情報特別演習最終レポート

筑波大学情報学群情報科学類 (201611350) 江畑 拓哉

January 31, 2018

Contents

1 概要 **2**

2 序論 **3**

3 時系列データベースの比較と OpenTSDB の利用法 **3**
- 3.1 InfluxDB . . . 4
- 3.2 Graphite . . . 5
- 3.3 Datomic . . . 6
- 3.4 OpenTSDB . . . 9
  - 3.4.1 HBase とその周辺知識 . . . 9
  - 3.4.2 HBase . . . 9
  - 3.4.3 Hadoop . . . 10
  - 3.4.4 Zookeeper . . . 10
  - 3.4.5 OpenTSDB . . . 11
  - 3.4.6 OpenTSDB の HTTP API . . . 12

4 Clojure を用いた JVM における高速計算技法 **12**
- 4.1 Clojure 自身の高速化手法 . . . 14
- 4.2 ClojureCL . . . 14
- 4.3 Neanderthal . . . 14
- 4.4 Clojure.core.matrix . . . 17
  - 4.4.1 vector-z . . . 19
  - 4.4.2 clatrix . . . 19
- 4.5 ACM3 (Apache Common Math 3) . . . 19

5 ARIMA モデルによる時系列分析 **19**
- 5.1 ARIMA モデルの概要 . . . 19
- 5.2 BackShift 記法 . . . 19
- 5.3 単位根検定 . . . 19
  - 5.3.1 ADF 検定 . . . 19
  - 5.3.2 KPSS 検定 . . . 19

5.4 AR モデル 19
5.5 MA モデル 19
5.6 係数推定 19
5.6.1 対数尤度 19
5.6.2 AIC 19
5.6.3 最小二乗法 19
5.6.4 アメーバ法 19
5.7 SARIMA モデルについて 19

6 Clojure/ClojureScript を用いた Web 開発 19
6.1 Clojure によるバックエンド開発 19
6.1.1 Luminus Framework 19
6.1.2 Swagger UI 19
6.2 ClojureScript によるフロントエンド開発 19
6.2.1 基本的な開発 19
6.2.2 Reagent 19
6.2.3 core.async による非同期処理 19

7 MKKL の開発 19

8 発展 : ARIMA 推定 と Random Forest による予測 19
8.1 概要 19
8.2 実験方法 19
8.3 実験結果 19
8.4 考察 19

9 まとめと今後の課題 19

1 概要

今情報特別演習において私は、機械学習を初学者が学ぶための Web アプリ開発を行った。当初は大規模データベースを用いた機械学習 API を作るという目標であったが、特に機械学習を学んでいくにあたり、これの中身を理解するための初学者向けの解説の供給が少ないと感じたため、開発目標をやや変更した。

今回学習した内容は、 ① 大規模データベースの、特に時系列データベースの比較とその利用法 ② 機械学習等を実装する際に重要となる高速計算を JVM 言語内で行う手法 ③ 代表的な時系列分析の一つである (S)ARIMA モデル ④ 開発言語として取り上げた Clojure/ClojureScript の、言語自体・これを用いた Web 開発手法、の 4 分野である。

結果として 時系列データベースとして OpenTSDB を採用し ARIMA モデルとそれに付随する ADF 検定などを実装・解説を作成した。更に JVM 上で高速計算を行うために、*jblas* や *Intel®MATH KERNEL LIBRARY* を用いた GPU 演算 、 OpenCL の利用例を調べ比較し、一部を Web アプリの実装に活用した。そして *Clojure* という言語を身に着け、Clojure/ClojureScript を用いて JavaScript のライブラリである *React.js* など

を利用する手法についてまとめ、これらを利用して Web アプリの概形を実装した。

2 序論

この情報特別演習の初期テーマの決定はグループメンバーからの提案が元であった。その概要は、大規模データベースである *Apache HBase*™ (以降 “HBase” と呼称する) を用いて入力されたデータの因果関係を分析、予測する機械学習 API を作成するというものである。ここから因果関係と相関関係の違いについて学習し、機械学習手法について吟味した結果、ニューラルネットを用いた学習手法と、(S)ARIMA モデルを用いた時系列データ予測と Random Forest を用いた欠損値補完を組み合わせたものの 2 つの手法が議題に上がったが、後者を選択することになりこれを研究することになった。その中で機械学習を学ぶ際にその内部を知る必要があり、学んでいく際にその資料の供給が少ないことを感じ、その資料も兼ね備えたいと考え、また機械学習結果を視覚的にわかりやすく伝える目的も合わせて、Web アプリという形で開発を行うことに目標を定めた。

機械学習という名前は世間に流布しており、それを用いた API として例えば Microsoft Azure の $ComputerVisionAPI$ 等を上げることができるが、この中身が解説されることはその必要性や機密性の問題から非常に少ない。また Python や R といった言語やそのライブラリ等に付属している統計処理、機械学習の関数などもその殆どが簡便化されており、例えば $auto.arima()$ 関数などはその中身に踏み込むことなく実行、モデルの比較を行うことができる。これは機械学習の利用者という立場からすれば素晴らしい進歩であると考えられるが、その反面機械学習を学ぶ立場になった場合、その中身を知らない状態で API や関数を利用することができるため、手放しに機械学習を理解できたと誤解してしまう可能性がある。

