Hadoop 調査報告書

エヌ・ティ・ティ レゾナント株式会社
株式会社 Preferred Infrastructure

平成 20 年 8 月 25 日

- **免責条項**

  本報告書はエヌ・ティ・ティ レゾナント株式会社 (以下「NTT レゾナント」) と株式会社 Preferred Infrastructure(以下「Preferred Infrastructure」) が作成したものですが、報告書の内容及び情報の正確性、完全性、有用性について、NTT レゾナント及び Preferred Infrastructure は保証を行なっておらず、また、いかなる責任を持つものでもありません。

  本報告書の著作権は NTT レゾナントに帰属します。

  本報告書の「プリントアウト」「コピー」「無料配布」は可能ですが、変更、改変、加工、切除、部分利用、要約、翻訳、変形、脚色、翻案などは禁止します。

  以上の点をご了承の上、ご利用ください。

- **執筆者**

  Preferred Infrastructure 太田一樹

  NTT レゾナント 金田有二

- **本報告書に関する問い合わせ先**

  Preferred Infrastructure: E-mail: info@preferred.jp

  NTT レゾナント 技術マーケティング部 E-mail: pr@nttr.co.jp

更新履歴

日付	修正箇所	内容
2008 年 8 月 25 日		調査報告書公開

目次

図目次

表目次

第 1 章

はじめに

1.1 目的

本報告書の目的は、オープンソースソフトウェア Hadoop[4] が実際に運用可能なレベルに達しているかを調査し報告することである。

まず 2 章で、Hadoop についての概要説明を行う。

3, 4 章で、それぞれ Google の基盤システムである Google File System[10] と MapReduce[9] の機能を列挙し、それらが Hadoop でどのように実装されているのかを洗い出す。

5 章で、Hadoop をコードレベルで解析し、どのファイルでどのような操作が行われているのかを調査する。

6 章で、実験により Hadoop の性能を確認する。

7 章で、本報告書についてまとめる。

なお、本報告書では Hadoop バージョン 0.16.4 のソースコードを解析した結果を元に記述している。また実験にも同バージョンを用いた。

第 2 章

Hadoop の概要

Hadoop は主に Yahoo! Inc. の Doug Cutting 氏によって開発が進められているオープンソースソフトウェアである。元々は検索エンジン Lucene[8] を分散化させるために作られたプロジェクトであったが、Lucene プロジェクトからは切り離され現在は単独のプロジェクトとして開発が進められている。Hadoop は Google の基盤ソフトウェアである Google File System(以下 GFS) と、MapReduce のオープンソース実装となっている。

Hadoop は HDFS（Hadoop Distributed File System)、Hadoop MapReduce Framework から構成されている。Google の基盤技術に対応させると、前者は GFS、後者は MapReduce に対応する（図 2.1 参照)。また分散データベース BigTable についても、hBase と呼ばれるオープンソース実装が最近登場した。

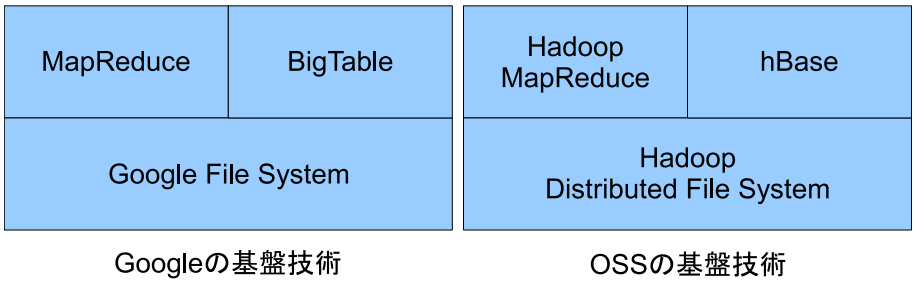

図 2.1　Google, OSS 基盤技術の対応関係

Hadoop はすべて Java で記述されており、MapReduce 処理を書く場合も基本的には Java でプログラムを書くことが想定されている。ただし Hadoop Streaming[5] という拡張パッケージを用いると、C/C++・Ruby・Python など任意の言語と標準入出力を用いて MapReduce 処理を書くことも出来る。

第 3 章

GFS と HDFS の機能比較

この章では分散ファイルシステムである GFS と HDFS の機能を比較する。GFS の機能を列挙し、各機能が Hadoop でどのように実装されているかについて述べる。

3.1 GFS の概要

GFS の概要について、論文 [10] の記述に基づいて説明する。

3.1.1 特徴

GFS は多数の PC 上に構築される分散ファイルシステムであり次の特徴を備える。

- 大容量
  数百 TB クラスの大規模なディスクを提供できる。
- スケーラビリティ
  PC を追加するだけでシステムを停止せずにディスク容量・読み込み性能・書き込み性能を拡大できる。
- 耐障害性
  故障の発生を前提としたソフトウェア設計により、障害を自動的に監視・検出・修復できる。

3.1.2 アーキテクチャ

GFS では、ファイルを複数のチャンクに分割して格納する。チャンクサイズのデフォルト値は 64MB である。各チャンクは PC 上のローカルディスクに格納され、これらのチャンクを仮想的に統合することで巨大なファイルを実現する。

GFS では、単一のマスターと多数のチャンク・サーバーを用いて、ファイル及びチャンクを管理する。マスターは次の 3 つの情報を管理する。

- ファイルのディレクトリ構造
- ファイルを構成するチャンクの情報
- チャンクの格納先のチャンクサーバ

チャンク・サーバーはローカル・ディスク上に格納された、個々のチャンクを管理する。

GFS では、チャンクを格納する際に、チャンクの内容を複数のマシンにコピーして格納する。そのため、あるマシンに格納したチャンクがハード障害などにより失われたとしても、別のサーバに複製があるため、チャンクそのものが失われることはない。

GFS ではクライアントとマスター間の通信量が少なくなるように設計されている。例えば、GFS のクライアントがファイルを読み込む際には、マスターから取得するのはメタデータ（ファイルを構成するチャンクと各チャンクを格納するチャンク・サーバーの情報）のみであり、データそのものはチャンク・サーバから直接取得する。また、チャンク・サーバーからチャンクを取得する際には、複数のチャンクの複製の中から、最寄のマシンに格納されたチャンクを取得することにより、ラック間・ラック内での通信量を削減している。

3.1.3 HDFS との関係

HDFS のアーキテクチャは GFS を踏襲している。なお、HDFS では、マスター、チャンクサーバ、チャンクを、それぞれ、NameNode、DataNode、ブロックと呼んでいる。また、HDFS のみが持つ機能も調査したが、特に目立った機能はなかった。

3.2 機能一覧

表 3.1 に GFS の持つ機能、および、各機能の Hadoop での実装の有無を示す。以下、各機能の HDFS での実装方式および実装箇所のソースを調査した結果を示していく。

表 3.1: Google File System の持つ機能一覧と Hadoop 実装の有無

機能分類	機能概要	Hadoop 実装の有無
基本機能	ディレクトリの作成	○
基本機能	ディレクトリの消去	○
基本機能	ファイルの作成	○
基本機能	ファイルの削除	○
基本機能	削除ファイルのバックアップ・メタデータの自動削除	○
基本機能	ファイルの読み込み	○
基本機能	ファイルの書き込み	○
基本機能	ファイルのランダム読み込み	○
基本機能	ファイルのランダム書き込み	×
基本機能	ファイル・ディレクトリのリネーム	○
基本機能	ファイルリストの取得	○
基本機能	ファイル属性	○
基本機能	ディレクトリ属性	○
基本機能	ファイルのアトミックな追記	×

機能分類	機能概要	Hadoop 実装の有無
基本機能	スナップショット	○
基本機能	マスターによるチャンク・サーバの監視	○
管理	チャンク・サーバの動的な追加	○
管理	チャンクのリバランシング	○
管理	アクセスコントロール	○
管理	ロギング	○
性能	最寄サーバーからの読み込み	○
耐障害性	チャンクのバージョン管理	○
耐障害性	チャンクのチェックサム機能	○
耐障害性	バックグラウンドでの自動チェックサム検査	×
耐障害性	チャンク・サーバ障害時のチャンク自動レプリケーション	○
耐障害性	レプリケーション時の複数サーバーへの書き込み	○
耐障害性	接続・読み・書きエラー時のリトライ	○
耐障害性	マスターによるオペレーションログの保持	○
耐障害性	オペレーションログのスレーブへの保持	△
耐障害性	オペレーションログからのマスターの復旧	○
耐障害性	シャドウマスター (Read-Only なマスター)	×

3.3 基本機能

3.3.1 ディレクトリの作成

機能概要

ディレクトリの作成を行う。

Hadoop の実装

クライアントは NameNode に対して要求を出し、NameNode はそれをうけて保持しているディレクトリツリーの情報を更新する。

ソース

DFSClient::mkdirs

3.3.2 ディレクトリの消去

機能概要

ディレクトリの削除を行う。

Hadoop の実装

クライアントは NameNode に対して要求を出し、NameNode は要求されたディレクトリを削除する。

ソース

DFSClient::delete

3.3.3 ファイルの作成

機能概要

ファイルの作成を行う。

Hadoop の実装

クライアントは NameNode に対して要求を出し、NameNode は要求された情報に基づきファイルを作成する。

ソース

DFSClient::create

3.3.4 ファイルの削除

機能概要

ファイルの削除を行う。

Hadoop の実装

クライアントは NameNode に対して要求を出し、NameNode は要求されたパスのファイルを削除する。次の項目でも述べるが、削除されたファイルはすぐには削除されない。

ソース

DFSClient::delete

3.3.5 削除ファイルのバックアップ・メタデータの自動削除

機能概要

削除したファイルはすぐには削除せず、遅延して削除する (5.6.6)。削除したファイルは一定時間内には復旧できるようになっている。

Hadoop の実装

ユーザーによって delete されたファイルはすぐに削除はされず、一度/trash ディレクトリに移動させられる。そして一定時間が経つと実際の削除が行われる。/trash ディレクトリに有る間は、復旧することが可能である。

ソース

NameNode.emptier

3.3.6 ファイルの読み込み

機能概要

ファイルの読み込みを行う。

Hadoop の実装

DFSInputStream でブロック単位で読み込む (5.7.5)。最初に NameNode にパスを送信し、そのファイルを構成するブロックを保持している DataNode のリストを得る。そのリストを利用し、クライアントは 1 つずつブロックを取得する。

ソース

DFSInputStream::read

3.3.7 ファイルの書き込み

機能概要

ファイルの書き込みを行う。

Hadoop の実装

DFSOutputStream でブロック単位で書き込む (5.7.5)。NameNode にパスを送信すると、ブロックを書き込むべきサーバーのリストが返ってくるので、その情報を元にブロックを DataNode に送信する。

ソース

DFSOutputStream::writeChunk

3.3.8 ファイルのランダム読み込み

機能概要

ファイルのランダムな位置の読み込みを行う。

Hadoop の実装

DFSInputStream でブロック単位で読み込む (5.7.5)。NameNode にパスとオフセットを渡すと該当ブロックを保持している DataNode のリストが返ってくるので、NameNode に問い合わせる。

