非破壊的木構造を用いた
分散CMSの設計と実装

A design and implementation of the Distributed CMS using Nondestructive tree structure

平成24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

玉城 将士

要 旨

アブストラクト

目 次

第1章　序論 **1**
1.1 序論 1
1.1.1 研究背景と目的 1
1.1.2 本論文の構成 1

第2章　分散データベースのスケーラビリティ検証 **2**
2.1 はじめに 2
2.2 スケーラビリティとは 2
2.3 分散データベース Cassandra 2
2.3.1 Google BigTable 2
2.3.2 Amazon Dynamo 3
2.3.3 SEDA 3
2.4 分散データベース Cassandra のスケーラビリティ検証 3
2.4.1 検証環境 4
2.4.2 検証結果 4
2.4.3 まとめ 6

第3章　分散コンテンツマネージメントシステムの設計 **7**
3.1 はじめに 7
3.1.1 スケーラブルにするためには 7
3.2 木構造を用いたデータ表現 9
3.3 提案手法 10
3.3.1 非破壊的木構造 10
3.3.2 分散バージョン管理システム 13
3.3.3 グラフデータベース 16
3.3.4 ブラウザサイドの実装 17
3.4 全体の設計 18

第4章　分散コンテンツマネージメントシステムの実装 **19**
4.1 はじめに 19
4.2 開発環境 19
4.2.1 言語 19
4.3 全体の構造 19

4.4 木構造の検索 19
4.4.1 Traverser 19
4.5 木構造読み込み 19
4.5.1 Journal 19
4.6 木構造への書き込み 19
4.6.1 TreeEditor 19
4.7 木構造のマージ 19
4.7.1 Merger 19

第 5 章　結論 **20**
5.1 まとめ 20
5.2 今後の課題 20

謝辞 **21**

参考文献 **22**

発表文献 **23**

図 目 次

2.1 SEDA 3
2.2 PC クラスタを用いた Cassandra の検証環境 4
2.3 Read の結果 5
2.4 Write の結果 5
2.5 Read の結果 5
2.6 Write の結果 5

3.1 データを他の全ノードに通知する 8
3.2 他の全ノードに最新のデータを確認する 8
3.3 過半数のノードに通知, 過半数のノードの答えを採用する 8
3.4 木構造の破壊的編集 10
3.5 競合状態に陥る木構造の破壊的編集 11
3.6 木構造の非破壊的編集 11
3.7 ステップ 1:変更したいノードまでのパスを求める. 12
3.8 ステップ 2:変更したいノードをコピーし, コピーしたノードの内容を変更する. 12
3.9 ステップ 3:求めたパス上に存在するノードをルートノードまでコピーする. 12
3.10 ステップ 4:影響のないノードは共有する. 12
3.11 並列に読み書きが可能な非破壊的木構造 13
3.12 分散バージョン管理システムと集中型バージョン管理システム 14
3.13 変更が衝突しない場合 15
3.14 変更が衝突したが, 自然に解決できる場合 15
3.15 変更が衝突したが, 自然に解決できない場合 15
3.16 プロパティグラフの例 16
3.17 プロパティグラフのトラバース例 17
3.18 システム全体の概要図 18

表 目 次

2.1 スケーラビリティ検証環境のマシン性能一覧 4
3.1 HTML5 で実装される新機能 17

第1章　序論

1.1　序論

1.1.1　研究背景と目的

従来，ウェブサービスとは，サーバーサイドでクライアントからのリクエストを処理し，クライアントは結果の表示を担当してる．しかし，スマートフォンやタブレット端末の普及によりサーバーサイドの負担が増大し，ウェブサービスがダウンする自体が多々見られるようになった．そのため，スケーラビリティがウェブサービスにとって重要な性質の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出来る性質である．
分散システムと聞くと，ネットワークを想像しがちであるが，並列に動作するソフトウェアもスレッドも他のスレッドと協調動作する必要があるため，分散ソフトウェアである．すなわち，マルチコア上で並列に動作するソフトウェアにおいて，コア数に比例して性能も上昇するようなアーキテクチャが必要になる．またウェブサービスは，サーバーサイドにてスケーラビリティを確保することが多く，クライアント側は有効活用されないことが多い．しかし，クライアントはウェブサービスの需要に比例して増加するため，クライアントを有効活用したスケーラビリティ確保も有効である．

本研究では，マルチコア上のスケーラビリティ確保のために非破壊的木構造を利用し，クライアントを活用するため分散版管理システムを参考にしたスケーラブルな分散コンテンツマネージメントシステムの設計と開発を行う．

1.1.2　本論文の構成

本論文では, 初めにどのような方法でスケーラビリティを検証するか確認するために, 学科の PC クラスタを利用した分散データベース Cassandra の検証と得られた知見について述べる.
第 3 章では, 非破壊的木構造と分散リポジトリを参考にしたスケーラビリティ確保の方法について考察し分散コンテンツマネージメントシステムの設計について述べる.
第 4 章では, 第 3 章で述べた設計に基づいた Java での設計と実装について述べる.

第2章　分散データベースのスケーラビリティ検証

2.1　はじめに

スケーラブルなシステムを開発するためには, スケーラブルな性能を測ることの出来る環境とどのようなシステムがスケーラブルであるか知ることが必要である. 本章では, スケーラブルな分散データベース Cassandra を用いて検証を行う. スケーラビリテイを高めるための使い方には工夫が必要である. 検証では, 特に, CoreDuo などの安価だが非力なマシンの振舞を調べることを行なった. そしてその環境上でスケーラビリティを確認する実験手法とスケーラブルなシステムを構成する方法を考察する.

2.2　スケーラビリティとは

スケーラビリティとは, 分散システムの性能がノードの数に対して線形に上昇するような性質である. 通常, システムの性能を上昇させる場合, 高価なサーバーを購入方法と安価なサーバーを複数用意し連携させる方法がある. 前者の方法をスケールアップと呼び, 後者の方法をスケールアウトと呼ぶ. 高価なサーバーを用意するより, 安価なサーバーを複数用意するほうがコストパフォーマンスが良い. しかし, スケールアップでは, 同じ古いサーバーで動作していたソフトウェアがそのまま高価なサーバーで動作させることが可能であるため簡単に性能を向上させることができるが, 複数のサーバーを連携させて性能を向上させるのは簡単なことではない.

