高水準言語で記述可能なストリーム処理とバッチ処理の統合フレームワーク

長　裕敏† 塩川　浩昭††,††† 北川　博之††,†††

† 筑波大学院システム情報工学研究科　〒 305–8573 茨城県つくば市天王台 1-1-1
†† 筑波大学計算科学研究センター　〒 305–8573 茨城県つくば市天王台 1-1-1
††† 筑波大学システム情報工学域　〒 305–8573 茨城県つくば市天王台 1-1-1
E-mail: †denam96@kde.cs.tsukuba.ac.jp, ††{shiokawa,kitagawa}@cs.tsukuba.ac.jp

あらまし　近年のセンサーデバイスの発展やマイクロブログの普及から大量のデータを容易に取得することが可能となり, 大規模データ処理のニーズが高まっている. 代表的な処理方式にはデータを継続的に処理するストリーム処理方式とデータを一括で処理を行うバッチ処理方式があり, 利用者は処理内容に合わせて適切な処理方式を選択する必要がある. しかしながら, 両処理方式は処理モデルが大きく異なるため, 利用者には両処理方式のモデルを理解し, 選択した方式に合わせた適切な実装を用意するコストが生じる. そこで本稿ではストリーム処理方式及びバッチ処理方式を統合し, 容易に実装及び実行可能な統合フレームワークを提案する. 本稿で提案する統合フレームワークでは, (1) 両方式に対する統一型処理記述, および (2) 統一型処理記述解析系を与えることで, 両方式を容易かつ適切に連携させる. 本稿では, 我々が提案する統合フレームワークを概説するとともに, プロトタイプを用いた評価実験を通じて提案フレームワークの有用性について議論する.

キーワード　ストリーム処理，バッチ処理，半構造データ

1. はじめに

情報化社会の発展に伴い, 人々が扱うデータ量は増加の一途をたどっている. 巨大なデータから人々にとって有益な情報を獲得するためには, データを解析処理することが重要であり, 様々な分野において高度かつ高速なデータ処理系が求められている. 巨大なデータを処理するために一般的に用いられるアプローチとして, ストリーム処理方式とバッチ処理方式が存在する. ストリーム処理方式はデータを蓄えず即時的に処理する方式であるのに対し, バッチ処理方式はデータを予めストレージに蓄えて処理する方式である. 前者はメモリに収まらないデータ量の処理には処理効率が落ち, 処理のレイテンシが大きくかかるが, 差分計算により低い処理のレイテンシで処理結果を得ることができる方式であり, 既存の処理系に Storm [2] や STREAM [5], S4 [6] 等が存在する. 後者は 二次記憶領域へのアクセスが発生するため処理のレイテンシは大きいが, メモリに収まらない大規模データを効率よく処理できる方式であり, 既存の処理系に OSS の Hadoop [1] や Spark [7] 等が存在する. 従来, これらの処理系の利用者は対象データや処理要求に応じて適切な処理方式を選択して解析処理を行う必要がある.

ところが近年のデータ利用の高度化に伴い, 両処理方式を組合せて使用する機会が増加してきている. 例として, Twitter から最近のニュースに関連があるツイートデータの取得を行う場合を考える (図 1). この例では, ストリーム処理方式でオンラインニュースのテキストに対して単語分割を行いニュースに出現した単語データを二次記憶領域に蓄積し, 一方でバッチ処理系で蓄積したニュースの単語データに対して一定時間ごとに集計処理を行い頻出単語を抽出して二次記憶領域に保存し, さらに, ストリーム処理方式でツイートデータ中にニュース高頻出単語を含んでいるツイートデータを取得するといった処理手順を考えることが一般的である. ところが, ストリーム処理方式とバッチ処理方式ではフレームワークが異なるため, 利用者はそれぞれのフレームワークに沿った実装をそれぞれ行う必要があり学習コスト, 実装コストがかかり大きな負担となる.

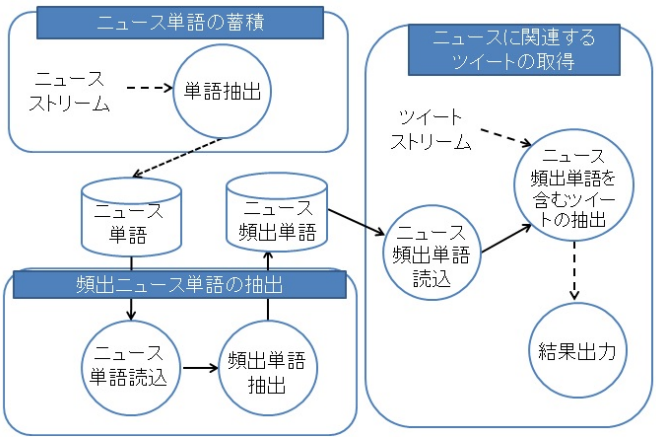

図 1　最近のニュースに関連するツイートの取得

そこで本稿ではストリーム処理方式とバッチ処理方式を統合し, 高水準言語を用いて両方式を容易に記述可能な統合フレームワークを提案する. 本稿で提案する統合フレームワークはストリーム処理方式とバッチ処理方式の両方式に対して柔軟に対応可能な半構造データ形式 JSON [3] を用いてデータを管理す

```
{
 “string” : “Hello, World.”,
 “number” : 256,
 “boolean” : true,
 “null” : null,
 “array” : [1, 2, 3, 4],
 “objects” : { “nest” : “object” }
}
```