ソース

DFSInputStream::read

3.3.9 ファイルのランダム書き込み

機能概要

ファイルのランダムな位置への書き込みを行う。

Hadoop の実装

未実装

ソース

該当無し

3.3.10 ファイル・ディレクトリのリネーム

機能概要

ファイル・ディレクトリの名前を変える。

Hadoop の実装

DFSClient が NameNode に要求を出す。NameNode は保持しているディレクトリツリーに変更を加え、名前を変更する。

ソース

DFSClient::rename

3.3.11 ファイルリストの取得

機能概要

パスを指定し、ディレクトリの中身のリストを取得する。

Hadoop の実装

DFSClient が NameNode に要求を出すと、NameNode は保持しているディレクトリツリーを参照し、該当するディレクトリやファイルの情報を返す。

ソース

DFSClient::listPaths

3.3.12 ファイル属性

機能概要

通常のファイルシステムの様に、ファイルに各種属性（更新時刻、サイズ、オーナー、パーミッションなど）が付けられる。

Hadoop の実装

更新時間、パーミッション、ユーザー、グループ、サイズ、レプリケーション数が実装されている。ユーザーについてはクライアントの `whoami` コマンド、グループについては `bash -c groups` コマンドで取得した値が用いられる (参考文献参照)。全ての属性はコマンドラインから変更可能である (`bin/hadoop dfs`)。

ソース

DFSClient::getFileInfo

参考文献

`http://hadoop.apache.org/core/docs/current/hdfs_permissions_guide.html`

3.3.13 ディレクトリ属性

機能概要

通常のファイルシステムの様に、ディレクトリに各種属性が付けられる。

Hadoop の実装

更新時間、パーミッション、ユーザー、グループが実装されている。ユーザーについてはクライアントの `whoami` コマンド、グループについては `bash -c groups` コマンドで取得した値が用いられる (参考文献参照)。全ての属性はコマンドラインから変更可能である (`bin/hadoop dfs`)。

ソース

DFSClient::getFileInfo

参考文献

`http://hadoop.apache.org/core/docs/current/hdfs_permissions_guide.html`

3.3.14 ファイルのアトミックな追記

機能概要

既存ファイルに対して、追記を行うことが出来る。この操作はアトミックに行われ、レプリケーションによりチャンクが複数のチャンクサーバに保存されている場合でも一貫性が保たれる。また、複数クライアントから追記した場合にも一貫性が保たれる。

Hadoop の実装

未実装

ソース

HADOOP-1700 で現在実装が進められている。

参考文献

`http://issues.apache.org/jira/browse/HADOOP-1700`

3.3.15 スナップショット

機能概要

ファイルシステムのスナップショット (ディレクトリツリーのコピー) を取る。

Hadoop の実装

FSImage クラスが NameNode の保持している情報のスナップショットを取る機能を担当している。

ソース

FSImage

3.3.16 マスターによるチャンク・サーバの監視

機能概要

マスターはチャンク・サーバの死活を定期的に監視し、チャンク・サーバの異常終了を検知する事が出来る。

Hadoop の実装

HeartBeat メッセージのやりとりにより実現している。

DataNode はある一定の間隔で NameNode に HeartBeat メッセージを送る。NameNode は、DataNode からある一定以上の秒数 HeartBeat メッセージを受け取らなかった場合は DataNode が異常終了したと判断する。HeartBeat メッセージの送信間隔は設定ファイルの”dfs.heartbeat.interval”で設定する。デフォルトは 3 秒となっている。

ソース

DataNode::offerService → NameNode::sendHeartbeat

3.4 管理

3.4.1 チャンク・サーバの動的な追加

機能概要

クラスタ全体をとめずに、チャンク・サーバを追加しディスク領域を増加させる事が出来る。

Hadoop の実装

DataNode を起動すると NameNode に HeartBeat メッセージを送信する。これにより NameNode は DataNode の追加を知ることができる。ただし、今のところ、新しく追加した DataNode に既存ブロックが自動的にコピーされることはないため、ディスク使用量が偏る等の問題がある。

ソース

DataNode::offerService → NameNode::sendHeartbeat

3.4.2 チャンクのリバランシング

機能概要

ディスク使用量が偏らないようにチャンクの再配置を行う。ディスク使用量が偏っている場合は、ディスク使用率の多いディスクから少ないディスクへとチャンクを移動させる。

Hadoop の実装

自動的なリバランシングは実装されておらず、明示的に管理者がリバランス操作を行う必要が有る。

`./bin/hadoop/start-balancer.sh` でリバランシング操作が行える。

空き容量のカウントは DF::run で行っている (5.6.9)。実際には `df` プログラムをシェル経由で起動しているだけである。DataNode は定期的に DF を使用して空き容量のチェックを行い、HeartBeat メッセージに載せて NameNode に報告している。

ソース

Balancer

参考文献

`https://issues.apache.org/jira/browse/HADOOP-1652`

3.4.3 アクセスコントロール

機能概要

ファイルやディレクトリに対して、読み込み・書き込みに対するアクセスコントロールを行える。

Hadoop の実装

パーミッションによる管理が可能である。

ソース

FSNameSystem::checkPermission → PermissionChecker::checkPermission

3.4.4 ロギング

機能概要

障害解析・デバッグ・性能測定のために、RPC・内部処理のログを残す

Hadoop の実装

Hadoop 全体でデフォルトのロガーとして Commons-Logging([2]) を採用しており、任意のタイミングでログ出力が可能である。ロガーの実装については log4j がデフォルトで使用される。デフォルトではブロックの追加などに関してログが吐かれており、バグトラッキングシステム上で障害解析に利用されている。

ソース

外部ソース (org.apache.commons.logging.*)

3.5 性能

3.5.1 最寄サーバーからの読み込み

機能概要

ファイルを読み出す際に一番最寄のサーバーから読み込むことで、ネットワーク転送量を削減する。

Hadoop の実装

ノード間の距離を定義し、一番近いノードから読み込みを行っている。近さを測る為に、ノードが位置するラックの階層を返すスクリプトをユーザが用意する必要がある (5.5.2, 5.5.3)。このスクリプトのパスは”dfs.network.scripts”で設定される。デフォルトでは全てのマシンが同じラックに位置しているとみなされる。

ソース

FSNameSystem::getBlockLocations → NetworkTopology::pseudoSortByDistance

参考資料

http://issues.apache.org/jira/secure/attachment/12345251/Rack_aware_HDFS_proposal.pdf

3.6 耐障害性

3.6.1 チャンクのバージョン管理

機能概要

チャンクのバージョンを管理することで、クラスタ内に古いチャンクが残っていても、それが古いと認識できる。

Hadoop の実装

バージョンではなく、id によって管理されている。NameNode では現在保持している最新のブロックについて id を保持している。そのため、万が一古いブロックが残っている場合でもそれが古いと認識できるため、古い内容が読み出される事が無い。

ソース

Block

FSNamesystem::allocateBlock

3.6.2 チャンクのチェックサム機能

機能概要

チャンクを送受信する際にチェックサムを計算し、受けて側と送り手側で一致するのを確認することで、クライアントが不正なデータを受け取らないことを保障する。

Hadoop の実装

書き込み時には”io.bytes.per.checksum”で設定されるバイト毎にチェックサムが計算され (デフォルトは 512 バイト)、ディスクにデータと同じくチェックサムも書き込まれる。

読み出し時には送信側と受信側でチェックサムの整合性を確認し、不整合が起こった場合はそれを NameNode に報告する。

ソース

DFSOutputStream

DFSInputStream

3.6.3 バックグラウンドでの自動チェックサム検査

機能概要

ファイルを保存している間にファイルが壊れる可能性がある為、ある一定のタイミングで保持しているチャンクのチェックサムを取り、ファイルが壊れていない事を保障する。

Hadoop の実装

未実装

ソース

該当無し

3.6.4 チャンク・サーバ障害時のチャンク自動レプリケーション

機能概要

チャンクがクラスタ内で無くなってしまわないように、チャンク・サーバが落ちた場合はそのチャンク・サーバが保持していたチャンクのレプリカを別のチャンク・サーバに持たせる。

Hadoop の実装

ブロックのクラスタ内のレプリカ数がある一定値を下回った場合はこのロジックが働く。この一定値は設定ファイルで変更する事が出来る。ただし増加させると write 性能が下がるので注意が必要。どのサーバーにレプリカを置くかは参考資料、もしくは 5.7.10 参照。

ソース

FSNameSystem::pendingTransfers

ReplicationTargetChooser(5.7.10)

参考資料

http://issues.apache.org/jira/secure/attachment/12345251/Rack_aware_HDFS_proposal.pdf

3.6.5 レプリケーション時の複数サーバーへの書き込み

機能概要

レプリカ数が 2 以上の場合、書き込みが行われた際にはレプリカ数と同じだけのチャンク・サーバにチャンクを書き込む。

Hadoop の実装

GFS と同じく、数珠繋ぎでブロックの内容が複数の DataNode に送信される。つまり、クライアントから複数の DataNode にブロックの内容を送信するのではなく、DataNode から DataNode へとデータが送信される。

ソース

DFSOutputStream

3.6.6 接続・読み・書きエラー時のリトライ

機能概要

読み込み時や書き込み時にエラーが起きた場合は、適宜リトライを行う。

Hadoop の実装

RPC の接続時には、適宜リトライが行われる (5.4.2)。読み込み、書き込みに関してはそれぞれに応じたリトライロジックが実行される (5.7.5)

ソース

RPC

DFSOutputStream

DFSInputStream

3.6.7 マスターによるオペレーションログの保持

機能概要

マスターが落ちたときに備えて、マスターが処理した全てのメタデータ操作をディスク上にログとしてとっておく。万が一マスターが落ちて再起動した場合は、そのログに従って再度オペレーションを行うことで、落ちた直前の状態に復旧する事が出来る。

Hadoop の実装

FSEditLog クラス (5.7.9) がオペレーションログの管理を行っている。オペレーションログは dfs.name.dir 以下に出力される。

ソース

FSImage

FSEditLog

3.6.8 オペレーションログのスレーブへの保持

機能概要

マスターのオペレーションログをスレーブに保持させることで、オペレーションログが失われる可能性を低くする。

Hadoop の実装

スレーブではなく、SecondaryNameNode が NameNode から定期的にログを取得し、保持している形になっている (5.7.11)。NameNode と SecondaryNameNode が落ちない限り、最新のオペレーションログはクラスタ内に保持されている事になる。

SecondaryNameNode は定期的にログを取得するので、NameNode が落ちた場合、SecondaryNameNode が最後にログを取得した時刻以降のログが失われることがある。

ソース

SecondaryNameNode::run

3.6.9 オペレーションログからのマスターの復旧

機能概要

保存しておいたオペレーションログから、マスターの状態を復旧することができる。

Hadoop の実装

NameNode ではオペレーションログに書かれた動作を順次実行し、落ちた直前の状態に復旧するための機能が実装されている。具体的には FSImage クラスの loadFSImage 関数内でオペレーションログからの復旧を行っている (5.7.9)。