2.3　分散データベース Cassandra

Cassandra は, FaceBook が自社のために開発した分散 Key-Value ストアデータベースであり，Dynamo[1] と BigTable[2] を合わせた特徴を持っている. 2008 年にオープンソースとして公開され, 2009 年に Apache Incubator のプロジェクトとなった. 2010 年には Apache のトップレベルプロジェクトとなり, 現在でも頻繁にバージョンアップが行われている.

2.3.1　Google BigTable

BigTable は, Google が開発したデータベースの 1 つである.

2.3.2　Amazon Dynamo

Dynamo は, Amazon が開発したデータベースの 1 つである.

2.3.3　SEDA

SEDA(Staged Event-Driven Architecture) は, Cassandra で使用されているアーキテクチャである [3][4]. 処理を複数のステージに分解しタスクキューとスレッドプールを用意し処理を行う. 処理の様子を図 2.1 に示す.
タスクが各ステージのタスクキューに入ると, スレッドプールにどれかのスレッドがタスクキューの中からタスクを取り出し処理を行う. 処理が終わるとそのタスクを次のステージのタスクキューに入れる. 同様にして次のステージのスレッドプールがタスクキューからタスクを受け取り処理を行う.
このアーキテクチャは数多くのスレッドを生成するためマルチコアな PC と多数のタスクがある状況で性能を発揮することができるため, スケーラブルなソフトウェアアーキテクチャである.
実際, Cassandra には 20 以上のスレッドが動作している. しかし, あまりにもスレッドプールやタスクが多すぎると, コンテキストに切り替えに時間がかかり性能は低下する.

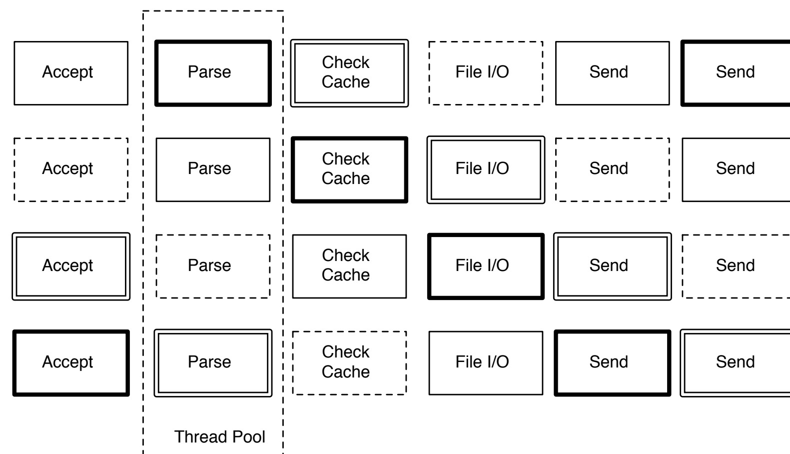

図 2.1: SEDA

2.4　分散データベース Cassandra のスケーラビリティ検証

検証方法

クラスタ管理ツールの Torque を使用し, 使用するノード数を指定してクラスタにジョブを投げて PHP スクリプトを実行させる. この PHP スクリプトは対象のサーバーにリク

エストを 10000 回送信するスクリプトである．検証対象として Cassandra, 比較対象として MySQL を用意した．実験の概要図を図 2.2 に示す.

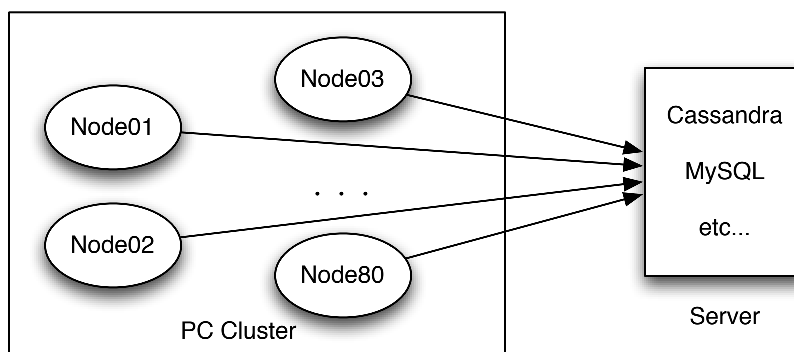

図 2.2: PC クラスタを用いた Cassandra の検証環境

この実験では，徐々に負荷をかけるクラスタの台数を増加させ，並列度を上げていく．クラスタすべてが処理を完了するまでの平均をグラフにプロットし，比較した.

2.4.1　検証環境

利用したスケーラビリティの検証環境を以下に示す.

表 2.1: スケーラビリティ検証環境のマシン性能一覧

種類	CPU	メモリ	OS
クラスタ (クライアント)	Core Duo	1GB	CentOS 5
MacMini	Core2 Duo	4GB	Mac OSX SnowLeopard
サーバー	Core i7	16GB	CentOS 5

2.4.2　検証結果

2Core を搭載したコア数の少ないサーバーを用いた検証

Read は Cassandra/MySQL ともに，似たような性能低下の推移をしていたが Cassandra の方が遅い．しかし，Write は Cassandra の方が断然速く動作している．この実験では，Cassandra の動作を基準に考えたため書き込みのコマンドに REPLACE を使用した．REPLACE は置き換えるようなコマンドである.

そのため, INSERT に比べて多少遅くなる. SEDA は複数のスレッドで動作しているためコア数が少ないサーバーでは性能が出にくいことがわかる.

