

修士(工学)学位論文
Master's Thesis of Engineering

ソフトウェア内部で使用するのに適した
木構造データベース Jungle
Tree Structured Database Jungle for software
internal use

2017年3月
March 2017

金川 竜己
Tatsuki Kanagawa



琉球大学大学院 理工学研究科
情報工学専攻
Information Engineering Course
Graduate School of Engineering and Science
University of the Ryukyus

本論文は、修士(工学)の学位論文として適切であると認める。

論文審査会

印

(主 査) Mohammad Reza Asharif 氏

印

(副 査) 岡崎 威生 氏

印

(副 査) 赤嶺 有平 氏

要 旨

プログラムからデータを分離して扱うデータベースには、プログラム中のデータ構造と表構造とのインピーダンスミスマッチという問題がある。データベースのレコードをプログラム中のオブジェクトとして使える OR Mapper や、データベース自体も、表に特化した Key Value Store、Json などの不定形のデータ構造を格納するように機能拡張されてきている。しかし、プログラム中のデータは複雑な構造をメモリ上に構築しており、これらの方法でもまだギャップがある。

そこで当研究室では、これらの問題を解決するためにプログラム内部に木構造を格納できるデータベース Jungle を提案している。Jungle は、木構造の変更を非破壊的に行う。Jungle は、読み込みは高速に行える反面、書き込みの手間は木の形・大きさに依存しており、最悪の場合 $O(n)$ となってしまう。また、Index の構築も大幅なネックとなっていた。そこで、本研究では Jungle の木と Index の編集機能の改善を行う。また、実際に Jungle を使用した複数のアプリケーションを作成し、PostgreSQL と MongoDB との比較を行った。

Abstract

Database which handles data from programs separately has an impedance mismatch problem. Several technologies are introduced such as OR Mapper which uses records as objects in a program, Key Value Store(KVS) which is a specialized databases for tables, storing Json tree structures in conventional Relational Databases. Despite of these technologies, it is not sufficient to handle complex data structures in the program memories.

In this paper, Tree structured database Jungle is introduced. Jungle stores tree structures in the program and it modifies trees non-destructively, which is suitable for parallel processings. Jungle is very good at reading tree structures, but updating tree structures may requires $O(n)$ in worst case depending on configurations of the trees. Updating indexes on the trees are also having computational complexity difficulties, Several improvements are presented and comparisons are shown to PostgreSQL and MongoDB in Jungle applications.

目次

第 1 章	ソフトウェア内部で使用するのに適した 木構造データベース Jungle	1
1.1	本論文の構成	1
第 2 章	既存のデータベース	3
2.1	PostgreSQL	3
2.2	MongoDB	4
2.3	Cassandra	4
2.4	非破壊的木構造データベース Jungle	5
第 3 章	非破壊的木構造データベースの利点	7
3.1	破壊的木構造	7
3.2	非破壊的木構造	7
第 4 章	Jungle の構成要素	10
4.1	木の生成	10
4.2	JungleTree	10
4.3	Either	11
4.4	TreeNode	12
4.5	NodePath	13
4.6	木の編集 API	14
4.7	Commit	16
4.8	Log	16
4.9	検索 API	17
4.10	Index	17
第 5 章	データベース Jungle に 新たに追加した構成要素	19
5.1	非破壊 Red Black Tree の実装	19
5.2	Index の差分 Update	19
5.3	Differential Jungle Tree の実装	19
5.4	Red Black Jungle Tree	20

第 6 章	非破壊 TreeMap の実装	21
6.1	Red Black Tree	21
6.2	非破壊 TreeMap の定義	21
6.3	非破壊 TreeMap の API	22
6.4	非破壊 Red Black Tree へのデータの挿入	22
6.5	非破壊 Red Black Tree のノード削除	25
第 7 章	Index の差分 Update	30
7.1	差分 Update の実装	30
7.2	編集前のノードの削除	30
7.3	Index へのノードの挿入	31
7.4	Full Update との使い分け	31
第 8 章	Differential Jungle Tree	32
8.1	Differential Tree Context	33
8.2	Differential Jungle Tree の作成	33
8.3	末尾ノードを使用した木の編集	33
8.4	Differential Jungle Tree の検索	35
8.5	Differential Jungle Tree の整合性	35
第 9 章	Red Black Jungle Tree	37
9.1	Red Black Jungle Tree の作成	37
9.2	NodePath の拡張	38
9.3	Red Black Jungle Tree の編集	38
9.4	Jungle Red Black Tree の検索	40
第 10 章	Jungle を使ったアプリケーション	41
10.1	Jungle Tree ブラウザ	41
10.1.1	木構造の表示	41
10.1.2	Jungle Tree ブラウザを使った木の編集	41
10.2	HTML Rendering Engine	42
10.2.1	Contents Tree の Jungle 上での表現	42
10.2.2	Layout	43
10.2.3	Layout Tree のデータ設計	45
10.3	許認可管理アプリケーション	
	maTrix	47
10.3.1	Jungle 上での maTrix データ構造の表現	47
10.3.2	木構造の版管理	48
10.4	申請の許認可	49

第 11 章 性能測定	50
11.1 測定環境	50
11.2 TreeMap の測定	51
11.3 Index の差分 Update の測定	52
11.4 正順の線形木の構築時間の測定	54
11.5 Red Black Jungle Tree の測定	55
11.6 既存のデータベースとの比較	56
第 12 章 結論	57
12.1 まとめ	57
12.2 今後の課題	58
12.2.1 過去のデータの掃除	58
12.2.2 木の設計手法の確立	58
謝辞	59
参考文献	60
発表文献	62

目次

3.1	非破壊的木構造の編集	8
3.2	非破壊的木構造による利点	8
4.1	NodePath	14
6.1	データ挿入時のバランス 3	23
6.2	データ挿入時のバランス 4	24
6.3	データ挿入時のバランス 5	25
6.4	データ削除時のバランス 2	26
6.5	データ削除時のバランス 3	27
6.6	データ削除時のバランス 4	27
6.7	データ削除時のバランス 5	28
6.8	データ削除時のバランス 6	29
8.1	PushPop	32
8.2	末尾ノードを使用した木の編集	34
8.3	複数の版の木の表現	35
8.4	木の整合性が崩れる例	36
10.1	ContentTree	42
10.2	LayoutTree	43
10.3	複数の Component を参照する Layout	45
10.4	コードとギャップのない Layout の格納方法	46
10.5	コードとギャップのある Layout の格納方法	46
10.6	Jungle 上での人物 Tree の表現例	47
10.7	構成情報モデル Tree のノードの中身	48
11.1	TreeMap への Get	51
11.2	Index の Update	52
11.3	Commit を行うまでに木に加えた変更回数と、Index の構築時間	53
11.4	Differential Tree と Default Jungle Tree	54
11.5	Red Black Jungle Tree と Default Jungle Tree	55
11.6	既存の DB との比較	56

表 目 次

4.1	Jungle に実装されている API	10
4.2	TreeContext が保持している値	11
4.3	JungleTree に実装されている API	11
4.4	Either に実装されている API	12
4.5	TreeNode に実装されている API	12
4.6	Children に実装されている API	12
4.7	Attribute に実装されている API	13
4.8	Editor に実装されている API	15
6.1	非破壊 TreeMap に実装されている API	22
8.1	Jungle に新しく実装した API	33
9.1	Jungle に新しく実装した API	37
9.2	Red Black Jungle Tree と Default Jungle Tree の API の違い	39
9.3	Red Black Jungle Tree Interface Traverser が提供している API	40
10.1	ノードが保持している Contents 一覧	43
10.2	LayoutTree の主要な要素	44
10.3	tag と contents の対応	44
10.4	図 10.7 の構成情報モデル Tree の中身	48
11.1	実験環境	50

第1章 ソフトウェア内部で使用するのに 適した 木構造データベース Jungle

プログラムからデータを分離して扱うデータベースには、プログラム中のデータ構造と表構造のミスマッチという問題がある。例えばRPGゲーム中のユーザが持つアイテムという単純なものでも、RDBではユーザとアイテムの組をキーとする巨大な表として管理することになる。プログラム中では、ユーザが持つアイテムリストという簡単な構造を持つが、データのネスト構造を許さない第一正規形を要求するRDBとは相容れない。レコードをプログラム中のオブジェクトに対応させるOR Mapperという技術、データベース自体を表に特化したKey Value Store技術、PostgreSQLやMySQLにJsonなどの不定形のデータ構造を格納技術が開発されてきたが、これを本質的には解決することはできていない。この問題はデータベースのインピーダンスミスマッチと呼ばれている。不定形の構造の変更をトランザクションとしてJsonの一括変更という形で処理するのは並列処理が中心となってきている今のアプリケーションには向いているとは言えない。そこで当研究室では、これらの問題を解決するためにプログラム内部に木構造を格納できるデータベースJungleを提案している。Jungleは、木構造の変更を非破壊的、つまり、元の木を保存しつつ、新しい木を構築する方法を取り、木のルートを実アトミックに入れ替えることでトランザクションを実現する。プログラムは、この木を内部のデータ構造として直接取り扱うことができる。Jungleは、全体の整合性ではなく、木ごとに閉じた局所的な整合性を保証している。また、整合性のある木同士をマージすることで新しい整合性のある木を作り出すことも可能であるため、データの伝搬も容易である。

1.1 本論文の構成

Jungleは、読み込みは高速に行える反面、書き込みの手間は木の形・大きさに依存しており、最悪の場合 $O(n)$ になってしまう。また、Indexの構築も大幅なネックとなっていた。そこで、本研究では、Jungleの木とIndexの編集機能の改善を行う。また、実際にJungleを使用した複数のアプリケーション作成し評価を行う。

本論文の構成は以下のようになっている。

- 第2章で既存のデータベース

- 第3章では Jungle の基本的な機能・API
- 第4章では Jungle に新しく加えた構成要素
- 第5章では改良後の Index の実装に使用する非破壊 TreeMap の実装
- 第6章では Index の差分 Update の実装
- 第7章では線形の木を、正順で $O(1)$ で構築することが可能な、Differential Jungle Tree の実装
- 第8章では自身が Index としての機能を持つ、Red Black Jungle Tree の実装
- 第9章では実際に Jungle を使用した、例題アプリケーションの実装
- 第10章では今回実装した機能の測定

第2章 既存のデータベース

本章では、既存のデータベースの例として、PostgreSQL・MongoDB・Cassandra・当研究室で開発している Jungle について記述する。

2.1 PostgreSQL

PostgreSQL は、列と行からなる 2次元のテーブルにより実装されるデータベースである。厳密な型を持つデータベースであり、きちんと設計を行えば、柔軟なクエリを用いてどんな検索にも対応できる力を持つ。

データベースへのアクセスは、SQL を用いて行う。データの格納を行う際は、まずテーブルのデータの型と・制約を定義する。テーブルの定義は CREATE TABLE 文を用いて行う。ソースコード 2.1 に int 型の id と TEXT 型の name を持つテーブル person を作成するサンプルコードを記述する。

ソースコード 2.1: テーブルの作成

```
1 create table person(id int,name TEXT);
```

テーブルへのデータの格納は、Insert 文を用いて行う。ソースコード 2.2 に、id が 3、name が kanagawa の値を格納するサンプルコードを記述する。

ソースコード 2.2: データの挿入

```
1 insert into person(id,name) values(3,'kanagawa');
```

テーブルにデータを格納する際にはスキーマの定義に沿ってなければならない。もし、スキーマに反するデータを格納した場合エラーが発生しデータの格納に失敗する。

テーブルへのデータの検索は、select 文を用いて行う。select 文では、表示するデータの指定などが行える。ソースコード 2.3 に、テーブル person から name が kanagawa のデータの id を取得するコードを記述する。

ソースコード 2.3: データの検索

```
1 select id from person where name='kanagawa';
```

PostgreSQL では、Json 形式を Json・JsonB という型を用いて格納する。Json 型は、Json データを文字列で格納し、JsonB はバイナリで格納する。Json 型で格納した場合、データにアクセスするたびにパースする必要がある。

2.2 MongoDB

MongoDB[1] 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、スキーマレスと呼ばれる。

MongoDB では、テーブルの代わりにコレクションにデータを保持する。スキーマが無いので、事前にデータの定義を行う必要がなく、同じコレクションであっても、他のドキュメントが持っていないフィールドやデータ型をドキュメントに含めることができる。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。コレクションへのデータの格納は、`insert()` を用いて行う。ソースコード 2.4 に、`id` が 5、`name` が `kanagawa` のデータをコレクション `person` に挿入するサンプルコードを記述する。

ソースコード 2.4: データの検索

```
1 db.person.insert({id:5,name:"kanagawa"});
```

また、Json 形式のデータは任意の深さまでネストすることが可能である。

作成したコレクションからのデータの取得は、`find` を用いて行う。ソースコード 2.5 に、`id` が 5 のユーザーの `name` を取得するサンプルコードを記述する。

ソースコード 2.5: データの検索

```
1 db.person.find({id:5},{name:1});
```

`find` に引数を渡さなかった場合、コレクションの中身が全て表示される。

MongoDB は、あらゆる箇所で JavaScript を用いており、前述した `insert()`・`find()` といった関数も JavaScript で実装されている。db ですら JavaScript のオブジェクトである。`find` で使用するクエリも JavaScript で記述できる。

MongoDB は実装に `mmap` を使用しているため、トランザクションを安全に行うことが難しい。そのためデータの喪失などが発生することが知られている。

2.3 Cassandra

Cassandra は 2008 年 7 月に Facebook によって開発された Key-Value なデータベースである。Cassandra は、分散環境下では複数のノードにデータのレプリカを置くことで、データの信頼性を確保する。いくつかのノードにレプリカを置くかはレプリケーション係数を指定することで決定する。またノードの集合をデータセンターとして管理している。

Cassandra はデータを [KeySpace] [ColumnFamily] [Key] [Column] の 4 次元の連想配列で持つ。データを格納する場合初めに KeySpace を構築する。KeySpace を作る際は、データセンターの設定とレプリケーション係数の設定を同時に行う。ソースコード 2.6 に単一のデータセンターにレプリケーションを一つしか持たない KeySpace である `testKeySpace` を作成するサンプルコードを記述する。

ソースコード 2.6: KeySpace の作成

```
1 create keyspace testKeySpace with replication = {'class':'SimpleStrategy', '
  replication_factor':1};
```

KeySpace は、PostgreSQL でいうところのデータベースに近い働きをする。

次は、作成した Keyspace に対して ColumnFamily を構築する。ColumnSpace は、RDB というテーブルに似た働きをする。ソースコード 2.7 に、int 型の id と text 型の name を持つ ColumnSpace を作成するサンプルコードを記述する。

ソースコード 2.7: ColumnFamily の作成

```
1 create table person (id int PRIMARY KEY, name text);
```

また、ColumnSpace の定義を行う際最低 1 つ Key に値を特定するための PRIMARY KEY を指定する必要がある。

作成した ColumnFamily への値の挿入は、insert 文を用いて行う。ソースコード 2.8 に、id が 1、name が kanagawa のデータを挿入するサンプルコードを記述する。

ソースコード 2.8: ColumnFamily の作成

```
1 insert into person(id,name) values(1,'kanagawa');
```

作成した ColumnFamily からのデータの取得は、select 文を用いて行う。ソースコード 2.9 に、id が 1 のユーザーの name を取得するサンプルコードを記述する。

ソースコード 2.9: ColumnFamily の作成

```
1 select name from person where id = 1;
```

Cassandra はデータの読み書きをいくつかのノードに行うかを Consistency Level で設定する。Consistency Level は主に、ONE、QUORUM、ALL がある。データの複製であるレプリケーションの数を 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。Cassandra は Consistency Level を変えることで、最新のデータを取得するかどうかを決めることができる。しかし、最新のデータを取得するために、Consistency Level を QUORUM や ALL に変えた場合、データの読み書き時に問い合わせるノードが増えるため速度は落ちてしまう問題もある。

2.4 非破壊的木構造データベース Jungle

Jungle は、当研究室で開発を行っているデータベースで、Java を用いて実装されている。Jungle は名前付きの複数の木の集合からなり、木は複数のノードの集合で出来ている。ノードは自身の子のリストと属性名と属性値の組でデータを持ち、データベースのレコードに相当する。通常のレコードと異なるのは、ノードに子供となる複数のノードが付くところである。Jungle では、親から子への片方向の参照しか持たない。

通常の RDB と異なり、Jungle は木構造をそのまま読み込むことができる。例えば、XML や Json で記述された構造を、データベースを設計することなく読み込むことが可能である。また、この木を、そのままデータベースとして使用することも可能である。しかし、木の変更の手間は木の構造に依存する。特に非破壊木構造を採用している Jungle では、木構造の変更の手間は $O(1)$ から $O(n)$ となりえる。つまり、アプリケーションに合わせて木

