非破壊木構造データベース Jungle のサービス利用法にそった改良

仲 松　栞[†2]　照屋　のぞみ[†1]　河 野 真 治[†1]

現代のインターネットやアプリケーションで使われるデータは、複雑な構造を持っており、RDB の第一正規系には納まらないようになってきている。また、RDB では、編集中にロックがかけられ、複雑な構造に対する変更に向いていない。これらの問題は、不定形のデータ構造と RDB の表構造のミスマッチである。これを DB のインピーダンスミスマッチと言うことがある。そこで当研究室では、非破壊的木構造データベース Jungle を開発している。Jungle では、不定形のデータ構造を、直接木構造として格納することができる。木構造の変更は、過去の版を保存する非破壊で行われ、木のルートを Atomic に入れ替える操作がトランザクションになる。これにより、複雑な構造を DB として素直に取り扱うことができる。木構造の変更は、木の形に依存している。サービスやアプリケーションに適した木の形があり、それに対応した木の変更方法を採用する必要がある。本研究では Jungle の木と Index の編集機能の改善を行う。直線的なリスト構造に適した Push/Pop と差分リストを提案し、実装と評価を行なった。巨大な木が必要な場合は、木を特定の Key を用いて balance させることにより、変更を O(log n) にすることができる。Jungle は分散構造を取ることもできる。複数の木を複数の分散した Jungle ノード間で通信することにより、Jungle をスケールさせる。Jungle の木の変更 Log を当研究室で開発した分散フレームワーク Alice[1)] を用いて通信する。それぞれの木は、ルートノードに集約され、集約の過程で、競合する変更の Merge を行う。分散 Jungle の性能を測定する手法について述べる。

Nakamatsu Shiori ,[†2] Teruya Nozomi [†1] and Shinji KONO [†1]

1. サービス利用に適したデータベース

データのネスト構造を許さない第一正規系を要求する RDB の表構造は、Json などの不定形のデータ構造とミスマッチを起こしてしまうという問題がある。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方法が考えられるが、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。これらの問題を解決するため、当研究室では、分散処理環境で動くことができるデータベースを目指して非破壊的木構造データベース Jungle を開発している。木構造を扱えることによって、従来の RDB とは異なり、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、そ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。非破壊的木構造とは、データの元の木を直接書き換えずに保存し、新しく構築した木のデータ構造を編集する方法である。これにより、データの書き込み中であっても、元のデータは変更されない為、読み込みも同時に行うことができる。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、分散環境における Jungle の書き出し速度の測定方法の提案をする。

2. 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非破壊的木構造データベースで、Java を用いて実装されている。非破壊的木構造とは、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のノードをコピーする特徴を持つ（図 1 ）。これにより、読み込み中にデータが変更されないことが保証されているため、書き込みと読み込みを同時に行うことできる。 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できている。ノードは自身の

†1 琉球大学大学院理工学研究科情報工学専攻
Interdisciplinary Information Engineering, Graduate School of Engineering and Science, University of the Ryukyus.
†2 琉球大学工学部情報工学科
Information Engineering, University of the Ryukyus.

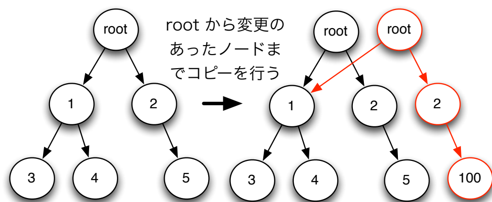

図 1　非破壊的木構造の編集

子のリストと属性名、属性値を持ち、データベースのレコードに相応する。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。また、親と子相互で参照しあうことができる破壊的木構造データベースとは違い、非破壊的木構造である Jungle は、親から子への片方向の参照しかできない。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。

3. Jungle の構成要素

[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 <-1,0,2,3 > を表している際の例を (図 2) に示す。