図 2　JSON の例

る．そして, 2 つの処理方式を基本演算の組み合わせによるデータ処理フローで統一的に記述可能なイベント駆動型処理記述法及び処理記述を解析し適切な処理方式で選択する解析系を提供する．これにより利用者は処理記述法を基に JSON 形式のストリームデータ及びストレージに蓄積したデータに対する処理記述を用意するのみで, 両処理方式を実行可能にする．

本稿では提案する統合フレームワークの実装として, JSON に対する処理記述法の一つである Jaql [9] を拡張したプロトタイプを構築し評価を行う．本プロトタイプシステムはストリーム処理系に Spark Streaming [8] を用い, バッチ処理系に Spark [7] を用いており利用者は処理要求を処理記述法に従って記述することで適切な処理系を動的に選択し処理を実行する．また評価実験として, 統合フレームワークによる処理内容と汎用言語により実装した処理内容のスループットとレイテンシの性能差およびコード量を比較することで処理性能の劣化の有無, および実装コストが削減できているか確認する．

本稿の構成は以下の通りである．まず第 2 節で本研究の前提となる知識について概説する．その後, 第 3 節では提案する統合フレームワークについて述べる．第 4 節で提案する統合フレームワークを評価するための実験について述べ, 第 5 節で本稿に関連する研究について述べる．最後に第 6 節で本稿のまとめを述べる．

2. 事 前 準 備

本節では提案する統合フレームワークで用いるデータ構造である JSON [3], 提案処理記述の基となる Jaql [9], プロトタイプで用いる処理系である Spark [7] および Spark Streaming [8] について概説する．

2.1 JSON

JSON [3] は JavaScript 言語を用いた半構造データオブジェクトの表記方法である．JSON はデータ全体を配列または変数名と値のペアを列挙したオブジェクトとして記述する．値として利用可能なデータ型には数値型, 文字列型, ブール型, null, 配列, オブジェクトがある．

オブジェクトはデータを中括弧 ({}) で囲むことで記述する．オブジェクトの内部は変数名と値をコロン（:）で区切ったペアをカンマ区切りで列挙していく．値に配列やオブジェクトを取ることができるため, 配列やオブジェクトを入れ子にすることができる．図 2 に JSON オブジェクトの例を示す．オブジェクトは前述の通り中括弧で囲まれ、変数名と値のペアがカンマ区切りで列挙されている．入っている値は上から, 数値型, 文字列型, ブール型, null, 配列, オブジェクトを示している．

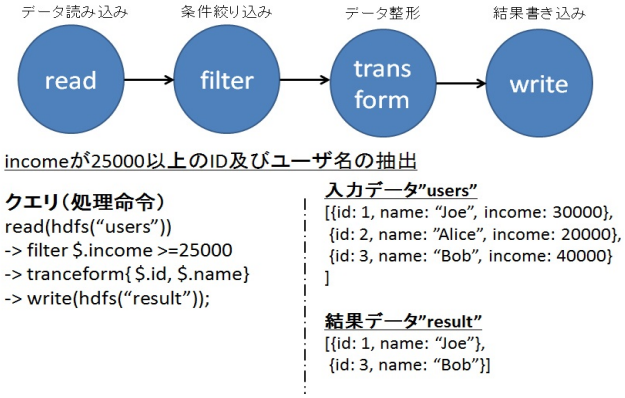

図 3　Jaql 処理記述の例

2.2 Jaql

Jaql [9] は Hadoop 上の JSON 形式の蓄積データを処理するための関数型照会言語である．Jaql は処理をデータフロー形式で記述することで, JSON データに対するフィルタや結合, 集約などといった基本的な演算をシンプルかつ短いコード量で記述することが可能である．また, Jaql は Java で記述されたユーザ定義関数が扱えるため, 基本演算だけでは行えない複雑なデータ処理にも対応できる．Jaql は記述した処理を登録すると, 処理内容を自動的に Hadoop ジョブに変更することができるため並列処理への対応が容易であるという特徴を持つ．Jaql 処理記述の例を図 3 に示す．入力データ中の JSON オブジェクトの内, income が 25000 以上の ID と名前を取得する処理記述となっている．処理記述の詳細は以下通りである．

1)　read 演算によって HDFS に格納されている users ファイルからデータを読み込む．

2)　filter 演算によって income が 25000 以上のデータに絞り込む．

3)　transform 演算によってデータを ID, 名前に整形する．

4)　write 演算によって整形後のデータを HDFS 上の result ファイルへ書き込む．

2.3 Spark

Spark は RDDs [7] というデータ構造を利用した分散処理システムである．RDDs は読み取り専用の分割されたレコードの集合体であり, ストレージ及び他の RDDs に対してフィルタや結合処理等の結果が確定した操作を行うことで生成される．RDDs はその RDDs が生成された系統情報を保持しており, 一括書込みのみを許容する．これによって RDDs は効果的な耐障害性が提供され, 一括操作のみ実行可能であるためにデータの局所性に基づいて処理をスケジュールし実行時のパフォーマンスを向上させることが可能である．また, Spark は処理中に発生する中間データを二次記憶領域に書き込まずオンメモリで保持するため, 中間データを二次記憶領域に書き込む必要のある Hadoop と比較して高速に処理を行うことができる．