を設計しない限り、十分な性能を出すことはできない。逆に、正しい木の設計を行えば高速な処理が可能である。

Jungle は基本的に on memory で使用することを考えており、一度、木のルートを取得すれば、その上で木構造として自由にアクセスして良い。Jungle は分散データベースを構成するように設計されており、分散ノード間の通信は木の変更のログを交換することによって行われる。持続性のある分散ノードを用いることで Jungle の持続性を保証することができる。詳しい API については次章で記述する。

第3章 非破壊的木構造データベースの 利点

本章では、非破壊的木構造を選択した理由について述べる。従来の破壊的木構造は親から子へのポインタと子供から親へのポインタを持つ二重リンク構造を持っていた。これによりノードの挿入削除を $O(1)$ で行うことができる。しかし、変更前の木を保存することとは両立せず、変更中に前の版の木を使用することはできない。そこで、子供から親へのポインタを無くし、木を共有しながら変更部分を付け加える非破壊的木構造を提案する。これにより、木の変更は $O(n)$ になるが、並列実行向けの複数の版が存在できるデータベースを実現することができる。

3.1 破壊的木構造

破壊的木構造は親から子までのポインタとともに、子から親へのポインタを持つ。ノードの属性値を書き換える場合は、そのまま上書きすることが可能である。ノードの追加を行うときには、行う位置の親と子供の両方のポインタを変更する。この複数の変更を一つのトランザクションとして行う必要がある。変更中は親と子をロックする必要がある。平行度を上げるために、前の版の木にアクセスするためには複製を作る必要がある。一つの方法は、常に二つの木を用意し交互に変更を行うものである。変更自体は $O(1)$ だが、平行実行下では複雑な工夫が必要になる。

3.2 非破壊的木構造

データの編集を一度生成した木を上書きせず、ルートから編集を行う位置までノードをコピーする(図3.1)。この時に、子供からの親へのポインタは持たないとする。木のルートを Atomic に置き換えることで、木のアップデートを行う。変更前の木が残っているので、そのまま使用できる。変更されないノードは変更前と変更後のルートから共有されることになる。

破壊的変更と比べて、ノードの作成量は増えるが複数の版が同時に存在できる(3.2)。したがって、現在および過去の木を分散ノードまたはディスクに安全に格納することができる。実際には、木自体ではなく木の変更 Log を分散ノードまたはディスクに書き出すのが妥当である。変更の計算量は最悪 $O(n)$ になるので、大量の挿入を行うときには工夫が必要になる。

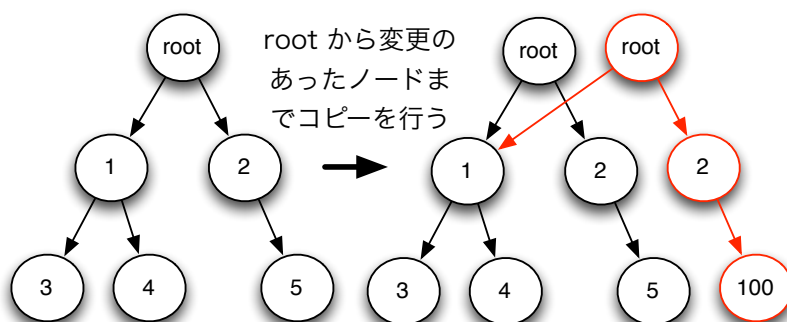


図 3.1: 非破壊的木構造の編集

木のルートの上に新しくノードを付け加える場合は $O(1)$ となる。この場合は木は追加順と逆順の構成を持つことになる。木の末端に追加する場所を覚えておき、そこに Atomic にノードを追加すると $O(1)$ で正順で追加することができる。この場合は木が破壊的に変更されているように見えるが、前の版の末端部分を超えてアクセスすることがなければ複数の版を同時に使用することができる。これを Differential Tree と呼ぶ。

木構造自体をバランス木、例えば赤黒木とすることもできる。これにより、 $O(\log n)$ で木の変更を行うことができる。しかし、この場合木構造自体を自由に構成することはできないが、赤黒木の Key を用いて任意のノードに $O(\log n)$ でアクセスすることができる。

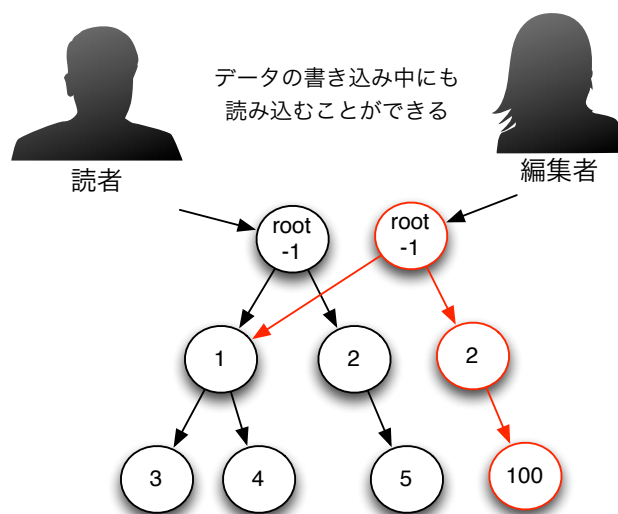


図 3.2: 非破壊的木構造による利点

また、過去の木を保持する場合、破壊的木構造は、木の複製後編集を行う必要がある。一方、非破壊的木構造は過去の木を上書きしないため、過去の木のルートノードを保持す

るだけで良い。

第4章 Jungleの構成要素

本章では、Jungleの構成要素について説明する。Jungleは名前を用いて木の生成と木の取得を行う。木の中の特定のノードにアクセスするには、木の中のノードの位置を表すNodePathを用いる。木の変更は非破壊的に行われ、木のルートを変更後の木に置き換えるAtomicOperationがトランザクションとなる。木の変更はLogとして記録され、分散ノードあるいはディスクに格納される。JungleのAPIは、Eitherを返すようになっており、Eitherの型をチェックすることにより成功と失敗がわかるようになっている。

4.1 木の生成

Jungleにおける木の生成について述べる。Jungleは複数の木構造を、名前を利用して作成・編集・削除を行い管理している。Jungleクラスが提供している木の生成・管理を行うAPIを表4.1に記す。

表 4.1: Jungleに実装されているAPI

JungleTree createNewTree(String treeName)	Jungleに新しく木を生成する。木の名前が重複した場合、生成に失敗しnullを返す。
JungleTree getTreeByName(String treeName)	JungleからtreeNameと名前が一致するtreeを取得する。名前が一致するTreeがない場合取得は失敗しnullを返す。

4.2 JungleTree

Jungleは複数の木の集合で出来ている。Jungleの木は、自身の情報をTreeContextという木構造のデータを持つ(表4.2)オブジェクトに保持している。

表 4.2: TreeContext が保持している値

自身のルートノード
1つ前のバージョンの TreeContext
編集のログ
木の uuid
木の名前
木の version
木の検索に使う Traverser

Jungle の木は、自身の木構造に編集を行う Editor や 検索に使用する Traverser を提供しており、ユーザーはそれを用いて木構造にアクセスする。また、過去のバージョンの木に対するアクセスや、特定のノードの Path を検索する機能も持っている。以下に JungleTree クラスが提供している API(表 4.3) を記す

表 4.3: JungleTree に実装されている API

TreeNode getRootNode()	木のルートノードを取得する。
long revision()	木のバージョンを取得する。初めは 0 から始まり、木への変更が Commit される度に 1 上昇する。
JungleTreeEditor getJungleTreeEditor()	木へ変更を加える Editor を取得する。
Either<Error, JungleTree> getOldTree(long revision)	引数で指定した int revision に等しいバージョンの木を取得する。
InterfaceTraverser getTraverser()	木の検索を行う Traverser を取得する。
Either<Error, TreeNode> getNodeOfPath(NodePath path)	NodePath で指定した位置と値なるノードを取得する。
NodePath getNodePath(TreeNode node)	引数で渡したノードの位置を表す NodePath を返す。

4.3 Either

Jungle は、失敗する可能性のある関数では戻り値を Either<A, B> に包んで返す。A には Error、B には処理に成功した際の戻り値の型が入る。Either は、A か B どちらかの値しか持たない。以下に Either クラスが提供している API(表 4.4) を記す。

Either<A, B> の使い方は、isA() を用いて関数が Error を返していないかを調べる。Error でない場合は b() で関数の戻り値を取得する。

表 4.4: Either に実装されている API

<code>boolean isA()</code>	Either が A を持っているかどうかを調べる。持っている場合 <code>true</code> を返す。
<code>boolean isB()</code>	Either が B を持っているかどうかを調べる。持っている場合 <code>true</code> を返す。
<code>A a()</code>	A の値を取得する。
<code>B b()</code>	B の値を取得する。

4.4 TreeNode

Jungle の木構造は、複数のノードの集合で出来ている。ノードは、自身の子のリストと属性名と属性値の組でデータを持つ。ノードに対するアクセスは、表 4.5 に記述されている API を用いて行われる。

表 4.5: TreeNode に実装されている API

<code>Children getChildren()</code>	ノードの子供を扱う <code>Children</code> オブジェクトを返す
<code>Attribute getAttribute()</code>	ノードが保持しているデータを扱う <code>Attribute</code> オブジェクトを返す。

`Children` クラスは表 4.6 に記述された API を、`Attribute` クラスは表 4.7 に記述された API を提供する。これらを利用しノードが保持している値や、子供にアクセスする。

表 4.6: Children に実装されている API

<code>int size()</code>	子供の数を返す。
<code><Either Error,TreeNode> at(int num)</code>	ノードが持つ子供の中から、変数 <code>num</code> で指定された位置にある子ノードを返す。存在しない位置を指定した場合、 <code>Error</code> を返す。

表 4.7: Attribute に実装されている API

ByteBuffer get(String key)	ノードが持つ値から、属性名 key とペアの属性値を ByteBuffer 型で返す。ノードが保持していない key を渡した場合エラーを返す。
String getString(String key)	ノードが持つ値から、属性名 key とペアの属性値を String 型で返す。ノードが保持していない key を渡した場合エラーを返す。

以下にルートノードの 2 番目の子供から、属性名 name とペアになっている属性値を取得するサンプルコード 4.4 を記述する。

```

1 JungleTree tree = jungle.getTreeByName("TreeName");
2 TreeNode root = tree.getRootNode();
3 Children children = root.getChildren();
4 Either<Error,TreeNode> either = children.at(1);
5 if (either.isA())
6     return either.a();
7 TreeNode child = either.b();
8 Attribute attribute = child.getAttribute();
9 String value = attribute.getString("name");

```

ソースコード 4.4 の説明を行う。

1 行目で Jungle から木を取得し、2 行目で、取得した木のルートノードを取得している。3 - 7 行目でルートノードの 1 番目の子ノードを取得し、8 - 9 行目で、ルートの 1 番目の子ノードから、属性名 "name" とペアの属性値を取得している。

4.5 NodePath

Jungle では、木のノードの位置を NodePath クラスを使って表す。NodePath クラスはルートノードからスタートし、対象のノードまでの経路を数字を用いて指し示す。また、ルートノードは例外として -1 と表記される。NodePath クラスを用いて < -1, 1, 2, 3 > を表している際の例を図 4.1 に記す。

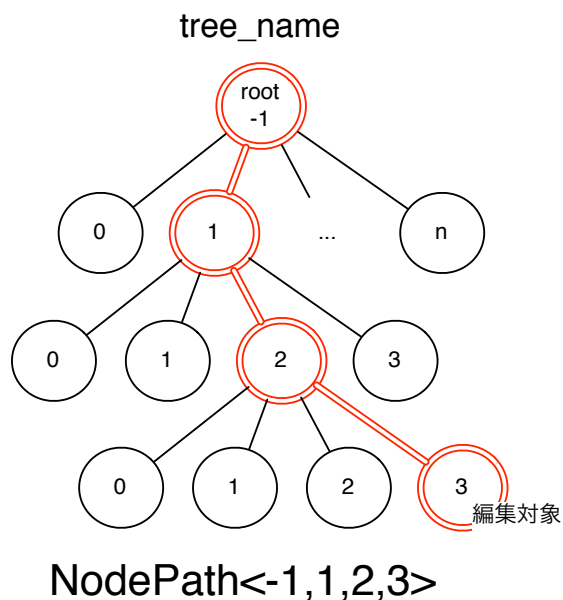


図 4.1: NodePath

4.6 木の編集 API

Jungle の木の編集は Default Jungle Tree Editor クラスを用いて行われる。Default Jungle Tree Editor クラスは、木に対する編集を行う API が定義されている Jungle Tree Editor インターフェースを実装している。Default Jungle Tree Editor は、Jungle Tree から、`getTreeEditor()` を用いて取得する。表 4.8 に Jungle Tree Editor インターフェースに定義されている API を記述する。また、表 4.8 に記述している API は全て、`Either<Error, JungleTreeEditor>` を返す。

表 4.8: Editor に実装されている API

<code>addChildAt(NodePath path, int pos)</code>	path で指定したノードの pos 番目の子の後にノードを追加する。
<code>deleteChildAt(NodePath path, int pos)</code>	path で指定したノードの pos 番目の子ノードを削除する。
<code>putAttribute(NodePath path, String key, ByteBuffer value)</code>	path で指定したノードに属性名 key 属性値 value のペアで値を挿入する。
<code>deleteAttribute(NodePath path, String key)</code>	path で指定したノードが持つ、属性名 key とペアで保存されている属性値を削除する。
<code>moveChild(NodePath path, int num, String move)</code>	path で指定したノードの num 番目の子供を move の方向に移動させる。
<code>pushPop()</code>	ルートノードの上に新しいルートノードを追加する。線形の木を作る際に使用することで木の変更の手間を $O(n)$ から $O(1)$ にできる。
<code>success()</code>	木へ行った変更をコミットする。自分が編集を行っていた間に、他の JungleTreeEditor クラスによって木が更新されていた場合、コミットは失敗する。

編集後に返される Default Jungle Tree Editor クラスは、編集後の木構造を保持しているため、編集前の木構造を保持している Default Jungle Tree Editor クラスとは別のオブジェクトである。編集を行った後は、関数 `editor.success()` で今までの編集をコミットすることができる。他の Default Jungle Tree Editor クラスによって木が更新されていた場合はコミットは失敗し、`success()` は Error を返す。その場合は、木の編集を最初からやり直す必要がある。

以下に JungleTreeEditor クラスを用いて、木の編集を行うサンプルコードを記述する。

```

1 JungleTreeEditor editor = tree.getTreeEditor();
2 DefaultNodePath editNodePath = new DefaultNodePath();
3 Either<Error, JungleTreeEditor> either = editor.addChildAt(editNodePath, 0);
4 if (either.isA())
5     return either.a();
6 editor = either.b();
7 editor.success();

```

1 行目で、木から Editor を取得している。2 行目で、編集を行うノードの Path を作成している。Default Node Path は、生成時はルートノードを指しているため、今回はルートノードに対する変更であることがわかる。3 行目で、実際にルートノードに対して、子ノードの追加を行っている。4 行目以降で、編集が成功したかどうかを Either を使って確かめ、成功していた場合 7 行目で変更を木に Commit している。

また、木に対して行われた変更は、Log として書き出される。

4.7 Commit

Jungle Tree Editor を用いて木に変更を加えた後は、Commit を行う必要がある。Commit は、前節で記述した Jungle Tree Editor が持つ関数 `success()` を使用することで行われる。

Commit を行うと、Jungle Tree Editor は、保持している編集後の木構造のデータを持つ `TreeContext` を作成する。そして、編集前の木が持つ `TreeContext` と新しく作った `TreeContext` を置き換えることで Commit は完了する。`TreeContext` の置き換えは、編集を行っている他の Thread と競合した際に、木の整合性を保つために以下の手順で行われる。

1. 新しく作った `TreeContext` が持つ、1つ前のバージョンの `TreeContext` と、編集前の木構造の `TreeContext` を比較する。
2. 一致しなかった場合、他のが Thread が Commit をすでに行っているため、失敗する (競合に負けた)。
3. 一致した場合、`TreeContext` を Atomic に入れ替える (競合に勝った)。

競合に負けた場合は、新しい木に対してもう一度同じ変更を行う必要がある。これらの API により、Jungle は木構造を格納、編集する機能を持っている。

4.8 Log

Jungle は、Editor を用いて木に編集を加える際、使用した API に応じて対応する `NodeOperation` を作成する。`NodeOperation` は `NodePath` とペアで扱わなければならない、このペアを `TreeOperation` という。Jungle によるデータの編集は `TreeOperation` が複数集まった単位で commit されていく。この `TreeOperation` の集まりを `TreeOperationLog` という。`TreeOperationLog` の仕様をソースコード 4.1 に示す。