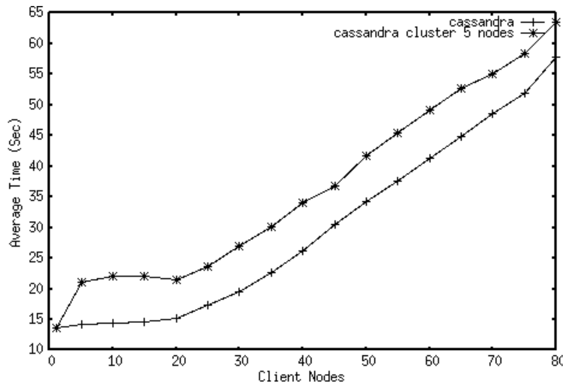

図 2.3: Read の結果

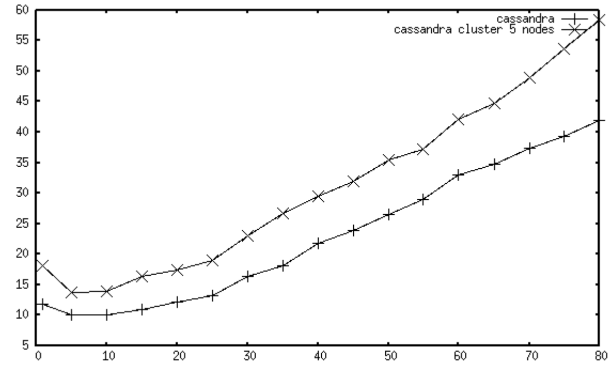

図 2.4: Write の結果

4Core8Threads を搭載したコア数の多いサーバーを用いた検証

Read/Write 共に MySQL の性能を超えることに成功した. Read においてはコア数が少ない場合に超えることが出来なかったが, 並列度が 70 度付近で MySQL を上回る性能がでていた. Cassandra の平均時間は並列度が増加しても, MySQL よりは平均時間の上昇は少ない. これは, SEDA の特徴である多くのタスクを並列に実行すると性能を発揮することを確認することが出来た.
また, SEDA はマルチスレッド前提であるため, コア数が少ないサーバーでは性能が出ず, コア数の多いサーバーで性能が発揮できるということが分かる.

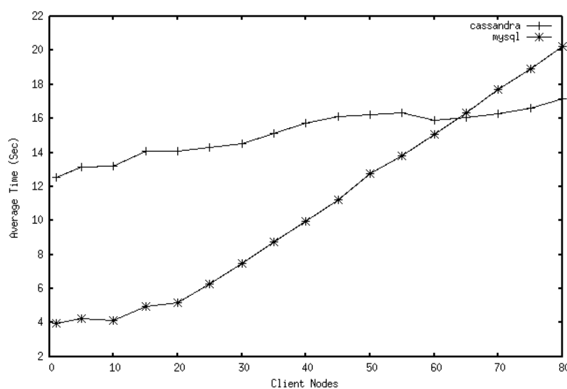

図 2.5: Read の結果

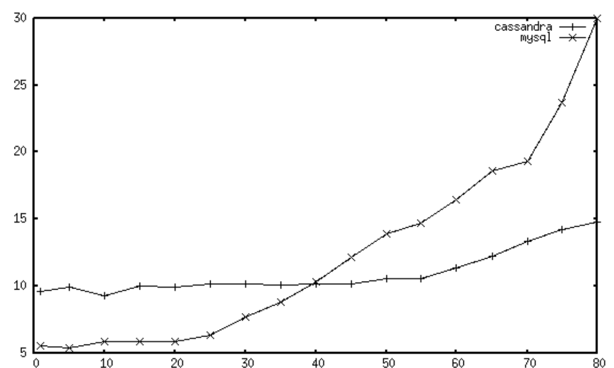

図 2.6: Write の結果

2.4.3 まとめ

本章では, 90 台 PC クラスタサーバーを用いてスケーラビリティの検証環境を構築した. 環境を用いて実際にスケーラブルな分散データベースである Cassandra の性能と SEDA の効果を確認することができた. よって, 本方法がスケーラビリティを検証する上で有効であることが解った.

第3章　分散コンテンツマネージメントシステムの設計

3.1　はじめに

本章では, スケーラブルなコンテンツマネージメントシステムの設計を行う. まずはじめに, スケーラブルにする方法について考察する. 次に, 本研究で我々が提案する幾つかの方法を紹介する. それを用いた, 本システムの基本的なアーキテクチャについて述べる.

3.1.1　スケーラブルにするためには

分散システムでは, 次の3つの性質は同時には成り立たない.

- Consistency データの整合性
- Availability データの可用性
- Pertition-tolerance データの分割耐性

これはCAP定理と呼ばれる定理である. この定理を踏まえて考察する.

スケーラブルにするのは簡単ではなくいくつか弊害がある. 例えば, データベースなどデータを取り扱うシステムに取っては, データの整合性が大事になる. あるデータをリクエストしたとして, 異なるノードが同じデータをレスポンスすることが望ましい. しかし, そのためには, 全ノードのデータの整合性がとれている必要がある.
実現するためには, 以下の方法があげられる.

1. データの更新を受けたノードが, 他の全ノードに通知する.（図3.1）
2. あるデータのリクエストをノードが受けた時, 他の全ノードに最新のデータを確認する.（図3.2）
3. データの更新を受けたノードが, 過半数以上のノードに通知する. また, データのリクエストを受けたノードは, 全ノードにリクエストを送信し過半数以上の回答が得られた時点で最新と判断する

