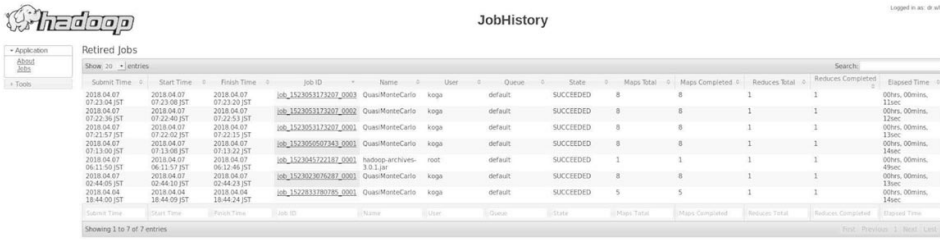


図 4-9　実行したジョブの履歴を表示

さらに、ジョブの履歴画面上の「Job ID」列に表示されているジョブ ID（図の例では、job_1523053173207_0003 など）をクリックすると、そのジョブの詳細が表示されます（図 4-10）。

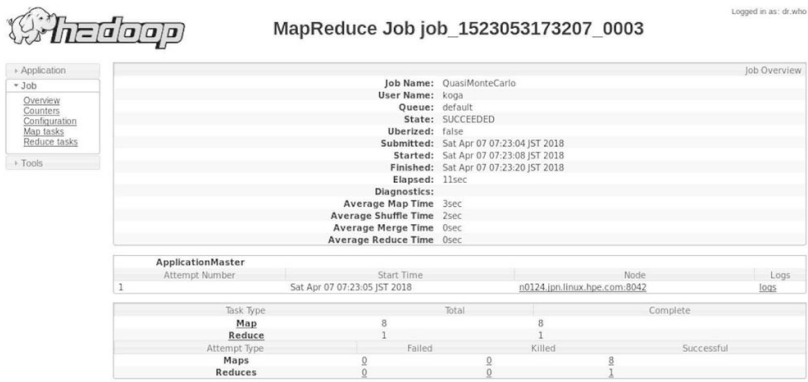


図 4-10　ジョブの詳細を表示

■ NodeManager の概要を表示

ワーカーノードの 8042 番ポートをアクセスすると、ワーカーノードが提供している NodeManager サービスに関する情報が表示されます。図 4-11 は、ワーカーノードの n0122.jpn.linux.hpe.com が提供する、NodeManager サービスに関する Web 管理画面の例です。

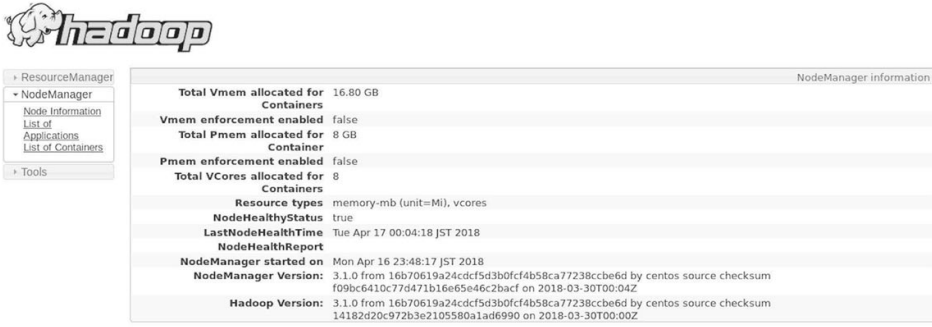


図 4-11　NodeManager の概要を表示

4-2　MapR クラスターの運用管理手法

商用版の Hadoop ディストリビューションである MapR は、コミュニティ版の Apache Hadoop とは異なり、独自の機能、管理コマンド、Web 管理画面が用意されています。ここでは、企業システムの MapR クラスターでよく利用される NFS サーバーとスナップショットによるデータの管理手法を紹介します。また、MapR で提供される、主要な管理コマンドの使用例に加え、MCS、Grafana を使った MapR クラスターの GUI 管理について簡単に紹介します。

4-2-1　MapR クラスターを NFS サーバーとして利用する

Apache Hadoop 3 においても、NFS ゲートウェイの機能が提供されていますが、MapR は、Hadoop のディストリビューションの中でも古くから NFS サービスを提供しており、エンタープライズ Hadoop でのファイルサーバーとしての採用で実績があります。MapR クラスターでは、複数のクラスターノードによって NFS サービスを提供しますが、NFS サービスに対して、仮想 IP アドレスを割り当てます。これにより、クライアントは、複数の NFS サービスを単一の IP アドレスでアクセスできます。NFS サービスを提供するクラスターノードの 1 台に障害が発生しても、NFS

サービスを提供する他のクラスターノードによって仮想 IP アドレスが引き継がれるため、耐障害性のある NFS サーバーを実現できます。

■ MapR クラスター用のファイルシステムタブの記述

MapR クラスターでは、fstab に似た NFS サービスを提供するディレクトリを記述するファイルシステムタブが存在します。NFS サービス用のファイルシステムタブは全クラスターノードに設定します。MapR クラスター用のファイルシステムタブは、mapr_fstab と呼ばれます。まずは、mapr_fstab ファイルを記述します。mapr_fstab ファイル内の「**ホスト名**:/mapr」と「/mapr」の間、「/mapr」と「hard,nolock」の間は、タブではなく半角スペースを入力して記述します。

```
# hostname
n0130.jpn.linux.hpe.com

# clush -a mkdir /mapr
# clush -a \
'echo "$HOSTNAME:/mapr  /mapr  hard,nolock" > \
/opt/mapr/conf/mapr_fstab'

# clush -aL "cat /opt/mapr/conf/mapr_fstab"
n0131: n0131.jpn.linux.hpe.com:/mapr  /mapr  hard,nolock
n0132: n0132.jpn.linux.hpe.com:/mapr  /mapr  hard,nolock
n0133: n0133.jpn.linux.hpe.com:/mapr  /mapr  hard,nolock
```

MapR クラスター全ノードを再起動します。

```
# clush -a reboot
```

NFS サービスが稼働しているかどうかを確認します。

```
# clush -aL "ps -ef | grep nfs | grep -v grep"
n0131: mapr      5079     1  0 18:53 ?        00:00:00 /opt/mapr/server/nfsserver
n0131: mapr      5081  5079  0 18:53 ?        00:00:00 /opt/mapr/server/nfsserver
n0131: root      6984     2  0 18:53 ?        00:00:00 [nfsiod]
n0132: mapr      5022     1  0 18:53 ?        00:00:00 /opt/mapr/server/nfsserver
n0132: mapr      5024  5022  0 18:53 ?        00:00:00 /opt/mapr/server/nfsserver
n0132: root      7036     2  0 18:53 ?        00:00:00 [nfsiod]
n0133: mapr      5314     1  0 18:53 ?        00:00:00 /opt/mapr/server/nfsserver
n0133: mapr      5316  5314  0 18:53 ?        00:00:00 /opt/mapr/server/nfsserver
n0133: root      7438     2  0 18:53 ?        00:00:00 [nfsiod]
```

MapR クラスターが提供する NFS サービス用の/mapr ディレクトリを、クライアントノードか

らマウントできるかどうかを確認します。

```
# mount -o hard,nolock n0131.jpn.linux.hpe.com:/mapr /mnt
# ls -lF /mnt
total 1
drwxr-xr-x 8 mapr mapr 7 Feb 14 11:09 hpe-mapr-cluster01/

# ls -F /mnt/hpe-mapr-cluster01/
apps/  hbase/  opt/  test01/  tmp/  user/  var/
```

クライアントノードから NFS サービス用ディレクトリ内にテスト用のファイル「testfile.txt」を作成できるか確認します。

```
# echo "Hello MapR" > /mnt/hpe-mapr-cluster01/testfile.txt
# cat /mnt/hpe-mapr-cluster01/testfile.txt
Hello MapR
```

■ NFS マウントされた MapR-FS を確認

MapR クラスターの Mapr-FS のディレクトリ構成が、NFS で提供されるディレクトリと同じかどうかを確認します。

```
# clush -w n0131 ". $HOME/.bash_profile; hadoop fs -ls /"
n0131: Found 7 items
n0131: drwxr-xr-x   - mapr mapr          0 2018-02-14 09:59 /apps
n0131: drwxr-xr-x   - mapr mapr          0 2018-02-14 10:01 /opt
n0131: drwxr-xr-x   - mapr mapr          5 2018-02-14 12:11 /test01
n0131: -rw-r--r--   3 root root         11 2018-02-19 00:19 /testfile.txt
n0131: drwxrwxrwx   - mapr mapr          0 2018-02-14 09:54 /tmp
n0131: drwxr-xr-x   - mapr mapr          1 2018-02-15 02:47 /user
n0131: drwxr-xr-x   - mapr mapr          1 2018-02-14 09:59 /var
```

MapR-FS 上でファイル testfile.txt の中身が見るかどうかを確認します。

```
# clush -w n0131 ". $HOME/.bash_profile; hadoop fs -cat /testfile.txt"
n0131: Hello MapR
```

MapR-FS 上のファイルが正常に見ましたので、NFS マウントを解除しておきます。

```
# umount /mnt
```

■ NFS 用の仮想 IP アドレスの設定

以下では、NFS 用の仮想 IP アドレスの設定手順を述べます。まず、全ノードの NIC の MAC アドレス一覧を取得します。仮想 IP アドレスの設定では、データ用 IP アドレスを割り当てた NIC の MAC アドレスを使用します。

```
# clush -aL "ip addr show | grep -B 1 10.0.0. | grep ether"
n0131:     link/ether 52:54:00:e0:f8:0a brd ff:ff:ff:ff:ff:ff
n0132:     link/ether 52:54:00:9a:fd:33 brd ff:ff:ff:ff:ff:ff
n0133:     link/ether 52:54:00:cd:42:cd brd ff:ff:ff:ff:ff:ff
```

得られた MAC アドレスの NIC に対して、仮想 IP アドレスのプールを割り当てます。今回は、3 ノードのクラスターに対して、仮想 IP アドレスを 5 つ（10.0.0.250/24 から 10.0.0.254/24）割り当てます。

```
# clush -w n0131 \
"maprcli virtualip add \
-virtualip 10.0.0.250 \
-virtualipend 10.0.0.254 \
-netmask 255.255.255.0 \
-macs 52:54:00:e0:f8:0a 52:54:00:9a:fd:33 52:54:00:cd:42:cd"
```

割り当てた仮想 IP アドレスの一覧を表示します。

```
# clush -w n0131 "maprcli virtualip list"
hn     ip          vip         mac                AssignableTo
n0132  10.0.0.132  10.0.0.250  52:54:00:9a:fd:33  52:54:00:9a:fd:33, ...
n0131  10.0.0.131  10.0.0.251  52:54:00:e0:f8:0a  52:54:00:9a:fd:33, ...
n0133  10.0.0.133  10.0.0.252  52:54:00:cd:42:cd  52:54:00:9a:fd:33, ...
n0132  10.0.0.132  10.0.0.253  52:54:00:9a:fd:33  52:54:00:9a:fd:33, ...
n0131  10.0.0.131  10.0.0.254  52:54:00:e0:f8:0a  52:54:00:9a:fd:33, ...
```

各ノードの NIC† に仮想 IP アドレスが割り当てられているかを確認します。

```
# clush -aL "ip -4 addr show dev ens9 | grep inet"
n0131:     inet 10.0.0.131/24 brd 10.0.0.255 scope global ens9
n0131:     inet 10.0.0.250/24 scope global secondary ens9:~m0
n0131:     inet 10.0.0.253/24 scope global secondary ens9:~m1
n0132:     inet 10.0.0.132/24 brd 10.0.0.255 scope global ens9
n0132:     inet 10.0.0.251/24 scope global secondary ens9:~m0
n0132:     inet 10.0.0.254/24 scope global secondary ens9:~m1
n0133:     inet 10.0.0.133/24 brd 10.0.0.255 scope global ens9
n0133:     inet 10.0.0.252/24 scope global secondary ens9:~m0
```

各ノードの NIC に仮想 IP アドレスが割り当てられていることがわかります。上記の場合、たとえば、n0131.jpn.linux.hpe.com には、10.0.0.250/24 と 10.0.0.253/24 の仮想 IP アドレスが割り当てられていることがわかります。

> † 例では、NIC のインターフェイス名が、ens9 になっていますが、ハードウェアに搭載されている NIC の種類や CentOS 側の設定によりインターフェイス名は異なります。

MCS の管理画面でも仮想 IP アドレスを確認しておきます。管理画面上部の［Services］をクリックします。すると、サービス一覧が表示されます (図 4-12)。

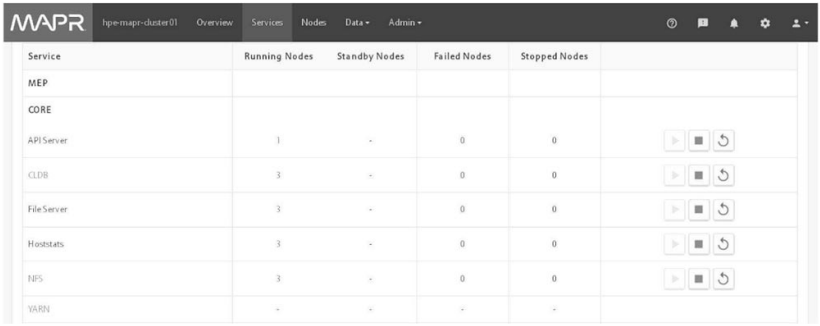


図 4-12 MapR クラスターで稼働しているサービス一覧を表示

画面左側の「Service」列に表示されている［NFS］をクリックすると、仮想 IP アドレスの管理画面になります (図 4-13)。

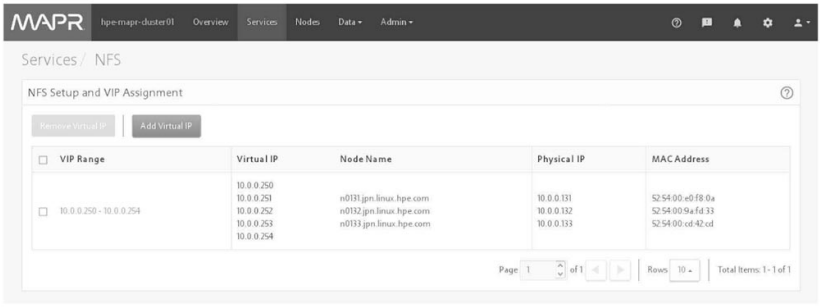


図 4-13 MapR クラスターで設定されている NFS 用の仮想 IP アドレスの管理画面

■ 仮想 IP アドレスを使った NFS マウント

クライアントノードから、仮想 IP アドレスを使って NFS マウントできるかを確認します。

```
# mount -o hard,nolock 10.0.0.250:/mapr /mnt
# ls -F /mnt/hpe-mapr-cluster01/
apps/  hbase/  opt/  test01/  testfile.txt  tmp/  user/  var/
```

■ 擬似障害による仮想 IP アドレスの引き継ぎテスト

仮想 IP アドレスの 10.0.0.250/24 が割り当てられている n0131.jpn.linux.hpe.com に障害が発生したことを想定し、n0131.jpn.linux.hpe.com を再起動します。

```
# clush -w n0131 reboot
```

仮想 IP アドレスの 10.0.0.250/24 が他のノードに割り当てられているかを確認します。

```
# clush -w n0132,n0133 "ip -4 addr show dev ens9 | grep inet"
n0132:     inet 10.0.0.132/24 brd 10.0.0.255 scope global ens9
n0132:     inet 10.0.0.251/24 scope global secondary ens9:~m0
n0132:     inet 10.0.0.254/24 scope global secondary ens9:~m1
n0132:     inet 10.0.0.253/24 scope global secondary ens9:~m2
n0133:     inet 10.0.0.133/24 brd 10.0.0.255 scope global ens9
n0133:     inet 10.0.0.252/24 scope global secondary ens9:~m0
n0133:     inet 10.0.0.250/24 scope global secondary ens9:~m1
```

上記の実行結果では、仮想 IP アドレスの 10.0.0.250/24 が、n0133 に引き継がれていることがわかります。引き続き、クライアントノードから、NFS マウントされた/mnt ディレクトリにアクセスできるかを確認します。

```
# df -HT | grep /mnt
10.0.0.250:/mapr nfs       108G     0  108G   0% /mnt

# ls -F /mnt/hpe-mapr-cluster01/
apps/  hbase/  opt/  test01/  testfile.txt  tmp/  user/  var/

# echo "Hello MapR." > /mnt/hpe-mapr-cluster01/testfile2.txt
# cat /mnt/hpe-mapr-cluster01/testfile2.txt
Hello MapR.

# su - mapr
```

```
$ hadoop fs -cat /testfile2.txt
...
Hello MapR.
```

以上で、MapR が提供する NFS の仮想 IP アドレス機能により、ノード障害時でも NFS サービスを継続して利用することができました。

Column　NFS サービスのメリット

古いバージョンの Apache Hadoop では、NFS サーバーの機能を提供していなかったため、クライアントノードから大量のファイルを Hadoop クラスターにコピーするといった操作を行う場合、ユーザーは、自身のスクリプトやアプリケーション内に hadoop コマンドを組み込まなければなりませんでした。しかし、NFS サーバーの機能を持つ Hadoop ディストリビューションの MapR が登場し、クライアントノード側から Linux OS 標準のコマンドを使って MapR クラスターの分散ファイルシステムにアクセス可能となり、データの投入が非常に容易になりました。

これは、実際に操作を行うと非常に良く実感できます。例として、テキストファイル群に含まれる単語の出現頻度を集計する「ワードカウント」を利用してファイルの操作を行ってみましょう。

このワードカウントの動作テストでは、通常、大量のテキストファイルを Hadoop クラスターにコピーします。NFS サービスの機能を持つ Hadoop クラスターであれば、クライアントノードから通常の `ls` コマンド、`mkdir` コマンド、`cp` コマンドなどの Linux OS 標準のコマンドでワードカウントのテストを実行できます。以下は、`hadoop` コマンドを一切使わず、クライアントノード上の`/usr/share/doc` ディレクトリ以下にあるファイル群において、拡張子が「`.txt`」のテキストファイルを MapR-FS 上の `testdir` ディレクトリにコピーし、ワードカウントを実行するワークフローです。

```
# hostname
n0130.jpn.linux.hpe.com

# mount -o hard,nolock 10.0.0.250:/mapr /mnt

# su - mapr
$ mkdir /mnt/hpe-mapr-cluster01/testdir
$ find /usr/share/doc/ -name *.txt |xargs -I{} cp -a {} \
/mnt/hpe-mapr-cluster01/testdir

$ ls -l /mnt/hpe-mapr-cluster01/testdir/
total 11289
-rw-rw-r-- 1 root root   4476 Dec 15  2005 16colors.txt
-rw-r--r-- 1 root root   4492 May 18  2010 30chg.txt
...
```

```
$ yarn jar \
/opt/mapr/hadoop/hadoop-2.7.0/share/hadoop/mapreduce\
/hadoop-mapreduce-examples-2.7.0-mapr-1803.jar wordcount \
/testdir /outputdir
...

$ ls -l /mnt/hpe-mapr-cluster01/outputdir/
total 929
-rwxr-xr-x 1 mapr mapr 950350 Apr 17 02:05 part-r-00000
-rwxr-xr-x 1 mapr mapr      0 Apr 17 02:05 _SUCCESS

$ cat /mnt/hpe-mapr-cluster01/outputdir/part-r-00000 |\
sort -k2 -rn | head -5
the     36855
to      19280
a       14304
of      12385
is      11648
```

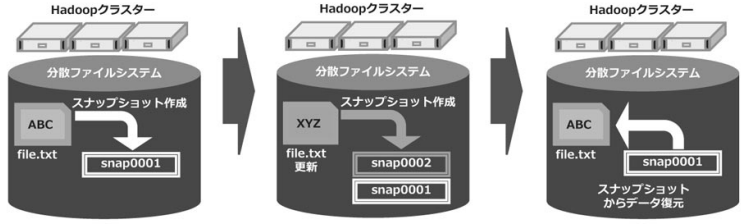
hadoop コマンドを使うことなく、find コマンドと cp コマンドを使って NFS マウントしたディレクトリにファイル群を大量にコピーしています。また、ワードカウントのアプリケーションの実行結果（テキストファイル群に含まれる、単語の出現頻度のベスト 5 を表示）も、hadoop コマンドを使うことなく、Linux OS 標準の ls コマンド、cat コマンド、sort コマンド、head コマンドを使って表示できています。

近年は、MapR クラスターの NFS サーバー機能に代表されるように、Hadoop クラスターをスケールアウト型の NAS（Network Attached Storage）して利用する運用も多く見られます。

4-2-2　スナップショットによるファイルの復元

MapR クラスターでは、エンタープライズ用途として欠かせないスナップショット機能が標準で搭載されています (図 4-14)。オペレーションミスなどによりファイルが消失した場合でも、事前に取得していたスナップショットから過去の状態に戻すことでファイルを復元できるため、Hadoop クラスターにおいてもスナップショットが広く利用されています。

以下では、MapR クラスターにおけるスナップショットの作成、および、スナップショットによるファイルの復元手順を紹介します。作業は、クラスターノードの n0131 で行います。



- **スナップショットを取得した過去の時点に戻ることが可能**
- **複数の過去の時点に切り替えて戻すことが可能**

活用例)
誤操作によるファイル消失等の事故発生時のファイルの復元

図 4-14　Hadoop におけるスナップショット機能

■ ボリュームの作成

まず、MapR-FS 上にボリュームを作成します。ボリュームは、MapR 独自のストレージにおける概念です。MapR-FS におけるボリュームには、名前を付与し、ボリュームに対応する MapR-FS 上のディレクトリを指定します。以下は、MapR-FS 上の/sales01 ディレクトリに関するボリューム「sales01」を作成する例です。

```
# hostname
n0131.jpn.linux.hpe.com

# su - mapr
$ whoami
mapr

$ maprcli volume create -name sales01 -mount 1 -path /sales01
```

作成したボリューム「sales01」の MapR-FS 上のディレクトリを確認します。

```
$ hadoop fs -ls / | grep /sales01
drwxr-xr-x   - mapr mapr          0 2018-04-19 03:30 /sales01
```

MapR-FS にアクセスできる NFS マウントポイント/mapr ディレクトリ†がマウントされていることを確認します。

```
$ df -HT | grep /mapr
```

```
n0131.jpn.linux.hpe.com:/mapr nfs       108G     0  108G   0% /mapr

$ ls -ld /mapr/hpe-mapr-cluster01/sales01
drwxr-xr-x 2 mapr mapr 0 Apr 19 03:30 /mapr/hpe-mapr-cluster01/sales01
```

/mapr/hpe-mapr-cluster01/sales01 ディレクトリは、MapR-FS 上の/sales01 に対応しています。

† /mapr ディレクトリに NFS マウントされていない場合は、/opt/mapr/conf/mapr_fstab ファイルの設定を再確認してください。

■ テスト用のファイルの作成

テスト用のファイル「test.txt」を MapR-FS 上の/sales01 ディレクトリ内に作成します。ファイルの内容は、「Hello Hadoop」という文字列です。

```
$ echo "Hello Hadoop" > /mapr/hpe-mapr-cluster01/sales01/test.txt
$ cat /mapr/hpe-mapr-cluster01/sales01/test.txt
Hello Hadoop
```

■ スナップショットの作成

MapR において、スナップショットは、ボリュームに対して作成します。スナップショットの作成は、「maprcli volume snapshot create」コマンドに「-volume」オプションを指定し、その後にボリューム名を付与します。さらに、「-snapshotname」オプションを指定し、作成するスナップショット名を付与します。以下は、ボリューム sales01 に対して、スナップショット snap0001 を作成する例です。

```
$ maprcli volume snapshot create -volume sales01 -snapshotname snap0001
```

作成したスナップショットを確認します。

```
$ maprcli volume snapshot list
snapshotid  ... snapshotname  ... volumepath  volumeSnapshotAces
256000049   ... snap0001      ... /sales01    ...
```

テスト用のファイル「test.txt」に内容を追記し、2 つ目のスナップショット snap0002 を取得し

ます。

```
$ echo "Hello MapR" >> /mapr/hpe-mapr-cluster01/sales01/test.txt
$ cat /mapr/hpe-mapr-cluster01/sales01/test.txt
Hello Hadoop
Hello MapR

$ maprcli volume snapshot create -volume sales01 -snapshotname snap0002
$ maprcli volume snapshot list
snapshotid  ... snapshotname  ... volumepath  volumeSnapshotAces
256000049   ... snap0001      ... /sales01    ...
256000050   ... snap0002      ... /sales01    ...
```

■「.snapshot」ディレクトリの確認

スナップショットを作成したボリュームは、「ls -la」コマンドでは見ることができない特殊な「.snapshot」ディレクトリが作成されます。

```
$ ls -al /mapr/hpe-mapr-cluster01/sales01/
total 2
drwxr-xr-x  2 mapr mapr  1 Apr 17 03:45 .
drwxr-xr-x 11 mapr mapr 10 Apr 17 03:30 ..
-rw-rw-r--  1 mapr mapr 24 Apr 17 03:50 test.txt
```

MapR-FS 上の/sales01 ディレクトリには、test.txt しかありませんが、スナップショットの情報が格納された「.snapshot」ディレクトリが作成されています。

```
$ ls -al /mapr/hpe-mapr-cluster01/sales01/.snapshot
total 2
drwxr-xr-x 4 root root 2 Apr 17 03:50 .
drwxr-xr-x 2 mapr mapr 1 Apr 17 03:45 ..
drwxr-xr-x 2 mapr mapr 1 Apr 17 03:45 snap0001
drwxr-xr-x 2 mapr mapr 1 Apr 17 03:45 snap0002
```

■ スナップショットによるファイルの復元

ボリュームのスナップショットを取得したので、スナップショットから過去のファイル内容を復元できます。先ほどの test.txt ファイルの中身を上書きします。

```
$ echo "Hello World" > /mapr/hpe-mapr-cluster01/sales01/test.txt
$ cat /mapr/hpe-mapr-cluster01/sales01/test.txt
Hello World
```

スナップショットで過去の状態に戻します。まずは、スナップショット snap0001 を使ってファイルを復元します。「.snapshot」ディレクトリに存在するスナップショット snap0001 と同じ名前のディレクトリ以下にある test.txt ファイルが復元したいファイルです。よって、「.snapshot/snap0001/test.xt」を MapR-FS 上の/sales01 ディレクトリにコピー（復元）します。

```
$ cd /mapr/hpe-mapr-cluster01/sales01
$ cp -a .snapshot/snap0001/test.txt .
$ cat test.txt
Hello Hadoop
```

同様に、スナップショット snap0002 を使って、ファイルを復元します。

```
$ pwd
/mapr/hpe-mapr-cluster01/sales01

$ cp -a .snapshot/snap0002/test.txt .
$ cat test.txt
Hello Hadoop
Hello MapR
```

スナップショット snap0001 と snap0002 から、過去におけるそれぞれのスナップショット取得時のファイルに復元できました。

■ スナップショットの削除

スナップショットの削除は、「maprcli volume snapshot remove」コマンドで行います。

```
$ maprcli volume snapshot remove -volume sales01 -snapshotname snap0002
```

Note　.snapshot ディレクトリの削除

スナップショットが保管されている「.snapshot」ディレクトリは、rm コマンドで削除できません。

```
$ cd /mapr/hpe-mapr-cluster01/sales01/
$ rm -rf .snapshot
rm: cannot remove ‘.snapshot/snap0002/test.txt’: Read-only file system
rm: cannot remove ‘.snapshot/snap0001/test.txt’: Read-only file system
```

4-2-3　MapR 6.0 における管理コマンド例

MapR 6.0 では、管理者向けのコマンドやユーザーが利用するコマンドなど、非常に多くの管理コマンドが用意されています。以下では、運用管理で利用される、主なコマンドの使用例をまとめておきます。maprcli コマンドは、クラスターノードのいずれかで実行すると仮定します。

例）MapR のバージョンを表示

```
$ maprcli dashboard info -version true
version
6.0.1.20180404222005.GA
```

例）CLDB ノードをリストアップ

```
$ maprcli node listcldbs
CLDBs
n0132.jpn.linux.hpe.com,n0131.jpn.linux.hpe.com,n0133.jpn.linux.hpe.com
```

例）ZooKeeper ノードをリストアップ

```
$ maprcli node listzookeepers
Zookeepers
n0131.jpn.linux.hpe.com:5181,n0132.jpn.linux.hpe.com:5181,n0133.jpn.linux.hpe.com:5181
```

例）クラスターノードのリストアップと死活状態の表示（health 列に 0 が表示されると正常と判断、2 が表示されると障害と判断）

```
$ maprcli node list -columns health
hostname                ip                          health
n0131.jpn.linux.hpe.com  10.0.0.131,172.16.1.131  0
n0132.jpn.linux.hpe.com  10.0.0.132,172.16.1.132  2
n0133.jpn.linux.hpe.com  10.0.0.133,172.16.1.133  0
```

例）クラスターノードに登録されているデータ用ディスクのリストアップ

```
$ /opt/mapr/server/mrconfig disk list
ListDisks resp: status 0 count=3
ListDisks /dev/sdb
size 3072000MB
        DG 0: Single SingleDisk1 Online
        DG 1: Raid0 Stripe1-2 Online
        DG 2: Concat Concat1-3 Online
        SP 0: name SP1, Online, size 8842446 MB, free 8841320 MB, path /dev/sdb
ListDisks /dev/sdc
...
```

例）アドバイザリークォータが 500MB†、ハードクォータが 1GB のボリューム vol01（マウントポイントは、MapR-FS の/vol01 ディレクトリ）の作成

```
$ maprcli volume create -name vol01 -path /vol01 -quota 1G -advisoryquota 500M
$ ls -ld /mapr/hpe-mapr-cluster01/vol01
drwxr-xr-x 2 mapr mapr 0 Apr 19 05:31 /mapr/hpe-mapr-cluster01/vol01
```

† 500MB に到達するとアラームが発生しますが、データは書き込み可能です。

例）ボリューム vol01（マウントポイントは、MapR-FS の/vol01 ディレクトリ）のアンマウント。

```
$ maprcli volume unmount -name vol01
$ ls -ld /mapr/hpe-mapr-cluster01/vol01
ls: cannot access /mapr/hpe-mapr-cluster01/vol01: No such file or directory
```

例）ボリューム vol01（マウントポイントは、MapR-FS の/vol01 ディレクトリ）のマウント

```
$ maprcli volume mount -name vol01
```

例）ボリューム vol01 の削除

```
$ maprcli volume remove -name vol01
$ ls -ld /mapr/hpe-mapr-cluster01/vol01
ls: cannot access /mapr/hpe-mapr-cluster01/vol01: No such file or directory
```

例）ボリューム名と対応するディレクトリパスの一覧表示

```
$ maprcli volume list -columns volumename,mountdir
volumename                                  mountdir
...
sales01                                     /sales01
test01                                      /test01
...
```

例）クラスターノード n0131 のサービスの一覧と状態を表示（稼働しているサービスは、state 列が 2 で表示される）

```
$ maprcli service list -node n0131
logpath                                displayname       name              ...  state
/opt/mapr/logs/mfs.log                 FileServer        fileserver        ...  2
/opt/mapr/hadoop/hadoop-2.7.0/logs     JobHistoryServer  historyserver     ...  2
/opt/mapr/hadoop/hadoop-2.7.0/logs     ResourceManager   resourcemanager   ...  2
/opt/mapr/logs/cldb.log                CLDB              cldb              ...  2
/opt/mapr/logs/nfsserver.log           NFS Gateway       nfs               ...  2
/opt/mapr/hadoop/hadoop-2.7.0/logs     NodeManager       nodemanager       ...  2
/opt/mapr/logs/gateway.log             GatewayService    gateway           ...  2
/opt/mapr/logs/hoststats.log           HostStats         hoststats         ...  2
/opt/mapr/apiserver/logs/apiserver.log APIServer         apiserver         ...  0
```

例）トポロジの表示

```
$ maprcli node topo
path
/
/data
/data/default-rack
/default-rac
```

例）トポロジに所属するノードをリストアップ

```
$ maprcli node list -json | grep topo
                        "racktopo":"/data/default-rack/n0131.jpn.linux.hpe.com",
                        "racktopo":"/data/default-rack/n0132.jpn.linux.hpe.com",
```

```
                                "racktopo":"/data/default-rack/n0133.jpn.linux.hpe.com",
```

例）ユーザーごとの ACL（Access Control List）の表示（表 4-5）

```
$ maprcli acl show -type cluster
Allowed actions         Principal
[login, ss, cv, a, fc]  User mapr
[login, ss, cv, a, fc]  User root
```

表 4-5　クラスターのアクセス制御の種類（クラスターパーミッション）

アクセス制御の種類	意味
login	MCS、API、CLI の利用、クラスタとボリュームの読み取りが可能
ss	サービスの起動、停止が可能
cv	ボリュームの作成が可能
a	ACL の設定、編集が可能
fc	全制御が可能（ただし、ACL の編集を除く）

例）ユーザー koga† に MapR クラスターのアクセス制御として「login」を付与

```
$ maprcli acl edit -type cluster -user koga:login
$ maprcli acl show -type cluster
Allowed actions      Principal
[login, ss, cv, fc]  User mapr
[login, ss, cv, fc]  User root
[login]              User koga
```

† ユーザーは、useradd コマンドを使って、全ノードに同じ UID、同じ GID で事前に作成しておく必要があります。

例）ボリューム「sales01」に設定されている ACL の表示（表 4-6）

```
$ maprcli acl show -type volume -name sales01
Allowed actions               Principal
[dump, restore, m, a, d, fc]  User mapr
```

表 4-6　ボリュームのアクセス制御の種類（ボリュームパーミッション）

アクセス制御の種類	意味
dump	ボリュームのダンプ、およびバックアップが可能
restore	ボリュームのミラー、およびリストアが可能
m	ボリュームのプロパティの変更、スナップショットの作成、削除が可能
a	ACL の設定、編集が可能
d	ボリュームの削除が可能
fc	全制御が可能（ただし、ACL の編集を除く）

例）ユーザー koga にボリューム「sales01」のアクセス制御として「fc」を付与

```
$ maprcli acl edit -type volume -name sales01 -user koga:fc
$ maprcli acl show -type volume -name sales01
Allowed actions              Principal
[dump, restore, m, a, d, fc]  User mapr
[dump, restore, m, d, fc]     User koga
```

例）ノード n0132 をメンテナンスモードへ移行し、レプリケーションは、60 分後に再開する

```
# maprcli node maintenance -nodes n0132 -timeoutminutes 60
# ssh -l root n0132-mgm "systemctl stop mapr-warden"
# ssh -l root n0132-mgm "systemctl stop mapr-zookeeper"
```

例）ノード n0132 をメンテナンスモードから正常モードへ移行する

```
# maprcli node maintenance -nodes n0132 -timeoutminutes 0
# ssh -l root n0132-mgm "systemctl restart mapr-warden"
```

例）CLDB サービスが提供する、Web 管理画面の URL の表示

```
$ maprcli urls -name cldb | grep http
http://n0131.jpn.linux.hpe.com:7221/cldb.jsp
```

例）ResourceManager サービスが提供する、Web 管理画面の URL の表示

```
$ maprcli urls -name resourcemanager | grep http
http://n0133.jpn.linux.hpe.com:8088
```

例）NodeManager サービスが提供する、Web 管理画面の URL の表示

```
$ maprcli urls -name nodemanager | grep http
http://n0131.jpn.linux.hpe.com:8042/
```

例）JobHistoryServer サービスが提供する、Web 管理画面の URL の表示

```
$ maprcli urls -name historyserver | grep http
http://n0131.jpn.linux.hpe.com:19888/jobhistory
```

4-2-4　MapR Control System（MCS）によるクラスター管理

ユーザーは、MapR Control System（以下、MCS）を使って MapR クラスターの管理を行います。MCS では、クラスター全体の CPU 利用率、メモリ利用率、ディスク利用率や、サービスの状態などのメトリックをリアルタイムに監視できます。また、MCS では、過去の CPU、メモリ、ディスクの利用率を時系列で表示させることも可能です。この時系列表示は、MapR が提供する「MapR モニタリング」と呼ばれる一連のメトリック監視用パッケージで提供されており、MCS の管理画面だけでなく、Grafana と呼ばれる可視化ソフトウェアの管理画面にシステム負荷の履歴を表示できます。以下は、MCS に MapR モニタリングを組み込む手順、MCS、Grafana の利用法について説明します。

■ MapR モニタリングのインストール

MapR クラスターの管理は、MapR モニタリングがなくても MCS で可能ですが、今回は、CPU、メモリ、ディスクの利用率を時系列表示する機能を MCS に組み込みます。また、可視化ソフトウェアの Grafana もインストールします。まず、MapR モニタリングのパッケージ† をインストールします。

> † 可視化ソフトウェアの Grafana は、通常、情報をデータベースに蓄えます。Grafana は、さまざまな種類のデータベースをサポートしていますが、MapR クラスターの場合、Grafana が利用するデータベースは、OpenTSDB です。MapR クラスターの Grafana による可視化では、MapR 社が提供している mapr-grafana パッケージと mapr-opentsdb を使用します。

```
# hostname
n0130.jpn.linux.hpe.com
```

```
# clush -a "yum install -y mapr-collectd"
# clush -w n0132 "yum install -y mapr-grafana mapr-opentsdb"
```

Note　パッケージインストール上の注意

メトリック監視の mapr-collectd パッケージは、全クラスターノードにインストールしますが、mapr-grafana と mapr-opentsdb は、クラスターノードのうち、少なくとも 1 台にインストールする必要があります。これらのパッケージのインストールを行うには、MEP のリポジトリ（本書では、/etc/yum.repos.d/maprtech.repo ファイル内の記述）を全ノードで事前に設定しておく必要があります。

configure.sh スクリプトを使ってクラスターを再構成します。configure.sh スクリプトに「-OT」オプションを付与し、mapr-opentsdb パッケージをインストールしたノードを指定します。今回は、mapr-opentsdb をクラスターノードの n0132 にインストールしたので、以下のように実行します。

```
# clush -a "/opt/mapr/server/configure.sh -R -OT n0132"
```

apiserver サービスを再起動します。

```
# clush -g web \
'maprcli node services -name apiserver -nodes `hostname -f` -action restart'
```

以上で、MCS の管理画面で、MapR モニタリングにより、クラスターノードの負荷状況を時系列で表示させることができるようになります。

■ MCS によるクラスターノードの負荷状況の確認

実際に、MCS にログインし、クラスターノードに負荷状況を時系列表示できるかを確認します。MCS にログインしている場合は、いったんログアウトし、再度、MCS の管理画面にログインします。

```
$ firefox https://n0133-mgm.jpn.linux.hpe.com:8443 &
```

MCS の管理画面左側の「Node Health」に表示されている「collectd」「grafana」「opentsdb」が表示

され、その右側に表示されている四角が黄色や赤色になっていないことを確認します（図 4-15）。

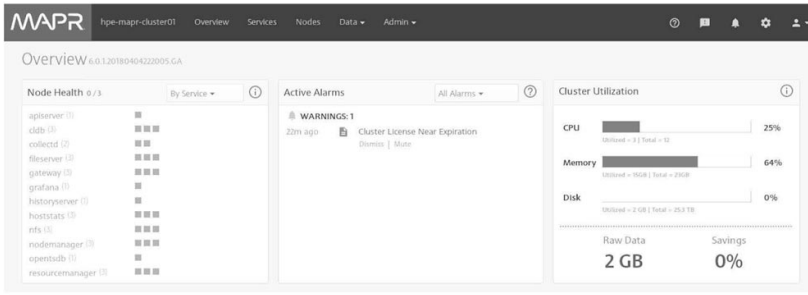


図 4-15　MCS の管理画面で collectd と grafana と opentsdb サービスを確認

MCS の管理画面を下にスクロールさせ、画面右側にある［Last 24 Hours］のプルダウンメニューをクリックし、「Last 15 Minutes」を選択します（図 4-16）。

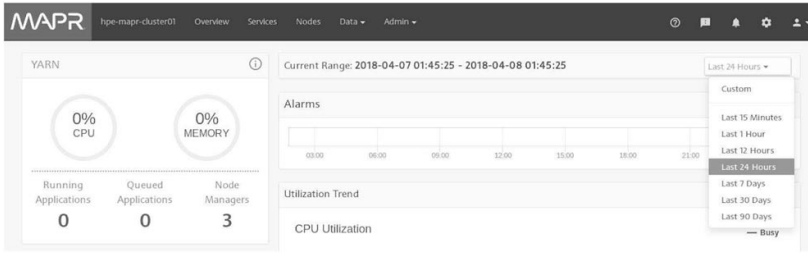


図 4-16　MCS の管理画面の「Last 24 Hours」のプルダウンメニューをクリックし、「Last 15 Minutes」を選択

画面をさらに下にスクロールさせると、CPU、ディスク、メモリの利用率を時系列で確認できます†（図 4-17）。

† MCS の画面で、CPU、ディスク、メモリの利用率のグラフが表示されない場合は、Web ブラウザを開いているクライアントノードが MapR クラスターのデータ用 LAN（10.0.0.0/24）と管理用 LAN（172.16.0.0/16）の両方にアクセスできているかを確認してください。