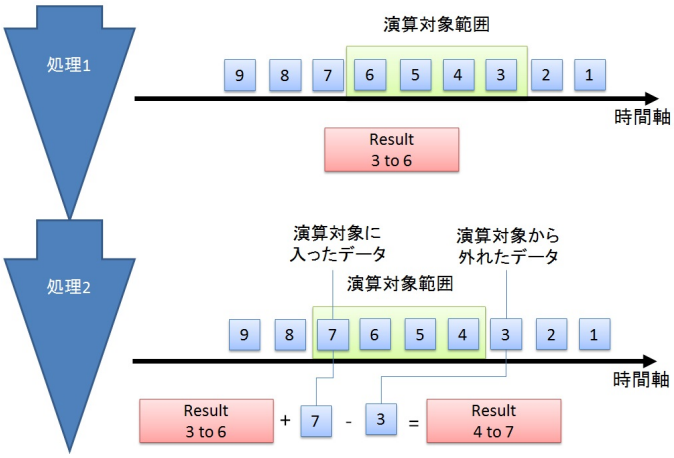

図 4　差 分 処 理

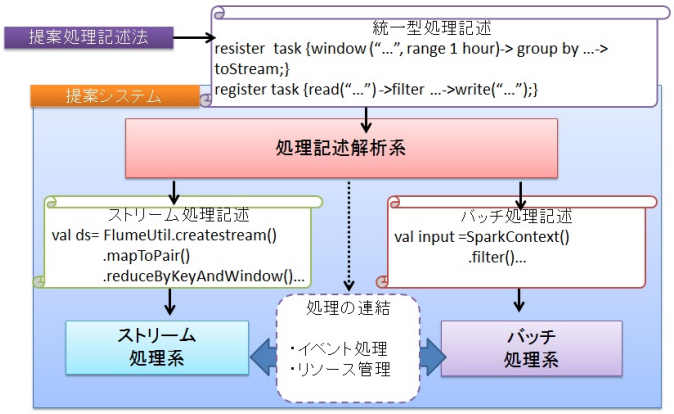

図 5　提案フレームワーク

2.4　Spark Streaming

Spark Streaming は Spark [7] を拡張した大規模ストリーム処理フレームワークであり, DStream [8] というストリーム処理モデルを基に実装されている. ストリームデータを短い時間間隔で区切ったチャンクとしてまとめ, チャンクを RDDs として扱うことで, Spark 同様に RDDs に対する処理記述で処理出来るようにしている. そのため 1 レコードごとに処理を行うイベント駆動モデルの S4 や Storm と比較して処理のレイテンシは落ちるが, 高いスループットを出すことができるのが特徴である. また, Spark Streaming は, 図 4 のように処理結果を中間データとして保持することで, 新たな入力データに対する差分処理を行う. これにより, ストリームデータに対する処理で必要となる計算コストを最小化し, 処理のレイテンシの増加を抑える工夫をしている.

3.　提案する統合フレームワーク

3.1　概　　要

本稿で提案する統合フレームワークの概要を図 5 に示す. 提案する統合フレームワークでは JSON 形式のストリームデータと蓄積データを処理対象とする. 利用者は統合フレームワークに対して Jaql を拡張した統一型処理記述を用いて処理を登録することで, 両処理方式を利用する. 統一型処理言語の詳細は 3.2 節で述べる. 登録された処理記述は処理記述解析系を通じて, ストリーム処理方式とバッチ処理方式の中から適切な処理方式を自動で選択し, 対応する処理方式で実行可能なコードを生成する. 処理記述解析系の詳細については 3.3 節で述べる. 上記の統合フレームワークを利用することで, 利用者はデータ処理を容易かつ適切な処理方式で実行可能になる.

3.2　処理記述法

本節では図 5 における統一型処理記述の詳細について述べる. 提案する処理記述法は Jaql を拡張しストリームデータと蓄積データに対して基本演算のデータフローで処理を記述する. 本稿で扱う演算子は表 1 の通りである. 演算は大きく分けて 3 つに分類され, データの入力演算, 操作演算, 出力演算からなる.

最初に入力演算はストレージまたはストリームからデータを読み込む. ストレージから読み込む処理を行う場合は read 演算, ストリームからデータを読み込む場合は window 演算を用いる. read 演算は指定したファイルに対して指定したタイミングで読み込みを行う演算である. これに対して window 演算では指定した範囲ごとにストリームデータを読み込み, 後続の演算へストリームデータを受け渡す.

次に操作演算について述べる. 操作演算では行いたいデータ処理内容に合わせてデータ操作演算を適用する. 表 1 にあるように, 提案する統合フレームワークではフィルタリングや集約, 結合演算など主要な基本演算を提供する.

最後に出力演算について述べる. 出力演算では, 操作演算から得られた出力をストレージまたはストリームとして出力する. 二次記憶領域へ出力する場合には write 演算, 二次記憶領域に追記する場合には append 演算, ストリームへ出力する場合には tostream 演算を用いる. 本稿では入力演算・操作演算・出力演算という一連の流れを Task としてシステムに登録することで適切な処理方式で処理を実行する.　各演算間は従来の Jaql 同様に”->” 記号で演算を組み合わせてパイプライン方式で記述する. 記述例の詳細については 3.4 節にて述べる.