ソース

FSImage::loadFSImage → FSEditLog::loadFSEdits

3.6.10 シャドウマスター (Read-Only なマスター)

機能概要

マスターの負荷対策の為に、読み込み専用のマスターを用意する。書き込みのオペレーションに関してはマスターを通じて行われ、その状態が読み込み専用マスターにレプリケーションされる事で、ある程度の遅延を許しつつも負荷を軽減する事が出来る。

Hadoop の実装

未実装。SecondaryNameNode は現在オペレーションログの一時的な保存先としてしか働かず、メタデータ操作を行うことは出来ない。

ソース

該当無し

3.7 考察

HDFS は GFS の主要な機能を実装しており、分散ファイルシステムとしての基本機能を備えている。また、バージョン 0.16.4 ではファイルのアトミックな追記は未実装だが、バージョン 0.19 以降からサポートされる予定である。

チャンク・サーバは動的に追加可能であり、管理者はクラスタ構成を容易に変更できる。ただし、チャンクのリバランシングは手動で実行する為、管理者がディスクの使用量に注意する必要がある。

HDFS は、DataNode の障害に対応するための機能も備えており、ある程度の耐障害性を備えている。ただし、NameNode の障害に対する機能は不十分である。まず、NameNode が異常終了した場合、手動で NameNode を復旧させる必要がありその間サービスは完全に停止してしまう。また、オペレーションログのスレーブへの保持は一定間隔でのコピーなため、NameNode でディスク障害が発生した場合、オペレーションログの一部が失われファイルシステムの一貫性が一部失われてしまう可能性がある。

第 4 章

Google MapReduce と Hadoop MapReduce の比較

4.1 Google MapReduce の概要

Google MapReduce の概要について、論文 [9] の記述に基づいて説明する。

4.1.1 特徴

Google MapReduce は多数の PC 上で分散処理を実行する為のフレームワークであり、次の特徴を備える。

- MapReduce プログラミング・モデル

  MapReduce と呼ばれるプログラミング・モデルに基づくデータ処理を多数の PC 上で分散実行できる。このモデルでは、データ処理を次の 3 つの工程に分離する。

  - Map

    入力データの各レコードから中間データを生成する。中間データはキーと値の組とする。

  - Shuffle

    キーが同じ中間データをまとめて、キーと値のリストを生成する。

  - Reduce

    キーとそのキーに対応する値のリストから出力データを生成する。

  ユーザが記述可能なプログラムは限定されるが、分散処理特有の問題を MapReduce フレームワークが全て面倒を見てくれるため、分散処理を簡単に実行できる。

- 分散ファイルシステムとの密結合

  GFS の性質を利用することにより、高い処理性能を実現している。

- 耐障害性

  故障の発生を前提としたソフトウェア設計により、障害を自動的に監視・検出・修復できる。

4.1.2 アーキテクチャ

Google MapReduce では、まず、入力データを M 個に分割する。そして、分割されたデータの各レコードに対して Map 処理を実行する。この M 個の処理を Map タスクと呼び、Map タスクは複数の PC で独立に並列実行される。

Map タスク完了後、Map タスクが出力した中間データは R 個に分割される。そして、分割された中間データの各キーに対して Reduce 処理を実行する。この R 個の処理を Reduce タスクと呼び、Reduce タスクも複数の PC で独立に並列実行される。

なお、M 個の中間データを Reduce タスクに割り当てる際には、キーに基づいて各中間データを R 個に分割する。従って、中間データ全体は MR 個に分割される。Reduce タスクでは、Reduce 処理を実施する前に、M 個の分割された中間データを取得し、キーが同じ中間データの値をマージした後、各キーに対して Reduce 処理を実施する。

Google MapReduce では、単一のマスター、および、多数のワーカーを用いて、MapReduce フレームワークを実現する。マスターはクライアントからの MapReduce ジョブの受付、Map タスク、Reduce タスクのワーカーへの割り当てなどを担当する。また、ワーカーは個々の Map タスク、Reduce タスクの実行を担当する。

4.1.3 Hadoop MapReduce との関係

Hadoop MapReduce は Google MapReduce のアーキテクチャを踏襲している。なお、Hadoop MapReduce では、マスター、ワーカーを、それぞれ、JobTracker, TaskTracker と呼んでいる。

また、論文にはない Hadoop のみが持つと思われる機能として、ファイルの分散キャッシュ機能と HadoopStreaming が挙げられる。分散キャッシュ機能は、辞書データのように MapReduce プログラム全体で使用するようなファイルを簡単に使う為の機能である。HadoopStreaming[5] は、標準入出力を使用して、任意の言語で MapReduce プログラムを記述出来るようにするための機能である。

4.2 機能一覧

表 4.1 に Google MapReduce の持つ機能、および、各機能の Hadoop での実装の有無を示す。以下、各機能の Hadoop MapReduce での実装方式および実装箇所のソースを調査した結果を示していく。

表 4.1: Google MapReduce の持つ機能一覧と Hadoop 実装の有無

機能分類	機能概要	Hadoop 実装の有無
基本機能	MapReduce プログラムの実行	○
基本機能	マスターによるワーカーの監視	○
基本機能	Shuffle 関数の指定	○
基本機能	Map 処理・Reduce 処理でのタスク数の指定	○
基本機能	Map タスクの自動分割	○
基本機能	ワーカー台数の指定	△

機能分類	機能概要	Hadoop 実装の有無
基本機能	入力・出力フォーマットの指定	○
基本機能	自動カウンタ	○
基本機能	ユーザ定義カウンタ	○
管理	進捗状況のモニタリング	○
管理	デバッグ目的でのローカルでの逐次実行	○
管理	プログラムの強制終了	○
性能	Combine フェーズ	○
性能	ファイルのローカリティを考慮したタスク割り当て	○
性能	Map タスクが全て終了する前の Shuffle 開始	○
性能	出力結果の圧縮機能	○
性能	Map が出力した中間結果の圧縮機能	○
耐障害性	ワーカー障害により結果が失われたタスクの再実行	○
耐障害性	処理終盤で残タスクを複数ワーカーで同時実行（バックアップタスク）	○
耐障害性	バックアップタスク時に最速タスクの結果採用と残タスクの処理中断	○
耐障害性	マスターの状態の定期的なチェックポインティング	×
耐障害性	エラーレコードのスキップ	×

4.3 基本機能

4.3.1 MapReduce プログラムの実行

機能概要

MapReduce 型のプログラムを記述する事が出来る。

Hadoop の実装

Hadoop では Java で MapReduce プログラムを記述する事が可能である。ただし、HadoopStreaming という拡張パッケージを用いることで、標準入出力を介した MapReduce プログラムを任意の言語で記述する事が出来る。

ソース

JobClient

JobTracker

TaskTracker

4.3.2 マスターによるワーカーの監視

機能概要

マスターはワーカーの死活を定期的に監視し、ワーカーの異常終了を検知する事が出来る。

Hadoop の実装

HeartBeat メッセージのやりとりにより実現している。

TaskTracker が定期的に JobTracker に対して HeartBeat メッセージを送る。HeartBeat の間隔はクラスタの大きさに応じて設定され、(ノード数/50 + 1) 秒となる (JobTracker::getNextHeartbeatInterval)。ただし最低秒数は 5 秒となっている。

ソース

TaskTracker::transmitHeartBeat → JobTracker::heartbeat

4.3.3 Shuffle 関数の指定

機能概要

中間データをキーに基づいて Reducer に割り振る際 (Shuffle)、通常は Hash 関数を Shuffle 関数として使用するが、ソートなど特殊な Shuffle 関数が必要となるケースが有るので、ユーザープログラムが Shuffle 関数を指定出来る。

Hadoop の実装

JobConf.setPartitioner クラスでプログラムから指定可能。HashPartitioner, KeyFieldBasedPartitioner などがある。HashPartitioner は単純にキーのハッシュを取って Reducer に振り分ける。KeyFieldBasedPartitioner はキーの一部分のみのハッシュを使用する。

ソース

HashPartitioner

KeyFieldBasedPartitioner

4.3.4 Map 処理・Reduce 処理でのタスク数の指定

機能概要

Map 処理、Reduce 処理のタスク数を指定する事でタスクの粒度を調節できる。

Hadoop の実装

MapReduce プログラムから指定可能。JobConf.setNumReduceTasks, JobConf.setNumMapTasks を用いる (5.8.1)。

ソース

JobConf::setNumMapTasks

JobConf::setNumReduceTasks

4.3.5 Map タスクの自動分割

機能概要

入力ファイルを自動で適当に分割し、Map タスクに割り当てる。

Hadoop の実装

InputSplit により入力が分割され、それぞれが Map タスクに割り当てられる (5.8.5)。InputSplit を継承したクラスを作ることによって入力をどのように分割するかを細かく制御できるようになっている。

ソース

InputSplit

4.3.6 ワーカー台数の指定

機能概要

MapReduce プログラムが使用できるマシン台数を指定することが出来る。

Hadoop の実装

マシン台数を指定することが出来ないが、タスクに対してプライオリティが指定できる。

ソース

JobConf::setJobPriority

4.3.7 入力・出力フォーマットの指定

機能概要

MapReduce プログラムの入力フォーマットと出力フォーマットを指定することができる。

Hadoop の実装

JobConf.setInputFormat, JobConf.setOutputFormat を用いて MapReduce プログラムから指定する (5.8.1)。頻繁に用いる入力フォーマットとしては生のテキストファイルを扱うための TextInputFormat、key-value が 1 行 1 行記述してある SequenceFileAsTextInputFormat などが有る。出力フォーマットとしては、同じく key-value が 1 行 1 行記述してある TextOutputFormat 等がある。

ソース

InputFormat

OutputFormat

TextInputFormat

TextOutputFormat

SequenceFile

4.3.8 自動カウンタ

機能概要

処理したデータの入出力バイト数、入出力ペア数などの統計量を計測するためのカウンタがある。

Hadoop の実装

Task クラスの Counter enumeration に種類が列挙してある。レコードというのは、1 つの key-value のペアを指す。

- MAP_INPUT_RECORDS, - Map の入力レコード数
- MAP_OUTPUT_RECORDS, - Map の出力レコード数
- MAP_INPUT_BYTES, - Map の入力バイト数
- MAP_OUTPUT_BYTES, - Map の出力バイト数
- COMBINE_INPUT_RECORDS, - Combine の入力レコード数
- COMBINE_OUTPUT_RECORDS, - Combine の出力レコード数
- REDUCE_INPUT_GROUPS, - Reduce の入力キー数
- REDUCE_INPUT_RECORDS, - Reduce の入力レコード数
- REDUCE_OUTPUT_RECORDS - Reduce の出力レコード数