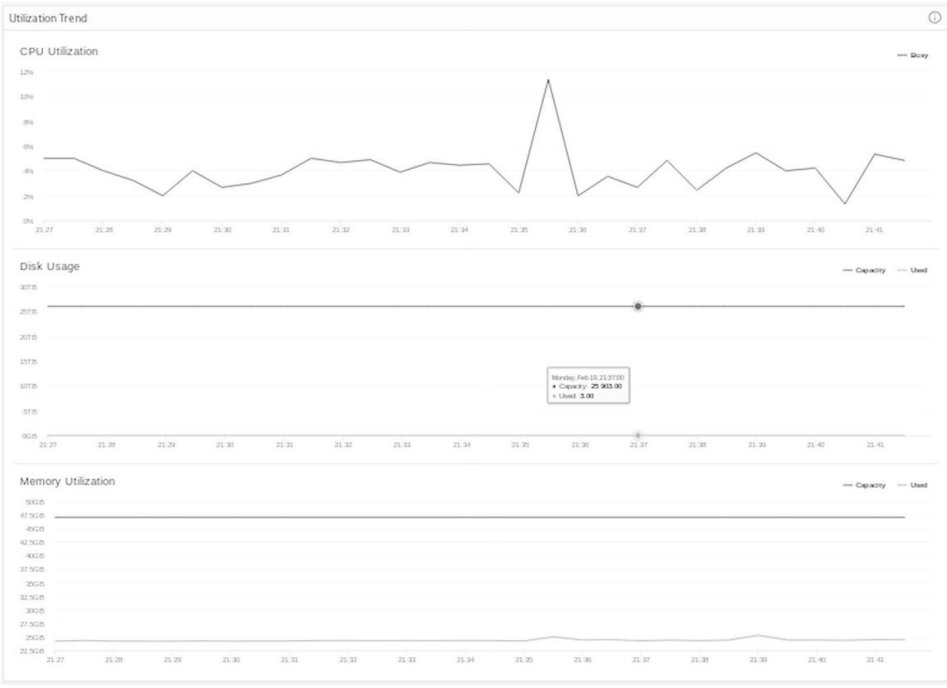


図 4-17　CPU、ディスク、メモリの利用率を時系列で表示

さらに、MCS の管理画面上部の［Nodes］をクリックし、「Hostname」列に表示されているクラスターノードの 1 つをクリックします（図 4-18）。

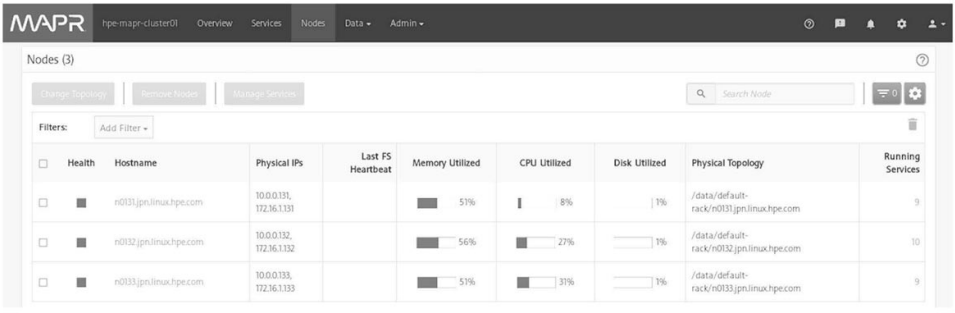


図 4-18　MCS の管理画面上部の「Nodes」をクリックし、「Hostname」列に表示されているクラスターノードの 1 つをクリック

クラスターノードの管理画面が表示されるので、［Metrics］タブをクリックし、画面右側にあ

る［Last 24 Hours］のプルダウンメニューをクリックし、「Last 15 Minutes」を選択します。MapR モニタリングによるクラスターノードに関するさまざまな負荷情報が表示されます（図 4-19）。

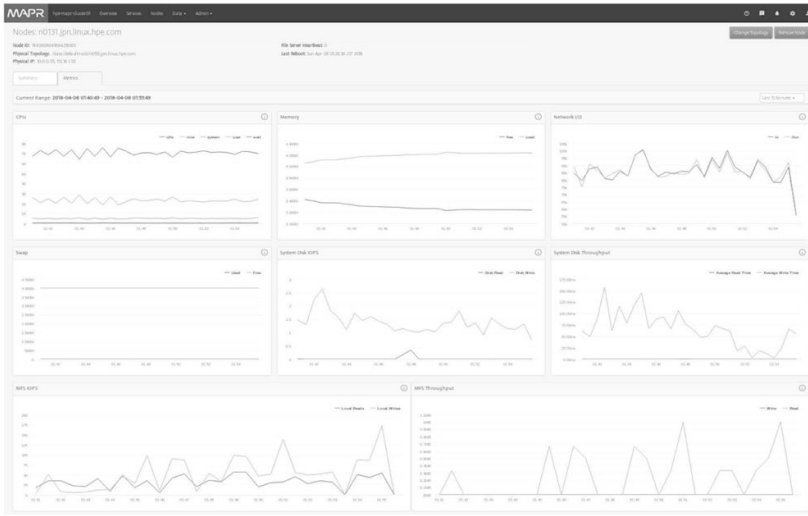


図 4-19　クラスターノードの負荷状況を表示

以下では、MCS が提供するその他の管理画面をいくつか紹介します。

■ サービスの管理

MCS の管理画面上部の［Services］をクリックすると、MapR クラスターで稼働しているサービスの管理画面が表示されます。サービスが稼働しているノード数の表示に加え、サービスの起動、停止、再起動の操作も可能です（図 4-20）。

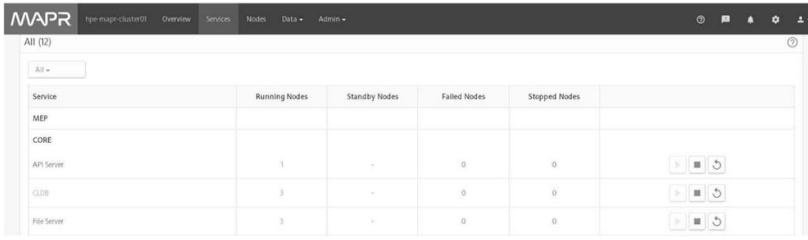


図 4-20　サービスの管理画面

■ ボリュームの管理

MCS の管理画面上部の［Data］をクリックし、プルダウンメニューで表示される［Volumes］をクリックすると、MapR クラスターのボリューム管理画面が表示されます。画面左上の［Summary］タブをクリックすると画面下部に現在のボリュームの状況が表示されます。図 4-21 では、sales01、test01、users ボリュームが存在することがわかります。

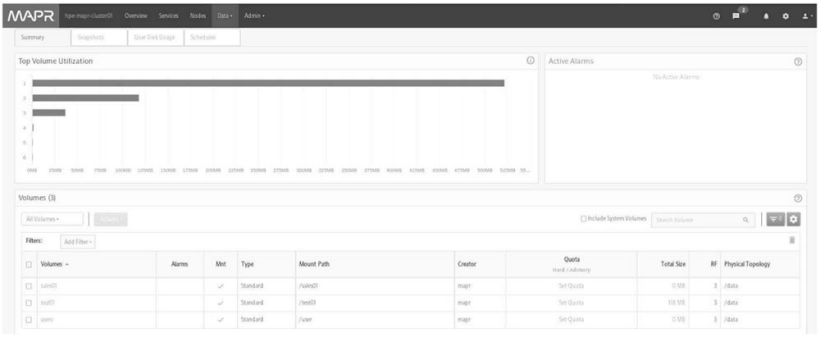


図 4-21　ボリュームの管理画面

■ ユーザーの管理

MCS の管理画面上部の［Admin］をクリックし、プルダウンメニューで表示される［User Settings］をクリックすると、MapR クラスターのユーザー管理画面が表示されます (図 4-22)。

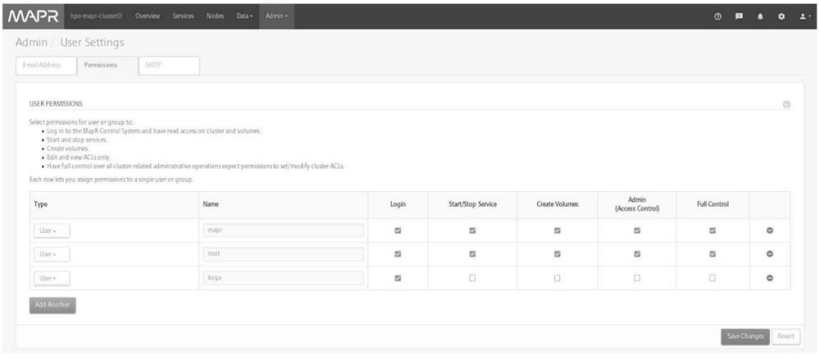


図 4-22　ユーザーの管理画面

画面左上の［Permissions］タブをクリックすると画面下部に現在のユーザーに付与されている権限が表示されます。ここでユーザーやグループに対する ACL を設定することができます。

■ Grafana による負荷の監視

すでに MapR モニタリングを MCS に組み込み済みですので、MCS から Grafana の管理画面のリンクを辿ることが可能です。MCS の管理画面上部の［Services］をクリックし、サービスの管理画面が表示されたら、左側に表示されている［Grafana］のリンクをクリックします (図 4-23)。

図 4-23　MCS のサービス管理画面の「Ganglia」のリンクをクリック

Grafana のログイン画面が表示されるので、「User」欄に admin、「Password」欄に admin を入力し、［Login］ボタンをクリックして Grafana の管理画面にログインします (図 4-24)。

図 4-24　Grafana のログイン画面

Grafana にログインすると、左上にある「Home」をクリックします (図 4-25)。

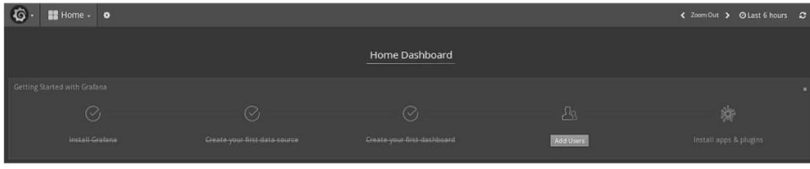


図 4-25　Grafana にログイン直後の画面

プルダウンメニューが表示され、ダッシュボードへのリンクが用意されています (図 4-26)。［CLDB Dashboard］をクリックすると、MapR クラスターノードの稼働ノード数、利用可能な

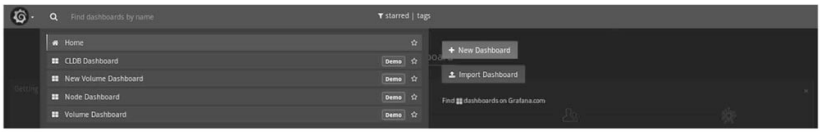


図 4-26　ダッシュボードへのリンクが表示される

ディスク容量、負荷状況などが確認できます (図 4-27)。

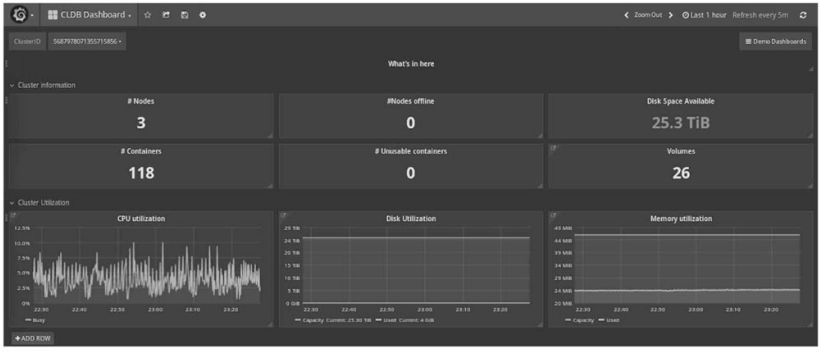


図 4-27　CLDB Dashboard の画面

「Node Dashboard」をクリックすると、CPU やメモリ利用率に加え、ディスクのスループット、IOPS、ネットワーク I/O などが確認できます (図 4-28)。

図 4-28　Node Dashboard の画面

4-3　まとめ

本章では、Apache Hadoop 3 クラスターと MapR 6.0 クラスターの基本的な運用管理について紹介しました。Hadoop のスケールアウトメリットを活かした高速分析基盤の恩恵を享受するには、日々の運用管理、障害監視、性能監視が欠かせません。コマンドラインによるクラスターの管理、監視、分散ファイルシステムのファイル管理、GUI 画面による操作、ログの解析など、マスターしなければならない項目が数多くありますが、まずは、本書に掲載した運用管理手順を実際に試して、Hadoop クラスターの初歩的な運用管理の手順をマスターしてください。

Column　MapR クラスターの高可用性

エンタープライズ向けの Hadoop 基盤では、クラスターの分散並列処理の性能管理だけでなく、サービスの高可用性（High Availability、通称、HA）の維持管理も求められます。

MapR クラスターでは、サービスの連続稼働を実現するための自動フェールオーバー機能を提供します。この機能を利用するには、そのサービスの冗長化が必要です。以下に、MapR クラスターにおける高可用性を実現するために必要なサービスの必要最低限の個数を示します。

サービス	最小個数	説明
CLDB	2	アクティブ、スタンバイ構成で最低 2 つ。一般に、3 台以上で構成
ZooKeeper	3	定足数の維持には、最低 3 台必要。奇数台で構成
NFS	2	仮想 IP アドレスの自動フェールオーバー。全ワーカーノードで構成
ResourceManager	2	1 つをアクティブで構成し、残りをスタンバイで構成

管理者は、本番稼働前に、自動フェールオーバーが行われるかどうかの障害試験を実施しなければなりません。また、本番稼働後も、サービスの死活監視業務において、サービスが適切に冗長化構成を維持しているかどうかの確認が必要です。さらに、MCS の Web 管理画面を使った目視による性能監視や障害監視だけでなく、管理者にメールが届く設定も必要です。Hadoop に限らず、高可用性クラスターにおいては、障害発生時の運用管理ノウハウが必要になる場合が多く、初心者にとって、決してハードルは低くありません。必要に応じて、Hadoop の導入経験が豊富なハードウェアベンダーやシステムインテグレーターとの情報交換が必要です。

【参考情報】

MapR クラスターにおける高可用性の計画：

`https://maprdocs.mapr.com/home/AdministratorGuide/c_high_availability_planning.html`

MapR クラスターにおける高可用性のベストプラクティス：

`https://maprdocs.mapr.com/home/AdministratorGuide/Best-Practices.html`

第５章

Spark－SQL、ストリーミング、グラフデータの処理、分類器の利用

高速なビッグデータ分析基盤を構築するには、サーバーの台数、CPU、メモリ容量といったハードウェア要件の検討は当然行わなければなりませんが、採用する分析ソフトウェアの検討も非常に重要です。

近年、高速にデータを処理する仕組みとして、インメモリ型のソフトウェアが注目されています。インメモリ型のソフトウェアは、データベースや分析ソフトウェアなどをサーバーのメモリ上に配置し、できるだけ外部記憶装置への I/O を減らすことで、従来に比べて非常に高速にデータを処理します。こうしたタイプのソフトウェアのうち、特に、分析を行うためのフレームワークを提供するソフトウェアの一つに Apache Spark があります。本章では、Hadoop 基盤のデータ処理エンジンの 1 つとして、Apache Spark の利用方法について取り上げます。

5-1 ビッグデータ処理の高速化

Hadoop では、大量の x86 サーバーの内蔵ディスクを使って分散ファイルシステムを構成し、クラスターのリソース管理とスケジューリングを行う YARN と MapReduce 呼ばれる並列処理の仕組みを使って、スケールアウトメリットを活かして高速なバッチ集計処理や分析を行うことができます。しかし、データをディスクに保存するため、ディスク I/O に多くの時間を費やす必要があり、これが処理全体のスループットに大きく影響します。たとえば、ニューラルネットワークによるシミュレーションや機械学習では、プログラムにおける「繰り返し処理」（Iteration）が多用されます。繰り返し処理を MapReduce で実行すると、反復処理ごとにディスクとの読み書きが発生し、ディスク I/O に多くの時間が費やされてしまいます。

こうした処理を高速化するには、分散ファイルシステムのディスク I/O をできるだけ減らし、CPU に近いメモリにデータを配置する必要があります。メモリ上にデータを配置するアーキテクチャは、インメモリ型と呼ばれ、近年では、処理速度の劇的な向上が見込めることから、データベースもインメモリ型で導入されることが増えてきています。その中でも、スケールアウト型の基盤において、分析を目的としたインメモリ型のソフトウェアとして有名なものに Apache Spark があります。

5-1-1 Apache Spark とは？

Apache Spark は、2009 年にカリフォルニア大学バークレー校の Algorithms, Machines and People Lab（通称 AMP Lab）において、ビッグデータ分析用のアプリケーション向けに開発されました。その後、Apache コミュニティに提供され、2010 年にオープンソース化されたことから、世界中のビッグデータ利用者に知られるようになりました。2018 年現在も、Spark は、Apache コミュニティによって精力的に改良が進められています。

■ Apache Spark のエコシステム

Apache Spark には、SQL クエリーエンジンである Spark SQL や、機械学習ライブラリ MLlib、リアルタイムに生成されるデータをストリーミング処理する Spark Streaming、大量のグラフデータを処理するフレームワーク GraphX、そして、統計解析に利用される R 言語から Spark を操作する SparkR を備えています。また、Spark には、Scala と呼ばれる言語が実装されており、対話型のプログラミングが可能なほか、Scala 以外にも、Python、Java にも対応しています（図 5-1）。

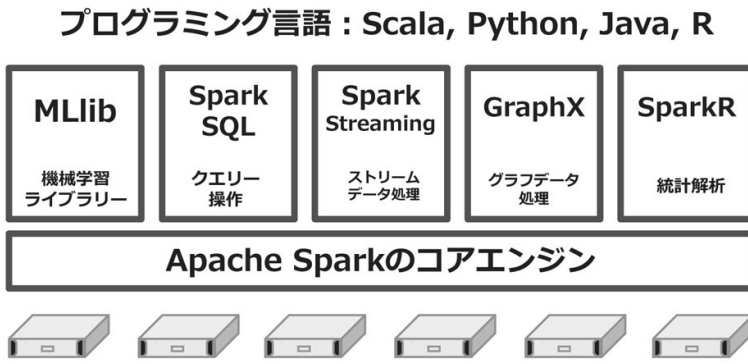


図 5-1　Apache Spark のエコシステム

■ Hadoop と Spark の関係

Hadoop において、データは Hadoop クラスターを構成する各ノードの内蔵ディスク上にブロックに分割されて記録・保管されるのが特徴です。一方、機械学習や深層学習のような繰り返し処理を得意とする Spark では、対象データをメモリ上に配置し、インメモリで分散処理を行います。データをメモリ上に配置し、インメモリでデータを処理するため、非常に高速な分析エンジンとして利用できます。Apache Spark プロジェクトの Web サイトでも紹介されていますが、ロジスティック回帰分析と呼ばれる処理では、Hadoop 上で MapReduce を使うのに比べ、Spark のインメモリ処理は格段に高速に処理できることが知られています。

Apache Spark プロジェクトの Web サイトの URL：
`https://spark.apache.org/`

Apache Spark 自体は単独で動作し、Hadoop が必須というものではありません。ですが、Hadoop で従来の MapReduce を使った分散並列処理のプログラムを実行するのと同様に、YARN 上でインメモリ処理のプログラムをジョブとして投入し、Hadoop の分散ファイルシステム上に保管されている巨大データを取り扱うことが可能です。そのため、分散アーキテクチャのメリットを活かすという観点から、Hadoop のエコシステムと組み合わせて Spark を利用することが少なくありません。本章では、Hadoop の分散ファイルシステムのデータを処理する高速エンジンとして、事例を交えながら Spark の利用方法を説明します。

5-1-2 Apache Spark の動作環境

次節以降では、Apache Hadoop クラスター上での Spark の利用方法を解説しますが、その前に、Apache Spark 自体の具体的な動作環境や必要条件などを簡単に紹介します。

Apache Spark は、1 台の物理サーバーだけで利用することも可能ですが、スケールアウト型のシステムとしても構成可能です。スケールアウト型の構成としては、マスターノードとワーカーノードに分かれ、マスターノードでは、Spark アプリケーションの起動のための管理プロセスやクラスターの管理を行うノードです。Hadoop 同様に、マスターノードとワーカーノード全体で 1 つのクラスターを構成し、Spark クラスターと呼ばれます。Apache Spark は、ワーカーノードを大量に並べ、スケールアウトのメリットが得られるアーキテクチャです。Hadoop 同様に、ワーカーノードを増やすことで性能が向上します。ワーカーノードでは、ユーザーの Spark アプリケーションがインメモリで稼働します（図 5-2）。

- **x86サーバー　：高速なメニーコアCPU, 大容量メモリを搭載**
- **ネットワーク　：10GbE以上を強く推奨**
- **サーバーOS　：RHEL, CentOS, Ubuntu Serverなどで稼働**
- **オープンソース　：Java, Python, Sparkなどのバージョンを統一**
- **HDFS, MapR-FS　：データ保管場所としてよく利用される**

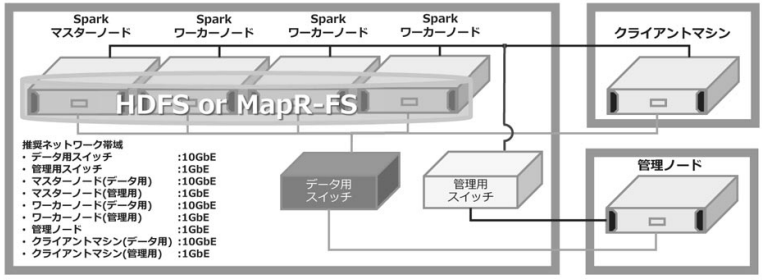


図 5-2　Apache Spark のマスターノードとワーカーノード

マスターノード、ワーカーノードともに、一般的な x86 サーバーが利用されますが、Apache Spark の得意とするインメモリによる繰り返し処理を考慮し、動作周波数が高くコア数の多い CPU と大容量メモリを搭載できるサーバーで構成すべきです。ワーカーノード間は、10Gbit イーサネットなどの高速なインターコネクトを利用するのがよいでしょう。OS については、RHEL、CentOS、Ubuntu Server などの代表的な Linux OS で動作が可能です。Red Hat 系 OS を採用する場合、テラバイト級の大規模メモリへの対応やファイルシステムの性能などの観点から、できるだけ RHEL 7.x や CentOS 7.x などの最新のサーバー OS を使います。また、OS のバージョン、Java、Python、

そして、Spark などのオープンソースソフトウェアのバージョンは、マスターノードとワーカーノードで統一しておくのがよいでしょう。

5-1-3 Spark の Driver プログラムと Executor

Apache Spark では、アプリケーションの実行の仕組みを担う Driver プログラムと Executor を理解する必要があります。Apache Spark における Driver プログラムは、アプリケーションの main 関数を実行するプロセスであり、Java プログラムです。Driver プログラムは、Spark コンテキスト（SparkContext、プログラムのソースコード内では「sc」と記述されます）と呼ばれるオブジェクトを生成します。一方、Executor は、ワーカーノードで稼働し、お互いに通信しながら、タスクを実行し、メモリまたはストレージ上にあるデータを保持します。Driver プログラムは、どの Executor でいつタスクを実行するかを決定する役目を担っています。

Apache Spark では、Driver と Executor に割り当てるメモリ容量などがアプリケーションの実行可否や性能に大きく影響します。これらの Driver や Executor は、Apache Spark が提供する設定ファイルで、パラメーターとしてクラスターの管理者が事前に定めます。また、ユーザーが Spark アプリケーションを実行する際にも、明示的に指定可能です（図 5-3）。

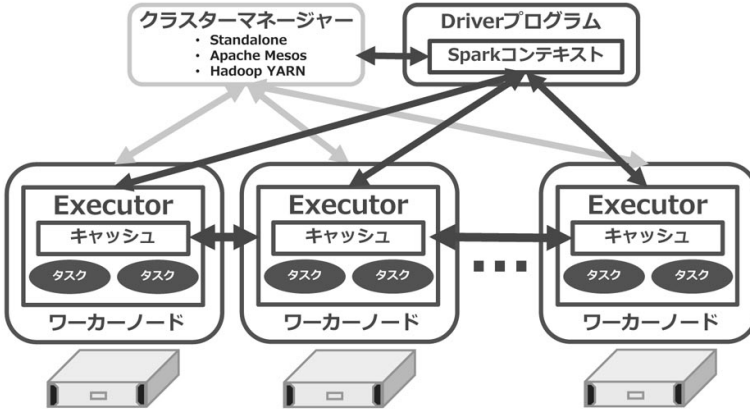


図 5-3 Apache Spark の Driver プログラムと Executor

5-1-4　クラスターマネージャ

Apache Spark では、クラスター全体のリソースを管理するクラスターマネージャが存在し、以下の 3 タイプが存在します。

Standalone：Spark だけで稼働するシンプルなクラスターマネージャ
Apache Mesos：Apache Mesos が提供するクラスターマネージャ
Hadoop YARN：Hadoop 2.0 以降で利用されているリソースマネージャ

●Standalone

Standalone は、Hadoop の YARN の仕組みを利用せずに、ジョブを Spark のみで投入するモードです。そのため Hadoop を必要とせず、Spark のみでインメモリ処理を行えます。ただし、インメモリで処理したデータは、なんらかの形で、全クラスターノードから参照することが少なくないため、Hadoop 分散ファイルシステムなどに保管したものを利用し、Spark のジョブを投入します。

●Apache Mesos

Apache Mesos は、Apache Mesos プロジェクトが提供するクラスターマネージャであり、分散システムのリソース管理などを行うソフトウェアです。Standalone では、ジョブ投入したユーザーがクラスターリソースを占有するのに対し、Apache Mesos を使った Spark のジョブ投入では、Apache Mesos のクラスターマネージャが空いたクラスターノードのリソースを Spark ジョブに割り当てます。これにより、クラスター全体の資源の有効利用が期待できます。

●Hadoop YARN

Hadoop YARN は、Hadoop が提供する YARN（リソースマネージャ）を使って Spark のジョブを投入するモードです。一般に YARN モードと呼ばれ、YARN を使って Spark のジョブを投入することを「Spark on YARN」といいます。Spark on YARN では、Apache Mesos 同様に、Hadoop 内蔵の YARN リソースマネージャが Spark ジョブに資源を割り当てます。Spark ジョブは、Hadoop の YARN 管理下のジョブとして取り扱われるため、YARN リソースマネージャの管理画面などから Spark ジョブの投入、実行、終了などの状態を確認できます。Hadoop クラスターに投入するジョブ管理を YARN で統一しているようなシステム要件では、この Spark on YARN が利用されます。

これらのクラスターマネージャは、Driver プログラムが持つ Spark コンテキストやワーカーノー

ドとやり取りを行い、Spark 上で実行するアプリケーションに Executor を割り当てる役目を担っています。

近年は、この 3 つに加え、Kubernetes（クーバネティス）と呼ばれるソフトウェアも試験的なサポートが開始されています。Kubernetes は、Docker コンテナに代表される、軽量なアプリケーション環境の操作を自動化するためのプラットフォームです。現在、「Apache Spark on Kubernetes」と呼ばれる GitHub が存在し、精力的な開発が続いています。

5-2 Spark on Hadoop 3 クラスターの構築

本節では、Hadoop 3 クラスター上に Apache Spark を構築・利用する基本的な方法を紹介します。今回は、システム構築上押さえておくべき必要最低限のパラメーターで構築しますが、実際の本番環境では、必要に応じて、さまざまなパラメーターを調整する必要があります。

5-2-1 Apache Spark クラスターの構築

Hadoop 3 クラスターのマスターノードに Spark マスターノードを構成し、Hadoop 3 クラスターのワーカーノードに Spark ワーカーノードを構成します。**表 5-1** は、今回構築するマスターノードとワーカーノードのソフトウェアの構成です。

表 5-1 マスターノードとワーカーノードのソフトウェアの構成

ソフトウェアコンポーネント	種類	バージョン
OS	CentOS	7.4.1708（本書では、CentOS 7.4 と表記）
Hadoop エンジン	Apache Hadoop	3.1.0
Spark エンジン	Apache Spark	2.3.0

Column ローカルモードのインストール

Apache Spark は、Hadoop クラスター上にインストールし、マスターノードとワーカーノードからなるクラスターを構成しますが、1 台のサーバーにインストールすることもあります。1 台のサーバーで稼働させる場合も、基本的には、マスターノードのサービスとワーカーノードのサービスの両方を稼働させます。1 台で構成された Apache Spark でも、Spark が提供する機械学習ライブラリ、グラフ解析パッケージなどが利用できるため、開発や小規模なテスト用と

して利用されます。ただし、スケールアウトメリットは得られないため、インメモリによる高速計算を求める場合は、複数ノードのクラスター構成で Spark を実行する必要があります。

■ DNS への登録、あるいは、/etc/hosts ファイルの設定

Apache Spark では、Apache Hadoop と同様に、クライアントノード、マスターノード、ワーカーノードすべてにおいて、DNS サーバーによる名前解決が必要です。DNS サーバーによる名前解決ができない場合は、全ノードの/etc/hosts ファイルに、クラスターを構成する全ノードのホスト名と IP アドレスの対応を記述しておく必要があります、

■ Java のインストール

Apache Hadoop クラスターにアクセスできる、クライアントマシンの n0120.jpn.linux.hpe.com で作業します。root 権限のコマンドプロンプトは、「#」、一般ユーザーのプロンプトは、「$」で表します。事前に clush コマンドをインストールし、全ノードで UID と GID が同一の一般ユーザーを作成し、全ノードのパスワード入力なしで SSH 接続できるようにしておいてください。Hadoop クラスターの構築においてすでにインストールされているはずですが、Apache Spark を動かすには、Java が必要になるので、事前にインストールしておきます。

```
# hostname
n0120.jpn.linux.hpe.com

# clush -g cl,all \
"yum makecache fast && yum install -y \
java-1.8.0-openjdk \
java-1.8.0-openjdk-devel"
```

■ Apache Spark の入手と展開

Apache Spark コミュニティの Web サイトから「spark-2.3.0-bin-hadoop2.7.tgz」を入手します。入手した spark-2.3.0-bin-hadoop2.7.tgz を全ノードの/opt ディレクトリに展開します。

```
# whoami
root
```

```
# wget \
https://www.apache.org/dist/spark/spark-2.3.0/spark-2.3.0-bin-hadoop2.7.tgz

# clush -g cl,all -c ./spark-2.3.0-bin-hadoop2.7.tgz --dest=/root/
# clush -g cl,all "tar xzvf /root/spark-2.3.0-bin-hadoop2.7.tgz -C /opt/"
```

■ 環境変数の設定

Apache Spark の圧縮済み tar アーカイブを展開すると、spark-2.3.0-bin-hadoop2.7 ディレクトリが生成されます。$HOME/.bash_profile に Hadoop の環境変数「HADOOP_HOME」が適切に設定されているかを確認します。

```
# clush -g cl,all -L "grep HADOOP_HOME= $HOME/.bash_profile"
n0121: export HADOOP_HOME=/opt/hadoop-3.1.0
...
```

「.bash_profile」に環境変数「SPARK_HOME」と「PATH」を追記します。

```
# cp $HOME/.bash_profile $HOME/.bash_profile.bak
# cat >> $HOME/.bash_profile << '__EOF__'
export SPARK_HOME=/opt/spark-2.3.0-bin-hadoop2.7
export PATH=$SPARK_HOME/bin:$SPARK_HOME/sbin:$PATH
__EOF__

# clush -a -c $HOME/.bash_profile --dest=$HOME/
```

同様に、一般ユーザー koga の「.bash_profile」にも環境変数を設定しておきます。

```
# su - koga
$ whoami
koga

$ cp $HOME/.bash_profile $HOME/.bash_profile.bak
$ cat >> $HOME/.bash_profile << '__EOF__'
export SPARK_HOME=/opt/spark-2.3.0-bin-hadoop2.7
export PATH=$SPARK_HOME/bin:$SPARK_HOME/sbin:$PATH
__EOF__

$ clush -a -c $HOME/.bash_profile --dest=$HOME/
$ exit
#
```

作業の効率化のため、クライアントマシン上で、環境変数をロードします。

```
# . $HOME/.bash_profile
```

環境変数の$SPARK_HOME の内容を確認します。

```
# clush -g cl,all -L "echo $SPARK_HOME"
n0120: /opt/spark-2.3.0-bin-hadoop2.7
...
```

■ slave ファイルの作成

Spark クラスターのワーカーノードのリストを記述した slave ファイルを作成し、全ノードにコピーします。以下は、ワーカーノードを 3 台で構成する場合の slave ファイルの作成例です。

```
# cat > $SPARK_HOME/conf/slave << __EOF__
n0122.jpn.linux.hpe.com
n0123.jpn.linux.hpe.com
n0124.jpn.linux.hpe.com
__EOF__

# clush -a -c \
$SPARK_HOME/conf/slave --dest=$SPARK_HOME/conf/
```

■ spark-defaults.conf ファイルの作成

全ノードにおいて、spark-defaults.conf ファイルのパラメーターを設定します。

```
# cat > $SPARK_HOME/conf/spark-defaults.conf << __EOF__
spark.master                spark://n0121.jpn.linux.hpe.com:7077
spark.serializer            org.apache.spark.serializer.KryoSerializer
spark.driver.memory         2g
spark.executor.memory       4g
spark.yarn.archive          hdfs:///spark
spark.yarn.jars             hdfs:///spark/*
__EOF__
```

上記は、Spark の Driver プログラムに割り当てるメモリ容量を 2GB、Executor に割り当てるメモリを 4GB に設定している例です。標準では 1GB が設定されているため、大容量メモリを搭載している場合は、Driver プログラムと Executor に割り当てるメモリ容量を明示的に指定するなどの調整が必要になります。ただし、実際には、Driver プログラムや Executor に割り当てるメモリ

容量だけでなく、Executor の数や Executor が利用する CPU コア数などの設定も併せて検討† する必要があります。

> † その他のパラメーターについては、Apache Spark の以下の URL に掲載されています。
> Apache Spark の Web サイトで公開されている「Spark Configuration」:
> https://spark.apache.org/docs/2.3.0/configuration.html

spark-defaults.conf ファイルを全ノードにコピーします。

```
# clush -a -c \
$SPARK_HOME/conf/spark-defaults.conf --dest=$SPARK_HOME/conf/
```

■ HDFS への jar ファイルのコピー

spark-defaults.conf ファイルにおいて「spark.yarn.archive」パラメーターに「hdfs:///spark」を指定しています。HDFS の/spark ディレクトリに Spark 関連の jar ファイルを配置しておくことで、Spark ジョブが投入されるたびに jar ファイルが HDFS へコピーされる処理を省略することができ、ジョブ投入にかかる時間を短縮できます。spark-defaults.conf ファイルで設定した「spark.yarn.archive」パラメーターに矛盾がないように、Spark 関連の jar ファイルを HDFS 上の/spark ディレクトリにコピーします。

```
# hdfs dfs -mkdir /spark
# hdfs dfs -put $SPARK_HOME/jars/* /spark/
# hdfs dfs -ls /spark/
Found 209 items
-rw-r--r-- 3 root supergroup   16993 2018-04-07 21:50 /spark/JavaEWAH-0.3.2.jar
-rw-r--r-- 3 root supergroup  201928 2018-04-07 21:51 /spark/RoaringBitmap-0.5.
11.jar
...
```

■ spark-env.sh ファイルの作成

spark-env.sh ファイルを作成します。Spark クラスター起動時にロードされる環境変数や各種パラメーターを記述します。

```
# cat > $SPARK_HOME/conf/spark-env.sh << '__EOF__'
HADOOP_CONF_DIR="/opt/hadoop-3.1.0/etc/hadoop/"
```

```
SPARK_EXECUTOR_MEMORY=4g
SPARK_WORKER_CORES=8
SPARK_WORKER_MEMORY=6g
SPARK_DAEMON_MEMORY=2g
SPARK_DAEMON_JAVA_OPTS="\
-Dspark.kryoserializer.buffer.mb=10 \
-Dspark.cleaner.ttl=43200\
"
__EOF__
```

上記は、あくまで一例であり、ワーカーノードとなる物理サーバーの CPU コア数とメモリ容量によってパラメーターの値が異なるため注意してください。ファイルを作成したら、全ノードにコピーします。

```
# clush -a -c \
$SPARK_HOME/conf/spark-env.sh --dest=$SPARK_HOME/conf/
```

■ Spark のマスターサービスの起動

Spark では、マスターノードの起動スクリプト start-master.sh が用意されています。事前に、マスターサービスを起動する start-master.sh スクリプトのパスを確認します。

```
# clush -g nn -L \
". $HOME/.bash_profile; which start-master.sh"
n0121: /opt/spark-2.3.0-bin-hadoop2.7/sbin/start-master.sh
```

マスターサービスを Spark マスターノード（今回は、n0121.jpn.linux.hpe.com）で起動します。ポートは、7077 番を使用します。start-master.sh スクリプトに、-h オプションを指定し、マスターノードのホスト名を付与します。また、-p オプションで、ポート番号を付与します。

```
# clush -g nn \
". $HOME/.bash_profile; \
start-master.sh \
-h n0121.jpn.linux.hpe.com \
-p 7077"
```

マスターノードで、Java のプロセス「Master」が起動できているかを確認します。

```
# clush -g nn jps | grep -v Jps
```

```
...
n0121: 55277 Master
...
```

■ スレーブサービスの起動

マスターノードと同様に、ワーカーノードのスレーブサービス起動する start-slave.sh スクリプトが用意されています。スレーブサービスを起動する start-master.sh スクリプトのパスを確認します。

```
# clush -g dn -L \
". $HOME/.bash_profile; which start-slave.sh"
n0122: /opt/spark-2.3.0-bin-hadoop2.7/sbin/start-slave.sh
...
```

スレーブサービスを起動します。start-slave.sh スクリプトでは、マスターサービスが稼働するホストの 7077 番ポートを指定します。

```
# clush -g dn \
". $HOME/.bash_profile; \
start-slave.sh \
spark://n0121.jpn.linux.hpe.com:7077"
```

すべてのワーカーノードにおいて Java のプロセス「Worker」が起動できているかを確認します。

```
# clush -g dn -L jps | grep Worker
n0122: 2043 Worker
...
```

■ Spark クラスターの状態確認

以上で、Spark クラスターが構築できました。Web ブラウザから Spark クラスターの状態を確認してみます。Spark クラスターの状態を確認するには、「http://マスターノードのホスト名:8080」にアクセスします（図 5-4）。

図 5-4　Spark クラスターの状態を Web ブラウザで確認

■ Spark クラスターの動作確認

Apache Spark の Scala 言語を使って、テキストファイルの行数を数えてみます。事前に対象となるテキストファイルを一般ユーザーのホームディレクトリに用意しておきます。Linux をインストールすると/usr/share/doc ディレクトリ以下に、アプリケーションごとにディレクトリが分かれており、さまざまなドキュメント類が保存されていますので、README などのテキストファイルをサンプルとして、一般ユーザー koga のホームディレクトリにコピーしておきます。

```
# hostname
n0120.jpn.linux.hpe.com

# su - koga
$ whoami
koga

$ cp /usr/share/doc/glibc-*/README $HOME/
```

まずは、Spark を使わずに、README ファイルの行数を数えてみます。

```
$ pwd
/home/koga
$ wc -l ./README
80 ./README
```

この README ファイルは、80 行であることがわかります。次に、Scala を使って、Hadoop の分散ファイルシステム「HDFS」上に保管した README ファイルの行数を数えてみます。まず、README ファイルを Hadoop クラスターの HDFS の/user/koga[†] にコピーします。

```
$ pwd
/home/koga

$ hdfs dfs -put ./README /user/koga/
```

† HDFS の/user/koga は、事前に作成し、ユーザー koga がアクセスできるように設定しておく必要があります。

Scala シェルを起動します。Scala シェルは、spark-shell コマンドで起動します。

```
$ which spark-shell
/opt/spark-2.3.0-bin-hadoop2.7/bin/spark-shell

$ spark-shell --master spark://n0121.jpn.linux.hpe.com:7077
...
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.3.0
      /_/

Using Scala version 2.11.8 (OpenJDK 64-Bit Server VM, Java 1.8.0_161)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

「--master」オプションの後に「spark://マスターノードのホスト名:7077」を付与します。Scala シェルが起動すると、プロンプトが「scala>」に変わり、対話型の操作が可能になります。Scala シェルで、先ほどの README ファイルの行数をカウントするには、以下のように sc.textFile("ファイルのパス") を指定し、変数に格納します。

```
scala> val line0001 = sc.textFile("hdfs://n0121:9000/user/koga/README")
...
```

変数 line0001 を使って、行数をカウントします。行数をカウントするには、変数の後に「.count()」を付与します。

```
scala> line0001.count()
res0: Long = 80

scala>
```

以上で、sc.textFile() で読み込んだ HDFS の/user/koga/README ファイルは、「Long = 80」となり、正しい結果が得られたようです。Scala シェルを終了するには、「sys.exit」を入力します。

```
scala> sys.exit
$
```

■ Scala におけるバッチ処理

上記の Scala プロンプトで入力した内容をファイルに保存しておき、バッチ処理することもできます。Scala 言語で作成したファイルをロードするには、spark-shell コマンドに「-i」オプションを指定します。以下は、HDFS 上の/user/koga/README ファイルの行数をカウントする Scala プログラム「linecount.scala」を作成し、spark-shell で実行する例です。

```
$ hostname
n0120.jpn.linux.hpe.com

$ cat > linecount.scala << '__EOF__'
val line0001 = sc.textFile("hdfs://n0121:9000/user/koga/README")
line0001.count()
sys.exit
__EOF__
```

作成した linecount.scala を Spark にロードし、HDFS の/user/koga/README ファイルの行数をカウントします。

```
$ spark-shell --master local[8] -i ./linecount.scala 2>/dev/null
...
res0: Long = 80
$
```

5-2-2　インメモリ処理の実行例

インメモリ処理の例として、繰り返し処理を行うモンテカルロ法によるシミュレーションを取り上げます。モンテカルロ法で円周率を求める python プログラムは、Apache Spark の tar アーカイブの中に収録されています（以下の URL からも入手可能です）。

モンテカルロ法で円周率を求める python プログラム pi.py：

https://github.com/apache/spark/blob/master/examples/src/main/python/pi.py

実際に Spark クラスターに pi.py を投入し、インメモリ処理により円周率の近似値を計算してみます。Python プログラムを Spark に投入するには、spark-submit コマンドを使用します。

「--num-executors」で、Executor の数を、「--executor-cores」で Executor に割り当てる CPU コア数を指定します。これもアプリケーションや環境によって調整が必要なパラメーターです。