[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、すべての版に独立した 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 の複製を行い、その後更新する必要があったため、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 は実装されている。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、データの更新を行なった際、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 で Index を実装した場合と違いは、木の回転処理が入るという点である。これにより複数の版すべてに対応した Index をサポートすることが可能になった。

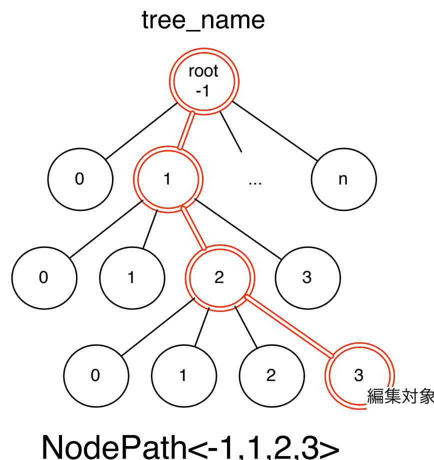

図 2　NodePath

[非破壊 TreeMap]

Jungle の Index は、Functional Java の非破壊 TreeMap を用いて実装を行なっている。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ている。RedBlackTree とは二分木探索の一つで、以下の条件を満たした木のことである。

- 各ノードは赤か黒の色を持つ。
- ルートノードは黒色である。
- 全ての葉は黒色である。
- 赤いノードの子は黒色である。
- 全ての葉からルートノードまでのパスには、同じ個数の黒いノードがある。

4. Index の差分 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、編集を行な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際、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。

編集前ののノードの削除

ndex 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。

- 編集を行なったノードのリストからノードを取得する。
- 取得したノードが、保持している値を Index から削除する。
- 自身と子供のペアを Parent Index から削除する。
- ParentIndex から親を取得する。
- 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ｋなるまで続ける。
- 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なっても構わない。

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。

- 木からルートノードを取得する
- 取得したノードが Index に登録されているかを調べる。
- 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。
- 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。
- 自身と子ノードを Parent Index に登録する。
- 自身の子ノードを取得したノードとして 2 に戻る。
- 全てのノードを登録したら終了する。

5. Differential Jungle Tree の実装

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 3)。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。 しかし、PushPop はルートノードを追加し

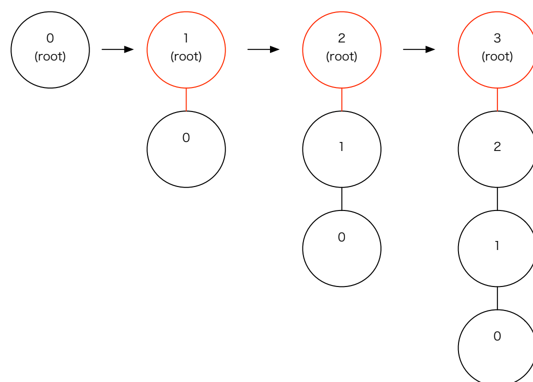

図 3 PushPop

ていくため、(図 3) 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。そこで、木の編集の手間を O(1) で、正順の木を構築できる Differential Jungle Tree の実装を行なった。 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ノードを保持する。木の編集は、別途構築した Sub Tree に対して破壊的に更新を行い、Commit 時に末尾のノードに Sub Tree を Append することで行う。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。

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を保持することが可能な Differential Tree Context を作成した。

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**??**)

末尾ノードを使用した木の編集

Differential Jungle Tree の木の編集は、Differen-

表 1　Jungle に新しく実装した API

createNewDifferenceTree (StringtreeName)	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。

tial 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、Editor 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 4)