ソースコード 4.1: `TreeOperationLog` の仕様

```
1 public interface TreeOperationLog extends Iterable<TreeOperation>
2 {
3     public TreeOperationLog add(NodePath _p, NodeOperation _op);
4     public TreeOperationLog append(TreeOperationLog _log);
5     public int length();
6 }
```

`Iterable<TreeOperation>` を継承しているため `Iterator` により `TreeOperation` を取り出せるようになっている。`add` や `append` メソッドを使って `TreeOperation` を積み上げていくことができる。積み上げた Log をディスクに書き出すことで、Jungle は永続性を持つ。分散版 Jungle では、Log を他ノードに送ることで、データの分散を行う。

4.9 検索 API

これまでに紹介した API により、Jungle は木構造を構築できるようになった。しかし、木に問い合わせを行う検索 API が実装されていなかったため、木の走査を行う Interface `Traverser` クラス内に、lambda 式を用いて実装した。以下に検索を行う関数 `find` の定義を記述する。

```
1 public Iterator<TreeNode> find(Query query, String key, String searchValue);
```

関数 `find` は、第一引数には、探索の条件を記述する関数 `boolean condition(TreeNode)` を定義した `Query` を、第二、第三引数には、`Index` を用いた絞込に使用する `String key`、`String value` を取り、条件に一致したノードの `Iterator` を返す。関数 `find` の使用例を以下に記す。

```
1 InterfaceTraverser traverser = tree.getTraverser(true);
2 Iterator<TreeNode> resultNodeIterator = traverser.find((TreeNode node) -> {
3     String name = node.getAttributes().getString("name");
4     if (name == null) return false;
5     return name.equals("kanagawa");
6 }, "belong", "ryukyu");
```

上記コードについて解説する。

1 行目で検索対象の木から、検索に使用する Interface `Traverser` を取得する。2 行目で、検索を行う関数 `find()` を使用する。今回は、`Index` を使って、属性名 "belong" 属性値 "ryukyu" を持つノード取得し、3 - 5 行目で定義されたクエリに渡す。そして、クエリの中で属性名 "name" 属性値 "kanagawa" を持つノードかどうかを確かめる。

結果として、属性名 "belong" 属性値 "ryukyu" と、属性名 "name" 属性値 kanagawa の 2 つのデータを持つノードの `Iterator` が取得できる。検索の際に使用する `Index` の実装については次節に記述する。

4.10 Index

Jungle は、非破壊的木構造というデータ構造上、過去の版の木構造を全て保持している。よって、全ての版に独立した `Index` が必要となるため、前の版の `Index` を破壊すること無く、`Index` を更新する必要がある。既存の `TreeMap` では、一度 `Index` の複製を行ない、その後更新する必要があるため、`Index` の更新オーダーが $O(n)$ となっていた。その問題を解決するため、Java 上で関数型プログラミングを行えるライブラリである、Functional Java の `TreeMap` を使用し、それを用いて `Index` の実装を行った。この `TreeMap` は、Jungle と同じようにルートから変更を加えたノードまでの経路の複製を行い、データの更新を行った際、前の版と最大限データを共有した新しい `TreeMap` を作成する。Jungle との違いは、木の回転処理が入ることである。これにより複数の版全てに対応した `Index` をサポートすることが可能になった。以下に Jungle における `Index` の型を記述する

```
1 TreeMap<String key, TreeMap<String attribute, List<TreeNode> nodeList> index> indexMap
```

Jungle の Index は IndexMap 内に保持されている。属性名で IndexMap に get を行うと、対応した Index が取得できる。取得した Index に属性値で get を行うと、ノードのリストが返ってくる。以下に Index から属性名 name 属性値 kanagawa のデータを持つ、ノードの Iterator を取得するサンプルコードを記述する。

```
1 Optional<TreeMap<String, List<TreeNode>>> indexOp = indexMap.get("name");
2 if (!indexOp.isPresent())
3     return new NullIterator<TreeNode>();
4     TreeMap<String, List<TreeNode>> index = indexOp.get();
5     Optional<List<TreeNode>> nodeListOp = index.get("kanagawa");
6 if (!nodeListOp.isPresent())
7     return new NullIterator<TreeNode>();
8 return nodeListOp.get().iterator();
```

1 - 4 行目で IndexMap から 属性名 "name" に対する値を持つ Index を取得している。5 - 8 行目で 取得した Index から、属性名 "kanagawa" を持つノードの Iterator を取得している。

Jungle はこれらの API により、木構造を格納、編集、検索する機能を持っている。

第5章 データベース Jungle に 新たに追加した構成要素

本章では、今回効率的な木の非破壊アップデートを可能にするために追加した機能である、Jungle の中で使用する非破壊赤黒木、Index の差分アップデート、Differential Jungle Tree、Red Black Jungle Tree について大まかな説明する。

5.1 非破壊 Red Black Tree の実装

Jungle の Index は、Java 上で関数型プログラミングが行える Functional Java の非破壊 TreeMap を使って実装されていた。しかし、Functional Java は、処理が重く、実用的な性能ではなかったため、非破壊の TreeMap の実装を新しく行った。

5.2 Index の差分 Update

Jungle は、Index の更新を Commit 時に Full Update で行っている。そのため、Commit を行うたびに、 $O(n)$ の Index の Update が入り、木の編集時大きなネックとなっている。

Index の更新処理を高速に行えるようにするため、前の木との差分だけ Index を更新する機能を Jungle に追加した。

5.3 Differential Jungle Tree の実装

Jungle の木の変更の手間は木の形によって異なる。特に線形の木は、変更の手間が $O(n)$ となってしまふ。線形の木を $O(1)$ で変更するために、Jungle は、ルートノードを追加していく PushPop の機能を持つ。しかし、PushPop は木の並びが逆順になってしまうため、正順の木を構築する際には使用できない。その問題を解決するために、Differential Jungle Tree の実装を行った。Differential Jungle Tree は、木のバージョンごとに、自身の末尾のノードを保持する。

Differential Jungle Tree は、木を破壊的に更新するが、編集・検索時に、末尾ノードを使用することで、過去の木の形を残すことが可能となっている。

5.4 Red Black Jungle Tree

Jungle は、木に編集を加えた際、編集を加えたノードと、経路にあるノードの複製を取る。その為、木の編集の手間は、木の大きさにも依存している。最適なバランスの取れた木構造を構築することで、編集の手間を $n(\log N)$ にすることは可能だが、Default Jungle Tree だと、木の形をユーザーが Path を用いて、バランスを取る必要がある。しかし、ユーザーが全ての木構造の形を把握し、バランスの取れた木を構築するのは難しい。そこで、自動で木のバランスを取り、最適な木構造を構築する機能を Jungle Tree に実装した。バランスは、木の生成時に特定の Balance Key 決定し、それを使って行う。

木の探索に関しては BalanceKey を用いた場合 $N(\log n)$ 、そうでない場合は $O(n)$ で行える。

第6章 非破壊 TreeMap の実装

Jungle の Index は、Functional Java の非破壊 TreeMap を用いて実装を行っている。しかし、Functional Java の TreeMap は、木の変更の手間が大きい、並列実行時処理速度が落ちるなど、実用的な性能を持っていなかった。そのため、Jungle の性能も、TreeMap 部分がネックとなっていた。

その問題を解決するため、Jungle で使用する非破壊 TreeMap を作成した。TreeMap は、Red Black Tree のアルゴリズムを用いる。

6.1 Red Black Tree

Red Black Tree は二分探索木の一つで、以下の条件を満たした木のことである。

1. ノードは赤か黒の色を持つ。
2. ルートノードの色は黒。
3. 全ての葉は黒である。
4. 赤いノードの子は黒色である。
5. 全ての葉からルートまでのパスには、同じ個数の黒いノードがある。

Red Black Tree は、データの挿入、削除時に、上記の条件を崩さないように木のバランスを取る。この条件を守っている限り、Red Black Tree はデータの検索、削除、探索を $O(\log n)$ で行える。

6.2 非破壊 TreeMap の定義

非破壊 TreeMap は、Java のジェネリクスを用いて、`TreeMap<K,V>Key` と定義される。TreeMap を作る際に、K,V に任意の型を記述することで、Key と Value で使用する型を設定できる。ソースコード 6.1 に、Key を String 型・Value を ByteBuffer 型で定義するサンプルコードを記述する。

ソースコード 6.1: TreeMap の定義サンプル

```
1 TreeMap<String, ByteBuffer> map = new TreeMap<>();
```

6.3 非破壊 TreeMap の API

非破壊 TreeMap は、値の検索・挿入・削除を行うために、表 6.1 に記述してある API を提供している。

表 6.1: 非破壊 TreeMap に実装されている API

<code>Node getRoot()</code>	TreeMap のルートノードを返す。
<code>boolean isEmpty()</code>	TreeMap が値を保持していないなら <code>true</code> を返す。
<code>TreeMap<K,V> put(K key, V value)</code>	TreeMap に <code>key:value</code> の組で値を挿入した、新しい TreeMap を返す。
<code>TreeMap<K,V> delete(K key)</code>	TreeMap に <code>key</code> とペアで格納されている値を削除した、新しい TreeMap を返す。
<code>V get(K key)</code>	TreeMap に <code>key</code> とペアで格納されている値を返す。
<code>boolean contain(K key)</code>	TreeMap に <code>key</code> とペアで格納されている値があるなら <code>true</code> を返す。
<code>Iterator<K> keys()</code>	TreeMap が保持している全ての <code>key</code> を Iterator で返す。

非破壊 TreeMap は、`put`・`delete` を行うと編集後の新しい TreeMap を返すため、新しい TreeMap で受ける必要がある (ソースコード 6.2)。この時返ってくる `newMap` と、編集前の `map` は別オブジェクトである。

ソースコード 6.2: TreeMap の編集例

```
1 TreeMap<String,String> map = new TreeMap<>();
2 TreeMap<String,String> newMap = map.put("key","value");
```

6.4 非破壊 Red Black Tree へのデータの挿入

非破壊 Red Black Tree へのデータの挿入は、以下の手順で行われる。

1. 挿入を行うノードと、現在のノードを比較する。
2. 比較の結果、大きかった場合右に、小さかった場合左のノードに進む。
3. 挿入を行う場所にたどり着くまで、1・2を繰り返す。
4. 現在の位置に、赤色でノードを挿入する。
5. 木のバランスを取りながら、ルートまでの経路の複製を行う。その際、変更が加えられないノードへは参照を行い過去の木と最大限共有を行う。

Red Black Tree のデータ挿入時のバランスは、次の 5 パターンに分けられる。これ以降の説明では、挿入したノードは、ノード ins と記述する。

データ挿入時のバランス ケース 1

バランス時、ノード ins の位置がルートだった場合、ルートノードの色を黒に変更することで木のバランスを取る。ルートノードの色を黒に変更しても、左右の Sub Tree の黒の個数が一つ増えるだけなので、Red Black Tree の条件は守られる。

データ挿入時のバランス ケース 2

バランス時、ノード ins の親ノード B が黒だった場合、Red Black Tree の条件は崩れないため、赤色でノードを挿入して問題はない。

データ挿入時のバランス ケース 3

バランス時、ノード ins の親の親ノード A が黒かつ、ノード A の両方の子ノード B・C が赤の場合、ノード B・C を黒に、ノード A を赤に変更する (図 6.1)。

また、本章の図に表記されている四角のノードの色は、Red Black Tree の条件を守られているなら問わず、それ以下の子ノードは省略してあるものとする。

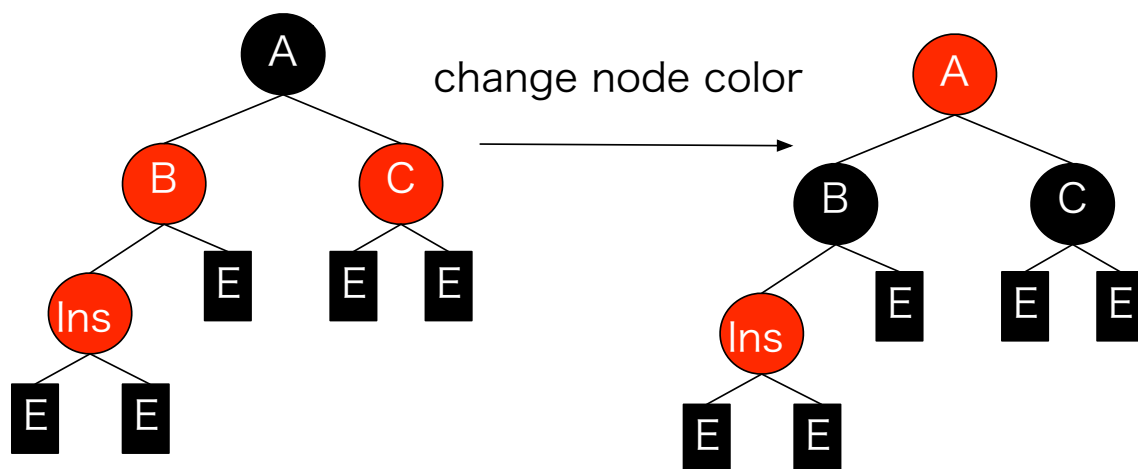


図 6.1: データ挿入時のバランス 3

バランス後、ノード A の色が変わってしまったため、ノード A を新しくノード ins とし再帰的に木のバランスを行う。この変更は最悪の場合ルートまで続く。

データ挿入時のバランス ケース 4

バランス時、ノード ins の親の親ノード A が黒かつ、ノード A のノード ins 側の子ノード B が赤かつ、逆側の子ノード C が黒かつ、ノード ins がノード B の木の中心側の子である場合、ノード B を中心に外側に回転処理を行い (図 6.2)、次節に記述するバランス 5 を行う。その際、ノード B をノード ins として扱う。

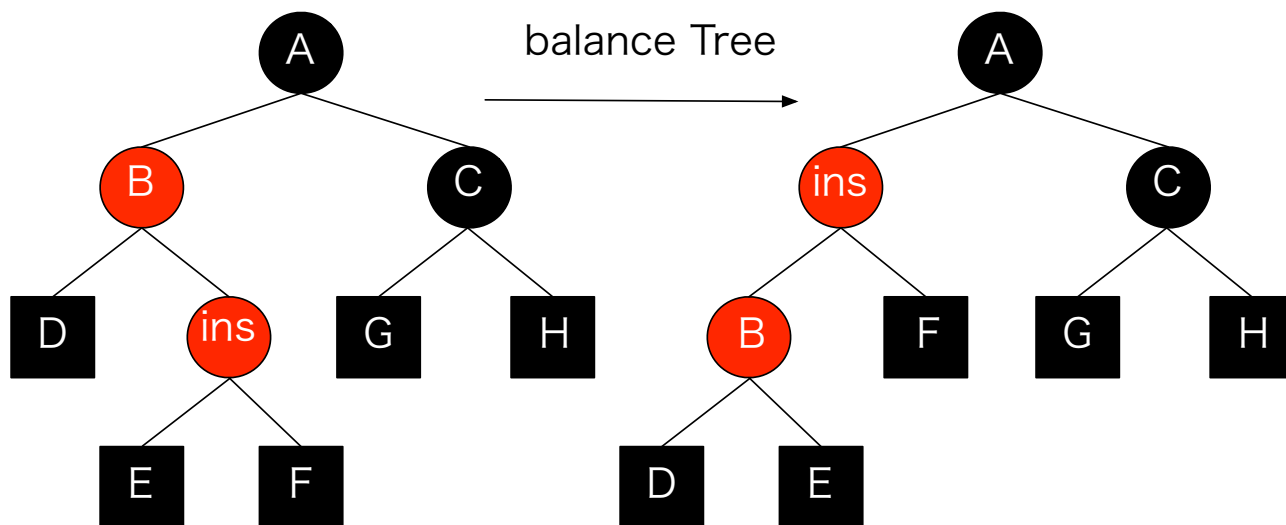


図 6.2: データ挿入時のバランス 4

データ挿入時のバランス ケース 5

バランス時、ノード ins の親の親ノード A が黒かつ、ノード A のノード ins 側の子ノード B が赤かつ、逆側の子ノード C が黒かつ、ノード ins が木の外側の子の場合、ノード A を中心にノード ins と逆側に回転処理を行い、回転後ノード A とノード B の色を入れ替える。

