分散機構造データベースJungleによる企業向け企業システム

琉球大学工学部情報工学科

金川竜己

指導教員 河野真治

目 次

図 目 次

表 目 次

第1章　知的構造を格納するためのデータベース

1.1　研究目的

知識を記した書物等において、節が複数集まって章になり、章が集まって書物になるように、我々があつかっている知識は主に木構造である。RDB 上に木構造を格納する際、node と node の関係を定義したテーブルを用意するが、これだとデータ構造が複雑になってしまい、検索や更新が非効率的なものとなってしまう。そのため、もっと木構造を簡単にデータベースで扱いたい。

また、データベースを使用するウェブサービスの規模も年々大きなものとなり、それに比例してデータベースへの負荷も増大し、その結果サービスが停止する自体が多々見られるようになった。そのため、データベースの処理性能はそのままサービスの質につながっている重要な項目となっている。

データベースの処理性能を向上させる代表的な方法として、ハードウェア的に高性能なマシンを用意することで処理性能を上げるスケールアップと、汎用的なマシンをいくつも用意し、処理を分散させることで処理性能を上げるスケールアウトの 2 つがある。単純に処理能力を上げたいのなら、スケールアップは有効ではあるが、単一のマシンを高性能にするのにも限界があり、いずれはそのマシンの限界を超える負荷がかかる可能性もある。それに対しスケールアウトは、処理が重くなるに連れて汎用的なマシンを順次追加していくことで性能を上げるため、ハードウェア的に高性能なマシンを要求せずにすみ、柔軟な対応を取ることが出来るため、データベースの性能を上げる方法としてはスケールアウトが求められている。本研究で扱うスケーラビリティとはスケールアウトのことをさす。

データを分散させて扱う分散データシステムにおいて、データの整合性 (一貫性)、常にアクセスが可能であること (可用性)、データを分散させやすいかどうか (分割耐性)、この 3 つを同時に保証することは出来ない。この定義を CAP 定理と言う。

今、最も使われているデータベースである RDB は、一貫性と可用性を重視している。そのため、データを分割し、複数のノードにデータを分散させると、整合性のチェックやデータの統合処理性能が低下する等の問題があり、データを分割し、マシンを追加して負荷を分散することが困難である。

当研究室では、これらの問題を解決した、煩雑なデータ設計が必要ないスケーラビリティのあ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。Jungle は、全体の整合性ではなく、木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す繰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。そのため、Jungle は可用性と分割耐性を重視しつつ、整合性も持つ。

Jungle は、これまでの開発によって木構造を格納する機能をもっている。そこで、当研究では、共同研究を行っている Symphonies 社が開発している組織の中の許認可を管理するアプリケーション maTrix に Jungle を組み込み、実装すべき API の洗い出しを行い、その後実用 DB としての性能があるか実証実験を行う。

第2章　分散木構造データベースJungle

Jungle とは、当研究室で開発を行っている、スケーラビリティのある、世の中の知識構造を煩雑なデータ設計を行うこと無く格納できることを目標としたデータベースである。本章では、Jungle の基本的な特徴についての解説を行う。

2.1　Jungleのデータ構造

研究目的でも述べたが、我々が扱う知識は主に木構造である。Jungle はそれらの知識をそのまま格納するため、データ形式は非破壊的木構造を採用している。はじめに、非破壊的木構造と通常の破壊的木構造の違いについて説明を行う。

通常の破壊的木構造は、データの編集を行う際に、データを上書き更新する (図 2.1) ため、編集を行っている間ずっと木にロックをかける必要があり、また、閲覧者がいる場合、検索途中にデータが変わることを避けるために、データの検索が終わるまで書き換えを待つ必要がある。しかし、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。

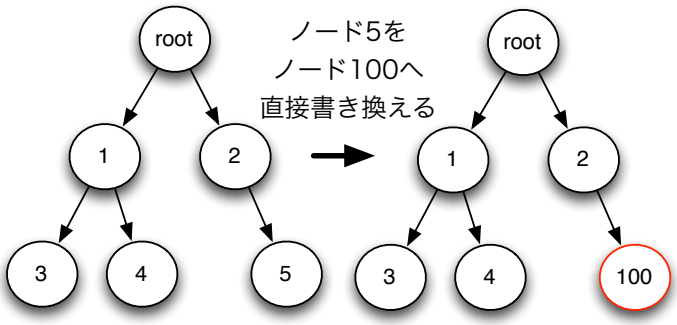

図 2.1: 破壊的木構造の編集

それに比べ非破壊的木構造は、一度生成した木を上書きすることはない。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構築することで行う (図 2.2)。

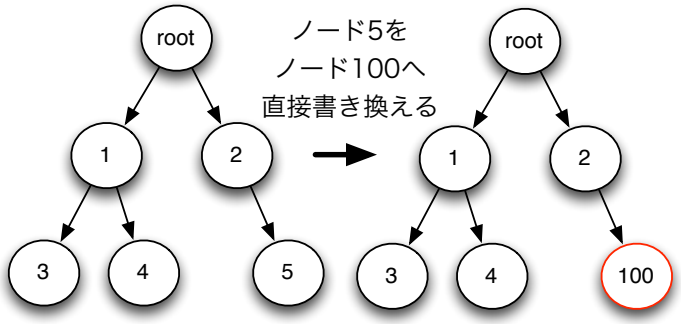

図 2.2: 非破壊的木構造の編集

非破壊的木構造においてデータのロックが必要になる部分は、木のコピーを作った後に、ルートノードを更新するときだけである。また、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、非破壊的木構造は検索中の木が変更されないことが保証されいているため、編集中においてもデータの読み込みが可能である。(図 2.3) そのため、非破壊的木構造に比べてスケールアウトがしやすくなっている。

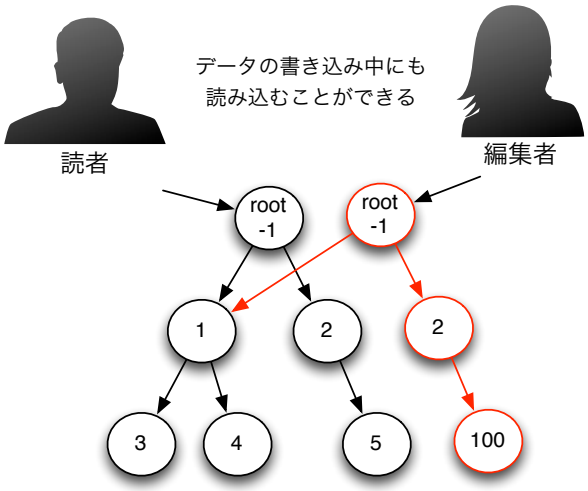

図 2.3: 非破壊的木構造の編集

また、過去の Tree は全て保持しているため、いつでもアクセスすることが可能である。

2.2 分散機能

Jungle の分散機能は、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 を使用している。Alice はユーザー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等、ネットワークトポロジー形成機能を提供している。

Jungle は、ネットワークトポロジーを構築する際に、木構造を想定したネットワークトポロジーを形成しサーバー同士を接続することで通信を行っている。木構造なら、一度 RootNode までデータを伝搬させることで整合性を取ることが出来る (図 2.4) からである。データの伝搬中に衝突が発生した場合、Merge を行い結果を改めて伝搬すれば良い。