```
$ hostname
n0120.jpn.linux.hpe.com

$ pwd
/home/koga

$ spark-submit \
--master spark://n0121.jpn.linux.hpe.com:7077 \
--num-executors 6 \
--executor-cores 2 \
$SPARK_HOME/examples/src/main/python/pi.py 999
...
Pi is roughly 3.141...
...
```

Note　リソース不足への対応

Spark のジョブを投入後、「Initial job has not accepted any resources; check your cluster UI to ensure that workers are registered and have sufficient resources」といったメッセージが表示される場合は、$SPARK_HOME/conf ディレクトリ以下の spark-defaults.conf ファイルと spark-env.sh ファイルに記載したパラメーターをチェックしてください。spark-defaults.conf ファイルと spark-env.sh ファイルのパラメーターを変更した場合は、マスターサービスと全スレーブサービスを再起動してください。

Column　モンテカルロ・シミュレーションとは？

モンテカルロ法（モンテカルロ・シミュレーション）は、乱数を使って近似解を求めるシミュレーション技法であり、機械学習、統計学でも利用されています。このモンテカルロ法を用いて、円周率πの近似値を求めることができます。モンテカルロ法は、正方形の中に一様に砂粒をまいて、円の内側に落ちた砂粒の数と、円の外側に落ちた砂粒の数をそれぞれ数え、正方形の面積に対して、円の中に入った砂粒の数の割合を算出することで円周率の近似値を求める算法です。図 5-5 のように、一辺が 2r の正方形に内接する半径 r の円を用意すると、正方形の面積は 2r × 2r であり、円の面積はπ × r × r です。円の面積π × r × r と正方形の面積 2r × 2r の比をとると、π/4 になります。両辺を 4 倍するとπ=4 ×面積比となります。面積比は、落下させる砂粒の数を多くすることにより真の値に近付いていき、円周率の近似値が得られます（図 5-5）。

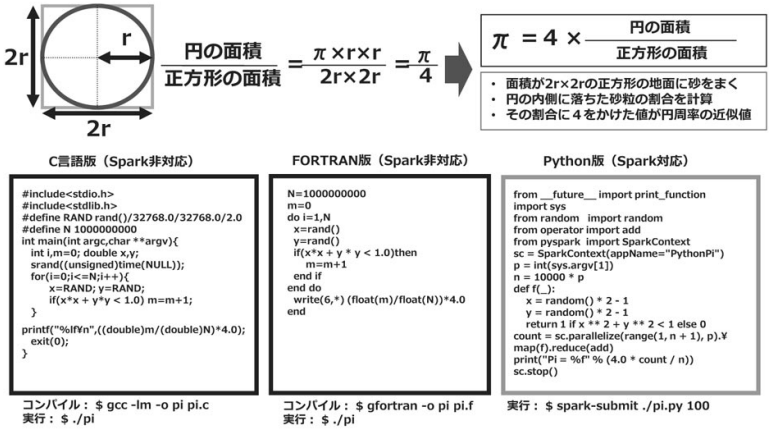


図 5-5　モンテカルロ法で円周率πの近似値を得る

モンテカルロ法などを使ったシミュレーションや数値解析の分野では、C 言語や FORTRAN などの開発言語が現在でも世界中で利用されています。C 言語や FORTRAN を使って分散環境で並列処理を行うには、一般に、Message Passing Interface（以下、MPI）などのライブラリが使われます。しかし、MPI を使ったプログラミングは、難易度が高く、研究者の多くが MPI による並列計算の実装で苦労しているのも事実です。筆者も学生時代は、ニューラルネットワークや人工知能のプログラムを C 言語と FORTRAN で記述していたのですが、メモリ管理を意識した関数などをすべて自前で作る必要があり、非常に苦労しました。最近では、シミュレーションや数値解析の分野においても、C 言語や FORTRAN だけでなく、Python や R のライブラリを駆使し、比較的容易に開発できるようになりました。人工知能、機械学習、深層学習といった知的情報処理への世界的なニーズもあり、Apache Spark では、並列処理のためのライブ

ラリが数多く提供されており、プログラムもシンプルに記述できます。C 言語や FORTRAN で並列計算の経験がある方は、ぜひ、機械学習用のサンプルプログラムやライブラリなどを使って、従来のプログラムと比較してみてください。

5-2-3 Spark on YARN による計算

Apache Spark では、Hadoop クラスターが提供する YARN を利用できます。これは、Spark on YARN と呼ばれ、Spark アプリケーションを Hadoop の YARN ジョブとして管理します。Spark on YARN には、YARN クライアントモードと YARN クラスターモードが存在します。

●YARN クライアントモード

YARN クライアントモードは、Spark のジョブ投入を行うコマンド（spark-submit）を実行したクライアントノードで Spark の Driver プログラムが稼働します。Spark の Driver プログラムは、クライアント側で稼働し、Spark の Executor は、YARN のクラスター上で稼働します。また、Hadoop クラスターのワーカーノード上で稼働している YARN の ApplicationMaster（アプリケーションのタスクの実行に関する調整役）は、YARN からのリソース要求のためだけに使用されます。Spark の Driver プログラム自体は、クライアントで実行された spark-submit コマンドのプロセス内で実行されるため、Driver プログラムが生成する出力結果は、クライアント上の標準出力に表示されます。そのため、アプリケーション開発者がデバッグする際や、結果をすぐに閲覧したい場合などに適したモードです。

●YARN クラスターモード

YARN クラスターモードは、Spark の Driver プログラムが、Hadoop クラスターのワーカーノード上で稼働している YARN の ApplicationMaster プロセス内で稼働します。すなわち、Spark の Driver プログラム、および、Executor の両方が YARN のクラスター上で稼働します。Spark のアプリケーションの結果は、標準出力ではなく、Hadoop クラスター側で提供されるジョブの Web 管理画面で確認します。実際の本番環境の Hadoop クラスターでアプリケーションをジョブとして投入して運用する場合に適したモードです。

以下では、YARN クライアントモードと YARN クラスターモードでのアプリケーションの実行と結果の確認手順を述べます。

■ YARN クライアントモードによるモンテカルロ・シミュレーションの実行

まずは、YARN クライアントモードでモンテカルロ・シミュレーションによる円周率の近似値を求めるアプリケーションを実行します。Spark on YARN では、「--master」に yarn を指定†します。また、YARN クライアントモードでは、--deploy-mode に「client」を指定します。

```
$ hostname
n0120.jpn.linux.hpe.com

$ pwd
/home/koga

$ $SPARK_HOME/bin/spark-submit \
--class org.apache.spark.examples.SparkPi \
--master yarn \
--deploy-mode client \
--num-executors 6 \
--executor-cores 2 \
$SPARK_HOME/examples/jars/spark-examples*.jar 999
...
Pi is roughly 3.141...
...
```

標準出力にアプリケーションの出力結果が得られました。

> † spark-env.sh スクリプトファイルに、HADOOP_CONF_DIR="/opt/hadoop-3.1.0/etc/hadoop/"が正しく設定されていないと、クライアントから Hadoop の ResourceManager（n121.jpn.linux.hpe.com:8032）にアクセスできず、ジョブの投入に失敗します。

■ YARN クラスターモードによるモンテカルロ・シミュレーションの実行

次に、YARN クラスターモードでアプリケーションを実行します。YARN クラスターモードでは、--deploy-mode に「cluster」を指定します。

```
$ $SPARK_HOME/bin/spark-submit \
--class org.apache.spark.examples.SparkPi \
--master yarn \
--deploy-mode cluster \
--num-executors 6 \
--executor-cores 2 \
$SPARK_HOME/examples/jars/spark-examples*.jar 999
```

```
...
... Application report for application_1519215057770_0008 (state: RUNNING)
... Application report for application_1519215057770_0008 (state: RUNNING)
... Application report for application_1519215057770_0008 (state: RUNNING)
...
... Application report for application_1515879367779_0006 (state: FINISHED)
...
```

YARN クラスターモードで実行したアプリケーションは、Hadoop の YARN アプリケーションとしてジョブ投入され、アプリケーション ID が振られます。アプリケーション ID は、上記の「application_1515879367779_0006_」で表される文字列です。アプリケーションの結果を確認するには、以下のように、yarn logs コマンドに「-applicationId」を付与し、その後にアプリケーション ID を指定して実行します。

```
$ which yarn
/opt/hadoop-3.1.0/bin/yarn

$ yarn logs -applicationId application_1515879367779_0006 | grep "Pi is roughly"
...
Pi is roughly 3.141893324743677
```

Note　YARN アプリケーションのログの確認

yarn logs コマンドを使って YARN アプリケーションのログを確認するには、Hadoop クラスターの yarn-site.xml ファイル内のパラメーター「yarn.log-aggregation-enable」と「yarn.nodemanager.remote-app-log-dir」の設定が必要です。yarn-site.xml ファイル内で、パラメーター「yarn.log-aggregation-enable」の値として「true」を設定すると、各 DataNode のログを統合できます。

```
$ grep -1 \
yarn.log-aggregation-enable \
$HADOOP_HOME/etc/hadoop/yarn-site.xml
<property>
  <name>yarn.log-aggregation-enable</name>
    <value>true</value>
```

ログの実体は、yarn-site.xml ファイル内で、パラメーター「yarn.nodemanager.remote-app-log-dir」に指定したディレクトリ（今回は、HDFS 上の/tmp/logs ディレクトリ）以下にアプリケーション ID のディレクトリが作成され、その中に保管されます。

```
$ grep -1 \
yarn.nodemanager.remote-app-log-dir \
$HADOOP_HOME/etc/hadoop/yarn-site.xml
<property>
  <name>yarn.nodemanager.remote-app-log-dir</name>
    <value>/tmp/logs</value>

$ hdfs dfs -ls /tmp/logs/koga/logs/
...
drwxrwx---  - koga hadoop ... /tmp/logs/koga/logs/application_1515879367779_0006

$ hdfs dfs -ls -C /tmp/logs/koga/logs/application_1515879367779_0006/
/tmp/logs/koga/logs/application_1515879367779_0006/n0122.jpn.linux.hpe.com_39531
/tmp/logs/koga/logs/application_1515879367779_0006/n0123.jpn.linux.hpe.com_34517
/tmp/logs/koga/logs/application_1515879367779_0006/n0124.jpn.linux.hpe.com_44448
```

Hadoop に限らず、分散システムにおいては、ノードごとに生成されるログだけでなく、分散システム全体のログを統合して管理することが少なくありません。このような分散したログの統合処理は、**ログ・アグリゲーション**と呼ばれます。Hadoop の場合は、YARN アプリケーションの結果の確認などを `yarn logs` コマンドで効率的に行うために、ログ・アグリゲーションの設定が不可欠です。

5-2-4　HDFS に保管されたデータの処理

別の例として、Spark クラスターを使って、HDFS に保管されたテキストファイル「README」をロードし、その README に含まれる単語ごとの出現回数をカウントしてみます。アプリケーション workdcount.py は、単語の出現回数をカウントする python プログラムです。単語数をカウントするサンプルプログラムは、$SPARK_HOME/examples/src/main/python ディレクトリに wordcount.py として収録されています。以下では、wordcount.py の入力ファイルとして、HDFS に格納されている README ファイルを指定しています。README ファイルは、HDFS の/user/koga ディレクトリに保管されている前提ですので、保管先のパスに注意してください。

```
$ hostname
n0120.jpn.linux.hpe.com

$ whoami
koga

$ spark-submit \
```

```
--master spark://n0121.jpn.linux.hpe.com:7077 \
--num-executors 6 \
--executor-cores 2 \
$SPARK_HOME/examples/src/main/python/wordcount.py \
hdfs://n0121.jpn.linux.hpe.com:9000/user/koga/README > /tmp/result.txt
```

HDFS 上の/user/koga/README ファイルに含まれる単語の出現回数ベスト 5 を表示します。

```
$ sort -nr -k 2 /tmp/result.txt | head -5
the: 38
: 23
C: 14
to: 12
GNU: 11
```

README ファイルに含まれる最頻出の単語は、定冠詞の「the」であることがわかります。今回の場合のように、分析対象のファイルサイズが小さいと、あまり Spark の効果がわかりませんが、ファイルサイズが巨大な場合や、膨大な数のファイルをロードすると、Spark を利用しない Hadoop のみでの処理時間と、Hadoop と Spark を組み合わせた処理時間に顕著な差が見られますので、ぜひ試してみてください。

5-3 Scala プログラム

Spark アプリケーションは、開発言語として Scala、Java、Python で記述されます。Scala で記述されたプログラムは、spark-shell コマンドを使って、コマンドラインからそのまま対話形式で実行可能です。しかし、本番環境では、対話形式ではなく、ソースコードから JAR ファイルをビルドし、ビルドされた JAR ファイルを使って spark-submit コマンドで実行するのが一般的です。

5-3-1 Scala プログラムのビルド方法

以下では、Scala プログラムから JAR ファイルをビルドする方法と実行方法を紹介します。

■ ビルドツールの入手

まずは、Scala プログラムをビルドするためのツール sbt を入手します。CentOS 7 の場合は、yum のリポジトリを追加して、パッケージをインストールします。

```
# hostname
n0120.jpn.linux.hpe.com

# curl https://bintray.com/sbt/rpm/rpm > /etc/yum.repos.d/sbt.repo
# cat /etc/yum.repos.d/sbt.repo
[bintray--sbt-rpm]
name=bintray--sbt-rpm
baseurl=https://sbt.bintray.com/rpm
gpgcheck=0
repo_gpgcheck=0
enabled=1

# yum install -y sbt
```

■ Scala プログラム用のディレクトリの作成

今回は、「Hello World」を表示する非常に単純な Scala プログラム helloworld.scala を作成し、コンパイル、実行します。まずは、helloworld.scala プログラムの開発のためのディレクトリ「/home/koga/helloworld」を用意します。この helloworld ディレクトリは、プロジェクトのルートディレクトリと呼ばれます。

```
# su - koga
$ mkdir /home/koga/helloworld
```

さらに、ソースコードを配置するためのディレクトリを作成します。

```
$ cd /home/koga/helloworld
$ mkdir -p src/main/scala/com/helloworld/spark
```

■ ソースコードの作成

helloworld.scala のソースコードを作成します。

```
$ cat > src/main/scala/com/helloworld/spark/helloworld.scala << __EOF__
package com.helloworld.spark
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
```

```
object helloworld {
  def main(args: Array[String]) {
    println("Hello World")
  }
}
__EOF__
```

■ helloworld.sbt ファイルの作成

プロジェクトのルートディレクトリ（helloworld ディレクトリ）直下にこのソフトウェアのビルドに関する定義ファイル「helloworld.sbt」を作成しますが、この helloworld.sbt ファイル内に Spark 本体のバージョン番号と Scala のバージョン番号を記述する必要があるため、事前にこれらのバージョンを確認しておきます。

```
$ spark-shell --version
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.3.0
      /_/

Using Scala version 2.11.8, OpenJDK 64-Bit Server VM, 1.8.0_161
...
```

Spark のバージョン番号が 2.3.0、Scala のバージョン番号が、2.11.8 であることを確認したので、helloworld.sbt ファイルをプロジェクトのルートディレクトリ直下に記述します。

```
$ cd /home/koga/helloworld
$ cat > helloworld.sbt << __EOF__
name := "Hello World"
version := "1.0"
scalaVersion := "2.11.8"
libraryDependencies += "org.apache.spark" %% "spark-core" % "2.3.0"
__EOF__
```

5-3-2　JAR ファイルのビルドと実行

helloworld.sbt ファイルができたので、ソースコードをコンパイルし、Jar ファイルをビルドし、アプリケーションを実行します。

■ ソースコードのコンパイル

ソースコードのコンパイルは、sbt コマンドで行います。

```
$ pwd
/home/koga/helloworld

$ sbt package
...
[success] Total time: 365 s, completed Apr 7, 2018 2:18:45 PM
```

ビルドが成功したら、生成された JAR ファイルを確認します。

```
$ ls -l ./target/scala-2.11/
total 4
drwxr-xr-x 3 koga hadoop   17 Apr 7 14:18 classes
-rw-r--r-- 1 koga hadoop 1853 Apr 7 14:18 hello-world_2.11-1.0.jar
drwxr-xr-x 5 koga hadoop   66 Apr 7 14:18 resolution-cache
```

■ アプリケーションの実行

ビルドが成功したら、アプリケーションを実行します。アプリケーションの実行は、spark-submit コマンドで行います。実行の際に、--class オプションを指定し、その後に「com.helloworld.spark.helloworld」を指定します。

```
$ which spark-submit
/opt/spark-2.3.0-bin-hadoop2.7/bin/spark-submit

$ spark-submit \
--master local \
--class com.helloworld.spark.helloworld \
./target/scala-2.11/hello-world_2.11-1.0.jar
Hello World
```

5-4 Spark on MapR クラスターの構築

Apache Hadoop 3 クラスターでは、Apache コミュニティが提供する Spark をインストールしました。一方、MapR クラスターでは、MEP（MapR Expansion Pack）で Spark の RPM パッケージ† が提供されており、非常に簡単にインストールできます。以下では、MapR 版の Spark を MapR クラスターにインストールする手順を紹介します。

> † 本書では、特に断りがない限り、Apache コミュニティ版の Spark を Apache Spark と呼び、MapR 社が提供する Spark パッケージを MapR 版の Spark と呼ぶことにします。

5-4-1 Spark パッケージのインストール

MapR 社から提供されている Spark パッケージをインストールします。clush コマンドが実行できるクライアントノードで作業します。

```
# hostname
n0130.jpn.linux.hpe.com

# clush -a "yum install -y mapr-spark mapr-spark-historyserver zip sshpass"
...
```

インストールした Spark パッケージを確認します。

```
# clush -aL "ls -l /opt/mapr/spark/"
n0131: total 0
n0131: drwxr-xr-x 16 mapr root 246 Apr 7 19:51 spark-2.2.1
...
```

上記より、Spark 2.2.1 がインストールされていることがわかります。Spark を利用する mapr ユーザーに環境変数を設定します。

```
# clush -a \
"cat >> /home/mapr/.bash_profile << '__EOF__'
export SPARK_HOME=/opt/mapr/spark/spark-2.2.1
export PATH=\$SPARK_HOME/bin:\$SPARK_HOME/sbin:\$PATH
__EOF__"

# clush -aL "grep SPARK_HOME /home/mapr/.bash_profile"
```

```
n0131: export SPARK_HOME=/opt/mapr/spark/spark-2.2.1
n0131: export PATH=$SPARK_HOME/bin:$SPARK_HOME/sbin:$PATH
...

# clush -a "chown mapr:mapr /home/mapr/.bash_profile"
```

■ JAR ファイルの配置

Spark on YARN は、クラスターノードの非 MapR-FS 上に格納されている Spark 用の JAR ファイルを使用します。この JAR ファイルを MapR-FS に格納しておけば、アプリケーションの実行時にファイルを配布せずに済むため、ジョブ実行時間の短縮が期待できます。以下では、Spark 用の JAR ファイルを MapR-FS 上に保存する手順を示します。

まず、MapR クラスターノード上の複数の JAR ファイルを単一の zip 圧縮ファイルにします。

```
# clush -w n0131 \
"cd /opt/mapr/spark/spark-2.2.1/jars; \
zip /opt/mapr/spark/spark-2.2.1/spark-jars.zip ./*"
```

作成した zip ファイルを MapR-FS の/user/mapr ディレクトリにコピーし、所有者を mapr、所有グループを mapr に変更します。クライアントマシンで hadoop コマンドが利用できるように事前に MapR クライアントの設定を行っておいてください。もしクライアントマシンで hadoop コマンドが利用できない場合は、クラスターノード上で hadoop コマンドを実行します。

```
# clush -w n0131 "hadoop fs -mkdir -p /user/mapr"
# clush -w n0131 "hadoop fs -chown mapr:mapr /user/mapr"
# clush -w n0131 \
"hadoop fs -put -f \
/opt/mapr/spark/spark-2.2.1/spark-jars.zip /user/mapr/"

# hadoop fs -chown mapr:mapr /user/mapr/spark-jars.zip
# hadoop fs -ls /user/mapr/spark-jars.zip
-rwxr-xr-x   3 mapr mapr  158873157 2018-04-07 21:14 /user/mapr/spark-jars.zip
```

■ spark-defaults.conf ファイルの編集

spark-defaults.conf ファイルをクライアントマシンにコピーします。

```
# hostname
```

```
n0130.jpn.linux.hpe.com

# scp \
n0131-mgm:/opt/mapr/spark/spark-2.2.1/conf/spark-defaults.conf /root/
```

spark-defaults.conf ファイルに zip ファイルのパスを記述します。

```
# cat >> /root/spark-defaults.conf << __EOF__
spark.yarn.archive maprfs:///user/mapr/spark-jars.zip
__EOF__
```

ファイルの内容を確認します。

```
# cat /root/spark-defaults.conf | grep -v ^# | grep -v ^$
spark.logConf                          true
spark.eventLog.enabled                 true
spark.eventLog.dir                     maprfs:///apps/spark
spark.history.fs.logDirectory          maprfs:///apps/spark
spark.sql.warehouse.dir                maprfs:///user/${system:user.name}/spark-war
ehouse
spark.sql.hive.metastore.sharedPrefixes  com.mysql.jdbc,org.postgresql,com.micr
osoft.sqlserver,oracle.jdbc,com.mapr.fs.shim.LibraryLoader,com.mapr.security.JN
ISecurity,com.mapr.fs.jni,com.mapr.fs.ShimLoader  (1 行入力)
spark.executor.memory                  2g
spark.yarn.archive maprfs:///user/mapr/spark-jars.zip
```

spark-defaults.conf ファイルを全ノードにコピーします。

```
# clush -a -c \
/root/spark-defaults.conf \
--dest=/opt/mapr/spark/spark-2.2.1/conf/
```

■ Spark アプリケーション用のディレクトリの作成

MapR-FS 上に Spark アプリケーションが利用するディレクトリを作成しておきます。

```
# hadoop fs -mkdir -p /apps/spark
# hadoop fs -chmod 777 /apps/spark
# hadoop fs -ls /apps
Found 1 items
drwxrwxrwx   - root root          0 2018-04-08 02:36 /apps/spark
```

5-4-2 Spark アプリケーションの実行

実行環境の動作を確認するため、MapR クラスターノード上で、モンテカルロ・シミュレーションによる円周率の近似値を求めるサンプルプログラムを実行します。

```
# ssh n0131-mgm
# su - mapr
$ whoami
mapr
$ which run-example
/opt/mapr/spark/spark-2.2.1/bin/run-example

$ run-example \
--master yarn \
--deploy-mode client \
--num-executors=9 SparkPi 8000
...
Pi is roughly 3.141595068926994
...
```

以上で、MEP で提供されている MapR 版の Spark を MapR クラスターにインストールできました。Hadoop の YARN を使って Spark のインメモリジョブを稼働させる「Spark on YARN」では、上記のように、`run-example` のオプション「`--master`」に「`yarn`」を指定します。

5-5 MapR 版 Spark のスタンドアロン構築

先ほどの環境は、Hadoop の YARN を使って Spark のジョブを稼働させる「Spark on YARN」環境でしたが、Hadoop の YARN が稼働しない環境で Spark ジョブを稼働させたい場合があります。そのような環境は、Spark スタンドアロンと呼ばれます。

以下では、先述の「Spark on YARN」の設定に追加する形で、Spark スタンドアロンを設定する手順を示します。まず、Spark スタンドアロンでは、Spark マスターサービスと Spark ワーカーサービスによって Spark クラスターを形成します。そのため、Spark マスターサービスを提供するパッケージを Spark マスターノードとなるホストにインストールします。今回は、Spark マスターノードを n0131 としました。

```
# clush -w n0131 "yum install -y mapr-spark-master"
```

■ Spark ワーカーノードのリストを記述

次に、ワーカーノードのリストを記述します。

```
# clush -a "cat > /opt/mapr/spark/spark-2.2.1/conf/slaves << __EOF__
n0131.jpn.linux.hpe.com
n0132.jpn.linux.hpe.com
n0133.jpn.linux.hpe.com
__EOF__"
```

■ Spark マスターノードの SSH 公開鍵のコピー

Spark スタンドアロンでは、Spark マスターノード（今回の場合は、n0131.jpn.linux.hpe.com）の SSH 公開鍵を全ワーカーノード（n0131、n0132、n0133）にコピーする必要があります。以下は、事前に root アカウントのパスワードが「password1234」で設定されている場合のコマンド実行例です。

```
# clush -w n0131 "ssh-keygen -f $HOME/.ssh/id_rsa -t rsa -N ''"
# clush -w n0131 \
'sshpass -p "password1234" \
ssh-copy-id -f -o StrictHostKeyChecking=no root@n0131'

# clush -w n0131 \
'sshpass -p "password1234" \
ssh-copy-id -f -o StrictHostKeyChecking=no root@n0132'

# clush -w n0131 \
'sshpass -p "password1234" \
ssh-copy-id -f -o StrictHostKeyChecking=no root@n0133'
```

以上で、Spark マスターノードから全ワーカーノードに対して、パスワード入力なしで SSH 接続ができるようになりました。

■ Spark ワーカーサービスの起動

Spark ワーカーサービス起動します。Spark ワーカーサービスは、start-slaves.sh スクリプトで起動します。ここで、MapR 版 Spark に特有の注意点があります。MapR 版の Spark において、Spark マスターサービスは、Warden サービスによって管理されているため、Spark マスターサービスを手動で起動してはいけません。

```
# clush -w n0131 " /opt/mapr/spark/spark-2.2.1/sbin/start-slaves.sh"
```

Spark ワーカーサービスが起動しているかを確認します。

```
# clush -aL jps | grep -i worker
n0131: 26332 Worker
n0132: 6489 Worker
n0133: 15375 Worker
```

クラスターを再構成します。

```
# clush -a "/opt/mapr/server/configure.sh -R"
```

以上で、Spark スタンドアロンが構築できました。

■ Spark スタンドアロン環境でのジョブの実行

Hadoop の YARN を利用せずに Spark ジョブを実行してみます。YARN を利用せずに Spark スタンドアロンでジョブを実行するには、「--master」に「spark://Spark マスターノードのホスト名:7077」を指定します。

```
# ssh n0131-mgm
# su - mapr
$ whoami
mapr

$ run-example \
--master spark://n0131.jpn.linux.hpe.com:7077 \
--num-executors=9 SparkPi 8000
```

以上で、MapR クラスターにおいて、Spark on YARN と Spark スタンドアロンの 2 種類でジョブ投入できるようになりました。

5-6 Spark SQL

Spark SQL は、その名のとおり、Spark 上から SQL のようなクエリー処理を実現するモジュールです。Spark SQL は、リレーショナルデータベースにおけるリレーショナル処理だけでなく、Spark の関数型プログラミングも可能です。SparkSQL では、宣言型の DataFrame API が提供され

ており、手続き型処理とリレーショナル処理を混在させたコードを記述できます。Scala、Python、Java、R で使用できる SQL への問い合わせ命令や、DataFrame API によって、データにクエリーできます。また、Spark SQL は、CSV 形式、JSON 形式、HDFS、MapR-FS、NoSQL の Cassandra、検索エンジンの Elasticsearch などのさまざまなデータ形式をサポートしています。

Python や R などのアプリケーションからの問い合わせや、Hadoop で SQL ライクな問い合わせを実現する Hive のクエリー言語 HiveQL 経由での問い合わせもサポートしています。

5-6-1　Spark SQL によるクエリー処理

以下では、Apache Hadoop 3 クラスターで Spark SQL を使ったクエリー処理を行います。まず、Spark SQL によってクエリー処理を行う対象となるとテストデータ「emp.json」を作成します。hadoop コマンドが使えるクライアントノードで作業します。

```
$ cat > $HOME/emp.json << __EOF__
{"name":"Masazumi Koga", "age":"43"}
{"name":"Satoshi Nakamoto", "age":"58"}
{"name":"Jiro Yamada", "age":"52"}
{"name":"Koji Sato", "age":"27"}
{"name":"Ken Hayashi", "age":"38"}
__EOF__
```

作成した「emp.json」を HDFS の/tmp ディレクトリにコピーします。

```
$ hadoop fs -put $HOME/emp.json /tmp/

$ which spark-shell
/opt/spark-2.3.0-bin-hadoop2.7/bin/spark-shell

$ spark-shell
...
scala>  以下、Scala のプロンプトで作業する
```

SparkSQL を利用するために必要なモジュールをインポートします。

```
scala> import org.apache.spark.sql.SparkSession
```

SparkSession を生成します。

```
scala> val spark = SparkSession.builder().appName("Spark SQL basic example").co
```

```
nfig("spark.some.config.option", "some-value").getOrCreate() 1 行入力
```

implicits クラスを Spark セッションにインポートします。

```
scala> import spark.implicits._
```

■ JSON ファイルのロード

JSON ファイルをロードし、DataFrame「df」を作成し、これを使ってデータを操作します。

```
scala> val df = spark.read.json("/tmp/emp.json")
```

まず、作成した DataFrame「df」から全レコードを表示します。

```
scala> df.show()
+---+----------------+
|age|            name|
+---+----------------+
| 43|   Masazumi Koga|
| 58|Satoshi Nakamoto|
| 52|     Jiro Yamada|
| 27|       Koji Sato|
| 38|     Ken Hayashi|
+---+----------------+
```

さらに、「name」列のみを表示してみます。

```
scala> df.select("name").show()
+----------------+
|            name|
+----------------+
|   Masazumi Koga|
|Satoshi Nakamoto|
|     Jiro Yamada|
|       Koji Sato|
|     Ken Hayashi|
+----------------+
```

以上で、HDFS に保管した JSON ファイルから、特定のレコードを抜き出して表示できました。

5-6-2 Spark SQL における特定の列の値に対する算術計算

Spark SQL では、表の中の値に対して、算術計算を行うクエリー処理が可能です。以下では表の「age」列の値が 57 を超える人物の名前を問い合わせます。

```
scala> df.filter($"age" > 57).show()
+---+----------------+
|age|            name|
+---+----------------+
| 58|Satoshi Nakamoto|
+---+----------------+
```

現在の年齢をプラス 5 歳した表示を表示します。

```
scala> df.select($"name", $"age" + 5).show()
+----------------+---------+
|            name|(age + 5)|
+----------------+---------+
|   Masazumi Koga|     48.0|
|Satoshi Nakamoto|     63.0|
|     Jiro Yamada|     57.0|
|       Koji Sato|     32.0|
|     Ken Hayashi|     43.0|
+----------------+---------+
```

Scala プロンプトを終了します。

```
scala> sys.exit
$
```

■ テキストファイル emp.txt のロード

Spark SQL では、区切り記号の「,」が入った CSV ファイルやテキストファイルもロードできます。例として、区切り文字「,」を含むテキストファイル「emp.txt」を作成します。

```
$ cat > $HOME/emp.txt << __EOF__
Masazumi Koga, 43
Satoshi Nakamoto, 58
Jiro Yamada, 52
Koji Sato, 27
Ken Hayashi, 38
```

```
__EOF__
```

テキストファイル「emp.txt」を分散ファイルシステムの/tmp ディレクトリにコピーします。

```
$ hadoop fs -put $HOME/emp.txt /tmp/
```

Spark Shell を起動します。

```
$ spark-shell
...
```

文字列の String 型の「name」と long 型「age」を定義します。

```
scala> case class empval(name: String, age: Long)
```

分散ファイルシステム上の/tmp/emp.txt ファイルをロードします。

```
scala> import org.apache.spark.sql.catalyst.encoders.ExpressionEncoder
scala> import org.apache.spark.sql.Encoder
scala> import spark.implicits._
scala> val df2 = spark.sparkContext.textFile("/tmp/emp.txt").map(_.split(",")).
map(attributes => empval(attributes(0), attributes(1).trim.toInt)).toDF() 1 行入力

scala> df2.createOrReplaceTempView("employee")
```

年齢が 20 から 40 の間の従業員の名前を問い合わせします。

```
scala> val df3 = spark.sql("SELECT name, age FROM employee WHERE age BETWEEN 20
AND 40") 1 行入力
scala> df3.map(youngster => "Name: " + youngster(0)).show()
+-----------------+
|            value|
+-----------------+
|  Name: Koji Sato|
|Name: Ken Hayashi|
+-----------------+

scala> sys.exit
$
```

以上で、HDFS に格納された表をロードし、SparkSQL によるクエリー処理を実行できました。

5-7 Spark Streaming

Twitter や Facebook などの SNS、全社 IT 基盤で生成されるサーバーシステムログ（たとえば、syslog や Apache Web サーバーのアクセスログなど）、IoT 基盤におけるセンサの出力など、日々データが生成されるシステムにおいては、それらのデータをリアルタイムでビッグデータ分析基盤に取り込んで分析を行う必要があります。このように、生成されるデータをリアルタイムに流し込むシステムは、ストリーミング基盤と呼ばれます。Apache Spark には、このデータストリーミングを行う機能があり、Spark Streaming と呼ばれます（図 5-6）。

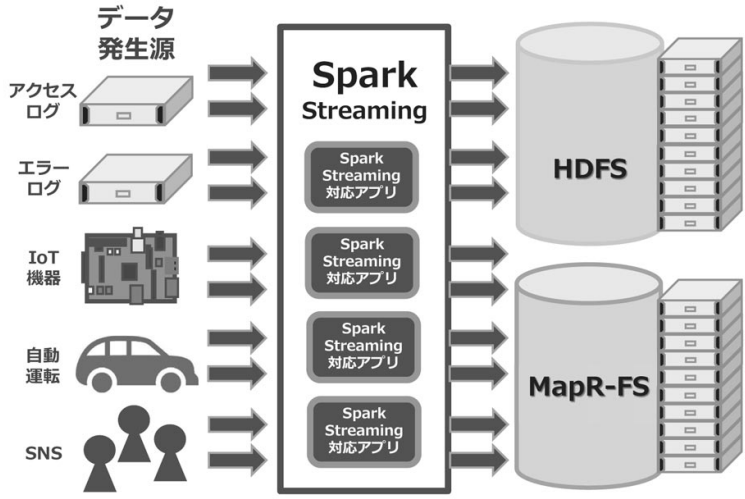


図 5-6　Spark Streaming を取り巻く環境

以下では、Apache Hadoop 3 クラスターの HDFS に流し込まれるデータをチェックし、そのデータに含まれる単語の出現回数を Spark Streaming でリアルタイムに表示する例を紹介します。

5-7-1 Spark Streaming 用テストプログラムの作成

HDFS の`/tmp/dir01` ディレクトリに入ってくるファイルに含まれる単語とその数をカウントする Spark Streaming 対応のアプリケーション「`hdfswc.py`」を作成します。この Python プログラムは、HDFS のマスターノードが n0121.jpn.linux.hpe.com で構成され、n0121.jpn.linux.hpe.com で名前解決ができ、Apache Hadoop 3 クラスター上で Spark クラスターが正常に稼働できていることが前

提条件です。

アプリケーション「hdfswc.py」は、結果が標準出力に表示されますが、標準では、Spark エンジンが INFO レベルのログが大量に表示されるため、事前準備として、アプリケーションの出力のみを表示するように、Spark エンジンのログ出力を抑制します。Spark エンジンのログの抑制は、$SPARK_HOME/conf/log4j.properties ファイルにログレベルを記述することで実現できます。設定ファイルのテンプレートである「log4j.properties.template」をコピーして利用します。

```
# hostname
n0120.jpn.linux.hpe.com

# cd $SPARK_HOME/conf
# cp log4j.properties.template log4j.properties
```

ログの設定ファイル「log4j.properties」の中の「log4j.rootCategory=」で指定されている「INFO, console」を「WARN, console」に修正します。これにより、Spark エンジンが出力ログを WARN 以上に抑制できます。

```
# vi log4j.properties
...
log4j.rootCategory=WARN, console
...
```

log4j.properties ファイルを記述したら Spark の全ノードにコピーします。

```
# clush -a -c log4j.properties --dest=$SPARK_HOME/conf/
```

以上で、Spark のログを WARN 以上に抑制できました。

次に、アプリケーション「hdfswc.py」を一般ユーザーで作成します。

```
# hostname
n0120.jpn.linux.hpe.com

# su - koga
$ whoami
koga

$ pwd
/home/koga

$ vi hdfswc.py
```

```
# -*- coding:utf-8 -*-
from pyspark.context import SparkContext
from pyspark import StorageLevel
from pyspark.streaming import StreamingContext  ←1
sc = SparkContext("spark://n0121:7077", "hdfswc")  ←2
ssc = StreamingContext(sc, 10)  ←3
L = ssc.textFileStream("hdfs://n0121.jpn.linux.hpe.com:9000/tmp/dir01")  ←4
W = L.flatMap(lambda line: line.split(" ")).filter(lambda x:x)  ←5
WC = W.map(lambda word: (word, 1)).reduceByKey(lambda x, y: x + y)  ←6
WC.pprint()  ←7
ssc.start()
ssc.awaitTermination()
```

プログラムの説明：

1. Spark Streaming を利用するためのモジュールをインポート
2. Spark クラスターのマスターノード「spark://n0121:7077」を指定
3. 流し込まれるファイルを 10 秒ごとに監視
4. HDFS の/tmp/dir01 ディレクトリをデータストリームの対象に指定
5. 半角スペースで区切って単語を読み取り
6. 得られた単語を集計
7. 単語を表示

5-7-2　Spark Streaming を使ったアプリケーションの実行

アプリケーション「hdfswc.py」を実行します。hdfswc.py は、HDFS の/tmp/dir01 ディレクトリに格納されるファイルを 10 秒ごとに監視しています。ファイルが新たに格納されない場合は、時刻しか表示されません。

```
$ which spark-submit
/opt/spark-2.3.0-bin-hadoop2.7/bin/spark-submit

$ hdfs dfs -mkdir /tmp/dir01
$ spark-submit \
--master spark://n0121.jpn.linux.hpe.com:7077 \
./hdfswc.py
-------------------------------------------
Time: 2018-04-08 03:51:30
-------------------------------------------

-------------------------------------------
Time: 2018-04-08 03:51:40
-------------------------------------------
```

■ HDFS にファイルを流し込む

別の端末を開き、テキストファイル「file01」を HDFS に流し込みます。

```
$ echo "Masazumi Koga - City: Tokyo - Country: Japan" |\
hdfs dfs -put -f - /tmp/dir01/file01
```

hdfswc.py は、HDFS の/tmp/dir01 ディレクトリに保管された、ファイル内の単語の種類とその出現回数を計算し、結果を標準出力に表示します†（図 5-7）。

```
File  Edit  View  Search  Terminal  Help
-------------------------------------------
Time: 2018-04-08 03:56:20
-------------------------------------------

-------------------------------------------
Time: 2018-04-08 03:56:30
-------------------------------------------

-------------------------------------------
Time: 2018-04-08 03:56:40
-------------------------------------------

-------------------------------------------
Time: 2018-04-08 03:56:50
-------------------------------------------
(u'City:', 1)
(u'Country:', 1)
(u'Koga', 1)
(u'-', 2)
(u'Japan', 1)
(u'Masazumi', 1)
(u'Tokyo', 1)
```

図 5-7　hdfswc.py の実行結果をリアルタイムで出力

† クライアントノード、および、Spark クラスターノードで時刻同期ができていない場合、hdfswc.py は、正常に機能しません。クライアントノードと Spark クラスターノードで同じ時刻を刻んでいるかどうかを確認してください。

さらに、別の内容のテキストファイル「file02」を HDFS の/tmp/dir01 ディレクトリに流し込みます。

```
$ echo "Satoshi Nakamoto - City: Unknown - Country: Unknown " |\
hdfs dfs -put -f - /tmp/dir01/file02
```

hdfswc.pyは、追加で流し込まれたファイル「file02」に含まれる単語とその数をカウントした結果をリアルタイムで出力します。

hdfswc.pyを見てわかるように、Sparkでは、ストリーミングのための開発フレームワークが提供されているため、非常に簡単にプログラムを作成、実行できます。ただし、このサンプルでは散発的なデータを拾っているだけですが、本来は、大量のログやSNSのメッセージなどが流し込まれるような環境で、Spark Streamingによる分析処理が行われます。このようなデータ処理では、ストリーミングデータを収集し、処理するというだけでなく、処理データの蓄積、加工、分析までも考慮した基盤の構築とアプリケーション開発が必要となります。

5-8 Spark GraphX

従来、グラフを使った理論は、研究機関や学術分野で議論されてきました。しかし、インターネット企業の成長に伴い、多くの米国のソーシャルメディア企業は、グラフ理論を使って、自社が抱えるソーシャルメディアのユーザーや興味の対象の関係性（パターン）を発見し、その関係性で広告収入を得ようと考えました。Twitter、Facebook、Googleなどがこぞってグラフ理論を研究していたのも、このソーシャルメディアにおける活用に他なりません。具体的には、ユーザーの画面のクリック情報を得ると、他のユーザーやコミュニティとのつながりを提示し、最も興味を引きそうな商品や広告を提案します。

グラフとは、複数のペアとなっている頂点（ノード）がエッジ（辺、枝）によって結ばれている頂点の集合を表現したものです。グラフには、有向グラフと通常のグラフの2種類があります。ある頂点Aから別の頂点Bに向かって一方向に走るという性質を持っていれば、そのグラフは、**有向グラフ**と呼ばれます。たとえば、Twitterのフォロワーは、有向グラフで表されます。Twitterのアカウント「Masazumi Koga」が別のTwitterのアカウント「Satoshi Nakamoto」のフォロワーであれば、アカウント「Masazumi Koga」と「Satoshi Nakamoto」は有向グラフでつながっているといえます。一方、Facebookのアカウントのように、相互に友達になっている場合、その2人は、通常のグラフでつながっているといえます。また、目的地までの最短飛行ルートを見つけるといった問題などにも、グラフ処理のアルゴリズムが利用されています。

5-8-1 Spark GraphXにおけるグラフ

Sparkには、グラフデータを処理する仕組みとしてSpark GraphXというライブラリが用意されています。Spark GraphXは、大規模なグラフ構造データを並列分散環境で高速処理するためのフ

レームワークを提供します。行列計算をせずにグラフデータの分散処理が可能であり、表構造のデータでは困難な処理が行えるという特徴があります。Spark GraphX は、グラフ処理用の API をユーザーに提供し、これを駆使することで、非常に簡素なソースコードの記述で大規模なグラフ構造のデータ処理を実現できます（図 5-8）。

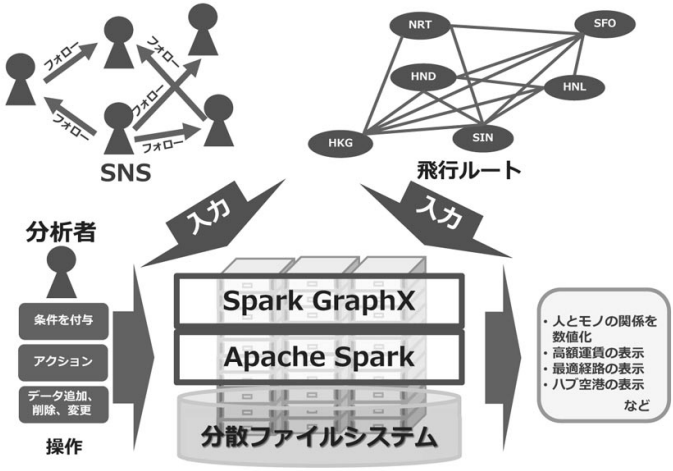


図 5-8　グラフ処理を行う Spark GraphX

5-8-2　Spark GraphX によるフライトデータ分析

以下では、Spark GraphX を使ってフライトデータの分析例を示します、分析対象のデータは、米国運輸省（United States Department of Transportation、DOT）の以下の URL で示される Web サイト Bureau of Transportation Statistics（以下、BTS）で公開されているものを使います。

BTS の Web サイト：

