Vol.2012-OS-120 No.7
2012/2/29

分散システム開発向け プログラミング言語 Bloom の評価

藤枝崇史[†1]　新井淳也[†2]　大村圭[†1]　藤田智成[†1]

近年，多数のコンピュータを組み合わせることで性能のスケールアウトや冗長化を実現する分散システムの需要が高まっている．分散システムを開発する際には一貫性の保証が焦点となる．強い一貫性は可用性を犠牲にするため，応答速度が重視される Web サービス等のシステムとの相性が悪い．この問題を解決する一貫性に，結果整合性というモデルが存在する．しかし，結果整合性を利用する分散システムの開発は困難である．本論文で紹介する Bloom は，分散システム開発をターゲットとしたプログラミング言語である．Bloom には，結果整合性を利用する処理を簡便に記述できる，一貫性を厳密に保証すべき処理と結果整合性を利用しても良い処理の判別を自動的に行うことが可能である，などの特徴がある．本論文では，Bloom の処理系である Bud の内部実装の解析と，実行速度の評価を行った．評価の結果，Bloom のプログラム内で扱うデータ量が増えるほどにアクセス速度が増加するため，1000 個のデータを扱う場合，Bud のプログラムの処理速度は Ruby の 100 倍前後遅いという結果が得られた．

Evaluation of Bloom Programming Language Designed for Distributed Systems Development

Takafumi Fujieda[†1]　Junya Arai[†2]　Kei Omura[†1]
Tomonori Fujita[†1]

Recently, we have seen a growing demand for distributed systems to achieve scale-out performance and availability by many computers. Tha main focus is ensuring consistency in distributed systems. Strong consistency sacrifice availability. Therefore strong consistency is unfit for the systems that require a high-speed response such as web services. Eventual consistency solves the problem. However development of distributed systems are difficult by using eventual consistency. Bloom is a programming language for distributed systems development. Bloom simplifies a distributed program to use in eventual consistency, and it is possible to automatically analysis whether to use strong consistency or eventual consistency. In this paper, we analyzed Bud, Bloom interpreter implementation, and evaluated of Bud processing speed. As a experimental result, processing speed to access objects increased linearly with increasing amount of objects handled by Bloom program. For example, If Bud program handle 1,000 objects, processing speed of Bud program is 100 times as slow as that of Ruby program.

1. 序論

近年，多数のコンピュータによって構成される分散システムが多くの場面で利用されている．より大容量なデータベースやファイルシステムを扱いたい，巨大なデータを高速に処理したい，といったニーズが存在しており，単体のコンピュータをスケールアップしてこれらのニーズに答えることは難しく，可能であってもコストがかかる状況であった．しかし，ネットワークの高速化と大容量化，汎用コンピュータの高性能化と低価格化によって，安価なコンピュータを多量に用いた分散システムによって前述のニーズを満たすことが可能となった．分散システムには，スケールアウトによる処理速度の高速化や記録領域の増大，複数のコンピュータを利用することによる耐障害性の向上などのメリットが存在する．有名な分散システムとしては，大規模なデータセットを分散並列処理によって高速に処理する MapReduce[7]フレームワークや，大容量で冗長性の高いストレージである分散 Key-Value-Store（KVS）や分散ファイルシステムなどが挙げられる．

分散システムを開発する際には，一貫性（Consistency）の保証をどうするかが焦点となる．分散システムにおいて厳密な一貫性を保証しようとした場合，可用性が損なわれる．例えば，あるクライアントが分散システムに対してデータの更新処理を行ったとする．強い一貫性を保証する場合には，そのデータを読み取ろうとする全てのクライアントに対して更新後のデータが返されることを保証しなければならないため，分散システム内でデータの同期が完了するまでの待ち時間が発生する．厳密な一貫性を保証する分散システムには，分散リレーショナルデータベースや，分散ファイルシステムなどが存在する．

分散システム上で可用性を向上させるための一貫性として，結果整合性（Eventual Consistency）が存在する．結果整合性は，データが更新されてから複製が完了するために十分な時間が経過していればいずれは新しい値にアクセスできるだろう，という楽観的な考えに立脚した一貫性のモデルである．結果整合性を利用することで，一貫性の保証は弱まる代わりに，可用性のスケーラビリティが保証される．例えば，結果整合性を採用した分散システムに対してデータの更新を行った場合，分散システム内でデータの同期が完了するまで待つことは無い．データを同期している最中は，更新前の古い値を返すノードと更新後の新しい値を返すノードが混在する状態となる．しばらく時間が経てば同期が完了しているため，全てのノードが更新後の値を返すようになる．結果整合性を採用した分散システムは，可用性が重視される Web サービスに利

[†1] 日本電信電話株式会社サイバースペース研究所
NTT Cyber Space Laboratories
[†2] 東京大学　大学院　情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo