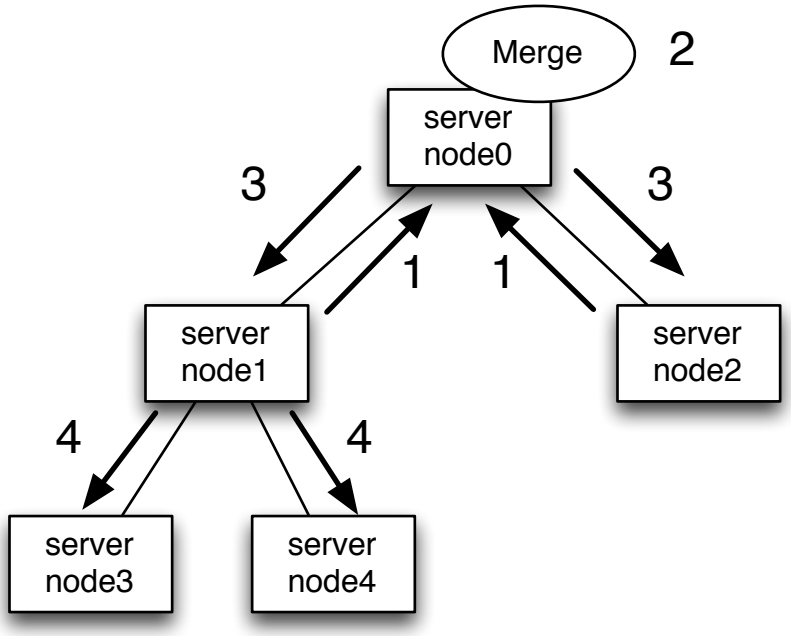

図 2.4: Jungle-network のトポロジの形成例とデータの伝搬順序

第3章　組織中の許認可管理アプリケーションmaTrix

maTrix とは Symphonies 社が開発しているアカウント管理、許諾判定システムのことである。人や組織の情報などを保持しており、それらの情報を関連付けることで組織の構造を表現し、許認可管理を行う。本章では maTrix の説明と、Jungle 上でどのように maTrix を表現したかの説明を行う。

3.1　maTrixの保持するデータ構造

matrix は人、役職、役割、役割記述要素といったデータ (表 3.1) を用いた組織構造と、許認可の判断に用いるポリシーファイルの２つのデータを持っている。

表 3.1: maTrix の組織構造

人物 Tree	人のデータが入っている。1 人 1 人ユニークな personId を持つ
組織 Tree	組織のデータが入っている。企業ごとにユニークな orgId を持つ
役割 Tree	人や組織の役割データが入っている。役割毎にユニークな roleId を持つ
役割記述要素 Tree	役割が持っている要素が記述されている。 役割記述要素毎にユニークな rdeId を持つ

表 3.1 に書いてあるように、maTrix の保持している組織のデータはそれぞれがユニークな Id を保持しているため、それを利用しお互いに参照することが可能である。これらのデータを元に maTrix は組織構造を構築している。例として、人物 Tree と組織 Tree の組み合わせで組織運用モデルを表現しており、役割 Tree は役割モデルを表現している。このように様々な情報が参照しあうことでたくさんのデータモデルを表現している。以下に maTrix の組織情報モデルを記す (図 3.1)。

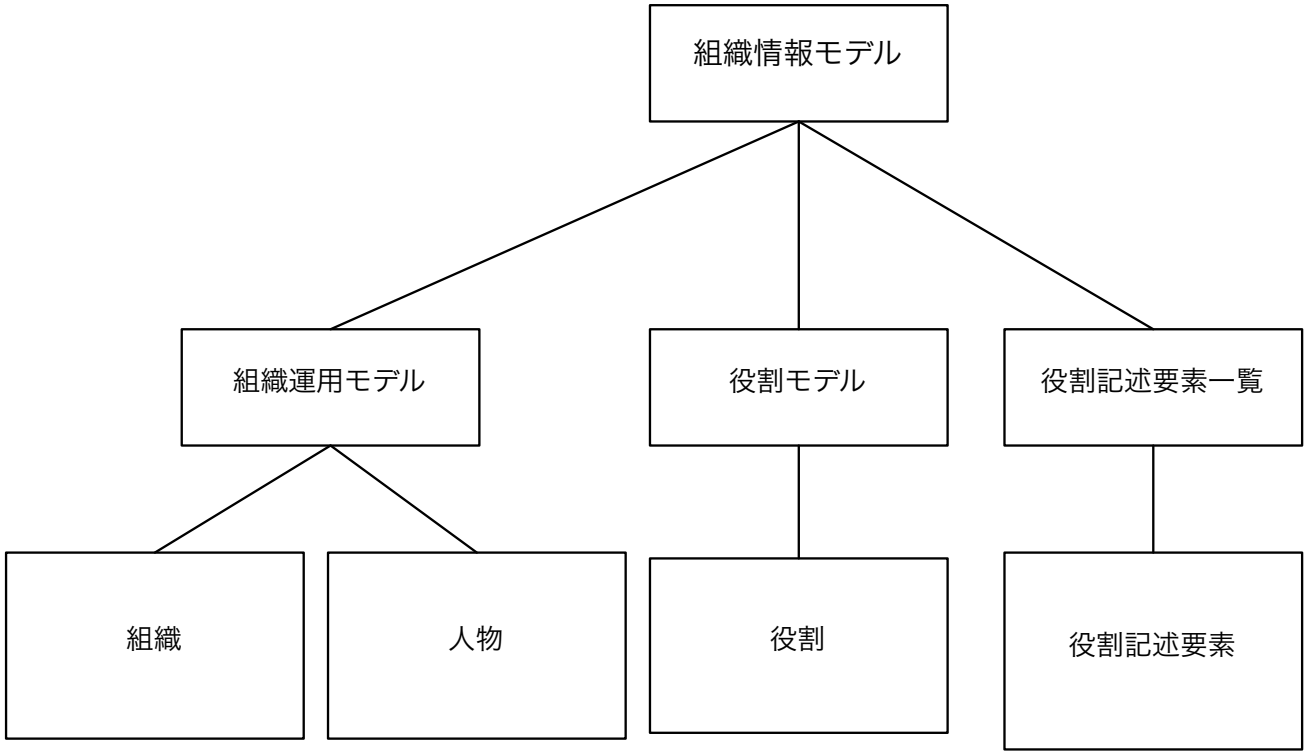

図 3.1: 組織モデル表現例

maTrix のデータ構造は、保持している組織情報を xml や json 形式で出力することができる。以下に人物 Tree を xml 形式で出力したデータの一部として、1 人分のデータを記述する

```
<Persons>
<Person id="p:1" type="Person">
<accountId>a:26</accountId>
<lastName>東</lastName>
<name>東俊一</name>
<nameReading>あずましゅんいちくん</nameReading>
<roleRefIds>r:10 r:34</roleRefId>
</roleRefIds>
<parentOrganizations type="OrganizationMappedByRole">
<OrganizationMappedByRole type="OrganizationMappedByRole">
<organizationRefId>o:2</organizationRefId>
<roleRefId>r:10</roleRefId>
</OrganizationMappedByRole>
<OrganizationMappedByRole type="OrganizationMappedByRole">
<organizationRefId>o:11</organizationRefId>
<roleRefId>r:34</roleRefId>
</OrganizationMappedByRole>
</parentOrganizations>
<priorities type="PriorityMappedByRole">
<PriorityMappedByRole type="PriorityMappedByRole">
<priority>0</priority>
<roleRefId>r:10</roleRefId>
</PriorityMappedByRole>
<PriorityMappedByRole type="PriorityMappedByRole">
<priority>1</priority>
<roleRefId>r:34</roleRefId>
</PriorityMappedByRole>
</priorities>
</Person>
</Persons>
```

