分散 database Jungle に関する研究

A Study of distributed database Jungle

平成25年度 学位論文(修士)

琉球大学大学院　理工学研究科
情報工学専攻

大城 信康

要旨

スマートフォンやタブレット端末の普及により, 大量の通信を扱うウェブサービスが現れてきている. それに伴い, サーバサイド側への負荷も増大しウェブサービスがダウンする事態が出てきている. そのため, スケーラビリティはウェブサービスにおいて重要な性質の 1 つとなっている. スケーラビリティとは, ある複数のノードから構成される分散ソフトウェアがあるとき, その分散ソフトウェアに対して単純にノード を追加するだけで性能を線形に上昇させることができる性質である. そこで, スケーラビリティを持たせるためにアーキテクチャの設計から考えることにした. 当研究室では非破壊的木構造を用いたデータベースである Jungle を開発している. 非破壊的木構造とは, データの編集の際に一度木構造として保存したデータを変更せず, 新しく木構造を作成してデータの編集を行うことを言う.

本研究では, Jungle に分散データベースと永続性の実装を行った. データ分散部分には当研究室で開発中である並列分散フレームワークである Alice を使用した. 結果, 学科の並列環境を用いて複数のサーバノード間でデータの分散を行うことを確認した. また, 例題アプリケーションとして簡易掲示板プログラムの作成を行った. Jungle と Cassandra により作成した掲示板プログラムに対して読み込みと書き込みの負荷をかけ比較を行った.

Abstract

Smartphone and tablet pc are widely used, thereby Web services that handle large amounts of data are emerging. It has caused the webserver is down. Therefore, scalability is important software factor today. Scalability in distributed system is able to increase performance linearly when just added new node to system. In order to make provide scalability, we considered design of architecture.

We are developing a database Jungle. It is use non-destructive tree structure. Non-destructive tree structure is not the destruction of data. Editing of data is done creating by new tree. Jungle was designed as a distributed database. But data distribution and persistent has not yet been implemented in the Jungle.

In this paper, we develop distributed database on jungle for pursuit architecture with scalability. Distributed data on Jungle is developing using parallel distributed framework Alice. As a result, we confirmed that data is distributed between the server node.

目 次

図 目 次

表 目 次

第1章　序論

1.1　序論

1.1.1　研究背景と目的

スマートフォンやタブレット端末の普及により, 大量の通信を扱うウェブサービスが現れてきている. それに伴い, サーバサイド側への負荷も増大しウェブサービスがダウンする事態が出てきている. そのため, スケーラビリティはウェブサービスにおいて重要な性質の 1 つとなっている. スケーラビリティとは, ある複数のノードから構成される分散ソフトウェアがあるとき, その分散ソフトウェアに対して単純にノード を追加するだけで性能を線形に上昇させることができる性質である. そこで, スケーラビリティを持たせるためにアーキテクチャの設計から考えることにした. 当研究室では非破壊的木構造を用いたデータベースである Jungle を開発している. 非破壊的木構造とは, データの編集の際に一度木構造として保存したデータには変更せず, 新しく木構造を作成してデータの編集を行うことを言う.

本研究では, Jungle に分散データベースと永続性の実装を行った. データ分散部分には当研究室で開発中である並列分散フレームワークである Alice を使用した. 結果, 学科の並列環境を用いて複数のサーバノード間でデータの分散を行うことを確認した.

1.1.2　本論文の構成

本論文では, 始めに分散データベースについて既存の製品を例に上げながら述べる. 第 3 章では, 非破壊的木構造による Jungle の基本設計と, 分散バージョン管理システムを参考にした分散設計について述べる. 第 4 章では, 第 3 章で行った設計を第 5 章では, 第 4 章で実装した分散データベース Jungle の評価を行うため, 簡易掲示板プログラムを実装する. この掲示板プログラムは Jungle と Cassanra それぞれのデータベースを使うものを用意した. 学科の並列環境上で開発した掲示板プログラムを複数のノードで実行させ, 負荷をかけることで Jungle と Cassandra の性能比較を行う. 第 6 章は, 本研究におけるまとめと今後の課題について述べる.

第2章　既存の分散データベース

本章ではまずデータベースの種類であるリレーショナルデータベース（RDB） と NoSQL について述べる. 次に, 分散データシステムにおいて重要な CAP 定理について触れる. 最後に既存の NoSQL データベースとして Cassandra, MongoDB, Neo4j の特徴について述べる.

2.1　RDB と NoSQL

データベースは大別すると RDB と NoSQL に分けられる. RDB とは行と列からなる 2 次元のテーブルによりデータを保持するデータベースである. RDB はデータベースアクセス言語として SQL 言語を持ち, 一台のマシンでデータを扱う分には最適である. しかし, RDB はマシン単体以上の処理性能をだすことができない. そこで, 汎用的な PC をいくつも用意しデータや処理を分散して管理できるデータベースが求められた. それらのデータベースは NoSQL(Not Only SQL) と呼ばれる. 2 次元のテーブルでは無く, Key-Value, ドキュメント, グラフといった表現形式でデータの保持を行う. NoSQL は, SQL を使用するデータベースには向いていない処理を行うことを目的にしている.

2.2　CAP 定理

分散データシステムにおいて次の 3 つを同時に保証することはできない

- 一貫性 (Consistency) 全てのノードはクエリが同じならば同じデータを返す.
- 可用性 (Availability) あるノードに障害が発生しても機能しているノードにより常にデータの読み書きが行える.
- 分断耐性 (Partition-tolerance) ネットワーク障害によりノードの接続が切れてもデータベースは機能し続けることができる.

これは CAP 定理 [1] と呼ばれる. 利用するデータベース選ぶ場合, この CAP 定理を意識しなければならない. 一貫性と可用性を重視したデータベースが, RDB である. 分断耐性を必要とする場合は NoSQL データベースとなる. そして NoSQL の場合, 分断耐性と後もう一つ, 一貫性か可用性のどちらを保証しているかで用途が変わってくる.

分散データシステムを考える場合は, この CAP 定理を意識していなければならない.

2.3 Cassandra

Cassandra[2] は 2008 年 7 月に Facebook によってオープンソースとして公開された Key-Value なデータベースである. Amazon の Dynamo[3] という分散キーバリューデータベースの影響を受けて作られている. スキーマレスな NoSQL データベースになる.

Cassandra はサーバノードの配置にコンシステント・ハッシングアルゴリズムを用いる. コンシステント・ハッシングによりノードは論理的にリング上に配置される. リングには数値で表される位置がある. データを書き込む際には, キーとなるハッシュ値に従いそのリングの位置から時計回りに近いサーバノードへと書き込まれる. コンシステント・ハッシングを用いることで, ノードの数が増減した場合に, 再配置をしなくてもよいという利点がある. データの偏りにより少数のサーバへの負荷が大きい場合に, 負荷が高いハッシュ値が指すリング上に新たなノードを追加することで負荷を下げるといった手段もとれる.

データを最大どれだけ配置するかを示すレプリケーションファクタと, データの読み書きをいくつのノードから行うのかを決めるコンシステンシーレベルを設定できる. コンシステンシーレベルには主に ONE, QUORAM, ALL がある. レプリケーションファクタの数値を N とした場合, ONE は 1 つのノード, QUORUM は $N/2 + 1$ のノード, ALL は N のノードへと読み書きを行う. コンシステンシーハッシング, レプリケーションファクタとコンシステンシーレベルの設定により Cassandra は高い可用性と分断耐性を持つ.

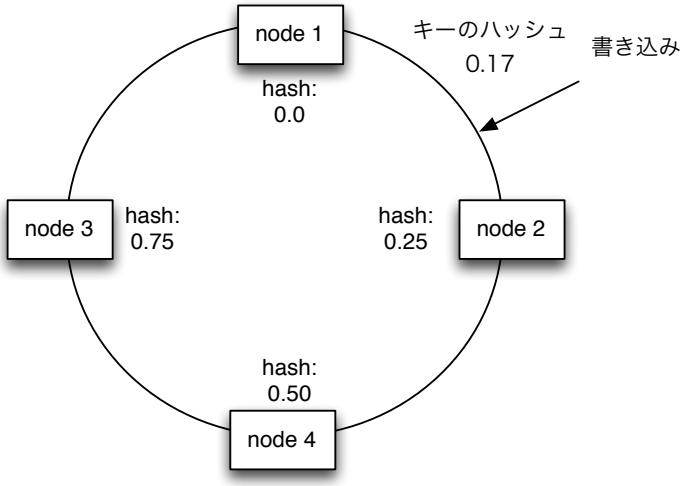

図 2.1: コンシステンシー・ハッシング

2.4 MongoDB

MongoDB は 2009 年に公開された NoSQL のデータベースである. JSON フォーマットのドキュメントデータベースであり, これはスキーマが無いリレーショナルテーブルに例えられる. スキーマが無いため, 事前にデータの定義を行う必要がない. そのためリレーショナルデータベースに比べてデータの追加・削除が行いやすい.

MongoDB は保存したデータを複数のサーバに複製をとる. これはレプリケーション (replication) と呼ばれる. また, 1 つのサーバが全てのデータを持つのでなく, ある範囲の値を別々のサーバに分割させて保持する. これをシャーディング (sharding) という. MongoDB はレプリケーションとシャーディングにより分断耐性と一貫性を持つ.