用されることが多い．例えば，Apache Cassandra[8]などの分散 KVS は，結果整合性を採用した分散システムである．

しかし，従来のプログラミング言語では結果整合性を利用する分散システムの開発は困難である．プログラム内で結果整合性を利用しても問題が無い箇所と，厳密な一貫性を保証すべき箇所との判別が難しいためである．この問題を解決するプログラミング言語として，Bloom[1]が存在する．Bloom は分散システムを開発するためのプログラミング言語として設計され，結果整合性を簡便に記述することが可能な言語仕様となっている．また, Bloom を用いて記述したプログラムに対して，一貫性を厳密に保証すべき箇所と，一貫性を弱めて結果整合性を利用しても良い箇所の判別を自動的に行う仕組みが存在する．

本論文では, Bloom に関する知見を述べ，評価を行う. 2 章では Bloom の概要, Bloom の理論的背景である CALM 原理, Bloom の言語仕様について述べる. 3 章では Bloom の処理系の概要について述べ，処理系の内部実装上の問題点を挙げる. 4 章では Bloom を用いたプログラムを実装して性能評価を行った結果と，結果に対する考察を述べる．5 章では関連研究について述べ, 6 章ではまとめを述べる．

2. Bloom について

本章では, Bloom の概要について述べる. 2.1 節では Bloom と結果整合性との関係性について, 2.2 節では Bloom の理論的背景である CALM 原理について, 2.3 節では Bloom の言語仕様についてそれぞれ述べる．

2.1 Bloom の概要

分散プログラミングにおける主要な課題の一つとして一貫性（Consistency）の保証がある．分散環境上のノード間通信ではメッセージやデータの配送が遅延したり順序が入れ替わったりする可能性があり，この時間的な非決定性が一貫性の保証を困難にしている．分散ロックシステムなどにより厳密な一貫性を保証することは可能だが，コストが大きくパフォーマンスの低下を招く．

そこで分散プログラミングでは結果整合性（Eventual Consistency）[2] を保証するのが一般的である．結果整合性とは厳密な整合性より緩い条件の一種で，ノード間で全てのメッセージの配信が完了した後の状態における一貫性を指す．厳密な一貫性に比べ結果整合性の保証は小さいコストで済むためパフォーマンスの向上が見込まれる．しかし厳密な一貫性の下で動作するプログラムを結果整合性のあるプログラムへ変更する際には，プログラム中の一貫性に対する要求を緩めてよい個所と緩めてはならない箇所を判別する必要がある．時間的に非決定的な環境でもメッセージの到着順序にプログラムの動作が依存しなければ結果整合性があると言えるため，順序に依存する箇所と順序に依存しない箇所を区別し適切に処理する必要がある．ところが現在主流である手続き型プログラミング言語ではプログラムが順序に依存しないことを検証するのが難しい．なぜなら手続き型言語は計算モデルとしてチューリングマシンに基づいており，生来順序性のある環境を前提として設計されているからである．

これに対し集合ベースの論理型プログラミング言語である Bloom ではプログラムから順序性がほぼ排除されており，順序に非依存で結果整合性のあるプログラムを自然に記述することが可能となっている. 一般的な手続き型言語と Bloom を比較するとこのことが分かりやすい．まず前者ではプログラムの状態（データ）を順序のある配列に格納するのに対し，後者では順序のない集合を使用する．さらに前者では論理を逐次実行される手続きの列で表現するのに対し，後者では順序のない宣言的なルールの集合で表現する．しかしながらあらゆる処理を順序性抜きで記述できるわけではない．例えば集合の要素の数は要素全てを受信し終わらないと求めることができないため，次の処理との間に順序が発生する．このような処理に対して Bloom は CALM（Consistency And Logical Monotoniciity）原理 [1] に基づく分析を行うことで結果整合性を保証する．

2.2 CALM 原理

CALM 原理とは集合上の操作が持つ単調性（Monotonicity）と一貫性の間の密接な関連について述べた原理である．単調な操作では操作対象である集合に新しく要素が追加されたとき，一度行われた出力または成立した命題が覆ることはない．例として整数の集合から偶数だけを取り出す操作を考えてみる. 最初操作対象の集合が {0, 1, 2, 3} であったなら結果は {0, 2} である. ここで操作対象の集合に要素が追加され {0, 1, 2, 3, 4} になったとき，結果は {0, 2, 4} となる．もともと存在した 0 と 2 は今後どんな数が操作対象に追加されようとも結果に含まれ続ける. このような選択（Selection），あるいは射影（Projection），結合（Join）などは全て単調な操作である．一方非単調な操作は操作対象である集合に新しく要素が追加されたとき出力結果は覆る可能性がある．例えば集合の要素数を求める操作は操作対象に要素が追加されると結果も変わってしまう．一般に集約（Aggregation），否定（Negation）などの操作は非単調である．以上を踏まえた上で，CALM 原理が述べるのは次の 2 つのことである．単調な操作から成るプログラムはメッセージの配送や計算の順番が入れ替わっても結果整合性は保証される．一方非単調な操作で結果整合性を保証するためには操作対象である集合の要素が全て揃うまで待機する処理を挿入しなければならない．