表 3.2: Person.xml の要素

Persons	この要素以下に Person の情報があることを意味する
Person	人の情報が以下にあることを示す。uniqueId が割り振られている
accontId	その Person のアカウント Id。Tree 同士の参照に用いる。
lastName	苗字
name	フルネーム
nameReading	名前のふりがな
roleRefIds	役割の Id を記述する
parentOrganizations	この要素以下にその人が所属している組織の Id を記述する
OrganizationMappedByRole	この要素以下に組織と、その組織の役割を記述する
organizationRefId	所属している組織の Id
priorities	人物に割り振られている役割の優先順位を以下に記述する
PriorityMappedByRole	この要素以下に役割と優先順位をペアで記述する
priority	役割の優先順位を記述する

Person.xml を例で上げたが、役職 Tree や役割 Tree も同じような構造でデータを保持している。また、maTrix には、組織情報からデータを取得する Function が 15 種類実装されている。

3.2 Jungle 上での maTrix のデータ構造の表現

maTrix が保持している、人物 Tree や役職 Tree は、木構造のデータであるためそのまま Jungle に格納することができる。実際に人物 Tree を格納した際のデータの形の一部を (図 3.2、図 3.2) に示した。(図 3.2) は前章で表示した Person.xml と対応している。

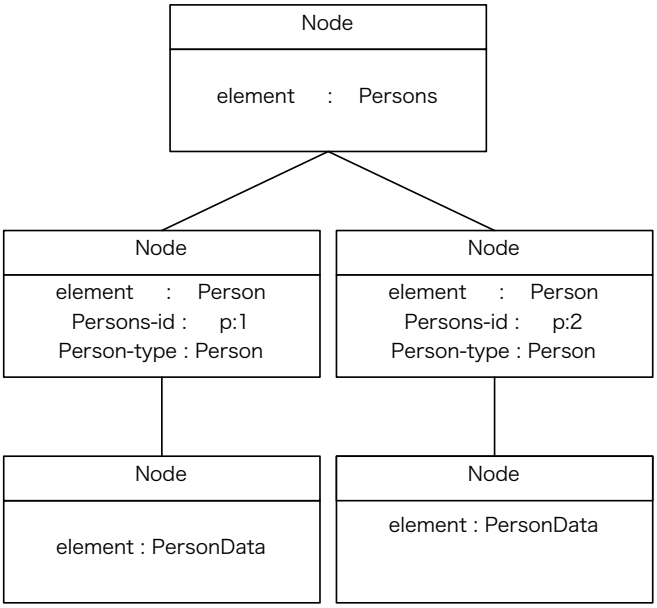

図 3.2: Jungle 上での人物 Tree の表現例 (1)

表 3.3: 図 3.2 に対応した XML

```
<Persons>
<Person id=”p:1” type=”Person”>
<PersonData> </PersonData>
</Person>
<Person id=”p:2” type=”Person”>
<PersonData> </PersonData>
</Person>
</Persons>
```

図 3.2 は、表 3.3 を Jungle に格納したものである。Jungle は、TreeNode に<Key String, Value Attribtue>の組み合わせでデータを保持する。そのため、XML のデータを TreeNode に格納する際、要素の格納は< Key ”element”, Value ”要素名”>、属性の場合は<Key ”要素名-属性名”, Value ”属性値”>で格納する。属性格納時の Key を、”要素名-属性名”にした理由は、属性名のみで格納してしまうと、Key で検索を行う際に該当する Node が増えすぎるため、key を”要素名-属性名”とし、なるべく Key の重複を避けるように格納した。