赤黒木は、データ挿入時にこれらの処理を行うことで、木のバランスを取る。

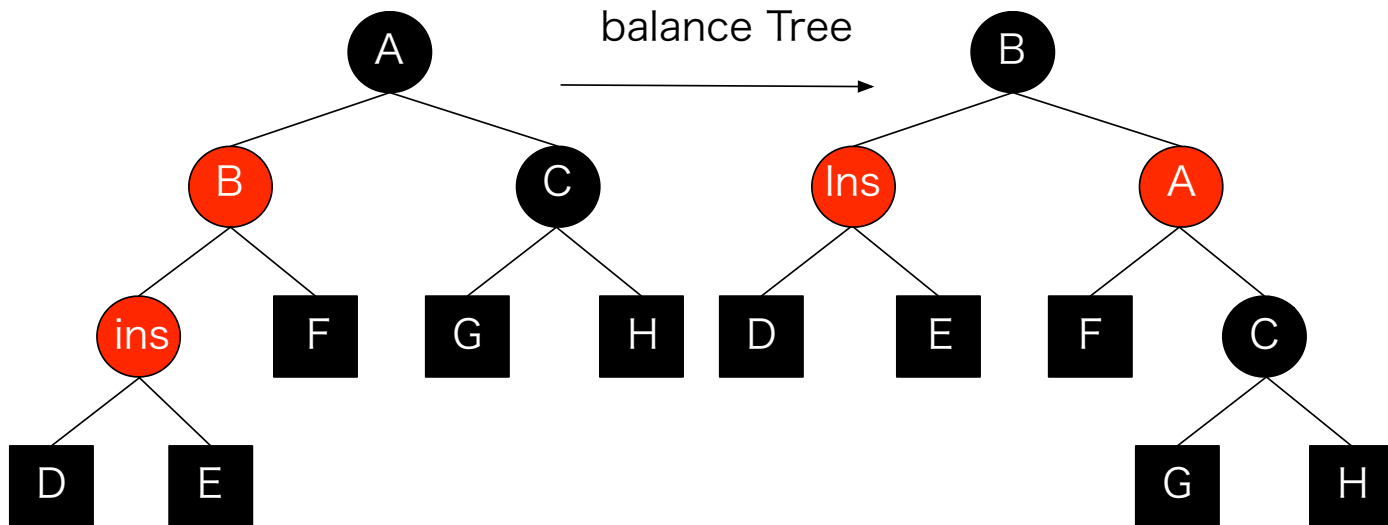


図 6.3: データ挿入時のバランス 5

6.5 非破壊 Red Black Tree のノード削除

Red Black Tree のノード削除は、以下の手順で行われる。

1. 削除を行うノードと、現在のノードを比較する。
2. 比較の結果、大きかった場合右、小さかった場合左のノードに進む。
3. 削除を行うノードにたどり着くまで、1・2を繰り返す。
4. ノードに子が無い場合削除を行う。
5. 片側に部分木 (子ノード) がある場合、ノードを削除した後、部分木のルートを上昇させる。
6. 両側に部分木 (子ノード) がある場合は、左部分木の最大の値を持つノードの値を取得し、削除を行うノードと同じ色に変更した後、置換する。
7. 置換したノードを削除する (ここで削除させるノードは部分木の最大値であるため、子ノードは1つ以下、つまり削除の手順4・5どちらかの手順で削除できる)。
8. 削除したノードから、木のバランスを取りながら、ルートまでの経路の複製を行う。その際、変更が加えられないノードへは参照を行い過去の木と最大限共有を行う。

RedBlackTree のバランスは、ノード削除時の状態によって、次の6パターンに分けられる。また、これ以降削除したノードを、ノード del と記述する。

データ削除時のバランス ケース 1

バランス時、ノード del がルートだった場合、全ての経路の黒ノード数が 1 つ減った状態で条件が成立しバランスは終了する。

データ削除時のバランス ケース 2

バランス時、ノード del が黒かつ、ノード A・B・C・D・E・F が黒の場合、ノード B を赤に変える (図 6.4)。そうすることで、ノード B・E・F 以下の黒ノードの階層が減って、ノード A 以下の木のバランスが回復する。その後、A を新たなノード del として木のバランスを行う。このバランスは最悪の場合ルートまで続き、ケース 2 で終了する。

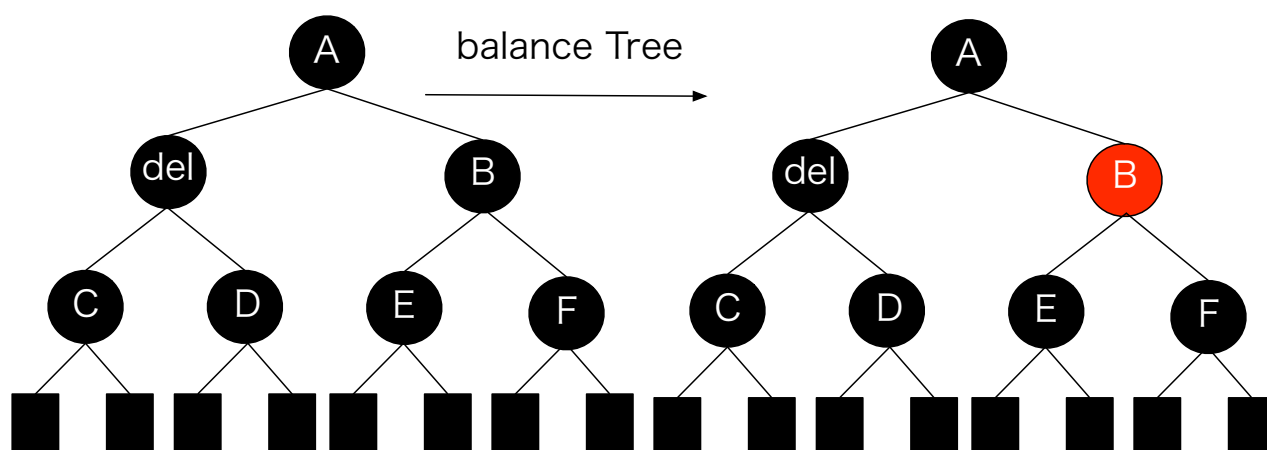


図 6.4: データ削除時のバランス 2

データ削除時のバランス ケース 3

バランス時、ノード del が黒かつ、ノード A・C・D・E・F が黒かつ、ノード B が赤の場合、ノード A を中心に外側に回転、その後ノード A を赤に、ノード B を黒に変更する (図 6.5)。その後、ノード del を基準に再び木のバランスを行う。この時のバランスは、図 6.5 における、ノード E の子供の色に応じてケース 4・5・6 のどれかに帰着する。

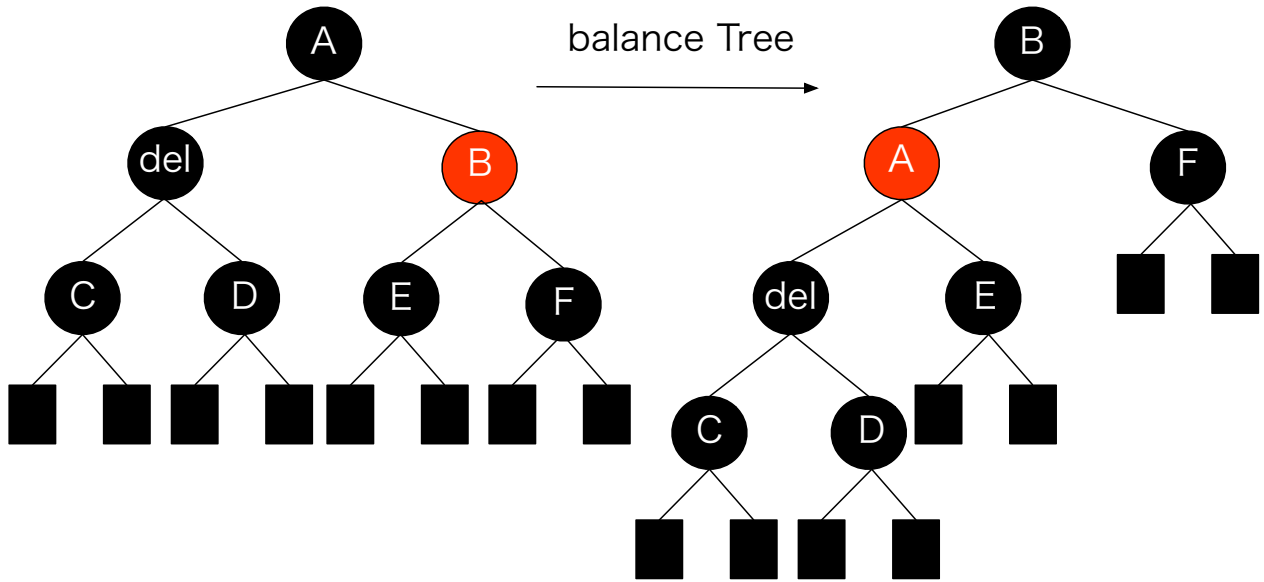


図 6.5: データ削除時のバランス 3

データ削除時のバランス ケース 4

バランス時、ノード del が黒かつ、ノード B・C・D・E・F が黒かつノード A が赤の場合、ノード A を黒に、ノード B を赤に変更する (図 6.6)。そうすることで、ノード A 側の右側の Sub Tree の黒の深さを変えることなく、左側の Sub Tree の黒の深さが 1 つ増え、バランスが取れる。

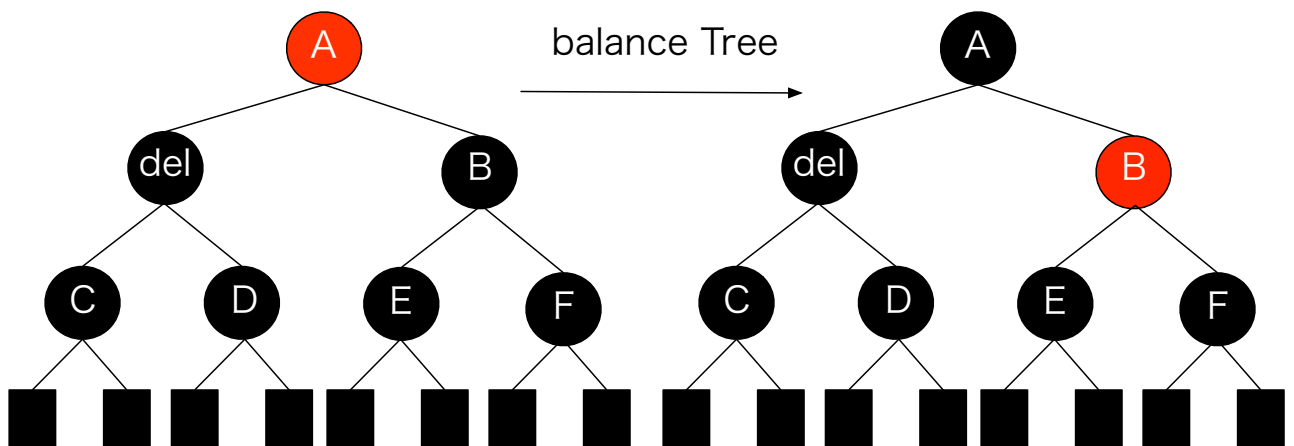


図 6.6: データ削除時のバランス 4

データ削除時のバランス ケース 5

バランス時、ノード del が黒かつ、ノード B・C・D・F が黒かつ、ノード E の色が赤の場合、ノード B を中心に外側に回転、その後、ノード E を黒に、ノード B を赤に変更する (図 6.7)。そして、データ削除時のバランス ケース 7 に帰着する。

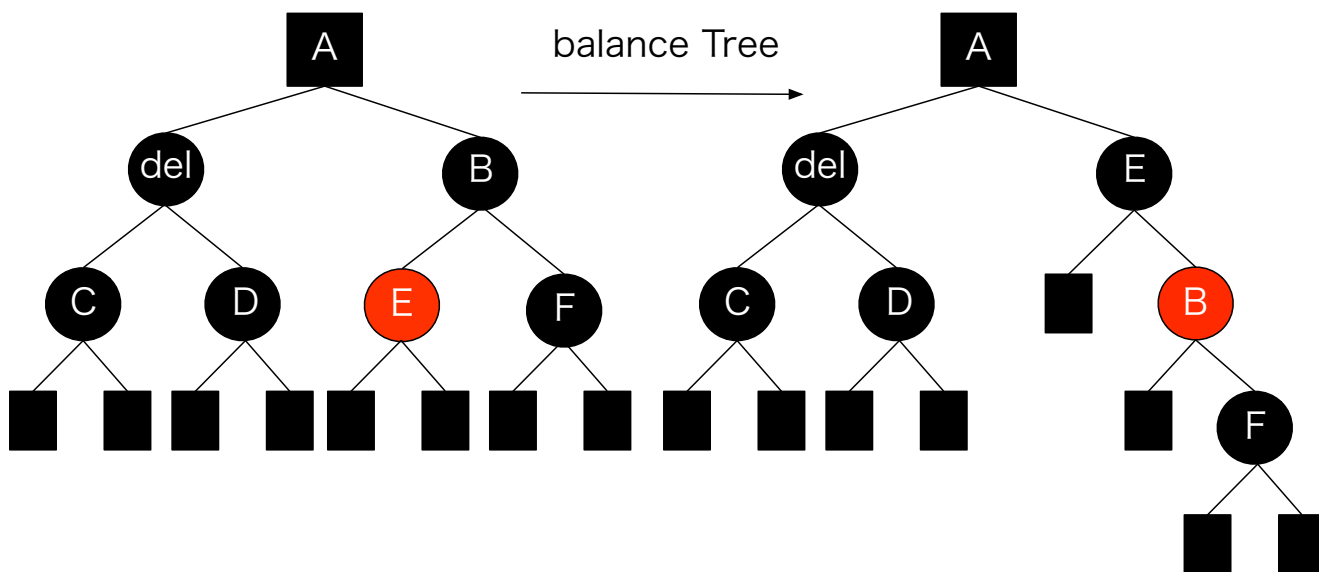


図 6.7: データ削除時のバランス 5

データ削除時のバランス ケース 6

バランス時、ノード del が黒かつ、ノード B・C・D が黒かつ、ノード F の色が赤の場合、ノード A を中心にノード del 側に回転、その後、ノード A とノード B の色を交換し、ノード F を黒にする (図 6.8)。そうすることで、ノード E・F の黒の深さを変えること無く、ノード rep の黒の深さを 1 増やせるため、木のバランスが取れる。

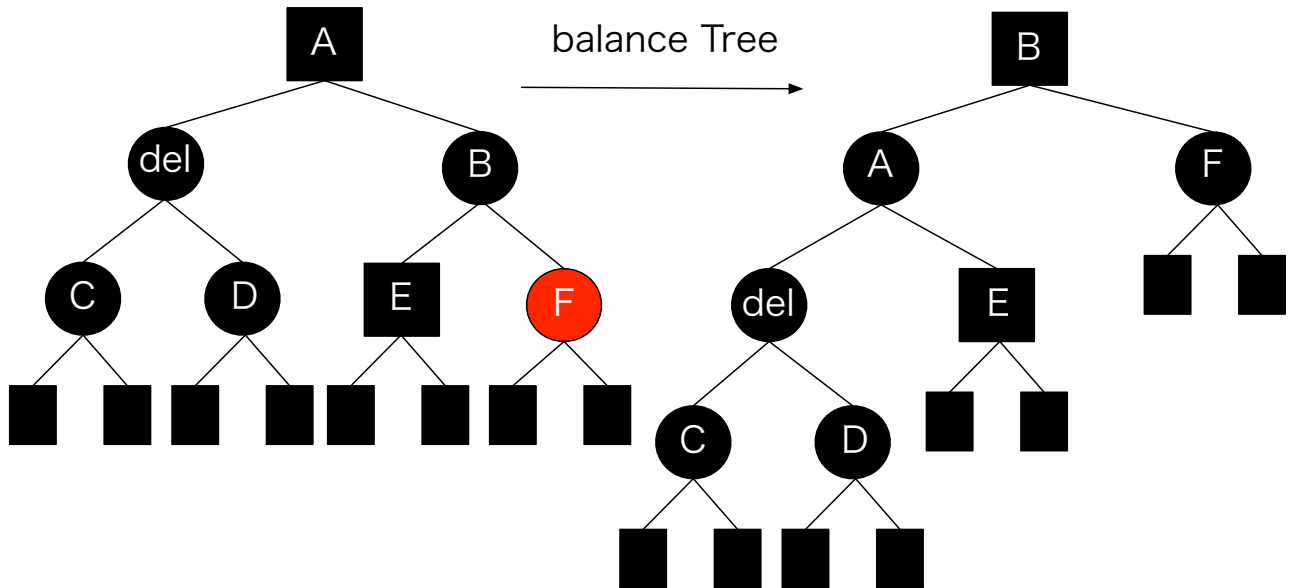


図 6.8: データ削除時のバランス 6

Red Black Tree は、削除時に上記のバランスを行うことで、木のバランスを保っている。
これらの機能を実装することで、非破壊 TreeMap は完成した。