そこで [1] において述べられているように，Bloom はプログラムの解析によって全ての操作を単調と非単調に分類する．集約や否定を行うシンボル（演算子）が登場しなければそれは単調な操作であるから，単調な操作の発見は単純な文法的解析で足り

る場合もある．しかし集約や否定が含まれながら単調であるような操作もある．例えば"MIN(x) < 100"（集合 x に含まれる値の最小値が 100 以下である）の場合"MIN"と"<"という集約操作を含み一見非単調であるが，集合 S について"MIN(S) < 100"が成立するならば S に任意の要素を追加した集合 S' についてもこの命題は成立し，結果が覆ることはない．ゆえにこの操作は単調である．この例のように単調性の検証に深い意味論的な解釈が要求される処理は他にも存在するため，浅い解析で単調性を保証できない場合は保守的な対応を行う．すなわち単調でないかもしれない操作には操作対象集合の全ての入力データが到着するまで待機する処理を挿入することによって結果整合性を保証する．

2.3 Bloom の言語仕様

Bloom では集合をデータ構造として扱い，集合同士のマージ式を宣言的に記述する．Bloom の処理系は，一定時間ごとに Timestep 更新と呼称される更新処理を実行する．この更新処理では，scratch 型の集合を空にする処理，前回の遅延マージ処理を反映させる処理，全てのマージ式を評価する処理などが行われる．

Bloom の集合は，table 型と scratch 型の 2 種類が存在する．table 型は永続的に値を保持し続ける集合である．scratch 型は Timestep 更新の度に保持する要素が破棄され，空集合として再初期化される集合である．scratch 型の派生として，ネットワーク越しに要素をやり取りする際に利用する channel 型が存在する．

マージ処理には，式が評価された際に即座にマージが行われる即時マージ処理と，評価された時点ではなく次回以降の TimeStep 更新時にマージが行われる遅延マージ処理の 2 種類が存在する．遅延マージには，次の Timestep 更新時に必ずマージが行われるものと，いつか必ずマージが行われるものの 2 種類が存在する．後者のマージは，channel 型に対するネットワーク越しのマージに利用される．

マージ式は，「集合 A マージ演算子 集合 B」の形式で記述する．図 1 にマージ式の具体例を示す．このマージ式は即時マージ演算子 <= を利用しているので，式が評価された瞬間に，集合 tbl に ["Hoge"], ["Foo"] の 2 要素がマージされる．tbl が ["Piyo"] の 1 要素を保持していた場合，このマージによって tbl は ["Piyo"], ["Hoge"], ["Foo"] の 3 要素を持つ集合となる．なお，集合なので順序は保証されていない．

これ以外に，Join 型という特殊な集合型が存在する．Join 型は，集合同士のペアを要素として持つ特殊な集合である．「集合 A * 集合 B」の形式で記述することで，集合 A と集合 B の全要素の組み合わせを要素として持つ Join 型の集合が作成される．図 2 に Join 型の具体例を示す．

```
tbl <= [ ["Hoge"], ["Foo"] ]
```

図 1. Bloom のマージ式のサンプルプログラム

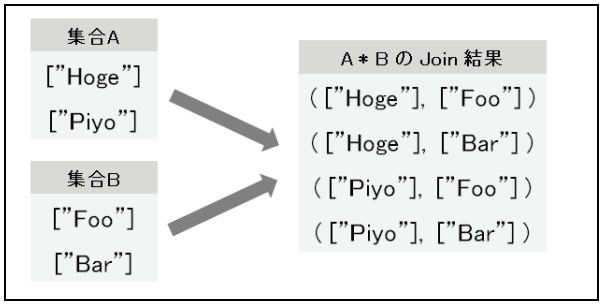

図 2. Join 型の具体例

Bloom のマージ式では，集合に対してコールバック呼び出しを行い，各要素をコールバック関数によって処理することが可能である．次節で述べる Bloom の処理系 Bud では，Ruby のブロックをコールバック呼び出しすることにより，集合の各要素に対して手続き型のプログラミングを可能としている．

3. Bloom の処理系の概要

本章では，Bloom の処理系について述べる．Bloom の処理系には，現在プロトタイプとして開発中の Bloom under development（Bud）と呼称される処理系が存在する．この処理系は Ruby の内部 DSL として実装されている．3.1 節では，Bud を利用するプログラムを記述する際の構造について述べる．3.2 節では，前章で述べた Bloom の Timestep 更新が，Bud の内部ではどのように行われているのかを述べる．3.3 節では，Bud プログラムの実行時の動作について述べる．3.4 節では，Bud の処理系の内部実装について，実行時のボトルネックとなりうる問題点を 3 点示す．