```
https://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236&DB_Short_Name=On-Time
```

■ データの取得

上記の BTS の Web サイトでは、過去のフライトデータについて、どのような情報を取得するかを選択できるようになっています。まずは、画面上部にある表の「On-Time: On-Time Performance」枠にある項目（表 5-2）をプルダウンメニューで選択します（図 5-9）。

表 5-2　On-Time Performance の選択肢

On-Time Performance	選択肢	意味
Filter Geography	All	米国本土のすべてのフライトを選択
Filter Year	2017	フライトがあった年
Filter Period	December	フライトがあった月

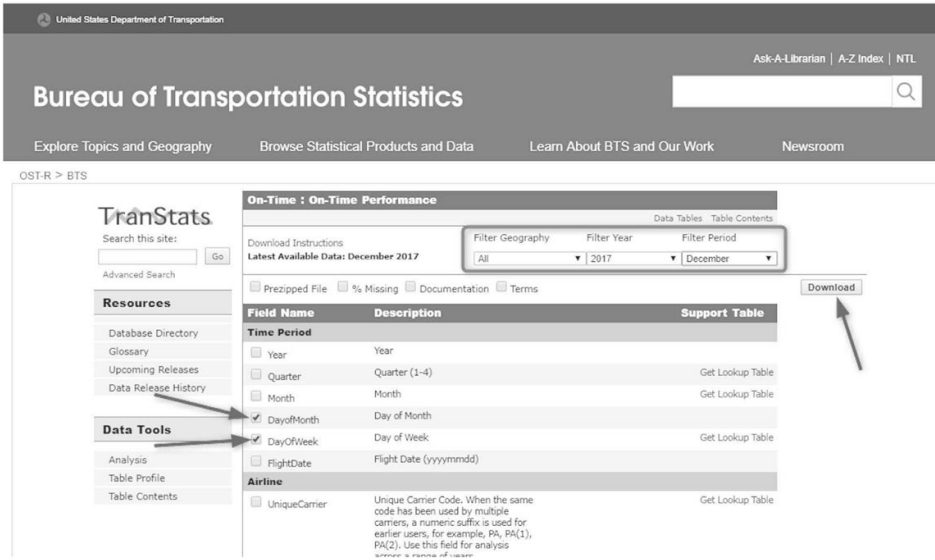


図 5-9　BTS の Web サイト画面

次に、「Time Period」枠と「Airline」枠について、表 5-3 と表 5-4 の項目のチェックボックスにチェックを入れます。以下の項目以外は、選択しません。

表 5-3　Time Period の選択肢

Time Period	意味
DayofMonth	フライト日
DayofWeek	曜日

表 5-4　Airline 項目の意味

Airline 項目	意味
Carrier	航空会社コード

TailNum	航空機の識別番号
FlightNum	便名

「Origin」枠について、**表 5-5** の項目のチェックボックスにチェックを入れます。

表 5-5　Origin 項目の意味

Origin 項目	意味
OriginAirportID	出発地の空港 ID
Origin	出発地の空港コード

上記の表の「Origin」枠内において、デフォルトでチェックボックスにチェックが入っている項目「OriginAirportSeqID」と「OriginCityMarketID」のチェックをはずします。

「Destination」枠について、**表 5-6** の項目のチェックボックスにチェックを入れます。

表 5-6　Destination 項目の意味

Destination 項目	意味
DestAirportID	目的地の空港 ID
Dest	目的地の空港コード

表 5-6 の「Destination」枠内において、デフォルトでチェックボックスにデフォルトが入っている「DestAirportSeqID」と「DestCityMarketID」のチェックをはずします。

「Departure Performance」枠について、**表 5-7** の項目のチェックボックスにチェックを入れます。

表 5-7　Departure Performance 項目の意味

Departure Performance 項目	意味
CRSDepTime	出発予定時刻
DepTime	実際に出発した時刻
DepDelayMinutes	出発遅延分数

「Arrival Performance」枠について、**表 5-8** の項目のチェックボックスにチェックを入れます。

表 5-8　Arrival Performance の意味

Arrival Performance 項目	意味
CRSArrTime	到着予定時刻
ArrTime	実際に到着した時刻
ArrDelayMinutes	到着遅延分数

「Flight Summaries」枠について、表 5-9 の項目のチェックボックスにチェックを入れます。

表 5-9　Flight Summaries の意味

Flight Summaries 項目	意味
CRSElapsedTime	予約システム（CRS）での経過時間（分）
Distance	距離（マイル）

上記の 17 項目にチェックが入っていることが確認できたら、BTS の Web サイト上部右上の［Download］ボタンをクリックします。すると、zip 圧縮されたフライトデータをダウンロードできるので、zip 圧縮を解凍し、中にある CSV ファイルを「flight.csv」という名前で手元のクライアントノードにコピーします。

```
$ hostname
n0120.jpn.linux.hpe.com

$ unzip 469864407_T_ONTIME.zip  ←入手したフライトデータ（名前はその都度変わります）
$ ls -lh ./469864407_T_ONTIME.csv
-rw-r--r-- 1 koga hadoop 46M Feb 26 21:35 ./469864407_T_ONTIME.csv

$ cp 469864407_T_ONTIME.csv flight.csv
$ head -4 ./flight.csv
"DAY_OF_MONTH","DAY_OF_WEEK","CARRIER","TAIL_NUM","FL_NUM","ORIGIN_AIRPORT_ID",
"ORIGIN","DEST_AIRPORT_ID","DEST","CRS_DEP_TIME","DEP_TIME","DEP_DELAY_NEW","CR
S_ARR_TIME","ARR_TIME","ARR_DELAY_NEW","CRS_ELAPSED_TIME","DISTANCE",
1,5,"AA","N723AA","2054",11057,"CLT",14100,"PHL","1300","1255",0.00,"1438","143
5",0.00,98.00,449.00,
1,5,"AA","N123AA","2055",11057,"CLT",13930,"ORD","1455","1452",0.00,"1607","154
9",0.00,132.00,599.00,
1,5,"AA","N123AA","2055",14635,"RSW",11057,"CLT","1201","1158",0.00,"1357","134
3",0.00,116.00,600.00,
```

取得したデータが 17 項目かどうかチェックし、各項目の内容（数値、2 桁コード、ID、時刻 等）

の並びを確認してください。

■ フライトデータの加工

入手したCSVファイルを加工します。まず、ファイル内の1行目に列の項目名が入っているため、sedコマンドで削除します。

```
$ sed -i '1d' flight.csv
$ head -2 flight.csv
1,5,"AA","N723AA","2054",11057,"CLT",14100,"PHL","1300","1255",0.00,"1438","143
5",0.00,98.00,449.00,
1,5,"AA","N123AA","2055",11057,"CLT",13930,"ORD","1455","1452",0.00,"1607","154
9",0.00,132.00,599.00,
```

次に、「"」が含まれている項目があるため、sedコマンドで「"」を削除します。

```
$ sed -i "s/\"//g" flight.csv
$ head -2 flight.csv
1,5,AA,N723AA,2054,11057,CLT,14100,PHL,1300,1255,0.00,1438,1435,0.00,98.00,449.
00,
1,5,AA,N123AA,2055,11057,CLT,13930,ORD,1455,1452,0.00,1607,1549,0.00,132.00,599
.00,
```

sedコマンドで行末の「,」を削除します。

```
$ sed -i 's/,$//g' flight.csv
$ head -2 flight.csv
1,5,AA,N723AA,2054,11057,CLT,14100,PHL,1300,1255,0.00,1438,1435,0.00,98.00,449.00
1,5,AA,N123AA,2055,11057,CLT,13930,ORD,1455,1452,0.00,1607,1549,0.00,132.00,599.00
```

フライトデータには、パラメーターが含まれていない項目もあります。パラメーターが含まれていない項目は、ファイル区切り文字の「,」が続くため、「,,」となっています。今回は、17項目が完全なデータのみを対象に分析を行いますので、ファイル中の「,,」が存在する行を削除します。

```
$ sed -i "/,,/d" flight.csv
```

「,,」が含まれている行が存在しないことを確認します。この時点で、項目が不完全なものが残っているとSpark GraphXのアプリケーションが正しくデータをロードできないので、必ず、「,,」が含まれている行が除外できているかを確認してください。

```
$ grep ",," flight.csv
```

パラメーターに含まれる「.00」と「.0」となっている小数点以下を削除します。

```
$ sed -i "s/\.00//g" flight.csv
$ sed -i "s/\.0//g" flight.csv
```

再度、「,,」が含まれていないか、パラメーターの小数点以下の「.00」「.0」が含まれていないかを確認します。

```
$ grep ',,' flight.csv
$ grep '\.00' flight.csv
$ grep '\.0' flight.csv
```

フライトデータの加工は、以上です。ファイルの先頭5行を確認します。

```
$ head -5 flight.csv
1,5,AA,N723AA,2054,11057,CLT,14100,PHL,1300,1255,0,1438,1435,0,98,449
1,5,AA,N123AA,2055,11057,CLT,13930,ORD,1455,1452,0,1607,1549,0,132,599
1,5,AA,N123AA,2055,14635,RSW,11057,CLT,1201,1158,0,1357,1343,0,116,600
1,5,AA,N650AA,2056,11057,CLT,14492,RDU,1616,1606,0,1716,1651,0,60,130
1,5,AA,N118AA,2056,11618,EWR,11057,CLT,1310,1309,0,1509,1450,0,119,529
```

加工済みのフライトデータをHDFSに格納します。

```
$ hdfs dfs -put -f flight.csv /user/koga/
```

■ フライトデータ分析アプリケーションの作成

Spark GraphXを使ってフライトデータの分析を行うアプリケーションを作成します。アプリケーションのソースコードが複数になるため、作業用のディレクトリ「$HOME/flight-analytics」を作成し、そこに移動して作業します。

```
$ mkdir $HOME/flight-analytics
$ cd $HOME/flight-analytics
```

作成するアプリケーションは、データのロードや分析を行うための本体プログラム（main-core.scala）とフライトデータへの問い合わせプログラムに分けました。プログラムは、Scalaで記述します。ユーザーは、事前に本体プログラムをspark-shellでロードし、分析者の要望に応じた

問い合わせ用の scala プログラムをロードすることで、フライトデータから望みの結果を得るという仕様にします。以下は、今回作成するソースコードとその役割です（表 5-10）。

表 5-10　各ソースコードの役割

ソースコード	役割
main-core.scala	HDFS 上のデータをロードし、分析に必要な変数などを定義
num-of-airports.scala	空港の数を表示する
num-of-routes.scala	フライト経路（ルート）の数を表示
long-flight.scala	最長ルートの上位 5 件を表示
over-N-miles.scala	4500 マイルを超える距離のルートを表示
maxout-airports.scala	出発便数が多い空港上位 5 箇所を表示
hub-airport.scala	他の空港との接続が最も多い空港を 5 件表示

まず、main-core.scala を作成します。

```
$ vi main-core.scala
import org.apache.spark._
import org.apache.spark.rdd.RDD
import org.apache.spark.util.IntParam
import org.apache.spark.graphx._  ←1
import org.apache.spark.graphx.util.GraphGenerators  ←2
case class F(
dofM:String, dofW:String, carrier:String, tailnum:String, flnum:Int,
src_id:Long, origin:String, dest_id:Long, dest:String, crsdeptime:Double,
deptime:Double, depdelaymins:Double, crsarrtime:Double, arrtime:Double,
arrdelay:Double, crselapsedtime:Double, dist:Int
  )
def parseF(str: String): F = {
  val L = str.split(",")
  F(
  L(0), L(1), L(2), L(3), L(4).toInt, L(5).toLong, L(6), L(7).toLong,
  L(8), L(9).toDouble, L(10).toDouble, L(11).toDouble, L(12).toDouble,
  L(13).toDouble, L(14).toDouble, L(15).toDouble, L(16).toInt
  )
}
val textRDD    = sc.textFile("hdfs://n0121:9000/user/koga/flight.csv")  ←3
val airports   = textRDD.map(parseF).cache().map(F => (F.src_id, F.origin)).dis
tinct  ←4
val R          = textRDD.map(parseF).cache().map(F => ((F.src_id, F.dest_id), F
.dist)).distinct  ←5
val nowhere    = "nowhere"
```

```
R.cache
val airportMap = airports.map { case ((src_id), name) => (src_id -> name) }.col
lect.toList.toMap  ←6
val edges      = R.map { case ((src_id, dest_id), distance) => Edge(src_id.toLo
ng, dest_id.toLong, distance) }  ←7
val graph      = Graph(airports, edges, nowhere)  ←8
```

プログラムの説明：

1. グラフ分析に必要な GraphX をインポート
2. グラフ生成に必要な GraphX をインポート
3. HDFS の/user/koga ディレクトリに保管したフライトデータ flight.csv のパスを textRDD に格納
4. textRDD に格納されているフライトデータの項目から、グラフの頂点となる空港情報を作成
5. textRDD に格納されているフライトデータの項目から、ルート情報を作成
6. 空港 ID と空港コードの対応付けを作成
7. グラフ作成に必要なルート情報（辺（エッジ））を作成
8. 空港情報、エッジ情報、デフォルトの頂点（nowhere）からグラフを作成

問い合わせ用のプログラムを作成します。

```
$ vi num-of-airports.scala
println("空港の数を表示")
graph.numVertices

$ vi num-of-routes.scala
println("ルートの数を表示")
graph.numEdges

$ vi long-flight.scala
println("最長ルートの上位5件を表示（出発地空港コード,到着地空港コード,距離）")
val N = 5
graph.triplets.sortBy(_.attr, ascending=false).map(triplet =>
triplet.srcAttr       + "から" +
triplet.dstAttr       + "までの距離は、" +
triplet.attr.toString + "マイル").take(N).foreach(println)

$ vi over-N-miles.scala
println("4500マイルを超える距離のルートを表示")
println("出発地空港ID,到着地空港ID,マイル数")
val M=4500
val N=100
graph.edges.filter {
 case ( Edge(src_id, dest_id,distance))=> distance > M
}.take(N).foreach(println)

$ vi maxout-airports.scala
```

```
println("出発便数が多い空港上位5箇所を表示")
println("空港ID,出発便数,空港コード")
val N=5
val maxout = graph.outDegrees.join(airports).sortBy(_._2._1, ascending=false)
maxout.take(N).foreach(println)

$ vi hub-airport.scala
println("他の空港との接続が最も多い空港を5件表示")
val N=5
val impAirports = graph.pageRank(0.1).vertices.join(airports).sortBy(_._2._1, f
alse).map(_._2._2)
impAirports.take(N).foreach(println)
```

以上でフライトデータ分析用の Scala プログラムがそろいました。

5-8-3　フライトデータの分析

では、フライトデータ分析を行います。今回は、Spark シェルを使って Scala プログラムを実行します。

```
$ which spark-shell
/opt/spark-2.3.0-bin-hadoop2.7/bin/spark-shell

$ spark-shell -i ./main-core.scala
...
scala>
```

最初に、データ自体が正しくロードできるかのチェックが必要です。HDFS に格納されたデータから空港 ID と空港コードを取得できるかを確認します。

```
scala> airports.take(2)
res1: Array[(Long, String)] = Array((14057,PDX), (11013,CIU))
```

Note　データの見直しポイント

この時点で、エラーが出る場合は、データ自体の加工に失敗している可能性があるため、データの加工手順と内容を見直す必要があります。

見直しポイント：

- 米国運輸省のBTSのWebサイトで間違った項目をチェックしていないか
- データの行末などに不要な「,」が入っていないか
- データの欠損などにより、データ内の区切り文字「,」記号が連続し「,,」などが含まれていないか
- sedコマンドを使ったデータの加工でオペレーションミスをしていないか
- フライトデータのデータが壊れていないか（フライトデータをクライアントマシンに転送する際に、FTPなどの転送時にバイナリモードやテキストモードなどを適切に設定せずに転送するなど）
- アプリケーション内において、HDFSの正しいパスとファイル名を指定できているか

データのロードに成功したら、データを使って問い合わせを行ってみます。まずは、「num-of-airports.scala」をロードし、空港の数を問い合わせます。Scalaのコマンドプロントから別のScalaプログラムをロードするには、「:load Scalaファイル名」と入力します。

```
scala> :load num-of-airports.scala
Loading num-of-airports.scala...
空港の数を表示
res3: Long = 294
```

筆者のフライトデータでは、294個の空港が登録されていることがわかりました。次に、飛行ルートの数を問い合わせてみます。

```
scala> :load num-of-routes.scala
Loading num-of-routes.scala...
ルートの数を表示
res5: Long = 4153
```

飛行ルートの数は、4153と表示されました。続けて、最長ルートの上位5件を問い合わせます。

```
scala> :load long-flight.scala
Loading long-flight.scala...
最長ルートの上位5件を表示（出発地空港コード,到着地空港コード,距離）
N: Int = 5
JFKからHNLまでの距離は、4983マイル
HNLからJFKまでの距離は、4983マイル
EWRからHNLまでの距離は、4962マイル
HNLからEWRまでの距離は、4962マイル
```

```
HNLからIADまでの距離は 、4817マイル
```

4500 マイルを超える長距離ルートを問い合わせます。

```
scala> :load over-N-miles.scala
Loading over-N-miles.scala...
4500マイルを超える距離のルートを表示
出発地空港ID,到着地空港ID,マイル数
M: Int = 4500
N: Int = 100
Edge(10397,12173,4502)
Edge(11618,12173,4962)
Edge(12173,11618,4962)
Edge(12478,12173,4983)
Edge(12173,10397,4502)
Edge(12173,12264,4817)
Edge(12173,12478,4983)
Edge(12264,12173,4817)
```

上記の距離 4502 マイルの場合を例に、出発地空港 ID（10397）と到着地空港 ID（12173）それぞれの空港コードを表示してみます。以下のように、プラス記号の後でキーボードの Enter キーを押して改行しています。すると、次の行の先頭に「|」記号が表示されますので、その後に、入力を続けることができます。このように、Scala シェルでは、1 行コマンド入力を複数行にわたって入力できます。

```
scala> println ("From: " +
     | "%s  ".format(airportMap(10397)) +
     | "To: %s ".format(airportMap(12173)))
From: ATL  To: HNL
```

上記より、出発地の空港コードが「ATL」で到着地の空港コードが「HNL」であることがわかりました。さらに、出発便数が多い空港上位 5 箇所を問い合わせます。

```
scala> :load maxout-airports.scala
Loading maxout-airports.scala...
出発便数が多い空港上位5箇所を表示
空港ID,出発便数,空港コード
N: Int = 5
...
(10397,(151,ATL))
(13930,(148,ORD))
(11292,(124,DEN))
```

```
(13487,(112,MSP))
(11298,(106,DFW))
```

最後に他の空港との接続が最も多い空港5件を問い合わせます。

```
scala> :load hub-airport.scala
Loading hub-airport.scala...
他の空港との接続が最も多い空港を5件表示
N: Int = 5
...
ATL
ORD
MSP
DEN
IAH
```

出発便数の多い空港と接続数が多い空港を知ることができましたので、フライトデータ分析を終了します。

```
scala> sys.exit
$
```

5-8-4 MapR クラスターでの GraphX を使ったフライトデータ分析

フライトデータ分析を MapR クラスターで行う場合は、Apache Hadoop 3 クラスターのクライアントノードにある Scala プログラム一式や flight.csv ファイルを MapR クラスターの spark-shell がインストールされているクラスターノードの1つ（n0131.jpn.linux.hpe.com など）の/home/mapr ディレクトリにコピーします。さらに、MapR クラスターノードにコピーされた/home/mapr/flight-analytics/flight.csv ファイルを MapR-FS の/user/mapr ディレクトリにコピーします。後は、main-core.scala ファイル内の flight.csv ファイルをロードする行を以下に変えるだけです。

```
$ vi main-core.scala
...
val textRDD    = sc.textFile("maprfs://n0131.jpn.linux.hpe.com:7222/user/mapr
/flight.csv")
...
```

上記の main-core.scala ファイルの修正ができたら、spark-shell コマンドをインストールし

ている MapR クラスターノード上で「spark-shell -i main-core.scala」を実行すれば、Apache Hadoop 3 クラスターでのフライトデータの分析とまったく同じ分析が行えます。

以上で、Scala GraphX を使った非常に簡単なフライトデータ分析ができました。

5-9 SparkR

一般に、統計解析処理におけるプログラミング言語としては、R 言語が利用されます。Spark には、R 言語で作成したプログラムを実行するための SparkR と呼ばれるモジュールが含まれています。R 言語と SparkR を組み合わせることにより、R 言語ベースの統計解析をインメモリで分散処理できるようになります。以下は、Apache Hadoop 3 クラスター上で SparkR を利用する方法について解説します。

5-9-1 R 言語のインストール

SparkR を利用するには、R 言語が必要です。CentOS 7 対応の EPEL リポジトリから yum コマンドで R 言語のパッケージをインストールします。

```
# hostname
n0120.jpn.linux.hpe.com

# clush -g cl,all "yum makecache fast && yum install -y epel-release"
# clush -g cl,all "yum install -y R"
# su - koga
$ whoami
koga
```

環境変数を設定します。

```
$ cat >> $HOME/.bash_profile << __EOF__
export R_LIBS_SITE=${R_LIBS_SITE}:${SPARK_HOME}/R/lib
export YARN_CONF_DIR=/opt/hadoop-3.1.0/etc/hadoop
__EOF__

$ clush -a -c $HOME/.bash_profile --dest=$HOME/

$ clush -g cl,all -L "which R && R --version | grep 'R version'"
n0120: /usr/bin/R
```

```
n0120: R version 3.4.3 (2017-11-30) -- "Kite-Eating Tree"
...
```

以上で、SparkR が利用可能になりました。

5-9-2 SparkR で HDFS に保管されたファイルの行数と列数をカウントする

SparkR が利用可能になりましたので、早速、プログラムを作成し、実行してみましょう。例として、先ほどの HDFS に保管したフライトデータ「flight.csv」の行数と列数をカウントするプログラム「rowcol.R」を作成します。

```
$ vi rowcol.R
library(SparkR)  ←1
sc <- sparkR.session(appName="RowCol")  ←2
mydata <- read.df("hdfs://n0121:9000/user/koga/flight.csv",  ←3
 header      = "false",
 delimiter   = ",",  ←4
 source      = "csv",
 inferSchema = "true",
 na.strings  = "")
cache(mydata)
sprintf("Number of row   : %d", nrow(mydata))  ←5
sprintf("Number of column: %d", ncol(mydata))  ←6
```

プログラムの説明：

1. SparkR パッケージを使用
2. SparkR のセッションを開始
3. HDFS の/user/koga ディレクトリに保管された flight.csv ファイルをロード
4. 列の区切り文字を「,」として設定
5. flight.csv ファイルの行数を表示
6. flight.csv ファイルの列数を表示

■ プログラムの実行

事前に Apache Hadoop3 クラスターの HDFS の/user/koga ディレクトリに flight.csv ファイルが存在するかどうかを確認します。

```
$ hdfs dfs -cat /user/koga/flight.csv | head -5
1,5,AA,N723AA,2054,11057,CLT,14100,PHL,1300,1255,0,1438,1435,0,98,449
```

```
1,5,AA,N123AA,2055,11057,CLT,13930,ORD,1455,1452,0,1607,1549,0,132,599
1,5,AA,N123AA,2055,14635,RSW,11057,CLT,1201,1158,0,1357,1343,0,116,600
...
```

spark-submit コマンドを使って、プログラム「rowcol.R」を実行します。

```
$ spark-submit \
--master spark://n0121.jpn.linux.hpe.com:7077 \
./rowcol.R
...
...
[1] "Number of row   : 457892"
[1] "Number of column: 17"
...
```

HDFS の/user/koga ディレクトリに格納された flight.csv は、457892 行、17 列で構成されていることがわかりました。

5-10 Spark MLlib

MLlib（Machine Learning Library）は、Spark の機械学習ライブラリです。もともとは、米カリフォルニア大学バークレー校の機械学習ライブラリ「MLbase」が前身であり、MLlib は、MLbase を Apache Spark 向けに最適化したものです。

MLlib には、さまざまな機械学習用のアルゴリズムが実装されています。一般によく利用される機械学習アルゴリズムやユーティリティが収録されていますが、Apache Spark クラスターでスケールするように設計されています。MLlib では、回帰分析、クラスタリング、分類、協調フィルタリングといった一般的な学習アルゴリズムに加え、特徴抽出、次元削減、線形代数、統計、データ前処理用のライブラリなども含まれます。

Spark MLlib の Web サイトの URL：
https://spark.apache.org/mllib/

5-10-1 機械学習ライブラリの現在

機械学習は、長年培われた人工知能の研究課題の一つです。その名前のとおり、人間と同様に、脳による学習と同じような処理をコンピュータで模倣し、より知的な情報処理を目指すものです。

コンピュータに学習をさせるためには、脳機能の特性をコンピュータのプログラムで実現するための数学モデルを利用します。人間の学習と同様の情報処理を模倣するためのさまざまな数学モデルが存在しますが、有名なものの一つにニューラルネットワークがあります。ニューラルネットワークは、脳の神経細胞の構造とその脳神経の情報伝達の仕組みを模倣したモデルです。ニューラルネットワークを使うことにより、物事の学習、過去の情報からの推定、分類など、人間が脳で行う機能の一部をコンピュータで実現することができます。

このニューラルネットワークを使った学習は、比較的、コンピュータハードウェアの高い計算能力が求められます。また、計算結果を人間にわかりやすく可視化することも必要です。前世紀から情報工学の分野では、非常に盛んに研究が行われていましたが、コンピュータ自体の計算能力が低かったこともあり、なかなか研究成果が上がらなかった分野でもあります。しかし、近年、コンピュータの計算処理能力が劇的に向上し、ニューラルネットワークにおける研究が進み、非常に使いやすいソフトウェアライブラリやニューラルネットワークの処理に適した高速 GPU ボードの登場に伴い、ニューラルネットワークが再び大きな注目を浴びるようになりました（図 5-10）。

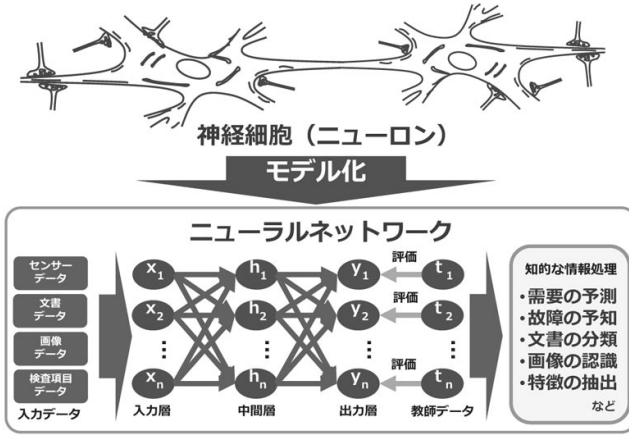


図 5-10　神経細胞とニューラルネットワーク

現在、ニューラルネットワークは、巷で話題となっている自動運転技術、物体認識、専門家の知見と同様の洞察や意思決定を導き出す人工知能システムなどの数学モデルとして利用されています。ニューラルネットワーク以外にもさまざまな機械学習向けの数学モデルがあり、通常は、それらの数学モデルを複数含む「機械学習ライブラリと開発フレームワーク」が提供されています。

Column　spark.mllib から spark.ml へ

Apache Spark には、インメモリでの高速処理を実現するために分散環境で並列実行可能な「データセット」に分割する仕組みが備わっています。この並列実行可能なデータセットは、RDD（Resilient Distributed Dataset）と呼ばれています。従来の MLlib は、この RDD ベースの API でしたが、現在は、よりデータの取り扱いが簡単になった DataFrame と呼ばれる API がプライマリとなっています。

Apache Spark のバージョン 2.0 以降、Spark MLlib の RDD ベースの API は、すでにメンテナンスモードになっています。従来の RDD API ベースの Spark MLlib は、「spark.mllib」と呼ばれます。spark.mllib は、プログラム内で、「`org.apache.spark.mllib`」パッケージをインポートすることで利用可能ですが、今後、RDD ベースの API は、廃止が予定されています。

一方、DataFrame API ベースの Spark MLlib は、spark.ml と呼ばれます。プログラム内で、「`org.apache.spark.ml`」パッケージをインポートすることで利用可能です。また、spark.ml では、ML パイプラインと呼ばれる連続した操作を 1 つにまとめる仕組みがあります。たとえば、Hadoop クラスターが提供する、YARN によるデータの前処理とモデル作成の 2 つの処理そのものを、プログラム内でインスタンスとして取り扱うことができるようになります。これにより、従来に比べて、プログラムのソースコードの簡素化が期待できます。ML パイプラインを使った機械学習の Scala プログラムは、Spark の tar アーカイブにも含まれていますし、以下の URL からも入手可能です。

DataFrame API ベースの「spark.ml」を使った Scala プログラムの入手先 URL：
`https://github.com/apache/spark/tree/master/examples/src/main/scala/org/apache/spark/examples/ml`

5-10-2 Spark MLlib による迷惑メールの分類

以下では、機械学習ライブラリの Spark MLlib を Apache Hadoop 3 クラスターで利用する方法について解説します。ここでは、具体例として Spark MLlib を使った迷惑メールの分類を取り上げます。

■ データの取得

まずは、クライアントノードにおいて、作業用の `spam-detect` ディレクトリを作成します。

```
$ hostname
n0120.jpn.linux.hpe.com
```

```
$ whoami
koga

$ mkdir /home/koga/spam-detect
$ cd /home/koga/spam-detect
```

今回は、米国カリフォルニア大学アーバイン校（University of California, Irvine、以下、UCI）が提供する「UCI マシンラーニングリポジトリ」からデータを取得します。UCI マシンラーニングリポジトリは、研究開発用のための機械学習のデータセットを大量に提供しています。取得するのは、スパムメール分類に使われるデータ「smsspamcollection.zip」です。wget コマンドでクライアントノードにダウンロードします。

```
$ wget \
https://archive.ics.uci.edu/ml/machine-learning-databases/00228\
/smsspamcollection.zip
```

zip 圧縮されているため、解凍します。

```
$ unzip smsspamcollection.zip
$ ls -lh SMSSpamCollection
-rw-r--r-- 1 koga hadoop 467K Mar 15  2011 SMSSpamCollection
```

入手したデータの中身を確認します。中身を確認したら、HDFS の/user/koga ディレクトリにコピーします。

```
$ less SMSSpamCollection
ham     Go until jurong point, crazy.. Available only in bugis n great world la
 e buffet... Cine there got amore wat...
ham     Ok lar... Joking wif u oni...
spam    Free entry in 2 a wkly comp to win FA Cup final tkts 21st May 2005. Tex
t FA to 87121 to receive entry question(std txt rate)T&C's apply 08452810075ove
r18's
ham     U dun say so early hor... U c already then say...
ham     Nah I don't think he goes to usf, he lives around here though
...

$ hadoop fs -put SMSSpamCollection /user/koga/
```

■ ロジスティック回帰を使った迷惑メール分類

今回は、分類モデルとして、Spark MLlib が提供するロジスティック回帰を使って、迷惑メールを分類する Scala アプリケーションを作成します。

```
$ vi spam-detect.scala
import org.apache.spark.mllib.feature.HashingTF
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.classification.LogisticRegressionWithSGD
val f    = sc.textFile("hdfs://n0121:9000/user/koga/SMSSpamCollection")  ←1
val spm  = f.filter(L => L.contains("spam")).map(x => x.split("\t")(1))  ←2
val ham  = f.filter(L => L.contains("ham" )).map(x => x.split("\t")(1))  ←3
var ftr  = new HashingTF(numFeatures = 100)
val SPM  = spm.map(txt => ftr.transform(txt.split("")))
val HAM  = ham.map(txt => ftr.transform(txt.split("")))
val pos  = SPM.map(features => LabeledPoint(1,features))  ←4
val neg  = HAM.map(features => LabeledPoint(0,features))  ←5
val dat  = pos.union(neg)
val Array(trn, tst) = dat.randomSplit(Array(0.7, 0.3))  ←6
val LR   = new LogisticRegressionWithSGD()
val mod  = LR.run(trn)  ←7
val prd  = tst.map(x => (mod.predict(x.features), x.label))  ←8
val ans  = prd.filter(r => r._1 == r._2).count.toDouble / tst.count  ←9
println("迷惑メール判定：" + (ans * 100) + "パーセント")
sys.exit
```

プログラムの説明：

1. HDFS に保管した分析対象のファイル「SMSpamCollection」からデータセットを作成
2. 分析対象のファイル内で文字列「spam」が含まれている行を取得
3. 分析対象のファイル内で文字列「ham」が含まれている行を取得
4. 「spam」には、ラベル 1 を付与
5. 「ham」には、ラベル 0 を付与
6. データの 7 割は、学習用、残りの 3 割は、テスト用で使用
7. ロジスティック回帰を使ってモデルを作成
8. 作成したモデルから推定
9. 迷惑メール判定の正確性の割合を計算

■ アプリケーションの実行

spark-shell コマンドを使ってアプリケーションを実行します。

```
$ which spark-shell
~/spark-2.3.0-bin-hadoop2.7/bin/spark-shell
```

```
$ spark-shell -i ./spam-detect.scala
...
ans: Double = 0.9607163489312536
迷惑メール判定：96.07163489312536パーセント
...
```

ロジスティック回帰を使って、96 %以上の割合で迷惑メールを分類できたという結果が得られました。

■ 単純ベイズ分類器を使った迷惑メール分類

先ほどは、ロジスティック回帰を使って迷惑メールを判定しましたが、別の分類器を使ってみます。以下は、単純ベイズ分類器（Naive Bayes classifier）を使った迷惑メールを分類するアプリケーションです。

```
$ vi spam-detect-naivebayes.scala
import org.apache.spark.mllib.feature.HashingTF
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.classification.{NaiveBayes, NaiveBayesModel}  ←1
val f    = sc.textFile("hdfs://n0121:9000/user/koga/SMSSpamCollection")
val spm  = f.filter(L => L.contains("spam")).map(x => x.split("\t")(1))
val ham  = f.filter(L => L.contains("ham" )).map(x => x.split("\t")(1))
var ftr  = new HashingTF(numFeatures = 100)
val SPM  = spm.map(txt => ftr.transform(txt.split("")))
val HAM  = ham.map(txt => ftr.transform(txt.split("")))
val pos  = SPM.map(features => LabeledPoint(1,features))
val neg  = HAM.map(features => LabeledPoint(0,features))
val dat  = pos.union(neg)
val Array(trn, tst) = dat.randomSplit(Array(0.7, 0.3))
val mod  = NaiveBayes.train(trn,lambda = 1.0, modelType = "multinomial")  ←2
val prd  = tst.map(x => (mod.predict(x.features), x.label))
val ans  = prd.filter(r => r._1 == r._2).count.toDouble / tst.count
println("迷惑メール判定：" + (ans * 100) + "パーセント")
sys.exit
```

プログラムの説明：

1 Spark MLlib が提供する単純ベイズ分類器をインポート

2 単純ベイズ分類器を使ってモデルを生成

ロジスティック回帰分析の場合と比べて、インポートする分類器（1）と、モデルの作成の処理（2）が異なることがわかります。

■ プログラムの実行

spark-shell コマンドを使ってアプリケーションを実行します。

```
$ spark-shell -i ./spam-detect-naivebayes.scala
...
ans: Double = 0.9485680888369374
迷惑メール判定：94.85680888369374パーセント
...
```

5-11　ニューラルネットワークによる学習

Apache Spark には、ニューラルネットワークの多層パーセプトロン分類器（MultiLayer Perceptron Classifier）（以下、MLPC）が含まれており、機械学習を行うことが可能です。以下では、Apache Spark に付属のニューラルネットワークの分類器を使ったデータの分類を行うプログラムの解説と実行法を解説します。

5-11-1 ニューラルネットワークによるアヤメの分類

Apache Spark には、ニューラルネットワークに学習させることが可能なサンプルとなる入力データが用意されています。サンプルデータのうち、今回は、機械学習の入門用として利用されている「**植物のアヤメの分類**」[*1]に挑戦します。サンプルデータに含まれるアヤメは、「Iris Setosa」「Iris Versicolor」「Iris Virginica」の 3 種類を対象としたもので、データとしては、花の種別、がく片の長さ、がく片の幅、花びらの長さ、花びらの幅、という 4 つの特徴量を持ちます。花の種別は、0 から 3 までのラベルが振られます。特徴量となる長さと幅は、Apache Spark が提供する LIBSVM（サポート・ベクター・マシン用ライブラリ）[*2]で取り扱えるように、-1 から 1 までの値に正規化したものを利用します。これらの花のデータは、150 個用意されており、ニューラルネットワークによる学習では、そのうちの大部分を学習用に利用し、残りのデータを分類のテスト用に利用します。正規化された値を含むデータセットは、Apache Spark 内に

＊1　アヤメのデータに関する情報源：
https://en.wikipedia.org/wiki/Iris_flower_data_set

＊2　LIBSVM は、国立台湾大学で開発された機械学習用のライブラリです。LIBSVM で利用可能な正規化済みのデータセットは、国立台湾大学の Web サイトから入手可能です。
LIBSVM で利用可能な正規化済みデータセットの入手先：
https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/

「sample_multiclass_classification_data.txt」として含まれています。

正規化前のデータの入手先：

http://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data

正規化後のデータの入手先（インターネット経由）：

https://raw.githubusercontent.com/apache/spark/master/data/mllib/sample_multiclass_classification_data.txt

正規化後のデータの入手先（Apche Spark の tarball 内、/opt ディレクトリ以下に展開した場合）：

/opt/spark-2.3.0-bin-hadoop2.7/data/mllib/sample_multiclass_classification_data.txt

5-11-2 Apache Spark に含まれる多層パーセプトロン分類器

多層パーセプトロン分類器（MLPC）は、順伝播型ニューラルネットワーク（Feed-Forward Neural Networks）（以下、FFNN）を構成し、入力された大量のデータを学習し、分類します。MLPC は、複数の層から構成されます。入力層からデータが入力され、各層は、ニューラルネットワーク内において、次の層にシナプスによって接続されており、データが伝播します（図 5-11）。

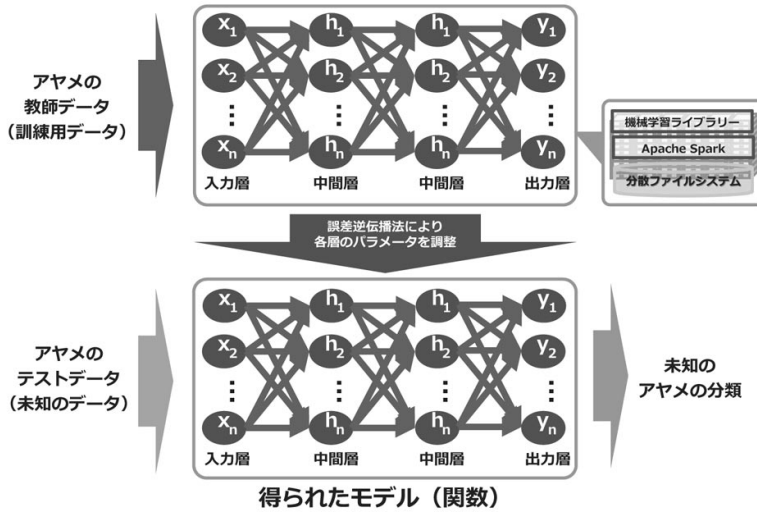


図 5-11　ニューラルネットワークにおける教師あり学習

ニューラルネットワークでは、モデルの学習に「**誤差逆伝播法**」と呼ばれる教師あり学習が用いられます。教師あり学習とは、「**教師データ**」と呼ばれる、事前に与えた学習用訓練データを使った機械学習の一手法です。学習とは、与えられたデータに対して、最もフィットする「**関数**」を見つけることです。この「関数」を求めることができれば、新たに用意した未知のテスト用データに対して、求めた「関数」を使って分類することができます。

■ アヤメのデータを HDFS に格納

では、実際に、Apache Spark に含まれているアヤメのデータを使ってニューラルネットワークによる学習と分類を行います。まず、入力データは、HDFS の/user/koga ディレクトリに保存しておきます。

```
$ hostname
n0120.jpn.linux.hpe.com

$ printenv SPARK_HOME
/opt/spark-2.3.0-bin-hadoop2.7

$ hdfs dfs -put \
$SPARK_HOME/data/mllib/sample_multiclass_classification_data.txt \
/user/koga/
```

■ ニューラルネットワークの Scala プログラム「mlp.scala」の作成

HDFS の/user/koga ディレクトリに保存したアヤメの入力データを使って、ニューラルネットワークにより学習を行い、アヤメを分類（種類の同定）する Scala プログラム「mlp.scala」を作成します。mlp.scala では、Apache Spark に搭載されている多層パーセプトロン分類器を使います。今回は、入力データの 70%で学習を行い、残りの 30%でどれほど正確に分類できるかをテストします。

```
$ vi mlp.scala
import org.apache.spark.ml.classification.MultilayerPerceptronClassifier  ←1
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator  ←2
val PATH  = "hdfs://n0121:9000/user/koga/sample_multiclass_classification_data.
txt"  ←3   (1 行入力)
val data  = spark.read.format("libsvm").load(PATH)  ←4
val SEED  = System.currentTimeMillis()  ←5
```