表 1　扱える演算子群

分類	演算子名	定義
入力演算	window	ストリームデータ取得と演算対象抽出
	read	二次記憶領域から静的データ取得
操作演算	filter	述部評価が真のオブジェクト抽出
	transform	射影, JSON 値の抽出
	sort	1 つ以上の JSON 値で入力ソート
	top	入力の開始 k 個のオブジェクトを選択
	group	グループ化し, 集約処理
	join	2 つ以上の入力を結合して出力
	udf	ユーザが定義した処理の実行
出力演算	tostream	結果をストリームデータで出力
	write	結果を二次記憶領域に上書き保存
	append	結果を二次記憶領域に追記保存

表 2　演算子メモリ使用量

演算子名	メモリ使用量 $C_i(byte)$	出力データ量 $O_i(byte)$
rowWindow	L・B	L・B
rangeWindow	R・B・T	R・B・T
filter	0	$\sigma \cdot O_{i-1}$
top		
group	$\alpha \cdot \mathrm{S} \cdot O_{i-1}$	$\alpha \cdot \mathrm{S} \cdot O_{i-1}$
join	$O_{i-1} + O_{i-2}$ $+ \sigma \cdot \mathrm{S} \cdot O_{i-1} \cdot O_{i-2}$	$\sigma \cdot \mathrm{S} \cdot O_{i-1} \cdot O_{i-2}$
transform	$\mathrm{S} \cdot O_{i-1}$	$\mathrm{S} \cdot O_{i-1}$
udf		
sort	O_{i-1}	O_{i-1}
tostream		
write		
append		

3.3　処理記述解析系

本節では図 5 における処理記述解析系について述べる．一般的にストリーム処理系は演算結果を中間結果としてオンメモリで保持する差分計算をすることで高速に処理するため, 実行頻度が多く, メモリをあまり使用しない継続的に実行する処理に適している．一方でバッチ処理系は演算対象が大きく, 以前の中間結果と相関がない処理に適している．

そこで, 差分計算を行う処理において処理内容の重なりが大きく, 再計算の計算コストが大きい処理はストリーム処理方式で実行し, それ以外の処理はバッチ処理方式で実行する．しかし, ストリーム処理方式は差分計算のために直前の処理結果を保持する必要があるため, バッチ処理方式より多くメモリを消費する．そのため複数の処理を登録すると処理速度が落ちると考えられるので, 処理内容の重複率が大きくてもメモリ使用量が一定量より大きい場合はバッチ処理方式で処理を実行する．

差分計算を必要としない read 演算から始まるデータフローはバッチ処理として振り分け, window 演算から始まるデータフローでは処理内容の重複率とメモリ使用量に従って処理方式を決定する．なお, 重複率とメモリ使用量の閾値については予備実験をして議論する．本稿では使用メモリ量を推定するために以下のコストモデルを導入する．処理記述をストリーム処理系で実行した時の使用メモリ合計量を C とし, 各演算子の使用メモリ量を C_i 処理記述に使用している演算子数を n とすると C は $C = \sum_{i=1}^{n} C_i$ で表される．

各演算子のメモリ使用量は表 2 に示す, 各種記号は下記の通りである．

- 演算対象とするデータ数：L(docs)
- 演算対象とする時間幅：T(s)
- ストリームの平均到着レート：R(docs/s)
- ストリーム 1 要素のデータサイズ：B(byte/docs)
- 出力するデータ量：O_i(byte)
- 選択率：σ
- 集約率：α
- 収縮率：S

3.4　統合フレームワークを用いた処理記述例

本稿で提案する統合フレームワークを用いた処理記述の例を図 6 に示す．例として扱う処理要求は「最近のニュースに関連があるツイートの取得」である．これを以下の 3 つの処理記述を登録することで実現する．

1)　ニュースデータを RSS から取得し続ける．1 日分のニューステキストを単語分割して名詞のみをファイルに保存する．

2)　1 日おきに保存したニュースの単語群に対して集約, カウント, ストップワードの除去を行い, ニュースで高頻出の単語を 200 語抽出し保存する．

3)　Twitter Streaming API から tweet データを取得し続け, tweet 中にニュースの高頻出の単語が含まれていたら出力する

処理記述中の register function はユーザ定義関数 (UDF) の登録であり, 上から「ニューステキストの単語分割と名詞抽出」,「ストップワードの除去」,「ツイートテキスト中に含まれる名詞をオブジェクトに追加」を行い, 各 register task にて処理の内容を登録している．

collectNewsWords は処理記述 1) に該当する．window 演算にてニュースデータを取得し, UDF を用いて単語分割, 名詞抽出を行い, write 演算で二次記憶領域に保存する．この処理は window 演算内で演算対象を range 1 day, すなわち 1 日分のデータとしているので, ニュースデータが一日分貯まるたびに実行される．

getNewsTopic は処理記述 2) に該当する．read 演算で collectNewsWords で貯めたデータを読み込み．group 演算によって名詞ごとに集約, カウントを行い, udf 演算によってストップワードを除外する．sort 演算と top 演算によって高頻出の単語 200 以外を除外し, write 演算によって二次記憶領域に保存する．実行のタイミングは read 演算内で trigger collectNewsWords とあるので, collectNewsWords の一連の処理が終了する度に行われる．