3.1 Bud のプログラムの構造

Bud を利用するプログラムは，集合を定義する箇所，集合の初期化を行う箇所，集合に対する演算を行う箇所，の 3 つのブロックで構成される．以降では，図 3 のソースコードを題材に，各ブロックの概要について解説する．

集合を定義する箇所は，state ブロックと呼称される．state do 〜 end の形で記述した箇所が state ブロックとなる．do 〜 end 内部に，集合の形式を定義する構文を記述していく．図 3 のソースコードの state ブロック内の 1 行目は，table 型の集合 tbl を定義

```
module SampleModule
  state do
    table :tbl, [:key, :value]
    interface input, :get, [:key]
    interface output, :result, [:value]
  end

  bootstrap do
    tbl <= [ ["Hoge", "Piyo"], ["Foo", "Bar"] ]
  end

  bloom do
    result <= (get * tbl).pairs(:key => :key) { |k, v| [v.value] }
    stdio <~ result
  end
end
```

図 3. Bud のモジュールを定義するサンプルプログラム

する構文である. Bud では, 集合の各要素は, キーと値の組み合わせから成るハッシュテーブル的なものとして実装されている. 集合 tbl は, キーが:key であるメンバと, キーが:value であるメンバの, 合計 2 つのメンバを持つハッシュテーブルを要素として持つ集合として定義される. 同様に, state ブロックの 2 行目と 3 行目では, メンバを 1 つ持つハッシュテーブルが要素となる interface 型の集合がそれぞれ定義される. interface 型は, Bud インスタンスの外部からアクセスしても良いことを明示的に示すための型であり, 内部実装的には scratch 型と同等の型である. interface input 型は外部から値を入力する Setter としての用途を, interface output 型は外部から値を読み取る Getter としての用途を想定したものである.

集合を初期化する箇所は, bootstrap ブロックと呼称される. bootstrap do ～ end の形で記述した箇所が bootstrap ブロックとなる. bootstrap ブロック内には, 初期化のために Bloom の式を記述することが可能である. bootstrap ブロック内に記述した式は, Bud インスタンスが初めて実行された時（Timestep 更新が初めて行われた時）に初期化処理のために一度だけ評価される. 図 3 のソースコードでは, tbl を 2 つの要素を持つ状態で初期化している. 要素はハッシュテーブルなので, [:key="Hoge", :value="Piyo"] と [:key="Foo", :value="Bar"] の 2 つの要素を持つ状態となる.

集合に対する演算を行う箇所は, bloom ブロックと呼称される. bloom do ～ end の形で記述した箇所が bloom ブロックとなる. bloom ブロック内には Bloom の式を記述することが可能である. 記述した式は Bud インスタンスの Timestep 更新の度に評価される. 初回の Timestep 更新の場合には, bootstrap ブロックを評価してから Bloom ブロックを評価する順序となる. 図 3 のソースコードでは 2 つの Bloom 式を記述している. bloom ブロックは複数定義することが可能であり, ブロック毎に固有名前を付けることができる. ただし, この名前に関しては, 現時点での実装では名前を付けることによってプログラムの可読性を向上させる効果しかなく, この名前を利用して何らかの処理を行えるような実装とはなっていない.

3.2 Timestep 更新の概要

本節では, Bud が Timestep 更新をどのように行なっているかを述べる. Timestep 更新は, まず最初に全ての集合の更新処理を行う. table 型の場合, 集合に対して遅延マージの反映を行う. scratch 型の場合, 集合の中身を空にしてから遅延マージの反映を行う. channel 型の場合, 集合の中身を空にする処理のみを行う. 全集合の更新が終わったら, Timestep 更新が初回呼び出しである場合にのみ, bootstrap ブロック内の式を評価する. この後で, channel 型に対する遅延マージの反映を行う. 続いて, bloom ブロック内の全ての Bloom 式の評価を行う. 即時マージ演算時を用いた式であれば, この時にマージ処理が行われる. 遅延マージ演算子を用いた式であれば, 遅延マージ用のデータが専用の領域に一時プールされ, 次回の Timestep 更新時に遅延マージが行われる. ただし, ネットワーク（channel 型）や標準入出力（stdio）に対する遅延マージであれば, bloom ブロック内の全ての式が評価し終わった直後に, 対象への書き込みが行われる. Timestep 更新によって行われる処理は以上である.

Timestep 更新は, tick, sync_do, async_do の 3 つのメソッドを呼び出すことにより明示的な実行が可能である. sync_do, async_do には, ブロック引数として複数の Bloom 式を指定することができ, 引数の Bloom 式を実行した後で Timestep 更新が実行される. 前述の通り, Timestep 更新の最初に scratch 型の集合の中身が空になるため, scratch 型の集合に対する即時マージ処理の Bloom 式を引数に指定した場合, マージした後に scratch 型の集合が空にされてしまう. そのため, sync_do, async_do の引数に scratch 型の集合に対するマージ式を指定する場合には, 遅延マージ演算子を利用することとなる. なお, 引数の Bloom 式は, sync_do, async_do の実行時に 1 度だけ評価されるもので, Bud 用モジュールの bloom ブロック内に記述された Bloom 式のように毎回の Timestep 更新の度に評価されるものではない.