```
val split = data.randomSplit(Array(0.7, 0.3), seed = SEED)  ←6
val trn   = split(0)  ←7
val tst   = split(1)  ←8
val lyr   = Array[Int](4, 5, 4, 3)  ←9
val mlp   = new MultilayerPerceptronClassifier().setLayers(lyr).setBlockSize(11
0).setSeed(SEED).setMaxIter(145)  ←10  (1行入力)
val mod   = mlp.fit(trn)  ←11
val ans   = mod.transform(tst)  ←12
val prdct = ans.select("prediction", "label")  ←13
val eval  = new MulticlassClassificationEvaluator().setMetricName("accuracy")14
println("Accuracy:" + eval.evaluate(prdct))  ←15
ans.show(false)
sys.exit
```

プログラムの説明：

1 多層パーセプトロン分類器をインポート
2 多クラス分類器をインポート
3 HDFS 上のデータのパスを設定
4 HDFS 上のデータをロード
5 現在時刻から乱数の種となる値を設定
6 HDFS 上のデータを 70%（教師データ）と 30%（テストデータ）に分割
7 教師データを用意
8 テストデータを用意
9 入力層が 4 ノード、隠れ層が 5 ノード、その次の隠れ層が 4 ノード、出力層が 3 ノードのニューラルネットワークを形成
10 多層パーセプトロン分類器を用意
11 教師データをニューラルネットワークに入力し、学習を実行し、モデル（関数）を生成
12 生成したモデル（関数）にテストデータを入力し、種類を推定
13 得られた結果から推定値とラベルを選択
14 多クラス分類器のメトリックをセット
15 アヤメの種類の同定結果の正誤割合を表示

Note　ニューラルネットワーク層の変更

Apache Spark では、ニューラルネットワークのノード数だけでなく、層の数も変更できます。以下は、Scala プログラムにおいて、入力層（ノード数が 4）、中間層（ノード数が 5）、出力層（ノード数が 4）の 3 層から構成されるニューラルネットワークを利用する例です。

```
val lyr   = Array[Int](4, 5, 4)
val mlp   = new MultilayerPerceptronClassifier().setLayers(lyr).setBlock⇒
Size(110).setSeed(SEED).setMaxIter(145)
```

■ Scala プログラム「mlp.scala」の実行

Spark シェルで Scala プログラム「mlp.scala」を実行します。正確に分類できた割合が結果として表示されます。

```
$ spark-shell \
--master spark://n0121:7077 \
--executor-memory 1g \
--driver-memory 2g \
-i ./mlp.scala
...
Accuracy:0.9642857142857143
...
```

上記は、教師データによって学習したニューラルネットワークのモデル（関数）を作成し、そのモデルに未知のアヤメのテストデータを入力すると、約 96.4%のデータにおいて、アヤメ種の分類に成功したことを意味します。

5-11-3 Python プログラム「mlp.py」の作成

最近は、Python による機械学習プログラミングが国内外を問わず人気です。Apache Spark でも、Python による機械学習のプログラミングが可能です。以下では、先ほどのアヤメの分類学習プログラムを Python 言語で作成し、実行してみましょう。以下の Python プログラムのアルゴリズムは、Scala プログラムのものと同じです。

```
$ hostname
n0120.jpn.linux.hpe.com

$ vi mlp.py
from __future__ import print_function
from pyspark.ml.classification import MultilayerPerceptronClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.sql import SparkSession
import time
if __name__ == "__main__":
    spark = SparkSession.builder.appName("MyMLP").getOrCreate()
    PATH  = "hdfs://n0121:9000/user/koga/sample_multiclass_classification_dat
a.txt"   (1 行入力)
    data  = spark.read.format("libsvm").load(PATH)
    SEED  = int(time.time() * 1000)
```

```
    split = data.randomSplit([0.7, 0.3], SEED)
    trn   = split[0]
    tst   = split[1]
    lyr   = [4, 5, 4, 3]
    mlp   = MultilayerPerceptronClassifier(maxIter=145, layers=lyr, blockSize=1
10, seed=SEED)  (1行入力)
    mod   = mlp.fit(trn)
    ans   = mod.transform(tst)
    prdct = ans.select("prediction", "label")
    eval  = MulticlassClassificationEvaluator(metricName="accuracy")
    print("Accuracy: " + str(eval.evaluate(prdct)))
    ans.show()
    spark.stop()
```

■ Python プログラムの実行

Apache Spark が提供する「spark-submit」コマンドで Python プログラムを実行します。

```
$ spark-submit --master spark://n0121:7077 ./mlp.py
...
Accuracy: 0.980392156863
...
```

以上で、Apache Spark の多層パーセプトロン分類器による学習と未知のデータの分類ができました。一般に、ニューラルネットワークでは、巨大な行列計算を行うことが多く、かなりの計算量を必要とします。近年は、スーパーコンピュータなどの High Performance Computing（HPC）の世界において、ニューラルネットワークの中間層を多く持つ「**深層学習**」が主流になっています。機械学習、特に、深層学習などのニューラルネットワークに代表される知識工学の手法は、画像認識、知的ロボットの制御、自動運転などに不可欠なものとして、現在世界中で研究が加速しています。

Spark クラスターでは、本書で紹介したように、従来の C 言語や FORTRAN 言語によるプログラミングに比べて、ニューラルネットワークのプログラミングのハードルは、非常に低くなっています。まずは、本書で紹介したニューラルネットワークの基本的なプログラミングを行い、多層パーセプトロン分類器のプログラムを実行し、機械学習の基本を学んでみてください。そして、ぜひ、深層学習などのライブラリを組み合わせることで、Spark クラスターによる次世代型の機械学習基盤を手に入れてみてください。

5-12 まとめ

本章では、Apache Spark の概要、構築手順、そして、各コンポーネントについて簡単に紹介しました。データが次々と生成されるリアルタイムの分析では、Hadoop システムへのビッグデータの保管に加え、ストリーミング処理、グラフ処理、機械学習ライブラリによる分類、推定が欠かせません。Apache Spark は、これらの一連のデータ処理を可能にするフレームワークを提供します。今までは、複雑で膨大なプログラミングが必要だった知的情報処理基盤が、Apache Spark の登場により、比較的簡単に構築・利用できるようになりました。ぜひ、Apache Spark と Hadoop をベースにした知的情報処理基盤を構築し、インメモリによる超高速データ分析を体験してみてください。

第6章
Hive/Impala/HBase/Pig – データベースの操作

Hadoop 登場以前のデータ分析基盤では、主に RDBMS が使われており、今でもアプリケーションのバックエンドや IT 部門における日常のデータ管理に広く利用されています。RDBMS が大規模化するとともに、Hadoop 基盤においても SQL によるデータの処理が求められるようになり、エコシステムとして、SQL に似た操作を行える Apache Hive、Apache Impala などが登場してきました。また、最近では、データベースそのものも従来の RDBMS とは異なる「列指向型データベース」（キーバリュー）が利用されるようになるとともに、そうしたフレームワークも整備されており、現在では、非常に多くの事例で利用されています。

本章では、Hadoop クラスターにおける SQL/NoSQL データベースの操作が可能なエコシステムとして、Apache Hive、Apache Impala に加え、Apache HBase、そしてデータ操作のためのスクリプト言語の Apache Pig を紹介します。

6-1 Apache Hive

Hadoop クラスターの大規模データを SQL に似た操作[*1]で扱えるソフトウェアの一つに Apache Hive があります。Apache Hive は、Hadoop 分散ファイルシステム上に保管されているデータの抽出や変換を行います。ユーザーは、HiveQL（以下 HQL）と呼ばれるデータ問い合わせ言語（クエリー言語）を使ってデータを操作します。HQL は、従来の RDBMS で利用されている SQL に似ており、SQL に精通しているユーザーであれば、学習コストをかけずに Hive を利用できるため、Hadoop クラスターでよく利用されているソフトウェアの一つです。

Apache Hive は、Hadoop のその他の周辺ソフトウェアと組み合わせて利用可能です。Java Database Connectivity（JDBC）を使って、Java から Hive にアクセスすることや、Python などから利用することも可能です。また、手続き型言語である Apache Pig と組み合わせて利用する事例も少なくありません。

一般に、RDBMS で利用される SQL は、耐障害性、整合性、オンラインリアルタイムのクエリー処理が可能ですが、取り扱えるデータサイズの大きさが限られています。一方、Apache Hive で利用される HQL は、リアルタイムというよりは、むしろバッチ処理系で利用されることが少なくありません。データサイズは、Hadoop のスケールアウトメリットをそのまま享受できるため、ペタバイト級のデータも取り扱えます（図 6-1）。

表 6-1 SQL と HiveQL の比較

	一般的な SQL	HiveQL
稼働環境	RDBMS	Hadoop 分散ファイルシステム
クエリー操作	主にオンラインリアルタイム処理	主にバッチ処理
取り扱うデータサイズ	ギガバイト級	ペタバイト級

6-1-1 Apache Hive のコンポーネント

Apache Hive は、以下に示す複数のサービスで構成されます。

●MetaStore

MetaStore サービスは、Hadoop 分散ファイルシステム上のデータを表のように見せるフィルタの役目を担います。この MetaStore は、通常、RDBMS で構築します。ユーザーは、HQL

＊1 HQL は、SQL に似た構文でデータを操作できますが、ANSI SQL ではありません。

として、hive コマンド、あるいは、beeline コマンドを使って分散ファイルシステム上のデータにアクセスします。

●HiveServer

hive コマンドは、Hive の初期のバージョンから実装されている HQL ですが、後に beeline コマンドが登場しました。ユーザーの hive コマンドの命令は、HiveServer サービスが処理を担当します。

●HiveServer2

beeline コマンドの命令を処理するのは、HiveServer2 サービスです。現在は、新しい beeline コマンド、および、HiveServer2 サービスの利用が推奨されています。

●WebHCat

WebHCat サービスは、クライアントノードから curl コマンドなどの REST API を経由して Hive を操作するためのサービスです。

Hive は、Hadoop クラスター側が提供する MapReduce や Tez と呼ばれる分散処理の仕組み（フレームワーク）を使ってデータの操作を行います。さらに、ODBC、および、JDBC の API に対応しているため、サードパーティ製の可視化ツールや分析ツールなどが Hive 経由で分散ファイルシステム上のデータを取り扱うことが可能です（図 6-1）。

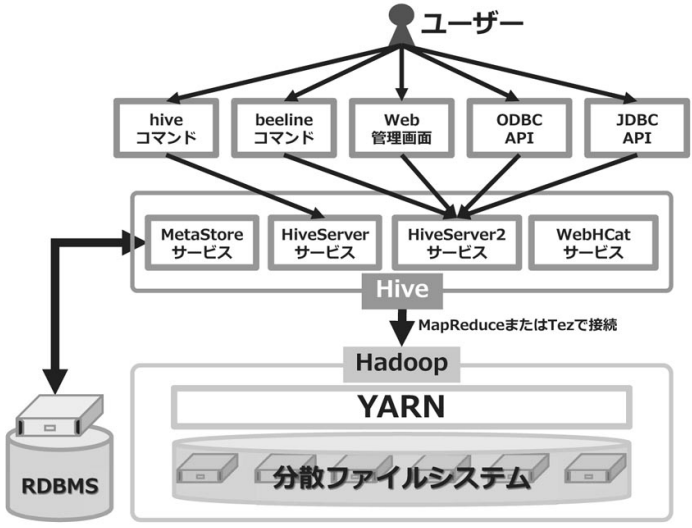


図 6-1　Apache Hive のコンポーネント

6-1-2 MapR クラスターへの Apache Hive のインストール

以下では、Apache Hive を MapR クラスターにインストールし、利用する手順を紹介します。今回、Apache Hive の MetaStore サービスの格納先は、RDBMS の MariaDB とします。MetaStore サービスの格納先である RDBMS は、MariaDB 以外にも、MySQL、PostgreSQL などで構成可能です。また今回の MetaStore サービスは、Hadoop クラスターノードの n0131.jpn.linux.hpe.com に構築します (図 6-2)。

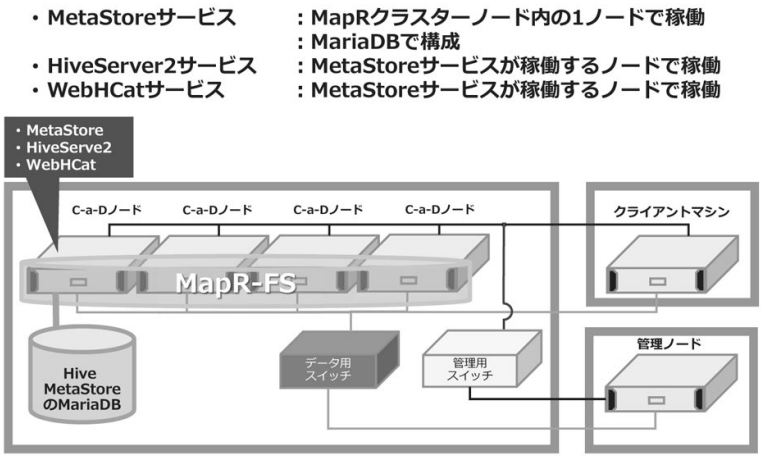


図 6-2　今回構築する Apache Hive のシステム構成

■ MEP リポジトリの確認

まず、MapR クラスターの全ノードに MapR が提供する MEP（MapR Expansion Pack）のリポジトリが正しく設定されているかを確認します。

```
# hostname
n0130.jpn.linux.hpe.com

# clush -g cl,all -L \
"grep -A 6 maprecosystem /etc/yum.repos.d/maprtech.repo"
n0130: [maprecosystem]
n0130: name=MapR Technologies
n0130: baseurl=http://package.mapr.com/releases/MEP/MEP-5.0.0/redhat/
```

```
n0130: enabled=1
n0130: gpgcheck=0
n0130: protect=1
...
```

■ 環境変数の設定

ユーザーのホームディレクトリの「.bash_profile」ファイルにJavaの環境変数「JAVA_HOME」と、Hiveの環境変数「HIVE_HOME」を記述します。今回は、rootユーザーとmaprユーザーに対して環境変数を記述します。

```
# clush -g cl,all "cat >> /root/.bash_profile << '__EOF__'
export JAVA_HOME=/usr/lib/jvm/jre
export HIVE_HOME=/opt/mapr/hive/hive-2.1
export PATH=\$HIVE_HOME/bin:\$PATH
__EOF__"

# clush -g cl,all "cat >> /home/mapr/.bash_profile << '__EOF__'
export JAVA_HOME=/usr/lib/jvm/jre
export HIVE_HOME=/opt/mapr/hive/hive-2.1
export PATH=\$HIVE_HOME/bin:\$PATH
__EOF__"
```

Note　環境変数 JAVA_HOME

環境変数「JAVA_HOME」には、事前にインストールしたJavaのディレクトリパスを指定する必要があります。本書では、/usr/lib/jvm/jreを指定していますが、これは、/etc/alternatives/jreへのシンボリックリンクになっています。さらに、/etc/alternatives/jreもシンボリックリンクになっており、CentOS 7系のjava-1.8.0-openjdk-headless RPMパッケージの場合、jreディレクトリの実体は、以下のディレクトリになります。

/usr/lib/jvm/java-1.8.0-openjdk-<バージョン番号>.el7_4.x86_64/jre

記述した環境変数を確認します。

```
# clush -g cl,all -L \
"grep -E 'JAVA_HOME|HIVE_HOME' /root/.bash_profile"
```

```
n0130: export JAVA_HOME=/usr/lib/jvm/jre
n0130: export HIVE_HOME=/opt/mapr/hive/hive-2.1
n0130: export PATH=$HIVE_HOME/bin:$PATH
...

# clush -g cl,all -L \
"grep -E 'JAVA_HOME|HIVE_HOME' /home/mapr/.bash_profile"
```

ユーザー mapr の.bash_profile の所有者を mapr、所有グループを mapr に設定します。

```
# clush -g cl,all "chown mapr:mapr /home/mapr/.bash_profile"
```

■ Hive のインストール

Hive のパッケージ mapr-hive をクライアントノードと全クラスターノードにインストールします。

```
# clush -g cl,all "yum makecache fast && yum install -y mapr-hive"
```

インストールした Hive のバージョンを確認します。

```
# clush -g cl,all "ls /opt/mapr/hive/"
n0133: hive-2.1
...
```

Hive 関連の設定ファイルをコピーします。

```
# clush -gl,all \
"cp -a /opt/mapr/hive/hive-2.1/conf.new/* \
/opt/mapr/hive/hive-2.1/conf/"
```

今回は、HiveServer2、HiveMetaStore、HiveWebhcat を n0131.jpn.linux.hpe.com のみにインストールします。

```
# clush -w n0131 \
"yum install -y \
mapr-hiveserver2 \
mapr-hivemetastore \
mapr-hivewebhcat"
```

Hive クラスターの構成を更新します。

```
# clush -a "/opt/mapr/server/configure.sh -R"
```

■ MariaDB のインストール

クラスターノードの n0131.jpn.linux.hpe.com に、MetaStore サービスのための RDBMS サーバーを構築します。MariaDB の設定において、一部、対話形式による入力があるため、MariaDB サーバーとなる n0131.jpn.linux.hpe.com にログインします。

```
# ssh n0131-mgm

# hostname
n0131.jpn.linux.hpe.com
```

MariaDB のリポジトリの設定ファイルを記述します。今回は、MariaDB のバージョンは、10.3.5 を採用しました。

```
# cat > /etc/yum.repos.d/mariadb.repo << __EOF__
[mariadb]
character-set-server=utf8
name = MariaDB
baseurl = http://yum.mariadb.org/10.3.5/centos7-amd64
gpgkey=https://yum.mariadb.org/RPM-GPG-KEY-MariaDB
gpgcheck=1
enabled=1
__EOF__
```

MariaDB の RPM パッケージの GPG キーをインポートします。自社のプロキシーサーバー経由でインターネットにアクセスする環境では、事前に環境変数の `https_proxy` を設定しておきます。

```
# export https_proxy=http://proxy.your.site.com:8080
# rpm --import https://yum.mariadb.org/RPM-GPG-KEY-MariaDB
```

MariaDB の RPM パッケージをインストールします。

```
# yum install -y MariaDB-client MariaDB-server
# rpm -qa | grep MariaDB
MariaDB-common-10.3.5-1.el7.centos.x86_64
MariaDB-compat-10.3.5-1.el7.centos.x86_64
MariaDB-client-10.3.5-1.el7.centos.x86_64
MariaDB-server-10.3.5-1.el7.centos.x86_64
```

MariaDB の設定ファイルを作成します。

```
# cp /etc/my.cnf /etc/my.cnf.org
# cat > /etc/my.cnf << __EOF__
[mysqld]
datadir=/var/lib/mysql
socket=/var/lib/mysql/mysql.sock
symbolic-links=0
[mysqld_safe]
log-error=/var/log/mariadb/mariadb.log
pid-file=/var/run/mariadb/mariadb.pid
character-set-server=utf8
!includedir /etc/my.cnf.d
__EOF__
```

MariaDB のデータベースをインストールします。/var/lib/mysql ディレクトリ以下に、データベースの実体が生成されます。

```
# mysql_install_db
Installing MariaDB/MySQL system tables in '/var/lib/mysql' ...
OK
...
```

データベースの実体が保管される/var/lib/mysql ディレクトリ配下のすべてのファイルとディレクトリの所有者と所有グループを mysql に設定します。

```
# chown -R mysql:mysql /var/lib/mysql
```

MariaDB データベースを起動します。

```
# systemctl start mariadb
# systemctl status mariadb | grep Active
   Active: active (running) since Wed 2018-02-28 17:06:12 JST; 4s ago

# systemctl enable mariadb
```

MariaDB の初期設定は、mysql_secure_installation コマンドで行います。質問に回答する対話形式で設定します。

```
# mysql_secure_installation
...
Enter current password for root (enter for none):  ←そのまま ENTER キーを入力
...
```

```
Set root password? [Y/n] Y  ←Yを入力
New password:   ←今回は、MariaDBのrootユーザーのパスワードをpassword1234に設定
Re-enter new password: password1234  ←再度パスワードを入力
...
Remove anonymous users? [Y/n] Y  ←匿名ユーザーを削除するためYを入力
...
Disallow root login remotely? [Y/n] Y  ←遠隔からrootでの接続を拒否するためYを入力
...
Remove test database and access to it? [Y/n] Y  ←test DBを削除するためYを入力
...
Reload privilege tables now? [Y/n] Y  ←特権テーブルをリロード
...
Thanks for using MariaDB!
```

mysqlコマンドを使って、MariaDBへの接続を確認します。「-p」オプションには、先ほどのmysql_secure_installationコマンドの対話形式の設定で入力したパスワードを指定します。

```
# mysql -uroot -ppassword1234 -e "show databases;"
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
+--------------------+
```

以上でMariaDBが利用できるようになりました。

■ MySQLコネクタのインストール

HiveからMariaDBに接続するためのMySQLコネクタ「mysql-connector-java.jar」をインストールします。

```
# hostname
n0131.jpn.linux.hpe.com

# yum install -y mysql-connector-java
# ln -s \
/usr/share/java/mysql-connector-java.jar \
/opt/mapr/hive/hive-2.1/lib/mysql-connector-java.jar
```

■ データベース「metastore」と Hive ユーザー「hive」の作成

MariaDB のデータベース「metastore」と Hive ユーザー「hive」を作成する SQL 文をテキストファイルとして作成します。今回、データベースにアクセスできる hive ユーザーの IP アドレスは、データ用 LAN の「10.0.0.0/24」に限定します。

```
# cat > sql-for-hive.sql << __EOF__
CREATE DATABASE metastore;
USE             metastore;
CREATE USER 'hive'@'10.0.0.%' IDENTIFIED BY 'password1234';
REVOKE ALL PRIVILEGES, GRANT OPTION FROM 'hive'@'10.0.0.%';
GRANT  ALL PRIVILEGES  ON metastore.* TO 'hive'@'10.0.0.%';
FLUSH      PRIVILEGES;
__EOF__
```

作成した SQL 文を MariaDB にロードします。

```
# mysql -uroot -ppassword1234 < ./sql-for-hive.sql
# exit
```

Column　SQL 文の記述ファイル

データベースの作成やユーザーの作成を行う SQL 文以外にも、ユーザーとデータベースの削除を行う SQL 文（drop.sql）も合わせて用意しておくと便利です。MySQL や MariaDB では、DROP 命令により、データベースやユーザーを削除できます。SQL 文は、人間が手動で入力することもありますが、作業工程を後から見直すことも考慮し、sql-for-hive.sql や drop.sql のように、一連の SQL 文を記述したテキストファイルをデータベースソフトウェアにロードするのがよいでしょう。

```
# cat > drop.sql << __EOF__
DROP DATABASE metastore;
DROP USER 'hive'@'10.0.0.%';
__EOF__

# mysql -uroot -ppassword1234 < ./drop.sql
```

■ hive-site.xml ファイルの設定

Hive の設定の中核をなすファイルが hive-site.xml です。hive-site.xml に指定できるパラメーターは、膨大な数ですが、必要最低限の設定手順を以下に示します。まず、全ノードの/opt/mapr/hive/hive-2.1/conf ディレクトリにファイルが格納されていることを確認します。以下、クライアントノードで作業します。

```
# hostname
n0130.jpn.linux.hpe.com

# clush -g cl,all -L "ls /opt/mapr/hive/hive-2.1/conf/"
n0130: beeline-log4j2.properties.template
n0130: hive-default.xml.template
n0130: hive-env.sh.template
n0130: hive-exec-log4j2.properties.template
n0130: hive-log4j2.properties.template
n0130: hive-site.xml
n0130: hplsql-site.xml
n0130: ivysettings.xml
n0130: llap-cli-log4j2.properties.template
n0130: llap-daemon-log4j2.properties.template
n0130: parquet-logging.properties
n0130: warden.hcat.conf
n0130: warden.hivemeta.conf
n0130: warden.hs2.conf
...
```

テンプレートの hive-default.xml.template ファイルを hive-site.xml という名前でコピーを作成します。また、hive-site.xml ファイルのオリジナルを残しておきます。

```
# clush -g cl,all \
"cp -a \
/opt/mapr/hive/hive-2.1/conf/hive-default.xml.template \
/opt/mapr/hive/hive-2.1/conf/hive-site.xml"

# clush -g cl,all \
"cp /opt/mapr/hive/hive-2.1/conf/hive-site.xml \
/opt/mapr/hive/hive-2.1/conf/hive-site.xml.org"
```

今回、hive-site.xml ファイル内で変更するパラメーターは、**表 6-2** のとおりです。

表 6-2　hive-site.xml ファイル内で変更するパラメーター

hive-site.xml 内の主なパラメーター	説明
javax.jdo.option.ConnectionURL	接続先となる MetaStore サーバーを指定
javax.jdo.option.ConnectionDriverName	JDBC ドライバの指定
javax.jdo.option.ConnectionUserName	JDBC 経由で接続するユーザー名を指定
javax.jdo.option.ConnectionPassword	JDBC 経由で接続するユーザーパスワードを指定
hive.metastore.uris	MetaStore の URI を指定
system:java.io.tmpdir	一時ファイルを格納するディレクトリ
system:user.name	Hive を実行するユーザー名

hive-site.xml ファイルを編集します。表 6-2 に示したパラメーターにおいて、「system:java.io.tmpdir」と「system:user.name」は、hive-site.xml ファイルの最下行の「</configuration>」の前に追記します。この 2 つについては、他のパラメーターと異なり、hive-site.xml ファイル内において、新規に追加で記述する形になるため、「<property>」と「</property>」を忘れずに記述してください。

```
# vi /opt/mapr/hive/hive-2.1/conf/hive-site.xml
...
    <name>javax.jdo.option.ConnectionURL</name>
    <value>jdbc:mysql://n0131.jpn.linux.hpe.com/metastore</value>
...
    <name>javax.jdo.option.ConnectionDriverName</name>
    <value>com.mysql.jdbc.Driver</value>
...
    <name>javax.jdo.option.ConnectionUserName</name>
    <value>hive</value>
...
    <name>javax.jdo.option.ConnectionPassword</name>
    <value>password1234</value>
...
    <name>hive.metastore.uris</name>
    <value>thrift://n0131.jpn.linux.hpe.com:9083/</value>
...
  <property>
    <name>system:java.io.tmpdir</name>
    <value>/tmp/hive/java</value>
  </property>
  <property>
    <name>system:user.name</name>
    <value>${user.name}</value>
```

```
  </property>
...
```

■ /tmp/hive/java ディレクトリのパーミッションの設定

hive-site.xml ファイルの「system:java.io.tmpdir」にパラメーターとして設定した/tmp/hive/java ディレクトリを全ノードに作成し、一般ユーザーの書き込み権限† を付与します。

```
# clush -g cl,all "mkdir -p /tmp/hive/java"
# clush -g cl,all "chmod 1777 /tmp/hive/java"
```

> † パーミッションにかかわらず、ファイルの所有者のみが読み書きできるようにするため、chmod コマンドには、スティッキービットを付与した「1777」を指定します。

■ hive-site.xml ファイルのコピー

編集した hive-site.xml ファイルを全ノードにコピーします。

```
# clush -a -c \
/opt/mapr/hive/hive-2.1/conf/hive-site.xml \
--dest=/opt/mapr/hive/hive-2.1/conf/
```

以上で、hive-site.xml ファイルの記述が完了しました。

■ データベーススキーマの初期化

次に、schemaTool を使って、データベースを構成します。

```
# clush -w n0131 \
/opt/mapr/hive/hive-2.1/bin/schematool \
-dbType mysql \
-initSchema
n0131: Metastore connection URL:         jdbc:mysql://n0131.jpn.linux.hpe.com/
metastore
n0131: Metastore Connection Driver :     com.mysql.jdbc.Driver
n0131: Metastore connection User:        hive
n0131: Starting metastore schema initialization to 2.1.0
```

```
n0131: Initialization script hive-schema-2.1.0.mysql.sql
n0131: Initialization script completed
n0131: schemaTool completed
```

■ beeline コマンドの確認

クライアントノードから、beeline コマンドを使って Hive サーバーにアクセスできるかを確認します。Hive サーバーにアクセスできれば、beeline のコマンドプロンプトが出現† します。

```
# hostname
n0130.jpn.linux.hpe.com

# su - mapr
$ whoami
mapr

$ beeline
Beeline version 2.1.1-mapr-1803 by Apache Hive
beeline>
```

† この時点で、beeline コマンドが出現せずにエラーになる場合は、再度、configure.sh スクリプトの実行、MariaDB、hive-site.xml ファイルの設定などを見直す必要があります。

beeline コマンドプロンプトで、Hive に接続します。Hive に接続するには、beeline コマンドプロンプトで、以下のように入力します。

```
beeline > !connect jdbc:hive2://n0131:10000/default
```

続けて、以下のようにユーザー名とパスワードの入力が求められるので、ユーザー名に mapr、パスワードに、「password1234」を入力します。

```
Enter username for jdbc:hive2://n0131:10000/default: mapr
Enter password for jdbc:hive2://n0131:10000/default: ********
```

コマンドプロンプトが以下のように変化します。

```
0: jdbc:hive2://n0131:10000/default>
```

デフォルトのデータベース「default」が見るかどうかを確認します。

```
0: jdbc:hive2://n0131:10000/default> show databases;
+----------------+--+
| database_name  |
+----------------+--+
| default        |
+----------------+--+
1 row selected (0.164 seconds)
0: jdbc:hive2://n0131:10000/default>
```

beeline のコマンドプロントで「!quit」と入力し、beeline コマンドを終了します。

```
0: jdbc:hive2://n0131:10000/default> !quit
$
```

以上で、HiveMetastore サーバーを MariaDB で構成した Hive を利用できるようになりました。

6-1-3 Hive の動作確認

beeline コマンドを使って SQL のようなテーブル操作を行います。今回は、従業員リストを作成し、従業員リストから項目を抽出する作業を行います。まず、以下のような従業員リストのテキストファイル「employee.txt」を作成します。ここで、employee.txt 内の各項目（従業員番号、事業所、所属組織、名前）間の空白は、スペースキーではなく、タブキーで入力します。

```
$ hostname
n0130.jpn.linux.hpe.com

$ whoami
mapr

$ vi /home/mapr/employee.txt
EmpNum  Office  Org     EmpName
012345  Tokyo   Sales   Masazumi Koga
023456  Osaka   Dev     Satoshi Nakamoto
034567  USA     Ops     John William Mauchly
045678  USA     IT      John von Neumann
```

各項目がタブで区切られた employee.txt ファイルが作成できたら、Hive サーバーのユーザー mapr のホームディレクトリにコピーしておきます。

```
$ scp employee.txt n0131:/home/mapr/
```

beeline コマンドを入力し、Hive を起動します。beeline には、Hive サーバーへの接続は、「-u」オプション、ユーザー名は、「-n」オプション、ユーザーのパスワードは、「-p」オプションで指定可能です。

```
$ beeline \
-u jdbc:hive2://n0131:10000/default \
-n mapr \
-p password1234
...
0: jdbc:hive2://n0131:10000/default>
```

Hive サーバーへの接続を確立済みの beeline のコマンドプロンプトは、「0: jdbc:hive2://n0131:10000/default>」で表されますが、紙面の都合上、本書では、beeline のコマンドプロンプトを「beeline>」とします。

■ テーブルの作成

beeline コマンドプロンプト上で、テーブル「emplist」を作成します。beeline コマンドプロンプトでは、長い 1 行のコマンドを複数行にわたって入力可能です。複数行に跨がる場合は、コマンドプロンプトが「. . . .>」で表されます。命令の末尾に「;」を入力して Enter キーを入力すると、1 つの命令として認識されます。

```
beeline> create table
. . . .> emplist(empnum INT, office STRING, org STRING, empname STRING)
. . . .> row format delimited fields terminated by '\t';
No rows affected (1.704 seconds)
```

テーブルが作成できているかを確認します。

```
beeline> show tables;
+-----------+--+
| tab_name  |
+-----------+--+
| emplist   |
+-----------+--+
1 row selected (0.121 seconds)
```

Note　テーブルの削除

テーブルを削除するには、drop コマンドを使います。

```
beeline> drop table emplist;
```

Hive サーバー（n0131）上のユーザー mapr のホームディレクトリにある従業員リストのファイル「employee.txt」を Hive にロードします†。

```
beeline> load data local inpath
beeline> '/home/mapr/employee.txt'
beeline> into table emplist;
No rows affected (0.181 seconds)
```

† この時点で、employee.txt ファイルが、MapR-FS の/user/hive/warehouse ディレクトリに生成されます。

ロードした従業員リストを表示できるか確認します。†

```
beeline> select emplist.* from emplist;
+-----------------+-----------------+--------------+-----------------------+--+
| emplist.empnum  | emplist.office  | emplist.org  |    emplist.empname    |
+-----------------+-----------------+--------------+-----------------------+--+
| NULL            | Office          | Org          | EmpName               |
| 12345           | Tokyo           | Sales        | Masazumi Koga         |
| 23456           | Osaka           | Dev          | Satoshi Nakamoto      |
| 34567           | USA             | Ops          | John William Mauchly  |
| 45678           | USA             | IT           | John von Neumann      |
+-----------------+-----------------+--------------+-----------------------+--+
5 rows selected (0.131 seconds)
```

† 従業員リストが表示されず、「NULL」が表示される場合は、employee.txt ファイルの項目間がタブで区切られているかどうかを確認してください。また、不要な半角スペースや空行が入っている場合は、取り除く必要があります。

Hive から正常に従業員リストをロードできました。さらに、大阪事業所に所属する従業員情報を抽出できるかを確認します。

```
beeline> select emplist.* from emplist
beeline> where emplist.office like 'Osaka';
+-----------------+-----------------+--------------+--------------------+--+
| emplist.empnum  | emplist.office  | emplist.org  |  emplist.empname   |
+-----------------+-----------------+--------------+--------------------+--+
| 23456           | Osaka           | Dev          | Satoshi Nakamoto   |
+-----------------+-----------------+--------------+--------------------+--+
1 row selected (0.522 seconds)
```

大阪事業所に所属する従業員の情報を抽出できました。beeline のコマンドプロンプトを終了します。

```
beeline> !quit
$
```

■ MapR-FS に格納されたテーブルを確認

MapR-FS 上でも従業員情報の employee.txt にアクセスできるかを確認しておきます。

```
$ hostname
n0130.jpn.linux.hpe.com

$ hadoop fs -cat /user/hive/warehouse/emplist/employee.txt
...
EmpNum  Office  Org     EmpName
012345  Tokyo   Sales   Masazumi Koga
023456  Osaka   Dev     Satoshi Nakamoto
034567  USA     Ops     John William Mauchly
045678  USA     IT      John von Neumann

$ exit
```

クライアントから NFS マウントして、アクセスできることを確認しておきます。

```
# hostname
n0130.jpn.linux.hpe.com

# whoami
root

# mount -t nfs n0131:/mapr /mnt
# cd /mnt/hpe-mapr-cluster01/user/hive/warehouse/emplist/
```

```
# cat ./employee.txt
EmpNum  Office  Org     EmpName
012345  Tokyo   Sales   Masazumi Koga
023456  Osaka   Dev     Satoshi Nakamoto
034567  USA     Ops     John William Mauchly
045678  USA     IT      John von Neumann

# cd; umount /mnt
```

以上で、MapR クラスター上で Hive によるデータ操作ができました。

6-2 Apache Impala

Hadoop には、Hive 以外にもクエリーエンジンが存在します。Hive では、Hadoop の分散処理の仕組みである MapReduce を使っていましたが、MapReduce に依存しない高速なクエリーの実現に関する議論がコミュニティの間で行われていました。そこで、従来の MapReduce の仕組みに依存せず、まったく新しい分散型の高速クエリーエンジンとして開発されたのが、Apache Impala です。

Apache Impala は、Hive 同様に、使い慣れた SQL に似た構文でデータ操作が可能です。また、Hive サーバーに格納されたデータ資産をそのまま利用できるメリットがあります。Apache Impala は、impala-shell と呼ばれるコマンドラインツールが用意されており、データへのクエリーはこのツールを使用します。

Impala は、Hive が稼働する環境で利用されることが多く、Hive の MetaStore サービスと連携して稼働します。Impala 環境における Hive の MetaStore には、Impala がアクセス可能なテーブル情報が格納されます。ユーザーデータのファイルやテーブル自体は、分散ファイルシステムに格納されます。

Impala の実行環境では、通常、すべてのノードに impalad デーモンが稼働し、データの操作を行います。impalad デーモンが稼働するノードは、2 種類に分けられます。1 つは、通常のデータの読み書きを行うノードで、分散ファイルシステムにアクセスするクラスターノード、もう 1 つはコーディネータノードです。

コーディネータノードは、クライアントからのクエリー操作の依頼を受け付けます。そして、クエリーを並列化し、他の impalad デーモンが稼働するクラスターノードへのクエリーの配布、クエリー結果の受け取り、クライアントへのクエリーの返答といった役目を担います。その他、Impala には impalad デーモンが稼働するノードの死活監視を行う statestored デーモンが存在しま

す (図 6-3)。Impala を構成する主なコンポーネントには、表 6-3 に示すものがあります。

表 6-3　Impala を構成する主なコンポーネント

デーモン	役割
impalad	データの書き込み、コーディネータへのクエリー結果の送信
statestored	impalad デーモンが稼働するノードのヘルスチェック
catalogd	データ操作時に、impala ノードに対してメタデータの更新を許可

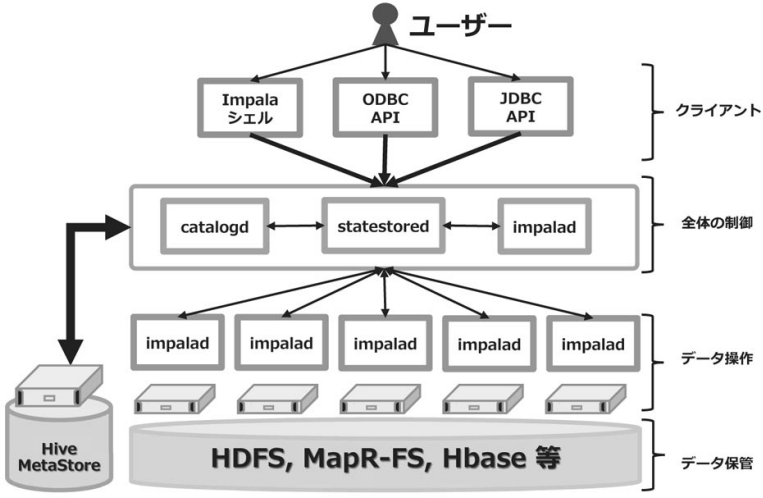


図 6-3　Apache Impala のコンポーネント

6-2-1　MapR クラスターへの Apache Impala のインストール

以下では、MapR クラスターに Apache Impala をインストールし、データ操作を行う例を示します。今回は、MapR クラスターの全ノードにおいて、事前に MapR 版の Hive がインストール済みであり、かつ、MapR の MEP のリポジトリが設定されていることが前提です（図 6-4)。

・impaladデーモン　　：全クラスターノードで稼働
・statestoredデーモン　：Hive MetaStoreノードで稼働
・catalogdデーモン　　：Hive MetaStoreノードで稼働
・impala-shell　　：全クラスターノードで発行可能

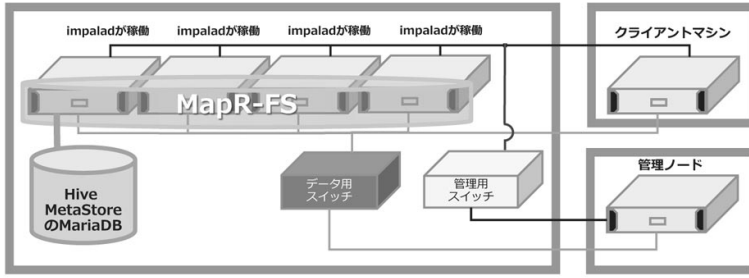


図 6-4　今回構築する Apache Impala のシステム構成

■ Apache Impala のインストール

MEP で提供される Apache Impala の RPM パッケージを全クラスターノードにインストール† します。

```
# hostname
n0130.jpn.linux.hpe.com

# clush -a "yum makecache fast && yum install -y mapr-impala"
```

† クライアントノードには、インストールしません。

■ 環境変数の設定

Impala の環境変数を設定する env.sh ファイルを、クラスターノードから作業用のクライアントノードの/root ディレクトリにコピーします。

```
# pwd
/root

# scp n0131-mgm:/opt/mapr/impala/impala-2.10.0/conf/env.sh /root/
# cp ./env.sh ./env.sh.org
```

Impala が参照する Hive の設定ファイル hive-site.xml ファイルのパラメーターを確認します。

```
# clush -g cl,all -L \
"grep -R -1 '<name>hive.metastore.uris</name>' \
/opt/mapr/hive/hive-2.1/conf/hive-site.xml"
n0130:   <property>
n0130:     <name>hive.metastore.uris</name>
n0130:     <value>thrift://n0131.jpn.linux.hpe.com:9083/</value>
...
```

設定ファイル env.sh にパラメーター HIVE_METASTORE_URI を記述します。このパラメーターには、Impala が参照する Hive の設定ファイル hive-site.xml ファイルのパラメーター「hive.metastore.uris」に設定されている値を指定します。この例では、「thrift://n0131.jpn.linux.hpe.com:9083」です。

```
# cat >> /root/env.sh << __EOF__
HIVE_METASTORE_URI=thrift://n0131.jpn.linux.hpe.com:9083
__EOF__
```

パラメーター「IMPALA_STATE_STORE_HOST」と「CATALOG_SERVICE_HOST」を env.sh ファイルに記述します。今回は、MapR クラスターノードの n0131.jpn.linux.hpe.com を指定します。

```
# sed -i \
's/IMPALA_STATE_STORE_HOST=localhost/\
IMPALA_STATE_STORE_HOST=n0131.jpn.linux.hpe.com/g' \
/root/env.sh

# sed -i \
's/CATALOG_SERVICE_HOST=localhost/\
CATALOG_SERVICE_HOST=n0131.jpn.linux.hpe.com/g' \
/root/env.sh
```

さらに、env.sh ファイル内のパラメーター「IMPALA_SERVER_ARGS」の「-mem_limit」オプションに「90% \」を記述し、「-num_threads_per_disk=2 \」を追記します。

```
# vi /root/env.sh
...
IMPALA_SERVER_ARGS=" \
    -log_dir=${IMPALA_LOG_DIR} \
    -state_store_port=${IMPALA_STATE_STORE_PORT} \
    -use_statestore \
    -authorized_proxy_user_config=mapr=* \
```