このため主要な機械学習についてその中身を学習できるツールの作成は、統計や機械学習を学びたいと志している、或いは先述のようなツールの中身について興味を持った者に対して、一定の需要があるのではないかと考えている。

3 時系列データベースの比較と OpenTSDB の利用法

一般的に時系列データのような、単調増加する要素を持つデータを通常の大規模データベースに保存することはデータの分散という点から問題が発生する。 [5] このため時系列データを扱うためのデータベースを考える必要がある。幸いなことに、初期案にあった HBase に対して時系列データを扱うことができるように拡張した $OpenTSDB$ というデータベースがある。OpenTSDB は HTTP API として操作が許されており、また保存されているデータを確認することが容易であるように設計されている。今回は HBase を完全分散モードで利用する、というチームメンバーの目標に沿ってこちらのデータベースを採用した。

他の時系列データベースの提案として考えられるものに、① $InfluxDB$ 、② $Graphite$ 、 ③ $Datomic$ などを挙げることができる。

この章ではそれぞれのデータベースの特徴と利用法を簡潔にまとめる。

3.1 InfluxDB

InfluxDB は InfluxData が開発を行っているデータベースであり、高機能なクラウドシステムを有料で使うことができるほか、オープンソースソフトウェアとしての利用も可能である。このデータベースの利点の一つに、利用方法が簡単であることが挙げられる。シングルノードでの利用に関してのみに焦点を絞れば、2017 年 12 月において *Arch Linux* でのインストールは、パッケージのインストールとサービスの起動の 2 つのコマンド で利用可能になる。またデータベースの読み書きに関しては、SQL に近い記法を用いた HTTP API を用いて行うことができ、今回の演習における開発言語である Clojure で必要な部分に関するラッパーを書くことは非常に簡単であった。また TICK stack と呼ばれる InfluxDB を含む時系列データを扱うための環境を追加でセットアップすればデータのモニタリング、収集、リアルタイム処理をより効率的に行うことができる。本演習の初期目標ではデータの入力をユーザが行うことができる設定になっていたため、悪意あるデータを監視することが容易であるという点、データベースの外側の分野までの広いサポート環境があるという点からこのデータベースは非常に魅力的である。

このデータベースが扱うデータモデルの概要を以下に示す。

Table 1: InfluxDB data model

name(required)		
timestamp (required)	fields (required)	tags (optional)
⋮	⋮	⋮

それぞれの用語についてその意味と例を挙げると以下のようになる。

- name データの名前 (ex. 日経平均株価)
  データの名前であり、何に関してのデータであるかを表す。

- timestamp 時刻データ (ex. 2018-01-27T00:00:00Z)
  時刻データであり、いつのデータであるのかを示す。この場合の “いつのデータ” とは、データの登録日時ではなく、そのデータの発生日時である。

- fields 測定値群 (ex. (終値：12000) (始値：11000))
  そのデータが持つ値を示す。いくつかの属性に従って複数の値を格納することができるが、ここに登録されるデータは索引付けされるべきものではないという点でタグ群と意味が異なる。

- tags タグ群 (ex. (記録者：A) (ソース：東京株式市場))
  　そのデータの持つ属性や追加情報を示す。ここに登録されるデータは索引付けさており、データの絞り込みを行う目的に用いられる。

3.2 Graphite

Graphite は Python を中心にして書かれた時系列データベースであり、同じく Python の Web フレームワークである Django と組み合わせることが、 Graphite 自身の Web UI コンポーネントが Django であるという点から、非常に容易である。同時に Python は機械学習に関する API・ライブラリ が豊富に存在しているため、本演習が純粋に “Web API の作成” のみの目標であったならば当然こちらを用いて開発を行っていただろう。またデータベースの導入自体も、 Python のパッケージ管理システムである pip を用いて行うことができることから、純粋に Python のみによってすべてを解決することができる。更に Graphite のドキュメントは豊富に存在しており、例えば Monitoring with Graphite [1] を挙げることができる。

Graphite の内部について簡単に説明を行うと、主に 4 つのコンポーネント、Carbon、Whisper、Cario、Django を中心に展開する。

- Carbon は、後述するデータベースそのものと言える Whisper にデータを登録する役割を担っており、メトリクス[1]のバッファリングを行ったり他のデータベースにメトリクスをリレーさせたりすることができる。

- Whisper は、入手したデータをファイルシステムに書き込み・読み出しを行う役割を担っており、この部分は Ceres と呼ばれるコンポーネントに置き換えることができる。両者の違いは、Whisper が保存領域を固定サイズとして確保するのに対して、Ceres は任意のサイズで保存領域を確保できるということにある。

- Cario は、Graphite のグラフィックエンジンを担当しており、保存されているデータを視覚化する上で非常に重要な役割を果たしている。

- Django は、Cario によって出力されたデータを表示する役割を担っており、データを扱う開発者はこの部分を見てデータを確認することになる。

　このデータベースが扱うデータモデルは階層構造を取っており、一例を紹介すると以下のようになる。

　“stock_price.nikkei_index.close_price 12000 1517055464”