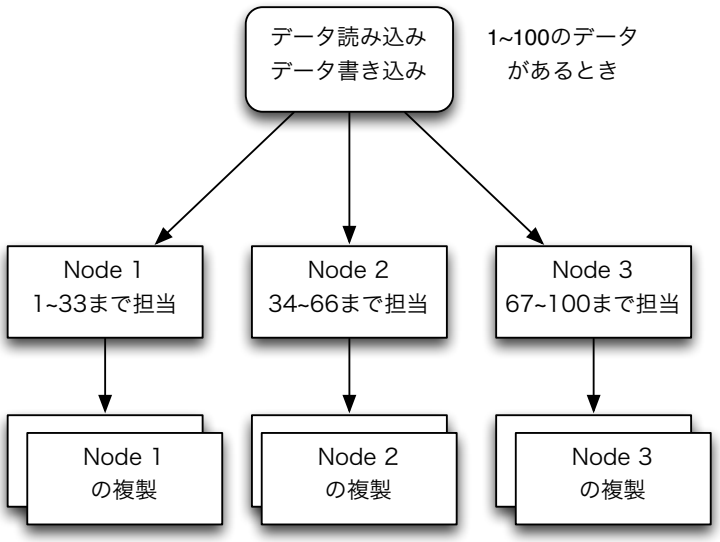

図 2.2: シャーディング

2.5 Neo4j

Neo4j は, グラフデータベースと呼ばれる NoSQL のデータベースである. データをグラフとして保存する. グラフはノードとリレーションシップにより表され, それぞれがプロパティを持つことができる. リレーションシップはグラフでいうところのエッジにあたる. ノードからリレーションシップを辿り, 各プロパティをみることでデータの取得を行うことができる. 通常データベースでは, データの取り出しに価の結合や条件の判定を行う. だが, グラフデータベースグラフはどれだけデータが大きくなろうがノードからノードへの移動は 1 ステップですむ. そのため, どれだけデータが大きくなろうと, データが小さい時と同じ計算量でデータの取得が行える.

Neo4j はマスターとスレーブの関係になるクラスタを構成することで分散データベースとして機能する. マスターに書かれたデータはスレーブに書き込まれるが, すぐに全てのスレーブに書き込まれるわけではない. したがってデータの整合性が失われる危険がある. スレーブサーバは現在保持しているデータを返すことができる. そのため Neo4j は高い読み取り性能の要求に答えることができる可用性と分断耐性を持つ.

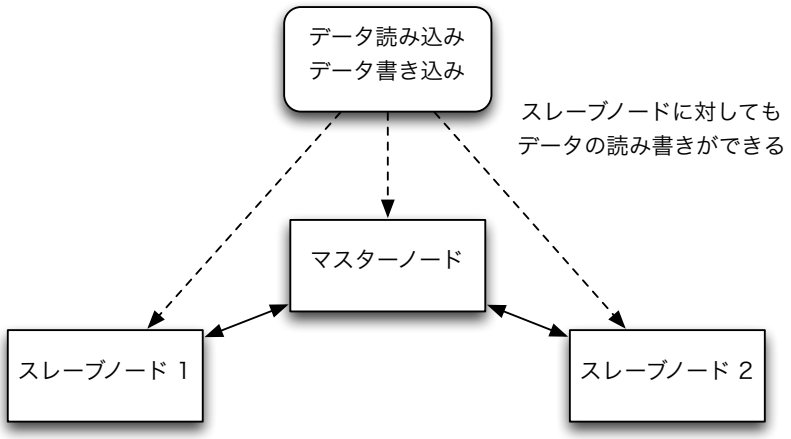

図 2.3: マスターとスレーブによるクラスタ

第3章　木構造データベースJungleの分散設計

Jungle はスケーラビリティのある CMS の開発を目指して当研究室で開発されている非破壊的木構造データベースである．一般的なコンテンツマネジメントシステムではブログツールや Wiki・SNS が多く, これらのウェブサイトの構造は大体が木構造であるため, データ構造として木構造を採用している．現在 Java と Haskell によりそれぞれ言語で開発されており本研究で扱うのは Java 版である．

本章ではまず破壊的木構造と, 非破壊的木構造の説明をし, Jungle におけるデータ分散の設計について述べる．

3.0.1　破壊的木構造

破壊的木構造の編集は, 木構造で保持しているデータを直接書き換えることで行う．図 3.1 は破壊的木構造の編集を表している．

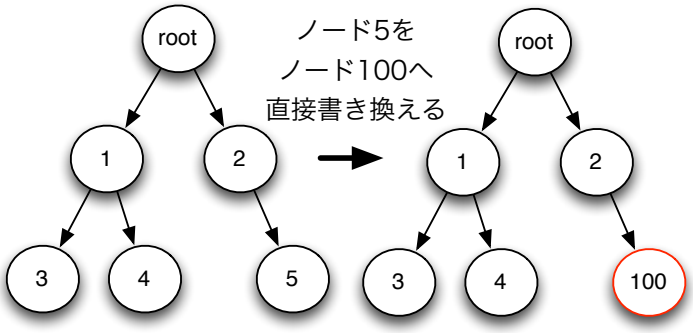

図 3.1: 破壊的木構造の編集

破壊的木構造は, 編集を行う際に木のロックを掛ける必要がある．この時, データを受け取ろうと木を走査するスレッドは書き換えの終了を待つ必要があり, 閲覧者がいる場合は木の走査が終わるまで書き換えをまたなければならない．これではロックによりスケーラビリティが損なわれてしまう．

3.0.2　非破壊的木構造

非破壊的木構造は破壊的木構造とは違い, 一度作成した木を破壊することはない. 非破壊的木構造においてデータの編集は, ルートから編集を行うノードまでコピーを行い新しく木構造を作成することで行われる. 図 3.2 は非破壊的木構造のデータ編集を示している.

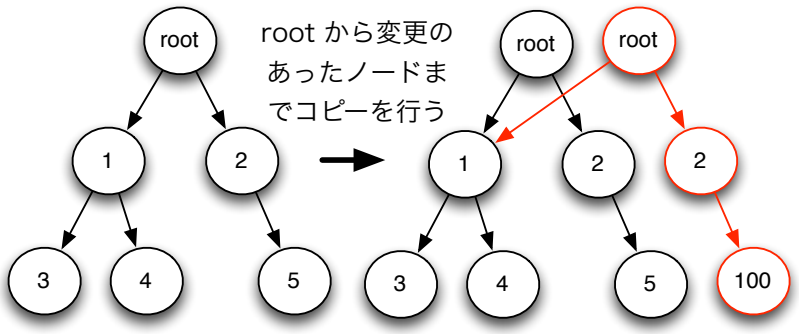

図 3.2: 非破壊的木構造の編集

非破壊的木構造におけるデータ編集の手順を以下に示す.

1. ルートから編集を行うノードまでのパスを調べる (図 3.3).
2. 編集を行うノードのコピーをとる. コピーをとったノードへデータの編集を行う (図 3.4).
3. 調べたパスに従いルートからコピーしたノードまでの間のノードのコピーをとり繋げる (図 3.5).
4. コピーしたルートノードは編集を行っていないノードへの参照を貼り新しい木構造を作る (図 3.6).

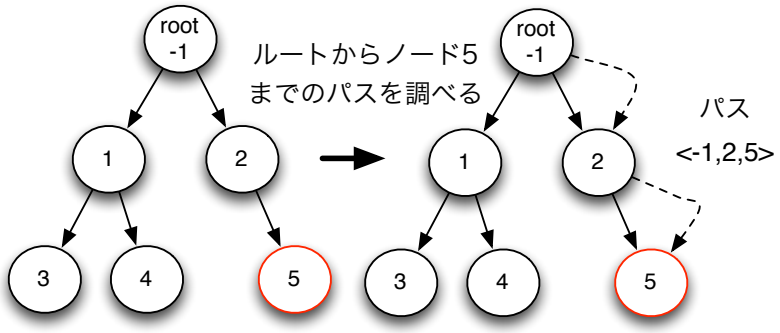

図 3.3: 非破壊的木構造の編集手順 1

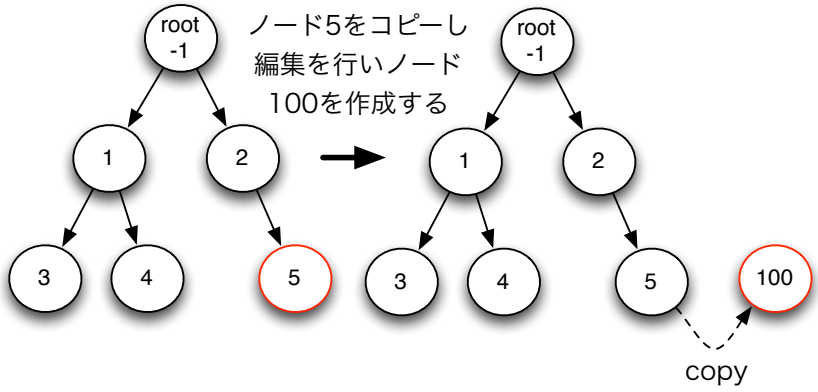

図 3.4: 非破壊的木構造の編集手順 2

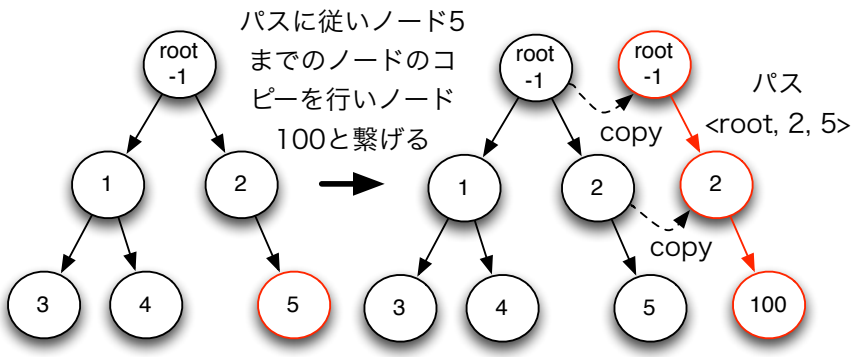

図 3.5: 非破壊的木構造の編集手順 3

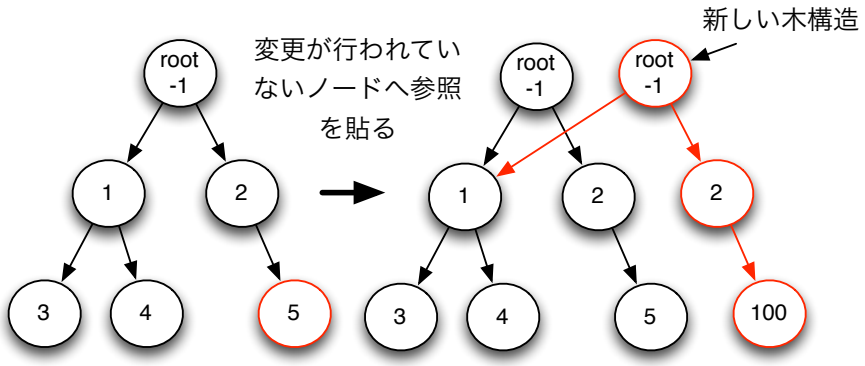

図 3.6: 非破壊的木構造の編集手順 4

非破壊的木構造においてデータのロックが必要となる部分は, 木のコピーを作終えた後にルートノードを更新するときだけである. データ編集を行っている間ロックが必要な破壊的木構造に比べ, 編集中においてもデータの読み込みが可能である (図 3.7). そのため, 破壊的木構造に比べスケールがしやすくなっている.