getNewsTweet は処理記述 3) に該当する．window 演算で直近 5 分のツイートデータを取得し, udf を用いてオブジェクトにツイートテキスト中の名詞を付与する．read 演算で getNewsTopic で計算したニュースの高頻出単語を読みこみ, join 演算でツイート中の名詞にニュースで高頻出の単語を含む場合, 結果を stream で出力する．これは window 演算から始まるデータフローと read 演算から始まるデータフローでの結合処理なのでそれぞれの条件である 1 分おき, あるいは getNewsTopic の処理が終わるごとに後続の演算が実行される．

処理の振り分けは collectNewsWords, getNewsTopic は差分処理を行わないためバッチ処理, getNewsTweet は差分処理を行い, メモリ使用量も少ないためストリーム処理系として振り分けられ実行される．

このように, 利用者はストリームデータとストレージに蓄積されたデータに対する複雑な処理を各処理系を意識することなく容易に記述することができる．

```
register function(“SplitWords” , ”myfunction.SplitWords”);
register task collectNewsWords {
window(“NewsStream”,  range 1 day)
-> SplitWords()
-> write(“NewsWords”);}

register function(“EraceStopWords”, “myfunction.EraceStopWords”);
register task getNewsTopic{
read(“NewsWords”,trigger collectNewsWords)
->  group by $.word into { $.word , ct : count($)}
->  EraceStopWords()
-> sort by [$.ct desc]
-> top 200
-> write(“NewsTopic”);}

register function(“AppendSplitWords”, ” myfunction.AppendSplitWords”);
P = read(“NewsTopic”, trigger getNewsTopic);
register task getNewsTweet{
window(“TwitterStream”, range 5 minute, trigger 1minute)
->AppendSplitWords()
-> join P where P.word == $.word into {$.text, P.word}
-> tostream;}
```

図 6　本稿で提案する統合フレームワークを用いた処理記述例

4. 評 価 実 験

4.1 実 験 概 要

本稿で提案した処理記述に従って自動生成した処理内容と, 提案システムを用いずに汎用言語により実装した処理内容のスループットとレイテンシの性能差を比較する. また提案処理システムを用いた処理記述と提案システムを用いずに Java 言語を用いて手動で実装したコード量を比較し, 実装コストに関する優位性を確認する.

本実験で評価の対象とした処理の内容は Twitter のツイートテキストにおける単語の出現回数を計測する wordCount 処理および, 頻出単語上位 10 件を獲得する wordCountTop10 処理の 2 つである. wordCount 処理では, 1 秒毎に直近 30 秒のツイートテキストに対して単語ごとに集計して出力する. wordCountTop10 処理では, 10 秒毎に直近 1 分のツイートテキストに対して単語ごとに出現数を集計して頻出単語 10 個を抽出する.

本実験で使用したデータセットは Twitter の 2016 年 2 月 2 日から 2016 年 2 月 4 日までのツイートテキストである. 入力はスループットやレイテンシの計測の簡易化のためにデータセットを予め単語分割して {word : ツイート単語 } を 1 つのレコードとして二次記憶領域に保存しておき, 処理内容を実行する前に二次記憶領域からメモリに読み込み実行間隔ごとに一定数送信することで同様の条件で処理を行う.

さらに, 本実験では WordCountTop10 処理の実行頻度を変更して計測を行い, レイテンシやメモリ使用量を鑑みて, どちらの処理方式で行うのが適切か決定する処理内容の重複率およびメモリ使用量の閾値についても検討する. 実験環境として Intel(R) Core(TM) i7-4820K CPU @ 3.70GHz と 16.0GB のメモリを使用した.

4.2 WordCount 処理

提案処理記述法を用いた処理記述は図 7 のようになる. 1 行目の register stream によって入力ストリームの名称とストリームデータの形式を指定する. 入力ストリームである {word:ツイート単語 } を TweetWordStream とし, ツイート単語の型を String 型で指定している.

```
 1: register stream TweetWordStream(word String);
 2: register task wordCount{
 3: window("TweetWordStream",
 4:        range 30 second,
 5:        trigger 1 second)
 6: -> filter $.word != ""
 7: -> group by $.word into
 8:           {$.word, ct:count($.word)}
 9: -> tostream;
10: };
```

図 7　wordCount 処理 提案手法プログラム

2 行目から 10 行目に亘る register task wordCount によって単語ごとの集計処理を記述する. 3 行目から 5 行目の window 演算の第 1 引数で入力ストリームを先に登録した TweetWordStream に設定し, 第 2 引数で演算対象範囲を直近 30 秒, 第 3 引数で実行間隔を 1 秒に指定している. 6 行目の filter 演算で空文字を除去し, 7,8 行目の group 演算によって word 毎にグループ化し count 処理によって合計数を求め, データ形式を単語名, 出現数の組に変換している. 最後に 9 行目の tostream 演算で結果を出力する.