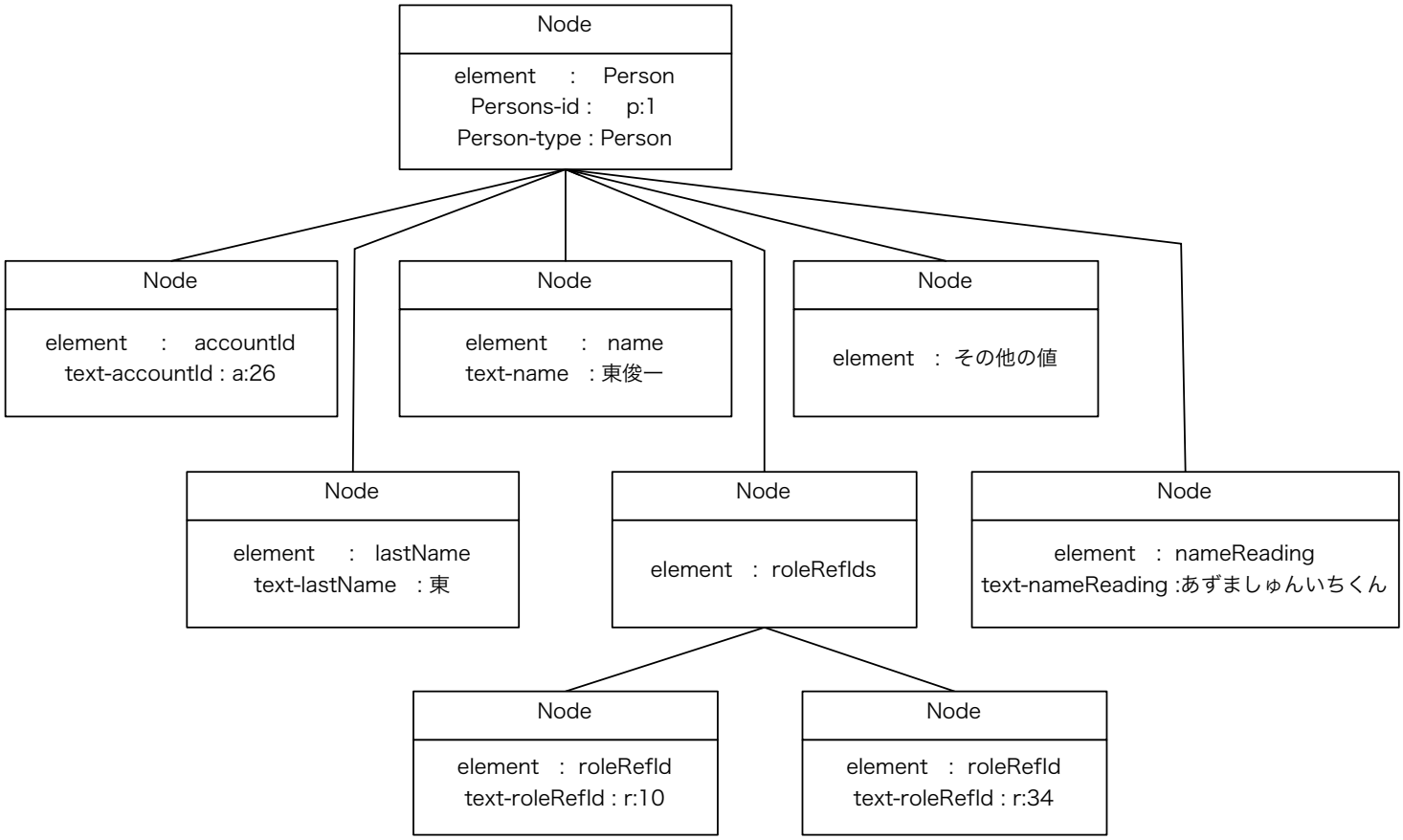

図 3.3: Jungle 上での人物 Tree の表現例 (2)

図 3.3 と表 3.4 も同じく PersonTree の一部を Jungle に格納した時の図である。基本的な構造は図 3.2 と変わらないが、1 箇所 XML の構造を変えて Jungle に格納した箇所があるのでそこの解説を行う。Jungle はノードにデータを格納する際に、Key,Value で格納するが、key に対応する value は 1 つしか格納できない。そのため、表 3.4 の<roleRefIds>r:10 r:34</roleRefId> の様に、2 つの値をテキスト部分が保持している場合どちらか片方しか格納することが出来ない。そこで、読み込む際にデータ構造を、表 3.5 の様に書き換えて Jungle に格納する。

表 3.4: 図 3.3 に対応した XML

```
<Person id=”p:1” type=”Person”>
<accountId>a:26</accountId>
<lastName>東</lastName>
<name>東俊一</name>
<nameReading>あずましゅんいちくん</nameReading>
<roleRefIds>r:10 r:34</roleRefIds>
<その他の値> </その他の値>
</Person>
```

表 3.5: Jungle に対応した XML 文例

```
<roleRefIds>
<roleRefId>r:10</roleRefId>
<roleRefId>r:34</roleRefId>
</roleRefIds>
```

この様に maTrix のデータ構造を Jungle にマッピングしていく。

3.3 maTrixのデータの版管理

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 version 毎に存在し、各 version に対応した組織構造を構成する Tree を保持している。

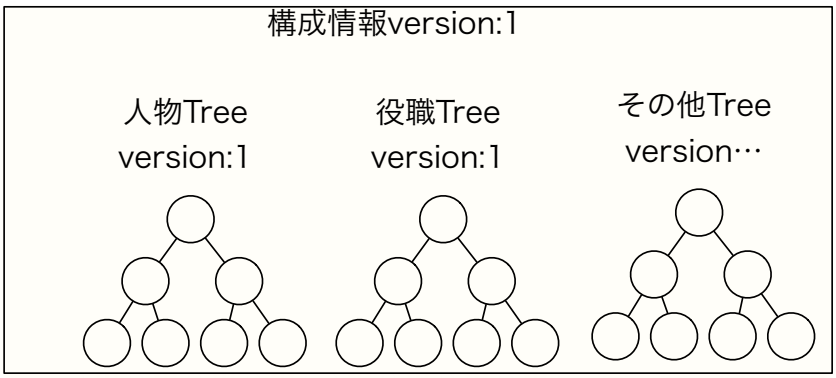

図 3.4: maTrix の構成情報例 1

人物と役職の Tree の version が共に 1 の時、構成情報モデルの version も 1 とする。

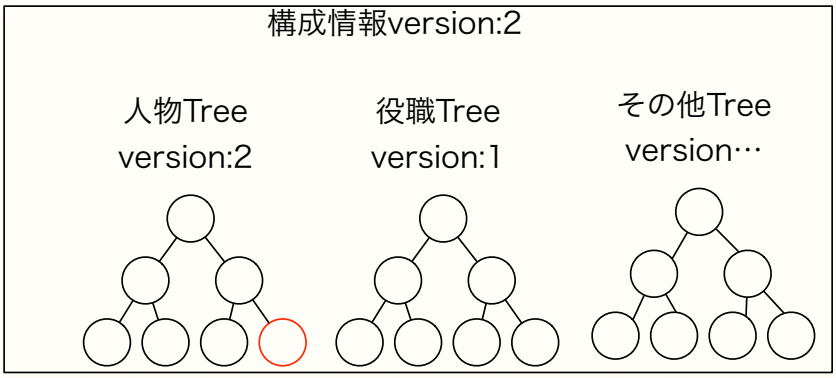

図 3.5: maTrix の構成情報例 1

構成情報モデルの version:1 の人物 Tree を更新した場合、version:2 の構成情報が構築され、maTrix は version:1 の構成情報と version:2 の構成情報の両方を保持し、両方の構成情報にアクセスすることが可能である。

3.4 Jungle での maTrix の版管理の表現

Jungle の Tree は、全ての version の Tree でユニークな Id を保持しているため、Id を指定すれば、過去の Tree にアクセスすることが可能である。それを利用し、Jungle 上での過去の変更履歴を保持する構成情報モデルの表現は、構成情報モデルの version と、各 Tree の version を保持し、関連付ける構成情報モデル Tree を作成し表現した。(図 3.6)。

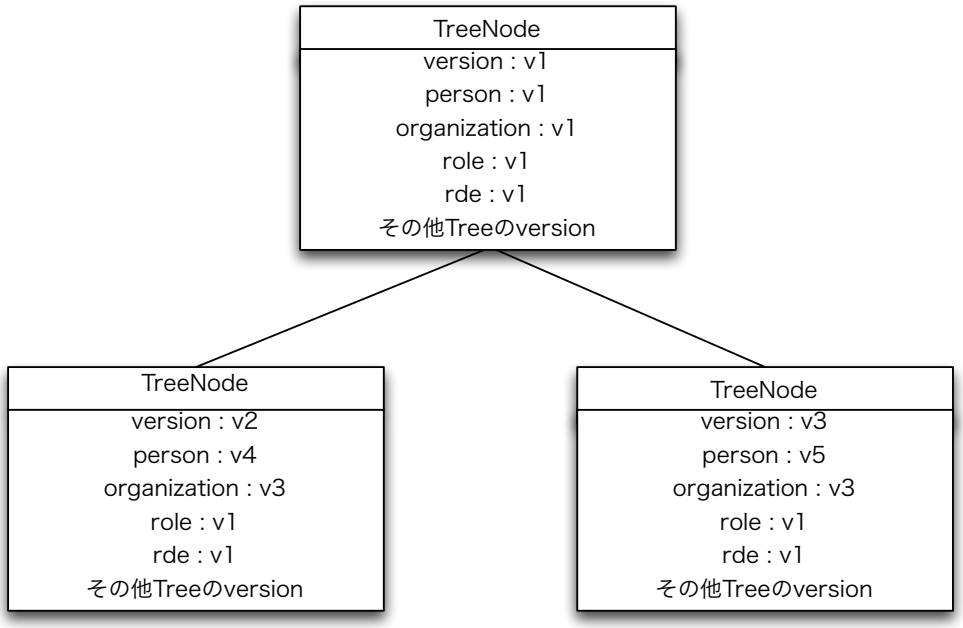

図 3.6: 構成情報モデル Tree 表現例

表 3.6: 構成情報 Tree の TreeNode が保持している Attribute

version	構成情報モデルの version
person	構成情報モデルの version に対応する人物 Tree の version
organization	構成情報モデルの version に対応する組織 Tree の version
role	構成情報モデルの version に対応する役割 Tree の version
rde	構成情報モデルの version に対応する役割記述要素 Tree の version