ソース

Task

4.3.9 ユーザ定義カウンタ

機能概要

ユーザープログラムでカウンタを定義することができ、プログラム中からそのカウンタを増やすことが出来る。

Hadoop の実装

任意のカウンタを定義可能。Reporter::incrCount を使用してカウンタの数を増やすことが出来る。

ソース

Reporter

参考文献

http://www.jakobhoman.com/2007/11/quick-tour-of-hadoops-reporter-object.html

4.4 管理

4.4.1 進捗状況のモニタリング

機能概要

ユーザーがサブミットした MapReduce ジョブの進捗状況などをコンソールまたはブラウザ上に表示できる。

Hadoop の実装

HTTP 経由で途中結果をモニタリングすることが可能。JobTracker が HTTP サーバー機能を持っており、全てのジョブの進行状況や優先度を見ることができる (5.8.7)。また優先度の変更も行うことが出来る。

手元の CUI には進捗具合が % で表示される。

ソース

JobClient

StatusHttpServer

4.4.2 デバッグ目的でのローカルでの逐次実行

機能概要

デバッグ用にローカルで MapReduce プログラムの逐次実行が行える。

Hadoop の実装

完全にローカルで実行することが可能。JobConf にて”mapred.job.tracker”を local に設定すると、入力の分割・Map タスクの実行・Shuffle・Reduce タスクの実行が 1 プロセス内で行われる (5.8.1)。スレッドは使われず、完全に逐次実行される。

ソース

JobClient::init → LocalJobRunner

4.4.3 プログラムの強制終了

機能概要

実行中の MapReduce プログラムを強制的に終了する。

Hadoop の実装

`./bin/hadoop job -kill-task` で任意のタスクを終了可能。タスクのリストを表示するには、`./bin/hadoop job -list` とすれば良い。

ソース

JobClient

4.5 性能

4.5.1 Combine フェーズ

機能概要

Map フェーズを行った後、ローカルでキーをまとめ上げる Combine フェーズを上手く用いる事で、転送量の削減が実現できる。

Hadoop の実装

JobConf.setCombinerClass で設定可能 (5.8.1)。

ソース

JobConf

4.5.2 ファイルのローカリティを考慮したタスク割り当て

機能概要

ファイルが保存してあるノードで Map タスク・Reduce タスクを実行することでネットワーク転送を回避することができる。

Hadoop の実装

入力を分割する際、入力となるブロックを保持しているノードについて、近い順にソートしておく。近さに関しては 3.5.1 と同じロジックが使用される (参考文献参照)。この情報を利用して Task を割り当てることにより、処理対処のブロックの近くのノードで Map タスクを実行する。

Reducer を上手く配置して Shuffle 時のデータ転送量を削減するところまでは行っていない。

ソース

JobInProgress::createCache → InputFormat::getLocations → DistributedFileSystem::getFileBlockLocations

参考文献

`http://issues.apache.org/jira/secure/attachment/12345251/Rack_aware_HDFS_proposal.pdf`

4.5.3 Map タスクが全て終了する前の Shuffle 開始

機能概要

実行時間を減らすため、Map 処理が全て完了してから Shuffle を開始するのではなく、Map 処理が終わったものから順に Shuffle フェーズを開始しておく。

Hadoop の実装

現在は明示的に Shuffle フェーズが有るのでは無く、Reduce フェーズの一番最初に Map フェーズの出力をコピーしてくるフェーズが有る。Reduce タスクは Map タスクが全て実行するまでに TaskTracker に割り当てられるので、結果的に先に Shuffle を行うことが出来ている。

また中間ファイルを Fetch する際には 1 つの TaskTracker に負荷が集中しないように、適当にランダムにシャッフルした順番で中間ファイルを取りに行く工夫もなされている。

ソース

ReduceTask.ReduceCopier::fetchOutputs

4.5.4 出力結果の圧縮機能

機能概要

ディスク領域を食わないようにするため、または I/O 時間を少なくするため、MapReduce プログラムの出力を圧縮する。

Hadoop の実装

SequenceFile にて、key-value ペアを圧縮して保存する事が可能。圧縮コーデックとしては gzip と lzo が実装されている。プログラムから”mapred.output.compress”フラグを true にする事で出力が圧縮される (5.8.1)。

ソース

OutputFormatBase

4.5.5 Map が出力した中間結果の圧縮機能

機能概要

Map タスクが出力した結果を圧縮する機能。これにより Shuffle フェーズでやりとりされるデータ量が削減出来る。

Hadoop の実装

JobConf.setCompressMapOutput で指定出来る (5.8.1)。実際には MapTask によってこのフラグがチェックされ、True ならばシリアライズして中間ファイルを生成する際に圧縮が行われる。

ソース

JobConf

MapTask.MapOutputBuffer::MapOutputBuffer

4.6 耐障害性

4.6.1 ワーカー障害により結果が失われたタスクの再実行

機能概要

ジョブ実行中にワーカーに障害が起きた場合、いくつかのタスクが失敗してしまう。そのようなタスクを他のワーカーで再実行することにより、障害が起きてもジョブ自体は最後まで実行することが出来る。

Hadoop の実装

あるノードで失敗したタスクはリスケジュールされ、他のノードで再実行される。

ソース

JobInProgress::failedTask

4.6.2 処理終盤で残タスクを複数ワーカーで同時実行（バックアップタスク）

機能概要

ジョブの終盤では、少数のワーカーがタスクを実行している事になる。これらの中に、特に実行が遅いワーカーが有った場合全体の実行時間が長くなってしまう。これを回避する為に、終盤は複数のワーカーで同じタスクを実行することで実行時間を効率化する事が出来る。

Hadoop の実装

SpeculativeTask という名前で実装されている。この機能はプログラムで JobConf.setMapSpeculativeExecution, JobConf.setReduceSpeculativeExecution に true を渡すことで enable 出来る。TaskInProgress::hasSpeculativeTask にバックアップタスクが発動する際の条件が設定されている (5.8.5)。

ソース

TaskInProgress::hasSpeculativeTask

4.6.3 バックアップタスク時に最速タスクの結果採用と残タスクの処理中断

機能概要

バックアップタスクが走っていた場合、最初に終了したものを最終的な出力とし、他のタスクはすぐに中断させる。

Hadoop の実装

最初に終了したタスクのみが成功したとみなされ、同じタスクでそれ以降に終了したものは KILL された状態とみなされる。具体的には JobInProgress::completedTask が呼ばれるとまずそのタスクが全体として既に終了したかどうかを確認し、終了していたら alreadyCompletedTask で KILL 状態に設定し、終了していなかったら completed 関数で SUCCEEDED 状態に設定する。

ソース

JobInProgress::completedTask → TaskInProgress::alreadyCompletedTask

4.6.4 マスターの状態の定期的なチェックポインティング

機能概要

マスターの状態を定期的にチェックポインティングすることにより、何か障害が起きて復旧した際に、落ちた時の状態に復旧する事が出来る。

Hadoop の実装

未実装

ソース

該当無し

4.6.5 エラーレコードのスキップ

機能概要

不正な入力やプログラムのバグにより、ある入力レコードで必ずエラーが起きるようなタスクが有った場合、そのレコードを無視をしてジョブ全体を実行する。HTML 等のあいまいなデータを扱う際に特に有用である。

Hadoop の実装

未実装

ソース

HADOOP-153 で現在実装が進められている。

参考文献

`http://issues.apache.org/jira/browse/HADOOP-153`

4.7 考察

Hadoop MapReduce は Google MapReduce の主要な機能を実装しており、分散処理環境としての基本機能を備えている。

Hadoop MapReduce はワーカー障害など、ある程度の耐障害性は備えている。ただし、マスターの状態の定期的なチェックポインティング・エラーレコードのスキップなど一部の耐障害性機能は備えていない。

第 5 章

ソースコード解析

本章では Hadoop のソースコードを調査し、ディレクトリツリーの概要と、各種重要な処理について説明する。

5.1 ソースツリーの概要

ソースコードは src/以下に配置されている。各サブディレクトリに含まれるソースの概要は以下のようになっている。

表 5.1: 各ディレクトリの機能

ディレクトリ	機能
conf	設定ファイルを扱うクラス郡
dfs	分散ファイルシステム HDFS
filecache	分散ファイルキャッシュ
fs	ローカルファイルシステム
io	シリアライズ可能なデータ構造・圧縮コーデック
ipc	IPC(Inter Process Communication)
log	ログレベルの設定
mapred	MapReduce
metrics	サーバーの状態監視
net	ネットワークトポロジー管理
record	任意のデータのシリアライズ・デシリアライズ
security	ユーザーとグループ管理
tools	ログ解析のためのツール
util	ソートやファイルコピーなどの各種ユーテリティー

ここからは、各ディレクトリ内で重要なクラスを中心に見ていく。順番としては、ユーティリティーなど、他のパッケージに依存されている所から開始し、ボトムアップに進めていく。

5.2 org.apache.hadoop.util

このディレクトリには、Hadoop 全体に渡って使用されるユーティリティークラスが格納されている。

5.2.1 MergeSort

マージソートが実装されている。Map の出力をソートする際などに用いられる。

5.2.2 PriorityQueue

プライオリティーキューが実装されている。ジョブを優先度順に管理したりする際などに用いられる。

5.2.3 ReflectionUtils

Java のリフレクションを使用するためのユーティリティー。ReflectionUtils::newInstance は、クラス名を渡すとそのクラスのインスタンスを作成する関数で、色々な箇所で用いられている。

5.2.4 RunJar

Jar ファイルを解凍する。

5.2.5 Tool

MapReduce プログラムを実装する際には、このインターフェースを実装したクラスを用意し、ToolRunner::run で実行を開始させることが出来る。

5.3 org.apache.hadoop.io

5.3.1 Writable

シリアライズ可能な型を表すインターフェース。MapReduce で使用される key, value の型は全てこのインターフェースを継承している必要がある。java.io.DataInput, java.io.DataOutput を経由したシリアライズ・デシリアライズが実装される。デフォルトで IntWritable, LongWritable, FloatWritable, BytesWritable, ArrayWritable, TwoDArrayWritable, MapWritable などが用意されている。

5.3.2 SequenceFile

Key-Value 形式のデータの列をファイルに保存するためのクラス。Key-Value それぞれに対して圧縮をかけて保存したり、ブロック単位で圧縮をかけたりも出来る。

5.3.3 compress

compress ディレクトリ以下には、圧縮をするための各種コーデックや、そのコーデックを利用してブロック単位で圧縮を行う BlockCompressorStream などが含まれる。現在のところ GzipCodec と LzoCodec が実装されている。

5.4 org.apache.hadoop.ipc

5.4.1 VersionedProtocol

プロトコルに変更が加わった際に、バージョンを上げて、古いクライアントと不正な通信を行わないようにするためのインターフェース。新しいバージョンがリリースされたり、互換性の失われる変更が加えられたときにはバージョンが上げられる。

5.4.2 RPC, Server, Client

RPC(Remote Procedure Call) を実現するためのクラス群。この仕組みは主にやり取りするデータ量が少ない場合に用いられ、ブロック転送など大規模なデータを送信する場合は、個々の通信に合わせた通信用のコードが使われる。

典型的なサーバー・クライアントを実装するためのコードは図 5.1 のようになる。

```
Configuration conf = new Configuration();
Server server = RPC.getServer(this, "localhost", 8000, conf);  // localhost:8000 でサーバーを開始
server.start();
```