提案処理記述法を用いず Java 言語を用いて手動実装した処理記述は図 8 のようになる. 12 行目から 19 行目の ssc から連なる処理が登録する wordCount である. 1 行目から 7 行目において登録する上で必要な関数を全て実装している. filterFunc は文字列が空文字かの確認処理, makePairFunc は入力データの<入力データ,1>の key-value 型への変換処理, reduceFunc で差分処理における演算範囲に追加されるデータの加算処理, reduceFuncInv で差分処理における演算範囲から出ていくデータの減算処理, outputFunc で入力データを出力する処理をそれぞれ実装する.

8 行目から 10 行目の ssc ではログ等の詳細設定を指定し, 実行間隔を 1 秒に設定している. 12 行目の queueStream 演算で入力ストリームを設定する. 13 行目の filter 演算で空文字除去, 14 行目の mapToPair 演算で単語を<入力データ, 1>のペアに変換する. 15 行目から 18 行目の reduceByKeyAndWindow 演算で直近 30 秒の出現単語を 1 秒おきに集計する. この時, reduceFunc 及び reduceFuncInv を用いて新たに直近 30 秒に入る 1 秒分のデータを加算し, 直近 30 秒から外れた 1 秒分のデータを減算することで計算コストを削減する. 最後に 19 行目の foreach 演算で結果を出力する.

このように利用者は提供される API に従い, 入力データをどのように変換するかに応じて適用する演算を選び, 引数に行いたい処理を記述した上で入力の型と出力の型をが正確に設定された関数を用意する必要がある. そのため学習コストが大きく, コード量においても提案手法だと 10 行で記述できるが, Java

```
 1: Queue<JavaRDD<String>> TweetWord = ...;
 2: Function<String, Boolean> filterFunc = ...;
 3: PairFunction<Str,Str,Int> makePairFunc=...;
 4: Function2<Int,Int,Int> reduceFunc=...;
 5: Function2<Int,Int,Int> reduceFuncInv=...;
 6: Function<JavaPairRDD<Str,Int>,Void>
 7:                              outputFunc=...;
 8: JavaStreamingContext ssc =
 9:    new JavaStreamingContext(
10:    new SparkConf().setAppName("wordCount"),
11:    Duration.seconds(1) );
12: ssc.queueStream(TweetWord)
13: .filter(filterFunc)
14: .mapToPair(makePairFunc)
15: .reduceByKeyAndWindow(reduceFunc,
16:         reduceFuncInv,
17:         Duration.seconds(30),
18:         Duration.seconds(1))
19: .foreach(outputFunc);
20: ssc.checkpoint("./checkpoint");
21: ssc.start();
22: ssc.awaitTermination();
23: };
```

図 8　wordCount 処理 Java プログラム

言語で記載した場合は省略している分も含めて 39 行記述する必要があり, 提案手法と比較して約 4 倍多く記述する必要があるので実装コストが大きくかかる.

図 9 に提案手法および Java を用いた実装に対するレイテンシの比較を記載する. 本実験では入力ストリームの入力レートを 10 K records/s, 100 K records/s, 1 M records/s と変化させた際の結果を述べる.

図 9 からも分かるように, 提案フレームワークは全体的に Java による実装と比較してレイテンシの小さな増加が確認できるが, Java による実装とほぼ同程度の性能を示していることが分かる. レイテンシの増加の原因は Java による実装では処理に必要なデータのみを対象とするのに対し, 提案フレームワークではどのようなデータ型が来ても対応できるように, 処理の開始時に対象 JSON データを JSON の文字型や数値型のデータを保持するクラスに変換し, 処理ごとにクラスから対象とする値を取り出す処理を行うように実装しているためであると考えられる. 1 M records/s を用いた場合に約 0.8 秒の処理時間を要するため, これ以上の入力レートを増加させると実行間隔 1 秒に追いつかないので最大スループットは約 1.2 M records/s と判断できる.

本実験では以上の結果を通じて提案手法は実装に必要なコード量を大幅に削減しつつ, Java による実装に対してほぼ同程度の性能処理を達成していることを明らかにした.

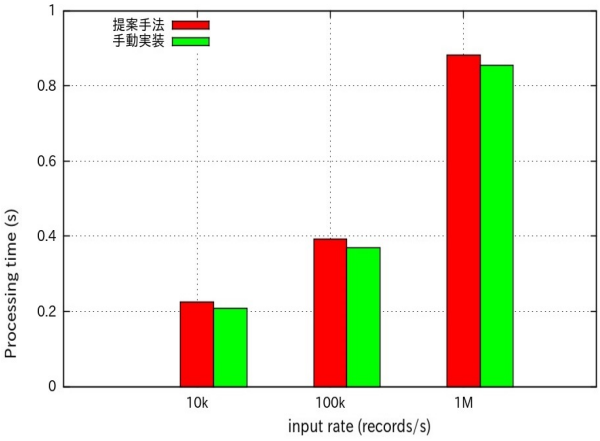

図 9　wordCount 処理の処理遅延

```
 1: register stream TweetWordStream(word String);
 2: register task wordCount{
 3: window("TweetWordStream",
 4:        range 1 minute,
 5:        trigger 10 second)
 6: -> filter $.word != ""
 7: -> group by $.word into
 8:           {$.word, ct:count($.word)}
 9: -> sort by $.ct desc
10: -> top 10
11: -> tostream;
12: };
```