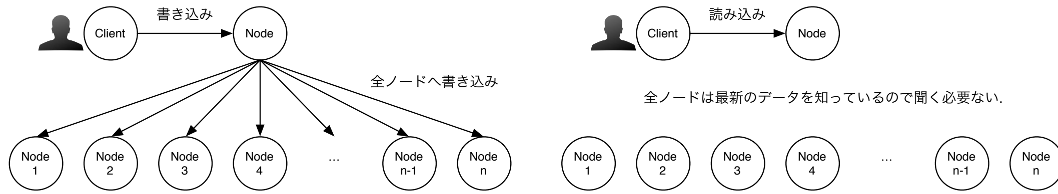

図 3.1: データを他の全ノードに通知する

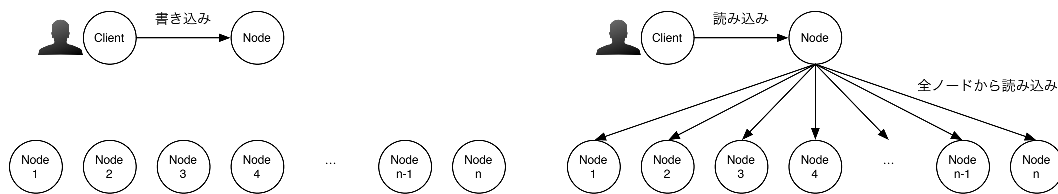

図 3.2: 他の全ノードに最新のデータを確認する

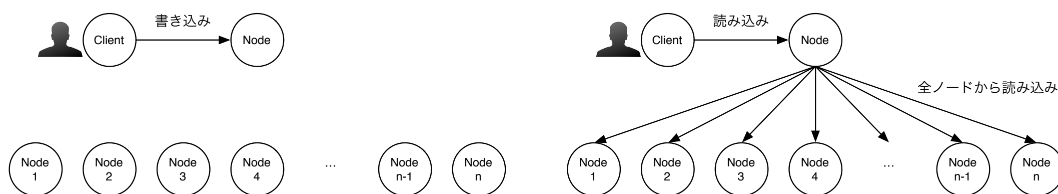

図 3.3: 過半数のノードに通知, 過半数のノードの答えを採用する

最初は書き込み時に負荷をかける方法で, ノードの数に応じて書き込み時の負荷は増えるが, 読み込みは全サーバーが最新のデータを保持しているためスケールする. 二番目の方法は最初の方法と逆で, 読み込みに負荷をかける方法で, ノード数に応じて書き込みの負荷は増えるが, 書き込みはスケールする. 三番目の方法は書き込みと読み込みに半々の負荷をかけて分散する方法である. どちらにせよ, これらの分散かつ整合性を取るにはスケーラブルに出来ない部分が存在することがわかる. 銀行などのシステムはお金が関係するため, 整合性は欠かすことが出来ない性質である.

整合性が必須であるシステムがある中, 大して整合性が必要のないシステムについて考える. Twitter や Facebook などはタイムラインが数分程度遅延しても全く問題の無いサービスである. これらの SNS は, 遅延はあるものの最終的に整合性のとれたデータがタイムラインに現れる. これを, 結果整合性という. 結果整合性は, 負荷が少ない時や細かい更新をまとめて行うなど細かい工夫を入れることで実現でき負荷を軽減することも可能である. つまり, CAP 定理の Consistentcy は無視して, 以下の組み合わせをとることが出来る.

- Avaliability + Pertition-tolerance + Eventual Consistency

この組み合わせが現在のシステムで多く利用されている方法の 1 つである. そして, 前述した三種類の方法を結果整合性になるように変更すると以下のようになると考える.

- データの更新を受けたノードが一定時間ごとに, 他のノードに通知する. 最新データは自分のみに書き込む.
- ノードは一定時間ごとに他のノードから更新が無いかチェックし, リクエストを受けたときは自分が保持するデータを利用する.

3.2　木構造を用いたデータ表現

我々のシステムでは, データ構造として木構造を採用する. 一般的なコンテンツマネージメントシステムはブログツールや Wiki・SNS が多い. これらのウェブサイトの構造は大体が木構造である.
ウェブログはトップページとカテゴリが存在し, カテゴリの中にサブカテゴリが存在しエントリーに到達する. このように, ウェブサイトは木構造でうまく表現できると考える, また, 木構造は並列に編集可能なデータ構造としても実装できるため, 前章で述べたマルチコア環境でのスケーラブルなアーキテクチャと相性が良い.

3.3　提案手法

3.3.1　非破壊的木構造

非破壊的木構造は, 一度構築した木構造を破壊することなく新しい木構造を構成することで, 木構造を編集する方法である.
破壊的木構造と異なりロックフリーであるため, 並列に読むことが出来きるため自由にコピーを作成することが可能である.
よって, コピーを複数作成しアクセスを分散させることで性能を維持することが出来ると考えられる.

通常の破壊的木構造との違いを説明する.

破壊的木構造

破壊的木構造は, 木構造を編集する際に木構造を書き換えることにより編集を実現する.
図 3.4 では, ノード 6 をノード A へ書き換える処理を行なっている.

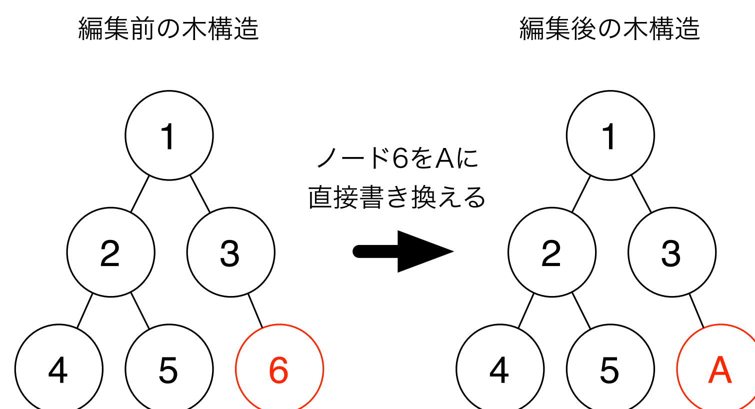

図 3.4: 木構造の破壊的編集

この方法では, 並列環境において問題が発生する. ある時点の木構造を参照している閲覧者と編集する編集者を考える. 閲覧者が木構造を参照中に編集者が木構造を書き換えると, 閲覧者が参照を開始した時点での木構造ではく整合性は崩れている. 整合性が崩れた状態では正しい状態のコンテンツを参照することは出来なくなってしまう. (図 3.5)

この状態を回避するためには, 木構造にアクセスする際は木構造をロックする. しかし, ロックすることは排他処理を行い, 木構造を利用している処理が 1 つであることを保証するものであるため, 並列度を下げることになる. 結果, スケーラビリティを損なってしまい本システムには破壊的木構造は向かないことが分かる.