実際にどのように Jungle 上で過去の構成情報モデルにアクセスするか、例題を用いて説明する。構成情報モデル version:3 に対応する人物 Tree にアクセスする手順を以下に示す。

1. 構成情報 Tree(図 3.6) からアクセスしたい version 情報を保持している Node を取得する (今回の例題では version3)

2. 取得した TreeNode には、構成情報:version:3 に対応した人物 Tree などの version が記述されている (図 3.6) ので、人物 Tree の verson(v:5) を取得する

3. 2で取得したversionの人物Treeにアクセスする (図3.7)。

といった手順でJungleではmaTrixの構成情報モデルを表現する。

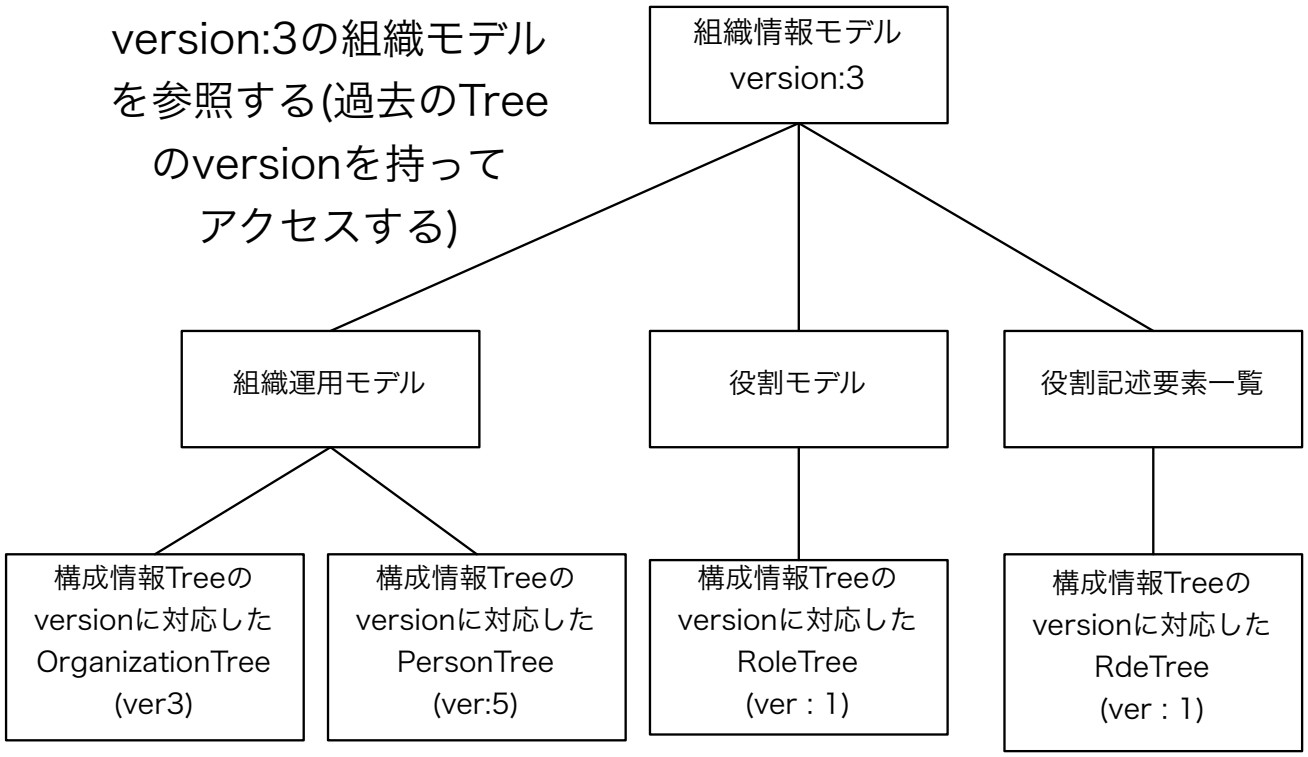

図 3.7: 構成情報モデル:3 に対するアクセス例

3.5 申請の許認可

maTrix を用いた許認可は、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権限を与えるかどうかを判断する。
4. maTrix は権限に応じて、ノート PC を A さんに貸出許可を与える。

といった流れになる。

maTrix の許認可を使用するメリットとして、許認可のログをとっておくことで、いつ、誰が、どのポリシーを元に、どんなことをしたか、の情報がいつでも取得可能であるため、不正アクセス等の問題発生時の解決等役立つ。といったメリットもある。

3.6 XACML

本節では maTrix のポリシーファイルの記述に用いられている XACML についての説明を行う。XACML は、データに関するアクセス要求について、その要求元の情報と要求の内容、アクセス対象の組み合わせから、その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記述できる。実際に使用する際は、XACML 自体がアクセス制御を行うのではなく、アクセス管理アプリケーションが XACML を参照しアクセス制御を行う。XACML は以下の様なデータ構造を持つ (図 3.8)

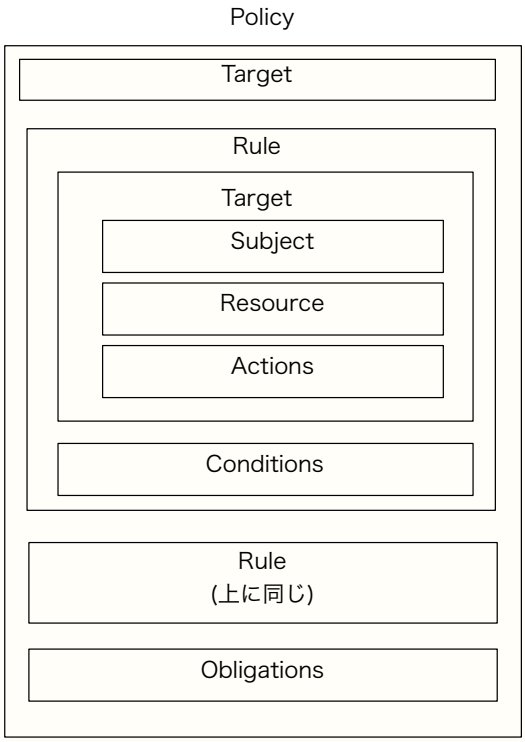

図 3.8: XACML のデータ構造

Target は、誰に対して (Subject)、どのような資源を (Resource)、どのように扱うか (Action) の 3 つの要素を持ち、それぞれの要素で、評価関数を用いて評価を行う。評価の結果は、Match、No-Match、Indeterminate(評価不能) の 3 つである。