図 5.1 クライアント側のコード

また、サーバー側のコードは図 5.2 のようになる。この例では ClientProtocol クラスで実装されたプロトコルを通じて通信を行う。

```
Configuration conf = new Configuration();
InetSocketAddress addr = new InetSocketAddress("localhost", 8000);  // サーバーのアドレス
ClientProtocol client = (ClientProtocol)RPC.waitForProxy(ClientProtocol.class,
  ClientProtocol.versionID, addr, conf);
```

図 5.2 クライアント側のコード

最後に ClientProtocol の実装は図 5.3 のようになる。例としてハートビートメッセージを実装した場合のコードが示してある。

サーバークラスは ClientProtocol インターフェースを実装する必要がある。また、ClientProtocol で実装される関数の入力と出力の型が、両方 Writable インターフェースを実装している必要がある。

それさえ満たしていれば、Java のリフレクション機能を用いて、クライアントが非常に簡単な形でサーバーの関数を呼び出すことができる (図 5.4)。

接続に失敗した際には、”ipc.client.connect.max.retries”で設定された回数だけ再試行を行う (デフォルトは 10

```
interface ClientProtocol extends org.apache.hadoop.ipc.VersionedProtocol {
  public static final long versionID = 1L;
  HeartbeatResponse heartbeat();
}
public class HeartbeatResponse implements org.apache.hadoop.io.Writable {
  String status;
  public void write(DataOutput out) throws IOException {
    UTF8.writeString(out, status);
  }
  public void readFields(DataInput in) throws IOException {
    this.status = UTF8.readString(in);
  }
}
```

図 5.3 プロトコルの実装

```
client.heartbeat();
```

図 5.4 クライアント側からサーバー側のコードの呼び出し

回)。接続のタイムアウトは 60 秒 (FSConstants.READ_TIMEOUT) で、再試行の間隔は 1 秒となっている。

5.5 org.apache.hadoop.net

5.5.1 DNS

DNS の逆引き (reverseDns 関数) と、マシンのネットワークインターフェースに設定されている IP を取得する関数 (getIPs 関数) が実装されている。

5.5.2 Node, NodeBase

ノードの階層関係を実現するためのインターフェースとその実装。ノードの階層関係は、”dfs.network.scripts”で指定されるスクリプトによって取得される (3.5.1 参照)。

5.5.3 NetworkTopology

Hadoop が動作しているクラスタのトポロジーを管理するクラス。Node を葉に持つツリーの形でトポロジーが表現されている。スイッチ/ルーターなどは中間ノードとして表現される。

同じ親を持つノードは同じラックに位置しているとみなされる (isOnSameRack 関数)。

ノード間の距離の近さは getDistance 関数で表現される。あるノードから親ノードまでの距離を 1 とする。任意の２ノードの距離は、２つのノードから一番近いノードへの距離を足し合わせたものになる。

5.6 org.apache.hadoop.fs

5.6.1 FileSystem

一般的なファイルシステムにを抽象化したクラスである。このクラスには、ファイルシステムに対する一般的な操作が定義されている。分散ファイルシステムや Amazon S3 用のファイルシステム (サブディレクトリ s3 に含まれる) は、このクラスを継承して作成される。

Hadoop 内では分散ファイルシステムは hdfs://から始まるパス、ローカルファイルシステムは file://で始まるパスで表される。他にも Amazon S3 であれば s3://、Kosmos ファイルシステム [7] であれば kfs://で表される。

createFileSystem 関数に対してこのパス (URI) を渡すと、設定ファイル内の”fs.[scheme].impl”で定義されている項目を参照し、そのクラスのインスタンスを作成する。例えば”fs.hdfs.impl”はデフォルトでは org.apache.hadoop.dfs.DistributedFileSystem に設定されている。

```
Configuration conf = new Configuration();

FileSystem fs1 = FileSystem.getNamed("hdfs:///", conf);
Path inFile = new Path("hdfs:///user/kzk/infile");
FSDataInputStream in = fs1.open(inFile);

FileSystem fs2 = FileSystem.getNamed("s3:///", conf);
Path outFile = new Path("s3:///user/kzk/outfile");

FSDataOutputStream out = fs2.create(outFile);
while((bytesRead = in.read(buffer)) > 0){
  out.write(buffer, 0, bytesRead);
}
in.close();
out.close();
```

図 5.5 異なるファイルシステム間のファイルコピー操作

以上の仕組みにより、各種ファイルシステムに対する操作を実装によらず透過的に実行する事が出来るようになっている。ファイルコピー操作を行うコードの一例を図 5.5 に掲載する。

5.6.2 LocalFileSystem

ローカルファイルシステムを実装したクラス。FileSystem クラスを継承している。

5.6.3 InMemoryFileSystem

メモリ内にデータを保持するためのファイルシステム。ただし、このファイルシステムを使用するには予め格納するファイルのサイズが分かっている必要がある。まず reserveSpace 関数もしくは reserveSpaceWithCheckSum 関数で、あるパス用のメモリを確保しておき、その領域に対してデータを書き込む形になる。InMemoryFileSystem は ReduceTask において、Key 毎に Value をマージするのに用いられる。

5.6.4 FSOutputSummer, FSInputStream

FileSystem に対する入出力に用いられる。

5.6.5 Path

一般的な Path の扱いに関する操作が実装されている。

5.6.6 Trash

ゴミ箱機能を実装している。HDFS 上ではファイルは削除されても、データはすぐには消去されず、ある一定期間経った後に削除される (3.3.5)。Emptier クラスが実際にデータを削除するスレッドとなっている。

5.6.7 FileUtil

copy や完全消去など、ファイル操作に関するユーティリティー関数が実装されている。

5.6.8 FsShell

ファイルシステム操作を行うコマンドラインツールが実装されている。

5.6.9 DU, DF

UNIX の `du` コマンド、`df` コマンドを使用してディスクの使用状況を確認するためのクラス。DataNode が現在のディスク使用状況を確認するために用いられる。

5.7 org.apache.hadoop.dfs

5.7.1 ClientProtocol

クライアントから NameNode に対して RPC する際のプロトコル。

5.7.2 DatanodeProtocol

DataNode が NameNode に対して RPC する際のプロトコル。

5.7.3 NamenodeProtocol

Balancer が NameNode に対して RPC する際のプロトコル。

5.7.4 DistributedFileSystem

FileSystem(5.6.1) を継承したクラス。このクラスでは分散ファイルシステム”hdfs”についての実装がなされており、このクラスのメソッドを呼ぶと DFSClient クラスを用いたファイル操作が行われる。

5.7.5 DFSClient

DFSClient は HDFS にアクセスするためのクライアントロジックが実装されたクラスである。open(), create(), exists(), listPaths(), mkdir() など、各種ファイルシステムに対する操作が実装されている。

コンストラクタから呼ばれる createNamenode 関数で NameNode への接続を行う。プロトコルには ClientProtocol を用いる。

このクラスの内部クラスとして重要なのは、DFSInputStream と DFSOutputStream である。それぞれ HDFS からの入力・出力を行うためのクラスである。

DFSInputStream

DFSInputStream では、NameNode からブロックの位置を取得した後、DataNode からデータを転送する処理が行われる。この処理を実際に担当しているのは BlockReader クラスである。BlockReader は DFSInputStream の blockSeekTo 関数の内部で作られる。

BlockReader では RPC 経由で読み込むのではなく、別に Socket を作成しそれを使用してブロックの内容を読み込む。この際、受信したデータのチェックサムが、ブロックに付随したチェックサムと一致するかどうかを確認する。チェックサムが一致しない場合は、同じブロックを保持している他の DataNode からのデータ読み込みを試みる。同じブロックを保持している全ての DataNode から読み込みに失敗すると、読み込みが失敗する (DFSInputStream::readBuffer 関数)。

DFSOutputStream

DFSOutputStream が書き込む際には、データは 64K バイトの”Packet”と呼ばれる単位に分割される。それぞれのパケットは 512K バイトのチャンクを形成する。それぞれのチャンクにはチェックサム情報が付随する。DFSOutputStream でもデータの転送用に Socket が生成される。

パケットはまず dataQueue に入れられる。DataStreamer スレッドが dataQueue からパケットを拾い上げ、数珠繋ぎの最初の DataNode にパケットを転送する。送信したパケットは ackQueue に移動する。DataNode はパケットを受け取ると ack パケットをクライアントに返す。ResponseProcessor クラスでは DataNode からの ack を待ち受ける。

全ての DataNode からの ack が返ってきたらそのパケットを ackQueue から除外する。何らかのエラーが起きた場合は ackQueue から dataQueue にパケットを移動し、エラーが起こった Datanode を除外して再度転送を試行する (DataStreamer::processDatanodeError 関数)。

パケット転送時には、パケットのサイズ・ブロック内でのオフセット・シーケンス番号・ブロック内の最後のパケットかどうかのフラグ・実際のデータが送信される (DataStreamer::run 関数)。

5.7.6 DataNode

DataNode は実際のブロックデータを保持するサーバーである。クライアントからのブロック読み込みや作成の要求に答えるのに加えて、NameNode からのブロックの削除やレプリケーション等の要求にも答える。

DataNode は定期的に NameNode に対して HeartBeat メッセージを送信する (DataNode::offerService 関数)。HeartBeat メッセージには DataNode のブロック情報や統計情報等が含まれている。

送信には RPC 機構を用い、プロトコルには DatanodeProtocol が用いられる。HeartBeat メッセージを送信すると、DatanodeCommand が返って来る事が有る。このコマンドには、ブロックを削除などの NameNode からの命令が含まれる。NameNode からの指令は全て HeartBeat メッセージに対する返答の形で行われる。

DataNode はデータ転送用のソケットを公開しており、その情報は NameNode に報告される。クライアントは NameNode からその場所を聞き出し、そこに繋ぎに行く事でデータ転送を行う事が出来る。

5.7.7 NameNode

NameNode はディレクトリツリーの名前空間と、各ファイル情報を管理する。クラスタ内には NameNode は 1 つしか無いことが仮定されている。

NameNode は ClientProtocol を実装し、クライアントに対してファイルシステムのインターフェースを提供する。DatanodeProtocol も実装し、DataNode からの HeartBeat メッセージ (ブロック情報等) を受け取ったりもする。

5.7.8 FSNamesystem

ディレクトリ操作を受け持つクラスである。NameNode での ClientProtocol の実装では、ほぼ FSNamesystem の各ファイル操作関数を呼び出している。NameNode は単純にクライアントに対して RPC のインターフェースを見せるためのクラスになっている。

FSNamesystem では以下の情報が保持されている。

- (1) ファイル名からブロックリストへのマッピング
- (2) ブロックからファイル名へのマッピング ((1) の逆)
- (3) ブロックからノードリストへのマッピング
- (4) ノードからブロックリストへのマッピング ((3) の逆)