また, run_fg, run_bg メソッドを呼び出すことで Bud がプロセスとして実行状態となる. この状態の Bud プロセスは, データを受信する度に Timestep 更新を 1 回実行する. この Timestep 更新時に, 受信したデータが channel 型の集合へとマージされる. なお, Bud の実装ではネットワーク処理に EventMachine ライブラリを利用している. EventMachine がデータを受信した際, つまり receive_data メソッドが呼び出される度

に, Timestep 更新が 1 回実行される.

3.3 Bud プログラムの実行について

Bud 上に定義した Bloom のプログラムは, 定義したモジュールと Bud モジュールを include したクラスのインスタンスを作成することで利用する. 図 4 のソースコードでは, Bud と SampleModule を include した SampleClass クラスを定義し, そのクラスのインスタンスとして bud_ins を生成している. このようにして生成された Bud インスタンスに対しては, 前節で述べた通り, tick, sync_do, async_do の 3 つのメソッドを呼び出すと Timestep 更新が 1 回行われる.run_fg, run_bg メソッドを呼び出すと Bud インスタンスが実行状態となり, データを受信する度に Timestep 更新が 1 回行われる.

以降では, 図 5 で示したサンプルプログラムの動作を追いかける形で, Bud プログラムの挙動について説明していく. サンプルプログラムでは, 引数に 1 つの Bloom 式を指定して sync_do メソッドを実行している. sync_do メソッドが実行されると, まず引数の Bloom 式が評価される. 遅延マージ演算子 <+ によって, get に対して [["Foo"]] が遅延マージ設定される. その後で Timestep 更新が実行される. Timestep 更新では, まず get, result の中身が空にクリアされた後で, 遅延マージによって get の中身が [["Foo"]] となる. 次に, 初回の Timestep 更新なので bootstrap ブロック内の式が評価され, 集合 tbl の中身が [["Hoge", "Piyo"], ["Foo", "Bar"]] となる. その後で, Bud インスタンス内の Bloom ブロック内の式が評価される. 1 行目の式では, * 演算子によって join 型の集合が生成される. Join 型は 2 つの集合同士を組み合わせた集合なので, 今回生成される Join 型は, (["Foo"], ["Hoge", "Piyo"]), (["Foo"], ["Foo", "Bar"]) の 2 つの要素を持つ集合となる. この集合の pairs メソッドを, (:key => :key) を引数として記述することで, 左辺（get）の:key に対応する値と, 右辺（tbl）の:key に対応する値同士が合致するものだけが抽出される. 抽出された結果の Join 型は, (["Foo"], ["Foo", "Bar"]) の 1 つの要素を持つ集合となる. その後, ブロック引数のコールバック呼び出しによって集合の各要素が評価され, このブロックが返した要素の集合が result へと即時マージされる. このブロック内には, Ruby によって手続き型のプログラムが記述可能である. ブロックの第 1 引数には要素の左側が, 第 2 引数がそれぞれ渡される. 今回は Join 型の要素は 1 つなので, 第 1 引数に ["Foo"] が, 第 2 引数に ["Foo", "Bar"] が渡される. 戻り値として, 第 2 引数の value に対応する値, つまり"Bar"を返している. 結果, [["Bar"]] が result へと即時マージされ, result の中身は [["Bar"]] となる. 2 行目は I/O 用の遅延マージ式で, result の中身を標準出力（stdio）に遅延マージしている. マージが完了すると, 画面に result の中身が出力される. 今回のプログラムは Bar が出力される. 以上が, Bud を用いてプログラミングを行う際の概略である. 集合の各要素に対して Ruby による手続き型のプログラミングが可能であり, Ruby の豊富なライブラリが利用できる点が Bud の利点であると言える.

```
class SampleClass
  include Bud
  include SampleModule
end

bud_ins = SampleClass.new()

bud_ins.sync_do do
  bud_ins.get <+ [ ["Foo"] ]
end
```

図 4. Bud のモジュールを利用するサンプルプログラム

3.4 Bud の内部実装の問題点

Bud の内部実装には, いくつかの問題点が散見される. 本節では, これらのうち, 特に大きな問題だと考えられる 3 つについて述べる. それぞれ, Join 型に関する話題, Bloom 式の評価に関する話題, ネットワーク処理に関する話題である.