```
    -state_store_host=${IMPALA_STATE_STORE_HOST} \
    -catalog_service_host=${CATALOG_SERVICE_HOST} \
    -be_port=${IMPALA_BACKEND_PORT} \
    -mem_limit=90% \
    -num_threads_per_disk=2 \
"
...
```

■ env.sh ファイルのコピー

env.sh ファイルを全クラスターノードにコピーします。

```
# clush -a -c \
/root/env.sh \
--dest=/opt/mapr/impala/impala-2.10.0/conf/
```

■ Warden サービスの再起動

Warden サービスを再起動します。

```
# clush -a "systemctl restart mapr-warden"
```

■ パッケージのインストール

env.sh ファイル内のパラメーターの設定と矛盾がないように、Impala 関連のパッケージを n0131.jpn.linux.hpe.com にインストールします。

```
# clush -w n0131 \
"yum install -y \
mapr-impala-statestore \
mapr-impala-catalog \
mapr-impala-server"
```

Impala クラスターの構成を更新します。

```
# clush -a "/opt/mapr/server/configure.sh -R"
```

以上で、Impala を利用できるようになりました。

Note　Impala の再起動

クラスターノードの env.sh ファイルを編集した場合、maprcli コマンドを使って、Impala のサービスの再起動が必要です。ただし、Warden サービスの再起動は不要です。

```
# clush -w n0131 \
"maprcli node services -name impalaserver \
-action restart -nodes n0131.jpn.linux.hpe.com"

# clush -w n0131 \
"maprcli node services -name impalastore \
-action restart -nodes n0131.jpn.linux.hpe.com"

# clush -w n0131 \
"maprcli node services -name impalacatalog \
-action restart -nodes n0131.jpn.linux.hpe.com"
```

6-2-2　Impala の動作確認

impala-shell コマンドを使って SQL のようなテーブル操作を行います。今回は、MapR 社がサンプルとして提供している CSV ファイルをロードします。まず、クラスターノードの n0131.jpn.linux.hpe.com にログインし、ユーザー mapr で CSV ファイルを入手します。

```
# ssh n0131-mgm
# hostname
n0131.jpn.linux.hpe.com

# su - mapr
$ whoami
mapr

$ export http_proxy=http://proxy.your.site.com:8080
$ pwd
/home/mapr
$ wget http://doc.mapr.com/download/attachments/22906623/customers.csv
```

■ CSV ファイルの内容を Hive に格納するバッチファイルの作成

Impala で読み込む CSV ファイルの内容は、Hive に格納しておきます。Hive に CSV ファイルの内容を格納する SQL 文を記述したファイル「load_customers.sql」を作成します。この load_customers.sql は、Hive 上にデータベース「testdb01」を作成し、さらに、そのデータベース上にテーブル「customers」を作成し、CSV ファイル「customers.csv」ファイルの内容をテーブルの項目として割り当てます。

```
$ cat > load_customers.sql << __EOF__
create database  testdb01;
use              testdb01;
create table customers
(
FirstName string,
LastName string,
Company  string,
Address  string,
City     string,
County   string,
State    string,
Zip      string,
Phone    string,
Fax      string,
Email    string,
Web      string
)
row format delimited fields terminated by ',' stored as textfile;
load data local inpath '/home/mapr/customers.csv'
overwrite into table customers;
__EOF__
```

上記の CSV ファイルでは、customers.csv ファイルが/home/mapr ディレクトリに保管されていることを想定しています。

■ Hive へのデータの格納

作成した load_customers.sql ファイルを使って、Hive 上にデータベースとテーブルを作成し、customers.csv ファイルのデータをテーブルに記録します。

```
$ beeline \
-u jdbc:hive2://n0131:10000/default \
```

```
-n mapr -p password1234 < load_customers.sql
...
```

■ Hive に記録したデータの確認

Hive 上にデータベース「testdb01」が存在し、テーブル「customers」が存在するか確認します。

```
$ beeline \
-u jdbc:hive2://n0131:10000/default \
-n mapr -p password1234 \
-e "use testdb01; show tables;"
...
+------------+--+
|  tab_name  |
+------------+--+
| customers  |
+------------+--+
1 row selected (0.386 seconds)
...
```

Hive に格納したテーブル「customers」が Impala からアクセスできるかを impala-shell コマンドで確認† します。

```
$ impala-shell \
-i n0131.jpn.linux.hpe.com:21000 \
-q "show databases;"
...
Query: show databases
+------------------+-----------------------------------------------+
| name             | comment                                       |
+------------------+-----------------------------------------------+
| _impala_builtins | System database for Impala builtin functions  |
| default          | Default Hive database                         |
| testdb01         |                                               |
+------------------+-----------------------------------------------+
Fetched 3 row(s) in 0.00s
```

> † この時点で、Hive に格納されたデータベース「testdb01」が見えない場合は、env.sh ファイルを確認し、Impala のサービスを再起動してみてください。

Impala が正常に Hive と接続できれば、Hive 上に作成した testdb01 データベースが表示されま

す。実際に、testdb01 に作成したテーブル「customers」の行数を表示できるかを確認します。

```
$ impala-shell \
-i n0131.jpn.linux.hpe.com:21000 \
-q "use testdb01; select count(*) from customers;"
...
+----------+
| count(*) |
+----------+
| 501      |
+----------+
Fetched 1 row(s) in 0.16s
```

データベース「testdb01」に作成したテーブル「customers」が 501 行であることがわかります。

念のため、ユーザー mapr のホームディレクトリに保管した customers.csv に含まれる行数（ここでは、改行の数）が 501 行かどうかを確認します。

```
$ wc -l /home/mapr/customers.csv
501 /home/mapr/customers.csv
```

以上で、Hive に格納されたデータを Impala から利用することができました。

6-3 Apache HBase

Apache HBase は、Hadoop クラスター上で稼働する列指向型の NoSQL データベースソフトウェアです。分散アーキテクチャのデータベースであり、低遅延で、かつ、一貫性の高いデータの読み書き操作を可能にします。また、非常に巨大なテーブルの操作を高速に処理することを目標に設計されています。不正検出や大規模システム向けの認証管理ソフトウェアのデータベースエンジンとしても採用されており、医療、Web 広告、SNS 企業など、さまざまな分野で利用されています。

6-3-1 HBase のアーキテクチャ

HBase は、Hadoop の分散ファイルシステム上にデータを保持し、これによりデータの冗長化と永続性を確保します。また、Zookeeper サービスと組み合わせて稼働し、可用性を確保します。HBase には、ユーザーがデータ操作を対話型で行うための HBase シェルが搭載されています。また、HBase シェルだけでなく、Apache Impala から HBase 上のデータへのアクセスも可能です。

Hadoop クラスターの上に構成された HBase は、**HBase クラスター**と呼ばれます。HBase クラスターは、マスターノードとスレーブノードに分かれます。マスターノードでは、HMaster サービスと HQuorumPeer サービスが稼働し、マスターサーバーとも呼ばれます。一方、スレーブノードは、HRegionServer サービスが稼働し、リージョンサーバーと呼ばれます。リージョンサーバーが、HDFS に格納されているデータとやり取りを行います。リージョンサーバーの処理対象となるデータは、**リージョン**という単位で複数のワーカーノードに分割され、読み書きが行われます（図 6-5）。各コンポーネントの役割は、**表 6-4** に示したとおりです。

表 6-4　各コンポーネントの役割

コンポーネント	役割
HMaster	テーブル操作やリージョンの割り当てを行う
HRegionServer	データの読み書きを担う。クライアントは HRegionServer と直接通信し、データへアクセスする
HQuorumPeer	ZooKeeper で提供されるサービスで、クラスタの状態を維持
DataNode	HRegionServer が管理するデータを HDFS に格納
HDFS	HBase の全データは、分散ファイルシステムに格納される

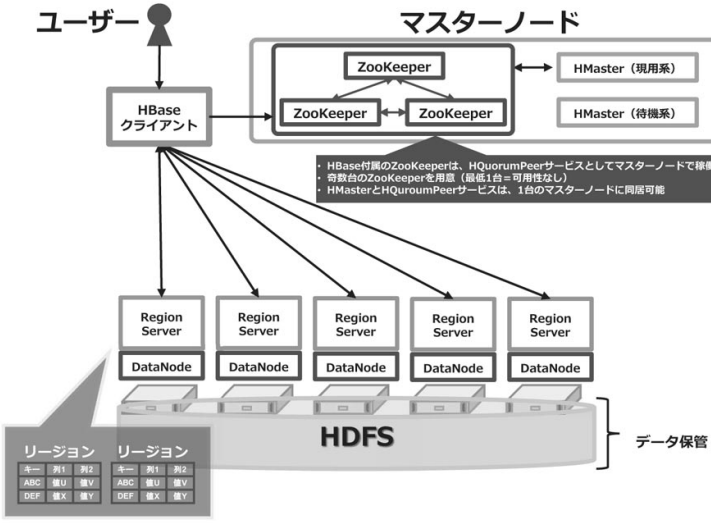


図 6-5　HBase のコンポーネント

Note　MapR-DB の利用

MapR 6 系に対応の MEP バージョン 4.x 以降では、マスターサーバー（パッケージ名は、mapr-hbase-master）とリージョンサーバー（パッケージ名は、mapr-hbase-regionserver）が提供されなくなりました。そのため、MapR 6 系のクラスターでは、MapR-DB を利用してください。

6-3-2　Apache Hadoop 3 クラスターへの HBase のインストール

以下では、Apache Hadoop 3 クラスターに HBase をインストールする手順を紹介します。今回、Apache HBase の HMaster サービスと ZooKeeper が提供する HQuorumPeer サービスは、Hadoop マスターノードの n0121.jpn.linux.hpe.com の 1 台のみで稼働させ、ワーカーノードには、HRegionServer サービスを稼働させます。また、クライアントから HBase シェルによるデータ操作を行います（図 6-6）。

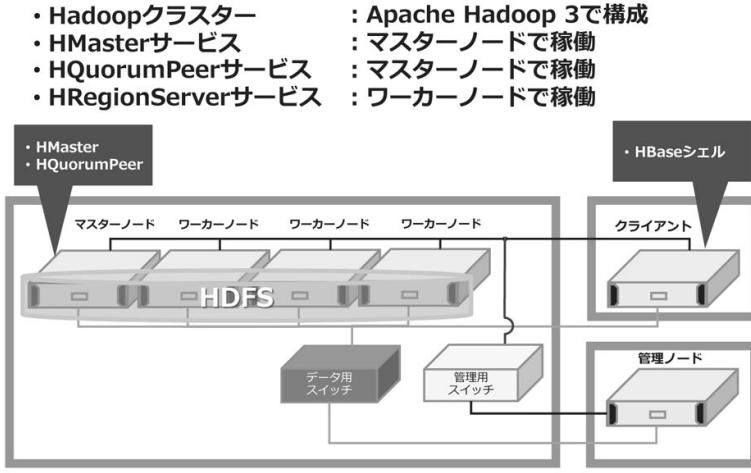


図 6-6　今回構築する Apache HBase のシステム構成

まず、Apache HBase の tar アーカイブを入手します。

```
# hostname
```

```
n0120.jpn.linux.hpe.com

# wget http://www-us.apache.org/dist/hbase/1.4.3/hbase-1.4.3-bin.tar.gz
```

入手した HBase の tar アーカイブを全ノードにコピーします。

```
# clush -a -c ./hbase-1.4.3-bin.tar.gz --dest=/root/
```

HBase の tar アーカイブを/opt ディレクトリに展開します。

```
# clush -g cl,all \
"tar xzvf /root/hbase-1.4.3-bin.tar.gz -C /opt"
```

■ 環境変数の設定

HBase は、Java を利用するため、環境変数の JAVA_HOME の設定が必要です。Java の環境変数は、HBase の設定ファイル「hbase-env.sh」に記述します。また、hbase-env.sh ファイル内の「HBASE_MASTER_OPTS」と「HBASE_REGIONSERVER_OPTS」に指定されているオプション「-XX:PermSize=128m -XX:MaxPermSize=128m」を削除します。さらに、「HBASE_REGIONSERVERS」の行の先頭の「 # 」を削除します。

```
# cp -a \
/opt/hbase-1.4.3/conf/hbase-env.sh \
/opt/hbase-1.4.3/conf/hbase-env.sh.org

# vi /opt/hbase-1.4.3/conf/hbase-env.sh
...
export JAVA_HOME=/usr/lib/jvm/jre
...
export HBASE_MASTER_OPTS="$HBASE_MASTER_OPTS -XX:ReservedCodeCacheSize=256m"
...
export HBASE_REGIONSERVER_OPTS="$HBASE_REGIONSERVER_OPTS -XX:ReservedCodeCache
Size=256m"   1 行入力
...
export HBASE_REGIONSERVERS=${HBASE_HOME}/conf/regionservers
...
```

全ノードに hbase-env.sh ファイルをコピーします。

```
# clush -a -c \
```

```
/opt/hbase-1.4.3/conf/hbase-env.sh \
--dest=/opt/hbase-1.4.3/conf/
```

■ ユーザーの環境変数の設定

root ユーザーの環境変数を設定します。

```
# cat >> $HOME/.bash_profile << '__EOF__'
export HBASE_HOME=/opt/hbase-1.4.3
export PATH=$HBASE_HOME/bin:$PATH
export HBASE_CONF_DIR=$HBASE_HOME/conf
__EOF__

# clush -a -c $HOME/.bash_profile --dest=$HOME/
# . $HOME/.bash_profile
# printenv HBASE_HOME
/opt/hbase-1.4.3
```

■ hbase-site.xml ファイルの作成

HBase の設定ファイル「hbase-site.xml」を作成します。

```
# cd /opt/hbase-1.4.3/conf
# cp -a hbase-site.xml hbase-site.xml.org
# cat > /opt/hbase-1.4.3/conf/hbase-site.xml << __EOF__
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
    <name>hbase.rootdir</name>
    <value>hdfs://n0121.jpn.linux.hpe.com:9000/hbase</value>
  </property>
  <property>
    <name>hbase.zookeeper.property.clientPort</name>
    <value>2181</value>
  </property>
  <property>
    <name>hbase.cluster.distributed</name>
    <value>true</value>
  </property>
  <property>
```

```
    <name>hbase.zookeeper.quorum</name>
    <value>n0121.jpn.linux.hpe.com</value>
  </property>
</configuration>
__EOF__
```

Note　HBase クラスターのサービスの可用性要件

本来、HBase のマスターノードは、可用性を確保するために、ZooKeeper を複数台で構成します。ZooKeeper の場合は、過半数を確保した ZooKeeper ノードが正常系として稼働し続けるクォーラムと呼ばれる仕組みを採用しています。このクォーラムを実現するには、奇数台（可用性を確保する場合は、最低 3 台）で構成します。そのため、`hbase-site.xml` ファイル内のパラメーター「`hbase.zookeeper.quorum`」では、ZooKeeper ノードを奇数台指定する必要がありますが、今回は、非冗長構成（HBase マスターが 1 台のみ）なので、ZooKeeper サービスが稼働させるノードも 1 台のみの記述にしています。本番システムでは、HBase クラスターのサービスの可用性要件をよく考慮し、ZooKeeper サービスの個数と HMaster サービスの個数を決定してください。

設定ファイルを記述したら、全ノードに `hbase-site.xml` ファイルをコピーします。

```
# clush -a -c ./hbase-site.xml --dest=/opt/hbase-1.4.3/conf/
```

■ リージョンサーバーの一覧を作成

データを処理する HBase リージョンサーバーの一覧を `regionservers` ファイルに記述します。

```
# pwd
/opt/hbase-1.4.3/conf

# cat > ./regionservers << __EOF__
n0122.jpn.linux.hpe.com
n0123.jpn.linux.hpe.com
n0124.jpn.linux.hpe.com
__EOF__

# clush -a -c ./regionservers --dest=/opt/hbase-1.4.3/conf/
```

■ SSH の設定

HBase では、起動スクリプトにおいて、複数ノードに対して、SSH を実行し、RegionServer サービスの起動などを行うため、HBase マスターとなるノードからパスワード入力なしの SSH 接続の設定が必要です。今回は、HBase マスターが n0121.jpn.linux.hpe.com なので、そこから自分自身を含め、全クラスターノードに root ユーザーのパスワードを入力せずに SSH 接続の設定を行います。まず、HBase マスターノードにログインします。

```
# ssh n0121-mgm
```

公開鍵を生成します

```
# ssh-keygen -f $HOME/.ssh/id_rsa -t rsa -N ''
```

鍵を各ノードにコピーし、HBase マスターノードから全リージョンサーバーノードにパスワード入力なしで SSH 接続できるように設定します。root ユーザーには、パスワードとして password1234 が事前に設定されています。

```
# yum makecache fast && yum install -y sshpass
# sshpass -p "password1234" ssh-copy-id -f -o \
StrictHostKeyChecking=no root@n0121
# sshpass -p "password1234" ssh-copy-id -f -o \
StrictHostKeyChecking=no root@n0122
# sshpass -p "password1234" ssh-copy-id -f -o \
StrictHostKeyChecking=no root@n0123
# sshpass -p "password1234" ssh-copy-id -f -o \
StrictHostKeyChecking=no root@n0124
# sshpass -p "password1234" ssh-copy-id -f -o \
StrictHostKeyChecking=no root@n0121-mgm
# sshpass -p "password1234" ssh-copy-id -f -o \
StrictHostKeyChecking=no root@n0122-mgm
# sshpass -p "password1234" ssh-copy-id -f -o \
StrictHostKeyChecking=no root@n0123-mgm
# sshpass -p "password1234" ssh-copy-id -f -o \
StrictHostKeyChecking=no root@n0124-mgm
```

HBase マスターノードから全リージョンサーバーノードにパスワード入力なしで、SSH 接続できるかを確認します。

```
# for i in `seq 121 124`; \
do \
```

```
ssh n0${i}.jpn.linux.hpe.com hostname; \
done
n0121.jpn.linux.hpe.com
n0122.jpn.linux.hpe.com
n0123.jpn.linux.hpe.com
n0124.jpn.linux.hpe.com

# for i in `seq 121 124`; \
do \
ssh n0${i}-mgm.jpn.linux.hpe.com hostname; \
done
n0121.jpn.linux.hpe.com
n0122.jpn.linux.hpe.com
n0123.jpn.linux.hpe.com
n0124.jpn.linux.hpe.com
```

Note　SSH の設定確認

パスワード入力なしの SSH 接続の設定が不完全な場合は、後述の HBase 起動スクリプトの実行に失敗するため、再度、SSH の設定を確認してください。公開鍵をコピーしても、パスワード入力のプロンプトが表示される場合、/root/.ssh ディレクトリのパーミッションが 600 になっているかを確認してください。

6-3-3　HBase クラスター起動

HBase には、HBase マスターサーバー、ZooKeeper、リージョンサーバーを自動的に起動する start-hbase.sh スクリプトが用意されています。start-hbase.sh スクリプトを実行し、HBase クラスターを起動します。

```
# hostname
n0121.jpn.linux.hpe.com

# which start-hbase.sh
/opt/hbase-1.4.3/bin/start-hbase.sh

# start-hbase.sh
n0121.jpn.linux.hpe.com: running zookeeper,   ...
running master,   ...
```

```
n0124.jpn.linux.hpe.com: running regionserver,   ...
n0123.jpn.linux.hpe.com: running regionserver,   ...
n0122.jpn.linux.hpe.com: running regionserver,   ...
```

■ サービスの確認

HBase クラスターの起動に必要なサービスが稼働しているかを確認します。

```
# hostname
n0121.jpn.linux.hpe.com

# jps | grep HM
24350 HMaster

# jps | grep HQ
24279 HQuorumPeer

# for i in `seq 122 124`; do ssh n0$i "jps | grep HR"; done
1107 HRegionServer
1728 HRegionServer
962 HRegionServer
```

HBase マスターノードで、HMaster サービスと HQuorumPeer サービス、そして、全リージョンサーバーノードで、HRegionServer サービスが稼働しているかをチェックしてください。もしサービスが稼働できない場合は、HBase の `env.sh` ファイル、`hbase-site.xml` ファイル、`regionservers` ファイル、そして`$HOME/.bash_profile` の環境変数、SSH 接続、時刻同期などの設定を再確認してください。

Column　HBase クラスターのデーモン起動

セキュリティやメンテナンスの都合で、パスワード入力なしの SSH 接続が不可能な場合でも、HBase クラスターのデーモンを個別に起動させたい場合があります。その場合は、`hbase-daemon.sh` スクリプトで起動します。

```
# clush -w n0121 ". $HOME/.bash_profile; hbase-daemon.sh start zookeeper"
# clush -w n0121 ". $HOME/.bash_profile; hbase-daemon.sh start master"
# clush -g dn ". $HOME/.bash_profile; hbase-daemon.sh start regionserver"
```

■ HBase シェルの起動

クライアントノードにインストールされた HBase シェルから HDFS のデータを操作します。事前に一般ユーザー koga の環境変数を$HOME/.bash_profile に記述し、ロードしておきます。

```
# hostname
n0120.jpn.linux.hpe.com

# su - koga
$ whoami
koga

$ cat >> $HOME/.bash_profile << '__EOF__'
export HBASE_HOME=/opt/hbase-1.4.3
export PATH=$HBASE_HOME/bin:$PATH
export HBASE_CONF_DIR=$HBASE_HOME/conf
__EOF__

$ clush -a -c $HOME/.bash_profile --dest=$HOME/
$ . $HOME/.bash_profile
```

HBase シェルの起動は、「hbase shell」コマンドです。

```
$ which hbase
/opt/hbase-1.4.3/bin/hbase

$ hbase shell
...
hbase(main):001:0>
```

上記の「hbase(main):001:0>」が、HBase のコマンドプロンプトです。この HBase のコマンドプロンプトから HBase データベースにアクセスし、テーブルなどを作成します。今回は、カラムファミリ「stats」を持つテーブル「weblog」を作成します。

```
hbase(main):001:0> create 'weblog','stats'
0 row(s) in 2.6860 seconds
```

テーブルが作成できているかを確認します。

```
hbase(main):002:0> list
TABLE
weblog
1 row(s) in 0.0240 seconds
```

テーブル「weblog」に値をセットします。

```
hbase(main):003:0> put 'weblog', 'row1', 'stats:daily', 'daily-val'
hbase(main):004:0> put 'weblog', 'row2', 'stats:weekly', 'weekly-val'
```

テーブルにセットしたすべての値を表示します。

```
hbase(main):005:0> scan 'weblog'
ROW      COLUMN+CELL
 row1    column=stats:daily, timestamp=1519831060263, value=daily-val
 row2    column=stats:weekly, timestamp=1519831073879, value=weekly-val
2 row(s) in 0.0640 seconds
```

row1 の内容を表示してみます。

```
hbase(main):006:0> get 'weblog', 'row1'
COLUMN          CELL
 stats:daily    timestamp=1519831060263, value=daily-val
1 row(s) in 0.0710 seconds
```

以上で、HBase のテーブルを作成することができました。HBase のコマンドプロンプトを終了するには、exit を入力します。

```
hbase(main):007:0> exit
$
```

HDFS 上からも HBase テーブルを確認します。hbase-site.xml ファイル内で指定した「hbase.rootdir」のパラメーターは、「hdfs://n0121.jpn.linux.hpe.com:9000/hbase」なので、HDFS の /hbase ディレクトリ配下にデータが格納されます。作成した weblog テーブルは、HDFS の/hbase/data/default ディレクトリの下に作成されています。

```
$ hdfs dfs -ls /hbase/data/default/
Found 1 items
drwxr-xr-x   ...  2018-04-08 06:15 /hbase/data/default/weblog
```

以上より、HDFS 上に HBase のテーブル情報が格納されていることが確認できました。

■ HBase 上のテーブルの削除

作成したテーブルを削除してみます。HBase のコマンドプロンプトで作業します。

```
$ hbase shell
hbase(main):001:0> list
TABLE
weblog
1 row(s) in 0.2870 seconds
```

disable コマンドで、テーブルを使用不可にした後、drop コマンドでテーブルを削除します。

```
hbase(main):002:0> disable 'weblog'
0 row(s) in 4.5680 seconds

hbase(main):003:0> drop 'weblog'
0 row(s) in 2.3700 seconds

hbase(main):004:0> list
TABLE
0 row(s) in 0.0160 seconds
```

以上で、HBase のテーブル「weblog」を削除できました。

6-4 MapR-DB

MapR 社は、キー・バリュー型のデータベースソフトウェアの MapR-DB を提供しています。

以下では、MapR が提供するデータベースソフトウェアである MapR-DB の使用法を簡単に紹介します。MapR-DB は、dbshell と呼ばれる対話的なアクセスを可能にするシェルが用意されています。dbshell を起動するには、クライアントノードにおいて、「mapr dbshell」を実行します。

```
# hostname
n0130.jpn.linux.hpe.com

# su - mapr
$ whoami
mapr

$ mapr dbshell
...
================================================
*                  MapR-DB Shell                 *
* NOTE: This is a shell for JSON table operations. *
================================================
Version: 6.0.1-mapr
```

```
MapR-DB Shell
maprdb mapr:>
```

MapR-DB のシェルが起動し、ユーザーは、対話形式で MapR-DB にアクセスできます。以下、MapR-DB のシェルが提供するコマンドプロンプトは、「maprdb mapr:>」で表します。

■ MapR-DB でのテーブル作成

MapR-DB 上にテーブル「/mytable01」を作成します。テーブルは、MapR-DB シェルのコマンドプロンプトにおいて、「create」を入力し、データベース名を付与します。

```
maprdb mapr:> create /mytable01
Table /mytable01 created.
```

■ テーブルの確認

作成したテーブルを確認します。テーブルは、「list」を入力し、パスを指定します。

```
maprdb mapr:> list /
/mytable01
1 table(s) found.
```

■ テーブルへのデータの格納

作成したテーブル「mytable01」にデータを格納してみます。MapR-DB のテーブルにデータを格納するには、データごとに ID を付与します。以下は、「car001」という ID を作成し、その「car001」の値「Name」と「Year」と「Fuel」に、それぞれ、「ABC」、「2018」、「Gas」を格納する例です。

```
maprdb mapr:> insert /mytable01 --id car001 --value '{"Name":"ABC", "Year":
"2018", "Fuel":"Gas"}'   (1 行入力)
Document with id: "car001" inserted.
```

続けて、mytable01 テーブルに、ID「car002」を作成し、異なる値を入れてみます。

```
maprdb mapr:> insert /mytable01 --id car002 --value '{"Name":"DEF", "Year":
```

```
"2017", "Fuel":"FCV"}'  (1 行入力)
Document with id: "car002" inserted.
```

■ テーブルの値の確認

テーブルに格納したデータを表示します。テーブル全体の値の表示は、find コマンドにテーブル名を付与します。

```
maprdb mapr:> find /mytable01
{"_id":"car001","Fuel":"Gas","Name":"ABC","Year":"2018"}
{"_id":"car002","Fuel":"FCV","Name":"DEF","Year":"2017"}
2 document(s) found.
```

ID で絞って表示することも可能です。その場合は、find コマンドに「--id」オプションを付与し、ID を指定します。

```
maprdb mapr:> find --id car001 /mytable01
{"_id":"car001","Fuel":"Gas","Name":"ABC","Year":"2018"}
1 document(s) found.
```

■ 値の変更

値を変更するには、replace コマンドを使用します。

```
maprdb mapr:> replace /mytable01 --id car001 --value '{"Fuel":"FCV","Name":"
GHI"}'  (1 行入力)
Document with id: "car001" inserted.
```

変更した値を確認します。

```
maprdb mapr:> find  /mytable01
{"_id":"car001","Fuel":"FCV","Name":"GHI"}
{"_id":"car002","Fuel":"FCV","Name":"DEF","Year":"2017"}
2 document(s) found.
```

replace コマンドで指定した際の項目が「Fuel」と「Name」であるため、「Year」がなくなっている点に注意してください。

■ ID の削除

ID「car001」自体を削除します。ID の削除は、delete コマンドを使用します。

```
maprdb mapr:> delete /mytable01 --id car001
Document with id: "car001" deleted.

maprdb mapr:> find  /mytable01
{"_id":"car002","Fuel":"FCV","Name":"DEF","Year":"2017"}
1 document(s) found.
```

■ テーブルの削除

テーブルの削除は、drop コマンドを使用します。

```
maprdb mapr:> drop /mytable01
Table /mytable01 deleted.
```

以上で、MapR-DB の一部の機能を紹介しました。紹介したテーブルは、一般に JSON テーブルと呼ばれるものですが、MapR-DB は、この JSON テーブル以外にバイナリテーブルと呼ばれる列指向の KVS（キーバリューストア）の形式もサポートしており、Apache HBase と同様の列指向型のデータベースとして利用することも可能です。

MapR-DB の情報源：

https://maprdocs.mapr.com/home/MapROverview/maprDB-overview.html

dbshell の情報源：

https://maprdocs.mapr.com/home/ReferenceGuide/mapr_dbshell.html

6-5 Apache Pig

Apache Pig は、Hadoop の分散処理のフレームワークである MapReduce を使ったアプリケーションを作成するための高水準のスクリプト言語です。Pig は、2006 年に米国 Yahoo! Inc. で開発され、間もなくして、Apache Software Foundation に開発プロジェクトが移譲されました。その後、eBay や LinkedIn などがデータ分析用のソフトウェア開発ツールとして Apache Pig を使い始めました。

Apache Pig のプログラミング言語は、Pig Latin と呼ばれます。この Pig Latin は、スクリプト言語であり、複雑な MapReduce コードをラッピング（抽象化）する役目を担います。開発者がデータ分析用のソフトウェアを記述する際に、複雑な MapReduce プログラミングを知らなくても、Pig Latin を使えば、簡単に Apache Hadoop 向けのデータ分析プログラムを書くことができます。また、Grunt と呼ばれるシェルが用意されており、対話型のデータ処理が行えます。

Pig では、User Defined Function（以下 UDF）と呼ばれるユーザー独自の関数を定義する機能があり、この UDF を使用すると、Python や Java などのプログラムを Pig のコードから呼び出して実行できます。

Apache Pig は、データの解析の問題を「データの流れ」として表現する「データパイプライン」のシステムで利用されます。データパイプラインは、さまざまな分析対象から生成されるデータの取り込み、加工、分析、結果の取得、可視化といった一連の処理のプロセスを指します (図 6-7)。

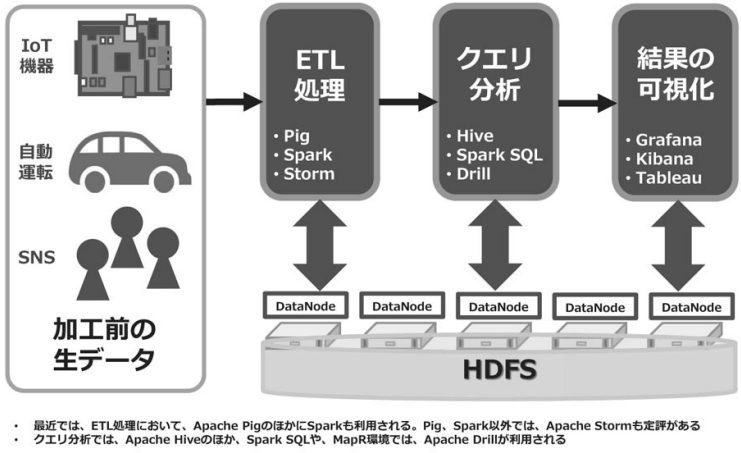


図 6-7　Apache Pig とデータパイプライン

ビッグデータの分析では、さまざまな種類のデータ発生源が存在します。それに伴い生成されるデータ形式もさまざまです。分析者は、これらの多種多様なデータ形式を分析できる形に加工する必要があります。この加工の段階においては、一般に、ETL（Extract、Transform、Load）ツールが利用されます。Pig は、データパイプラインにおける ETL ツールとして利用され、さまざまな種類のデータを加工できます。すなわち、Pig は、さまざまなデータ形式を取り込んで、分析の前段階で利用されるツール † です。

† 一方、Pig に対比されるツールとしては、Hive が挙げられます。Hive は、分析者にとって、望みのデータを抽出しやすいようにクエリーを発行できるため、クエリー分析や可視化ツールから利用されます。

6-5-1 Pig の実行モード

Apache Pig には、**ローカルモード**と MapReduce モードと呼ばれる 2 つの実行モードが存在します。ローカルモードでは、HDFS を使用することなく、ローカルの Linux のファイルシステムを使って Pig で作成したプログラムを実行できます。ローカルモードは、開発環境において使われるモードです。一方、MapReduce モードは、HDFS に存在するデータを処理します。MapReduce モードでは、Pig スクリプトによるデータ処理の際、バックエンドで MapReduce ジョブとして実行されます。

6-5-2 Apache Hadoop 3 クラスターへの Pig のインストール

以下では、Apache Hadoop 3 クラスターと MapR クラスターにおいて、Apache Pig をインストールする方法と、簡単な Pig の操作方法を紹介します。

■ Apache Pig の入手

Apache Pig の tar アーカイブを Apache の Web サイトから入手します。以下の作業は、クライアントノードで行います。

```
# hostname
n0120.jpn.linux.hpe.com

# whoami
root

# wget https://www-us.apache.org/dist/pig/pig-0.17.0/pig-0.17.0.tar.gz
```

■ tar アーカイブの展開

入手した Apache Pig の tar アーカイブを全ノードにコピーし、展開します。今回は、/opt ディレクトリ以下にインストールします。

```
# clush -a -c pig-0.17.0.tar.gz --dest=$HOME/
# clush -g cl,all "tar xzvf $HOME/pig-0.17.0.tar.gz -C /opt/"
```

■ 環境変数の設定

Pig を使ってデータを操作や開発を行う一般ユーザー koga の環境変数を設定します。環境変数は、PIG_HOME と PIG_CLASSPATH です。PIG_HOME には、/opt ディレクトリに展開した pig アーカイブのディレクトリ（今回の場合は、pig-0.17.0 ディレクトリ）を指定します。PIG_CLASSPATH には、$HADOOP_HOME/etc/hadoop ディレクトリを指定します。$HOME/.bash_profile ファイル内で、環境変数「HADOOP_HOME」が設定されていることが前提です。

```
# su - koga
$ whoami
koga

$  cat >> $HOME/.bash_profile << '__EOF__'
export PIG_HOME=/opt/pig-0.17.0
export PATH=$PIG_HOME/bin:$PATH
export PIG_CLASSPATH=$HADOOP_HOME/etc/hadoop
__EOF__

$ tail -3 $HOME/.bash_profile
export PIG_HOME=/opt/pig-0.17.0
export PATH=$PIG_HOME/bin:$PATH
export PIG_CLASSPATH=$HADOOP_HOME/etc/hadoop
```

$HOME/.bash_profile ファイルを全ノードにコピーします。

```
$ clush -a -c $HOME/.bash_profile --dest=$HOME/
```

$HOME/.bash_profile ファイルをロードし、pig コマンドのパスを確認します。

```
$ . $HOME/.bash_profile
```

```
$ clush -g cl,all -L ". $HOME/.bash_profile; which pig"
n0120: /opt/pig-0.17.0/bin/pig
...
```

Apache Hadoop 3 クラスターで Pig を利用できるようになりました。

6-5-3 Pig の動作確認

以下では、Apache Pig で簡単なデータ操作を行います。テスト用のデータ「emplist.txt」を作成します。emplist.txt は、連番、従業員の名前、年齢、勤務地、月収（ドル）をデータの項目とし、各データが「,」で区切られて構成されるテキストファイルです。

```
$ hostname
n0120.jpn.linux.hpe.com

$ cat > $HOME/emplist.txt << '__EOF__'
1, Masazumi Koga          , 43, Tokyo   , 2500.00
2, Satoshi Nakamoto       , 58, Osaka   , 9500.00
3, John von Neumann       , 49, USA     , 2000.00
4, Alan Mathieson Turing , 38, England, 6500.00
5, Kenneth Harry Olsen    , 27, USA     , 8500.00
6, John Vincent Atanasoff, 24, USA     , 4500.00
7, Charles Babbage        , 28, England, 6000.00
__EOF__
```

■ HDFS へのデータのコピー

Pig で取り扱うデータ「emplist.txt」用に HDFS のディレクトリ「/user/koga/pigtest01」を作成し、従業員データ「emplist.txt」をコピーします。

```
$ hdfs dfs -mkdir /user/koga/pigtest01
$ hdfs dfs -copyFromLocal ./emplist.txt /user/koga/pigtest01/
```

■ Pig スクリプトの作成

データ分割を行う「emp_split.pig」スクリプトを作成します。「emp_split.pig」スクリプトは、年齢が 40 歳を越えている従業員の情報を HDFS の/user/koga/pigtest01/dir01 ディレクト

リに振り分け、月収が 7000 ドルを超える従業員の情報を/user/koga/pigtest01/dir02 に振り分けます。

```
$ cat > emp_split.pig << '__EOF__'
A =
LOAD 'hdfs://n0121:9000/user/koga/pigtest01/emplist.txt'
USING PigStorage(',')
AS (id:int, name:chararray, age:int, address:chararray, salary:int);
SPLIT A INTO X IF age >= 40, Y IF (salary > 7000);
STORE X INTO 'hdfs://n0121:9000/user/koga/pigtest01/dir01';
STORE Y INTO 'hdfs://n0121:9000/user/koga/pigtest01/dir02';
__EOF__
```

■ Pig スクリプトの実行

pig コマンドは、デフォルトは、MapReduce モードで実行されます。「-x」オプションを指定し、明示的に MapReduce モードやローカルモードを指定することも可能です。pig コマンドで実行するスクリプトは、「-f」オプションの後に指定します（表 6-5）。

表 6-5　pig コマンドのモード

pig コマンドのモード	実行例
MapReduce モード	$ pig -f ./emp_split.pig
	$ pig -x mapreduce -f ./emp_split.pig
ローカルモード	$ pig -x local -f ./emp_split.pig

では、実際に pig コマンドを使って、emp_splist.pig スクリプトを実行します。

```
$ which pig
~/pig-0.17.0/bin/pig

$ pig -f ./emp_split.pig
...
Successfully stored 3 records (129 bytes) in: "/user/koga/pigtest02/dir01"
Successfully stored 2 records (86 bytes) in: "/user/koga/pigtest02/dir02"
...
```

Note　0% complete

pig コマンドを MapReduce モードで実行し、「0% complete」のメッセージで止まってしまう場合は、Hadoop システム全体のメモリ不足が考えられます。稼働している不必要なソフトウェアをいったん停止する、あるいは、物理メモリを増設するなどの対策が必要です。

HDFS に生成された結果を確認します。

```
$ hdfs dfs -ls /user/koga/pigtest01/dir01/
Found 2 items
-rw-r--r-- 3 koga supergroup ... /user/koga/pigtest01/dir01/_SUCCESS
-rw-r--r-- 3 koga supergroup ... /user/koga/pigtest01/dir01/part-m-00000

$ hdfs dfs -cat /user/koga/pigtest01/dir01/part-m-00000
1        Masazumi Koga          43       Tokyo          2500
2        Satoshi Nakamoto       58       Osaka          9500
3        John von Neumann       49       USA            2000

$ hdfs dfs -ls /user/koga/pigtest01/dir02/
Found 2 items
-rw-r--r--  3 koga supergroup ... /user/koga/pigtest02/dir02/_SUCCESS
-rw-r--r--  3 koga supergroup ... /user/koga/pigtest02/dir02/part-m-00000

$ hdfs dfs -cat /user/koga/pigtest01/dir02/part-m-00000
2        Satoshi Nakamoto       58       Osaka          9500
5        Kenneth Harry Olsen    27       USA            8500
```

以上で、Hadoop 3 クラスターで Pig を使ったデータの操作を行いました。

6-5-4　MapR クラスターへの Pig のインストール

MapR クラスター向けの Pig のインストールは、非常に簡単です。MEP から RPM パッケージが提供されており、MapR クラスターの全ノードにおいて、事前に MapR のリポジトリが設定されていれば、yum コマンドでインストール可能です。以下は、クライアントノードで作業します。

```
# hostname
n0130.jpn.linux.hpe.com

# clush -g cl,all -L "cat /etc/yum.repos.d/maprtech.repo"
```

```
...
n0130: [maprecosystem]
n0130: name=MapR Technologies
n0130: baseurl=http://package.mapr.com/releases/MEP/MEP-5.0.0/redhat
n0130: enabled=1
n0130: gpgcheck=0
n0130: protect=1
...
```

■ Pig のインストール

Pig の RPM パッケージ「mapr-pig」を全クラスターノードにインストールします。

```
# clush -g cl,all "yum makecache fast && yum install -y mapr-pig"
```

以上で、Pig を利用できるようになりました。

6-5-5　Pig の動作確認

今度は、Pig を使って、テキストファイルに含まれる単語の出現回数をカウントします。クライアントノード、あるいはクラスターノードのユーザー mapr で作業します。

```
# hostname
n0131.jpn.linux.hpe.com

# su - mapr
$ whoami
mapr
```

MapR-FS にテスト用の/user/mapr/pigtest02 ディレクトリを作成します。

```
$ hadoop fs -mkdir /user/mapr/pigtest02
```

MapR-FS の/user/mapr/pigtest02 ディレクトリに README ファイルをコピーします。

```
$ hadoop fs -put /usr/share/doc/wget-*/README /user/mapr/pigtest02/
```

今回は、pig コマンドが提供する Grunt シェルを起動し、対話形式でデータを操作してみます。Grunt シェルを起動するには、コマンドプロンプトから pig コマンドにオプションを付けずに実

行します。

```
$ pig
...
grunt>
```

Grunt シェルが起動すると、「grunt>」プロンプトが表示されます。以下、Grunt シェルのコマンドプロンプトを「grunt>」で表します。Grunt シェルから、MapR-FS に配置した README ファイルをロードします。長いスクリプト文は、Enter キーを押すと、Grunt シェルのプロンプトが「>>」に変化し、複数行に跨がって入力が可能です。1 行の最後は、「;」が必要です。

```
grunt>  A = LOAD '/user/mapr/pigtest02/README'
>> USING TextLoader() AS (words:chararray);
```

引き続き、Grunt シェルで、以下のように入力します。

```
grunt> B = FOREACH A GENERATE FLATTEN(TOKENIZE(*));
grunt> C = GROUP B BY $0;
grunt> D = FOREACH C GENERATE group, COUNT(B);
grunt> STORE D INTO '/user/mapr/pigtest02/resultdir';
...
 - Success!
```

実行結果において、エラーが出ずに「Success!」と出力されたら、Grunt シェルを終了します。