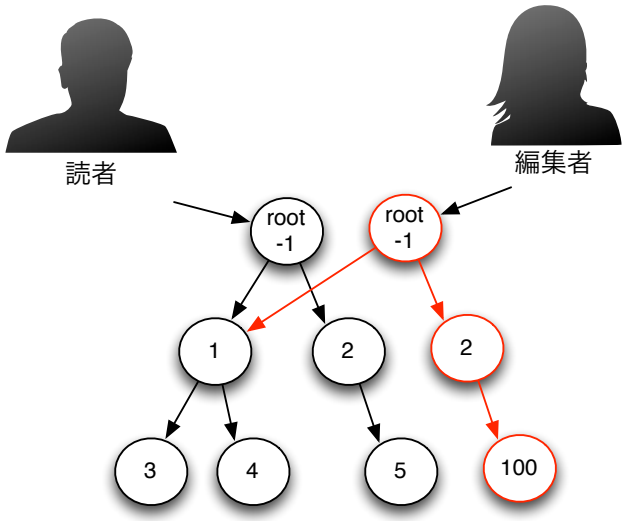

図 3.7: 非破壊的木構造による利点

3.1　Jungle におけるデータへのアクセス

Jungle ではデータをそれぞれの Node が attribute として保持する. attribute は String 型の Key と ByteBuffer の value のペアにより表される. Jungle でデータへのアクセス

は, この Node へのアクセスをさす. Node へのアクセスは, 木の名前と Node を指すパスにより行える. このパスは NodePath と呼ばれる (図 3.8).

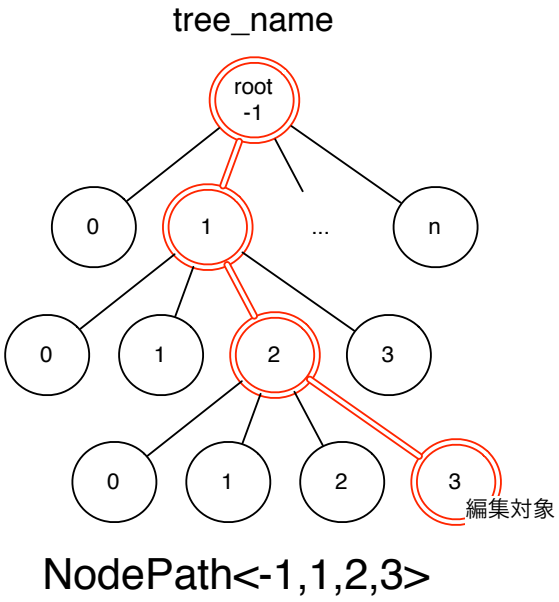

図 3.8: Node の attribute と NodePath

3.2　Jungle におけるデータ編集

3.2.1　NodeOperation

Jungle による最小のデータ編集は Node の編集を指す. Node 編集のために API が用意されており, この API は NodeOperation と呼ばれる. NodeOperation には次の 4 つの API が用意されている.

- `addNewChild(NodePath _path, int _pos)` NodePath で指定された Node に子供となる Node を追加する API である. pos で指定された番号に子供として追加を行う.
- `deleteChildAt(NodePath _path, int _pos)` NodePath と pos により指定される Node を削除する API である.
- `putAttribute(NodePath _path, String _key, ByteBuffer _value)` Node に attribute を追加する API である. NodePath は attribute を追加する Node を指す.
- `deleteAttribute(NodePath _path, String _key)` _key が示す attribute の削除を行う API である. NodePath は Node を示す.

NodeOperation はあくまで最小のデータ編集の単位である．アプリケーションレベルの実装にもよるが, Jungle によるデータの編集は NodeOperation が複数集まった単位によって行われる．この複数の NodeOperation の集まりを TreeOperationLog という．

3.2.2　TreeOperationLog

Jungle 内部では NodeOperation は順次ログに積まれていき, 最終的に commit されることで編集が完了する．この時, ログに積まれた複数の NodeOperation は TreeOperationLog として扱われる．以下に TreeOperationLog の具体的な例を示す (3.1).

Listing 3.1: トポロジーマネージャーの利用

```
1 [APPEND_CHILD:<−1>:pos:0]
2 [PUT_ATTRIBUTE:<−1,0>:key:author,value:oshiro]
3 [PUT_ATTRIBUTE:<−1,0>:key:mes,value:hello]
4 [PUT_ATTRIBUTE:<−1,0>:key:timestamp,value:0]
```

このログは今回の研究で使用したベンチマーク用掲示板プログラムにおける書き込みにより行われるログである (図 3.9).

大文字の英字は実行した NodeOperation の種類を表す．<> により囲まれている数字は NodePath を示す．NodePath の表記以降は Node の position や attribute の情報を表している．

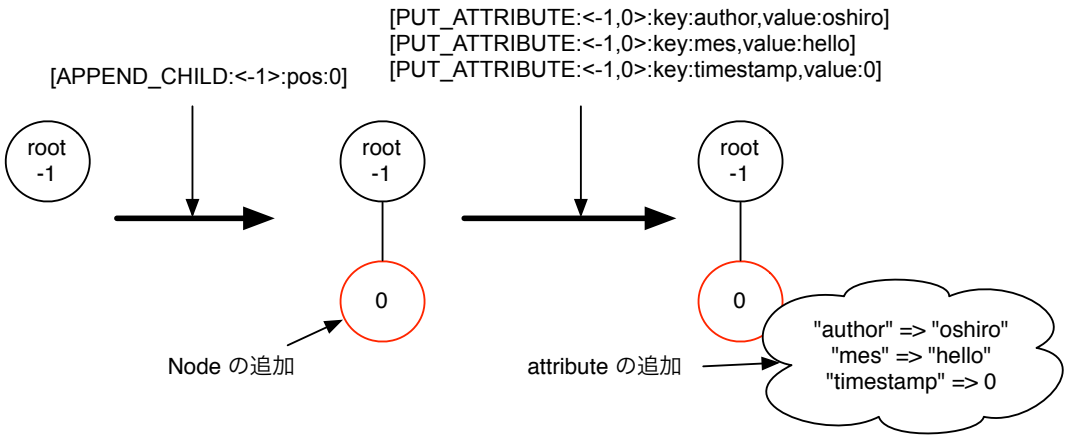

図 3.9: TreeOperationLog の具体例

図 3.9 の説明を行う．まず, `APPEND_CHILD` により Root Node の 0 番目の子供となる Node の追加を行う．次に, 追加を行った Node に対して `PUT_ATTRIBUTE` により attribute の情報を持たせていく．attribute の内容に作者の情報を表す auther, メッセージの内容を表す mes, そしてタイムスタンプを timestamp とそれぞれキーにすることで追加される．

以上が掲示板プログラムにおける 1 つの書き込みで発生する TreeOperationLog である．

3.3　分散バージョン管理システムによるデータの分散

Jungle は Git や Mercurial といった分散バージョン管理システムの機能を参考に作られている. 分散バージョン管理システムとは, 多人数によるソフトウェア開発において変更履歴を管理するシステムである. 分散管理システムでは開発者それぞれがローカルにリポジトリのクローンを持ち, 開発はこのリポジトリを通すことで進められる (図 3.10). ローカルのリポジトリは独立に損刺し, サーバ上にあるリポジトリや他人のリポジトリで行われた変更履歴を取り込みアップデートにかけることができる. また逆に, ローカルのリポジトリに開発者自身がかけたアップデートを他のリポジトリへと反映させることもできる. 分散管理システムでは, どれかリポジトリが壊れたとしても, 別のリポジトリからクローンを行うことができる. ネットワークに障害が発生しても, ローカルにある編集履歴をネットワーク復旧後に伝えることができる. そのため, 可用性と分断耐性が高いと言える.