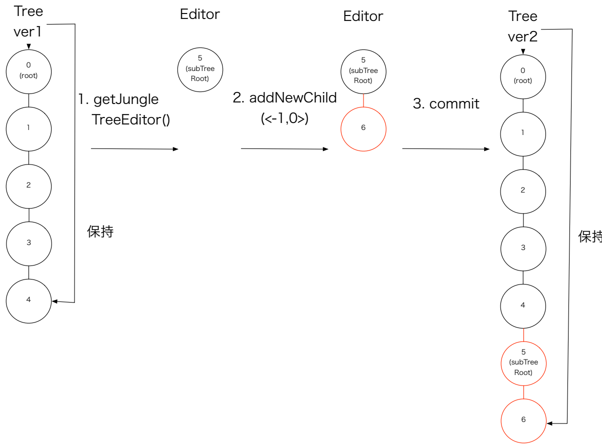

図 4　末尾ノードを使用した木の編集

- 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)。
- 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。
- 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。

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、過去の木に対して全探索を行なった場合、その版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 5)。ここで、Tree ver1 に対して、全探索を行なった場合、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 を用いて木の全探索を行なった場合、Tree ver1 に存在しないノード 3 ・ 4 は検索対象から省かれる。

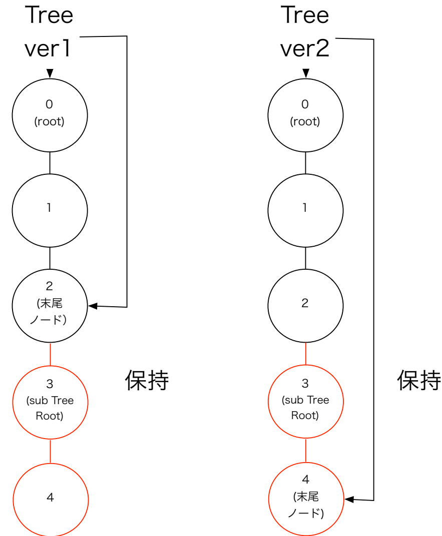

図 5　複数の版の木の表現

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。

Differential Jungle Tree の検索

Differential 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、TreeContect 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 を行なった場合、木の整合性が崩れてしまう問題もある。(図 5)(6) を例に解説する。

（図 5）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 に対して全検索を行う。differential Jungle Tree に対する全検索は、末尾ノードよ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 つの子ノードを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。

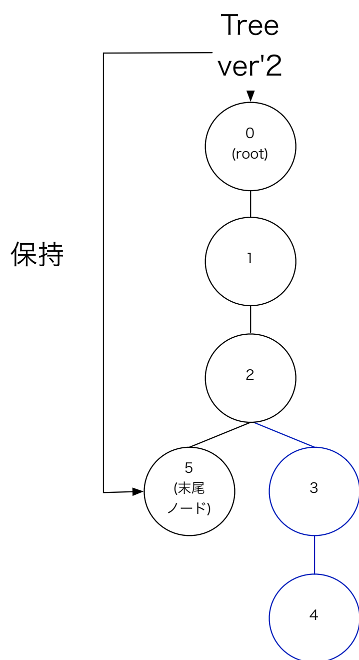

図 6　木の整合性が崩れる例

6. Red Black Jungle Tree の実装

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。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 O(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 を実装した。バランスは、木の生成時に特定の Balance Key を決定し、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。

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 2)

表 2　Jungle に新しく実装した API

createNewRedBlackTree (StringtreeName, String balanceKey)	Jungle に新しく Red Black Jungle Tree を生成する。第一引数に木の名前、第 2 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。

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。 Red Black Jungle Node Path は、引数に String 型の BalanceKey と ByteBuffer 型の value を取る。Red Black Tree Node Path で指定できる属性名は、木の生成時に宣言した属性名しか使用できない。これは、Red Black Jungle Tree が木の生成時に宣言した属性名でソートされているからである。

7. 分散環境での JungleDB の書き出し実験方法の提案

Jungle は分散環境で動くデータベースを目指して開発している。Jungle には大量の様々な木が存在し、その大きさも様々である。それぞれの木を異なる Jungle ノードに格納することにより、木が大量になる場合にも一定した性能が出るようにしたい。一つの木が極端に大きくなる場合もあるが、それを分散して格納するのは難しい。ここでは、一つの木は一つのノードに納まる大きさだと仮定する。Jungle の木は信頼性向上とアクセス速度の向上のために、複数のノードに格納される。木の変更は複数のノードを伝搬し、特定の