第7章 Indexの差分Update

Jungleの木はIndexを持っており、木のCommit時にFull Updateを行っている。そのため、Commitを行うたびに $O(n)$ のIndexのUpdateが入り、木の編集時大きなネックとなっていた。なので、高速にIndexの更新を行うため、Indexの差分アップデートを実装した。

7.1 差分Updateの実装

Jungleは木の編集を行う際に、編集を行うノードと、経路にあるノードの複製を行い新しい木構造を構築するため、Indexの中には、編集後の木には存在しない複製前のノードが残ってしまう。なので、Indexの差分Updateを行う際には、それらのノードをIndexから削除して、新しく複製されたノードをIndexに登録する必要がある。

そのためには、編集を行ったノードを覚えておく必要がある。そこで、Jungle Tree Editor内に、編集を加えたノードを覚えておくためのリストを定義した。Editorは、木に編集を加えたら、リストに編集前のノードを保存する。そして、Commit時にリストにあるノードを使ってIndexの中に残っている、編集後の木に存在しないノードを削除する。その後、新しく作られたノードをIndexに登録してUpdateは終了する。

7.2 編集前のノードの削除

IndexのUpdateを行う際、初めに、編集後の木に存在しないノードをIndexから削除する。削除の対象は、変更を加えたノードと、ルートから変更を加えたノードまでの経路にあるノードである。ノードの削除は、以下の手順で行われる。

1. 編集を行ったノードのリストからノードを取得する。
2. 取得したノードが、保持している値をIndexから削除する。
3. 自身と子供のペアをParentIndexから削除する。
4. ParentIndexから親を取得する。
5. 2 - 4をルートノードにたどり着くか、ParentIndexから親を取得できなくなるまで続ける。
6. 1 - 5をリストからノードが無くなるまで続ける。

Parent Index に現在のノードが登録されていない場合は、現在のノードからルートまでの経路にあるノードは Index から削除されていることが保証されているため、削除を終えて、リストに入っている次のノードの削除処理を行っても構わない。

7.3 Index へのノードの挿入

Index から不要なノードを削除した後は、木の編集時新しく作られたノードを Index に挿入する。ノードの挿入は、以下の手順で行われる。

1. 木からルートノードを取得する。
2. 取得したノードが Index に登録されているかを調べる。
3. 登録されている場合、そのノード以下の Sub Tree は、全て Index に登録されているので、次のノードに移動する。
4. 登録されていなかった場合、自身が保持している値を Index に登録する
5. 自身と子ノードを Parent Index に登録する。
6. 自身の子ノードを取得したノードとして 2 に戻る。
7. 全てのノードを登録したら終了する。

7.4 Full Update との使い分け

Index の差分 Update は、不要なノードの削除と新しく木に追加されたノードの挿入を行っているため、1 ノードに対する処理は Full Update より大きい。少ない回数編集を行った後の Commit は、差分 Update の方が高速に行えるが、多くの編集を行った後の Commit だと、Full Update の方が高速に動作する可能性がある。これに関しての検証は、性能測定 の章に記述する。

第8章 Differential Jungle Tree

Jungle は木の編集時、ルートから編集を行う位置までのノードの複製を行う。そのため、木の編集の手間は、木構造の形によって異なる。特に線形の木は、全てのノードの複製を行うため、変更の手間が $O(n)$ になってしまう。そこで、Jungle は、線形の木を $O(1)$ で変更する PushPop の機能を持つ。PushPop とは、ルートノードの上に新しいルートノードを付け加える API である (図 8.1)。すると、木の複製を行う必要が無いため、木の変更の手間が $O(1)$ でノードの追加を行える。

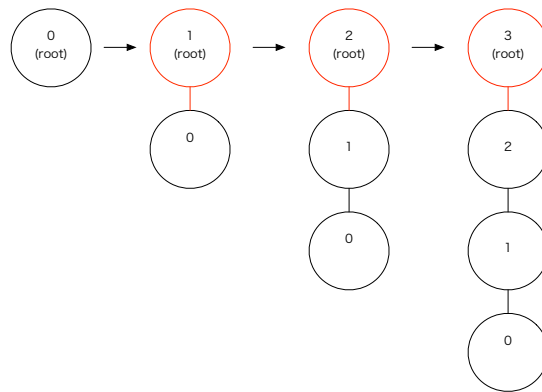


図 8.1: PushPop

しかし、PushPop はルートノードを追加していくため、図 8.1 のようにノードの並びが逆順になってしまう。Log などの正順の木でなければデータを表現できない場合、木の編集時 PushPop を使用できない。その問題を解決するために、木の編集の手間を $O(1)$ で、正順の木を構築できる Differential Jungle Tree の実装を行った。Differential Jungle Tree は、木のバージョンごとに、自身の木の最後尾を表す末尾のノードを保持する。

8.1 Differential Tree Context

Jungleの木はTreeContextというオブジェクトに自身の木の情報を保持している。Differential Jungle Treeでは、現在の版の木構造の末尾ノード保持することが可能なDifferential Tree Contextを作成した。

8.2 Differential Jungle Treeの作成

Differential Jungle Treeを作成するためにJungleに、新しいAPIを実装した(表8.1)。

表 8.1: Jungleに新しく実装したAPI

<code>createNewDifferenceTree(String treeName)</code>	Jungleに新しくDifferential Jungle Treeを生成する。木の名前が重複した場合、生成に失敗しnullを返す。
---	--

ソースコード 8.1に新しいDifferential Jungle Treeを作成するサンプルコードを記載する。

ソースコード 8.1: Differential Jungle Treeの生成

```
1 Jungle jungle = new DefaultJungle(null, "hoge", new DefaultTraverser());
2 String treeName = "difTree";
3 JungleTree tree = jungle.createNewDifferenceTree(treeName);
```

Jungleでは、`TreeMap<String, Jungle Tree>`を用いてJungle Treeを管理している。Differential Jungle TreeとDefault Jungle Treeは、同じTreeMapに保持されるため、別々の木に同じ名前をつけることはできない(ソースコード8.2)。

ソースコード 8.2: 名前の重複

```
1 Jungle jungle = new DefaultJungle(null, "hoge", new DefaultTraverser());
2 String treeName = "treeName";
3 JungleTree defaultTree = jungle.createNewTree(treeName);
4 JungleTree dfTree = jungle.createNewDifferenceTree(treeName);
```

ソースコード8.2では、4行目でDifferential Jungle Treeの名前が、3行目で生成したDefault Jungle Treeの名前と重複するため、木の生成に失敗する。

8.3 末尾ノードを使用した木の編集

Differential Jungle Treeの木の編集は、Differential Jungle Tree Editorを使用して行う。Differential Jungle Tree Editorは、Default Jungle Tree Editorと違い、生成時に新しい木構造(Sub Tree)を自身の中に構築する。そして、木の編集は、自身が保持している木

構造に対して行う。編集後、Commit を行う際に構築した木構造を、Differential Jungle Tree の末尾ノードに Append する。その際木の複製は行わない。

また、Differential Tree は自身が保持している木構造に対する変更しか行えないため、一度 Commit した木に対して変更は行えない。図 8.2 に Differential Jungle Tree の編集の流れを記述する。

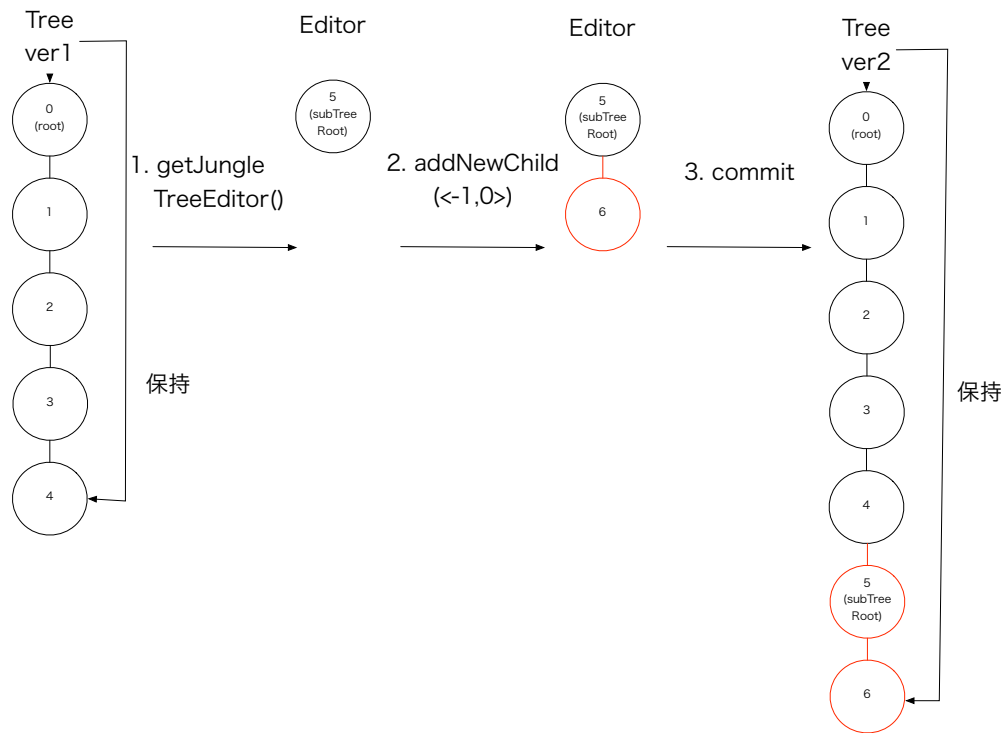


図 8.2: 末尾ノードを使用した木の編集

1. 木から `getJungleTreeEditor` で Editor を取得する。(このとき Editor は新しい木構造 (Sub Tree) を持つ)。
2. Editor が保持している木構造に対して `addNewChild(<-1,0>)` を実行し、ノードの追加を行う。
3. Commit を行い、Tree の末尾ノードに Editor が保持している木構造を Append する。

Editor が保持している木構造に最後に追加したノードが、新しい木の末尾ノードとなる。また、Differential Jungle Tree は、木の編集時複製を行わないため、Index のアップデートは、Editor が保持している木構造のデータを Index に追加するだけで良い。

8.4 Differential Jungle Tree の検索

Differential Jungle Tree は、末尾ノードを使って、現在の木構造を表現している。なので、過去の木に対して、Index を使わずに全探索を行った場合、その版の木には無いはずのノードが取得できてしまう。例として、編集前の木である Tree ver1 と編集後の木である Tree ver2 があるとする (図 8.3)。ここで、Tree ver1 に対して、検索を Index を使わずに行った場合、本来 Tree ver1 に存在しないノード 3・4 も検索対象に含まれてしまう。

そこで、その版の木が持つ末尾ノード以下の Sub Tree を検索対象から除外する、Differential Interface Traverser を実装した。Differential Interface Traverser を用いて Index を使用せず木の全探索を行った場合、Tree ver1 に存在しないノード 3・4 は検索対象から省かれる。

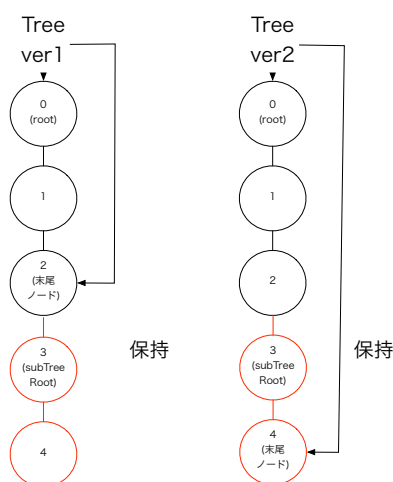


図 8.3: 複数の版の木の表現

Index を使用した検索を行う場合、各版の木に対応した Index があるため、Default Tree と検索のアルゴリズムは変わらない。これらの実装により Differential Jungle Tree は木構造の構築・検索を行う。

8.5 Differential Jungle Tree の整合性

Default Jungle Tree への Commit は、編集後の木のデータを持つ TreeContext を作り、編集前の木が持つ TreeContext と Atomic に入れ替えることで行われる。しかし、Differential Jungle Tree の Commit は、Default Jungle Tree の Commit と異なり、TreeContext の入れ替えと、Editor が保持している木構造の末尾ノードへの Append の 2つのプロセスからなる。

TreeContext の入れ替えに関しては、Default Jungle Tree と同じように行い、末尾ノードへの Editor が持っている木構造の Append は、TreeContext の入れ替えが成功した後に

行う。そうすることで、別 Thread で行われている Commit と競合した際に、TreeContext を入れ替えた Thread と別 Thread が Append を行い、木の整合性が崩れることを回避している。

また、過去の版の木に対して、編集を加え Commit を行った場合、木の整合性が崩れてしまう問題もある。図 8.3・8.4 を例に解説する。図 8.3 の過去の版の木 Tree ver1 に新しいノード 5 を追加・Commit を行うと、新しい木 Tree ver`2 が構築される。ここで、Tree ver`2 に対して Index を使用しないで検索を行う。Differential Jungle Tree に対する Index を使用しない検索は、末尾ノードより上にあるノードを検索対象にする。しかしノード 3・4 という、本来存在しないはずのノードが検索対象に含まれてしまう。これは、過去の版の木である、tree ver1 の末尾ノードが 2 つ子ノード持っているせいで発生する。

この問題を解決するために、Differential Jungle Tree では、過去の木に対する変更を禁止している。具体的には、末尾ノードは子を 1 つしか持つことができないようにした。そうすることで木の整合性を保証している。

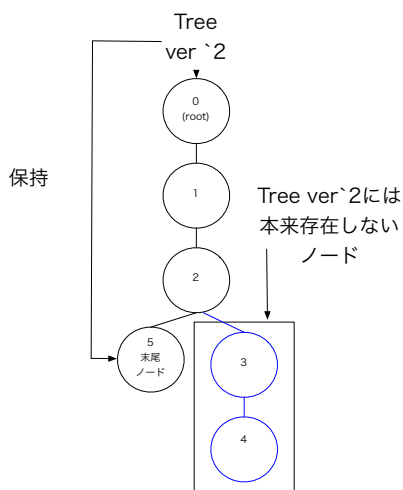


図 8.4: 木の整合性が崩れる例

第9章 Red Black Jungle Tree

Jungle は木に編集を加えた際、ルートから編集を行う位置までのノードをコピーする。その為、木の編集の手間は、木の大きさにも依存している。最適なバランスの取れた木構造を構築することで、編集の手間を $n(\log N)$ にすることは可能だが、Default Jungle Tree の場合、ユーザーが Path を用いて、バランスを取りながら木を構築する必要がある。しかし、ユーザーが全ての木構造の形を把握し、バランスの取れた木を構築するのは困難である。そこで、自動で木のバランスを取り、最適な形の木構造を構築する機能を持つ Red Black Jungle Tree に実装した。バランスは、木の生成時に特定の Balance Key 決定し、それを使って行う。木のバランスを取るアルゴリズムは、前述した非破壊 TreeMap と同じものを使用する。

しかし、木の編集を加えた際、木がどのようにバランスを取るか予想するのは困難であるため、木の構造で、組織構造等のデータを表現するのは難しい。なので、この機能が使えるのは、木の構造自体がデータを表現していない場合に限る。

また、自身の木構造が、Balance Key を使った Index と同じ働きを持つため、木の Commit 時に別途 Index を構築する必要が無い、といったメリットもある。

9.1 Red Black Jungle Tree の作成

Red Black Jungle Tree を作成するため、Jungle に新しい API を実装した (表 9.1)。

表 9.1: Jungle に新しく実装した API

<code>createNewRedBlackTree(String treeName, String balanceKey)</code>	Jungle に新しく Red Black Jungle Tree を生成する。第一引数に木の名前、第二引数に木のバランスを取る時に使用する Balance Key を受け取る。木の名前が重複した場合、生成に失敗し null を返す。
--	---

`createNewRedBlackTree` を使用したサンプルコード 9.1 を記述する。

ソースコード 9.1: Red Black Jungle Tree の生成

```
1 Jungle jungle = new DefaultJungle(null, "hoge", new DefaultTraverser());
2 String treeName = "redBlackTree";
3 String balanceKey = "balanceKey";
4 JungleTree tree = jungle.createNewRedBlackTree(treeName, balanceKey);
```

サンプルコード 9.1 では、3 行目で指定した `balance Key` を用いて木のバランスを取る、Red Black Jungle Tree が構築される。

9.2 NodePath の拡張

Red Black Jungle Tree は、ノードを追加・削除するたびに木のバランスが行われ、各ノードの Path が変わってしまう。その為、数字を使った NodePath では、編集を加える際、編集対象のノードの Path を毎回調べる必要がある。その問題を解決するために、NodePath を拡張した Red Black Tree Node Path を作成し 属性名 `BalanceKey` 属性値 `value` のペアでノードを指定できるようにした。RedBlackTreeNodePath は、引数に String 型の `BalanceKey` と `ByteBuffer` 型の `value` を取る。ソースコード 9.2 に、属性名 `"balanceKey"` 属性値 `value` を持つノードを指定する Red Black Tree Node Path を作成するサンプルを記述する。