これらを使用して HDFS のファイル操作を実現している。

FSDirectory

FSNameSystem の中でディレクトリに関する情報を管理している。

INode

ファイルの情報を保持しているクラス。この情報はメモリ上に保持される。

BlocksMap

ブロックについての情報を保持している。保持されている情報にはそのブロックが所属する INode と、そのブロックの前後のブロックについての情報が有る。

5.7.9 FSImage, FSEditLog

FSImage はファイルのチェックポインティングやログファイルを扱うクラス。

FSImage クラスが保持している FSEditLog クラスでは特にオペレーションログが管理される。

5.7.10 ReplicationTargetChooser

レプリケーションを行う先を決定するクラス。レプリケーションを行う際にはデータの空き容量などは見ていない。

レプリケーション先の DataNode は次のロジックで決定される。まず、書き込みプロセスと同じホストに DataNode が有るならばまずそこに書かれ、そうでなければランダムに選択される。2 番目のレプリカは 1 番目のレプリカと異なるラック上に置かれる。3 番目のレプリカは 1 番目のレプリカと同じラック上に置かれる。それ以上についてはランダムに選択される。

5.7.11 SecondaryNameNode

SecondaryNameNode は定期的に NameNode のチェックポインティングを行う。NameNode のオペレーションログが”fs.checkpoint.size”を上回った場合についてもチェックポインティングが行われる。チェックポイントが行われると、NameNode が保持しているオペレーションログは削除される。これによりログの肥大化が防がれている。

データは”fs.checkpoint.dir”で設定されるディレクトリに保存される。SecondaryNameNode が NameNode と通信する際には ClientProtocol を用いる。

5.7.12 Balancer

Balancer は DataNode 間のディスク使用量の偏りを補正するためのツールである。たとえば新しい DataNode を追加した際には現在の HDFS の実装では、自動的にディスク使用量の多い DataNode からデータを移動してくれず、管理者が明示的に Balancer を実行する必要がある (3.4.2)。実際のアルゴリズムに関しては 3.4.2 の参考文献参照。

5.7.13 NamenodeFsck

HDFS に対してチェックをかけるツール。コマンドとして管理者が実行する事ができる [3]。

どの DataNode にも保持されていないブロックが有る場合はそのファイルを報告したり、レプリケーションが足りなかったり余分な場合に報告を行ってくれる。

修復はこのツールでは行われない。エラーが有った場合には NameNode が自動的にリカバリーする。

5.8 org.apache.hadoop.mapred

5.8.1 JobConf

JobConf はプログラマが MapReduce ジョブの実行時の設定を行うためのインターフェースである。通常 JobConf を使用して、以下の項目が指定される。

- ジョブ名 (setJobName)
- Mapper クラス名 (setMapperClass)
- Combiner クラス名 (setCombinerClass)
- Reducer クラス名 (setReducerClass)
- InputFormat クラス名 (setInputFormat)
- OutputFormat クラス名 (setOutputFormat)
- 入力パス (setInputPath)
- 出力パス (setOutputPath)

基本的な JobConf の利用方法は図 5.6 のようになる。

```
// Create a new JobConf
JobConf job = new JobConf(new Configuration(), MyJob.class);

// Specify various job-specific parameters
job.setJobName("myjob");

job.setMapperClass(MyJob.MyMapper.class);
job.setCombinerClass(MyJob.MyReducer.class);
job.setReducerClass(MyJob.MyReducer.class);

job.setInputFormat(SequenceFileInputFormat.class);
job.setOutputFormat(SequenceFileOutputFormat.class);

job.setInputPath(new Path("in"));
job.setOutputPath(new Path("out"));
```

図 5.6 JobConf の設定

さらにジョブの細かい挙動を制御することが出来る (図 5.7)。

5.8.2 InputFormat

InputFormat は MapReduce ジョブの入力についての情報を表現するためのクラスである。InputFormat は主に３つの仕事を請け負う。

- 入力仕様のバリデーション (validateInput 関数)
- 各 Mapper に処理させるために入力データを分割する (getSplits 関数)

```
// Map タスクの数
conf.setNumMapTasks(100);
// Reduce タスクの数
conf.setNumReduceTasks(40);
// Map タスク失敗時に呼ばれるスクリプトのパス
conf.setMapDebugScript("/home/kzk/debug/map-fail.sh");
// Reduce タスク失敗時に呼ばれるスクリプトのパス
conf.setReduceDebugScript("/home/kzk/debug/reduce-fail.sh");
// Map の出力を圧縮
conf.setCompressMapOutput(true);
// 最終出力を圧縮
conf.setBoolean("mapred.output.compress", true);
// MapReduce プログラムをローカルで実行
conf.set("mapred.job.tracker", "local");
conf.set("fs.default.name", "local");
```

図 5.7 JobConf による細かな挙動の設定

- InputSplit(入力の断片) を読み込むための RecordReader を返す (getRecordReader 関数)

getSplits 関数では入力を分割し InputSplit インターフェースを実装したクラスを返す。多くのクラスでは単純にファイルを分割した FileSplit が返される。

getRecordReader 関数では、ファイルから Key-Value のペア (レコード) を読み込むための RecordReader インターフェースを実装したクラスを返す。RecordReader::next 関数でレコードを読み進めることが出来る。

以下ではよく使用すると思われる InputFormat について述べる。

TextInputFormat

TextInputFormat はファイルをバイト単位で分割するような InputFormat である。一定サイズのブロックにファイルを分割する。getRecordReader 関数では LineRecordReader クラスを返す。ただし圧縮されているファイルについては分割されない。

デフォルトではこのクラスが InputFormat として使用される。

KeyValueTextInputFormat

KeyValueTextInputFormat はタブ文字区切りで Key-Value が一行となったデータを入力とするような InputFormat である。KeyValueTextInputFormat についてもファイルは一定サイズのブロックに分割される。getRecordReader 関数では KeyValueLineRecordReader クラスを返す。ただし圧縮されているファイルについては分割されない。

5.8.3 OutputFormat

OutputFormat は MapReduce ジョブの出力についての情報を表現するためのクラスである。OutputFormat は主に 2 つの仕事を請け負う。

- 出力仕様のバリデーション (checkOutputSpecs 関数)
- ファイルを出力するための RecordWriter を返す (getRecordWriter 関数)

デフォルトでは TextOutputFormat が OutputFormat として使用される。このクラスでは `key\tvalue` という形式で最終結果をテキストファイルで出力する。OutputFormatBase::setCompressOutput 関数を呼び出すことで最終結果を圧縮するかどうかを決定できる。

5.8.4 JobClient

Job を JobTracker にサブミットするためのクラス。現在実行中のジョブリストや、各ジョブのステータスを問い合わせるのにも使われる。

JobClient.runJob 関数で Job のサブミットが行われる。サブミットが行われると、JobTracker に定期的にサブミットした Job の状態を問い合わせる。

5.8.5 JobTracker

JobTracker は Job の管理と、TaskTracker への Task の割り振りを行う。JobClient は JobTracker の submitJob 関数を RPC 経由で呼び出し、Job のサブミットを行う。

Job はまず jobInitQueue に add される。JobInitThread がそれを処理し、JobInProgress::initTasks 関数で Job の初期化を行う。この際に InputSplit を用いて入力の分割等が行われる。

TaskTracker からは HeartBeat メッセージが定期的に送られて来る (heartbeat 関数)。メッセージの返り値として TaskTracker に行わせる作業を返す。作業内容は TaskTrackerAction クラスを継承したクラスで表され、LaunchJobAction, KillJobAction, KillTaskAction, ReinitTrackerAction 等がある。TaskTracker に新しい Task を行わせたい場合は LaunchTaskAction を用いる。

新しくどの Task を TaskTracker に割り当てるかは getNewTaskForTaskTracker 関数で決定される。この関数ではまず Map タスクの割り当てを試み、次に Reduce タスクの割り当てを試みる。この関数から JobInProgress クラスの obtainNewMapTask 関数・obtainNewReduceTask 関数が呼び出される。

この 2 つの関数はさらに findNewTask 関数を呼び出す。通常時はこの関数で実行すべきタスクが選択される。バックアップタスクなどが発動する条件はこの関数で呼び出されている TaskInProgress::hasSpeculativeTask に条件が記述されている。以下の条件が全て満たされるとき、SpeculativeTask が実行される。

- Task の実行数に空きが有る
- 設定ファイルで SpeculativeTask を実行することが許されている
- 全てのタスクの平均完了率と、該当タスクの完了率が SPECULATIVE_GAP(2 割) 以上離れている
- 実行開始から SPECULATIVE_LAG(60 秒) 以上経過している
- まだ Task が完了していない

5.8.6 TaskTracker

TaskTracker は実際に Task を実行するサーバーである。offerService 関数で定期的に JobTracker に対して HeartBeat メッセージを送り、返り値として自分が次に行うべきタスクの情報を得る。LaunchTaskAction が渡ってくると、タスクの実行を開始する (startNewTask 関数)。

startNewTask 関数からは localizeJob 関数が呼ばれ、実行するプログラムの jar 等を HDFS から読み込みローカルに展開する。次に launchTaskForJob 関数を呼び出し、実際にタスクを実行する (TaskInProgress::launchTask 関数)。

launchTask 関数内ではまず localizeTask 関数が呼ばれ、タスクの出力先ディレクトリ等を準備する。その後 createRunner 関数で作成した TaskRunner を使用してジョブを開始する。

TaskRunner

TaskRunner ではユーザーがサブミットしたプログラムを実際に実行する作業を行う。実際には java コマンドに各種オプションを付けて子プロセスとして実行する。

MapTask

MapTask クラスは Map タスクを実行するためのクラスである。

run 関数で実際に Map タスクが実行される。run 関数では Map の結果を出力するための collector 等が作られ、MapRunner::run 関数で実際のユーザープログラムが実行される。MapRunner::run 関数では RecordReader を用いてレコードを読み取り、それをユーザーが定義した map 処理に渡していく。

collector には、Reduce タスクが無い場合は DirectMapOutputCollector、そうではない場合は MapOutputBuffer が使われる。今回は通常用いられる MapOutputBuffer についてのみ記述する。

MapOutputBuffer::collect 関数が実際にユーザープログラムが map の結果を出力するために用いられる関数である。この関数ではまず出力されたレコードをメモリ上にバッファリングするために、ReduceTask の数と同じだけの MergeSorter クラスのインスタンスを確保する。出力されたレコードは MergeSorter::addKeyValue 関数に渡され、メモリ上に溜められていく。

すべての MergeSorter クラスで使用されているメモリの合計がある一定サイズ (maxBufferSize) 以上になると、それらの領域をディスクに書き出すためのスレッド (bufferWriter) が生成され、sortAndSpillToDisk 関数が呼び出される。

sortAndSpillToDisk 関数では各 MergeSorter について、まずレコードのソートを行う (pendingSortImpl[i].sort())。次に Combiner が指定されている場合は combine の処理も行う。最後にメモリ上のレコードを RecordWriter を使用してディスクに書き出す (spill 関数)。startPartition 関数で RecordWriter が作成され、endPartition 関数でそれが破棄される。このソートされたディスク上のファイルを Partition と呼ぶ。