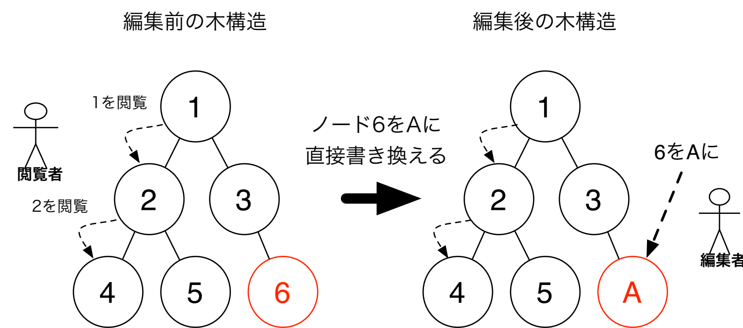

図 3.5: 競合状態に陥る木構造の破壊的編集

非破壊的木構造

一方, 非破壊的木構造は, 木構造を書き換えることなく編集を行うため, 読み書きを並列に行うことが可能である. 図 3.6 では, 図 3.4 同様に, ノード 6 をノード A へ書き換える処理を行なっている.

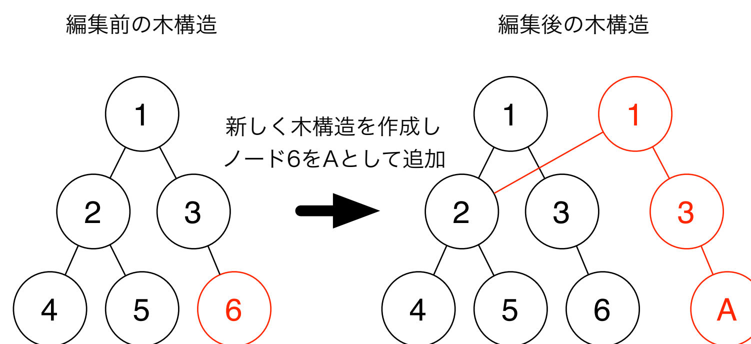

図 3.6: 木構造の非破壊的編集

赤色で示したノードが新しく追加されたノードである. 非破壊的木構造の基本的な戦略は, 変更したいノードへのルートノードからのパスを全てコピーする. そして, パス上に存在しない（編集に関係のない）ノードはコピー元の木構造と共有することである. 以下にその方法を示す.

1. 変更したいノードまでのパスを求める. (図 3.7)
2. 変更したいノードをコピーし, コピーしたノードの内容を変更する.(図 3.7
3. 求めたパス上に存在するノードをルートノードに向かって, コピーする. コピーしたノードに一つ前にコピーしたノードを子供として追加する.(図 3.7
4. 影響のないノードをコピー元の木構造と共有する.(図 3.7)

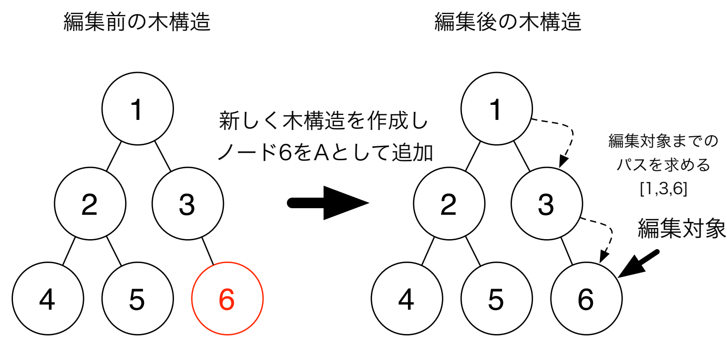

図 3.7: ステップ 1:変更したいノードまでのパスを求める.

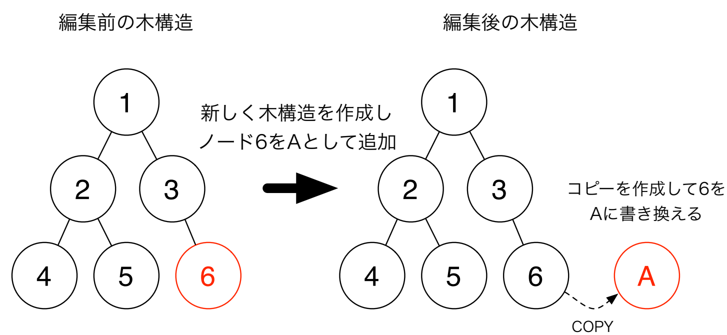

図 3.8: ステップ 2:変更したいノードをコピーし, コピーしたノードの内容を変更する.

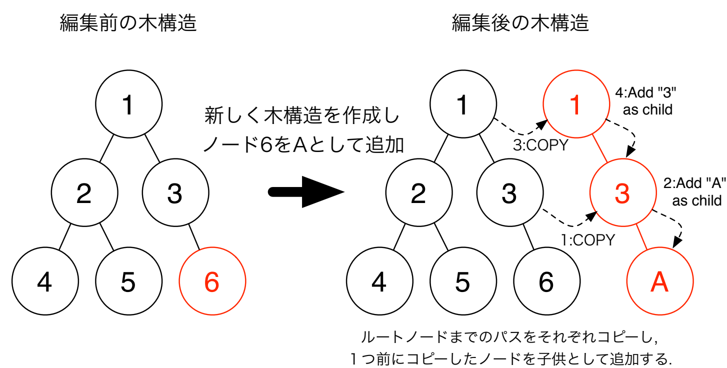

図 3.9: ステップ 3:求めたパス上に存在するノードをルートノードまでコピーする.