Join 型を作成する際には, * 演算子の左辺の集合の持つ要素と左辺の集合の持つ要素の全組み合わせから成る Join 型のインスタンスが必ず作成される. pairs メソッドや lefts メソッドはインスタンスの作成後に呼び出されるため, あらかじめ全組みあわせを要素とする集合を作成してから, その全要素に対して総当りで検索処理を行い, 引数の条件に合致する集合をさらに作成する手順となる. pairs メソッドや lefts メソッドによって絞り込みが行われることが予めわかっているのだから, 作成時に絞込み処理を行い, 集合の作成は 1 度で済ませるべきである. また, 要素をハッシュテーブルのように保持しているはずが, 全要素を線形探索している点も問題である. これだと要素数の増加に伴い探索時間が O(n)となってしまう. ハッシュテーブルとして扱うのであれば O(1)となるように実装すべきである.

Timestep 更新によって行われる Bloom 式の評価は, bloom ブロック内に記述されている全ての Bloom 式に対して行われる. 評価の際に, マージ演算子の右辺に記述されている集合が空集合であるかどうかは考慮されていない. そのため, Bloom 式の量が増えるほど, Timestep 更新時に式評価に費やされる時間は増大する. この点は, Join 型を作成する際に特に問題となる. 例えば, *演算子の左辺が空集合で右辺が要素数 n 個の集合であった場合, 組み合わせ数が 0 個の Join 型となるにもかかわらず, n 回のイテレーションが行われてしまう. 空集合であることが自明である式は評価する必要は無いだろう.

Eventmachine がネットワーク越しにデータを受け取った際に呼び出される receive_data メソッドは, 呼び出されるたびに Timestep 更新を実行している. 前述の 2

点が特にボトルネックとなり Timestep 更新に時間がかかることが予測されるため, Bloom プログラムの内容次第で, qps が低い値となってしまうことが懸念される.

4. 評価と考察

4.1 実験の概要

実験に用いるプログラムは, memcached プロトコルを受信して解析し, Key-Value-Store（KVS）に対して要素を追加（set）する処理と, KVS に対して要素を取得（get）する処理を行う, Ruby で実装したサーバープログラムである. KVS 以外の部分は全く同一の実装とし, KVS の部分だけを, 一方は Bud で, もう一方は Ruby で実装した. このような形とした理由は, Bud が Ruby の内部 DSL として実装されているためである. Ruby 版の KVS は, Ruby の Hash クラスのインスタンスがキーと値のペアを保持する形で実装した. Bud 版の KVS は, 公式 Web サイト 3)にて Bud のサンプルプログラムとして公開されている KVS を元に多少の改変を加える形で実装した. この KVS は 1)にも記載されているプログラムである.

このプログラムを用い, 3 つの実験を行った. 1 つ目の実験は, KVS に対して要素を追加していき, 一定の量を追加し終わるまでに費やした所要時間を比較するものである. 2 つ目の実験は, 一定の量を保持する KVS から, 要素を 100 個取得し終わるまでに費やした時間を比較するものである. 3 つ目の実験は, Bud 版の KVS に対し, 空集合と KVS の要素とを Join する式を複数記述し, その式が増える毎に所要時間がどの程度増加するかを計測するものである. 1 つ目, 2 つ目の実験は, 3-4 節で挙げた問題点の 1 点目, Join 型に関する問題点が, 処理速度に対して与える影響について検証するための実験である. 3 つ目の実験は, 問題点の 3 点目, 式評価に関する問題点が, 処理速度に対して与える影響について検証するための実験である.

なお, サーバーに対する要素の追加や要素の取得には, Ruby の MemCache ライブラリを利用した. KVS に対して用いるデータは、UIDTools ライブラリの生成するユニーク ID 文字列をキーとし, 1KB のランダムな文字列データを値とした. 実験環境には, CPU が AMD Athlon BE-2350 1GHz, メモリが 4GB の計算機を用いた.

4.2 実験結果

実験結果について述べる. 図 5 のグラフは, 要素を持たない空の状態の KVS に対して, 要素を追加（set）した際の所要時間を示すグラフである. 横軸は追加した要素数, 縦軸は処理が完了するまでの所要時間となっている. 要素を追加する処理は, Ruby 版, Bud 版のどちらの場合も, 追加しようとしている要素のキー値と同じものがハッシュテーブル内に既に存在するかどうかを調べ, 存在すればその要素を削除して, それか

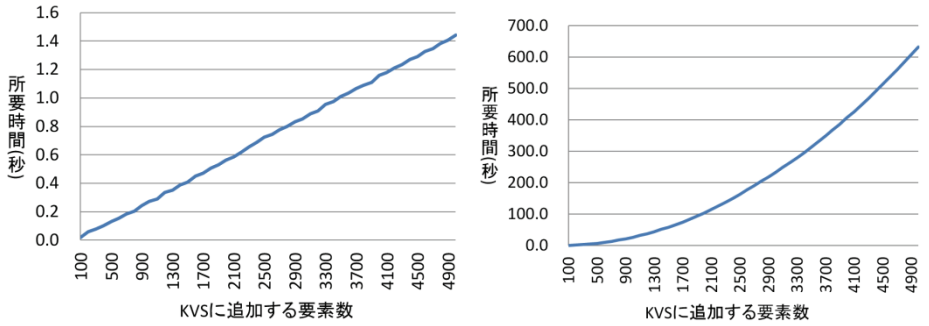

図 5. KVS に要素を追加する所要時間. 左図が Ruby 版, 右図が Bud 版.