ソースコード 9.2: Red Black Tree Node Path の生成

```
1 String balanceKey = "balanceKey";
2 ByteBuffer value = ByteBuffer.wrap(("value").getBytes());
3 NodePath path = new RedBlackTreeNodePath(balanceKey,value);
```

9.3 Red Black Jungle Tree の編集

Red Black Jungle Tree Editor は、既存の Jungle Tree Editor とくらべて API の使い方が異なる。表 9.2 に Default Jungle Tree Editor と Red Black Jungle Tree Editor の API の使い方の違いを記述する。

表 9.2: Red Black Jungle Tree と Default Jungle Tree の API の違い

<pre>Either<Error, JungleTreeEditor> addChildAt(NodePath path, int pos)</pre>	<p>前節に記述した Red Black Node Path を使用する。Path 生成時に指定した、Key と Value の組のデータを持つノードを Red Black Tree の挿入アルゴリズムに則って挿入し、バランスを取る。また、</p>
<pre>Either<Error, JungleTreeEditor> addChildAndPutAttribute (NodePath path, int pos, String key, ByteBuffer value)</pre>	<p>path と pos は使用せず、属性名 key 属性値 value を持ったノードを、Red Black Tree の挿入アルゴリズムに則って挿入し、バランスを取る。第二引数に BalanceKey 以外の値を渡した場合、バランスが取れない為、ノードの挿入は失敗する。</p>
<pre>Either<Error, JungleTreeEditor> replaceNewRootNode()</pre>	<p>赤黒木では使用しない。実行した場合エラーを返す。</p>
<pre>Either<Error, JungleTreeEditor> moveChild(NodePath path, int childNum, String move)</pre>	<p>ノードを動かすと木のバランスが崩れるため使用しない。実行した場合エラーを返す。</p>

Red Black Jungle Tree にノードを挿入するサンプルコードを記載する (ソースコード 9.3)。

ソースコード 9.3: RedBlackJungleTree の編集例

```
1 String balanceKey = "balanceKey";
2 ByteBuffer value = ByteBuffer.wrap(("Elphelt").getBytes());
3 JungleTree tree = jungle.createNewRedBlackTree("TreeName", balanceKey)
4 JungleTreeEditor editor = tree.getJungleTreeEditor();
5 NodePath path = new RedBlackTreeNodePath(balanceKey,value));
6 Either<Error, JungleTreeEditor> either = editor.addChildAt(path,0);
7 if (either.isA()) return either.a();
8 editor = either.b();
9 either = editor.success();
```

ソースコード 9.3 の解説を以下に記す。

1 - 2 行目で、挿入するノードが持つ 属性名 balanceKey と属性値 value を作成する。3 行目で、木の名前が"TreeName" バランスを balanceKey を使って行う Red Black Jungle Tree を作成する。4 行目で、Editor を取得し、5 行目で Path 作成している。6 行目で、Path で指定した属性名 balanceKey 属性値 value の組の値を持つノードを木に挿入している。そして 9 行目で、今回行った変更を Commit して編集を終了している。

Red Black Jungle Tree は、木の編集時 Index を更新しないので、Default Jungle Tree より高速に木の変更を行える。

9.4 Jungle Red Black Tree の検索

Red Black Jungle Tree への検索は、Red Black Jungle Tree Interface Traverser が提供している API を用いて行う (表 9.3)。

表 9.3: Red Black Jungle Tree Interface Traverser が提供している API

<pre>Iterator<TreeNode> find(Query query, String Balancekey, String BalanceValue)</pre>	<p>第二引数と第三引数で指定した値を持ち、Query の条件と一致するノードを返す。BalanceKey と BalanceValue を用いて木の二分探索を行うので、探索オーダーは $O(\log n)$ である。また第二引数に、木の生成時指定した、Balance Key 以外を渡した場合、探索は失敗する。</p>
<pre>Iterator<TreeNode> find(Query query)</pre>	<p>Query の条件と一致するノードを、木の全探索で検索する。探索オーダーは $O(n)$ である。</p>

Red Black Jungle Tree は、これらの実装により、木構造の構築・検索を行える。

第10章 Jungleを使ったアプリケーション

本章では、Jungle を使用した例題アプリケーションを記述する。

10.1 Jungle Tree ブラウザ

Jungle の木に対する変更において、JungleTreeEditor クラスを用いる方法はプログラム上では便利だが、手動で変更するのには向いていない。よって、組み込み WEB サーバーである Jetty を使用し、Servlet として木の表示と編集を実現した。

10.1.1 木構造の表示

JungleTree ブラウザにおいて、Jungle DB は WEB サーバー内に存在し、それから表示に必要な HTML を生成してブラウザに転送する。この流れは、Jungle の NodePath の処理を除けば通常のデータベースのレコードの表示と同等である。

編集するノードのパスは URL で記述されている。例えば、`http://localhost/showBoardMessage?bname=Layout&path=-1,0,2` などとなる。

以下に JungleTree ブラウザを用いて、ノードを表示するまでの流れを記述する。

1. ユーザーは表示したいノードのパスを URL で JungleTree ブラウザに送る。
2. JungleTree ブラウザは、WEB サーバ内にある Jungle から、対応した木を取得する。
3. JungleTree ブラウザは、パスで指定した位置のノードを木から取得する。
4. 取得したノードの中身を、JungleTree ブラウザが表示する。

10.1.2 Jungle Tree ブラウザを使った木の編集

以下に Jungle Tree ブラウザを用いた木の編集の流れを示す。

1. ユーザーは JungleTree ブラウザで編集したいノードを表示するページに移動する。
2. ユーザーは JungleTree ブラウザに木の変更要求を送る。

3. JungleTree ブラウザは Web サーバー内にある Jungle から、対応した木を取得する。
4. 編集を行う木から、JungleTreeEditor クラスを取得し、木の変更を行う。
5. 木の変更を Jungle にコミットする。
6. 木の変更の結果を表示する。

パスを使用することにより、木の変更を Restful に行うことができるように見えるが、木のパスは特定の木の版に固有のものである。ブラウザと WEB サーバは、セッションで結合されており、そのセッションが同じ版の木を編集していれば問題なく成功する。ただし、編集し終わった時に、他の編集が割り込んでいたら、その編集は無効となる。また巨大な木を操作する時には、Path を直接 URL に含むことはできないので、他の工夫が必要になると考えられる。このアプリケーションでは任意の木を取り扱うので、木の大きさの現実的な制限を除けば木の設計の問題はない。

10.2 HTML Rendering Engine

HTMLRenderingEngine は、出力するデータが記述された Contents Tree、出力する形式が記述された Layout Tree の 2 つの木構造を持ち、これらを参照しながら html のレンダリングを行う。またレンダリングする例題は日記を選択した。

10.2.1 Contents Tree の Jungle 上での表現

RenderingEngine では Contents Tree に図 10.1 のように出力するデータを格納した。

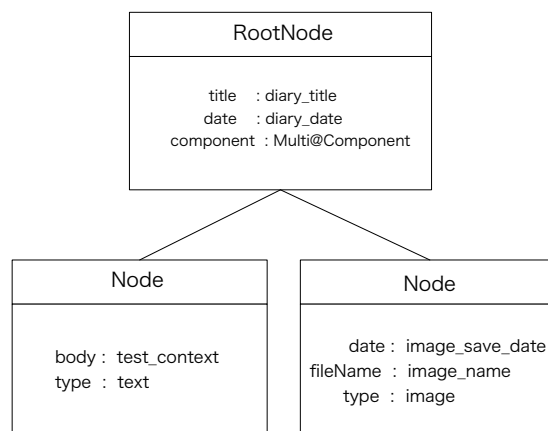


図 10.1: ContentTree

RootNode は Content の title、日時、レンダリングする時に参照する LayoutTree の NodeName を持つ。そして子ノードが日記の本文等のデータを持つ。表 10.1 にノードが保持している Contents の一覧を記述する。

表 10.1: ノードが保持している Contents 一覧

属性名	属性値
component	レンダリングする際に参照する LayoutTree の Node-Name
title	日記のタイトル
date(rootNode)	日記の日時
type	そのノードが保持している ContextType。 text(日記本文) or image(画像データ)
body	日記の本文
date(image)	画像の保存日時
fileName	画像の名前

10.2.2 Layout

html の出力形式を定義する Layout は、複数の Component からなる。表 10.2 に、LayoutTree の主要要素を記す。Layout Tree には図 10.2 のようにデータを格納した。また、LayoutTree はノード同士が NodeName を用いて参照を行う。

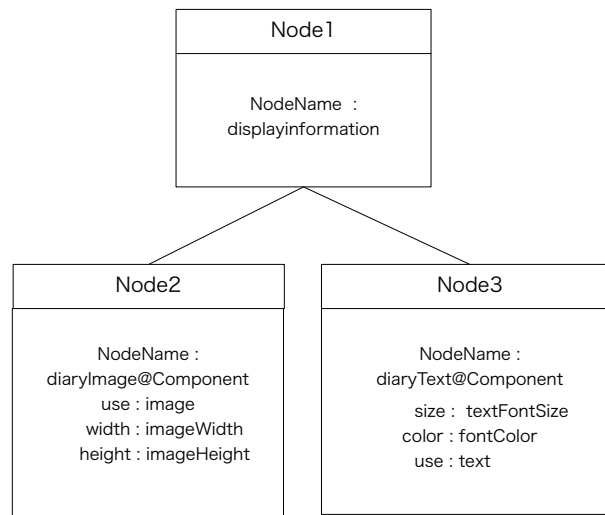


図 10.2: LayoutTree

Layout Tree は、ルートノードに属性名 NodeName 属性値 displayinformation の値を持つ (図 10.2 では Node1 が該当する)。ルートノードは、子ノードに複数の Component を保持する (図 10.2 では Node2、Node3 がそれに該当する)。Node2 は、属性名 use 属性値 image のペアでタグを保持しているため、日記の画像表示に対応する記述が行われて

表 10.2: LayoutTree の主要な要素

属性名	属性値
nodeName	ノードの名前。ノード同士の参照時に用いられる。例外としてルートノードだけは displayinformation という名前を持つ
displayComponent	参照するノードの名前。
Name	この属性名で取得できる値を持つ nodeName を持つノードを参照する。
use	このノードが、どの Contents に対しての Layout を持つかを記述するタグ。表 10.3 にタグと Contents の対応を記述する。
その他	css 等と同じ様な記述を行う。例 属性名 font 属性値 fontSize など

いる。Node3 は、属性名 use 属性値 text のペアでタグを保持しているため、日記の本文に対応する記述が行われている。表 10.3 にタグと Contents の対応を記述する。

表 10.3: tag と contents の対応

tag	content
image	画像の表示
cals	table
date	日付の表示
text	日記の本文
title	日記のタイトル

Layout が複数の Component を参照する際は図 10.3 のような木構造を構築する (この木は LayoutTree の一部であり、本来は参照先のノード等が存在している)。

図 10.3 の例では、diaryMulti@component は diaryText@component と diaryImage@component を参照している。

以下に図 10.1 の ContentsTree、図 10.3 の LayoutTree の 2 つを使用した、レンダリングの流れを記述する。

1. ContentsTree のルートノードは、属性名 component 属性値 Multi@Component の組を持つので、LayoutTree の Node2 を参照する。
2. Node2 は自身の nodeName しか持たないので、子ノードである Node3、Node4 に記述されているデータの参照を行う。

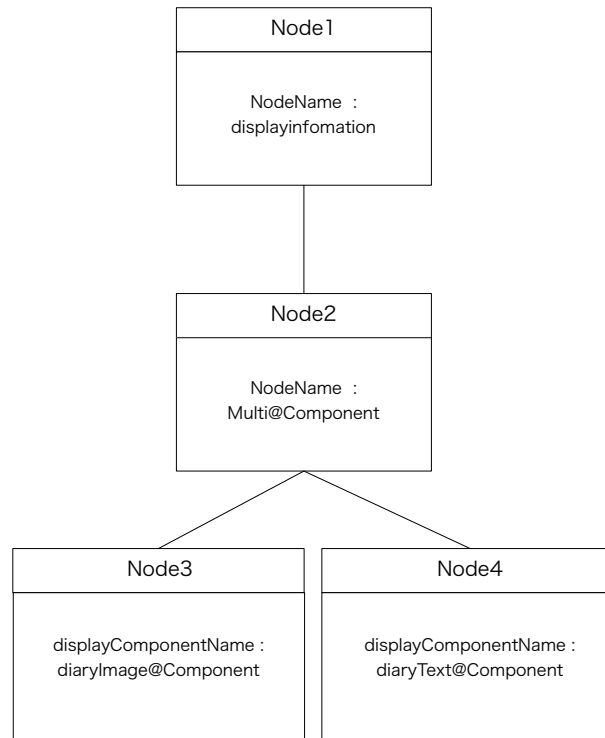


図 10.3: 複数の Component を参照する Layout

3. Node3 は属性名 displayComponentName 属性名 dialyImage@Component の組を持つため、nodeName が dialyImage@Component のノードを参照している。
 4. レンダリングエンジンは、参照先のノードに記述されたルールに則って Html を生成する。
 5. Node3 は、これ以上データを持たないため、次は Node4 を参照する。
 6. Node4 は属性名 displayComponentName 属性名 dialyText@Component の組を持つため、nodeName が dialyText@Component のノードを参照している。
 7. レンダリングエンジンは、参照先のノードに記述されているルールに則って html を生成する。
- (4)、(7) で参照しているノードに関しては、図 10.2 の Node2、Node3 の様な記述が行われている。

10.2.3 Layout Tree のデータ設計

Jungle は汎用の木構造を持つので、データベースを特に設計しなくても、あるがままの形で格納することが可能である。しかし、設計を行うことでより効率的に木構造を扱うことが可能になる。図 10.4、図 10.5 は同じデータを格納した 2 つの木の一部分である。

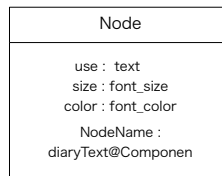


図 10.4: コードとギャップのない Layout の格納方法

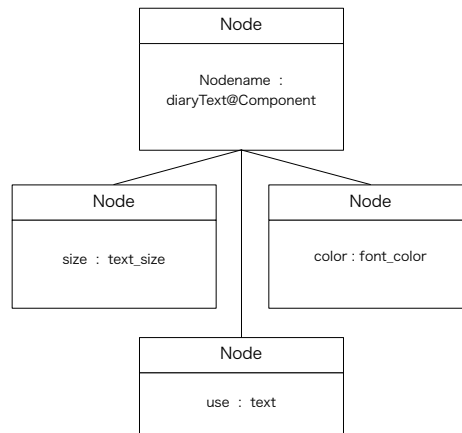


図 10.5: コードとギャップのある Layout の格納方法

図 10.4 の Tree は、1つのノードにレンダリングに必要な値が全て格納されている。そのため、レンダリングを行う際、複数のノードをまたぐ必要が無く、簡潔にコードを書くことができる。

一方、図 10.5 の木はレンダリングする際に必要な値が複数ノードに分散されて保存されている。そのため、全てのノードを参照し、値を集める処理を行う必要があり、コードの可読性が下がり余計な処理も増えてしまう。これより、Jungle の木構造を効率的に扱うためには、設計手法を確立する必要があることがわかった。

10.3 許認可管理アプリケーション

maTrix

本節では実際に企業で運用されている許認可管理アプリケーション maTrix の Jungle 上への実装について記述する。maTrix は、人物、役職、役割、権限、組織の木構造のデータとポリシーファイルを持つ。maTrix の組織構造は、それぞれの木構造が id を用いた参照を行うことで表現される。また、maTrix は過去のアクセス要求を全て保存する。アクセス要求は当時の組織構造を参照している。よって maTrix は過去の組織構造にアクセスするため、組織構造は版管理されている。ポリシーファイルは、データに対するアクセス要求が許可されるか否認されるかを判断するためのルールを、誰が (Target)、何を (Resource)、どうできるか (Action) の 3 つの要素で記述されている。maTrix はアクセス要求に応じたポリシーファイルを参照することで許認可の判断を行う。ポリシーファイルは組織構造中の人や役職を id を用いて参照している。

10.3.1 Jungle 上での maTrix データ構造の表現

maTrix の人物、組織、役割、権限等のデータ構造は木構造なので、Jungle の木構造にそのままマッピングできる。実際の maTrix のデータ構造の一部である人物のデータを格納した JungleTree(図 10.6) を以下に記す。