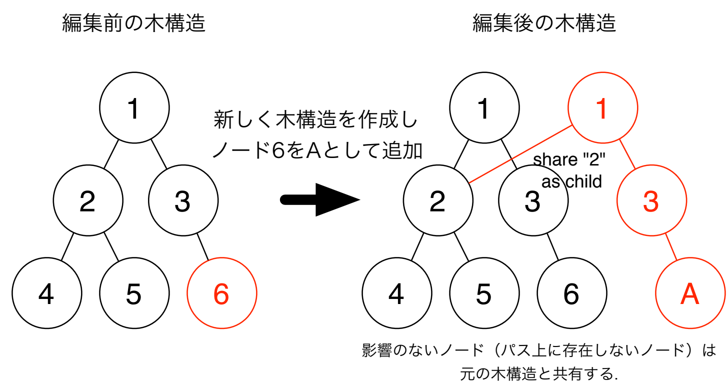

図 3.10: ステップ 4:影響のないノードは共有する.

以下に図示して説明する.

この編集方法で破壊的木構造の場合と同様に, ある時点の木構造を参照している閲覧者と編集する編集者を考える. 閲覧者が木構造を参照している中, 編集者が非破壊的な編集を行う. しかし, 破壊的方法とは異なり, 元の木構造は破壊されることはないため編集後も整合性は崩れることはない. (図 3.11) よって, ロックが必要なく並列に読み書きが可能であるため, スケーラブルなシステムに有用であると考えられる. また, 元の木構造は破壊されることがないため, 自由にコピーを作成しても構わない. すなわち, アクセスの負荷も分散させることが出来るはずである.

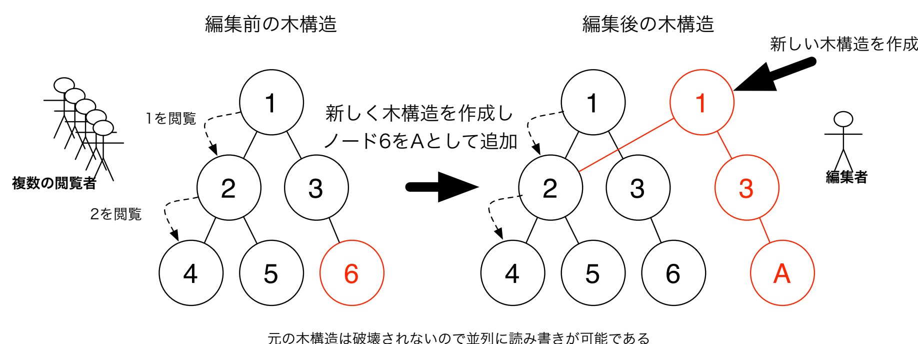

図 3.11: 並列に読み書きが可能な非破壊的木構造

我々のシステムでは, 非破壊的木構造を採用してシステムを設計する.

3.3.2　分散バージョン管理システム

分散バージョン管理システムとは, Git や Mercurial に代表される分散版管理システムである. バージョン管理システムとは, 多人数のソフトウェア開発などで利用され, 開発の変更履歴を管理するために用いる. 更新履歴を管理する対象のまとまりをレポジトリと呼ぶ. 従来の, バージョン管理システムは集中型システムであり, レポジトリを保存するサーバーが存在しローカルにチェックアウトすることで変更を加える. 変更を加えたレポジトリはチェックアウト元にコミット・マージされサーバーに更新が伝わる. 他のユーザーは, それを自身のローカルにアップデートし変更が伝搬する仕組みである.
一方, 分散型はローカルにクローンすることでクローン先自体が新たなレポジトリとして独立で存在できる. よって, 同じレポジトリからクローンされた複数の新レポジトリはお互いにマージすることができる. この仕組みより, 前述した結果整合性を用いたシステムが構築できると考える. (図 3.12)

Push/Pull 方式

Push/Pull 方式とは, 分散レポジトリで利用されているレポジトリの結合方法の 1 つである. 分散バージョン管理システムでのレポジトリはお互いに Push/Pull を用いて自身の

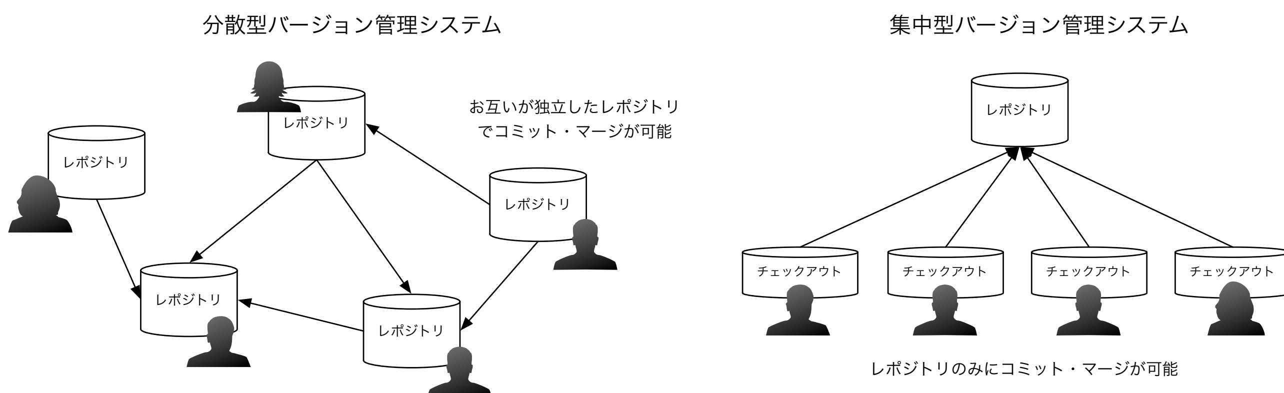

図 3.12: 分散バージョン管理システムと集中型バージョン管理システム

更新を通知・受信することができる.
Push とは, あるレポジトリが異なるレポジトリに自分の持つ更新を通知することで, Pull は, その逆の更新を他のレポジトリより受信することである. また, 異なる履歴を持つレポジトリ同士が Push/Pull を行う時お互いの更新がしょうとする可能性が考えられる, これらの問題はマージを用いることで解決する.
これを我々が提案した方法に当てはめると以下のようになる.

- データの更新を受けたノードが一定時間ごとに, 他のノードに結果を Push する. 最新データは自分のみに書き込む.
- ノードは一定時間ごとに他のノードから更新を Pull し, リクエストを受けたときは自信が保持するデータを利用する.

