ストリーム処理とバッチ処理の統合と実行最適化

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
E-mail: †denam96@kde.cs.tsukuba.ac.jp, ††{shiokawa,kitagawa}@cs.tsukuba.ac.jp

あらまし　センサーデバイスやソーシャルメディアの発展により, 様々な分野で大規模データの利活用に注目が集まっている. 大規模データ処理のアプローチとして, データを蓄えて一括で処理するバッチ処理方式とデータを継続的に処理するストリームデータ処理方式がある. 近年ではデータ利活用の高度化に伴い両処理方式を合わせて使用する機会は多いが, 各処理方式ではデータモデルが異なるため, データを処理する利用者の学習・実装コストは大きく, 適切な処理方式を選択するのは困難である. そこで本研究では, 両処理をデータフローで統一的に記述できる処理記述法と処理を適切な処理方式で自動実行する処理基盤を提供することで, 容易で効率的なデータ処理を実現する.

キーワード　ストリーム処理，バッチ処理，統合

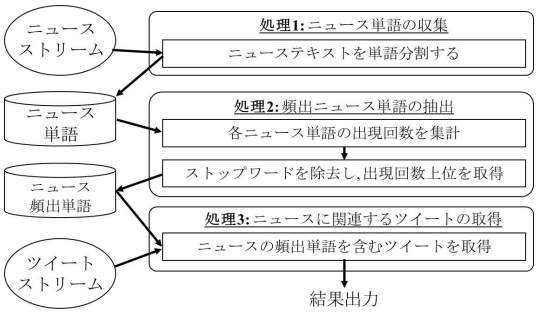

図 1　例：最近のニュースに関連するツイートの取得

1. 序　　論

近年ではセンサーデバイスやソーシャルメディアの発展により，人々が扱えるデータは爆発的に増大している．これに伴い，様々な分野で大量データの利活用に注目が集まっている．例えば，ソーシャルメディアの情報を利用した製品・サービスのポジネガ分析やセンサデータを活用したシステムの異常検知などが典型的な活用例として挙げられる．

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, データを蓄えて一括処理するバッチ処理方式と，データを蓄えずに常時処理するストリーム処理方式がある．バッチ処理方式ではデータを一括処理するため，レイテンシは大きくかかってしまうがメモリ効率がよくスループットも高い．既存の処理基盤として Hadoop [1] や Spark [2] などが開発されている．一方で，ストリーム処理方式ではデータを常時処理するため，バッチ処理方式と比較してメモリ効率が低下するものの，レイテンシを抑えて高速に処理を実行できる．既存の処理基盤には，Storm [3]，Spark Streaming [4] が開発されている．

データ処理を行う利用者は，処理内容に応じて様々なデータ処理基盤から適当な処理方式を選択して処理を実行する必要がある．特に近年ではデータ利用の高度化に伴い，両処理方式を組み合わせて処理要求を満たす機会も多く，データ処理基盤の利用形態が複雑化してきている．しかしながら，各処理方式は基本的にデータモデルが異なるため，実際の運用の場面においては適切な処理方式の割り当ての問題と学習・実装コストの問題が存在する．

複数の処理方式を組み合わせる例として，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 に処理内容を示す．この例では処理内容を 3 段階に分けて考える．処理 1 では，ニュースストリームからニュースに出現した単語群を収集する．ニュースストリームに含まれるテキストを単語分割し，ニュースに出現した単語群を二次記憶領域に格納する．処理 2 では，ニュースに頻繁に出現した単語群を摘出する．処理 1 で保存したニュースの単語群を読み込んだ後に，各単語に対して集計処理を行い，単語の出現数をカウントする．その後，出現頻度が高いものをいくつか抽出して二次記憶領域に格納する．最後に処理 3 において最新のニューストピックに対応するツイートデータを抽出する．ここでは，処理 2 にて得られたニュースの頻出単語とツイートストリームの 2 つを入力データとして読み込む．その後，2 つのデータを照会し，ニュースの頻出単語が含まれているツイートデータを抽出し結果として出力する．

この例では先に挙げた通り，データ処理の利用者にとって 2 つの課題が存在する．

課題 1：学習・実装コストが大きい

処理 1 にストリーム処理方式，処理 2 にバッチ処理方式を適用すると仮定する．この場合，利用者はそれぞれの処理方式の処理基盤に従ったプログラムを用意する必要がある．しかしながら，ストリーム処理方式とバッチ処理方式では処理モデルが多くの場合に異なるため，利用者に高い学習コストと実装コストを必要とする．したがって，処理のパフォーマンスを最適化するためには各処理方式の実行モデルを理解した上でそれぞれに適した構築・管理する必要がある．