[1]metrics: 入手したデータを分析して数値化したもの

上の文字列を送信することによって、stock_price の中の nikkei_index の中にある close_price という階層に 12000 という値を Unix 時間である 1517055464 のデータとして登録している。つまりこのデータは以下のようにな形に保存されたと考える。

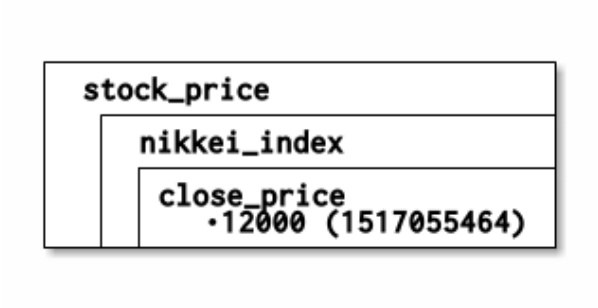

Figure 1: Graphite example

3.3 Datomic

Datomic は他のデータベースとはかけ離れた設計が行われた新しい世代の分散型データベースである。 Clojure の作者である Rich Hickey 氏らが作成し、有料でメンテナンスとアップデートが付属されたクラウドシステムを使うことができる。また一年に限っては無料でこの機能を利用することもできる。これとは別に存在する無料版に関しては分散できるピア数などの制限がかかる。
Datomic には 2 つの目標「情報を時間によって紐付け蓄積する」 「データベースアプリケーションのモデルをリモートアクセスするものからそれぞれのプログラムの中にあるものとする」[1] がある。この考え方によって得られた大きな 2 つの特徴に、① Append-Only ②データベースに独立したクエリーエンジンがある。

Append-Only とはその名の通り、追加のみという意味で言い換えれば変更ができないということを意味する。これは情報を時間に紐付けることによって最新の情報を見ることができるため、情報を “書き換える” 必要がなくなったためにできたことであり、トランザクション処理などのデータの管理を容易にすることができる。

データベースに独立したクエリーエンジンとは、アプリケーション側でトランザクションやクエリ処理を実行するという意味を示しており、データベースに HTTP API などを用いてクエリを投げデータベース側がそのクエリを処理して結果を送信していたものをアプリケーション側に移す、ということになる。その意味で Datomic はアプリケーション側をピア [2] と呼称する。

ピアが扱うデータはデータベースではなくピア側のキャッシュに Read Only な形で LRU [3]形式で保持される。データベースは書き込まれたデータを保存し、更新があればそれぞれのピアが持っている、データベースに対して常に開いているノードに告知し、アプリケーション側から要求されるデータ群をそのまま返すことになる。これによってピア側のメモ

[1] http://endot.org/notes/2014-01-10-using-datomic-with-riak/
[2] peer
[3] Least Reacently Use

h

Table 2: Datomic の特徴

目指すもの	· 情報は時間によって紐付ける · データベースアプリケーションのモデルをそれぞれのプログラム内に移動する
大きな特徴	· Append-Only データベース · データベース側ではなくアプリケーション側にクエリ処理エンジンがある

リキャッシュを疑似データベースとして貪欲に使うことができ、データベースのボトルネックを解消することができるようになっている。更にピア側のキャッシュ上のデータベースは実質ゼロコストで用いることができるため、 LRU が最適であるような目的のアプリケーションにこのデータベースを適用した場合、データへのアクセスという点において他のデータベースに性能で劣ることはない。またクエリ処理を分散しているため、多くのクエリ処理をこなさなければならないピアが増えたとしても、キャッシュ上のデータを使っている限りはその処理によってデータベースに負荷がかかることもない。またデータベースの更新をピアに告知しなければならないという点でデータベースへの書き込みがネックになる可能性もあるが、これは論理的に分かれているデータごとにデータベースそのものを分割することで解決することができる。

データベースのアクセス方法は Datalog と呼ばれる Clojure らしいシステムによって扱われるため、SQL に慣れている場合には苦労する可能性があるが、アプリケーションに柔軟に組み込むことができる。これはデータがキャッシュ上に Read-Only な形で存在しているという特性と、 Clojrue が関数型言語の側面を持っているという点を考えれば、データベース上のデータを手元にあるデータであるかのように利用することができるということを意味している。また保存しているデータは必ず Datom という最小単位に分割されており、これを元にして様々な形にデータを変形させることができる。

このデータベースが扱うデータ例を以下に示す。

```
{:nikkei-index/type “close-price”
  :nikkei-index/value 12000
  :nikkei-index/timestamp 1517055464}
```

データは nikkei-index/type に対する値として “close-price” が格納されている。nikkei-index に “close-price” が含まれているわけではない。