この方式は, 他のノードが故障またはネットワークが分断されてもノード自身は独立して動作するため Availability と Partition-tolerance があるのではないかと考えられる.

マージ

Push/Pull する際に更新が衝突しない場合は良いが衝突する場合もある. 木構造を例に以下の状態を考える.

この状態は衝突が発生していない, なぜなら Pull 元が編集されていないためである. 次に図 3.14 について考える.
この場合, お互いの異なる履歴をもつ木構造がマージされる場合で変更が衝突している. しかし, 一方の木構造に対する変更が, もう一方の木構造の編集と衝突していない.
よってこの場合は自然にマージすることが可能である.

次の図 3.15 は, 自然にマージできない場合である. 一方の変更箇所がもう一方と衝突している.

衝突する場合は, マージ処理が必要である. マージ処理はお互いの変更点を検出し, 更新を自身の木構造に衝突を取り除いた状態で取り込むことである.

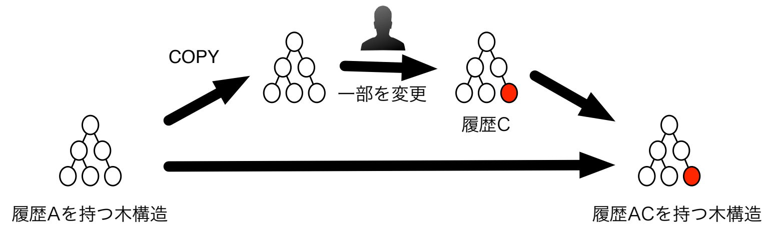

図 3.13: 変更が衝突しない場合

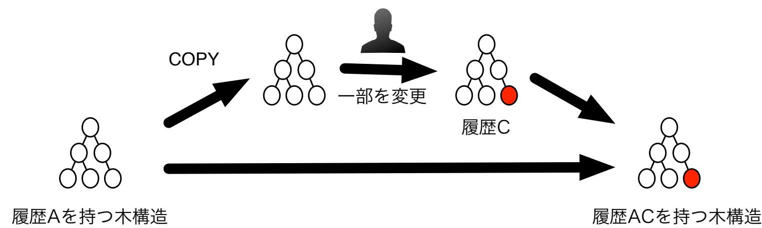

図 3.14: 変更が衝突したが, 自然に解決できる場合

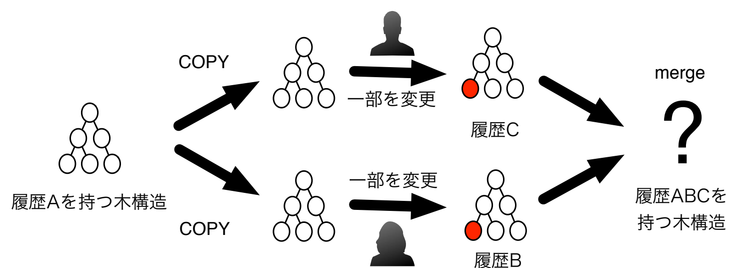

図 3.15: 変更が衝突したが, 自然に解決できない場合

分散レポジトリでは, お互いの更新点を検出する際に変更履歴を利用することで効率よくマージを実行することが出来ると考えられため, 我々のシステムでもマージの際に木構造に対しての更新履歴を参照できるようにする.
マージには衝突を回避する他に, スケーラブルにするために工夫を加えることが出来る. なぜならば, 管理するコンテンツにおいて厳密なマージ処理が必要なコンテンツとそうでないコンテンツが存在するからである.
そのため, 我々のシステムでは単純なマージアルゴリズムのほか, プログラマがマージアルゴリズムを記述できるようにできる仕組みを導入する.

3.3.3　グラフデータベース

グラフデータベースとはリレーショナルデータベースと異なり, グラフ構造を保存するデータベースである. 主な実装に Neo4j,OrientDB,InfiniteGraph があるほか, TinkerPop という様々なグラフデータベースの実装の差異を吸収するためのプロジェクトも存在する. グラフデータベースはプロパティグラフと呼ばれる種類のグラフを保持することが出来る. 主にソーシャルネットワークの人物関係を表すソーシャルグラフを表現するのに利用されるほか, ページランクのアルゴリズムにも利用される.

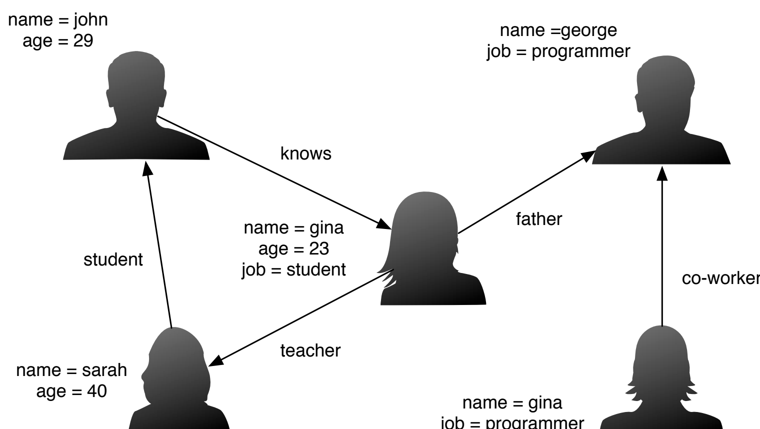

図 3.16: プロパティグラフの例

トラバース

グラフデータベースでのデータの検索方法はトラバースと呼ばれ, グラフ上を渡り歩く方法を示すことで目的のデータを取得する.
トラバースは, 並列に効率よく行うことが可能であるためスケーラビリティに貢献できると考えられる. 以下に TinkerPop でのトラバース方法を示す.