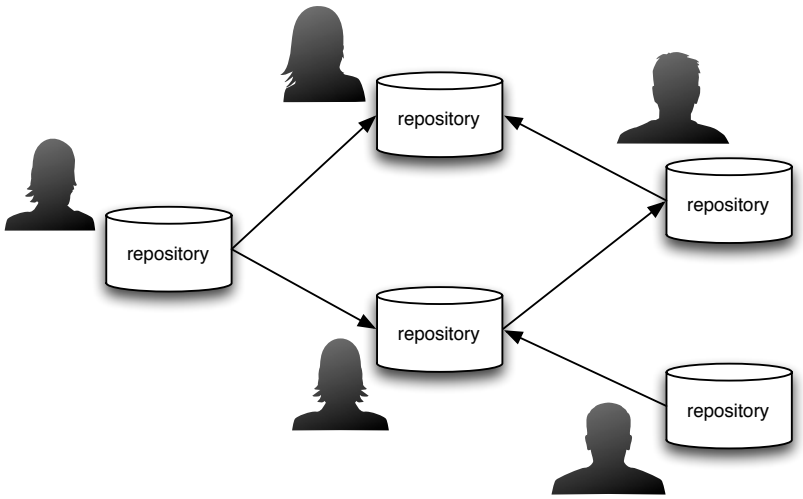

図 3.10: 分散バージョン管理システム

3.3.1　マージによるデータ変更衝突の解決

分散管理システムでは, データの更新時において衝突が発生する時がある. それは, 分散管理システムを参考にしている Jungle においても起こる問題である. データの変更を行うときには, 元のデータに編集が加えられている状態かもしれない. Jungle はリクエストがきた場合, 現在もっているデータを返す. そのためデータは最新のものであるかは保証されない. この場合, 古いデータに編集が加えられ, それを更に最新のデータへ伝搬させなければならない. このように他のリポジトリにより先にデータ編集が行われており, データの伝搬が素直にできない状態を衝突という. この衝突を解決する手段が必要である. 分散管理システムでは衝突に対してマージと呼ばれる作業で解決をはかる. マージは, 相手

のリポジトリのデータ編集履歴を受け取り, ローカルにあるリポジトリの編集と合わせる作業である. データ衝突に対して Jungle はアプリケーションレベルでのマージを実装して貰うことで解決をはかる.

以下にマージが必要な場合とそうでない場合のデータ編集についての図を示す (図3.11,3.12,3.13).

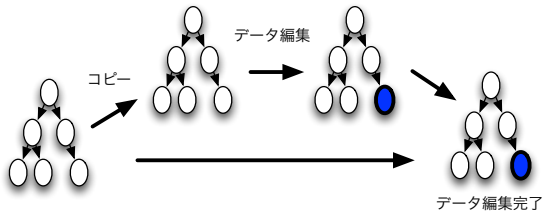

図 3.11: 衝突の発生しないデータ編集

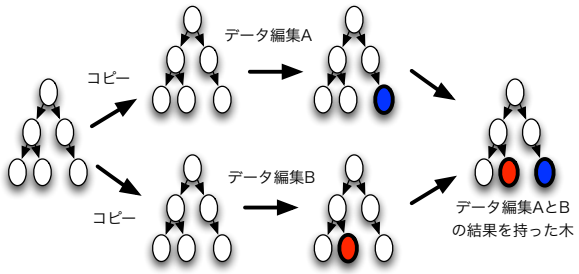

図 3.12: 自然に衝突を解決できるデータ編集

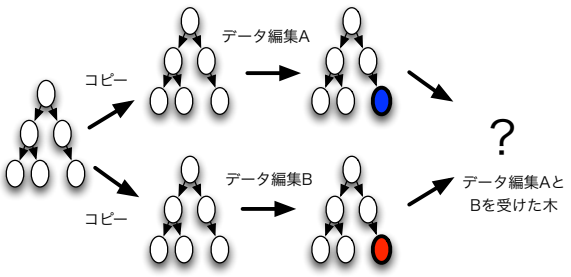

図 3.13: 衝突が発生するデータ編集

3.4　ネットワークトポロジーの形成

分散管理システムを参考に Jungle でもそれぞれのデータベースが独立に動くようにしたい. そのために必要なことはトポロジーの形成と, サーバノード間でのデータアクセス機構である. また, データ分散のために形成したトポロジー上で扱うデータを決めなければならない.

3.4.1　ツリートポロジーの形成

分散データーベス Jungle で形成されるネットワークトポロジーはツリー構造を想定している. ツリー構造ならば, データの整合性をとる場合, 一度トップまでデータを伝搬させることで行える. トップもしくはトップまでの間にあるサーバノードでデータ伝搬中に衝突が発生したらマージを行い, マージの結果を改めて伝搬すればよいからである. また, リング型, スター型, メッシュ側ではデータ編集の結果を他サーバノードに流すとき流したデータが自分自身にくることにより発生するループに気をつける必要がある. ツリー構造の場合は, サーバノード同士の繋がりで閉路が無い. そのため, 自分自身が行ったデータ編集の履歴を繋がっているノードに送信するだけですむ. このルーティングの方式はスプリットホライズンと呼ばれるものである.

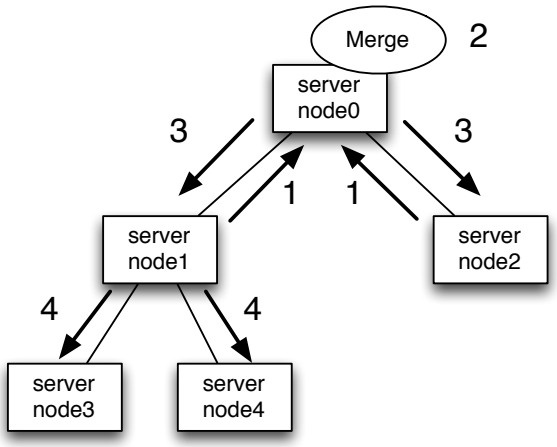

図 3.14: ツリー型の Network Topology

3.4.2　トポロジーの形成手段

Jungle で使用するネットワークトポロジーはツリー型を考えているが, リング型やメッシュ型といった他のネットワークトポロジーによる実装に関しても試す余地はある. そのため, ツリーだけでなく, 自由にネットワークトポロジーの形成を行えるようにしたい.

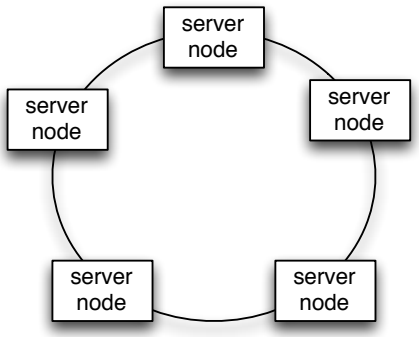

図 3.15: リング型のトポロジー

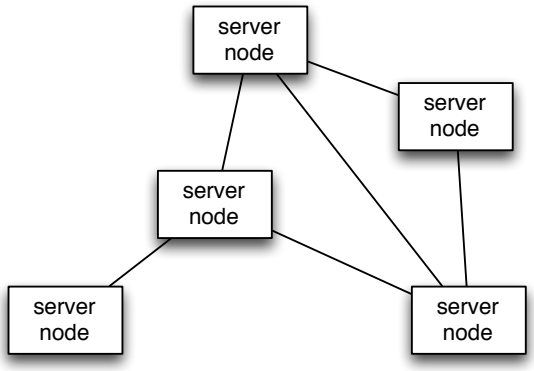

図 3.16: メッシュ型のトポロジー

そこで当研究室で開発を行っている並列分散フレームワークである Alice を使用する. Alice はユーザが望んだマシンへの接続や必要なデータへのアクセスを行う機構と, 接続トポロジー形成機能を提供している.

3.5　並列分散フレームワーク Alice

Alice は当研究室で開発している並列分散フレームワークである. Alice はデータを DataSegment, タスクを CodeSegment という単位で扱うプログラミングを提供している. コードの部分となる CodeSegment は, 計算に必要なデータである DataSegment が揃い次第実行が行われる (図3.17). CodeSegment の結果により出力される新たなデータでは, 別の CodeSegment が実行されるための DataSegment となる. DataSegment と CodeSegment の組み合わせにより並列・分散プログラミングの依存関係が表される.

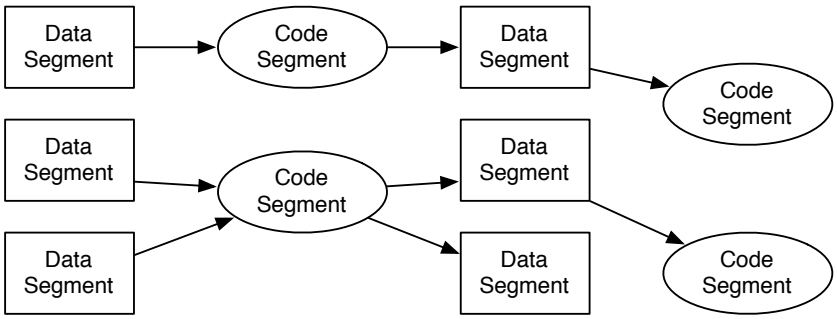

図 3.17: DataSegment と CodeSegment によるプログラムの流れ

3.5.1　MessagePack によるシリアライズ

Alice では DataSegment のデータ表現に MessagePack(http://msgpack.org) を利用している. MessagePack はオブジェクトをバイナリへと変換させるシリアライズライブラリである. Alice によりネットワークを介してデータにアクセスするときは, そのデータが MessagePack でシリアライズが行えることが条件である.

3.6　Jungle のデータ分散

Alice によりトポロジーの形成とデータアクセスの機構が提供された. 後はデータ分散の為にどのデータをネットワークに流すのか決めなければならない. そこで選ばれたのが TreeOperationLog である. TreeOperationLog はデータ編集の履歴になる. どの Node にどのような操作をしたのかという情報が入っている. この TreeOperationLog を Alice を使って他サーバノードに送り, データの編集をしてもらうことで同じデータを持つことが可能となる. Alice を用いるため, この TreeOperationLog は MessagePack によりシリアライズ可能な形にすることが必要である.

3.6.1　CAP 定理と Jungle

ここまでの Jungle の設計を踏まえて, CAP 定理における Jungle の立ち位置を考える. 分散管理バージョンのように独立したリポジトリもち, それぞれが独自の変更を加えることが行えることで一貫性はゆるい. だが, ネットワークから切断されてもローカルで行ったデータの変更をネットワーク復旧後で伝搬できることと, リクエストに対し持っているデータをすぐに返すことができる. つまり Jungle は可用性と分断耐性に優れたデータベースを目指している. 第 2 章で紹介した既存のデータベースと Jungle との CAP 定理の関係を図 3.18 に示す.