課題 2：適切な処理方式の選択が困難

効率的に処理を行うにはそれぞれの処理段階において適切な

処理方式を選択する必要がある．処理 1 ではストリームデータ，処理 2 では蓄積データのみを対象とするので，それぞれにストリーム処理方式とバッチ処理方式を適用するのが合理的である．しかし，処理 2 の処理の実行頻度が週に一度といった少ないケースの場合は，処理 1 の処理を常時実行するのは非効率であり，バッチ処理方式を用いた方がメモリを効率的に利用できる可能性がある．また，処理 3 ではストリームデータと蓄積データの両方を入力とするため，どちらの処理方式も利用することができる．そのため，処理を効率良く行うために試行錯誤を重ねて適当な処理方式の割り当てを見つける必要があり，利用者にとって大きな負担となる．

これらの課題を解決するために，近年では Suminngbird [7] や Cloud Dataflow [6] といった両処理方式を統合した処理基盤がいくつか開発されている．しかしながら，これらの処理基盤は課題 1 で挙げた学習・実装コストの問題のみに焦点を当てており，使用する処理方式については利用者自身が事前に選択する必要がある．そのため，課題 2 に示した適切な処理方式の割り当ての問題については依然として解決されていない．

そこで本稿では，簡潔なプログラミングモデルを使用してストリーム処理方式とバッチ処理方式を統合的に利用可能な新しい処理基盤 JsFlow を提案する．JsFlow では以下の 3 つの機能を提供することで，大規模データ処理の容易実行を可能にする．(1) 半構造データ形式として広く普及している JSON [5] を用いて，ストリームデータと蓄積データを JSON レコード集合として抽象化する．(2)Hadoop [1] の JSON データに対する照会言語である Jaql [8] を拡張し，JSON レコード集合に対する処理のデータフローを記述することで，ストリーム処理方式とバッチ処理方式を統一的に扱えるプ処理記述法を提供する．(3) 提案する処理記述法に基づいた処理のデータフローを各処理方式で実行した場合の処理時間とメモリ使用量を見積り，これらの情報から適切な処理方式を自動的に選択する．

本稿の構成は以下の通りである．まず 2 章で本研究の関連研究について述べ，3 章にて予備知識として本稿で提案する JsFlow が利用している JSON [5] や Jaql [8]，Flink [9] について述べる．4 章にて本研究で提案する JsFlow について述べる．5 章では評価実験について述べる．最後に 6 章で本稿のまとめと今後の課題について述べる．

2. 関 連 研 究

バッチ処理基盤とストリーム処理基盤を統合的に扱うための処理基盤がこれまでにいくつか研究開発されている [4], [6], [7], [9]. 本稿では，その中でも代表的な処理基盤である Summingbird [7] と CloudDataflow [6] について説明する．

Summingbird [7] は，MapReduce [10] 処理をストリーム処理方式，及びバッチ処理方式で実行できる処理基盤である．Summingbird では，自身が提供する API を利用した Scala 及び Java 言語で書かれたプログラムを MapReduce プログラミングモデルに基づくバッチ処理基盤 (Hadoop) 及びストリーム処理基盤 (Storm) のジョブに変換して実行できる．SummingBird の中核となる API に Producer API と Platform API がある．Producer API は MapReduce のデータフローを抽象化したものである．`map` や `filter`，`merge`，`join` 等のデータに対する変換処理のメソッドが多数定義されており，これらを組み合わせて入力データに対する処理のデータフローを記述する．Platform API は処理を実行する処理基盤である Hadoop や Storm のライブラリの両方を抽象化したものであり，Producer API を用いて記述したデータフローを指定された処理方式で実行する．そのため，利用者は Producer API に従った処理記述を一度用意することで処理を各処理方式で実行することができる．