Listing 3.1: トラバースの例

```
1 graph.v(1).out('knows').out('father);
```

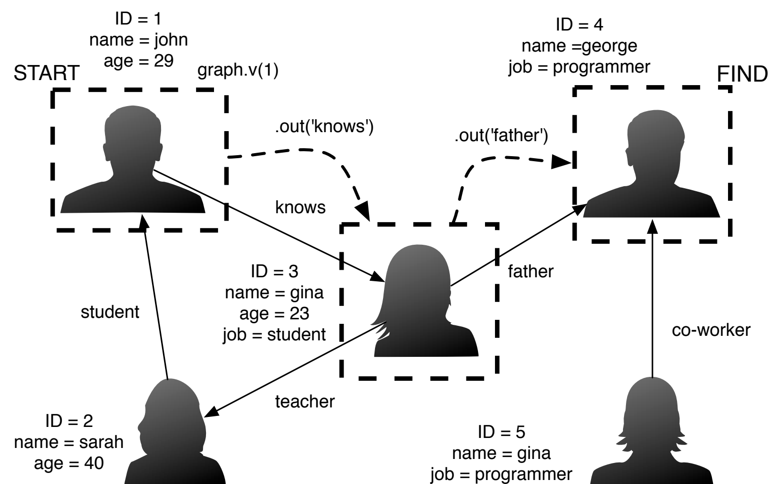

図 3.17: プロパティグラフのトラバース例

我々のシステムでは, トラバースを木構造の検索方法として採用する. 木構造はグラフ構造の 1 つであるため効率よく検索を行うことができると考えられるからである. なぜならば, 複数の結果が得られるようなトラバース例を考える. このとき,

3.3.4 ブラウザサイドの実装

近年のウェブブラウザは JavaScript でかなり複雑な処理が可能になっている. その上, HTML5 の制定によりウェブストレージやウェブソケットなどの IO までサポートされるようになる. 以下に, HTML5 で実装される主な IO 関連の機能を示す.

表 3.1: HTML5 で実装される新機能

名称	説明
WebSocket	ウェブサーバーとウェブブラウザが相互通信するための機能.
Indexed Database API	値とオブジェクトをローカルデータベースに保存できる.
WebStorage	値とオブジェクトの組みを保存できる, sessionStorage と localStorage の 2 種類が存在する.
App Cache	キャッシュマニフェストでブラウザのキャッシュを操作できるようになる.

通常, システムの負荷はクライアントの数に応じて増加する. 我々のシステムにおいてはクライアントはウェブブラウザに相当するため, ウェブブラウザをスケーラビリティに貢献できるようにすることでかなりの効果を期待できると考えられる.

3.4　全体の設計

提案手法を用いたシステム全体の概略図を以下に示す.

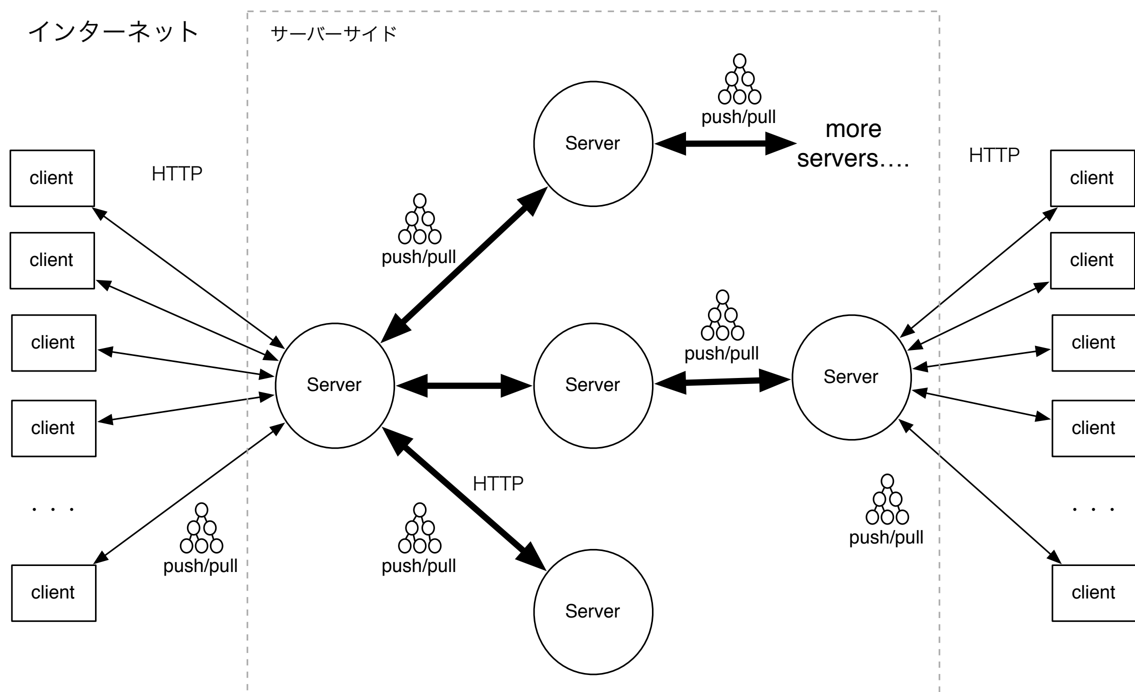

図 3.18: システム全体の概要図

本システムでは, データ構造として木構造を採用する. 各サーバー・クライアントはそれぞれ木構造が保存されており, サーバーはサーバー同士で木構造の push/pull を行い, クライアントはサーバーと木構造の push/pull を行う. 前述した, スケーラブルにする方法,

第4章　分散コンテンツマネージメントシステムの実装

4.1　はじめに

はじめに

4.2　開発環境

4.2.1　言語

4.3　全体の構造

4.4　木構造の検索

4.4.1　Traverser

4.5　木構造読み込み

4.5.1　Journal

4.6　木構造への書き込み

4.6.1　TreeEditor

4.7　木構造のマージ

4.7.1　Merger

第5章　結論

5.1　まとめ

まとめ

5.2　今後の課題

今後の課題

謝辞

ありがとうございました。

参考文献

[1] 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.

[2] 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.

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

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

[5] 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.

[6] 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.

[7] 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.

発表履歴

- 河野研ゼミ