以上の操作はメモリがあふれる度に行われるため、各 ReduceTask に対して複数個の Partition がディスクに書かれる。run 関数で最後に collector::flush 関数が呼ばれるが、この関数から呼ばれる mergeParts 関数で複数個の Partition を全てマージし、1 つのソートされた出力を得る。マージはディスク上で行われ、SequenceFile::Sorter ク

ラスが使用される。

以上の操作により、map の出力がディスクに書き込まれる。出力は各 ReduceTask について別々のファイルになっており、各ファイル内はソートされた状態になっている。

ReduceTask

ReduceTask クラスは Reduce タスクを実行するためのクラスである。

run 関数で実際に ReduceTask が実行される。まず ReduceCopier クラスのインスタンスが作成され、fetchOutputs 関数を用いて Map タスクの出力を取得する。取得する順番はシャッフルし、他の Reducer となるべく被らないような工夫がなされている。全ての Map タスクからの出力を取得した後、それらが 1 つのファイルにマージされる。

run 関数では次に実際の reduce 処理を行う。ReduceValuesIterator と、Reduce の結果を出力するための collector を reduce 関数に渡す。collector の colect 関数では RecordWriter を用いて出力をファイルに書き出す。

Mapper

map 関数を持つインターフェース。ユーザープログラムではこのインターフェースを実装し、map 処理を記述する。

Reducer

reduce 関数を持つインターフェース。ユーザープログラムではこのインターフェースを実装し、reduce 処理を記述する。

5.8.7 StatusHttpServer

JobTracker, TaskTracker は StatusHttpServer を内臓し、現在の状態を HTTP で見ることが出来るようになっている (4.4.1)。HTTP サーバーには Jetty[6] が使用されており、各種数値をテーブルに表示したりグラフを描画するなどの機能が有る。

5.9 考察

全体的に上手く抽象化されており、ソフトウェアとして非常に良く出来ている。

Map 処理や Reduce 処理でのソート部分など、速度が必要な部分については多少複雑にはなっているが、そのぶん最適化が施されている。またエラー処理についても全体に渡って気が配られている。

しかし、ディスク使用率を測定する部分に UNIX システムを仮定しているなど、ポータビリティという面では少し課題が残る。

品質管理という面ではユニットテストも充実しており、一部では分散環境をエミュレーションするコードを使用してテストが行われる。

第 6 章

性能評価

基本的なベンチマークプログラムを用いて、HDFS と Hadoop MapReduce の基本性能を評価した。

6.1 評価環境

評価環境として 12 台のマシンを用意した。全てのマシン上で DataNode と TaskTracker を動かした。またその内 1 台はマスターノードとし、NameNode と JobTracker も動かした。

それぞれのノードのスペックを表 6.1 に示す。各ノードは 100MBps の Ethernet で繋がれている。

表 6.1 評価用マシンの性能

CPU	Intel Xeon E5430 2.66 GHz Quad Core
Memory	16G
Disk	SAS
OS	Linux 2.6.18-53.1.14.el5PAE
NIC	Broadcom NetXtreme II BGM5708 Gigabit Ethernet
I/O Scheduler	CFQ(Completely Fair Queing)

6.1.1 ディスク速度の測定

ハードディスクベンチマークツール bonnie++[1] を用いて各マシンのディスク性能を測定した。このツールを用いると、シーケンシャル read/write、ランダムアクセスの性能測定を行う事が出来る。bonnie++ のビルド方法・実行方法・実行結果を図 6.1 に示した。

結果は連続書き込み (ブロック単位) が 80.2MB/sec、連続読み込み (ブロック単位) が 94.2MB/sec、秒間ランダムシーク回数が 347.7 回であった。

6.2 HDFS ベンチマーク

HDFS の性能を測るため、MapReduce を使用して書き込みと読み込みを行った。書き込み、読み込みベンチマーク共に Hadoop に付属する TestDFSIO(hadoop-0.16.4-test.jar に含まれる) を用いた。なお、ベンチマークは全てレプリケーション数を 1 にした時の値である。レプリケーション数を 3 にするとジョブの失敗が頻繁に起こる事が確認

```
$ tar vzxf bonnie++-1.03c.tar.gz
$ cd bonnie++-1.03c
$ ./configure
$ make
$ ./bonnie++
Version 1.03c       ------Sequential Output------ --Sequential Input- --Random-
                    -Per Chr- --Block-- -Rewrite- -Per Chr- --Block-- --Seeks--
Machine        Size K/sec %CP K/sec %CP K/sec %CP K/sec %CP K/sec %CP  /sec %CP
tmr001          32G 44896  66 80263  14 39105   7 66683  94 94257  11 347.7   0
                    ------Sequential Create------ --------Random Create--------
                    -Create-- --Read--- -Delete-- -Create-- --Read--- -Delete--
              files  /sec %CP  /sec %CP  /sec %CP  /sec %CP  /sec %CP  /sec %CP
                 16 99920  98 585100  96 116893 100 101574  99 917100 100 121861 100
```

図 6.1　bonnie++ のビルド、実行

された。

6.2.1 書き込みベンチマーク

書き込みベンチマークでは図 6.2 のコマンドを用いて 1G のファイルを 100 個生成した。書き込みに関しては 5.7.10 で述べたように必ずローカルに書かれる。

```
$ ./bin/hadoop jar hadoop-0.16.4-test.jar TestDFSIO -write -nrFiles 100 -fileSize 1000
```

図 6.2　1G * 100 個データ生成コマンド

マシン台数と時間当たりのデータ書き込み量の関係を図 6.3 に示す。

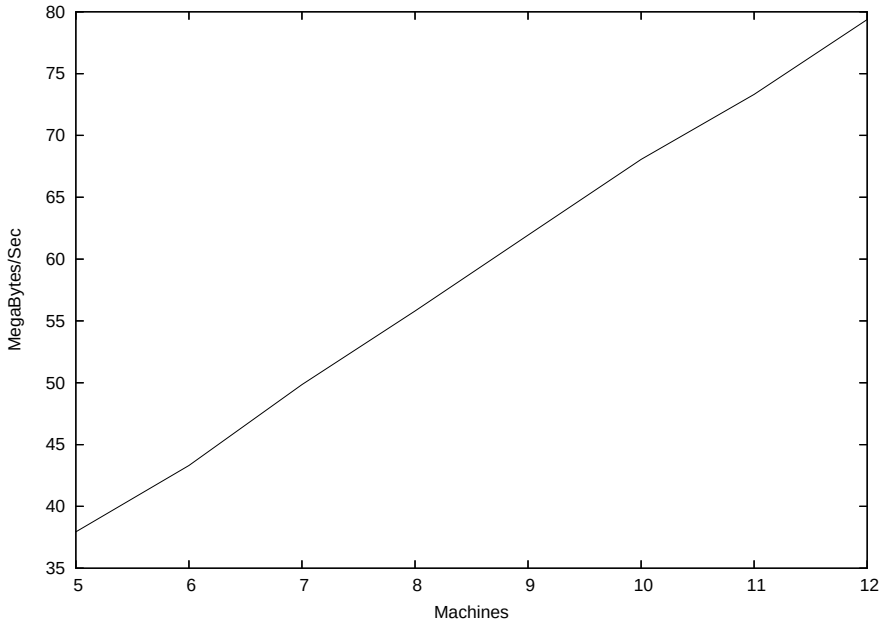

図 6.3　1G * 100 個データ生成 (秒間生成バイト数 (MB) / 台数)

マシン台数について綺麗にスケールしている事がわかる。

6.2.2 読み込みベンチマーク

次に書き込んだ 100G のデータを図 6.4 のコマンドで読みこむ。読み込みに関してもローカリティを考慮したタスクの分配が行われるため、ネットワーク転送はおこりにくくなっている。

```
$ ./bin/hadoop jar hadoop-0.16.4-test.jar TestDFSIO -read -nrFiles 100 -fileSize 1000
```

図 6.4 1G * 100 個データ読み込みコマンド

マシン台数と時間当たりのデータ読み込み量のグラフを図 6.5 に示す。

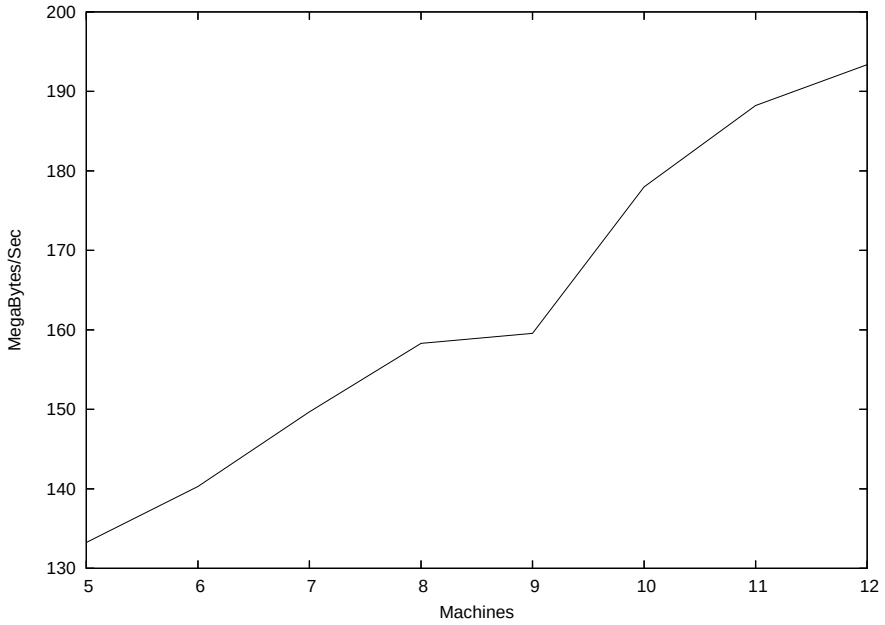

図 6.5 1G * 100 個データ読み込み (秒間読み込みバイト数 (MB) / 台数)

こちらも概ね台数に対して性能が比例している様子が見て取れる。

6.3 ソートベンチマーク

MapReduce の基本的な性能を測るため、サンプルプログラムに添付されているソートプログラムを用いて性能評価を行った。今回は 100G のデータを生成し、それをソートした。以下の実験ではレプリケーション数は 1 として実験を行った。

6.3.1 データの生成

データの生成には Hadoop に添付されている randomwrite プログラムを使用した。100G のデータを生成するために図 6.6 に示される設定ファイルを用意した。この設定ファイルでは Key, Value それぞれの長さが 10〜1000 バイトのレコードが合計 100G 生成される。それを 1G 毎に分割し、それぞれを 1 つの Map タスクに割り当てる。Key-Value 長の合計値の平均は約 1KB なので、全体のレコード数は平均 100M エントリと考えられる。