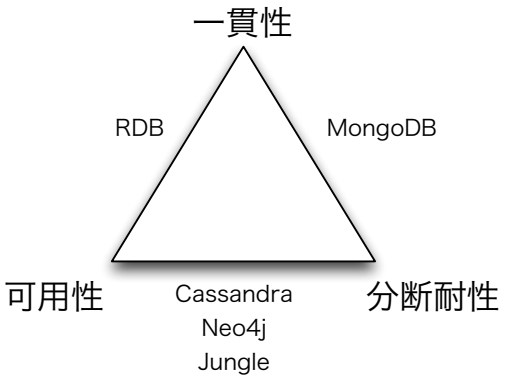

図 3.18: CAP 定理における各データベースの立ち位置

3.7　ログによるデータの永続性

Jungle は非破壊でさらにオンメモリにデータを保持するため, 使用するメモリの容量が大きくなる. そのため, ハードディスクに書き出し, 一定の区切りで保持している過去のデータをメモリ上から掃除しなければならない. そこで, ログによるデータの永続性の実装を行う. ここでログをどのようなデータ表現でハードディスクへと書きだすかという問題が発生するが, これは Alice を使うことで解決している. Alice を用いるため MessagePack によりシリアライズ可能な TreeOperationLog ができる. このシリアライズ可能な TreeOperationLog をそのままハードディスクへ書き込むこととでログの永続性ができる.

第4章　Jungle の分散実装

本章では Jungle に行った分散実装について述べる. 前章では Jungle のアーキテクチャと分散設計について説明した. トポロジーの形成と他サーバノードのデータのアクセス方法には Alice を使用する. また, Jungle ではデータ編集のログとして TreeOperationLog がある. この TreeOperationLog を Alice により他サーバノードへ送ることでデータの分散を行う.

4.1　Alice のトポロジーマネージャーの利用

4.1.1　トポロジーマネージャーの起動

Alice を用いてサーバノードでトポロジーの形成を行う方法を述べる. Alice のトポロジーマネージャーの起動は 4.2 の様に行う. (4.1).

Listing 4.1: Alice によるネットワークトポロジーマネージャーの起動

```
% java −cp Alice.jar alice.topology.manager.TopologyManager −p 10000 −conf ./topology/tree5.dot
```

-p オプションはトポロジーマネージャーが開くポートの番号, -conf オプションには dot ファイルのパスを渡す.

ポート番号は Alice により記述された並列分散プログラムの起動時に渡す必要がある. dot ファイルには, トポロジーをどのように形成するかが書かれている. 以下に, サーバノード数 5 で, 2 分木ツリー構造を形成する dot ファイルの例を示す (4.2).

Listing 4.2: ネットワークトポロジー設定用 dot ファイル

```
% cat tree5.dot
digraph test {
   node0 −> node1 [label="child1"]
   node0 −> node2 [label="child2"]
   node1 −> node0 [label="parent"]
   node1 −> node3 [label="child1"]
   node1 −> node4 [label="child2"]
   node2 −> node0 [label="parent"]
   node3 −> node1 [label="parent"]
   node4 −> node1 [label="parent"]
}
```

node0 や node1 はサーバノードの名前を示す. サーバノードの間にはラベルがあり, Alice 上ではこのラベルに指定される文字列 (キー) を使うことで他のサーバノードのデータへアクセスすることができる. node0 -¿ node1 はサーバノード同士の繋がりを示してい

る. 次に続く label=”child1” は, node0 が node1 のデータに”child1”という文字列を使うことでアクセスできることを示す.

dot ファイルを読み込んだ Alice のトポロジーマネージャーに対して, サーバノードは誰に接続を行えばよいかを訪ねる. トポロジーマネージャーは訪ねてきたサーバノードに対してノード番号を割り振り, dot ファイルに記述している通りにサーバノード同士が接続を行うよう指示をだす.

トポロジーマネージャーは接続要求先を聞いてくるサーバノードに対して名前を割り振り, 接続相手を伝える. dot ファイル 4.2 により形成されるトポロジーを図 4.1 に示す.

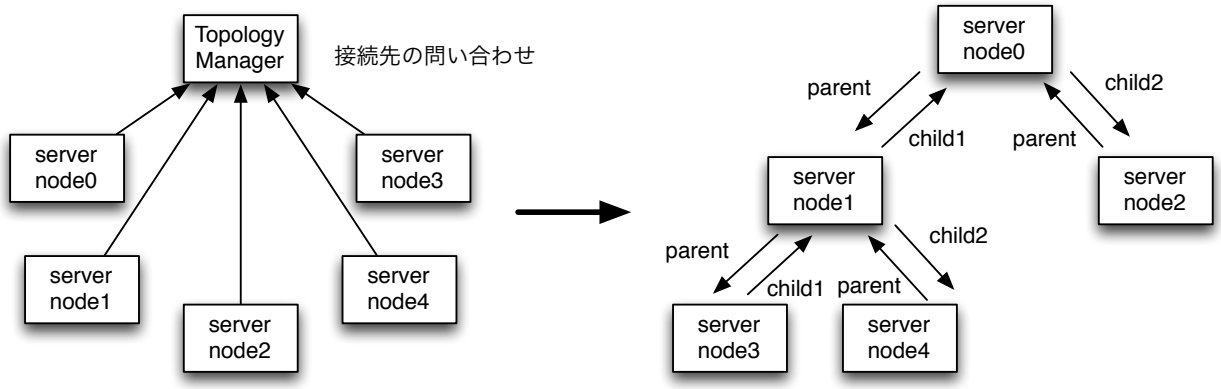

図 4.1: Alice によるネットワークトポロジー形成

矢印に書かれている文字列は, 相手のデータにアクセスするキーを示す. ”child1”, ”child2”, ”parent” というキーを使うことで別のサーバノードにあるデータを取得することができる. これでトポロジーマネージャーが起動される.

4.1.2 アプリケーション側の記述

次は Jungle 側のプログラムが最初に Alice のトポロジーノードと通信を行うようにする. そのためには Alice の TopologyNode クラスに必要な情報を渡してインスタンスを生成する (4.3).

Listing 4.3: アプリケーションの起動

```
1 public static void main( String[] args ) throws Exception
2 {
3   RemoteConfig conf = new RemoteConfig(args);
4   new TopologyNode(conf, new StartJungleCodeSegment(args, conf.bbsPort));
5 }
```

TopologyNode クラスは第 2 引数として CodeSegment を受け取る. TopologyNode のインスタンスはまず初めにトポロジーマネージャーへ接続を行う. 次にトポロジーマネージャーから受け取った情報を元に別のサーバノードとトポロジーの形成を行う. その後,

第 2 引数で渡された StartJungleCodeSegment の実行を行う．StartJungleCodeSegment には通常のアプリケーションの処理が書かれる．

アプリケーションの起動時にはコンフィグの情報として，トポロジーマネージャーが動いているサーバのドメインとポート番号を渡す必要がある． 例えば，mass00.cs.ie.u-ryukyu.ac.jp というサーバ上でポート番号 10000 を指定してトポロジーマネージャーを起動した場合は次のようになる (4.4).

Listing 4.4: トポロジーマネージャーの利用

```
% java Program −host mass00.cs.ie.u−ryukyu.ac.jp −port 10000
```

4.2 Alice を用いての分散実装

Alice のポロジー形成と他のサーバのデータへのアクセスする機構を用いるためには，Alice が提供するプログラミングスタイルに沿わなければならない．それは DataSegment(データ) と CodeSegment(タスク) によるプログラムである． ここではまず DataSegment と CodeSegment によるプログラムの方法について説明し，他サーバとの通信部分の実装について述べる．

4.2.1 Alice によるプログラミング

Alice は DataSegment（データ）と CodeSegment（タスク）単位でプログラミングを行うことを述べた．CodeSegment には計算に必要な DataSegment が登録される．そして DataSegment が準備され次第 CodeSegment による計算が実行される．DataSegment の取得は文字列のキーを使うことで行える．以下のコードに CodeSegment の例を示す．

Listing 4.5: CodeSegment の実行

```
1  public class TestCodeSegment extends CodeSegment {
2    public Receiver arg1 = ids.create(CommandType.TAKE);
3
4    public TestCodeSegment() { }
5
6    public void run() {
7      int count = ds.asInteger();
8      count++;
9      System.out.println("count␣=␣"+count);
10     if(c > 10) { exit(0); }
11     CodeSegment cs = new TestCodeSegment();
12     cs.setKey("count");
13     ods.update("local", "count", c);
14   }
15
16   public static void main(String[] args) {
17     CodeSegment cs = new TestCodeSegment();
18     cs.arg1.setKey("local", "count"); // setKey API
19     cs.ods.update("local", "count", 0);
20   }
21 }
```

これは, 数字を 1 から 10 まで出力を行い終了するプログラムである. コードの説明を行う. 17 行目から 19 行目の処理が最初に行われる. まず TestCodeSegment という CodeSegment のインスタンス cs を生成する. cs は arg1 という Receiver クラスのフィールドを保持しており, Receiver クラスは DataSegment を受けとるためのクラスである. arg1 に対し setKey API を使うことで, 使用したい DataSegment のキー”count”を登録することができる. これによりキー”count”に対してデータが登録された場合, そのデータを受け取り cs の計算が自動で始まる. setKey API の第一引数に渡している”local”はどのマシンの DataSegment にアクセスするのかを指定している. この場合は自分自身を表す”local”になる.

データの登録は `ods.update` により行える. 上記のコード 19 行目では update により”count”をキーとして数値の 0 を登録している. update がされると cs の計算が始まり別スレッドにより 8 行目からの処理が行われる.

update によりキー”count”に登録された数値 0 は Receiver である ds を使って取ることができる. 7 行目から 13 行目では `ds.asInteger()` により, ”count”に登録したデータの中身を受け取りインクリメントし出力する. そして最後には `ods.update` を行っている. 新たな TestCodeSegment も生成しており, これはインクリメントされた”count”が update されることで実行される. この一連の処理を”count”の数値が 10 以上になるまで行う.

DataSegment へデータの追加と CodeSegment の実行について表した図 4.2 になる.