CloudDataflow [6] は，Google Cloud Platform で提供される大規模データ分析処理基盤とそのマネージドサービスである．CloudDataflow ではバッチ処理とストリーミング処理を統合するプログラミングモデルを提供している．Cloud Dataflow のプログラミングモデルの中核となるコンセプトに Pipeline と PCollection がある．Pipeline は外部ソースから入力データを受け取り，そのデータを変換し，出力データを生成するという，一連の処理のデータフローを表す．Pipeline 中のデータは PCollection という，レコードを際限なく持てるデータ集合で表される．PCollection 内の各レコードにはタイムスタンプが関連付けられており，ストリームデータを扱う際に Pipeline が提供するウィンドウ処理を用いてタイムスタンプに従って有限サイズのデータ集合に変換すること可能としている．これにより，有限の蓄積データとストリームデータの両方に対して単一の API である Pipeline を用いて処理を記述できるようにしている．

これらの処理基盤はストリーム処理方式とバッチ処理方式が統一されたプログラミングモデルを提供しているため，序論で挙げた学習・実装コストに対する課題の解決に大きく貢献していると考えられる．しかしながら，これらの処理基盤は記述した処理のデータフローをストリーム処理方式とバッチ処理方式のどちらで実行するかについては利用者が全て選択する必要があるため，適切な処理方式の割り当てに関する課題点は解決できていない．

本稿で提案する JsFlow では，両処理方式を統一的に扱えるプログラミングモデルを提供するとともに，処理のデータフローから処理時間とメモリ使用量を概算し，処理方式を自動的に選択して実行するため，大規模データ処理を容易に実現することを可能とする．

3. 予 備 知 識

本章では提案する統合処理基盤 JsFlow で用いるデータ構造である JSON [5], 提案する処理記述法のベースとなる Jaql [8], JsFlow が用いる分散フレームワークの Flink [9] について述べる．

3.1 JSON

JSON [5] は軽量の半構造データフォーマットである．JSON では，データ全体を配列，またはキーと値のペア (フィールド) を列挙したオブジェクトとして記述する．値として利用できる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．配列は全体を角括弧 ([]) で囲み，値をカンマ区切りで列挙していく．オブジェクトは全体を中括弧 ({}) で囲み，キーと値をコロン（:）で区切ったペアをカンマ区切りで列

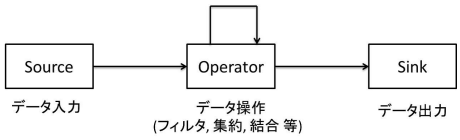

図 2　Jaql プログラミングモデル

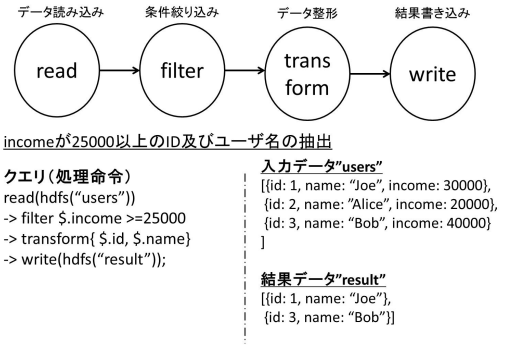

図 3　Jaql 処理記述例

挙していく．値として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して持つこともできる．そのため表形式に限らず，複雑な形のデータを保持できる．

3.2　Jaql

Jaql [8] 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 データに対してフィルタや結合，集約などといった基本的な演算をシンプルで短いコード量で記述することが可能である．Jaql で記述した処理は，システム側が処理内容を Hadoop の MapReduce ジョブに変換して実行するため，利用者は容易に並列処理を行うことができる．

Jaql のプログラミングモデルを図 2 に示す．Jaql ではまず入力データを Source から読み取る．Source はファイルといったデータを読み取る処理を指す．次にデータは照会で指定された演算子 (Operator) により操作する．照会が提供している演算には `filter` によるフィルタリングや `group` による集約，`join` による結合など一通りの基本的な演算子が提供されているため，容易にデータ操作を行うことができる．最後にデータは Sink に出力される．Sink はデータをディスクへ書き込む処理を指す．

図 3 に Jaql の処理記述例を示す．ここでは入力データ中の JSON オブジェクトの内，`income` が `25,000` 以上の ID とユーザ名を取得する処理記述となっている．例では，最初に Source にあたる `read` 演算によって HDFS に格納されている `users` ファイルから JSON データを読込んでいる．次に Operetor に該当する `filter` 演算を用いて `income` が `25,000` 以上のデータに絞り込み，続く `transform` 演算で絞り込んだデータの `id` と `name` を摘出している．最後に Sink にあたる `write` 演算によって結果を HDFS 上に `result` というファイル名で保存している．