ら要素を追加する手順をとっている. Ruby 版は Hash クラスを利用して実装しているため, キー値に対応する要素を探索する際の所要時間は O(1)である. そのため, 追加する要素が増えても一定の処理速度を保ち続けている.要素を 1 つ追加する際に費やす時間は, 平均して 0.0003 秒であった. 一方 Bud 版では, キー値に対応する要素を探索する際に Join 型を利用している. 3-4 節でも述べた通り, Join 型はハッシュテーブルのような O(1)ではなく, 一旦全ての要素にアクセスした上で, 探索処理を行なう実装となっている. つまり, 要素数 n の KVS からキー値に対応する要素を探索する際の所要時間は O(n)となり, 所要時間は KVS の要素が増えるほど階差数列的に増大する. 要素を 100 個追加する際の所要時間は 0.54 秒, 1000 個要素を追加する際の所要時間は 27 秒, 2000 個要素を追加する際の所要時間は 104 秒であった. なお, 100 個要素を追加する際の所要時間は Ruby 版の 18 倍遅く, 1000 個要素を追加する際の所要時間は Ruby 版の 90 倍遅く, 2000 個要素を追加する際の所要時間は Ruby 版の 173 倍遅い.

図 6 のグラフは, KVS から値を 100 個取得（get）した際の所要時間を示すグラフである. 横軸は KVS の保持する要素数, 縦軸は処理が完了するまでの所要時間となっている. 要素を取得する処理は, Ruby 版, Bud 版のどちらの場合も, キー値に対応する要素がハッシュテーブル内に既に存在するかどうかを調べ, 要素が存在すればその値を返す形をとっている. 前述の通り, Ruby 版はキー値に対応する要素の探索時間が O(1)なので, KVS の保持する要素数の増減の影響をあまり受けない. 結果として, KVS の要素数に関係なく, KVS から要素を取得する際の所要時間は一定の時間となる. 所要時間は, 平均して 0.03 秒であった. Bud 版はキー値に対応する要素の探索に Join 型を用いているため, キー値に対応する要素の探索時間が O(n)である. そのため, KVS から要素を取得する際の所要時間は, KVS の要素数が増加するほどに線形増加する. KVS の保持する要素が 100 個である場合の所要時間は 0.84 秒, 1000 個である場合の所要時間は 4.18 秒, 2000 個である場合の所要時間は 8.28 秒であった. なお, 100 個である場合

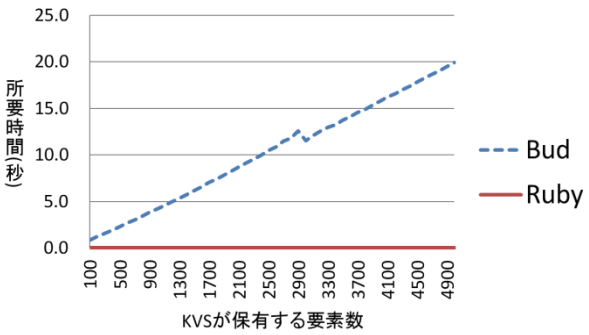

図 6. KVS から要素を取得する所要時間

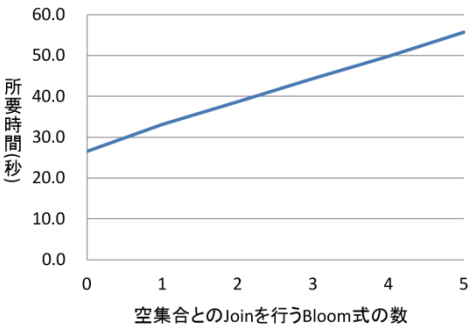

図 7. Join を行う Bloom 式を追加した際の所要時間

の所要時間は Ruby 版の 28 倍遅く，1000 個要素を取得する場合の所要時間は Ruby 版の 139 倍遅く, 2000 個要素を追加する場合の所要時間は Ruby 版の 276 倍遅い．