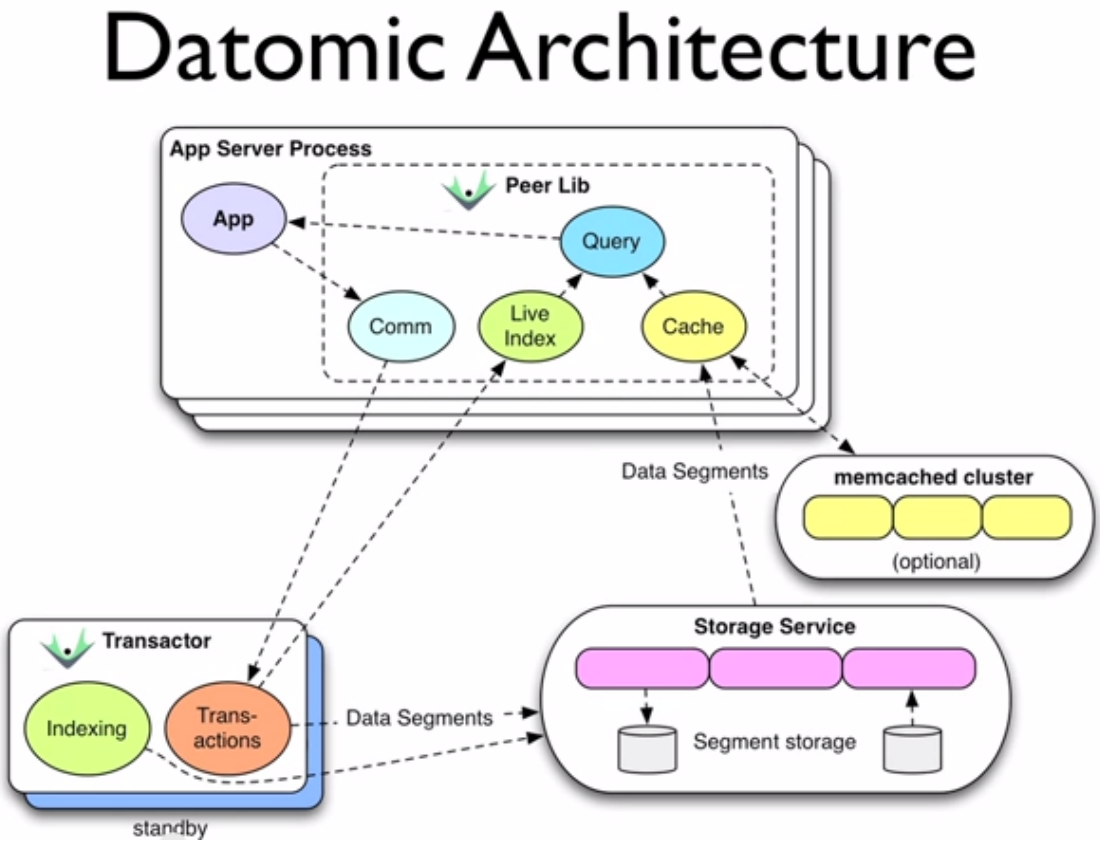

Figure 2: Talk Notes: Using Datomic With Riak より

3.4 OpenTSDB

OpenTSDB の特徴の説明、セットアップや利用方法に関して説明を行う前に、その基盤である HBase とその周辺知識について簡単にまとめ、その後 OpenTSDB についての説明を行う。

3.4.1 HBase とその周辺知識

HBase とは *Apache*™ *Hadoop*® (以降 “Hadoop” と呼称する) と呼ばれる、大規模データの分散処理フレームワークのためのデータベースである。そして Hadoop の分散サービスを形成するために *Apache Zookeeper*™ (以降 “Zookeeper” と呼称する) という管理ツールが使われる。

3.4.2 HBase

HBase は NoSQL の一つである。NoSQL は大別して、①キーバリュー型②ワイドカラム型③ドキュメント型④グラフ型、があり HBase はワイドカラム型[4]に属している。

Table 3: ワイドカラム型の例 (Name 列を取り出すこと等を得意とする)

ID	Name	Email	Birthday	Authorization
001	Bob	bob @ foo.com	1998/01/02	true
002	John	john @ bar.com	1987/02/01	false
⋮	⋮	⋮	⋮	⋮

Hadoop の HDFS (Hadoop Distributed File System) の補完を担っており、複数台のマシンのディスクを一台のディスクであるかのように扱うことができる。全体のデータは Region という単位で分割されており、これをそれぞれのディスクに 1 つ以上割り振っていくことで分散を行う。

続いて HBase の論理データモデルについて説明を行う。最上位概念は Namespace と呼ばれるもので、この中には Table と呼ばれるデータを表形式で保持している概念を 1 個以上含んでいる。 一つ以上の RowKey 、一つ以上の ColumnFamily で構成されている。そして ColumnFamily には一つ以上の ColumnQualifier が存在している。行キーである ColumnQualifier と列キーである RowKey の交差点にはそれぞれ Cell と呼ばれる領域があり、ここにデータが格納されることになる。データは Timestamp とともに保存されており、 Cell にはそのデータが重ねて保存される。つまり Cell には Timestamp に紐付けられたデータが複数存在することになる。また、ワイドカラム型であるという特性上、Table は Rowkey でソートされた状態で保存されることになる。

HBase の物理モデルの Table の構造はキーバリュー形式で保存されている。物理モデルの詳細はデータの分散などの説明も必要となるが、これ以上の内容は本演習で理解する

[4]簡単に説明するとデータを行ごとではなく列に対して管理しており特定の列を取り出して処理することに最適化されており、高いパフォーマンスやスケーラビリティを持っている。