図 10　wordCountTop10 処理 提案手法プログラム

```
 1: ssc.queueStream(TweetWord)
 2: .filter(filterFunc)
 3: .mapToPair(makePairFunc)
 4: .reduceBykeyAndWindow(reduceFunc,
 5:       reduceFuncInv,
 6:       Duration.minutes(1),
 7:       Duration.seconds(10))
 8: .mapToPair(swapFunc)
 9: .transformToPair(sortFunc)
10: .foreach(outputFunc);
11: };
```

図 11　wordCountTop10 処理 Java プログラム

4.3　wordCountTop10 処理

提案処理記述法を用いた処理記述は図 11 のようになる. 8 行目の group までの処理は wordCount の実行頻度や対象範囲以外は同一のものである. その後, 9 行目の sort 演算によって出現数をキーとして降順に並び替え, 10 行目の top 演算で上位 10 件を抽出し, 11 行目の tostream 演算で結果を出力する.

提案処理記述法を用いず Java 言語を用いて手動実装した

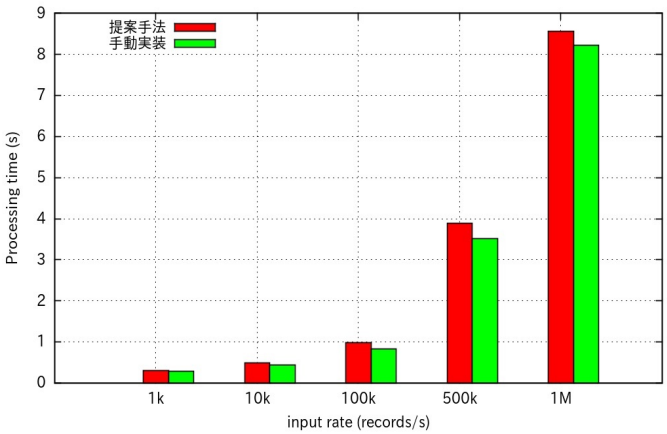

図 12　WordCountTop10 処理の処理遅延

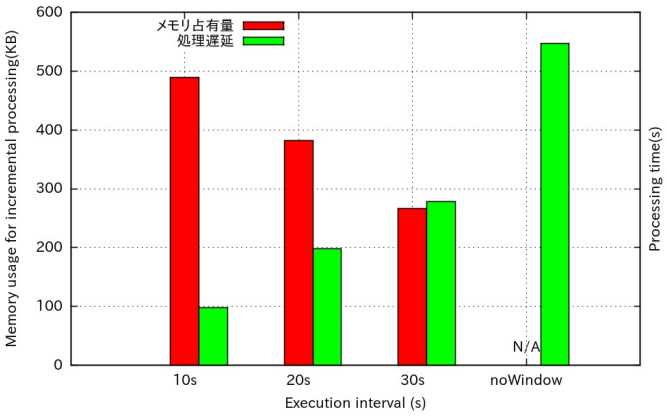

図 13　wordCountTop10 処理の差分計算に用いるメモリ占有量, 処理遅延

処理記述は図 11 のようになる．こちらも reduceByKeyAndWindow 処理までは wordCount の実行頻度や対象範囲以外は同一のものである．7 行目までの処理でデータは,<単語, 出現数>の key-value 形式の Tuple データである．この後 8 行目の mapToPair 演算で key 値と value 値を swap し, 9 行目の transformToPair 演算で出現数であるキーごとに降順に並び替える．最後に 10 行目の foreach 演算で上位 10 件を出力する．

図 12 に提案手法および Java を用いた実装に対するレイテンシの比較を記載する．本実験では入力ストリームの入力レートを 1 K から 1 M records/s に変化させた際の結果を述べる．

図 12 をみると, 1 M records/s を用いた場合に約 8 秒の処理時間を要している．wordCount の時より 10 倍程度処理時間がかかるのは, 実行間隔が 10 秒なので処理するレコード数も 10 倍になっているためである．入力レートをこれ以上増加させると実行間隔 10 秒に追いつかないのでこちらも wordCount と同様に最大スループットは約 1.2 M records/s と判断できる．

図 12 と図 9 を比較して, wordCount の時と同様に Java による実装とほぼ同程度の性能を示しているが, 図 9 の時よりも Java による実装より提案フレームワークの方がさらにレイテンシが増えている．これは sort 演算や top 演算が追加され, 提案フレームワークだと JSON の文字型や数値型のデータを保持するクラスから参照してくる回数が増えているためであると考えられるので, 今後この点を改善すれば提案フレームワークと汎用言語の実装において同様の性能処理を達成できると考えられる．

4.4　処理の重複率による変化

wordCountTop10 処理において入力レートを 100 K records/s, 演算対象範囲を 60 秒に固定し差分計算をしないで行う場合と実行頻度を 10, 20, 30 秒おきに差分計算を用いて実行し, レイテンシと差分計算に用いているメモリ占有量を計測する．

実験結果を図 13 に示す．差分計算を用いる場合は 30 秒おきならば直前の 30 秒分の中間結果, 10 秒おきならば直前の 50 秒分の中間結果を保持しておく必要があるため, 処理の重複率が高いほどメモリの占有量は増えている．しかし, 計算量は 10, 20, 30 秒おきに実行する場合それぞれ 1/6, 1/3 , 1/2 となるのでレイテンシも計算量に合わせて削減されている．このことから静的な閾値を設定するのでなく差分計算を行える場合はメモリ量が許す限りストリーム処理で行う方が処理効率が高いことが確認できる．