続いて，Timestep 更新の度に全ての Bloom 式を評価している点が，処理速度に対するボトルネックとなりうるのかを実験によって検証する．Bud 版の KVS に対して，KVS の保持する要素と空集合との Join を行う Bloom 式をいくつか追加し，式の増加による処理速度の変化を計測する．この Join の演算結果は空集合となるため，この Bloom 式は全く無意味な式である．図 7 のグラフは，この Bloom 式を複数持つ Bud 版の KVS に対して要素を 1000 個追加（set）した際の所要時間を示すグラフである．横軸は空集合との Join を行う Bloom 式の数，縦軸は処理が完了するまでの所要時間となっている．空集合と KVS の保持する要素との Join は O(n)の時間を消費するため，式が増えるほどに処理が完了するまでの所要時間が増加する．なお，この Bloom 式が 1 つ増える毎に，約 5.6 秒所要時間が増加している．

4.3 考察

実験結果によって, 3.4 節において述べた, Join 型に関する問題点と, Bloom 式の評価に関する問題点が，プログラムの実行時に多大なボトルネックとなることが示された．この 2 つの問題点は，処理系の内部実装を書き換えることで改善が可能であると考える. 3.3 節において述べた通り, Bud のプログラムでは，外部から渡された集合と内部に保持する集合との Join 型を作成し，キー値によって絞り込み，その結果を別の集合にマージする，という記述が頻繁に行われる．キー値による絞り込みを行うのであれば，現在のような集合の全要素に総当たりを行うような実装ではなく，ハッシュテーブルのような実装をして，キー値による絞り込みをしてから Join 型を作成する形での実装を行うことで高速化が見込める．加えて，マージ演算子の右辺が空集合となる Bloom 式の評価を行わないように式評価のアルゴリズムを改善することで，不必要な式評価を省くことによる高速化が見込める．ただし，マージ演算子の右辺が空集合となる Bloom 式がボトルネックとなるのは空集合との Join 処理を行う場合であるため，Join 処理のアルゴリズムを改善すれば，式評価による処理速度への影響は小さく抑えられるため，式評価の内部実装は現状のままで問題無いとも考えられる．

5. 関連研究

Bloom よりも以前に開発されたプログラミング言語として，Overlog[4]が存在する．Overlog は，SQL に類似した命令を持つ論理プログラミング言語である．Overlog では，外部から受け取ったイベントを元に Overlog のプログラム内で処理を行い，その結果を外部へと出力する. Overlog のプログラムは既存の手続き型言語とは異なるロジックであるため，手続き型言語と連携したプログラムを記述することが困難であった．Bloom ではこの点を改善するため，処理系を Ruby の内部 DSL として開発した．これにより，データに対する操作を Ruby によって手続き的に記述することが可能となった．これには，Ruby のライブラリ等の既存のプログラム資産を活用できるというメリットもある．

なお, Bloom を開発する以前に, Dadalus[5]というプログラミング言語が試作されている. Dadalus は Bloom のプロトタイプとして作成された言語で, Overlog と比較してシンプルな構文を採用している．Bloom の採用しているマージ演算子や Join 演算子を用いるシンプルな構文は, Dadalus での経験を元に設計されたものである．

分散システム開発用途のプログラミング言語として，Erlang[6]の存在が挙げられる．

Erlang は関数型言語である．言語自体が非同期なメッセージパッシングの仕組みを持っており，並行動作するアクター（オブジェクト）間でメッセージをやり取りすることで分散処理を実現する．Erlang は順序のあるリストを基本的なデータ構造として扱うため，一貫性を厳密にすべき箇所と一貫性を緩めても良い箇所の分析が，Bloom と比較して困難である．

6. まとめ

本論文では，プログラミング言語 Bloom の概要と，その処理系である Bud について述べ，Bud の内部実装にいくつかの問題点があることを示した．実際に Bud 上で動作する Bloom プログラムを作成して評価を行った結果，この問題点は Bud 上でプログラムを実行する際に多大なボトルネックとなっていることが確認された．Bloom の言語仕様自体の欠陥ではなく，Bud の内部実装を書き換えることで改善が見込めるボトルネックであるため，問題点を改善するアルゴリズムを処理系に適用させることが今後の課題である．

参考文献

1) P. Alvaro, N. Conway, J. Hellerstein, and W. Marczak. Consistency analysis in Bloom: a CALM and collected approach. In Biennial Conf. on Innovative Data Systems Research (CIDR), CA, USA, January 2011.
2) W. Vogels. Eventually Consistent. Commun. ACM, 52(1), pp.40-44, 2009.
3) BOOM -- Berkeley Orders of Magnitude -- Declarative Languages And Systems, http://boom.cs.berkeley.edu/
4) B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In ACM Symposium on Operating Systems Principles (SOSP), 2005.
5) P. Alvaro, W. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. C. Sears. Dedalus: Datalog in time and space. Technical Report UCB/EECS-2009-173, EECS Department, University of California, Berkeley, Dec2009.
6) Ericsson Computer Science Laboratory, Erlang Programming Language, http://www.erlang.org/
7) J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. Proceedings of 6th Symposium on Operating System Design and Implementation(OSDI), pp.137-150, 2004.
8) A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. ACM SIGOPS Operating Systems Review, 44(2), pp.35-40, 2010.