Rule は、Target に対する規則を定めるもので、ルールの方針にそってアクセス要求がこのルールに適合した場合に決定する値 (Permit、Deny) を、Effect として設定しておく。また、Target とは別にオプションとして条件 (Conditions) を付けることも可能である (条件の例としては、アクセスを許可する時間の指定などがある)。ルールの評価は、Target の評価関数の結果と Condition の、ルール結合アルゴリズム (表 3.7) にそって結合する。

表 3.7: Rule の評価

Target	Condition	評価
Match	true	Efferct(Permit or Deny)
Match	false	NotApplicable
Match	Indeteminate	Indeteminate
No-Macth	-	NotApplicate
Indeterminate	-	Indeteminate

Policy

複数の Rule をまとめたものを Policy という。Policy の子要素には、Target、Rule、Obligations(責務) がある。もしも、Policy に Obligations が規定された場合は、たとえ、ルール結合後の結果が permit であったとしても、Obiligations に記述されていることを同時に実施することが出来なかった場合、承認を拒否する必要がある。また Policy が複数の Rule を評価するときは、ルール結合アルゴリズム (表 3.8) を用いる。

表 3.8: XACML のルール結合アルゴリズム

Deny-overrides	どれか 1 つのルールの Effect が Deny であれば結果は Deny とする
Permit-overrides	どれか 1 つのルールの Effect がが Permit であれば結果は Permit とする
First-applicable	ポリシー内のすべてのルールについて順番に評価して、対象がマッチした場合、対象の評価結果を Match とし、次に Condition を評価し、これが True なら結果は Effect で指定された Permit または Deny とする。

XACML の評価が終わり、最終的に Permit だった場合その申請は許可される。

3.7 Jungle 上の許認可

Jungle 上で maTrix の許認可を行う際は、ポリシーファイルを読み込み、各 Tree に対して検索関数を用いて、その結果から許認可を行う。

maTrix と同じ例題で、A さんが、学科のノート PC の借りるために貸出申請を行った際の許認可の流れを以下に記す。

1. 人物 Tree から、A さんのデータを取得し、A さんに割り振られている役割の ID を取得する。

2. 役割 Tree から、1 で取得した役割 ID のデータを取得し、その役割 ID に割り振られている役割記述要素 ID を取得する

3. 役割記述要素 Tree から、2 で取得した役割記述要素 ID のデータを取得し、役割記述要素名を取得する

4. 3 で取得した役割記述要素名が、琉球大学工学部学生かどうかを調べ、一致した場合は貸出を許可し、一致しなかった場合は貸出を許可しない。

といった流れになる。

第4章　仮

4.1　検索 API の実装

Tree に対する検索は、java8 の新機能である lambda 式を用いて find 関数を実装した。lambda 式を使用することで、匿名クラスを使う時より簡潔にコードを記述できるようになった。

find 関数は、以下の様に定義されている。

find 関数の定義

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

find 関数は引数に Query、String key、String value の 3 つの引数を取り、条件に一致した Node の Iterator を返す。第一引数には、探索の条件を記述する関数 boolean comdition(TreeNode) を定義した InterfaceQuery を。第二、第三引数の、String key、String value は Index を使うために使用する。

QueryInterface の定義

```
public interface Query {
      boolean condition(TreeNode _node);
}
```

以下に実際に lambda 式を用いた find 関数の使用例と実際の処理の流れを記す。

Jungle の Query 部分のソースコード

```
Iterator<TreeNode> pairPersonIterator =
traverser.find((TreeNode node) -> {
  String element = node.getAttributes().getString("element");
  if (element == null)
    return false;
    if (element.equals("Person"))
      return true;
    return false;
}, "element", "Person");
```

1. find 関数の処理の流れは、まず、第 2、第 3 引数の String key,String value を用いて、これらの値に対応した Index が登録されているかを調べる、Index がある場合は Index を使用し値を返す。Index がない場合は、Tree から深さ優先で TreeNode を取得していく。(図 4.1)

2. TreeNode を取得したら、boolean Query.condition(TreeNode) を実行する。この condition は find を使用するときに lambda 式で記述する。

3. condition の中では、TreeNode の Attribute に対して get を行い探索対象のデータを TreeNode が保持しているかを調べる

4. データを持っていた場合は True を、持っていなかった場合は False を返す

5. condition の返り値が、True だった場合取得した TreeNode を返す。False だった場合は次の TreeNode を取得し、2 に戻る。

find 関数は、上記の処理を探索が終わるまで繰り返す。

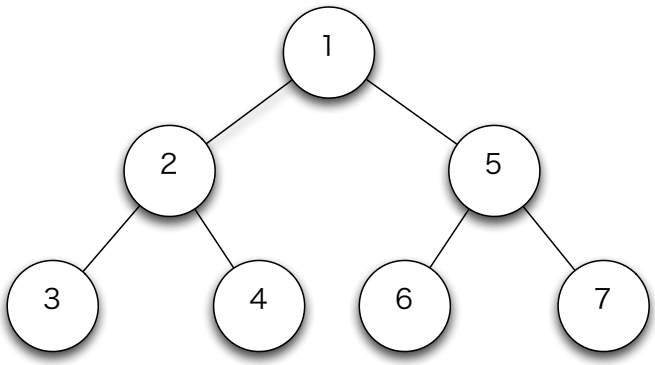

図 4.1: find の検索順序

検索 API は、他に特定の Node 以下に対して検索を行う findInSubTree(Query,node,key,value) も実装した。(図 4.2)

findInSubTree を使用することで、更に限定的な探索が行えるようになった。この関数も基本的な処理の流れは find 関数と同じである。また、maTrix 上に実装されていた、構成情報からデータを取得する Function も全て実装して、実際の maTrix と同じようにデータのアクセスを行えるようにした。

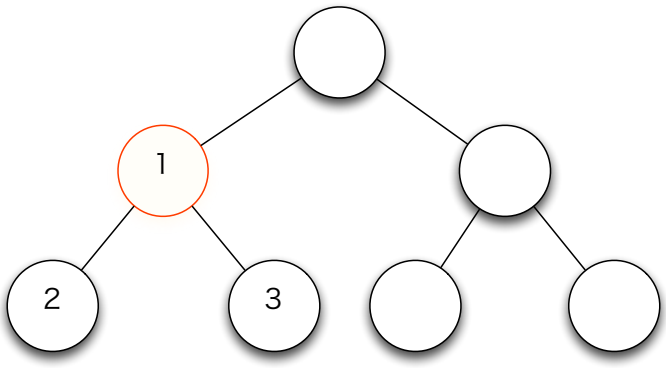

図 4.2: FindInSubTree の検索順序

4.2 Jungle上のIndexの設計

Jungleの探索はTreeを全探索するので、探索の計算量はO(n)である。そこで、Indexを使用することで効率よく探索を行えるようにする。Jungleは過去のTreeを全て保持しているため、Treeのversion毎にIndexを持っていることが望ましい。そこで、メモリの消費量を抑え、各versionのTreeにIndexをもたせる方法として、FunctionalJavaのTreeMapを使用したIndexの実装を提案する。