3.3　Flink

Flink [9] はオープンソースの分散処理基盤である．ストリームデータに対しては DataStream API，蓄積データに対しては

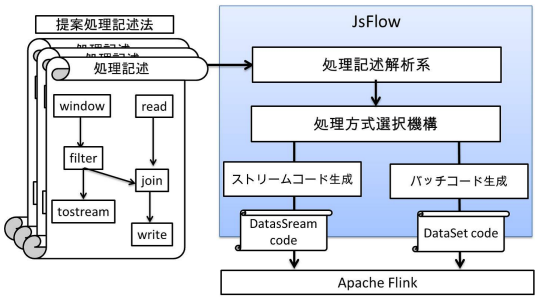

図 4　JsFlow 全体図

Dataset API が提供されており，これらを同一の基盤上で動作させることができる．Dataset API を用いて登録した処理はバッチ処理となるため，データフローに含まれる全ての演算子に対して処理が完了した場合に実行を終了する．これに対し，DataStream API を用いて登録した処理はストリーム処理となるため，データフローに含まれる全ての演算子は常に駆動し続け，継続的に処理を実行する．

各 API では，各レコードに対して同一処理を行い一つあるいは複数のレコードを返す `map`，`flatMap` 演算や，複数レコードに対して集約を行う `max` や `min`，`sum` といった集約演算などの演算子が共通して用意されている．入力データに対してそれらの演算を適用してデータ操作を行い処理のデータフローを構築する．また，DataStream API では際限のないストリームデータに対して集約演算や結合演算のような複数レコードを入力とする演算を適用するためにウィンドウ処理を提供している．

4.　統合処理基盤：JsFlow

4.1　概　　要

提案する処理基盤である JsFlow は，序論で述べた学習・実装コストの削減と適切な処理方式の割り当ての解決をするために，以下の 2 つのアプローチを採用する．(1) JSON データに対する処理の変換フローを記述することでストリーム処理方式とバッチ処理方式を統合的に扱える処理記述法の導入．(2) 処理をバッチ処理基盤およびストリーム処理基盤で実行した際の処理時間とメモリ使用量を推定し，使用可能なメモリ量に基づいた適切な処理方式の自動選択アルゴリズムの導入．

本研究では 2 つのアプローチに加え，処理を実行する処理基盤として Apache Flink [9] を組み合わせた JsFlow のプロトタイプシステムを構築した．図 4 に構築した JsFlow プロトタイプの全体像を示す．図 4 中の処理記述解析系においてユーザが記述した JSON データの処理フローに使用されている演算情報やパラメータを抽出し，続く処理方式選択機構にて抽出した情報を用いて登録された処理フローの処理方式の自動選択を行う．その後，選択された処理方式に合わせて処理フローを Flink が提供する各処理方式の API を用いたプログラムコードを生成し，Flink 上に展開して動作させる．

4.2　バッチとストリーム処理を統合的に扱う処理記述法

JsFlow が提供する処理記述法は Jaql を拡張することで実装する．3.2 節で述べた通り，Jaql は JSON 形式で保存された蓄

積データを処理するための照会言語であり，蓄積された JSON レコードに対する変換処理を filter や join， group by，transform などの提供される演算子群を適用して処理の内容を DAG として表現することで処理を記述する．

JsFlow では両処理方式を統合的に扱うために，蓄積された JSON レコードだけでなくストリームデータに対しても処理を適用可能にする必要がある．本研究ではストリームデータを有限のレコード集合としてまとめるために Jaql 照会言語の Source にウィンドウ処理の演算を追加する．際限のないストリームデータを一定時間，あるいは一定数の JSON レコード集合に区切ることで Source 以降の変換処理の演算子群を蓄積データに対する処理と同様に適用することができる．また，演算子群を適用してデータ変換したのちに再度データをストリームとして配信可能とするために，tostream 演算を追加した．また，永続的に処理結果を保存するためにファイルに追記保存する append 演算を Sink の演算として追加した．

また，ストリームデータに対するウィンドウ処理おいては JSON レコード集合を生成するタイミングを設定する必要があり，蓄積データに対しては処理を定期的に行うかアドホックに行うか指定する必要がある．したがって，window 演算では実行間隔，read 演算では実行タイミングを設定する．