図 6.7 のコマンドでこの設定ファイルを使用し、実際にデータを生成した。

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
    <name>test.randomwrite.min_key</name>
    <value>10</value>
  </property>
  <property>
    <name>test.randomwrite.max_key</name>
    <value>1000</value>
  </property>
  <property>
    <name>test.randomwrite.min_value</name>
    <value>10</value>
  </property>
  <property>
    <name>test.randomwrite.max_value</name>
    <value>1000</value>
  </property>
  <property>
    <name>test.randomwriter.bytes_per_map</name>
    <value>1000000000</value>
  </property>
  <property>
    <name>test.randomwrite.total_bytes</name>
    <value>100000000000</value>
  </property>
</configuration>
```

図 6.6 100G 生成のための設定ファイル (randomwriter.conf)

```
$ ./bin/hadoop jar hadoop-0.16.4-examples.jar randomwriter -conf randomwriter.conf random
```

図 6.7 100G 生成のためのコマンド

使用したマシン台数と 100G データの書き込みに掛かった時間のグラフを図 6.8 に示す。

またマシン台数と秒間スループットの関係を図 6.9 に示す。マシン台数に秒間生成バイト数が概ね比例している事が分かる。

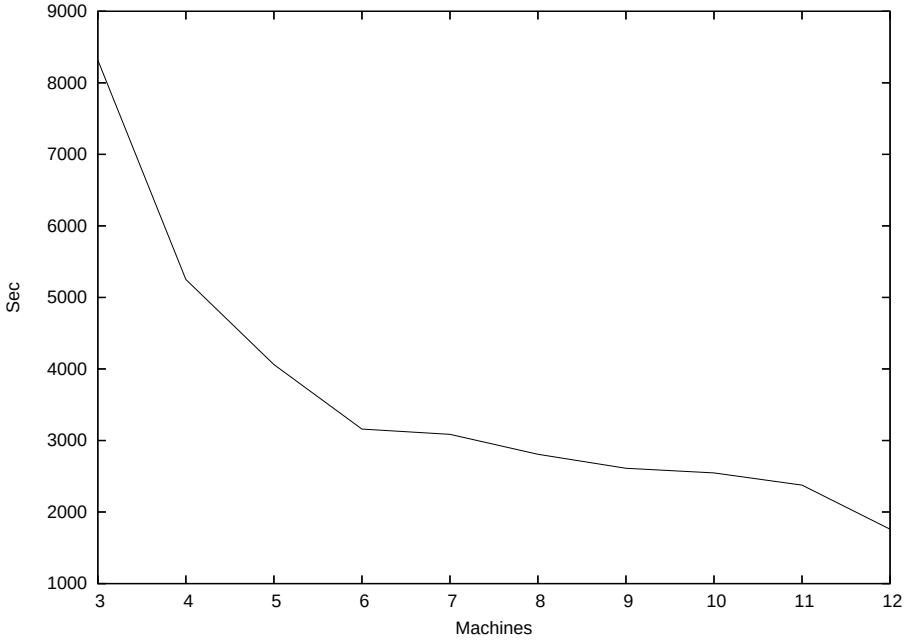

図 6.8　100G データ生成 (秒数 / 台数)

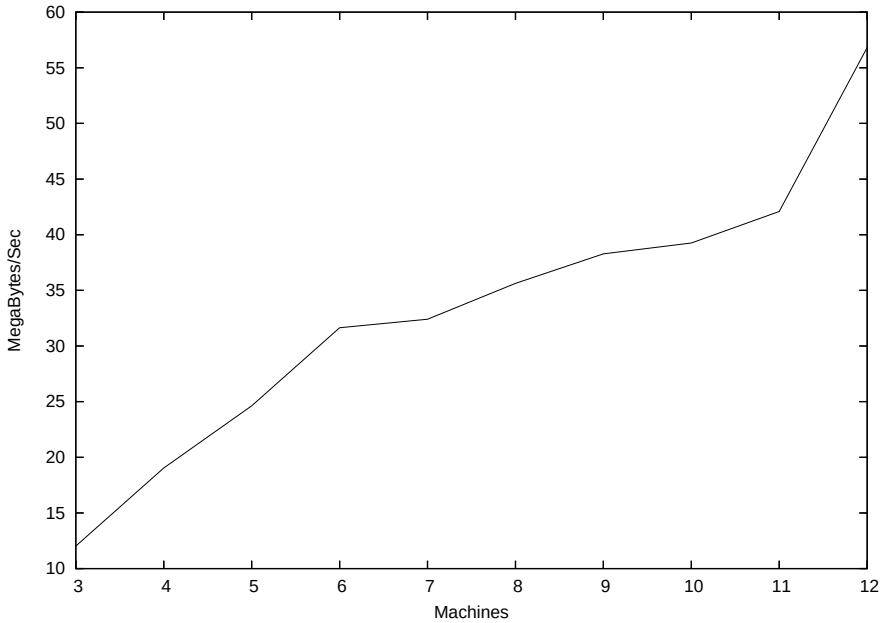

図 6.9　100G データ生成 (秒間生成バイト数 (MB) / 台数)

6.3.2 データのソート

上記のように生成したファイルを今度は図 6.10 のコマンドでソートした。

```
$ ./bin/hadoop jar hadoop-0.16.4-examples.jar sort random radom-sort
```

図 6.10 ソートのためのコマンド

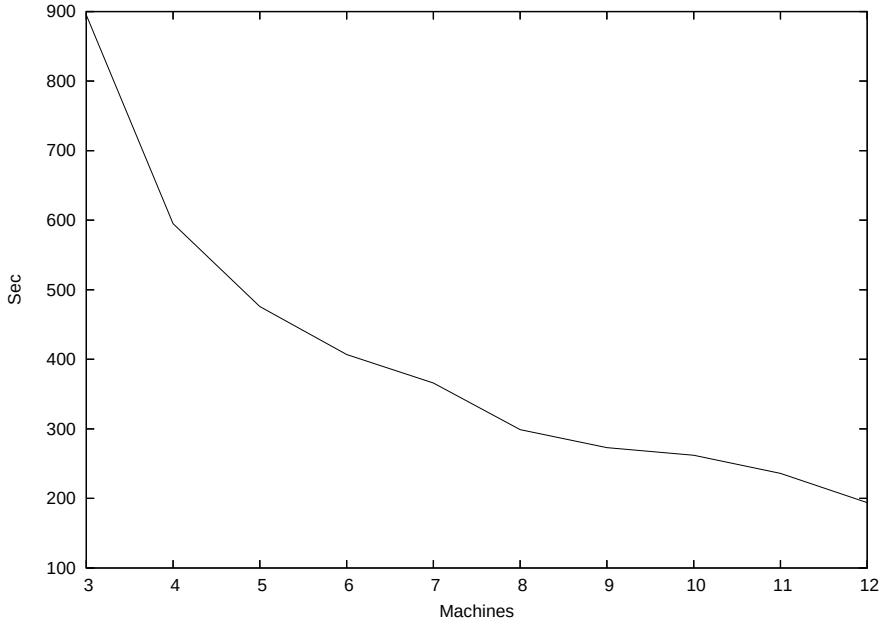

図 6.11 100G ソート (秒数 / 台数)

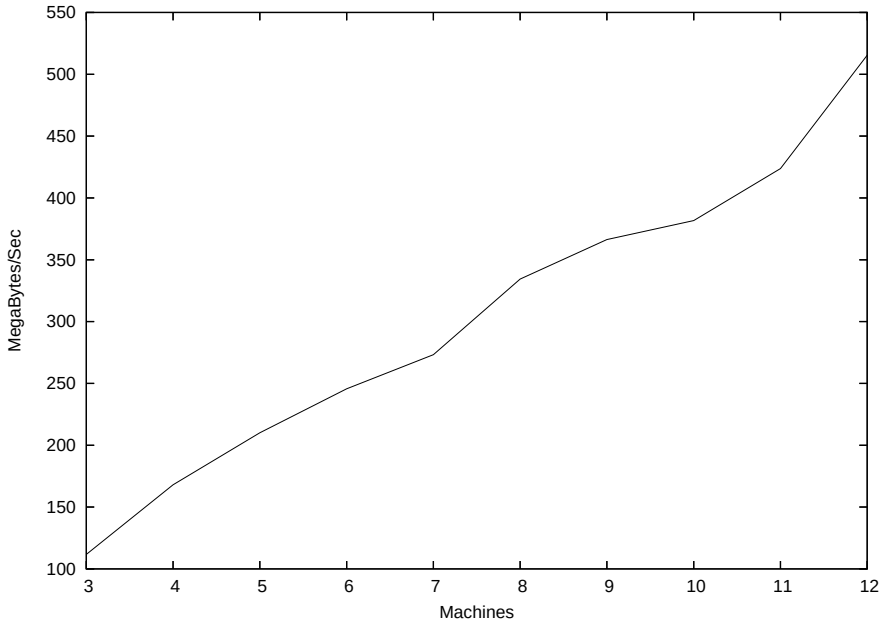

図 6.12 100G ソート (秒間処理バイト数 (MB) / 台数)

使用したマシン台数とソートに掛かった時間のグラフを図 6.11 に示す。またマシン台数と秒間スループットの関係を図 6.12 に示す。秒間処理バイト数がマシン台数に比例している事が分かる。

6.4 考察

本章での実験により、Hadoop は 12 台のサーバ台数に対して十分なスケーラビリティがあることを確認できた。

本章の実験ではネットワークがボトルネックとなることはなかった。これは、レプリケーション数が全て 1 であるのと、Hadoop がネットワーク転送量がなるだけ少なくなるように動作したためであろう。

レプリケーション数が 3 の場合に、適切に実験を行えなかったことがあった。これは、なんらかの設定誤りか、Hadoop の安定性に原因があると思うが、詳細は不明である。

第 7 章

まとめ

本調査のまとめを述べる。

本調査では、まず、Hadoop が備えている機能を把握する為、GFS, Google MapReduce の論文に記載している機能の Hadoop における実装の有無を比較した。その結果、Hadoop は論文の主要な機能を備えており、分散ソフトウェアとして十分な機能を備えていることが確認できた。ただし、主に信頼性に関する機能を中心に、論文の機能の一部を備えておらず、信頼性の観点では課題が残っている。

次に、Hadoop の詳細な実装方式を調査する為、ソースコードレベルの解析を行った。その結果、Hadoop が性能面を考慮した実装方式を採用していることを確認した。

最後に、実験により、Hadoop が 12 台のマシンに対してスケーラビリティを備えていることを確認した。

以上の調査結果より、信頼性を中心に課題を残すものの、Hadoop は十分な実用性を備えているとの結論が得られた。

参考文献

[1] Bonnie++ project homepage. http://www.coker.com.au/bonnie++/.

[2] Commons logging. http://commons.apache.org/logging/.

[3] Hadoop dfs user guide. http://hadoop.apache.org/core/docs/current/hdfs_user_guide.html.

[4] Hadoop project homepage. http://hadoop.apache.org/core/.

[5] Hadoop streaming documentation. http://hadoop.apache.org/core/docs/current/streaming.html.

[6] Jetty. http://www.mortbay.org/jetty-6/.

[7] Kosmos filesystem. http://kosmosfs.sourceforge.net/.

[8] Lucene project homepage. http://lucene.apache.org/.

[9] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[10] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.