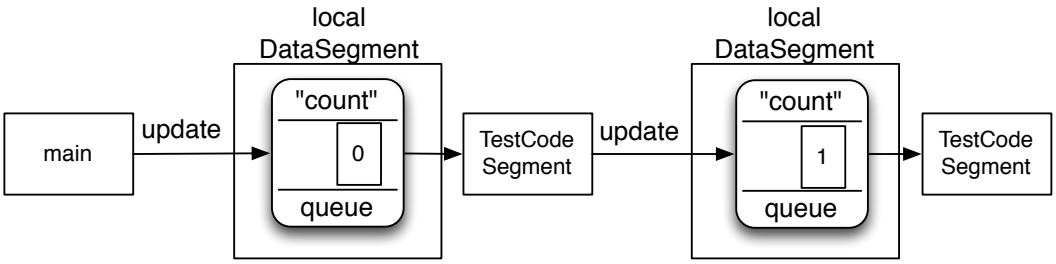

図 4.2: DataSegment と CodeSegment によるプログラムの例

4.2.2 他サーバノードの DataSegment へアクセス

Alice における基本的なプログラミングは述べた. 次はネットワークを介して他サーバノードの DataSegment にアクセスするプログラムについて述べる.

まず, Alice により 2 分木 3 ノードのトポロジーが形成された場合を想定する. その時に実際に作られるトポロジーを図 4.3 に示す.

ネットワークを介した DataSegment へのアクセスはそのサーバノードを示す文字列のキーを追加することで行える. 他サーバノードを示す文字列のキーとは図 4.3 に矢印の隣に書かれている文字列”parent”, ”child1”, ”child2” のことを指す. 例えば, server node0 が server node1 の DataSegment に入っている”count”というデータをを使用したい場合は, 次のように setKey を行えばよい（4.6）.

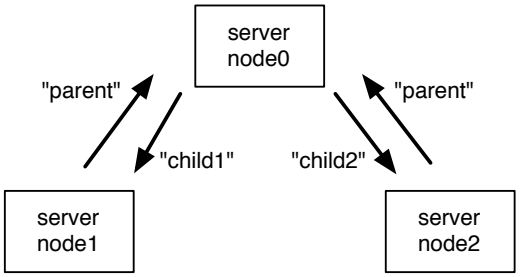

図 4.3: トポロジーの形成

Listing 4.6: CodeSegment で他サーバノードの DataSegment を使用する

```
1 CodeSegment cs = new RemoteCodeSegment();
2 cs.arg1.setKey("child1", "count");
```

また, 他サーバノードの DataSegment にデータを送りたい場合は, put を行うときにサーバノードへのキーを追加すればよい. 例として, server node1 や server node2 が server node0 の DataSegment に”message” というキーでデータを追加したい場合次のようになる（4.7）.

Listing 4.7: 他サーバーノードの DatasSegment にデータを追加する

```
1 ods.put("parent", "message", "Hello␣parent");
```

4.2.3 独自クラスのインスタンスの送受信

最後に, 独自クラスのインスタンスの DataSegment での扱い方について述べる. Alice では MessagePack を用いてシリアライズを行い他サーバノードへと送信している. MessagePack はクラス単位でシリアライズを行うことができる. そのため, Alice ではプリミティブな型に限らずクラスのインスタンスを DataSegment として扱うことができる.

MessagePack によりシリアライズとなるクラスはいくつか制限がある. それはそのクラスに@Message アノテーションを付けることと, そのクラスが保持するフィールドが MessagePack によりシリアライズ可能であることである. 例えば次のようなクラスである.

Listing 4.8: MessagePack によりシリアライズ可能なクラス 1

```
1 import org.msgpack.annotation.Message
2
3 @Message
4 public class Student {
5    String name;
6    int age;
7 }
```

上記の Studen クラスはプリミティブ型しか保持していない. そのためシリアライズが可能であるまた, 次のようなクラスもシリアライズ可能な型となる.

Listing 4.9: MessagePack によりシリアライズ可能なクラス 2

```
1 import org.msgpack.annotation.Message
2
3 @Message
4 public class Class {
5   List<Student> studentList;
6 }
```

この場合, フィールドはプリミティブな型でない Student クラスのフィールドを保持している. しかし, Student クラスはシリアライズ可能な形で作成しているため, クラスのフィールドとして保持しても問題はない.

これらの制約にそった形で作成し DataSegment にネットワークを介してクラスのインスタンスを update することができる. DataSegment から受け取ったデータはそのままではシリアライズされたものため, 一度手元で元のクラスにコンバートすることで扱う. 例として, Alice における Studen クラス (Listing4.8) のコンバートを次に示す.

Listing 4.10: DataSegment

```
1 // public Receiver arg1 = ids.create(CommandType.PEEK);
2 Student s = arg1.asClass(Student.class);
```

MessagePack でシリアライズ可能な形としているため DataSegment はネットワークを介して送受信が可能である.

4.3 Alice を用いた Jungle の分散実装

ここまで Alice を用いたプログラミングの方法について述べた.

4.3.1 ログのシリアライズ

ここでログのシリアライズについて述べる.

シリアライズとは, データをネットワーク上に流しても良い形式に変換することである.

4.3.2

4.4 掲示板プログラムにおけるマージの実装

Jungle に分散実装を行った後の問題としてデータ衝突がある. 他のサーバノードから送られてくるデータが既に手元で変更を加えた木構造を対象とした場合に発生する問題である. Jungle ではこれをアプリケーション毎にマージを実装することで解決させる.

今回分散実装を行い, 例題として掲示板プログラムを用意した. 掲示板プログラムに実装を行ったマージについて述べる. まず Jungle を用いた掲示板プログラムのデータ保持方法を図 4.4 に示す.

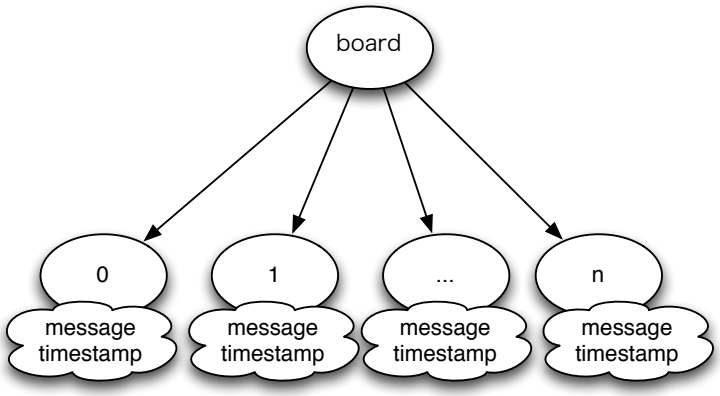

図 4.4: Jungle による掲示板プログラムのデータ保持方法

掲示板プログラムでは各掲示板毎に 1 つの木構造が作成される. 掲示板への 1 つの書き込みは子ノードを 1 つ追加することに相当する. また, 各子ノードは attributes として書き込みの内容である message と書き込まれた時間を表す timestamp を保持している. 先に追加された順で子ノードには若い番号が割り振られる.

他サーバノードからの書き込みをそのまま子ノードの後ろに追加してしまうと, データの整合性が崩れてしまう. この時の状態を表しているのが図 4.5 と 4.6 になる.

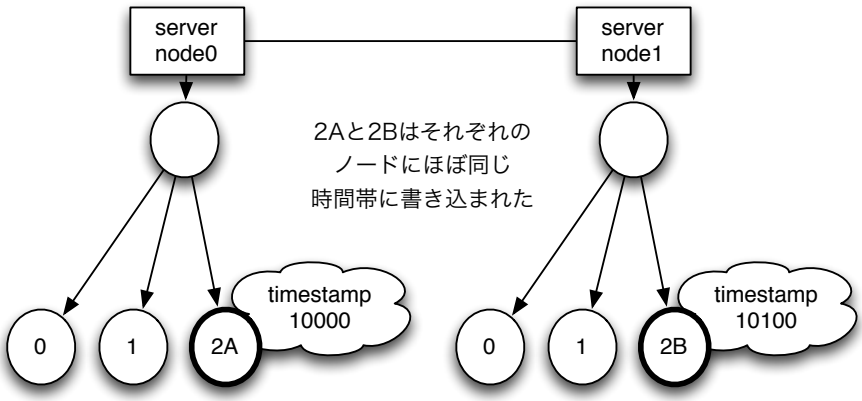

図 4.5: 他サーバノードの編集データ反映による整合性の崩れ 1

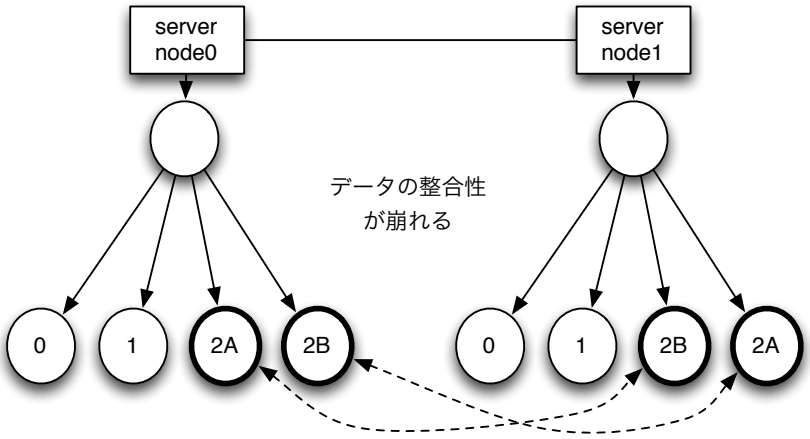

図 4.6: 他サーバノードの編集データ反映による整合性の崩れ 2

図 4.6 の server node0 の木の状態にするのが理想である. 掲示板のへの書き込みの表示は, 書き込みされた時間が早い順に表示されるようにしたい. これを timestamp を利用することで行う. 他サーバノードから来たデータに関しては, timestamp を参照し, 次に自分の保持している木の子ノードの timestamp と比べていくことでデータの追加する場所を決める. これが今回実装を行った掲示板システムにおけるマージになる.