ことができなかったため説明を省略する。

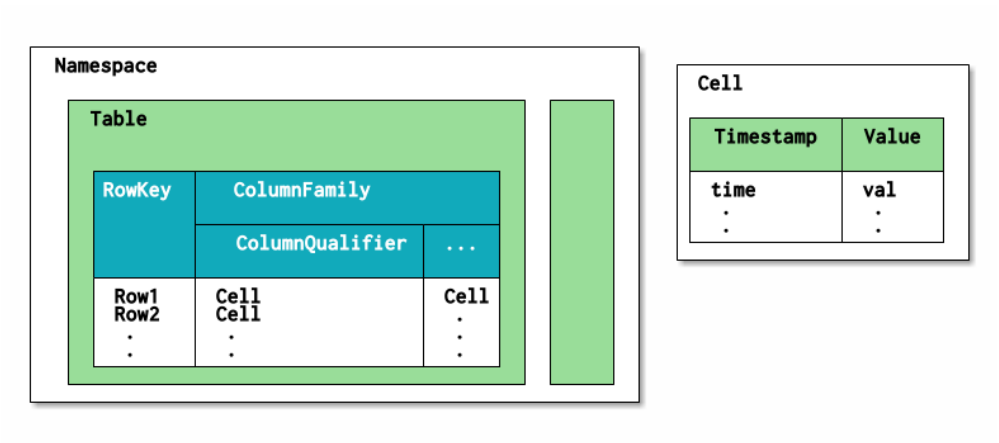

Figure 3: HBase の論理データモデル

3.4.3 Hadoop

Hadoop は大規模データセットの分散処理フレームワークである。 Hadoop はモジュール化されているため、そのコンポーネントの殆どを別のソフトウェアに入れ替えることもできる柔軟な設計がされている。今演習では標準的な Hadoop の構成に付随してインストールされる、①Hadoop Common ②Hadoop YARN[5] ③Hadoop MapReduce ④Hadoop Distributed File System (HDFS) をそのまま利用している。

Common は他のモジュールに利用される基本的なライブラリ群である。 YARN は Hadoop のリソース管理やスケジューリングを行い、 MapReduce は分散処理のためのフレームワークである。 HDFS は分散ファイルシステムで、大容量ファイルを扱うことができる。HDFS は大量の小さなデータを高速に扱うことを不得手としているので、HBase がこの補完を行っている。

3.4.4 Zookeeper

Zookeeper は Hadoop などにおける、構成情報の管理、分散処理の提供、またグループサービスの提供なども行う、分散アプリケーション全体を管理するツールである。使用用途は多岐にわたり、例えば Hadoop などにおける構成管理、 Apache Storm™ [6] における処理の同期などに用いられる。ツリー状の階層化された名前空間を持ち、ノードと呼ばれる要素にサーバなどを割り当てている。高速処理や高い信頼性があるにもかかわらず、非常に簡単な API を持っていることが特徴である。ベンチマークとしては Zookeeper 3.4 Documentation に記載されている。

[5] Yet Another Resource Negotiator
[6] リアルタイム高速分散処理フレームワーク

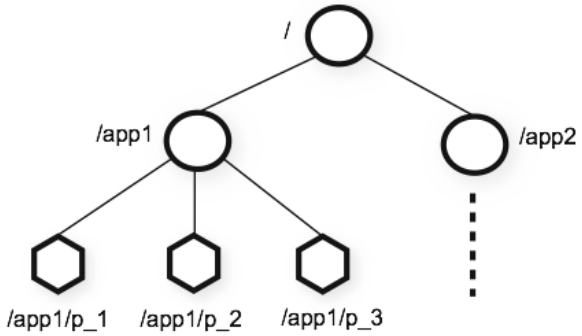

Figure 4: Zookeeper の階層構造

3.4.5 OpenTSDB

OpenTSDB とは HBase をホストとした [7] 時系列データベースで、その構成は① 時系列デーモン (以降 TSD と呼称する) ② コマンドラインユーティリティ、の 2 つである。特徴としては TSD にマスター・スレーブといった上下関係がないこと、HBase などのホストに各アプリケーションが直接触れる必要がないこと、標準的に保存されているデータをブラウザから視覚的に確認することができることなどが挙げられる。

これによって得られる恩恵として、アプリケーションをチームで開発・維持する際に OpenTSDB を軸にしてデータベース側とアプリケーション側に分割することができるということが考えられる。例えばアプリケーション側はデータベース側の分散等の開発が終わる前に仮設置の HBase に対して OpenTSDB を適用し、アプリケーションをほぼ本環境と同じように動かすことができる。またデータベースの分散数を増やしたい場合は、データベース側にのみ視点を当てて変更を行うことができる。