```
grunt> quit
```

MapR-FS の/user/mapr/pigtest01/resultdir ディレクトリに格納されている実行結果を出力します。

```
$ hadoop fs \
-cat /user/mapr/pigtest02/resultdir/part-r-00000 \
| sort -k2 -rn |head -3
the     40
of      19
Wget    16
```

以上で、MapR-FS に格納されたテキストファイル中の単語ごとの出現回数を Pig で算出することができました。

6-6 まとめ

本章では、Hive、Impala、HBase の簡単な使用方法について解説しました。これらのツールは、MapReduce プログラミングの知識がなくても、Hadoop クラスターをデータベースのように利用できます。従来の RDBMS における SQL の知識さえあれば、Hive、Impala、HBase を利用するハードルは、それほど高くありません。また、複雑な MapReduce プログラミングを学習することなく Hadoop 基盤でのデータ処理を実現する Pig スクリプト言語により、プログラミングの手間を大幅に低減できます。

本書に示した手順は、これらのソフトウェアが備える機能の一部にすぎませんが、ぜひ、本書に示した手順で使い方をマスターし、Hadoop 基盤におけるデータ操作の勘所をつかんでみてください。

第7章

Sqoop/Flume – データのインポート/エクスポート

データ処理の対象となるデータが、オンラインリアルタイム処理を行う本番系のRBDMS に存在する場合には、マスターデータをバックエンドのビッグデータ分析基盤上にインポートする必要があります。RDBMS からビッグデータ分析基盤へのデータのインポートには専用ツールがあり、ビッグデータ分析基盤を活用するには不可欠のツールとなっています。

本章では、RDBMS と Hadoop の間でデータをインポート（またはエクスポート）するためのツールとして有名な「Apache Sqoop」と、データベース以外の非構造化データを取り扱える「Apache Flume」についてその構築手順と使用法を簡単に紹介します。

7-1 Apache Sqoop

Apache Sqoop は、RDBMS と Hadoop クラスター間におけるデータ転送ツールです。外部の RDBMS から Hadoop クラスターへのデータ転送（インポート）、あるいは、Hadoop から外部の RDBMS へのデータ転送（エクスポート）が可能です。また、RDBMS だけでなく、NoSQL データベースなど、さまざまな種類のデータベースに対応しています。

Apache Sqoop は、Pig を使った ETL 処理や、Hive などのクエリー処理を行うソフトウェアと組み合わせて利用されることが少なくありません。典型的な利用例としては、商品を購入した顧客との通話記録データ保管庫から、オペレータと顧客のやり取りに関する定型的な項目を Sqoop で Hadoop 基盤に転送し、Hive の `beeline` コマンドを使ってクエリー処理を実行するような使い方などが挙げられます（図 7-1）。

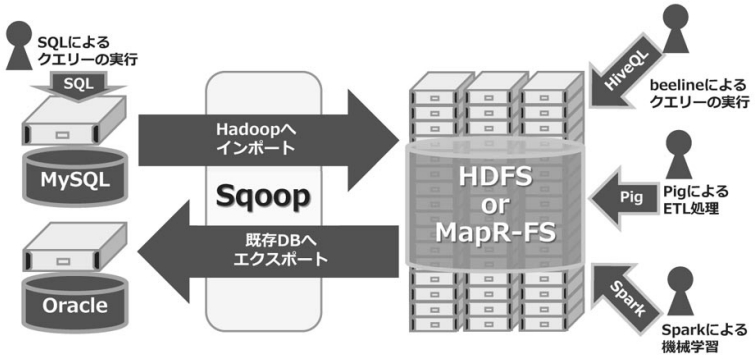


図 7-1 Apache Sqoop の使用例

7-1-1 Apache Sqoop のアーキテクチャ

Apache Sqoop[†] は、通常、Hadoop クラスター内の 1 ノードで稼働させます。ユーザーは、Sqoop クライアント（sqoop コマンド）を使ってデータ転送を行います。RDBMS と Hadoop クラスター間のデータ転送は、Hadoop クラスター上の Map タスクによって並列に処理されます。これにより、ユーザーは、Map タスクの並列処理によるデータ転送のスケールメリットを享受できます。

Sqoopは、RDBMSからHDFSやMapR-FS、そして、HiveやNoSQLのHBaseにデータを直接インポートできます。エクスポートの場合は、HiveやHBaseから直接RDBMSにエクスポートするのではなく、Hadoop分散ファイルシステムに格納されたHiveやHBaseのデータを指定して行います（図7-2）。

† Sqoopには、Sqoop 1とSqoop 2が存在します。Sqoop 1は、Sqoopクライアントから直接Mapタスクを使ったデータ転送の指示を出すアーキテクチャです。一方、Sqoop 2には、Sqoop 2サーバーが存在し、Sqoop 2クライアントからの命令をSqoop 2サーバーが受け取り、Sqoop 2サーバー経由でHadoopのMapタスクによるデータ転送が行われます。本書では、Sqoop 2に比べて運用管理が簡単で、かつ、導入実績が豊富なSqoop 1を取り上げます。

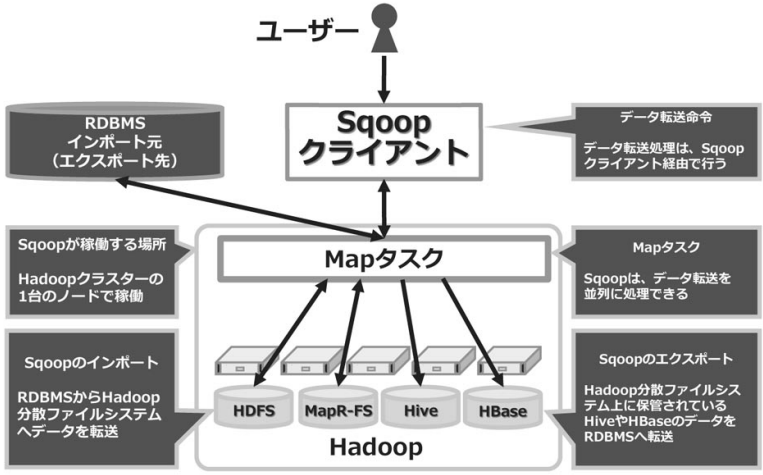


図7-2 Apache Sqoopのアーキテクチャ

Apache SqoopでRDBMSとのデータのやり取りを行うには、Apache SqoopにRDBMS用のJava Database Connectivity（JDBC）ドライバをインストールする必要があります。JDBCを使えば、アプリケーションは、RDBMS上のデータにアクセスできます。Apache Sqoopには、MySQL、PostgreSQL、Microsoft SQL Server、Oracleなどに対応したコネクタも存在します。

7-1-2　Apache Sqoop を使ったデータのインポート例

以下では、MapR クラスターにおいて、Apache Sqoop を使ったデータのインポート例を示します。今回は、RDBMS に PostgreSQL を想定し、MapR 版の Sqoop を MapR クラスターにインストールします（図 7-3）。

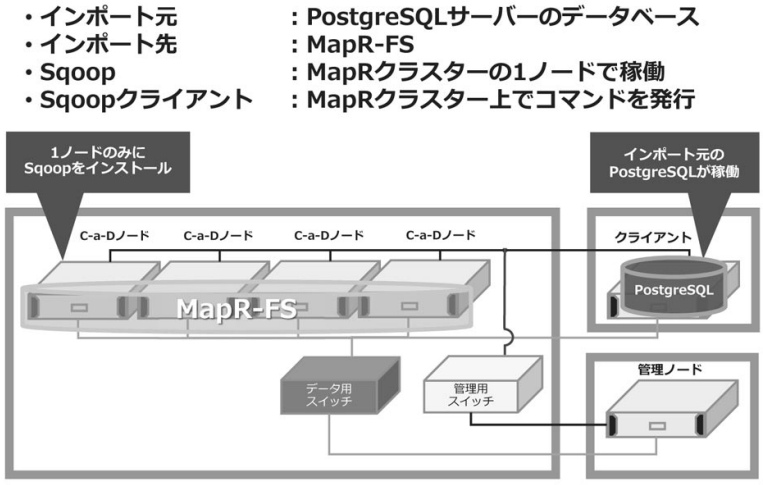


図 7-3　今回構築する PostgreSQL データベースをインポートする Sqoop システム構成

■ PostgreSQL サーバーの用意

Sqoop のインポート元となる PostgreSQL サーバーをクライアントマシンの n0130.jpn.linux.hpe.com に構築します。クライアントマシンの n0130.jpn.linux.hpe.com には、CentOS 7.4 がインストールされ、パッケージが最新の状態に更新されており、インターネット経由で RPM パッケージが入手できる状態とします。

```
# hostname
n0130.jpn.linux.hpe.com
```

古い PostgreSQL がインストールされている場合は、事前にアンインストールしておきます。

```
# yum remove -y postgresql postgresql-libs
```

PostgreSQL 10 のパッケージをインストールします。

```
# yum install -y https://download.postgresql.org/pub/repos/yum/10/redhat/rhel
-7-x86_64/pgdg-centos10-10-2.noarch.rpm  (1 行入力)
# yum install -y postgresql10-server postgresql10
```

■ データベースの初期化

データベースを格納するディレクトリ「/var/lib/pgsql/10/data」が空であることを確認します。

```
# ls -l /var/lib/pgsql/10/data/
total 0
```

PostgreSQL データベースを初期化します。

```
# /usr/pgsql-10/bin/postgresql-10-setup initdb
Initializing database ... OK
```

■ postgresql.conf の設定

PostgreSQL をインストールすると、postgres ユーザーが自動的に作成されます。postgres ユーザーになり、設定ファイル「postgresql.conf」を編集します。postgresql.conf ファイルの「listen_addresses = 'localhost'」を「listen_addresses = '*'」に変更します。

```
# su - postgres
$ whoami
postgres

$ cd /var/lib/pgsql/10/data/
$ cp postgresql.conf postgresql.conf.org
$ sed -i \
"s/#listen_addresses = 'localhost'/listen_addresses = '*'/g" \
./postgresql.conf
```

postgresql.conf ファイルの記述を確認します。

```
$ grep listen_addresses postgresql.conf
listen_addresses = '*'            # what IP address(es) to listen on;
```

■ pg_hba.conf の設定

設定ファイル「pg_hba.conf」を編集し、データベースに接続できるネットワークセグメントを明示的に設定します。

```
$ cp pg_hba.conf pg_hba.conf.org
$ echo "host all all 10.0.0.0/24 password" >> pg_hba.conf
$ echo "host all all 172.16.0.0/16 password" >> pg_hba.conf
```

■ データベースの起動

PostgreSQL サービスを起動します。

```
$ exit
# whoami
root

# systemctl restart postgresql-10.service
# systemctl status postgresql-10.service | grep Active
   Active: active (running) since Tue 2018-04-17 21:29:21 JST; 27s ago
```

■ データベースへの接続

psql コマンドにより PostgreSQL サーバーに接続します。時刻を表示する select 文を発行し、データベースが正常に稼働しているかを確認します。作業は、postgres ユーザーで行います。

```
# su - postgres
$ whoami
postgres

$ psql -U postgres -c "select now();"
              now
-------------------------------
 2018-04-17 22:21:43.778983+09
(1 row)
```

■ パスワードの付与

postgres ユーザーにパスワード「password1234」を付与します。

```
$ psql -c "alter role postgres with password 'password1234'"
ALTER ROLE
```

■ パスワード認証を使った接続確認

パスワード認証を使ったデータベースのアクセスが可能かどうかを確認します。psql コマンドにユーザー名、パスワード、PostgreSQL サーバーのホスト名を明示的に指定します。

```
$ psql \
"user=postgres password=password1234 host=n0130" \
-w -c "select now();"
              now
-------------------------------
 2018-04-17 01:33:24.957472+09
(1 row)
```

誤ったパスワード「abcd1234」では、データベースに接続できないことを確認します。

```
$ psql \
"user=postgres password=abcd1234 host=n0130" \
-w -c "select now();"
psql: FATAL:  password authentication failed for user "postgres"
```

■ テスト用データベースとテーブルの作成

PostgreSQL サーバーにテスト用のデータベース「testdb01」を作成します。

```
$ psql \
"user=postgres password=password1234 host=n0130" \
-w -c "create database testdb01;"
CREATE DATABASE
```

■ テーブルの作成

データベース「testdb01」にテーブル「weather」を作成する SQL 文を記した table01.sql ファイルを用意し、table01.sql を使って、データベース「testdb01」にテーブル「weather」を作成します。テーブル「weather」は、都市名、最低気温、最高気温、降水量、日付からなる表とします。

```
$ cat > table01.sql << __EOF__
\c testdb01;
create table weather (
    city            varchar(80),
    temp_lo         int,
    temp_hi         int,
    prcp            real,
    date            date
);
__EOF__

$ psql \
"user=postgres password=password1234 host=n0130" \
-w < table01.sql
You are now connected to database "testdb01" as user "postgres".
CREATE TABLE
```

■ テーブルの確認

作成したテーブル「weather」の確認用の SQL 文を作成し、テーブルが作成できているかを確認します。

```
$ cat > confirm_table01.sql << __EOF__
\c testdb01;
\dt;
__EOF__

$ psql \
"user=postgres password=password1234 host=n0130" \
-w < confirm_table01.sql
You are now connected to database "testdb01" as user "postgres".
          List of relations
 Schema |  Name   | Type  |  Owner
--------+---------+-------+----------
 public | weather | table | postgres
```

```
(1 row)
```

■ テーブルへの値の挿入

テーブル「weather」に値を挿入するSQL文を作成します。

```
$ cat > insertval_table01.sql << __EOF__
\c testdb01;
insert into weather values('Tokyo ',8,13,0.10,'2017-12-11');
insert into weather values('Osaka ',7,14,0.20,'2018-01-12');
insert into weather values('Nagoya',9,15,0.30,'2018-02-13');
__EOF__
```

■ 値の格納

```
$ psql \
"user=postgres password=password1234 host=n0130" \
-w < insertval_table01.sql
You are now connected to database "testdb01" as user "postgres".
INSERT 0 1
INSERT 0 1
INSERT 0 1
```

■ テーブルの値の確認

テーブル「weather」に挿入した値を確認するSQL文を記述したファイル「confirmval_table01.sql」を作成し、テーブルに格納された値を確認します。

```
$ cat > confirmval_table01.sql << __EOF__
\c testdb01;
select * from weather;
__EOF__

$ psql \
"user=postgres password=password1234 host=n0130" \
-w < confirmval_table01.sql
You are now connected to database "testdb01" as user "postgres".
  city   | temp_lo | temp_hi | prcp |    date
```

```
--------+---------+---------+------+------------
 Tokyo  |       8 |      13 |  0.1 | 2017-12-11
 Osaka  |       7 |      14 |  0.2 | 2018-01-12
 Nagoya |       9 |      15 |  0.3 | 2018-02-13
(3 rows)

$ exit
#
```

以上で、PostgreSQL サーバーにデータベース「testdb01」を作成し、値が格納されたテーブル「weather」が用意できました。

7-1-3　Apache Sqoop のインストール

今回は、MapR クラスターのノード n0131.jpn.linux.hpe.com に Sqoop をインストールします。MapR 社が提供する MEP のリポジトリが設定されていることが前提です。まずは、Sqoop と PostgreSQL JDBC ドライバのパッケージをインストールします。

```
# ssh n0131-mgm
# hostname
n0131.jpn.linux.hpe.com

# yum install -y mapr-sqoop postgresql-jdbc
```

■ JDBC ドライバの設定

PostgreSQL 用の JDBC ドライバを Sqoop に組み込みます。

```
# cd /opt/mapr/sqoop/sqoop-*/lib
# pwd
/opt/mapr/sqoop/sqoop-1.4.6/lib

# ln -s /usr/share/java/postgresql-jdbc.jar .
```

シンボリックリンクを確認します。

```
# ls -l postgresql-jdbc.jar
lrwxrwxrwx 1 root root ... postgresql-jdbc.jar -> /usr/share/java/postgresql-jd
bc.jar
```

■ PostgreSQL から MapR-FS へのデータのインポート

MapR クラスターの n0131.jpn.linux.hpe.com にインストールした sqoop コマンドを使って、クライアントマシン（n0130.jpn.linux.hpe.com）で稼働する PostgreSQL のデータベース「testdb01」上のテーブル「weather」を、MapR-FS 上の/sqoop01 ディレクトリにインポートします。以下の例では、sqoop コマンドを実行すると、「Enter password:」と表示されるので、PostgreSQL の postgres ユーザーのパスワード「password1234」を入力します。

```
# su - mapr
$ whoami
mapr

$ sqoop import \
--connection-manager org.apache.sqoop.manager.GenericJdbcManager \
--connect jdbc:postgresql://n0130.jpn.linux.hpe.com:5432/testdb01 \
--driver org.postgresql.Driver \
--username postgres \
-P \
--target-dir /sqoop01 \
--table weather \
--num-mappers 1
...
Enter password: password1234  ← PostgreSQL の postgres ユーザーのパスワードを入力
...
...
17/12/13 06:24:45 INFO mapreduce.Job: Job job_1512901607255_0012 completed succ
essfully
...
```

「completed successfully」が表示されれば、Sqoop によるインポート処理は完了です。MapR-FS 上に/sqoop01 ディレクトリが作成され、PostgreSQL サーバーからインポートしたデータベースのテーブルが格納されているかを確認します。

```
$ hadoop fs -ls /sqoop01/
Found 2 items
-rwxr-xr-x   3 mapr mapr          0 2018-04-17 00:21 /sqoop01/_SUCCESS
-rwxr-xr-x   3 mapr mapr         81 2018-04-17 00:21 /sqoop01/part-m-00000

$ hadoop fs -cat /sqoop01/part-m-00000
Tokyo ,8,13,0.1,2017-12-11
Osaka ,7,14,0.2,2018-01-12
Nagoya,9,15,0.3,2018-02-13
```

以上で PostgreSQL のデータベースに作成したテーブル情報を MapR-FS 上にインポートできました。

7-2 Apache Flume

Apache Sqoop が、主に RDBMS のテーブル情報を Hadoop にインポートする役目を担うソフトウェアであるのに対し、Apache Flume は、表形式で表されない非構造化データを Hadoop に取り込むためのソフトウェアです。Apache Flume の典型的な使用例としては、ログ集約が挙げられます。Apache Flume は、複数のデータ発生源を想定しており、分散型のデータ収集サービスといえます。また、Apache Flume はさまざまな種類のデータを取り扱うことが可能です。たとえば、Web サーバーのアクセスログ、サーバーで稼働する OS のログ、SNS の投稿文、電子商取引サイトで作成された大量のイベントデータなどが挙げられます（図 7-4）。

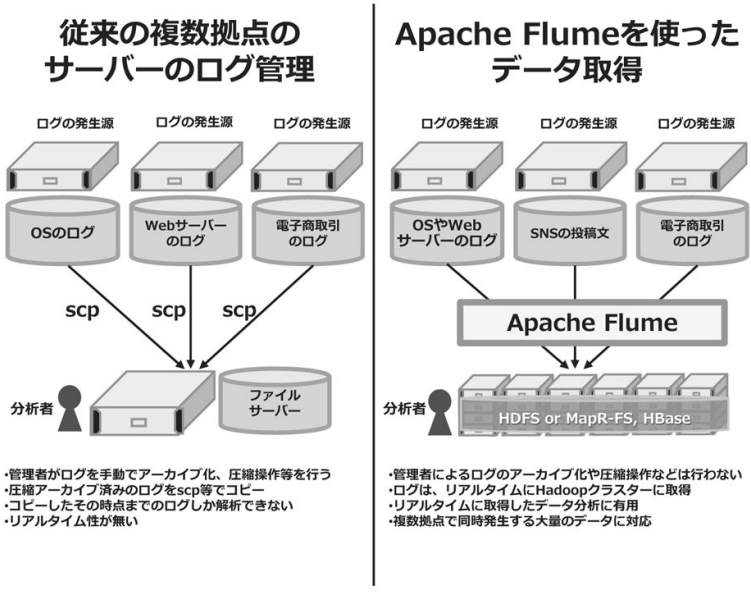


図 7-4　従来の複数拠点サーバーのログ管理と Apache Flume を使ったデータ取得

7-2-1 Apache Flume のアーキテクチャ

Apache Flume では、データの発生源を**データジェネレータ**と呼びます。Twitter や Facebook のソーシャルメディアの投稿文などは、データジェネレータに相当します。データジェネレータによって生成されたデータは、Flume エージェントによって収集されます。その後、Flume エージェントからのデータを収集するのが、**データコレクタ**です。データコレクタは、Hadoop の分散ファイルシステム、あるいは、HBase などにデータをプッシュします（図 7-5、表 7-1）。

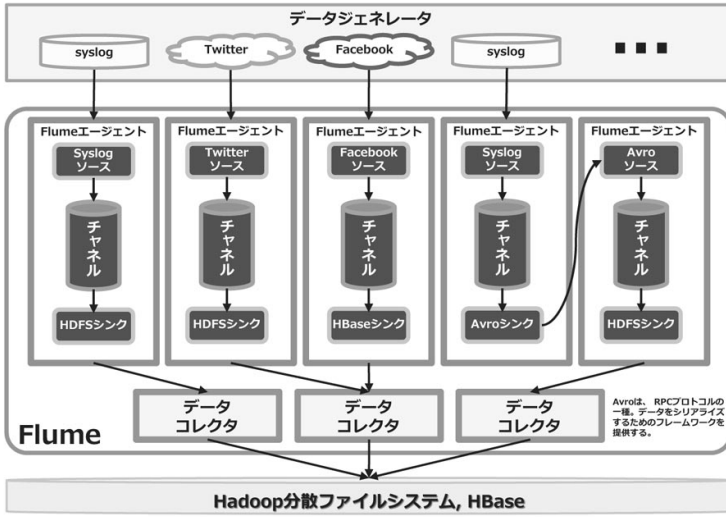


図 7-5 Apache Flume のアーキテクチャ

表 7-1 Flume の主要コンポーネント

コンポーネント名	役割
データジェネレータ	データの発生源
Flume エージェント	データジェネレータが生成したデータを収集
データコレクタ	Flume エージェントのデータを Hadoop 分散ファイルシステムに格納

さらに、Flume エージェントは、ソース、チャネル、シンクの 3 つのコンポーネントで構成されます。

●ソース

ソースはデータジェネレータからのデータを受け取ります。一般に、データジェネレータが Syslog の場合は、Syslog ソース、Twitter ならば Twitter ソースと呼ばれます。また、別の Flume エージェントからのデータを受け取ることも可能です。

●チャネル

チャネルは、ソースから Flume イベントを受信します。イベントとは、Flume エージェント内におけるデータ転送の基本単位です。

●シンク

シンクは、チャネルから来るデータを宛て先となるデータコレクタへ配信します。データの格納先が HDFS であれば、HDFS シンク、HBase ならば、HBase シンクと呼ばれます。

7-2-2 Apache Flume による syslog の取得

以下では、MapR クラスターに Apache Flume をインストールし、syslog が生成するログデータの取得を行います。今回は、MapR クラスターのノードの n0131.jpn.linux.hpe.com に Flume サーバーを構築し、Flume サーバーへの接続元となる rsyslog サーバーをクライアントマシンの n0130.jpn.linux.hpe.com に構築します。クライアントマシンの n0130.jpn.linux.hpe.com には、CentOS 7.4 がインストールされており、インターネット経由で RPM パッケージが入手できる状態とします（図 7-6）。

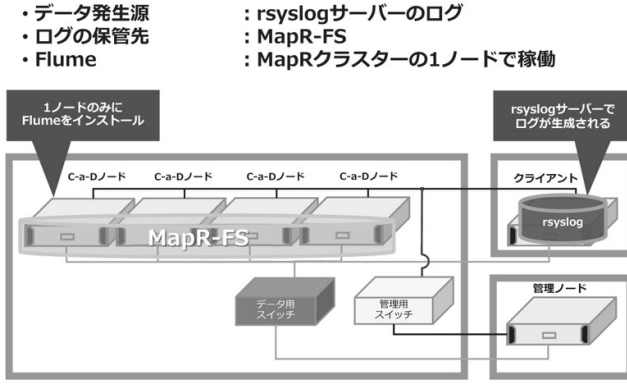


図 7-6 今回構築する syslog データを取得する Flume システム構成

■ rsyslog サーバーの用意

rsyslog のパッケージをインストールし、ログの発生源となる rsyslog サーバーを用意します。

```
# hostname
n0130.jpn.linux.hpe.com

# yum install -y rsyslog
```

■ rsyslog.conf ファイルの設定

n0130.jpn.linux.hpe.com 上で生成されるログを、Flume サーバーの n0131.jpn.linux.hpe.com にフォワードするように rsyslog を設定します。

```
# cp /etc/rsyslog.conf /etc/rsyslog.conf.org
# echo "*.* @@n0131.jpn.linux.hpe.com:7078" >> /etc/rsyslog.conf
```

■ rsyslog サービスの起動

rsyslog サービスを起動し、rsyslog が正常に稼働しているかを確認します。

```
# systemctl restart rsyslog
# systemctl status rsyslog | grep Active
   Active: active (running) since Sun 2018-04-08 03:10:13 JST; 4s ago
```

以上で、rsyslog サーバーでログを生成し、Flume サーバーにフォワードできる準備が整いました。

■ Flume サーバーの構築

MapR クラスターノードの n0131.jpn.linux.hpe.com に Flume をインストールします。MapR 社が提供する MEP のリポジトリが/etc/yum.repos.d/maprtech.repo に正しく設定されていることが前提条件です。

```
# ssh n0131-mgm

# hostname
n0131.jpn.linux.hpe.com
```

```
# whoami
root

# yum install -y mapr-flume
```

■ 設定ファイル flume-env.sh の作成

Flume の設定ファイル「flume-env.sh」を作成します。flume-env.sh ファイルには、環境変数の「JAVA_HOME」を記述します。

```
# cd /opt/mapr/flume/flume-1.8.0/conf
# cat > flume-env.sh << __EOF__
export JAVA_HOME=/usr/lib/jvm/jre
__EOF__
```

■ flume-conf.properties ファイルの作成

flume-conf.properties ファイルを作成します。

```
# cat > flume-conf.properties  << __EOF__
a1.sources = MySrcSyslog  ←1
a1.channels = MyMemChannel  ←2
a1.sinks = MyHadoop  ←3
a1.sources.MySrcSyslog.type = syslogtcp  ←4
a1.sources.MySrcSyslog.host = n0131.jpn.linux.hpe.com  ←5
a1.sources.MySrcSyslog.port = 7078  ←6
a1.sources.MySrcSyslog.keepFields = true
a1.channels.MyMemChannel.type = memory
a1.channels.MyMemChannel.capacity = 10000
a1.channels.MyMemChannel.transactionCapacity = 1000
a1.sinks.MyHadoop.type = hdfs  ←7
a1.sinks.MyHadoop.hdfs.path = maprfs:///flume01/  ←8
a1.sinks.MyHadoop.hdfs.fileType = DataStream  ←9
a1.sinks.MyHadoop.hdfs.writeFormat = Text  ←10
a1.sinks.MyHadoop.hdfs.batchSize = 1000
a1.sinks.MyHadoop.hdfs.rollSize = 0
a1.sinks.MyHadoop.hdfs.rollCount = 10000
a1.sources.MySrcSyslog.channels = MyMemChannel  ←11
a1.sinks.MyHadoop.channel = MyMemChannel  ←12
__EOF__
```

flume-conf.properties ファイルの説明：

1. a1 という名前の Flume エージェントのソースを「MySrcSyslog」と定義
2. a1 のチャネル名を「MyMemChannel」と定義
3. a1 のシンク名を「MyHadoop」と定義
4. a1 のソースのタイプを「syslogtcp」に設定
5. syslog のフォワード先のホストを指定
6. syslog のフォワード先のポート番号を指定
7. a1 のシンクのタイプを「hdfs」に設定
8. a1 のシンクのデータ格納先を MapR-FS の/flume01 ディレクトリに設定
9. a1 のシンクのデータのファイルタイプを「DataStream」に設定
10. a1 のシンクがデータを書き込む際のデータフォーマットをテキストに設定
11. a1 のソース「a1.source.MysrcSyslog」の接続先チャネルを「MyMemChannel」に指定
12. a1 のシンク「a1.sinks.MyHadoop」の接続先チャネルを「MyMemChanel」に指定

■ ログ格納用ディレクトリの作成

rsyslog サーバーが生成されるログの格納先ディレクトリ「/flume01」を MapR-FS に作成します。

```
# hostname
n0131.jpn.linux.hpe.com

# su - mapr
$ whoami
mapr

$ hadoop fs -mkdir /flume01
```

■ Flume エージェントの起動

Flume エージェントを起動します。Flume エージェントは、flume-ng コマンドに agent を付与して起動します。Flume エージェントを起動すると、ターミナルエミュレータのコマンドライン上では、flume-ng によるメッセージが表示され、コマンドプロンプトが表示されず、フォアグラウンドで稼働している状態† になっているはずです。

† フォアグラウンドで起動している Apache Flume は、Ctrl ＋ C キーを押すと強制的に停止できます。

```
$ /opt/mapr/flume/flume-1.8.0/bin/flume-ng agent \
-n a1 \
-c conf \
-f /opt/mapr/flume/flume-1.8.0/conf/flume-conf.properties \
-Dflume.root.logger=INFO,console
```

表 7-2　flume-ng コマンドのオプション

flume-ng コマンドで指定したオプション	意味
-n	エージェントの名前を指定
-c	conf ディレクトリ内の設定ファイルを使用
-f	設定ファイルのパスを指定
-D	Java システムプロパティーの値をセット

■ rsyslog サーバーのログを Flume 経由で MapR-FS へ格納する

MapR クラスターの n0131.jpn.linux.hpe.com にインストールした Flume を使って、ログ生成元のマシン（n0130.jpn.linux.hpe.com）で稼働する rsyslog のログを、MapR-FS 上に格納します。別のターミナルエミュレータを開き、rsyslog サーバーにログインし、logger コマンド† を使って、テスト用のログを生成します。

```
# hostname
n0130.jpn.linux.hpe.com

# logger -t test "Hello Flume."
```

† logger コマンドは、util-linux RPM パッケージに含まれています。

rsyslog サーバーで生成したテスト用のログが MapR-FS 上の/flume01 ディレクトリに格納されているかどうかを確認します。

```
# su - mapr
$ hadoop fs -ls /flume01/
...
Found 1 items
```

```
-rwxr-xr-x   3 mapr mapr  45 2018-04-08 03:44 /flume01/FlumeData.1514002922543

$ hadoop fs -cat /flume01/FlumeData.1520448264604 2>/dev/null | grep Hello
<13>Apr  8 03:44:39 n0130 test: Hello Flume
```

以上で、rsyslog サーバーのログを MapR-FS 上に格納できました。

7-2-3　Apache Flume による Twitter データの取得

以下では、Apache Hadoop 3 クラスターに Apache Flume をインストールし、Twitter のツイートデータを Apache Flume 経由で取得します。Flume サーバーは、Apache Hadoop 3 クラスターノードの n0121.jpn.linux.hpe.com に構築します。今回は、社内の Apache Hadoo 3 クラスターで稼働する Flume エージェントがプロキシサーバーを経由してインターネットにアクセスし、Twitter のツイートデータを取得する構成とします（図 7-7）。

- **データ発生源　　：Twitter**
- **データの保管先　：HDFS**
- **Flume　　：Apache Hadoop 3クラスターの1ノードで稼働**

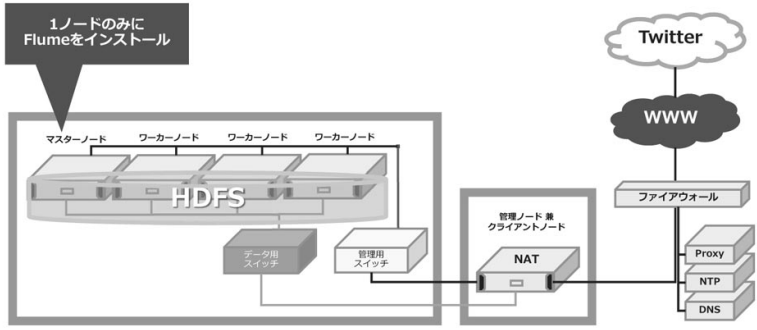


図 7-7　今回構築する Twitter の投稿データを取得する Flume システム構成

■ Twitter での事前準備

Twitter のデータを取得するには、Twitter にログインし、事前に Twitter アカウントの携帯電話番号の設定が必要です。Twitter アカウントの携帯電話番号は、Twitter にログイン後、画面右上にあるアカウントのアイコンをクリックし、プルダウンメニューで表示される［設定とプライバシー］

をクリックします（図 7-8）。

図 7-8　Twitter アカウントのアイコンをクリックしたら、[設定とプライバシー] をクリック

画面左側の［モバイル］をクリックします。すると、画面右側に携帯電話番号の入力欄が表示されますので、［国/地域］で日本を選択後、携帯電話番号を入力します。たとえば、携帯電話番号が、「090-1234-XXXX」の場合は、「+81」の後に「090」の先頭のゼロを抜いて「901234XXXX」を入力し、その下にある［続ける］をクリックします（図 7-9）。

図 7-9　携帯電話番号の入力

携帯電話に認証用コードが送られてきますので、その数字を画面上の「**認証用コード**」の右隣に入力し、［**携帯電話を認証する**］をクリックします (図 7-10)。

図 7-10　認証用コードの入力

以上で、Twitter アカウントの携帯電話番号の設定が完了しました。

次に、Twitter Apps の Web サイト「`https://apps.twitter.com`」にアクセスします (図 7-11)。

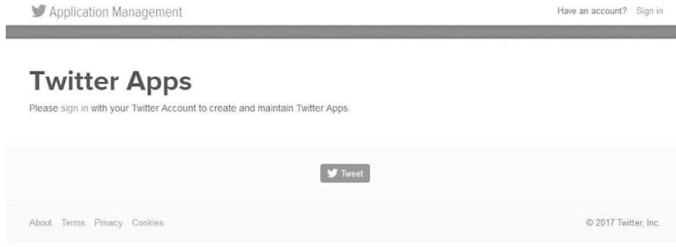


図 7-11　Twitter Apps Web サイトのサインイン画面

Twitter にサインインしていない場合は、［sign in］をクリックして Twitter にサインインします (図 7-12)。

図 7-12　Twitter のサインイン画面

Twitter Apps の Web サイトが表示されたら、［Create New App］をクリックします（図 7-13）。

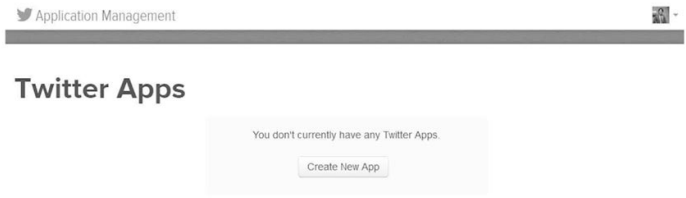


図 7-13　Twitter Apps で［Create New App］をクリック

「Create an Application」の画面が表示されたら、「Application Details」内の Name、Description、Website を入力します。Flume の動作に直接影響はしませんが、アプリケーションの説明としてわかりやすい情報を設定しておきます（図 7-14）。

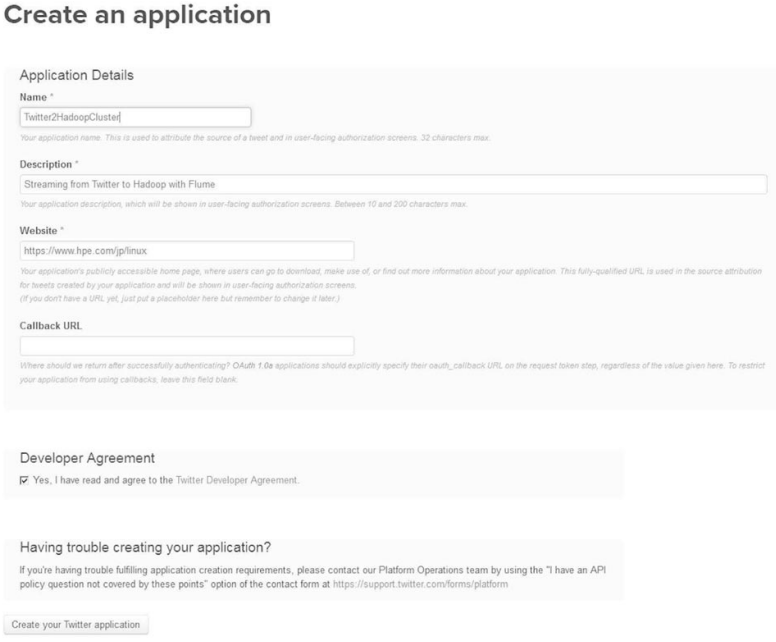


図 7-14　「Application Details」内の Name、Description、Website を入力し、一番下の「Create your Twitter application」をクリック

今回は、アプリケーション名を「Twitter2HadoopCluster」としました。さらに、画面下の

「Developer Agreement」内のチェックボックスにチェックを入れ、画面下の「Create your Twitter application」をクリックします。

アプリケーション「Twitter2HadoopCluster」に関する画面が表示されます（図 7-15）。

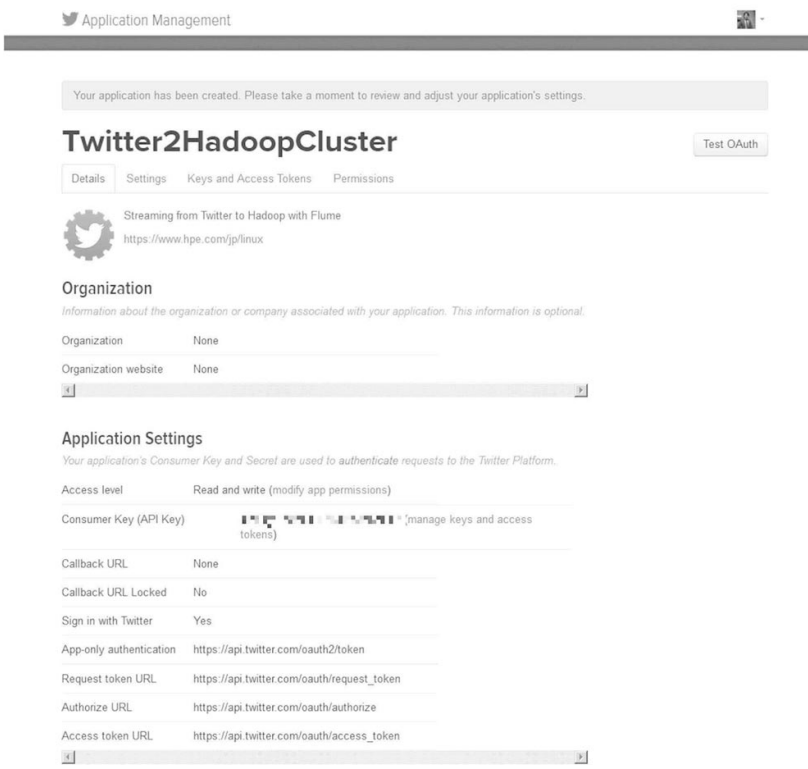


図 7-15　アプリケーション［Twitter2HadoopCluster］に関する画面

アプリケーション「Twitter2HadoopCluster」の画面内の「Keys and Access Token」タブをクリックします。すると、アプリケーションへの接続に必要なキー（Consumer Key）やアクセストークンに関する情報（Access Token、および、Access Token Secret）が表示されます（図 7-16）。

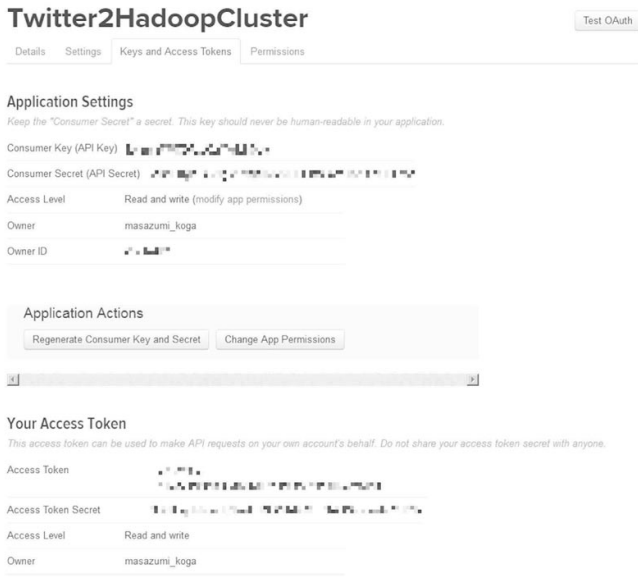


図 7-16　アプリケーション［Twitter2HadoopCluster］の画面内の［Keys and Access Token］タブをクリック

アプリケーション「Twitter2HadoopCluster」の画面内の「Keys and Access Token」タブ内で表示されているキーやアクセストークン情報は、Flume の設定で必要になります。

7-2-4　Apache Hadoop 3 クラスターにおける Flume の設定

Apache Hadoop 3 クラスターに Apache Flume を構築します。今回は、Apache Hadoop 3 クラスターのマスターノードである「n0121.jpn.linux.hpe.com」に Apache Flume をインストールします。マスターノードにログインし、Apache Flume の tar アーカイブを入手し、/opt ディレクトリに展開します。

```
# ssh n0121-mgm

# hostname
n0121.jpn.linux.hpe.com

# wget \
http://www-us.apache.org/dist/flume/1.8.0/apache-flume-1.8.0-bin.tar.gz
```

```
# tar xzvf apache-flume-1.8.0-bin.tar.gz -C /opt
```

■ 環境変数の設定

Apache Flume の環境変数「FLUME_HOME」を設定します。

```
# cat >> $HOME/.bash_profile  << __EOF__
export FLUME_HOME=/opt/apache-flume-1.8.0-bin
__EOF__

# . $HOME/.bash_profile
# ls -F $FLUME_HOME
bin/        conf/     doap_Flume.rdf  lib/     NOTICE      RELEASE-NOTES
CHANGELOG  DEVNOTES  docs/           LICENSE  README.md  tools/
```

■ flume-env.sh ファイルの設定

Apache Flume の設定ファイル「flume-env.sh」に環境変数「JAVA_HOME」を記述します。