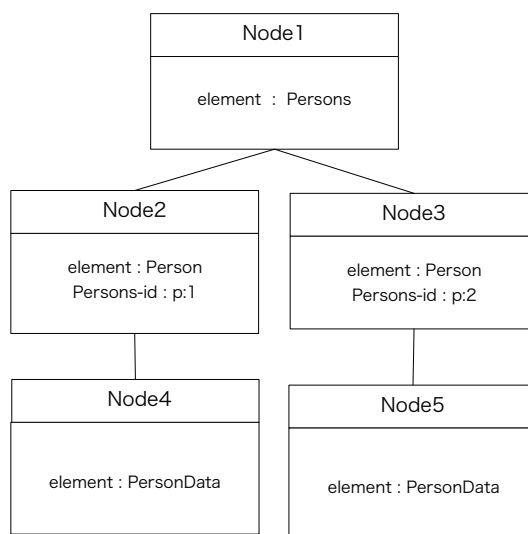


図 10.6: Jungle 上での人物 Tree の表現例

図 10.6 の人物 Tree には、maTrix で使用される人物のデータが記述されている。人物 Tree は、ルートノードが、属性名 element 属性値 Persons の組のデータを持つ。図 10.6 では、Node1 がルートノードとなる。ルートノードの下には、各人物のデータが格納されている (図 10.6 では Node2、Node3 が該当する)。属性名 element、属性値 Person の値

は、この値を持つノードが人物のデータを持っていることを表す。属性名 Person-id、属性値 p:1 はこのノードに記述された人物の maTriX 上での ID を表す。他の役職、役割、権限といった木構造は、この ID を用いて参照を行うことで組織構造を構築する。Node2、Node3 は子ノードに人物の名前や、参照する他の木構造の Id 等のデータを持つが、今回は省略している。

また、maTriX の木構造はお互いを id を用いて参照することで様々なモデルを表現している (例:組織と人物 Tree がお互いに参照して組織運用モデルを表現するなど)。

10.3.2 木構造の版管理

maTriX は組織構造を構成情報モデルとして表現し版管理している。構成情報モデルは、各版に対応した組織構造を構成する木構造のバージョンを保持している (図 10.7、表 10.4)。

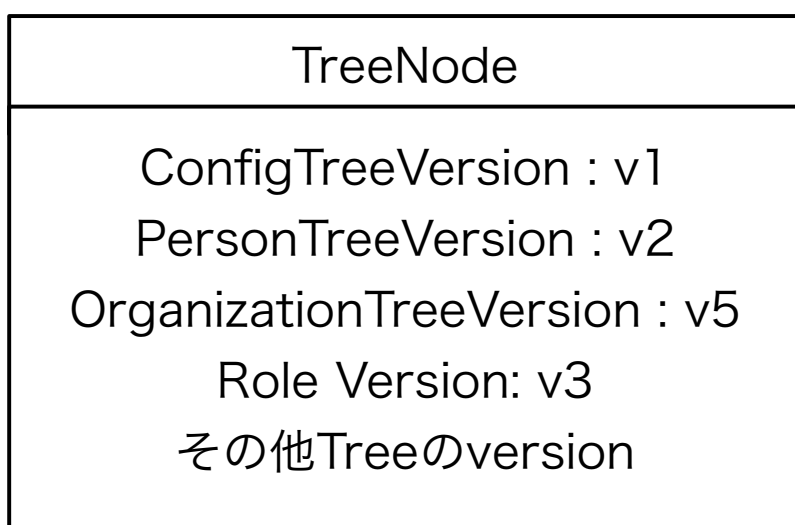


図 10.7: 構成情報モデル Tree のノードの中身

表 10.4: 図 10.7 の構成情報モデル Tree の中身

ConfigTreeVersion	構成情報モデルのバージョン
PersonTreeVersion	人物 Tree のバージョン
OrganizationTreeVersion	組織 Tree のバージョン
RoleVersion	役割 Tree のバージョン

例えば、図 10.7 の構成情報モデルの version1 で管理されている人物 Tree にアクセスしたい場合は、以下の手順で行われる。

1. 構成情報モデル Tree から、ConfigTreeVersion が 1 のノードを取得する。
2. 1 で取得したノードが持っている、PersonTreeVersion を取得する。
3. 2 で取得したバージョンの人物 Tree にアクセスする。

これらの機能により、maTrix のデータモデルの表現が可能になった。

10.4 申請の許認可

maTrix を用いた許認可は、組織構造だけで判断されるわけではない。アクセス可能な時間や、貸出の場合は同時貸出数の上限などのアクセスルールは組織構造だけでは表現できない。そのため、maTrix はアクセス管理のルールを表現方法を定義する XACML という言語で記述された、ポリシーファイルを用いて許認可判断を行う。ポリシーファイルは、リポジトリで管理されており、アクセス要求にあったポリシーファイルが利用される。以下に maTrix での申請の許認可の流れを記述する。

1. A さんが、学科のノート PC の借りるために、maTrix に貸出許可を求める。
2. maTrix はリポジトリから、貸出許可を与えるかを判断するためのポリシーを取得する。
3. ポリシーファイルを元に、データにアクセスを行い許可を与えるかどうかを判断する。

といった流れになる。

ポリシーファイルを読み込み Jungle に問い合わせを行う、インタプリタを実装することで、Jungle の上に maTrix を構築することに成功した。

第11章 性能測定

前章までに、Jungle へ行った改善点・開発したアプリケーションについて述べた。本章では、実装した新機能の性能測定を行う。また、最後に既存のDBとの検索速度の比較を行う。比較対象には、MongoDBとPostgreSQLを選択した。

11.1 測定環境

表 11.1 に、測定を行ったマシンの環境を記述する。

表 11.1: 実験環境

OS	MacOS Sierra 10.12.3
Memory	16 GB 1600 MHz DDR3
CPU	2.5 GHz Intel Core i7
Java	1.8.0.111
mongoDB	3.4.1
PostgreSQL	9.6.1

11.2 TreeMap の測定

5 章で実装した TreeMap の性能測定を行う。比較対象には、TreeMap 実装前に Jungle で使用していた Functional Java の TreeMap を使用する。

図 11.2 は、TreeMap に 1000 回の Get を行った際のグラフである。X 軸は Get を行う TreeMap のノード数。Y 軸は Get にかかった時間を表す

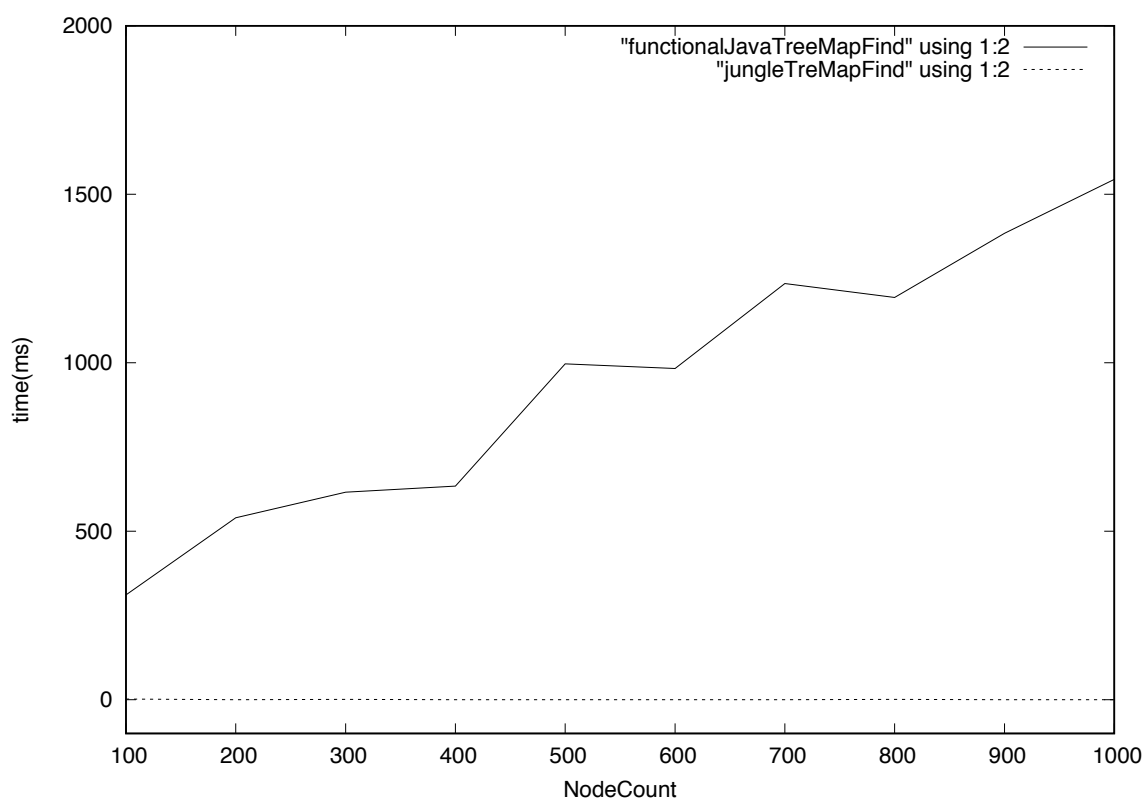


図 11.1: TreeMap への Get

11.2 より、Functional Java の TreeMap と比較して、Jungle の TreeMap の方が非常に高速に動いている。理由として、Jungle の TreeMap は、検索対象の値を持つノードを、二分探索木の探索アルゴリズムに則り探索するのに対し、Functional Java の TreeMap は、検索対象のノードがルートになる木を構築し、ルートを返す。といったアルゴリズムを採用していたため、探索アルゴリズムの差が図 11.2 の結果に出た。その他の処理についても、Jungle の TreeMap の方が高速に動作していた。

11.3 Index の差分 Update の測定

6章で実装した、Index の差分 Update の測定を行う。図 11.2 は、Index の差分 Update と FullUpdate の速度比較のグラフである。測定は、木にノードを追加、Commit を 1 セットの変更として行う。X 軸は、木に行った変更のセット数。Y 軸は、Commit にかかった時間を表す。

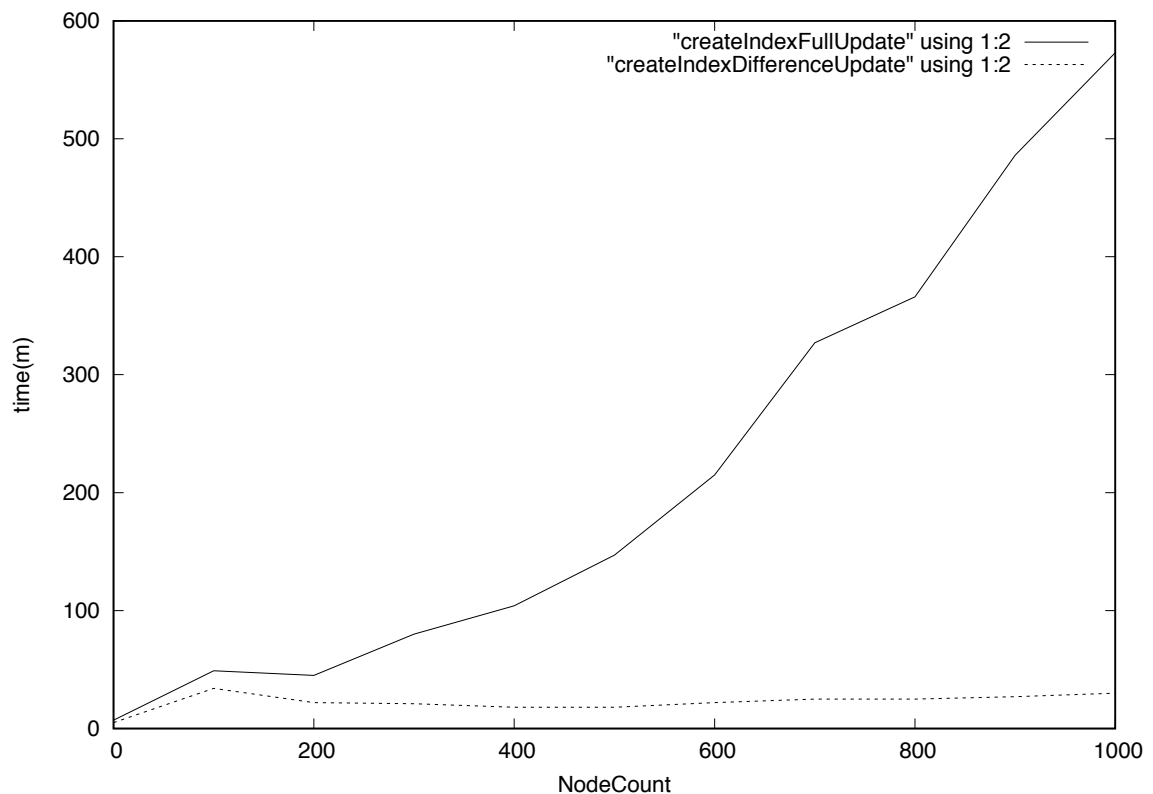


図 11.2: Index の Update

図 11.2 より、Index の Full Update に比べて差分 Updateの方が高速に木の構築に成功している。しかし、Jungleでは木に変更を加える際、毎回 Commitを行うわけではなく、基本的に複数回変更を行った後、一気に Commitを行う。差分 Updateは、変更を加えたノードを記憶し、Commit時に Indexの更新を行う。一方、Full Updateでは、Commitを行うまでに木に加えた変更の数に関係なく、新しい Indexを構築する。よって、Commit前の木の編集回数が増えた場合、Indexの Full Updateと差分 Updateでは、差分 Updateの方が、Indexに対して多くの変更を行うことになる。そのため、Commitを行うまでの

木に対する変更回数によっては、Full Update の方が高速に Index の構築を行える可能性がある。

そこで、図 11.3 に、Commit を行うまでに行った木の編集回数と、Index の Update 速度の測定結果を記述する。X 軸は、1 回の Commit を行うまでに木に行った編集回数。Y 軸は、Commit にかかった時間を表す。

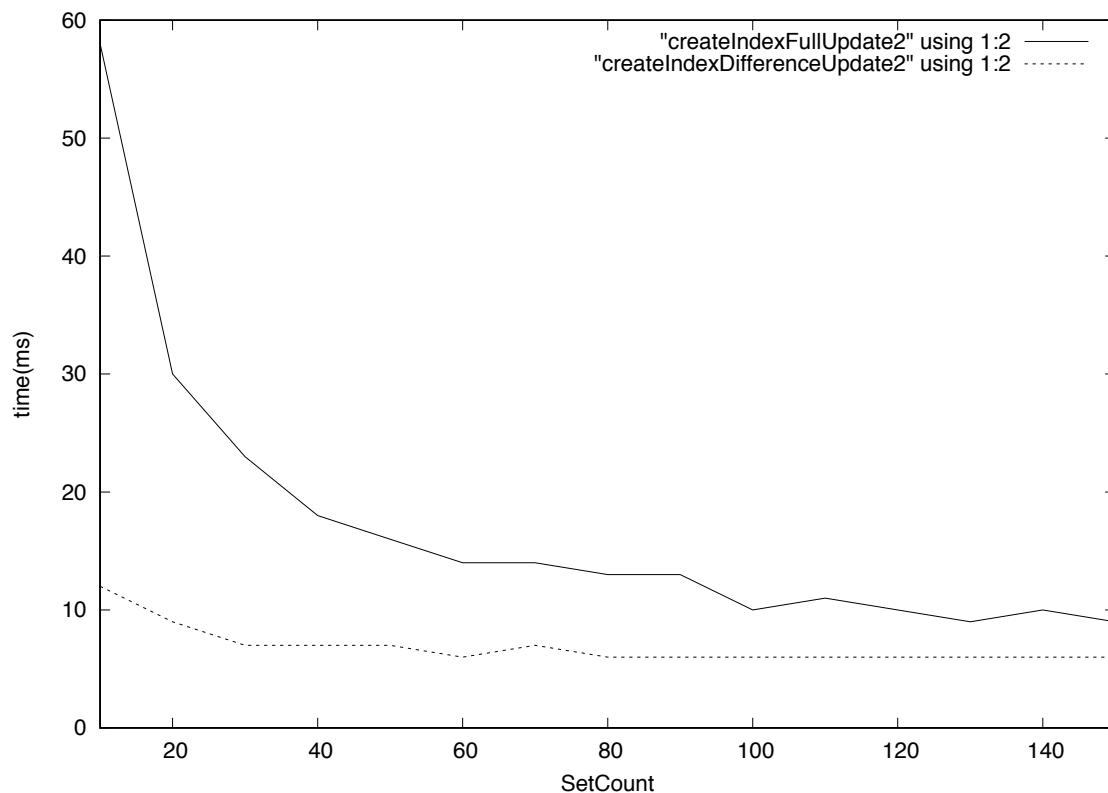


図 11.3: Commit を行うまでに木に加えた変更回数と、Index の構築時間

図 11.3 より、Commit の前に行った木の編集回数に関係なく、基本的に Index の更新は差分 Update の方が早いことがわかった。

11.4 正順の線形木の構築時間の測定

7章で実装した、Differential Jungle Tree の性能測定を行う。比較対象は、Default Jungle Tree を用いる。図 11.4 は、正順の木を構築するまでにかかった時間のグラフである。X 軸は、構築した木のノード数。Y 軸は、構築にかかった時間を表す。また、木のみを構築する時間を測定するため、Index は作っていない。