表 1　JsFlow が提供する演算子群

分類	演算子名	処理内容概略
入力演算	window	ストリームデータの取得と演算範囲の指定
	read	二次記憶領域から蓄積データの取得
操作演算	filter	条件を満たさないデータの除去
	transform	射影によるフィールドの整形
	top	入力の k 個以外のレコード除去
	group by	グループ化し，集約関数を適用
	join	2 つ以上の入力を結合
	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 1 に JsFlow が提供する演算子を示す．演算は大きく別けて 3 つに分類され，入力演算，操作演算，出力演算からなる．それぞれ Jaql の Source，Operetor，Sink に該当する．入力演算ではストリームデータあるいは蓄積データから設定した実行間隔毎，あるいは実行タイミング毎にデータを読み込み，JSON レコード集合を生成する．次に操作演算では，直前の演算から送られた JSON レコード集合に対して定義された処理を行い，データを操作する．表 1 に示す通り Jaql 同様に一通りの基本演算を提供しているので，容易にデータ操作を行うことができる．最後に出力演算で，操作演算を介して得られた結果を出力する．JsFlow では各演算間を–>記号を用いて繋ぎ合わせ，入力演算，操作演算，出力演算の一連のデータフローを Task として登録を受け付ける．

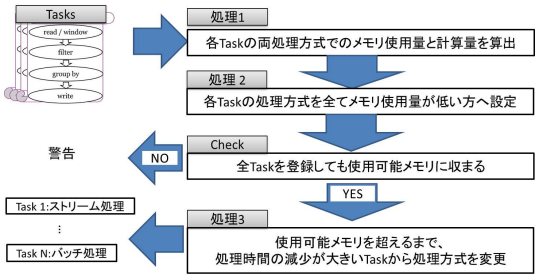

図 5　処理方式選択の流れ

4.3　処理方式の自動選択手法

JsFlow では 4.2 節で述べた処理記述法を用いて記述した処理のデータフロー (Task) を受け取りストリーム処理方式，またはバッチ処理方式を自動選択し処理を実行する．まず，4.3.1 節にて各処理方式で実行した場合の違いについて述べ，4.3.2 節にて登録された Task の処理方式の選択手法について述べる．

4.3.1　各処理方式の特徴

バッチ処理方式では，各演算は起動した後に入力されるデータ集合に対して処理を完了した後に実行を停止する． したがって，各演算は以前の処理内容を保持していないため全てのレコードに対して再計算する必要がある．一方でストリーム処理方式では，各演算は入力を待ち続け処理を永続的に継続する．そのため，入力データ集合に重複が発生する場合に各演算が前回分の演算対象の処理結果を保持し，差分があったデータに対してのみ計算することで計算量を削減して処理を高速化することができる．ただし，各演算子は処理結果の一部をオンメモリで保持しておく必要があるため，メモリ使用量と処理時間にトレードオフが生じる．

4.3.2　コストモデルに基づく処理方式の選択

4.3.1 節で述べたように各処理方式には必要とするメモリ使用量と処理時間にトレードオフの関係があるため，処理結果を高速に得るためには使用できる計算資源に応じて適切な処理方式を選択する必要がある．そこで JsFlow では登録された各 Task を各処理方式で実行した場合にかかるメモリ使用量と処理時間を推定し，使用できるメモリ量において処理時間が短くなるように各 Task の処理方式を自動的に選択する手法を導入する．処理方式の選択は図 5 に示す流れで行う．

メモリ使用量と処理時間の推定

最初に登録された各 Task について各処理方式で実行した場合にかかるメモリ使用量と処理時間を推定する．Task にかかるメモリ使用量と処理時間の推定は Task 中の各演算に必要なメモリ使用量と処理時間から算出する．バッチ処理方式では上流の演算からレコードを順次処理していくため，Task 中の演算で最も必要なメモリ使用量が Task に必要なメモリ使用量となる．したがって，Task t におけるバッチ処理方式でのメモリ使用量 $BM(t)$ は， n を Task に使用している演算子の数とし，M_i を Task 中の各演算での使用メモリ量とすると，$BM(t) = \max_{i=1}^{n}(M_i)$ で計算する．一方でストリーム処理方式では各演算が常時駆動し，以前の処理結果を保持するため，Task に必要なメモリ使用量は各演算のメモリ使用量の合計値と