第5章　分散木構造データーベース Jungle の評価

前章では Jungle における分散データベースの詳細な実装について述べた. 本章では実装を行った Jungle に対して Cassandra との性能比較を行い評価をする. 性能比較の為に簡易な掲示板プログラムを Jungle と Cassandra それぞれに作成した. 複数のノードに繋がっている状態においても性能を測りたいため, 学科提供する VMWare の並列環境を利用する. また, 我々の研究室が利用しているブレードサーバ上で動いている KVM もノードとして利用する.

5.1　実験方法

実験は同じ機能を提供している簡易掲示板プログラムを Jungle と Cassandra それぞれで動かし, HTTP リクエストにより負荷をかけて行う. レスポンスが帰ってくるまでの時間をはかる.

また, 実験は 2 つ行う. まず行う実験は, 複数のノードで起動してるうちの 1 つのノードに負荷をかける方法である. これはノードの数に比例してレスポンスが遅くなっていないか確かめるためである.

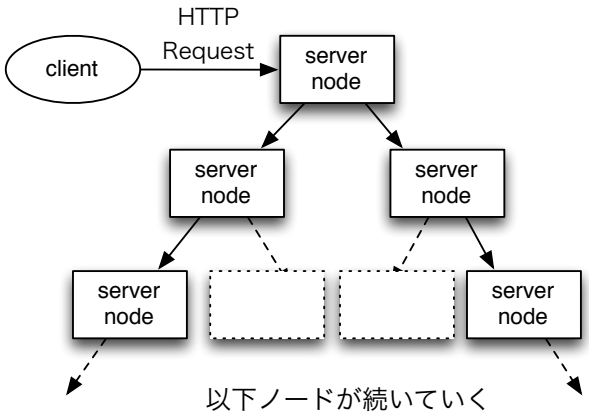

図 5.1: 複数起動中の Jungle の 1 ノードへの負荷

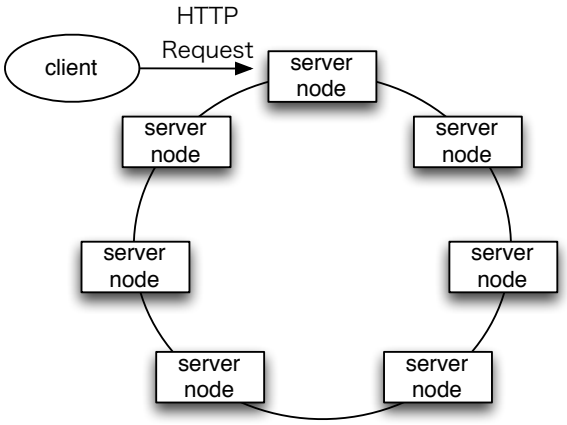

図 5.2: 複数起動中の Cassandra の 1 ノードへの負荷

次に行う実験は複数のノードに対し複数のクライアントから負荷をかける方法である. それぞれ大量の HTTP リクエストをだし, 全てのリクエストの処理にかかる時間を測定する.

クライアントの数に比例してノードを増やすことでレスポンスを維持できるかスケーラビリティを調べるためである.

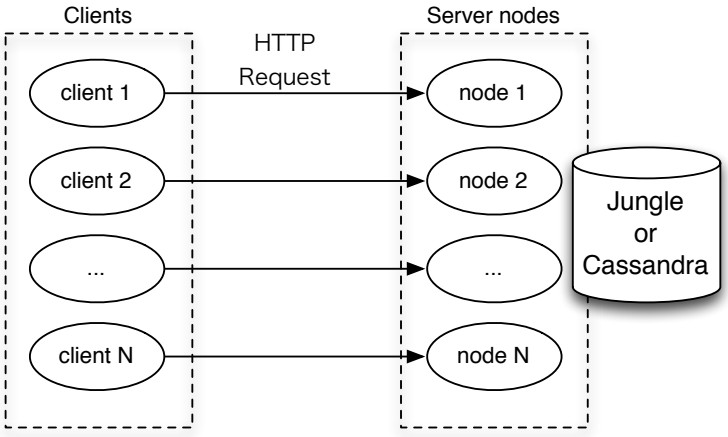

図 5.3: 複数のクライアントから複数のノードへの負荷

5.1.1　weighttp

最初の実験で 1 つのノードに負荷をかけるプログラムはウェブサーバの測定ツールである weighttp を使用する. weighttp は総リクエスト数, 同時接続数, ネイティブスレッド数

をオプションとして指定することができる C 言語でかかれたプログラムである.

5.1.2　掲示板プログラム

今回使用する掲示板プログラムは組み込み用ウェブサーバである Jetty をフロントエンドとして利用し, バックエンドに Jungle と Cassandra を利用している.

5.1.3　実験環境

ノードを実行させるサーバの仕様

使用する VMWare と KVM のクラスタの使用を以下に示す. クラスタは仕様を表 5.1 と表 5.2 に示す.

表 5.1: ノードを実行させる VMWare クラスタの仕様

名前	概要
CPU	Intel(R) Xeon(R) CPU X5650@2.67GHz
Memory	8GB
OS	CentOS 5.8
HyperVisor	VMWare ESXi
JavaVM	Java(TM) SE Runtime Environment (build 1.7.0-b147)

表 5.2: ノードを実行させる KVM クラスタの仕様

名前	概要
CPU	Intel(R) Xeon(R) CPU X5650@2.67GHz
Memory	8GB
OS	CentOS 5.8
HyperVisor	KVM
JavaVM	Java(TM) SE Runtime Environment (build 1.7.0-b147)

1 台に負荷をかけるブレードサーバの仕様

最初の実験で負荷をかける側としてブレードサーバを使用する. ブレードサーバの仕様を表 5.3 に示す

表 5.3:

名前	概要
CPU	Intel(R) Xeon(R) CPU X5650@2.67GHz
物理コア数	12
論理コア数	24
Memory	132GB
OS	Fedora 16

サーバの環境

HTTP によりノードに負荷を掛ける場合気をつけることがある. それはサーバの設定により最大コネクション数や開くことのできるファイル記述子の数に制限がかかっていることである. この 2 つの値はデフォルトでは小さなものとなっており, そのままではカーネルの設定がネックとなったベンチマーク結果がでる可能性がある. そこで次のようにコマンドを実行することでコネクション数の制限を増やすことができる.

Listing 5.1: コネクション数を増やす

```
1 % sudo sysctl -w net.core.somaxconn=10000
```

ファイル記述子の制限を増やす場合は次のコマンドを実行する

Listing 5.2: ファイル記述子の制限を増やす

```
1 % ulimit -n 10000
```

5.2　実験結果 1

サーバノード数は 10 台から 50 台まで 10 台単位で weighttp により負荷をかけ測定した. weighttp に付けたオプションは以下のとおりである

Listing 5.3: weighttp のオプション

```
1 weighttp -n 1000000 -c 1000 -t 10 -k "http://url"
```

ネイティブスレッドを 10 個生成し, 同時接続は 1000 までで, 百万リクエストを送るオプションとなっている.

実験の結果を示す. 縦軸は全てのリクエストに対してレスポンスが返ってくるのにかかった時間 (秒), 横軸はサーバノード数を表す. Jungle と, Cassandra のコンシステンシー・レベルを QUORUM, ONE と両方の結果を測定した. Cassandra のレプリケーションは 5 である.