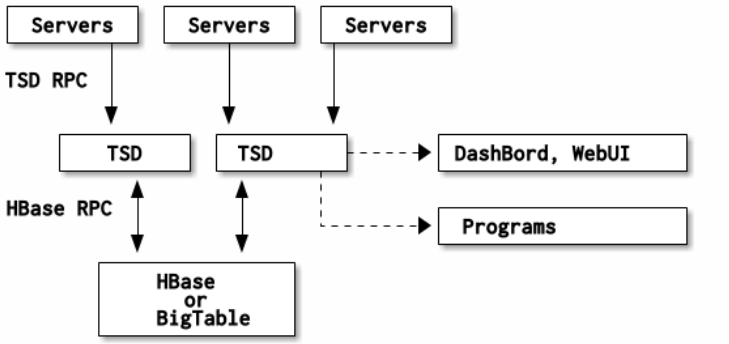

Figure 5: OpenTSDB の概略図

[7] 正確には Google の BigTable もホストとなりうる

OpenTSDB の論理モデルは Metric と呼ばれるその時系列データのタイトルとも言える概念が最も外側に位置しており、この中にテーブルに近い構造が一つ含まれていると考えることが出来る。このテーブルの行キーはソートされたタイムスタンプであり、時系列データベースの要である。テーブルの列キーはタグと呼ばれるキーバリュー形式の識別子が 0 以上割り当てられており、これによって欲しいデータの絞り込みを行うことができる。

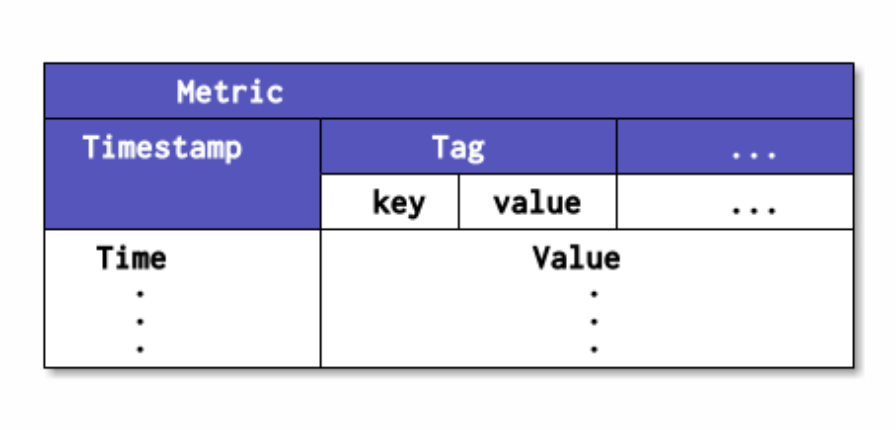

Figure 6: OpenTSDB の論理モデル

OpenTSDB はそのアクセスを HTTP API を用いて行うことができる。以降にその概要をまとめる。

3.4.6 OpenTSDB の HTTP API

OpenTSDB を利用するにあたって重要な要素に HTTP API の習得がある。このクエリによってアプリケーション開発者はデータの取得や送信を行うことになる。尚、HTTP API を使わずに Telnet を用いる手段もあるが、どちらも機能として同等であるためここでは HTTP API についての説明のみに留める。

API は、データの取得に関してはクエリ文字列とボディ部の両方の手段をサポートしており、ボディ部を用いる場合はクエリ文字列を用いるよりも詳細な検索をかけることが出来る。対してデータの送信は PUT メソッドによるボディ部を用いた手段のみが利用できる。 それぞれの具体例を示すと以下のようになる。

Table 4: OpenTSDB におけるクエリ例①

前提条件 クエリ内容	· http: //localhost:4242 に対して OpenTSDB が開いている 1 年前から現在までの Metric nikkei-index における タグについて key が “type” 、value が “close-price” であるデータを要求する
クエリ文字列	· http: //localhost:4242/api/query \ ?start=1y-ago&m=avg:nikkei-index{type=close_price}
ボディコンテンツ	· http: //localhost:4242/api/query · Content-Type JSON · Body { ”start” : 1y-ago, ”queries” : [{”aggregator” : ”sum”, ”metric” : ”nikkei-index”, ”tags” : {”type” : ”close-price”} }] }

Table 5: OpenTSDB におけるクエリ例②

前提条件 クエリ内容	· http: //localhost:4242 に対して OpenTSDB が開いている · Metric “nikkei-index” の、 タグが、key は “type”、value は “close-price” である UnixTime が 1517055464 である時間に、 12000 という値を保存する
	· http: //localhost:4242/api/put · Content-Type JSON · Body { ”metric” : ”nikkei-index”, ”timestamp” : 1717055464, ”value” : 12000, ”tags” : {”type” : ”close-price”} }

4 Clojure を用いた JVM における高速計算技法

本演習の開始時、自分のこれまでのプログラム言語学習経歴[8] から、 Lisp の影響を受けた言語を選択することが最も演習に適していると考えており、更に HBase を活用することが決定していたため、 Java に近い Lisp に近い言語として JVM[9]上で動作する Clojure を採用した。また演習を勧めていく上でフロントエンドの開発も行う必要が出てきたため、同様のシンタックスを用いる ClojureScript も採用し、この両方の言語を中心に学習した。

この章ではその内の Clojure における高速計算手法についての学習成果を完結にまとめる。

4.1 Clojure 自身の高速化手法

Clojure に GPU ライブラリ等を適用する以前に純粋な Clojure で最適化されたコードを書くことが高速計算を行う際に重要であることは言うまでもない。本演習では Clojure 自身の学習も兼ね Clojure for the Brave and True [2] 、Clojure High Performance Programming [3] を教材に Clojure の最適化手法を学習した。具体的な学習内容としてはプログラム設計の見直しや基本的なシンタックスの見直し、効率の良いスレッド化・並列処理、プログラム全体のパフォーマンス測定法 (プロファイリング法) などである。成果としてどこまでの性能向上が認められたかを具体的に比較することは難しいが、性格の良いプログラムを書くことが出来るようになったのではないかと考えている。