TreeMapは、KeyとValueのペアを用いて赤黒木を構築する。赤黒木の長所として、ソート済み二分木の探索なので計算量がO(logN)であることがあげられる。

それに加え、FunctionalJavaのTreeMapは、データの更新が行われた際に、一度作られたTreeに対して更新を行わず過去のTreeを再利用し、更新後のTreeMap新しく返すため、メモリの使用量を抑えつつ複数のversionのTreeMapを保持できる。

Indexの型を以下に記す。

Queryinterface

```
TreeMap$<$String key,TreeMap$<$String value,List$<$TreeNode$>>>>$
```

最初のTreeMap<String key,TreeMap>はIndexを格納するTreeMapである。このTreeMapに対しkeyでgetを行うと、keyに対応するIndexが登録されている場合、Indexを取得でき、取得したIndexに対しvalueでgetを行うと、valueの値を持つNodeのListが返ってくる。

ParentIndex

TreeNodeでgetを行うと、親Nodeを返すParentIndexを実装した。ParentIndexの型は、TreeMap<TreeNode,TreeNode>で定義されている。

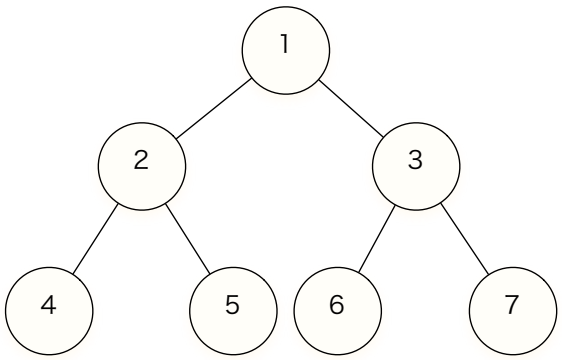

図 4.3: ParentIndexExample

上記の図4.3を用いたParentIndexの例を表4.1にまとめた。Jungleはノードの編集に、そのノードまでのNodePathを必要としたため、ParentIndexを実装するまでは、Indexの型を、TreeMap<String key, TreeMap< String value , List<Pair<TreeNode,NodePath>>>>

表 4.1: ParentIndex の返り値

返ってくる Node	渡す Node
Node2	Node4、Node5
Node3	Node6、Node7
Node1	Node2、Node3

と定義し、返り値を Pair<TreeNode,NodePath> というように、TreeNode とその TreeNode への NodePath のペアにすることで Node の編集を可能にしていた。しかし、Tree の編集を行った際に、TreeNode への Path は常に変動する為、Index 内の NodePath の更新コストが編集時のネックであった。しかし、ParentIndex を実装することで、TreeNode から NodePath を取得できるようになったため、Index に NodePath を入れる必要がなくなり、Index は今の型で定義できるようになった。また、ParentIndex も Index と同じように FunctionalJava の TreeMap を使用しているため、メモリの消費を抑えて、全ての version の Tree で保持することが可能である。

以下に ParentIndex を使用して、NodePath を取得する方法を示す。

1. ParentIndex を用いて親 Node を取得する。
2. 親 Node から子ノードのリストを取得する。
3. 子ノードのリストから、親 Node を get するのに使用した Node がある位置のインデックスを取得する。
4. 3 で取得した Index を Path に追加する
5. rootNode に辿り着くまで 1 - 4 を繰り返すと、対象への NodePath が取得できる。

4.3 過去の Tree に対するアクセス

Jungle 上で maTrix の構成情報モデルの表現を行う際に、過去の Tree にアクセスする必要があるが、Jungle には、過去の Tree に対し、アクセスする API は実装されていなかったため実装を行った。

Jungle は、クラス ChangeSet 内に Tree のデータを保持している。ChangeSet からは、今の revisionId と、1 つ前の Tree のデータを取得できるため、再帰的に過去の Tree のデータを取得できる。

アクセスしたい Tree の revisionId を引数に取り、過去の Tree のデータの取得と revisionId の比較を繰り返すことで過去の Tree にアクセスする、getOldTree(long revisionId) を DefaultJungleTree 内に実装した。

以下に getOldTree の実装部分のコードを示す。

getOldTree の実装部分

```
@Override
public Either<Error, JungleTree> getOldTree(long revision) {
  TreeContext tc = repository.get();
  ChangeSet cs = tc.getChangeSet();

  for (; cs.revision() != revision;) {
    cs = cs.prev();
    if (cs == null)
      return DefaultEither.newA(GetOldTreeError.OLD_TREE_NOT_FOUND);
  }

  TreeNode root = cs.getRoot();

  TreeContext oldTc = new DefaultTreeContext(root, cs);
  String oldTreeUuid = uuid + revision;
  JungleTree oldTree =
      new DefaultJungleTree(oldTc, oldTreeUuid, writer, treeEditor);
  return DefaultEither.newB(oldTree);
}
```

for 文の中で、過去の Tree の取得と、revision の比較を繰り返している。revision が一致した場合は、その version の Tree を返し、アクセスしたい revision を持つ Tree が見つからなかった場合は、エラーを返す。

4.4 JungleXMLReader

maTrix から XML 形式で書き出されたデータを Jungle に格納するための API として、XMLReader を実装した。JungleXMLReader の実装には sax(Simple API for XML) を用いた。

sax は、XML のタグやテキストデータを読んだ際にイベントを発生させながら、構文解析を行う。プログラマは、ContentHandler という特殊なイベントリスナを sax の parser に登録すると、発生するイベントを取得できる。イベントには、読み込んだタグの名前やテキストデータが含まれている。

表 4.2: sax の主要なイベント一覧

イベント名	呼び出されるタイミング
startDocument	XML の Parse が始まる時
endDocument	XML の Parse が終わった時
startElement	要素を読み込んだ時
endElement	要素が閉じられた時
Charactor	テキストが読み込まれた時

sax では、org.xml.sax.helpers.DefaultHandler という形で、ContentHandler のデフォルト機能が提供されているため、プログラマはこれを継承することで、必要なイベント処理のみを override して記述できるようになっている。

XMLReader で使用している ReadXmlHandler は、startElement、charactor、endElement、endDocument、の 4 つのイベントを使用しており、XML を読み込む際に、Tree を構築しながら Parse を行う。

startElement が呼ばれた時は、今いる地点の下に新しく Node を作りその Node へ移動し、Node に Attribute の値を格納する。charactor が呼ばれた時は、今いる Node にテキストデータを格納する。endElement が呼ばれたら、今いる Node の親ノードに移動する。

以上の 3 つの関数を用いて、XML のデータを Tree に格納していき、endDocument が呼ばれた時に木の Commit を行っている。データの格納は、3.2 章 Jungle 上での maTrix のデータ構造の表現述べたように、多少 Jungle に適合した形に書き換えてから行う。

4.5 XACMLInterpreter

XMLReader と同じよう sax にを用いて実装している。XACMLInterpreter は、引数に、使用する policyFile 名、どの Resource にアクセスするか、どんな処理を行うか、許認可を求める人の UserID 等を与える。