そのため最初にクラスタが使用できるメモリ量と登録できるタスクを設定しておき, タスクが登録される度にコストモデルを用いてメモリ使用量を推定し, 設定したメモリ使用量が許す限りにおいて, 処理の重複度が大きく, 実行間隔が小さく低レイテンシを求めらている処理を優先的に動的にストリーム処理として振り分けて実行することを今後の課題とする．

5.　関 連 研 究

ストリーム処理方式とバッチ処理方式を統合的に扱うための処理系はこれまでいくつか提案されている [4], [8], [10]．本稿ではそのなかでも近年最も代表的な研究である SummingBird [10] と Cloud Dataflow [4] ついて紹介する．いずれの研究もストリーム処理方式とバッチ処理方式を単一の言語で記述可能という観点では本研究と類似するが, 本研究では汎用言語でプログラムすることなく JSON 形式のストリームデータと蓄積データに対して高水準なデータ操作演算のみで両処理方式での実行が可能な点において異なる．

5.1　SummingBird

SummingBird [10] は MapReduce [11] 処理をストリーム処理方式とバッチ処理方式を同時に実行できる Scala のライブラリである．SummingBird は Hadoop [1] と Storm [2] の MapReduce ライブラリを抽象化することでバッチ処理とストリーム処理の MapReduce ジョブを同一ロジックで記述可能にしている．

SummingBird の核となるコンセプトは Producer と Platform である． Producer は Hadoop の Mapper や Reducer, Storm の Spout や Bolt といったデータに対する変換処理や他のサーバへの転送処理を抽象化している．したがって Producer にはデータに対する変換処理や他のサーバへの転送処理に関するメソッドが多数定義されており, そのメソッドを組み合わせて適用していくことで処理を記述する．

Platform は実際に実行するフレームワークである Storm や

Hadoop の MapReduce ライブラリを抽象化したものである. Platform の各インスタンスは Producer で記述した処理を受け取ると指定した処理系で処理を実行する. これにより利用者は Producer を用いた処理記述を 1 つ用意することで Hadoop と用いたバッチ処理と Storm を用いたストリーム処理を同時に実行することができる. しかし本研究とは異なり, 両処理方式を意識してデータ処理を記述する必要がある.

5.2 Cloud Dataflow

Cloud Dataflow [4] は, バッチ処理とストリーム処理を同一のコードで記述できる大規模データ分析処理基盤である. 専用の Java ライブラリを使うことでストリーム処理とバッチ処理を同一ロジックで記述できるようにモデル化されている.

初めに, ストリームデータ及びストレージに蓄積したデータを PCollection と呼ばれる, 要素を際限なく持てる同一型のデータ集合に変換する. その後, PCollection に対してパイプライン方式でメソッドを記述していくことで両処理を同一のロジックで記述できる. 各 Pcollectioln は要素数や時間幅を指定できるウィンドウを持つ. PCollectioln 内の各要素はそれぞれの要素が作成された時刻であるタイムスタンプを持っており, PCollectioln 内の各要素はウィンドウによって分割され保持される. 分割された集合ごとに適用されたメソッドを並列に実行ことで高速に処理を行う. 本研究では JSON データに対して, より高水準な言語で両処理方式を実行可能としている.

6. 結　　論

本稿では, ストリーム処理方式とバッチ処理方式を統一的に実行可能な統合フレームワークを提案した. 本稿で提案する統合フレームワークは, JSON による半構造データ形式に対する統一型処理記述および, その処理記述解析系を備える. これにより, 利用者に対して両処理方式を意識する必要のないデータ処理を提供した. また, 評価実験において統合フレームワークを利用することで汎用言語で実装した場合と比較して, 同程度の性能で実装コストの削減ができていることが確認できた.

今後の課題としては, 使用可能メモリ量と登録可能なタスク数を制限した場合における処理の振り分けの最適化が考えられる.

7. 謝　　辞

本研究の一部は，科研費・基盤研究 B(26280037) の助成を受けたものである.

文　　献

[1] Apache Software Foundation. Hadoop. http://hadoop.apache.org
[2] Storm. http://storm-project.net
[3] JavaScript Object Notation. http://json.org
[4] Tyler Akidau et al. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. In PVLDB, Volume 8, No.12, 2015.
[5] R. Motwani et al. Query processing, resource management, and approximation in a data stream management system. In CIDR, pages 245-256, 2003.
[6] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In ICDMW, pages 170-177. 2010.
[7] M. Zaharia et al. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. In NSDI, pages 2-2, 2012.
[8] M. Zaharia et al. Discretized streams: fault-tolerant streaming computation at scale. In SOSP, pages 423-438 2013.
[9] Kevin S. Beyer et al. Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. In PVLDB,Volume 4, No. 12, 2011.
[10] Oscar Boykin, Sam Ritchie, Ian O'Connell, and Jimmy Lin. 2014. Summingbird: A Framework for Integrating Batch and Online MapReduce Computations. In VLDB, Volume 7, No.13, 2014.
[11] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In OSDI, pages 137-150, 2004.