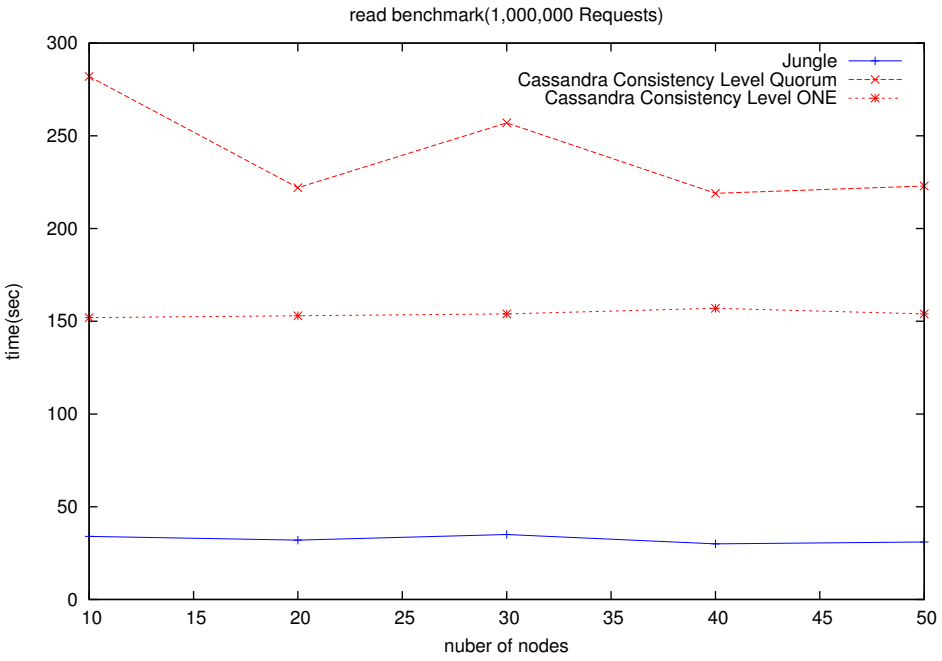

図 5.4: 読み込みベンチマーク結果

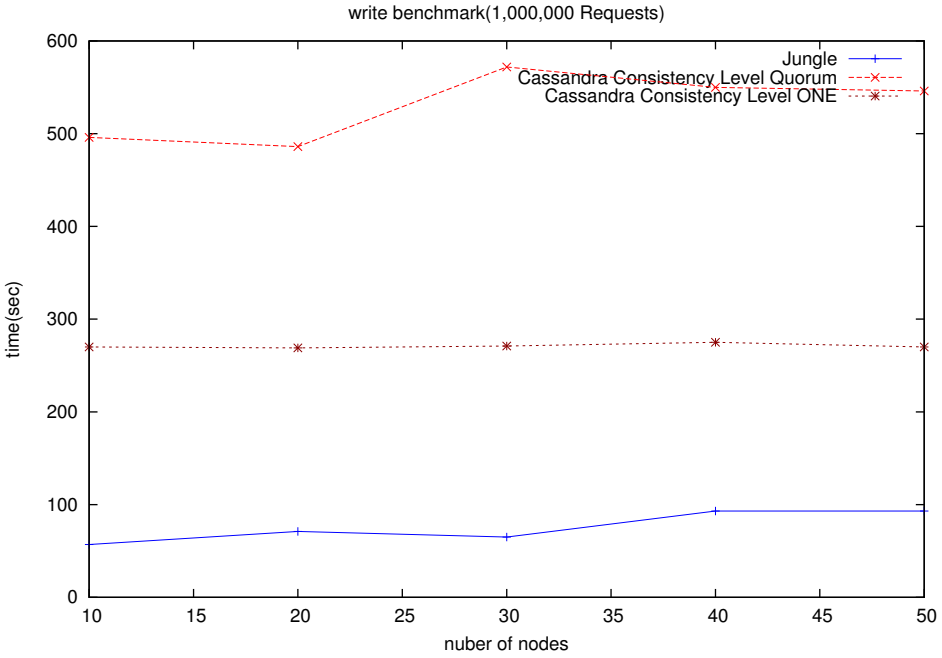

図 5.5: 書き込みベンチマーク結果

読み込み, 書き込み, どちらとも Jungle が 3 倍以上早くレスポンスを返していることが確認できる. また, Cassandra も Jungle もノードの数が増えてもレスポンスを返す時間が遅くならないことも分かる.

5.3　実験結果 2

学科の並列環境クラスタを用いて分散環境下での実験を行う学科の提供する VM は 48 台だが, ブレードサーバ上で動く KVM から 12 台を利用し, 合計 60 台を使用する. Jungle と Cassandra をそれぞれサーバノード 10 台, 20 台, 30 台で動かし, クライアントも 10 台, 20 台, 30 台と増やして負荷をかける. KVM 側はクライアント側だけに利用する. weighttp に付けたオプションを以下の通りである.

Listing 5.4: weighttp のオプション（実験 2）

```
1 weighttp -n 50000 -c 200 -t 2 -k "http://url"
```

クライアント 1 台からはそれぞれ 5 万の HTTP リクエストが送られる. 実験 1 に比べ同時接続数とネイティブスレッド数が少ないのは VM の環境に合わせてあるからである.

測定は読み込みと書き込みの両方を行う. 測定の結果をグラフにしたのを図 5.6, 5.7 に示す.

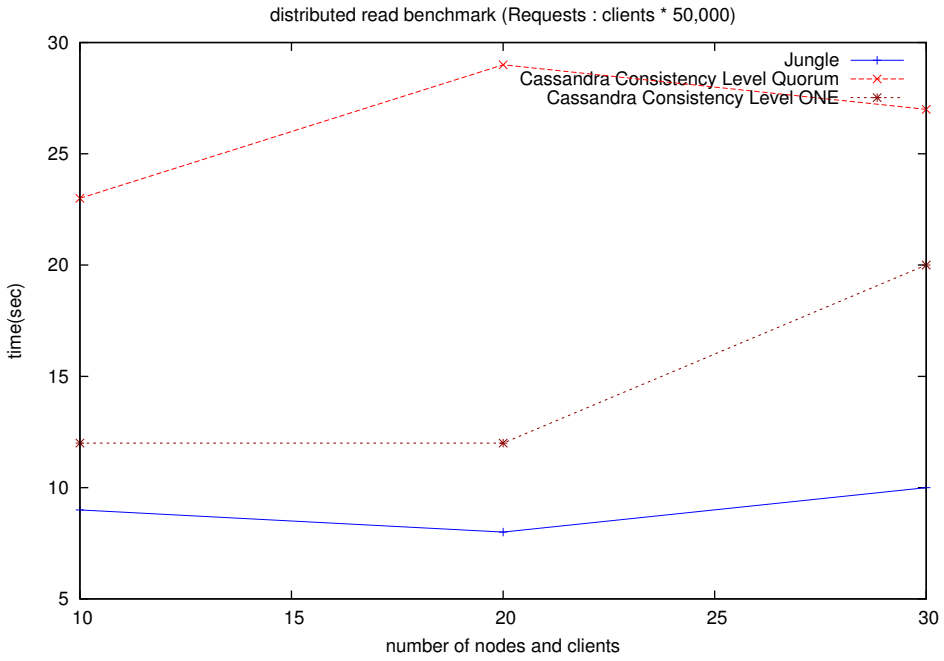

図 5.6: 分散環境下における読み込みベンチマーク結果

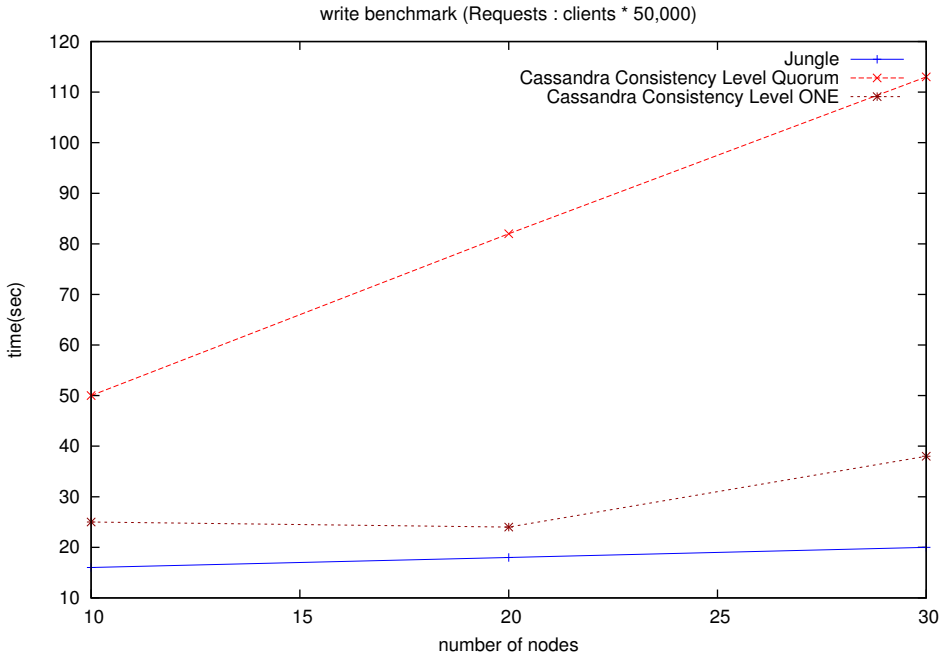

図 5.7: 分散環境下における書き込みベンチマーク結果

第6章　結論

6.1　まとめ

6.2　今後の課題

6.2.1　データ分割の実装

6.2.2　Merger アルゴリズムの設計

6.2.3　Compaction の実装・分断耐性の実装

謝辞

本研究を行うにあたり, ご多忙にも関わらず日頃より多くの助言, ご指導をいただきました河野真治助教授に心より感謝いたします.
また, 様々な研究や勉強の機会を与えてくださった, 株式会社 Symphony の永山辰巳さん, 同じく様々な助言を頂いた森田育宏さんに感謝いたします. 様々な研究に関わることで自身の研究にも役立てることが出来ました.
研究を行うにあたり, 並列計算環境の調整, 意見, 実装に協力いただいた谷成 雄さん, 杉本優さん, 並びに並列信頼研究室の全てのメンバーに感謝いたします.
最後に, 大学の修士まで支えてくれた家族に深く感謝します.

参考文献

[1] Nancy Lynch and Seth Gilbert. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 2002.

[2] Avinash Lakshman and Prashant Malik. Cassandra - a decentralized structured storage system. *LADIS*, Mar 2003.

[3] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly avaliable key-value store.

[4] 玉城将士, 河野真治. Cassandra を使った cms の pc クラスタを使ったスケーラビリティの検証. 日本ソフトウェア科学会, August 2010.

[5] 玉城将士, 河野真治. Cassandra を使ったスケーラビリティのある cms の設計. 情報処理学会, March 2011.

[6] 玉城将士, 河野真治. Cassandra と非破壊的構造を用いた cms のスケーラビリティ検証環境の構築. 日本ソフトウェア科学会, August 2011.

[7] Fay Changand Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable : A distributed storege system for structured data.

[8] Matt Welsh. The staged event-driven architecture for highly-concurrent server applications.

[9] Eric Brewer Matt Welsh, David Culler. Seda : An architecture for well-conditioned , scalable internet services. *SOSP*.

発表履歴

- Java による授業向け画面共有システムの設計と実装, 大城信康, 谷成雄 (琉球大学), 河野真治 (琉球大学), オープンソースカンファレンス 2011 Okinawa, Sep, 2011
- Continuation based C の GCC 4.6 上の実装について, 大城信康, 河野真治 (琉球大学), 第 53 回プログラミング・シンポジウム, Jan, 2012
- GraphDB 入門 TinkerPop の使い方, 大城信康, 玉城将士 (琉球大学), 第 15 回 Java Kuche, Sep, 2012
- ディペンダブルシステムのための木構造を用いた合意形成データベースの提案と実装, 大城信康, 河野真治 (琉球大学), 玉城将士 (琉球大学), 永山 辰巳 (株式会社 Symphony), 情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS), May, 2013
- Data Segment の分散データベースへの応用, 大城信康, 杉本優 (琉球大学), 河野真治 (琉球大学), 日本ソフトウェア科学会 30 回大会 (2013 年度) 講演論文集, Sep, 2013