XAMLInterpreter で Handler が使用しているイベントは、XMLReader と同じで、startElement、charactor、endElement、endDocument の 4 つを使用している。

startElement では、主に評価関数や、評価関数の引数等を取得する。また評価関数が入れ子になっていることがあるため、functionStack と attributeStack を用いて評価関数と引数を関連付けている。図 (4.4) の例だと、function3 を実行した後、function3 の返り値と、function2 の引数の 2 つを使って function2 を実行する。

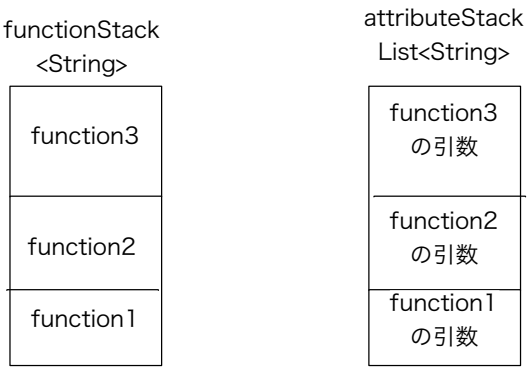

図 4.4: XACMLInterpreter で用いられている stack

charactor では、評価関数の引数を取得し、AttributeStack に push する

endElement では。主に評価関数の実行を行う。

endDocument では、これまでに実行した評価関数等の結果から今回の許認可を判断する。

XACMLInterpreter を用いることで Jungle 単体で XACML で記述されたポリシーファイルのテストが行えるようになった。

第5章　実装の評価

前章では、Jungle 上に maTrix を構築する方法と、構築に必要な API の実装について述べた。本章では、実装を行った Jungle の性能評価を行う。

5.1　実験環境

表 5.1: 実験環境

名前	概要
CPU	
Memory	
OS	
JavaVM	1.8.0-25

5.2 検索の API の測定

Jungle に対する検索 API の測定を行う。測定には、maTrix が保持しているデータにアクセスする際に用いる関数のうちの１つである、boolen isActive(String personId,String version) という関数を用いる。isActive は、第一引数で与えた PersonId を持つ人物が、第二引数で与えられた構成情報モデルの version に対応した人物 Tree にいるかどうかを調べる関数である。実験の結果は図 5.1 となる。横軸は人物 Tree にいる人の数を表しており、縦軸は探索にかかった時間を表している。

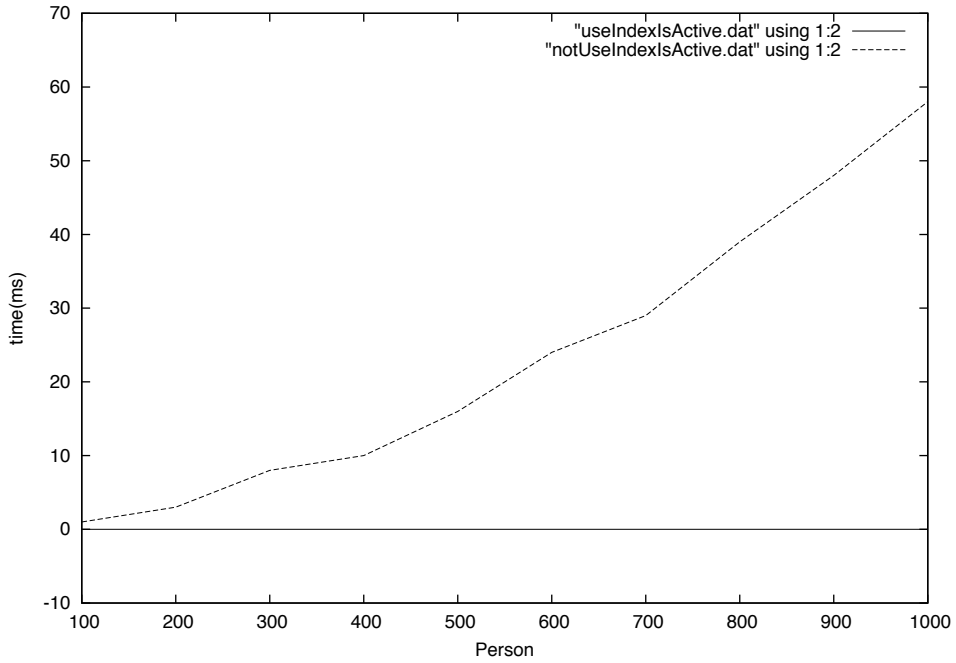

図 5.1: inActive の実行時間

isActive の実行時間は、Index を使用しない場合は、Person の数が増えると比例して増えていくのに対し、Index を使用すると Person の数が増えても実行時間は変わらなかった。この結果より、Jungle の Index の計算量は O(logn) であることがわかる。

第6章　結論

6.1　まとめ

本研究では、初めに当研究室が開発している分散データベース Jungle で使われている非破壊的木構造について述べ、破壊的木構造に比べてロックが少ないというメリットがあること、当研究室で開発している並列分散フレームワーク Alice を用いて分散実装を行っていることを述べた。

その後、株式会社 Symphony が開発しているアカウント管理、許諾判定システムである maTrix が、どのようなデータ構造を保持しており、どのようにアカウント管理を行い、申請の許認可を行っているかの説明を行って、Jungle 上で maTrix をどのように実装するかを述べた。その後 Jungle 上でどのように maTrix を表現するかの説明を行った。検索 API は、Java8 の新機能である lambda 式を用いて実装を行い、検索を高速化するため、functionalJava の TreeMap を用いて Index の実装も行った。過去の Tree に対するアクセスは、全ての Version の Tree が持っている固有の revisionId を使用しアクセスを可能にした。

また、maTrix が書きだした xml ファイルを読み込むために XMLReader を sax を用いて実装し、XML ファイルを Jungle に取り込めるようにした。その後、XACMLInterpreter を実装し、maTrix が使用しているポリシーファイルを読み込んで、Jungle 上で許諾判定を行えるようにした。

最後に、実際に maTrix がデータにアクセスする際に使用するアクセス関数を実装し、性能評価を行った。1 つ目の実験では Index の性能評価を行うため、Index を使用する場合と、使用しない場合で検索を行い、アクセス関数の実行速度の比較を行った。2 つ目の実験では、データの検索 API に対して複数の Thread から負荷をかけ、並列に動作するかどうかを調べ、どちらの実験でも Index の性能は出ていた。

6.2 今後の課題

6.2.1 push/pop

6.2.2 index の IncrementalUpdate

6.2.3 differencialList

6.2.4 exponential backoff

参考文献

[1] hoge

謝辞

本研究の遂行，また本論文の作成にあたり、御多忙にも関わらず終始懇切なる御指導と御教授を賜わりました河野真治助教授に深く感謝したします。
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合木茂さん、株式会社 PCI ホールディングスの皆さまに深く感謝いたします。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、数々の貴重な御助言と細かな御配慮を戴いた、大城信康さん、杉本優さん、並びに並列信頼研究室の全てのメンバーに深く感謝致します。
最後に、有意義な時間を共に過ごした情報工学科の学友、並びに物心両面で支えてくれた両親に深く感謝致します。

2015 年 3 月
金川竜己