```
# cat > $FLUME_HOME/conf/flume-env.sh << __EOF__
export JAVA_HOME=/usr/lib/jvm/jre
__EOF__
```

■ flume-conf.properties ファイルの設定

Twitter のデータを取得する Flume の設定ファイル「flume-conf.properties」を編集します。設定ファイル内の 6 、 7 、 8 、 9 （以下の例では、「XXXX...XXXX」と記載されている箇所）には、Twitter アプリケーション「Twitter2HadoopCluster」の画面に表示された「Consumer Key」「Consumer Secret」「Access Token」「Access Token Secret」を記述します。

```
# vi $FLUME_HOME/conf/flume-conf.properties
t2h.sources = MyTwitter  ←1
t2h.channels = MyMemChannel  ←2
t2h.sinks = MyHadoop  ←3
t2h.sources.MyTwitter.type = org.apache.flume.source.Twitter.TwitterSource  ←4
t2h.sources.MyTwitter.channels = MyMemChannel  ←5
t2h.sources.MyTwitter.consumerKey = XXXXXXX...XXXXXXXXX  ←6
```

```
t2h.sources.MyTwitter.consumerSecret = XXXXXXXXXXXXXXX...XXXXXXXXXXXXXXXXX  ←7
t2h.sources.MyTwitter.accessToken = XXXXXXXXXX...XXXXXXXXXXXXXXXXXXXXX  ←8
t2h.sources.MyTwitter.accessTokenSecret = XXXXXXXXXX...XXXXXXXXXXXXXXXXX  ←9
t2h.sinks.MyHadoop.channel = MyMemChannel  ←10
t2h.sinks.MyHadoop.type = hdfs  ←11
t2h.sinks.MyHadoop.hdfs.path = hdfs://n0121:9000/user/koga/tweetdata/  ←12
t2h.sinks.MyHadoop.hdfs.fileType = DataStream  ←13
t2h.sinks.MyHadoop.writeFormat = Text  ←14
t2h.sinks.MyHadoop.batchSize = 1000
t2h.sinks.MyHadoop.rollCount = 10000
t2h.sinks.MyHadoop.rollInterval = 600
t2h.sinks.MyHadoop.rollSize = 0
t2h.channels.MyMemChannel.type = memory
t2h.channels.MyMemChannel.capacity = 10000
t2h.channels.MyMemChannel.transactionCapacity = 100
```

flume-conf.properties ファイルの説明：

1. t2h という名前の Flume エージェントのソースを「MyTwitter」と定義
2. t2h のチャネル名を「MyMemChannel」と定義
3. t2h のシンク名を「MyHadoop」と定義
4. t2h のソースのタイプを「org.apache.flume.source.Twitter.TwitterSource」に設定
5. t2h のソース「t2h.source.MyTwitter」の接続先チャネルを「MyMemChannel」に指定
6. Twitter アプリケーション「Twitter2HadoopCluster」の Consumer Key を指定
7. Twitter アプリケーション「Twitter2HadoopCluster」の Consumer Secret を指定
8. Twitter アプリケーション「Twitter2HadoopCluster」の Access Token を指定
9. Twitter アプリケーション「Twitter2HadoopCluster」の Access Token Secret を指定
10. t2h のシンク「t2.sinks.MyHadoop」の接続先チャネルを「MyMemChanel」に指定
11. t2h のシンクのタイプを「hdfs」に設定
12. t2h のシンクのデータ格納先を HDFS の/user/koga/tweetdata ディレクトリに設定
13. t2h のシンクのデータのファイルタイプを「DataStream」に設定
14. t2h のシンクがデータを書き込む際のデータフォーマットをテキストに設定

■ Twitter のツイートデータ格納用ディレクトリの作成

flume-conf.properties ファイルと矛盾がないように、HDFS に Twitter のツイートデータ格納用ディレクトリを作成します。

```
# hostname
n0121.jpn.linux.hpe.com

# su - koga
$ whoami
koga
```

```
$ which hadoop
/opt/hadoop-3.1.0/bin/hadoop

$ hdfs dfs -mkdir /user/koga/tweetdata
$ exit
```

■ Flume エージェントの起動

Flume エージェントを起動します。flume-ng コマンドにおいて、「-Dtwitter4j.http.proxyHost」オプションに社内のプロキシサーバー「proxy.your.site.com」を指定しています。プロキシサーバー経由で、Flume を使って Twitter データを取り込む際に必要なオプションです。

```
# hostname
n0121.jpn.linux.hpe.com

# printenv FLUME_HOME
/opt/apache-flume-1.8.0-bin

# $FLUME_HOME/bin/flume-ng agent \
-n t2h \
-c conf \
-f $FLUME_HOME/conf/flume-conf.properties \
-Dflume.root.logger=INFO,console \
-Dtwitter4j.http.proxyHost=proxy.your.site.com \
-Dtwitter4j.http.proxyPort=8080
```

Note　全ノードにおける時刻の確認

Flume エージェントを稼働させるノードが正しい時刻を刻んでいない場合は、Twitter データの取得に失敗する可能性があります。Flume エージェントを起動する前に、Hadoop の全クラスターノードが正しい時刻を刻んでいるかを確認してください。

データの取り込みに成功すると、Twitter のツイートデータを取得し、HDFS に記録された旨のログが次々と表示されるはずです。

■ ツイートデータの確認

別のターミナルエミュレータを開いて、Flume が稼働している n0121.jpn.linux.hpe.com にログインし、Twitter のツイートデータが HDFS 上に格納されているかを確認します。

```
# ssh -l koga n0121-mgm

$ hostname
n0121.jpn.linux.hpe.com

$ hdfs dfs -ls tweetdata/
...
-rw-r--r--   3 root supergroup      15147 2018-04-18 09:56 tweetdata/FlumeData.
1520470586402
-rw-r--r--   3 root supergroup      16640 2018-04-18 09:56 tweetdata/FlumeData.
1520470586403
...
```

Twitter で生成されたツイートを Flume 経由で Apache Hadoop 3 クラスターの HDFS 上に格納できました。

■ 取得したツイートデータの確認

HDFS に格納されたツイートデータの中身を確認します。ツイートデータを JSON 形式に変換して中身を確認するため、「avro-tools」と呼ばれるソフトウェアを入手します。

```
$ wget http://www-us.apache.org/dist/avro/avro-1.8.2/java/avro-tools-1.8.2.jar
```

HDFS に格納されたツイートデータをローカルにコピーします。

```
$ hdfs dfs -get tweetdata/FlumeData.1520470586402
```

avro-tools を使って、ファイルの中身を確認します。

```
$ java -jar \
./avro-tools-1.8.2.jar tojson \
--pretty ./FlumeData.1520470586402 | grep string | less
...

...
    "string" : "abcdefg"
...
```

```
    "string" : "2018-03-08T09:56:44Z"
...
    "string" : "RT ...
...
...
```

7-3 まとめ

本章では、Apache Sqoop を使ったデータのインポート例として、PostgreSQL のデータベースに作成したテーブル情報を MapR-FS 上へ移す事例を取り上げ、Apache Flume の事例では、syslog データと Twitter のツイートデータの取得について説明しました。

Apache Sqoop は、さまざまな RDBMS をサポートしているため、各部門に散在する RDBMS のデータを 1 つの Hadoop クラスターに取り込んで分析するシステムには、必要不可欠です。あらゆる RDBMS を対象とした統合分析基盤としてぜひ、Apache Sqoop を活用してみてください。

一方、Apache Flume は、膨大なログデータや SNS のデータをリアルタイムに取得し、Hadoo クラスターに格納します。Apache Flume によってリアルタイムに得られた非構造化データは、機械学習ツールを組み合わせることで、行動パターン分析や感情分析などに活用できます。ぜひ、ApacheFlume と機械学習ツールを導入し、非構造化データを駆使した知的情報処理基盤の構築にチャレンジしてみてください。

Column　Kafka と MapR-ES

データストリーミングの世界では、Spark Streaming や Flume だけでなく、データストリーミングのためのメッセージングシステムとして、Kafka（カフカ）も有名です。Kafka は、スケールアウトメリットを享受でき、テラバイト級のデータを処理できます。また、Hadoop 基盤だけでなく、Vertica などの列指向データベース基盤とのデータのやり取りもサポートしています。

この Kafka よりもさらに性能を向上させたツールとして、MapR 社では MapR-ES（MapR Event Streams、旧名は MapR Streams) を提供しています。MapR のコミュニティサイトでは、MapR-ES と Kafka の比較表、ベンチマーク比較情報、Flume と MapR-ES の連携に関する情報などを提供しています。IoT を利用する場合には、さまざまな事業所や機器から生成され続ける膨大なデータを、Hadoop 基盤に絶え間なく格納できるかどうかの性能試験を行う必要がありますが、これらの MapR 社が提供する資料が役に立ちます。データストリーミングを検討している方は、一読をおすすめします。

【参考情報】

Kafka vs. MapR Streams: Why MapR?：

`https://mapr.com/blog/kafka-vs-mapr-streams-why-mapr/`

Integrate Flume with MapR-ES：

`https://maprdocs.mapr.com/home/Flume/Flume_IntegrateWithMapRStreams.html`

Apache Kafka vs MapR-ES: Fit for purpose/Decision tree：

`https://community.mapr.com/docs/DOC-2409-apache-kafka-vs-mapr-es-fit-for-purposedecision-tree`

第８章

Mahout – 機械学習（クラス分類と協調フィルタリング）

人間が処理するには作業量が膨大で、複雑な知的情報処理を遂行する要素技術として注目を浴びているのが、最近話題の機械学習です。ここで紹介する Apache Mahout は、Hadoop で稼働する、機械学習ソフトウェアの草分け的な存在です。Apache Spark 登場以前から世界中で利用されており、事例も豊富です。本章では、世界中で広く利用されている Hadoop 対応の機械学習ライブラリの「Apache Mahout」を利用したクラス分類とレコメンデーションの事例について取り上げます。

8-1 Apache Mahout とは？

インターネットを通じてさまざまな人がリアルタイムで意見を発信するソーシャルメディア時代では、顧客の動向や嗜好を読み取り、製品やサービスの改善などに素早く対応することにより、顧客対応の質向上や、SNS における人びとの発言と連動した広告の表示、購買確度が高い商品の自動的な提案など、従来の定型業務に比べると非常に複雑度が高い業務が増えてきています。

こうした業務を実現するには、以下のようなデータ分析と出力が必要です。

- 自社の膨大なデータ内に潜む傾向をあらゆる角度から検討する。
- 過去のデータの傾向から精度よく顧客の動向を予測する。
- 顧客に短時間で極めて妥当と判断できる回答を返す。

しかし、従来の定型業務に比べ、このような業務を実現するアルゴリズムをゼロから作ることは、容易なことではありません。Apache Mahout は、こうした課題を解決するための機械学習のアルゴリズムとともに、機械学習の開発基盤、および、実行基盤を提供します。

8-1-1 Mahout の主な機能

Apache Mahout は、2008 年、Apache の検索エンジンライブラリである Lucene のサブプロジェクトとして開発が始まり、その後、Apache のトップレベルプロジェクトになりました。Apache Mahout は、分類、クラスタリング、協調フィルタリングなどの機械学習で一般に利用される機能を搭載しています。

Apache Mahout では、分析を行う計算処理のタスクを複数区分に分解し、区分ごとに複数のマシンで処理するフレームワークを提供します。Apache Mahout で実装されている機械学習のアルゴリズムは、Hadoop クラスターの分散環境で動作します。Hadoop が提供するライブラリを Mahout が利用し、分散アーキテクチャによって、機械学習の性能がスケールします。Apache Mahout で実装されている主な機能を以下に示します。

●分類

コンピュータシステムで取り扱う文書には、Web 記事、メール、オフィス文書など、さまざまな種類の文書が存在します。さまざまな文書の中から、傾向を見出し、予測し、知見を得る（分類する）といった分析が行われます。たとえば、迷惑メールの判定は、文書分類に該当しますが、迷惑メールも多種多様であり、コンピュータに迷惑メールを学習させ、妥当

性のある精度の高い結果を得るには、膨大な計算量が必要になります。分類を行うアルゴリズムとしては、ナイーブ・ベイズ、ロジスティック回帰、ランダム木、隠れマルコフモデル（Hidden Markov Model、HMM）などが利用されます。

●クラスタリング

機械学習の世界に限らず、大量のデータの中から類似しているものをグルーピングすることは、データ分析において非常に重要です。たとえば、大量の論文データが存在し、同じ研究課題の論文をグルーピングする場合、クラスタリングのアルゴリズムが利用されます。また、IoT の世界では、センサから生成されるデータを取得し、クラスタリングすることにより、正常稼働なのか、異常が発生しているのかを判定することにも利用できます。クラスタリングの手法としては、**k 平均法**や**階層型クラスタリング**が有名であり、Mahout では、これらのクラスタリングのアルゴリズムが利用可能です。

●協調フィルタリング

協調フィルタリングは、ネットショッピングサイトでの**商品のおすすめ**機能で利用されています。ユーザーの購買履歴や Web サイトのクリック数、商品の評価などのデータを駆使し、別のユーザーに対して、おすすめの商品を表示します。ユーザーに対するおすすめ商品の表示は、**レコメンデーション**と呼ばれます。レコメンデーションでは、似ているユーザーに対して商品を推薦する、あるいは、複数の商品の類似性を数値化し、似ている商品も一緒にユーザーに推薦します。さらに、ユーザーが行った評価の値、複数のユーザーと評価のモデル作成し、レコメンデーションを行うこともあります。似ているユーザー、似ている商品、評価の値など、いずれのレコメンデーションも、ユーザーが評価を下した複数商品の類似性を数値化する処理です。

8-1-2　Apache Mahout のインストール

ここでは、Apache Mahout を Apache Hadoop 3 クラスターで利用する手順を紹介します。Apache Mahout は、root 権限がない一般ユーザーでインストールが可能です。2018 年 4 月時点で、Apache Mahout は、0.13.0 が最新版です。今回は、Apache コミュニティが提供している tar アーカイブを入手します。入手した tar アーカイブをユーザーのホームディレクトリにコピーし、tar コマンドで展開します。root 権限のコマンドプロンプトは「#」、一般ユーザーのプロンプトは「$」で表します。作業は、クライアントノードで行います。

Apache Mahout の入手先：

http://archive.apache.org/dist/mahout/0.13.0/

```
$ hostname
n0120.jpn.linux.hpe.com

$ whoami
koga

$ wget \
https://archive.apache.org/dist/mahout/0.13.0/apache-mahout-distribution-
0.13.0.tar.gz  (1 行入力)
```

「apache-mahout-distribution-0.13.0.tar.gz」を入手し、ユーザー koga のホームディレクトリに展開します。Apache Mahout の tarball を展開すると、apache-mahout-distribution-0.13.0 ディレクトリが生成され、そのディレクトリ配下に起動スクリプトなどが含まれています。

```
$ tar xzvf apache-mahout-distribution-0.13.0.tar.gz -C $HOME/
```

8-1-3　Apache Mahout のデモプログラムの設定

Apache Mahout が HDFS にアクセスするためのシェル変数などを設定するスクリプト「set-dfs-commands.sh」が Apache Hadoop 3 で正常に稼働するように、スクリプトを修正します。set-dfs-commands.sh スクリプト内の以下の箇所を変更します。

表 8-1　set-dfs-commands.sh スクリプトの変更箇所

変更前	変更後
「$v -eq "2"」	「$v -eq "3"」
「Discovered Hadoop v2.」	「Discovered Hadoop v3.」

```
$ cd $HOME/apache-mahout-distribution-0.13.0/examples/bin/
$ cp set-dfs-commands.sh set-dfs-commands.sh.org
$ vi set-dfs-commands.sh
...
elif [ $v -eq "3" ]
```

```
then
  echo "Discovered Hadoop v3."
...
```

ユーザー koga の$HOME/.bash_profile ファイルに環境変数「MAHOUT_HOME」と実行ファイルのPATH を追加します。また、環境変数「MAHOUT_LOCAL」に「""」を設定します。

```
$ cat >> $HOME/.bash_profile << '__EOF__'
export MAHOUT_HOME=$HOME/apache-mahout-distribution-0.13.0
export PATH=$MAHOUT_HOME/bin:$MAHOUT_HOME/examples/bin:$PATH
export MAHOUT_LOCAL=""
__EOF__
```

Hadoop の環境変数「HADOOP_CONF_DIR」が設定済みであることを確認します。

```
$ printenv HADOOP_CONF_DIR
/opt/hadoop-3.1.0/etc/hadoop
```

環境変数をロードします。

```
$ . $HOME/.bash_profile
```

mahout コマンドが実行できるかを確認します。

```
$ which mahout
~/apache-mahout-distribution-0.13.0/bin/mahout

$ mahout
MAHOUT_LOCAL is not set; adding HADOOP_CONF_DIR to classpath.
Running on hadoop, using /opt/hadoop-3.1.0/bin/hadoop and HADOOP_CONF_DIR=/opt/
hadoop-3.1.0/etc/hadoop
MAHOUT-JOB: /opt/apache-mahout-distribution-0.13.0/mahout-examples-0.13.0-job.j
ar
...
...
```

さまざまなオプションが表示されます。以上で、mahout コマンドが利用できるようになりました。

Column　Apache Mahout の最新版を試す

Apache Mahout は、コミュニティによって機械学習用のモジュールが改良されています。最新のモジュールを試す場合は、Apache Mahout の最新版を GitHub から入手してビルドするとよいでしょう。以下は、CentOS 7.4 環境において、プロキシーサーバー経由でインターネットにアクセスし、Apache Mahout の最新版の入手、ビルド、インストールを行う手順です。

1. クライアントノードで開発環境のツール群をインストール

```
# export http_proxy=http://proxy.your.site.com
# export https_proxy=http://proxy.your.site.com
# yum install -y epel-release
# yum groupinstall -y "Development Tools"
# yum install -y git
```

2. Java 用のプロジェクト管理ツール「Apache Maven」をインストール

```
# wget \
http://repos.fedorapeople.org/repos/dchen/apache-maven/\
epel-apache-maven.repo \
-O /etc/yum.repos.d/epel-apache-maven.repo

# yum install -y apache-maven
# mvn -version
```

3. Apache Maven の HTTP プロキシーに関する環境変数と Apache Mahout の環境変数を設定

```
# su - koga
$ cat >> $HOME/.bash_profile << '__EOF__'
export MAHOUT_HOME=$HOME/mahout
export PATH=$MAHOUT_HOME/bin:$MAHOUT_HOME/examples/bin:$PATH
export MAHOUT_LOCAL=""
export MAVEN_OPTS="-Dhttp.proxyHost=proxy.your.site.com -Dhttp.proxyPort=80
80 -Dhttps.proxyHost=proxy.your.site.com -Dhttps.proxyPort=8080"  (1 行入力)
__EOF__
```

4. Apache Mahout の最新版を git コマンドで入手し、Apache Maven の mvn コマンドを使ってビルド

```
$ cd $HOME/
$ . $HOME/.bash_profile
$ git clone https://github.com/apache/mahout
$ cd mahout
```

```
$ mvn -DskipTests clean install


5. ビルドされた Apache Mahout の最新版に付属する mahout コマンドの実行を確認

$ which mahout
~/mahout/bin/mahout
```

8-2 Apache Mahout の実行例

ここで、Mahout の分類機能と協調フィルタリングの機能を確認する事例として、2 つのサンプルプログラムを実行してみましょう。分類機能の事例として Apache Mahout に用意されている、Wikipedia のドキュメントを分類するためサンプルスクリプトを、協調フィルタリング機能としては、映画のレコメンデーションを取り上げます。

8-2-1 Wikipedia ドキュメントの分類

Apache Mahout に搭載されている分類エンジンを使って、Wikipedia のドキュメントを分類してみます。

■ サンプルスクリプトの修正

Wikipedia のドキュメントを分類するためサンプルスクリプト「classify-wikipedia.sh」は、内部で curl コマンドを使って Wikipedia のドキュメントをダウンロードします。この Wikipedia のドキュメントの URL を変更します。

```
$ cd $MAHOUT_HOME/examples/bin
$ cp -a classify-wikipedia.sh classify-wikipedia.sh.org
$ vi ./classify-wikipedia.sh
...
curl https://dumps.wikimedia.org/enwiki/latest/enwiki-latest-pages-articles10
.xml-p2336425p3046511.bz2 -o ${WORK_DIR}/wikixml/enwiki-latest-pages-articles.x
ml.bz2   (1行入力)
```

```
...
```

事前に、一時ディレクトリを削除しておきます。

```
$ rm -rf /tmp/mahout-work-wiki/
```

■ 環境変数の確認

環境変数を確認しておきます。プロキシーサーバー経由でインターネットにアクセスする環境の場合は、http_proxy、および、https_proxy の環境変数を確認しておきます。また、環境変数「MAHOUT_HOME」も正しく設定されていることを確認します。

```
$ printenv http_proxy
http://proxy.your.site.com:8080

$ printenv https_proxy
http://proxy.your.site.com:8080

$ printenv MAHOUT_HOME
/home/koga/apache-mahout-distribution-0.13.0
```

■ Wikipedia ドキュメントの分類

clasify-wikipedia.sh を使って、Wikipedia のドキュメントの分類を行います。この clasify-wikipedia.sh スクリプトは、国名を分類項目として Wikipedia ドキュメントを分類します。clasify-wikipedia.sh スクリプトを実行すると、以下のように、1 から 3 までの選択項目が表示されます。1 番を選択すると、Complement Naive Bays（CBayes）と呼ばれるナイーブベイズ分類器の拡張版を使って機械学習が実行されます。今回は、Wikipedia ドキュメントについて、10 か国の国名を分類項目として利用します。2 番を選択すると、分類項目として、2 か国の国名（「United States」と「United Kingdom」）を使って CBayes で機械学習が実行されます。3 番を選択すると、機械学習で利用するデータの一時ディレクトリが削除されます。今回は、1 番を選択します。

```
$ ./classify-wikipedia.sh
Discovered Hadoop v3.
Setting dfs command to /opt/hadoop-3.1.0/bin/hdfs dfs, dfs rm to /opt/hadoop-3.
0.0/bin/hdfs dfs -rm -r -skipTrash.
```

```
Please select a number to choose the corresponding task to run
1. CBayes (may require increased heap space on yarn)
2. BinaryCBayes
3. clean -- cleans up the work area in /tmp/mahout-work-wiki
Enter your choice :
...
```

しばらくすると、結果が表示されます（図 8-1）

```
File  Edit  View  Search  Terminal  Help
Running on hadoop, using /opt/hadoop-3.0.0/bin/hadoop and HADOOP_CONF_DIR=/opt/hadoop-3.0.0/etc/hadoop
MAHOUT-JOB: /home/koga/mahout/examples/target/mahout-examples-0.13.1-SNAPSHOT-job.jar
18/03/10 07:50:58 WARN MahoutDriver: No testnb.props found on classpath, will use command-line arguments only
18/03/10 07:50:58 INFO AbstractJob: Command line arguments: {--endPhase=[2147483647], --input=[/tmp/mahout-work-wiki/testing]
, --labelIndex=[/tmp/mahout-work-wiki/labelindex], --model=[/tmp/mahout-work-wiki/model], --output=[/tmp/mahout-work-wiki/out
put], --overwrite=null, --runSequential=null, --startPhase=[0], --tempDir=[temp], --testComplementary=null}
18/03/10 07:50:59 INFO HadoopUtil: Deleting /tmp/mahout-work-wiki/output
18/03/10 07:51:05 INFO ZlibFactory: Successfully loaded & initialized native-zlib library
18/03/10 07:51:05 INFO CodecPool: Got brand-new compressor [.deflate]
18/03/10 07:51:05 INFO CodecPool: Got brand-new decompressor [.deflate]
18/03/10 07:51:10 INFO TestNaiveBayesDriver: Complementary Results:
=======================================================
Summary
-------------------------------------------------------
Correctly Classified Instances          :       2068       85.3135%
Incorrectly Classified Instances        :        356       14.6865%
Total Classified Instances              :       2424

=======================================================
Confusion Matrix
-------------------------------------------------------
a       b       c       d       e       f       g       h       i       j       <--Classified as
736     5       38      10      7       7       5       17      6       11      |  842        a     = australia
2       106     3       1       1       0       0       1       1       0       |  115        b     = austria
0       0       9       0       1       0       0       0       0       0       |  10         c     = bahamas
5       4       19      474     2       7       2       10      8       7       |  538        d     = canada
0       1       0       0       28      2       0       4       0       0       |  35         e     = colombia
0       1       0       1       0       47      0       0       0       1       |  50         f     = cuba
0       0       1       0       1       1       85      1       2       0       |  91         g     = pakistan
0       4       0       2       1       3       0       11      0       0       |  21         h     = panama
6       24      65      7       2       11      13      7       504     8       |  647        i     = united kingdom
1       1       0       0       0       5       0       0       0       68      |  75         j     = vietnam

=======================================================
Statistics
-------------------------------------------------------
Kappa                                       0.7881
Accuracy                                   85.3135%
Reliability                                76.9128%
Reliability (standard deviation)            0.2814
Weighted precision                          0.9218
Weighted recall                             0.8531
Weighted F1 score                           0.8792

18/03/10 07:51:10 INFO MahoutDriver: Program took 12621 ms (Minutes: 0.21036666666666667)
[koga@n0120 bin]$
```

図 8-1　Apache Mahout による Wikipedia ドキュメントの分類結果

図 8-1 の分類結果を見ると、試験データ 2424 個に対し、正しく分類されたデータが 2068 個、一方、間違って分類されたデータが 356 個あり、正解率が 85.3135 %であることがわかります。一般に、分類では、結果の出力に混同行列（Confusion Matrix）が利用されます。混同行列は、分離した結果を行列（表）としてまとめたものです。分析対象で、正しく分離されたものと誤って分類されたものをまとめるために利用される表です。

図 8-1 では、混同行列が表示されています。行がカテゴリを表し、列に分類した結果のカテゴリが出力されています。左上から右下への対角線上の値が正解の数を表しており、それ以外に位置している値は、不正解の数を表します。

以上で、Apache Hadoop 3 クラスター上で Apache Mahout による Wikipedia ドキュメントの分類ができました。

8-2-2 mahout コマンドを理解する

Complement Naive Bays によるドキュメントの分類を実行する clasify-wikipedia.sh スクリプトは、内部で、Apache Mahout が提供する mahout コマンドを使って機械学習を行っています。clasify-wikipedia.sh スクリプトを読むと、mahout コマンドの実行方法がわかりますが、オプションの意味までは記述されていません。そこで、以下では、処理の流れと mahout コマンドの意味を理解するために、clasify-wikipedia.sh スクリプトで使わずに Wikipedia ドキュメントを分類する手順を紹介します。

■ データの準備

まず、準備として、クライアントマシン上の作業用ディレクトリに関する環境変数「WORK_DIR」と作業用ディレクトリ「$WORK_DIR/wikixml」を作成します。

```
$ export WORK_DIR=/tmp/mahout-work-wiki
$ rm -rf $WORK_DIR
$ mkdir -p $WORK_DIR/wikixml
$ mkdir -p $WORK_DIR/wiki
```

次に、圧縮済みの Wikipedia のドキュメントを入手し、展開します。約 1.3GB の XML 形式の Wikpedia ドキュメントに展開されます。

```
$ curl \
https://dumps.wikimedia.org/enwiki/latest/\
enwiki-latest-pages-articles10.xml-p2336425p3046511.bz2 \
-o ${WORK_DIR}/wikixml/enwiki-latest-pages-articles.xml.bz2

$ cd ${WORK_DIR}/wikixml && bunzip2 enwiki-latest-pages-articles.xml.bz2
$ cd ${WORK_DIR}
$ ls -lh wikixml/
total 1.3G
-rw-r--r-- 1 koga hadoop 1.3G Apr 18 15:09 enwiki-latest-pages-articles.xml
```

分類項目となる 10 か国の国名が書かれたファイル「country10.txt」を作業用ディレクトリにコピーします。

```
$ cp $MAHOUT_HOME/examples/bin/resources/country10.txt ${WORK_DIR}/country.txt
$ cat ${WORK_DIR}/country.txt
Australia
Austria
Bahamas
...
```

クライアントマシンの/tmp/mahout-work-wikiディレクトリ以下に保管したWikipediaのデータをHDFSの/tmp/mahout-work-wikiディレクトリにコピーします。

```
$ printenv WORK_DIR
/tmp/mahout-work-wiki

$ hdfs dfs -rm -r ${WORK_DIR}
$ hdfs dfs -mkdir -p ${WORK_DIR}
$ hdfs dfs -put ${WORK_DIR}/wikixml ${WORK_DIR}/wikixml
```

以上で、機械学習を行う準備が整いました。

■ データの変換

実際にmahoutコマンドを使って機械学習を行います。まず、mahoutコマンドに「seqwiki」を付与し、Wikipediaのドキュメント（XMLファイル）をMahoutが取り扱うシーケンスファイルに変換します。変換作業にかかる時間も、Hadoopクラスターの規模によって大きく異なります。

```
$ printenv MAHOUT_HOME
/home/koga/apache-mahout-distribution-0.13.0

$ $MAHOUT_HOME/bin/mahout seqwiki \  ←1
-c ${WORK_DIR}/country.txt \  ←2
-i ${WORK_DIR}/wikixml/enwiki-latest-pages-articles.xml \  ←3
-o ${WORK_DIR}/wikipediainput  ←4
```

コマンドの説明：

1. Wikipedia の XML ダンプファイルを Mahout で取り扱うシーケンスファイルに変換
2. カテゴリ（分類項目）が記されたファイルを指定
3. 入力ファイルとなる Wikipedia のドキュメント（XML 形式）
4. シーケンスファイルの保存先ディレクトリ（HDFS 上）

さらに、シーケンスファイルを分類器の入力ファイルとして利用できるように、ベクトル形式に変換します。

```
$ $MAHOUT_HOME/bin/mahout seq2sparse \  ←1
-i ${WORK_DIR}/wikipediainput \  ←2
-o ${WORK_DIR}/wikipediaVecs \  ←3
-wt tfidf \  ←4
-lnorm -nv \  ←5
-ow \  ←6
-ng 2  ←7
```

コマンドの説明：

1. シーケンスファイルをベクトル形式に変換
2. 入力ファイルとなるシーケンスファイルの保管先ディレクトリ（HDFS 上）
3. ベクトル形式のデータの保存先ディレクトリ（HDFS 上）
4. 各単語に付与する重み付けの方法（TF-IDF（Term Frequency, Inverse Document Frequency)）を指定 †
5. 出力されるベクトルを log で正規化する
6. 出力ディレクトリにベクトル形式のデータを上書き保存
7. N グラムの大きさ (2 を指定すると連続した単語を 2 文字で分割（バイグラム)、3 を指定すると 3 文字で分割)

† TF-IDF の TF は、ドキュメント内に現れる単語の出現頻度であり、頻出の単語ほど重要度が高いと評価します。IDF は、その単語がいくつのドキュメント内に出現するかを表し、多くのドキュメントに出現する単語は、重要度が低いと評価します。

■ データの分割

ベクトル形式のデータを訓練用のデータとテスト用のデータに分割します。

```
$ $MAHOUT_HOME/bin/mahout split \  ←1
-i ${WORK_DIR}/wikipediaVecs/tfidf-vectors/ \  ←2
--trainingOutput ${WORK_DIR}/training \  ←3
--testOutput ${WORK_DIR}/testing \  ←4
-rp 20 \  ←5
-ow \  ←6
-seq \  ←7
-xm sequential  ←8
```

コマンドの説明：

1. ベクトル形式のデータを訓練用データとテスト用データに分割
2. 入力となるベクトル形式のデータが保管されているディレクトリ（HDFS 上）
3. 訓練用データの保存先ディレクトリ（HDFS 上）
4. テスト用データの保存先ディレクトリ（HDFS 上）
5. MapReduce による分散処理の際にテスト用データとしてランダムに選択されるアイテムの割合（パーセント）
6. 保存先のディレクトリを上書き保存

7 入力データがシーケンスファイルの場合はセットする
8 実行方法の選択（sequential か mapreduce を選択）

■ モデルの作成

訓練用データを使ってモデルを作成します。

```
$ $MAHOUT_HOME/bin/mahout trainnb \  ←1
-i ${WORK_DIR}/training \  ←2
-o ${WORK_DIR}/model \  ←3
-li ${WORK_DIR}/labelindex \  ←4
-ow \  ←5
-c  ←6
```

コマンドの説明：
1 ナイーブベイズによるモデルの作成
2 入力となる訓連用データのディレクトリ（HDFS 上）
3 生成されるモデルの保存先ディレクトリ（HDFS 上）
4 ラベルインデックスを保管するディレクトリ（HDFS 上）
5 保存先のディレクトリを上書き保存
6 CBayes を利用

■ モデルのテスト

訓練用データにより生成されたモデルにテストデータをロードし、ドキュメントが分類できるかを確認します。

```
$ $MAHOUT_HOME/bin/mahout testnb \  ←1
-i ${WORK_DIR}/testing \  ←2
-m ${WORK_DIR}/model \  ←3
-l ${WORK_DIR}/labelindex \  ←4
-ow \  ←5
-o ${WORK_DIR}/output \  ←6
-c  ←7
```

コマンドの説明：
1 モデルのテスト
2 入力となるテスト用データのディレクトリ（HDFS 上）
3 テストに使用するモデルが格納されているディレクトリ（HDFS 上）
4 ラベルインデックスのディレクトリ（HDFS 上）

```
5  保存先のディレクトリを上書き保存
6  出力先のディレクトリ（HDFS 上）
7  CBayes を利用
```

これにより、混同行列の結果が得られます。

8-2-3　おすすめ映画のタイトルを表示する

次に、Apache Mahout によるレコメンデーションを実行してみます。今回は、おすすめ映画のタイトルを表示します。

■ データの入手

入手するデータは、zip ファイルで圧縮されているため、クライアントノードに事前に zip コマンドをインストールします[*1]。

```
# hostname
n0120.jpn.linux.hpe.com

# yum makecache fast && yum install -y unzip
```

一般ユーザー koga になり、zip 圧縮された映画のデータを入手し、解凍します。

```
# su - koga
$ whoami
koga

$ wget http://files.grouplens.org/datasets/movielens/ml-100k.zip
$ unzip ml-100k.zip
```

データの中身を確認します。

```
$ cd ./ml-100k/
$ cat u.data
196     242     3       881250949
186     302     3       891717742
22      377     1       878887116
```

＊1　データの出典は、米国ミネソタ大学のソーシャルコンピューティング研究「Movielens」から引用

```
...
```

u.data ファイルの中身は、左から会員 ID、映画 ID、評価、タイムスタンプを表しています。さらに、u.item ファイルの中身を確認します。

```
$ cat u.item
1|Toy Story (1995)|01-Jan-1995||http://us.imdb.com/M/title-exact?Toy%20Story%
20(1995)|0|0...
2|GoldenEye (1995)|01-Jan-1995||http://us.imdb.com/M/title-exact?GoldenEye%20
(1995)|0|1|0...
3|Four Rooms (1995)|01-Jan-1995||http://us.imdb.com/M/title-exact?Four%20Room
s%20(1995)|0...
...
```

u.item ファイルの中身は、左から映画 ID、タイトル、Internet Movie Database（インターネット・ムービー・データベース、以下、IMDb）の URL、映画のカテゴリなどを表しています。

■ Hadoop クラスターへのデータの格納

u.data ファイルを HDFS の/user/koga ディレクトリに格納します。

```
$ hdfs dfs -put u.data /user/koga/
```

■ レコメンデーションの実行

レコメンデーションを実行する前に、HDFS の/user/koga/temp ディレクトリがあれば、削除しておきます。

```
$ hdfs dfs -rm -R /user/koga/temp
```

Apache Mahout を使ってレコメンデーションを実行します。

```
$ printenv MAHOUT_HOME
/home/koga/apache-mahout-distribution-0.13.0

$ which mahout
~/apache-mahout-distribution-0.13.0/bin/mahout

$ mahout recommenditembased \
```

```
--similarityClassname SIMILARITY_COOCCURRENCE \
--input               /user/koga/u.data       \
--output              /user/koga/output01     \
--tempDir             /user/koga/temp
```

Apache Mahout が Hadoop の YARN の仕組みを利用して、レコメンデーションの処理を実行します。処理時間は、Hadoop クラスターのノード数やノードの CPU コア数で左右されます。

■ 結果のマージ

HDFS 上の/user/koga/output01 ディレクトリに生成された結果のファイル群をマージし、ローカルのクライアントマシン上に単一のファイル「output01.txt」として出力します。

```
$ pwd
/home/koga/ml-100k

$ hdfs dfs -getmerge /user/koga/output01 ./output01.txt
```

■ 整形表示用スクリプト recom.py を作成

次に、結果を整形して表示する recom.py を作成します。

```
vi recom.py
# -*- coding: utf-8 -*-
import sys
u, d, m, r = sys.argv[1:]
df = open(d); mid = []
for line in df:
  if line.split("\t")[0] == u:
    mid.append((line.split("\t")[1],line.split("\t")[2]))
df.close()
mf = open(m); mov = {}
for line in mf:
  mov[line.split("|")[0]] = line.split("|")[1:]
mf.close()
rf = open(r); rcmd = []
for line in rf:
  if line.split("\t")[0] == u:
    id_sc = line.split("\t")[1].strip("[]\n").split(",")
    rcmd = [ id_sc.split(":") for id_sc in id_sc ]
```

```
    break
rf.close()
print "====== 映画の評価 ======"
for id, rt in mid:
  R = ''
  for i in range(int(float(rt))):
      R=R+'☆'
  print "%s : 評価: %s" % (mov[id][0],R)
print;print "****** おすすめ映画 ******"
for id, sc in rcmd:
  S = ''
  for i in range(int(float(sc))):
     S=S+'☆'
  print "%s : おすすめ度: %s" % (mov[id][0],S)
```

■ ユーザーごとのおすすめ映画タイトルを表示

整形表示用の recom.py スクリプトを使って、ユーザーごとのおすすめ映画タイトルを表示します。recom.py は、第 1 引数に、会員 ID を指定します。今回は、ID が 16 番の会員を対象に推奨映画タイトルを表示してみます。

```
$ pwd
/home/koga/ml-100k

$ python ./recom.py 16 ./u.data ./u.item ./output01.txt
====== 映画の評価 ======
...
Desperado (1995) : 評価: ☆☆
Kingpin (1996) : 評価: ☆☆☆☆☆
While You Were Sleeping (1995) : 評価: ☆☆☆☆
Clueless (1995) : 評価: ☆☆☆
Strange Days (1995) : 評価: ☆☆☆☆☆

****** おすすめ映画 ******
Star Trek III: The Search for Spock (1984) : おすすめ度: ☆☆☆☆☆
Citizen Kane (1941) : おすすめ度: ☆☆☆☆☆
Fried Green Tomatoes (1991) : おすすめ度: ☆☆☆☆☆
Gandhi (1982) : おすすめ度: ☆☆☆☆☆
Batman Returns (1992) : おすすめ度: ☆☆☆☆☆
...
```

会員 ID が 16 番の人に対するおすすめの映画タイトルが出力できました。

以上で、Apache Hadoop 3 クラスター上で、Apache Mahout を使ったレコメンデーションが実行できました。

8-3 まとめ

本章では、Apache Mahout による Wikipedia ドキュメントの分類と映画のレコメンデーション例を紹介しました。Apache Mahout の tar アーカイブの `examples/bin` ディレクトリには、本書で紹介した以外の機械学習のサンプルスクリプトが含まれています。

`mahout` コマンドは、機械学習におけるさまざまな機能を提供しますが、すべてを頭に入れて覚える必要はありません。サンプルスクリプトには、Mahout の基本的な使い方が記されていますので、参考になりますが、本書に記載したように、`mahout` コマンドだけでなく、結果の整形表示用のプログラムなどを作成しなければならない場合も少なくありません。まずは、Mahout の tar アーカイブに含まれている `examples/bin` ディレクトリ以下のサンプルスクリプトを少しずつ試し、本書で紹介した基本的な手順を試して、`mahout` コマンドを使った、一連の機械学習処理の流れをつかんでみてください。さらに、余力があれば、Hadoop クラスターのワーカーノードを増設し、Mahout による機械学習の処理がスケールすることも確認してみてください。

索　引

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

P

R

S

T

U

W

Y

Z

あ

い

う

え

か

き

く

こ

さ

し

す

せ

そ

た

ち

て

と

な

ね

は

ひ

ふ

へ

ほ

ま

め

ゆ

ら

り

れ

ろ

わ

■ 商品に関する問い合わせ先

インプレスブックスのお問い合わせフォームより入力してください。

https://book.impress.co.jp/info/

上記フォームがご利用頂けない場合のメールでの問い合わせ先

info@impress.co.jp

●本書の内容に関するご質問は、お問い合わせフォーム、メールまたは封書にて書名・ISBN・お名前・電話番号と該当するページや具体的な質問内容、お使いの動作環境などを明記のうえ、お問い合わせください。
●電話やFAX等でのご質問には対応しておりません。なお、本書の範囲を超える質問に関しましてはお答えできませんのでご了承ください。
●インプレスブックス（https://book.impress.co.jp/）では、本書を含めインプレスの出版物に関するサポート情報などを提供しておりますのでそちらもご覧ください。
●該当書籍の奥付に記載されている初版発行日から3年が経過した場合、もしくは該当書籍で紹介している製品やサービスについて提供会社によるサポートが終了した場合は、ご質問にお答えしかねる場合があります。

■ 落丁・乱丁本などの問い合わせ先

TEL 03-6837-5016 FAX 03-6837-5023
service@impress.co.jp
（受付時間／10:00-12:00、13:00-17:30 土日、祝祭日を除く）
●古書店で購入されたものについてはお取り替えできません。

■書店／販売店の窓口

株式会社インプレス 受注センター
TEL 048-449-8040
FAX 048-449-8041
株式会社インプレス 出版営業部
TEL 03-6837-4635

ビッグデータブンセキキバンノコウチクジレイシュウ
ビッグデータ分析基盤の構築事例集
ハドゥープクラスターコウチクジッセンガイド
Hadoopクラスター構築実践ガイド

2018年5月21日 初版発行

著 者 古賀（こが） 政純（まさずみ）

発行人 土田米一

編集人 高橋隆志

発行所 株式会社インプレス
〒101-0051 東京都千代田区神田神保町一丁目105番地
ホームページ https://book.impress.co.jp/

印刷所 大日本印刷株式会社

ISBN978-4-295-00369-4 C3055

Printed in Japan