Jungle の木のルートノードに到達する。そこで、木の状態が確定する。一つのルートノードではなく、複数のノードに対して、多数決 Commit などの方法を用いることも考えられるが、今回は単一のルートノードを用いる。この方法は、読み込みに対して書き込みが少ない場合、あるいは書き込みが単一ノードのみからくる場合に有効であると考えられる。(表 7)

従来の JungleDB の分散機能の測定は Jetty Web サーバー込みで行なっており、DB に対する負荷は直接的には大きくなかった。JungleDB に対して十分な負荷をかける http リクエストを生成するのは困難であった。そこで、Jungle の分散性能そのものを調べるために、Web サーバー抜きで測定したい。

JungleDB の通信は、研究室で開発した分散フレームワーク Alice によって行われる。Alice はノードの木構造接続を自動的に行う TopologyManager を持っている。ここでは、Jungle の木それぞれに対してノードの木構造を別々に構成する必要がある。これは、TopologyManager を改良することによって行う。Alice はまた、通信を圧縮して行う機能も持っており、それによる高速化も期待できる。通信されるのは、Jungle の木に対する変更 Log であり、これをまとめて転送することにより、圧縮がうまく働くと期待される。複数のノードの変更は、互いに競合することがあり、ノード間をルートに向かって通信していく間に競合を解消する必要がある。これを Merge をいう。従来の DB のような変更では、Merge が失敗することがある。失敗した場合は、複数の木構造がまとめて失敗してしまう。Jungle では Merge が失敗しないような場合、例えば、SNS へのコメントの追加などではうまく働く。従来の DB として Jungle を使う場合には、単一ノードで使用するか、多数決 Commit を使うことになるが、ここではそれは測定しない。

今回、実験で分散構造として Jungle の動くノード 16 台を木構造に接続する。この木のルートノードをルート jungle と呼び、末端ノードをリーフ jungle と呼ぶ。 まず、末端の Jungle に User が書き込みをし、Jungle から user にレスポンスが返ってくるまでの時間を測定する。次に、Jungle の変更がルートの Jungle にコピーされるまでの時間の測定方法を提案する。Jungle 同士の接続には、当研究室で開発している分散フレームワークである Alice を用いる。

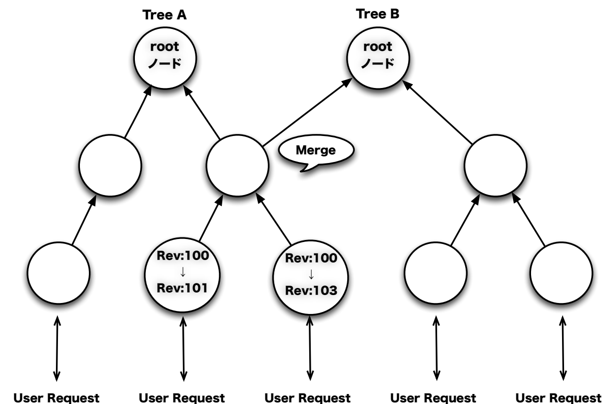

図 7 複数のルートノードを持つ木構造

8. ま と め

本研究では、始めに破壊的木構造データベースである Jungle について説明を行い、次に Jungle の性能を上げるために実装した点を挙げ、最後に分散環境での Jungle の書き出し実験の手法について述べた。実装した点は、まず Jungle の 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 の実装を行なった。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。次に、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 を構築する Default Jungle Tree により、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 の実装を行なった。また、Jungle の分散機能に対する測定手法を提案した。今後の課題として、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。

参 考 文 献

1) 照屋のぞみ，河野真治：分散フレームワーク Alice の PC 画面配信システムへの応用，第 57 回プログラミング・シンポジウム (2016).
2) 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 (OS) (2015).