4.2 ClojureCL

ClojureCL とは Clojure で OpenCL を用いるためのライブラリで C 言語で書かれる OpenCL のコードよりも簡潔なシンタックスで書くことが主張されている。このライブラリは JVM 上で OpenCL を動作させるため、 JNI[10] を基盤としたライブラリである、jocl を用いており、非常に低レベルな部分で OpenCL とリンクしているため、OpenCL そのものの知識が必要となるものの、その速度を十分に体感することが出来る。本演習ではドキュメントに記載されたソースコードを移し、自分の環境においてそれを体験するまでを行った。2018 年 2 月 1 日においてはより深い理解を行うために、OpenCL in Action [4] を学習している。

4.3 Neanderthal

Neanderthal は、Intel ® MKL を用いた高速行列演算・線形代数のためのライブラリである。その速度は GPU を利用するモードの場合には大規模サイズの行列演算に関しては、 Clojure / Java ライブラリに対しても大凡 3000 倍の高速化を達成し、CPU を利用する場合においても純粋な Java よりも 100 倍の高速化を達成している。この高速化に関しては後述する Clojure.core.matrix 系ライブラリに対してやや扱いが難しいが、その分

[8] 昨年の情報特別演習においては Python を中心に利用し、授業外で Common Lisp をある程度習得した
[9] Java virtual machine
[10] Java Native Interface

大幅な高速化が望むことが出来る。

Single precision floating point (vs jBlas single precision):

Neanderthal and jBlas run on 4 cores, Vectorz doesn't have parallelization.

Since Clatrix does not support single-precision floating point numbers, I did this comparison with jBlas directly for reference (Neanderthal is still considerably faster :), *but keep in mind that you can't use that from core.matrix*.

Matrix Dimensions	Neanderthal	jBLAS	Vectorz	Neanderthal vs jBLAS	Neanderthal vs Vectorz
2x2	232.36 ns	362.00 ns	61.36 ns	1.56	0.26
4x4	237.72 ns	369.99 ns	129.34 ns	1.56	0.54
8x8	253.22 ns	476.57 ns	568.02 ns	1.88	2.24
16x16	372.30 ns	598.43 ns	3.45 µs	1.61	9.27
32x32	903.14 ns	1.37 µs	23.44 µs	1.52	25.96
64x64	2.80 µs	7.52 µs	218.64 µs	2.69	78.21
128x128	16.30 µs	31.48 µs	1.55 ms	1.93	94.85
256x256	126.25 µs	191.15 µs	12.28 ms	1.51	97.24
512x512	1.07 ms	1.25 ms	96.94 ms	1.16	90.21
1024x1024	7.93 ms	10.63 ms	778.46 ms	1.34	98.12
2048x2048	57.47 ms	104.95 ms	6.22 sec	1.83	108.16
4096x4096	470.12 ms	568.46 ms	50.06 sec	1.21	106.49
8192x8192	3.76 sec	4.85 sec	6.68 min	1.29	106.56

Figure 7: ベンチマーク Neanderthal Benchmarks より

残念ながら本演習では成果物を稼働させるサーバをどのように扱うかについて協議が不足しており、必要とされるライブラリがその環境で入手することが出来るか不明であったため、実装に組み込ませることができなかったものの、実行例の一部をここで紹介することとする。

Listing 1: test-Neanderthal.clj

```
(ns test-neanderthal.core
  (:require
   [uncomplicate.neanderthal.core :refer :all]
   [uncomplicate.neanderthal.native :refer :all]
   [uncomplicate.neanderthal.linalg :refer :all]))

;; ------------------------------------------------------------
;; sample1
(def a (dge 2 3 [1 2 3 4 5 6]))
;; #RealGEMatrix[double, mxn:2x3, layout:column, offset:0]
;;           ↓       ↓       ↓       ┐
;; →       1.00    3.00    5.00
;; →       2.00    4.00    6.00
;;  └                             ┘

(def b (dge 3 2 [1 3 5 7 9 11]))
;; #RealGEMatrix[double, mxn:3x2, layout:column, offset:0]
;;           ↓       ↓       ┐
;; →       1.00    7.00
;; →       3.00    9.00
;; →       5.00   11.00
;;  └                     ┘

(mm a b)
;; #RealGEMatrix[double, mxn:2x2, layout:column, offset:0]
;;           ↓       ↓       ┐
;; →      35.00   89.00
;; →      44.00  116.00
;;  └                     ┘

;; ------------------------------------------------------------
;; sample2
(def A (dge 3 2 [1 0 1 1 1 2]))

(def or (qrfp A))
;; #RealGEMatrix[double, mxn:3x2, layout:column, offset:0]
```

```
;;          ↓        ↓       ┐
;; →        1.41     2.12
;; →       -0.00     1.22
;; →       -2.41     3.15
;; └                         ┘

(def r (dge 2 2 (:or or)))
;; #RealGEMatrix[double, mxn:2x2, layout:column, offset:0]
;;          ↓        ↓       ┐
;; →        1.41     2.12
;; →       -0.00     1.22
;; └                         ┘

(def q (org or))
;; #RealGEMatrix[double, mxn:3x2, layout:column, offset:0]
;;          ↓        ↓       ┐
;; →        0.71    -0.41
;; →        0.00     0.82
;; →        0.71     0.41
;; └                         ┘

(def b (dge 3 1 [1 0 -2]))

(def x (mm (tri (trf r)) (trans q) b))
;; #RealGEMatrix[double, mxn:2x1, layout:column, offset:0]
;;          ↓       ┐
;; →        1.00
;; →       -1.00
;; └                ┘

;; ------------------------------------------------------------
;; sample2 ~another solution~
(def A (dge 3 2 [1 0 1 1 1 2]))

(def b (dge 3 1 [1 0 -2]))

(def x_ (dge 2 1 (ls A b)))
;; #RealGEMatrix[double, mxn:2x1, layout:column, offset:0]
;;          ↓       ┐
;; →        1.00
;; →       -1.00
;; └                ┘
```