表 2　演算子メモリ使用量

演算子名	出力レコードサイズ	出力レコード数	メモリ使用量 M	計算時間 C	通信時間 D
window (Time)	RS	ストリーム：$EI \times (IR)$ バッチ：$WS \times IR$	$RS \times WS \times IR$		
window (Count)	RS	ストリーム：EI バッチ：WS	$RS \times WS$		
read	RS	ストリーム：$N \times (1-OR)$ バッチ：N	$RS \times N$		
filter	RS	$N \times \sigma$		$O_{filter}(N)$	
transform	$RS \times TR$	N	$RS \times N \times TR$	$O_{transform}(N)$	
top	RS	K	$RS \times K$	$O_{top}(N)$	DE($RS \times N \div X$)
group	$RS \times TR$	$N \times AR$	$RS \times N + RS \times N \times TR \times AR$	$O_{group}(N)$	DE($RS \times K \div X$)
join	$(RS_{left} + RS_{right}) \times TR$	$(N_{left} + N_{right}) \times \sigma$	$(RS_{left}+N_{left})+(RS_{right}+N_{right})+(RS_{left} \times N_{left}+RS_{right}+N_{right}) \times TR \times \sigma$	$O_{join}(N_{left}+N_{right})$	DE($(RS_{left} \times N_{left}+RS_{right}+N_{right}) \div X$)
udf	$RS \times TR$	$N \times \sigma$	$RS \times N \times TR \times \sigma$	$O_{udf}(N)$	
tostream	RS	N	ストリーム：$RS \times N$		
write	RS	N	ストリーム：$RS \times N$		
append	RS	N	ストリーム：$RS \times N$		

表 3　記 号 一 覧

記号	定義
RS	入力 JSON レコード 1 要素のデータサイズ
WS	ウィンドウサイズ
EI	実行間隔
N	入力レコード数
K	ユーザ指定の数
X	出力バッファサイズ
σ	データの選択率
IR	ストリームデータの入力レート
AR	データの集約率
OR	データの重複率
TR	データサイズの変換率

なる．したがって，Task t におけるストリーム処理方式でのメモリ使用量 $SM(t)$ は，$SM(t) = \sum_{i=1}^{n} M_i$ で計算する．

Task の各処理方式での処理時間は Task 中の各操作演算での入力レコード全てに対して定義した処理を行う計算時間と，group by 演算のように同一のキーを持ったレコードを同一ノードにまとめるネットワーク転送にかかる通信時間の総計となる．したがって，C_i を Task 中の各操作演算にかかる計算時間，D_i を Task 中の各操作演算にかかる通信時間として，Task t におけるストリーム処理方式での処理時間 $SPT(t)$ は，$SPT(t) = \sum_{i=1}^{n}(C_i + D_i)$，同様にバッチ処理方式での処理時間 $BPT(t)$ は，$BPT(t) = \sum_{i=1}^{n}(C_i + D_i)$ で計算する．

表 2 に各演算子にかかるメモリ使用量と計算時間を示し，表 2 中の各記号は表 3 に示す

Task の割り当て手法

図 5 の処理のアルゴリズムを Algorithm 1 に示す．最初に JsFlow が使用可能なメモリ量を設定し，複数の Task の登録を受け付ける．Algorithm 1 の 3 行目から 8 行目が処理 1 に相当する．Task を各処理方式で実行した場合のメモリ使用量と処理時間を前述した方法で推定する．Algorithm 1 の 10 行目から 12 行目が処理 2 に相当する．$\max_{i=1}^{n}(M_i) \leqq \sum_{i=1}^{n} M_i$ なので，全ての Task をメモリ使用量が低いバッチ処理方式に設定する．Algorithm 1 の 14 行目から 16 行目が check 処理に相当する．全ての Task のメモリ使用量の合計が使用可能メモリ量を上回っていた場合は実行を拒否し，使用可能メモリ量を増加させるか登録する Task を減らすように警告する．Algorithm 1 の 17 行目から 28 行目が処理 3 に相当する．$|SPT(t) - BPT(t)|$ から処理時間の減少量を求め，使用可能メモリ量を超えるまで処理時間の減少量が大きい Task から処理方式を変更していくことで全ての処理方式を確定する．