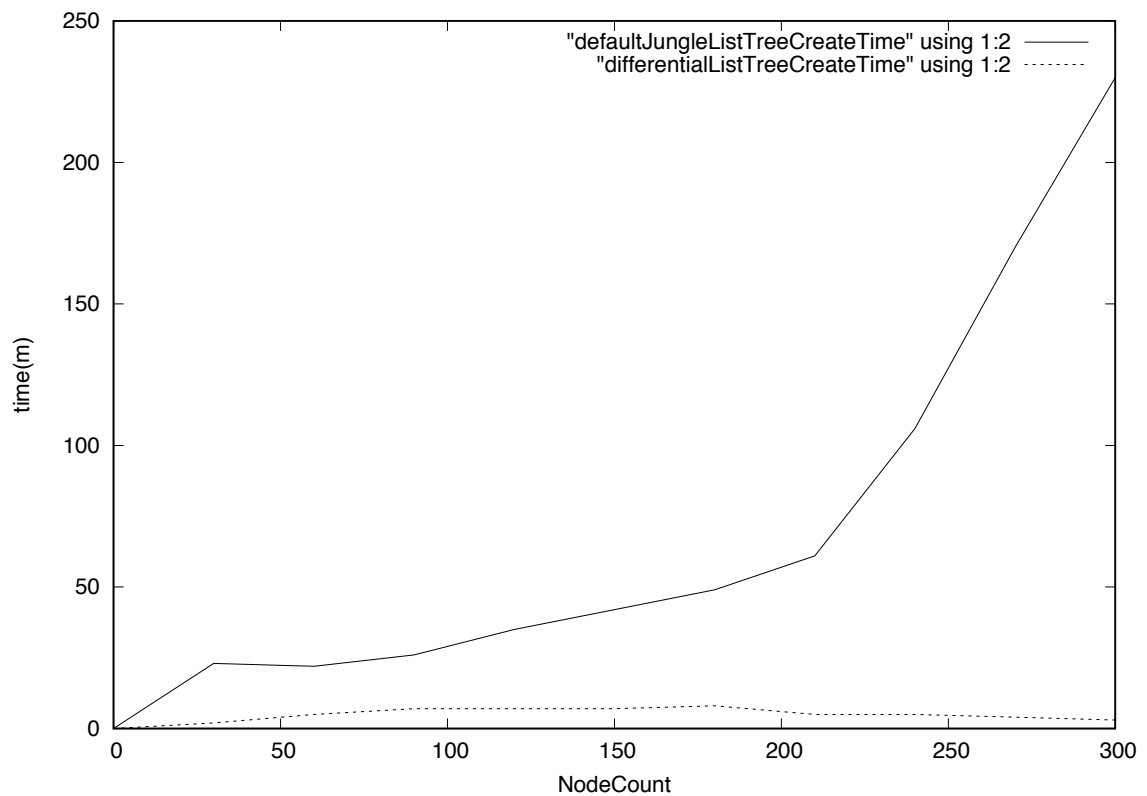


図 11.4: Differential Tree と Default Jungle Tree

図 11.4 より、Default Jungle Tree より、Differential Jungle Tree の方が高速に木の構築している。これは、Default Jungle Tree が、木を構築する際に複製を行うのに対し、Differential Jungle Tree は複製を行っていないからである。期待通りの結果が出たといえる。

11.5 Red Black Jungle Tree の測定

8章で実装した、Red Black Jungle Tree の性能測定を行う。比較対象は、Default Jungle Tree を用いる。図 11.5 は、木を構築するまでにかかった時間のグラフである。X 軸は、構築した木のノード数。Y 軸は、構築にかかった時間を表す。

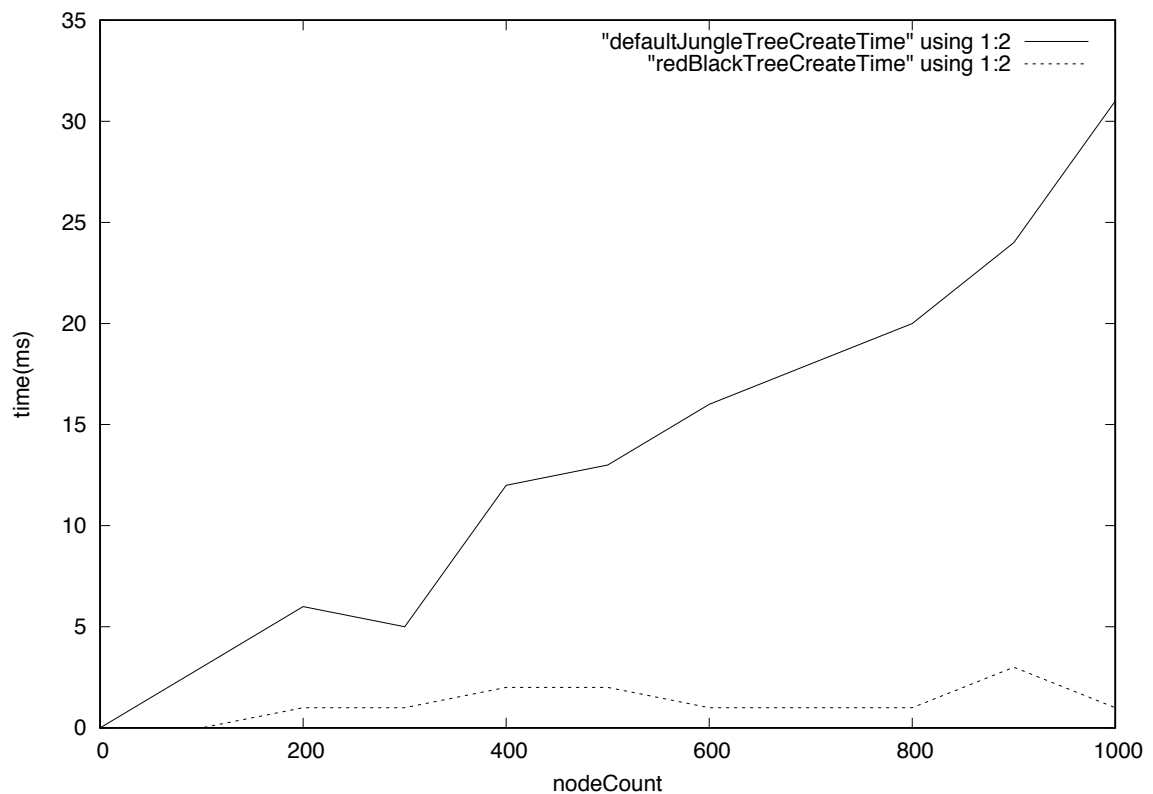


図 11.5: Red Black Jungle Tree と Default Jungle Tree

図 11.5 より、Default Jungle Tree より、Red Black Jungle Tree の方が高速に木を構築している。これは、Default Jungle Tree が、木を構築する際に Index を生成しているのに対し、Red Black Jungle Tree は、自身の木構造が Index と同等の働きを持つため、Index を構築する必要がないからである。

11.6 既存のデータベースとの比較

Jungle と既存のデータベースとの比較を行う。比較対象は PostgreSQL と mongoDB を選択した。データの検索の速度を比較した。図 11.6 に結果のグラフを記述する。X 軸は、データベースのデータ数。Y 軸は、検索にかかった時間を表す。

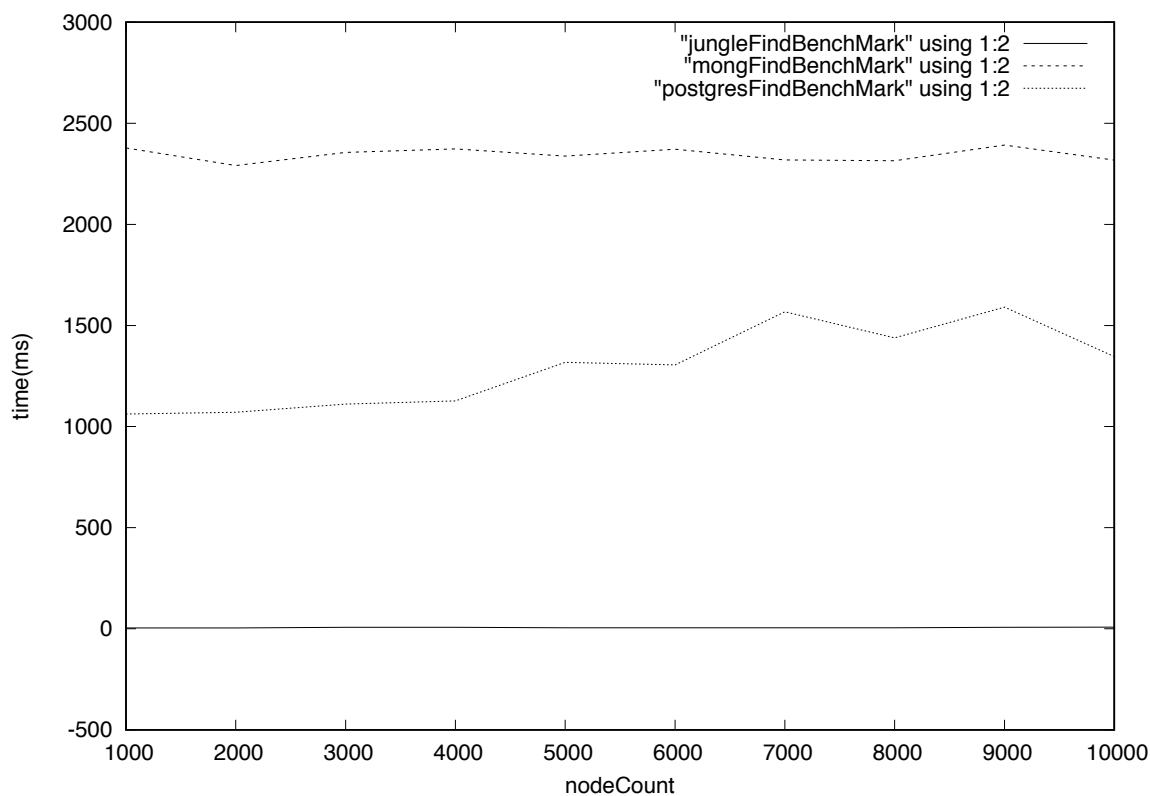


図 11.6: 既存の DB との比較

図 11.6 より、Jungle は PostgreSQL と mongoDB と比較して、非常に高速な検索を行っている。理由として、PostgreSQL と mongoDB は、通信を介してデータにアクセスするのに対し、Jungle は、アプリケーション内にデータがあるため、通信を介さないためだと考えられる。

第12章 結論

12.1 まとめ

本研究では、初めに既存のデータベースについて説明を行い、次に木構造データベース Jungle で使われている非破壊の木構造について述べ、破壊の木構造に比べロックが少ないというメリットがあることを論じた。Jungle の提供している API について記述を行った。

Jungle の Index の性能を向上させるため、非破壊の Red Blck Tree の実装を行った。Index の Update を高速化させるために、前の版の Index と値を共有しながら Update を行う、差分 Update の実装を行った。結果、既存の Index より高速に読み込み、Update ができるようになったことを確認できた。

次に、線形の木を正順で構築する際、木の変更の手間が $O(n)$ になる問題を解決するために、Differential Jungle Tree の実装をした。Differential Jungle Tree は、自身の末尾のノードの情報を保持している。この末尾ノードを使用して、木の編集や検索を行う。

また、Jungle の木の編集の手間は、木のサイズにも依存している。きちんとバランスの取れた木を構築できれば、 $O(\log N)$ で編集を行える。しかし、全ての木の形をユーザーが把握し、バランスを取るのは難しい。そこで、自動的に木のバランスを行い、最適な形の木構造を構築する Red Black Jungle Tree を実装した。Red Black Jungle Tree は、自身が Index と同じ働きをするため、別途 Index を構築する必要がない。よって、Index を構築する Default Jungle Tree より、編集できる。また、ノードは、木のバランスによって Path が編集ごとによって変わってしまうため、属性名と属性値のペアでノードを指定できる、Red Black Jungle Tree Editor の実装を行った。

次に、Jungle を使用した例題アプリケーションの実装を行った。1つ目は、Jungle Tree Browser という、ブラウザ上から Jungle の編集等を行うアプリケーションを開発した。2つ目は、Html Rendering Engine の開発を行った。これらのアプリケーションを開発・運用した結果、Jungle は設計を行うことなく、どんな構造のデータでも格納できるが、きちんと設計を行うと、より高速で簡潔なコードが書けることがわかった。

性能測定では、今回実装した機能の測定を行った。Index の Update・Differential Jungle Tree・Red Black Jungle Tree・全て予想通りの結果が確認できた。また、既存の DB である、MongoDB・PostgreSQL と Jungle の読み込み速度を測定した結果、Jungle の方が極めて高速に読み込みを行えることが確認できた。これは、MongoDB・PostgreSQL が、通信を介してデータにアクセスするのに対して、Jungle は直接メモリの中にあるデータを参照しているためである。

12.2 今後の課題

12.2.1 過去のデータの掃除

Jungle は非破壊でデータを保持し続けるため、非常に多くのメモリを使用してしまう。ある程度の単位で過去のデータの掃除を行いたい。Jungle は、過去の木に対するアクセスをサポートしているため、データの掃除を行うタイミングが明確ではない。なので、メモリから追い出すタイミングを定義する必要がある。

12.2.2 木の設計手法の確立

Jungle は RDB と異なり格納するデータの自由度は大きい。どのようなデータ構造も、設計を行わず格納できる。しかし、十分なパフォーマンスを出すためには、データを最適化する必要がある。また、最適な木構造はアプリケーションによって違うため、Jungle の設計手法を確立させる必要がある。

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准教授に心より感謝いたします。

研究を行うにあたり、協力いただいた並列信頼研究室の全てのメンバーに感謝いたします。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。

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。

2017年3月
金川竜己

参考文献

- [1] MongoDB documentation. <https://docs.mongodb.com/>.
- [2] 大城信康, 河野真治. Data segment の分散データベースへの応用. 日本ソフトウェア科学会, September 2013.
- [3] 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.
- [4] Avinash Lakshman and Prashant Malik. Cassandra: structured storage system on a p2p network. In *PODC '09: Proceedings of the 28th ACM symposium on Principles of distributed computing*, pp. 5–5, New York, NY, USA, 2009. ACM.
- [5] Avinash Lakshman and Prashant Malik. Cassandra - a decentralized structured storage system. *LADIS*, March 2003.
- [6] 玉城将士, 谷成雄, 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, April 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第 28 回大会 (2011 年度) 論文集, Sep 2011.
- [8] 金川竜己, 河野真治. 非破壊的木構造データベース Jungle とその評価. 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, May 2015.
- [9] 金川竜己, 武田和, 河野真治. ソフトウェア内部で使用するのに適した木構造データベース Jungle. 第 58 回プログラミング・シンポジウム, Jan 2017.
- [10] Kevin Henry. Objective viewpoint: Jdbc — java database connectivity. *Crossroads*, Vol. 7, No. 3, pp. 3–ff., March 2001.
- [11] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The objectstore database system. *Commun. ACM*, Vol. 34, No. 10, pp. 50–63, October 1991.
- [12] Malcolm Atkinson and Ronald Morrison. Orthogonally persistent object systems. *The VLDB Journal*, Vol. 4, No. 3, pp. 319–402, July 1995.

-
- [13] Rupali Arora and Rinkle Rani Aggarwal. Modeling and querying data in mongodb. *International Journal of Scientific and Engineering Research*, Vol. 4, No. 5, p. 141, 2013.
- [14] red black tree. <http://fujimura2.fiw-web.net/java/mutter/tree/red-black-tree.html>.
- [15] Postgresql documentation. <https://www.postgresql.org/docs/>.
- [16] Cassandra documentation. <http://cassandra.apache.org/doc/latest/>.

発表履歴

- 非破壊的木構造データベース Jungle とその評価, 金川竜己, 河野真治 (琉球大学), 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, 2015 Okinawa, May, 2015
- ソフトウェア内部で使用するのに適した木構造データベース Jungle, 金川竜己, 武田和馬 (琉球大学), 河野真治 (琉球大学), 第 58 回プログラミング・シンポジウム, Jan, 2017