5 行目までの内容は依存関係の解決である。 Sample1 において単純な行列の足し算を

行っており、 Sample2 は QR 分解を用いて $Ax = b$ の解を求めている。そして Sample2 ~ another solution ~ はこれを存在しているライブラリ関数を用いて解いたものである。両者の速度差はこのサイズの行列演算であればほぼないが、大規模サイズの行列であった場合は後者のほうが圧倒的に速い。後者も前者もほぼ直接 Fortran のライブラリである LAPACK [11] を触っているため、計算途中で結果を取り出している前者のほうが効率が悪いためである。

このコードからわかるように、このライブラリが返す値は必ずしも求めている・求まった解答の形を示していない。この理由は Intel MKL 内のソースコードが与えられたデータのメモリに解答を書き込む性質があるためである。この破壊的代入を行う性質は高速化に大きな貢献をしているとともに、高い副作用と難解さを招いている原因であると考えられるが、このライブラリを利用するためには Intel MKL のドキュメントを精読することや、内部の Fortran による実装を眺める他にない。

4.4 Clojure.core.matrix

先に紹介した 2 つに対してこちらは非常におとなしいライブラリであり、 Clojure の標準的な算術関数のラップや行列演算に関するライブラリの基盤を開発している。ライブラリの基盤というのは、 Java などのオブジェクト指向言語におけるインターフェースのようなもので、実装するべき関数を先に示しておくことで、それを様々な手法によって実装・更新されていくことで長期的にそのライブラリ群を使うことが出来るという利点がある。本演習では、高速さが持ち味である vectorz-clj や jblas を用いて実装されている幅広い関数を持つ clatrix の 2 つを検討しその両方を利用した。

4.4.1 vectorz-clj

Vectorz-clj は純粋に JVM で動作する高速な行列計算ライブラリを掲げており、導入にかかるコストの低さが魅力的である。

[11] Linear Algebra PACKage

4.4.2 clatrix

4.5 ACM3 (Apache Common Math 3)

5 ARIMA モデルによる時系列分析

5.1 ARIMA モデルの概要

5.2 BackShift 記法

5.3 単位根検定

5.3.1 ADF 検定

5.3.2 KPSS 検定

5.4 AR モデル

5.5 MA モデル

5.6 係数推定

5.6.1 対数尤度

5.6.2 AIC

5.6.3 最小二乗法

5.6.4 アメーバ法

5.7 SARIMA モデルについて

6 Clojure/ClojureScript を用いた Web 開発

6.1 Clojure によるバックエンド開発

6.1.1 Luminus Framework

6.1.2 Swagger UI

6.2 ClojureScript によるフロントエンド開発

6.2.1 基本的な開発

6.2.2 Reagent

6.2.3 core.async による非同期処理

7 MKKL の開発

8 発展：ARIMA 推定 と Random Forest による予測

8.1 概要

8.2 実験方法

8.3 実験結果

8.4 考察

9 まとめと今後の課題

ゲームエンジン・グラフィクエンジンの開発（GPU）、ゲーム画面をデータベースに保存した上で、解析を行う。（ゲームの進行ログではなく、一般的にユーザが見ることになる

ゲーム画面の遷移から強化学習を行い、ゲーム AI を作成する。）データベースと GPU 計算技術は学ぶことができた。時系列解析について入門することができた。ゲーム AI の入門については昨年度 Common Lisp を用いて学習済みである。

References

[1] Jason Dixon. *Monitoring with Graphite. Tracking Dynamic Host and Application Metrics at Scale*. Vol. 290. O'Reilly Media, 2017.

[2] Daniel Higginbotham. *Clojure for the Brave and True. Learn the Ultimate Language and Become a Better Programmer*. No Starch Press.

[3] Shantau Kumar. *Clojure High Performance Programing*. Packt Publishing.

[4] Mattew Scarpino. *OpenCL in Action. How to Accelerate Graphics and Computaion*. Manning Pubns Co, Nov. 2011.

[5] *The Apache HBase^{TM} Reference Guide*. Revision 0.94.27. Copyright © 2012 Apache Software Foundation. https://www.slideshare.net/hamadakoichi/randomforest-web, Dec. 2015.