Algorithm 1 処理方式選択アルゴリズム

1: 使用可能メモリ量 $maxMem$ を設定する
2: N 個の Task の登録を受け付ける
3: **for** $t \in [1, 2, ..., |N|]$ **do**
4: $BM(t) = \max_{i=1}^{n}(M_i)$
5: $BPT(t) = \sum_{i=1}^{n}(C_i + D_i)$
6: $SM(t) = \sum_{i=1}^{n} M_i$
7: $SPT(t) = \sum_{i=1}^{n}(C_i + D_i)$
8: **end for**
9: $curMemUse$=0
10: **for** $t \in [1, 2, ..., |N|]$ **do**
11: Task t をバッチ処理方式に設定
12: $curMemUse$ = $curMemUse$ + $BM(t)$
13: **end for**
14: **if** $maxMem \leqq curMemUse$ **then**
15: 実行を拒否し，使用可能メモリ量を増加させるか登録する Task を減らすように警告
16: **else**
17: **for** $t \in [1, 2, ..., |N|]$ **do**
18: $Dif(t) = |BPT(t) - SPT(t)|$
19: **end for**
20: Sort each t in the decending order of $Dif(t)$
21: **for** $t \in [1, 2, ..., |N|]$ **do**
22: **if** $maxMem > curMemUse \quad \wedge \quad Dif(t) > 0$ **then**
23: **if** $maxMem > curMemUse + SM(t) - BM(t)$ **then**
24: Task t をストリーム処理方式に設定
25: $curMemUse = curMemUse + SM(t) - BM(t)$
26: **end if**
27: **end if**
28: **end for**
29: **end if**

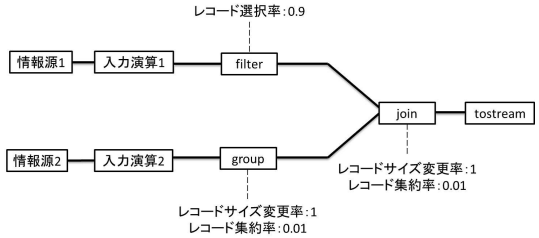

図 6　実験に用いる Task

表 4　各マシン性能

CPU	Xeon E5-2650 v3 2.3GHz
コア数	10
メモリ	16.0 GB
HDD	240 GB
OS	Ubuntu14.04

5. 評 価 実 験

5.1 実 験 概 要

4.2 節で述べた，JsFlow が提供する処理記述法を用いることで学習・実装コストを削減できているか JsFlow と Flink のコードサイズを比較して評価を行う．また，4.3.2 節で述べた処理方式の自動選択手法を用いて，適切に処理方式を選択しているか処理にかかるレイテンシの観点から評価を行う．本実験では実行環境に Flink [9] を用いた分散処理環境を構築し，4.2 節で述べた処理記述法を用いて複数の Task を登録する．本実験で用いた Task は図 6 のように 2 つの情報源と対応した入力演算子，`filter` 演算，`group by` 演算，`join` 演算，`tostream` 演算からなる DAG とした．各情報源にはストリームデータか分散配置されたファイルが割り当てられる．レコードサイズは 80 Byte，ストリームデータの入力レートは 10 KB records/s とした．各操作演算でのデータサイズの変更はなく，`filter` 演算，`group by` 演算，`join` 演算の選択率及び集約率をそれぞれ 0.9，0.01，0.01 とした．また，本実験では異なる Task において同様の処理を行う演算子の共有は考慮しないものとする．

処理方式の自動選択手法の有効性を評価するために，登録した各 Task を提案処理手法で割り当てを行った場合と，各 Task を一律でストリーム処理方式．または，一律でバッチ処理方式で行った場合での Task の実行タイミングから処理結果を返すまでのレイテンシの差で評価する． 実験ではそれぞれの処理方式に適した入力データ集合に重複がある設定が同一の Task を複数登録した場合と入力データ集合に重複がない設定が同一の Task を複数登録した場合，及び Task ごとに設定が異なる複数 Task を登録した場合の 3 通り行う．

実験環境については 2 台のマシンで分散環境を構築した．各マシンの性能と Flink の設定を表 4 と表 5 に示す．

```
{"id": Long, "key": Integer, "num": Integer, "text":
     String,}
```

Listing 1

表 5　Flink 設定

Flink Version	1.1.2
使用可能メモリ量	2.0 GB
使用 CPU 数	8
並列度	4

```
t = window("Source2", range 100 seconds ,10 seconds
    )
 -> group by $.key into {$.id, $.key, sum: sum($.
    num), $.text};
register task JsFlowSample {
     window("Source1", range 100 seconds ,10
         seconds )
 -> filter $.num > 10
 -> join t where t.key == $.key into {$.id, $.key,
    t.sum, $.text}
 -> tostream;
}
```

Listing 2

```
public class DataStreamSample{
 public static void main(String[] args) throws
     Exception {
  StreamExecutionEnvironment env =
       StreamExecutionEnvironment.
       getExecutionEnvironment();
  env.setStreamTimeCharacteristic(
       TimeCharacteristic.EventTime);
  DataStream<Tuple4<Long, Integer, Integer,
       String>> left = new SampleData.Source().
       getSource(env, 10000).
       assignTimestampsAndWatermarks(new
       TimeLagWatermarkGenerator()).filter(new
       myFilter());
  DataStream<Tuple4<Long, Integer, Integer,
       String>> right = new SampleData.Source().
       getSource(env, 10000).
       assignTimestampsAndWatermarks(new
       TimeLagWatermarkGenerator()).keyBy(1).
       window(SlidingEventTimeWindows.of(Time.
       seconds(100), Time.seconds(10))).sum(2);
  DataStream<Tuple4<Long, Integer, Integer,
       String>> mid = left.join(right)
  .where(new KeySelector<Tuple4<Long, Integer,
       Integer, String>, Integer>() {
   public Integer getKey(Tuple4<Long, Integer,
        Integer, String> t) { return t.f1; }})
   .equalTo(new KeySelector<Tuple4<Long, Integer,
        Integer, String>, Integer>() {
   public Integer getKey(Tuple4<Long, Integer,
        Integer, String> t) { return t.f1; }})
   .window(SlidingEventTimeWindows.of(Time.
       seconds(100), Time.seconds(10))).apply(new
       myJoin());
  mid.print();
  env.execute();
 }
}
```

Listing 3

5.2 プログラムコード量の比較評価

のそれぞれを用いて図 6 の Task を実装し，コード量を比較した．処理対象とするレコード構造は Listing 1 のように設定する．図 6 の Task において各情報源にストリームデータを設定

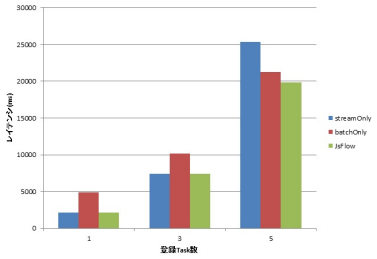

図 7　実験 1 における各方式のレイテンシ比較

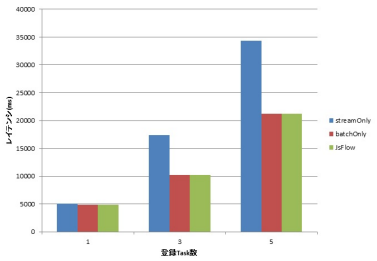

図 8　実験 2 における各方式のレイテンシ比較

し，各入力演算にウィンドウ長を 100 秒・実行間隔を 10 秒とした window 演算を設定する．

JsFlow の処理記述法を用いた例を Listing 2 に示し，Flink の DataStream API を用いた例の一部を Listing 3 に示す．

Listing 2 と Listing 3 でのコード量を比較すると，JsFlow は 8 行で処理を記述できるのに対し，Flink の DataStream API を用いる場合には主要部分だけでも 16 行，全体で 77 行のコード量が必要となる．そのため JsFlow を用いることで実装の容易化が期待できる．また，Flink では行いたい処理方式に合わせて DataSet API か DataStream API を用いる必要がある．JsFlow では入力データを全て JSON レコード集合として抽象化することで，入力データや処理方式を意識することなく，処理を JSON レコード集合に対する変換のデータフローで記述可能としている．これらの理由から JsFlow は提供する処理記述法によって学習・実装コストの削減に貢献できていると結論付ける．

5.3　実験 1：処理方式の自動選択手法の評価

5.3.1　入力データ集合に重複がある Task 群の比較

本実験では図 6 の Task において各情報源にストリームデータを設定し，各入力演算に `window` 演算を設定する．Task 中の各 `window` 演算のウィンドウ長を 10 分・実行間隔を 1 分と設定して設定が同一の Task を 1，3，5 個登録した．本実験ではこの状況下における，各 Task を一律ストリーム処理方式で行った場合と一律バッチ処理方式で行った場合，および提案処理手法で処理方式の選択を行った場合での Task のレイテンシの平均値を比較する．

実験結果を図 7 に示す．x 軸は登録した Task の数，y 軸はレイテンシを表す．この Task は 1 つの Task を登録するのにストリーム処理方式で 1.0 GB，バッチ処理方式で 0.48 GB のメモリが必要であるため，JsFlow では Task の登録数が 3 以下の時は全てストリーム処理方式で実行し，登録数が 5 個の時は 3 つの Task をストリーム処理方式，2 つの Task をバッチ処理方式で実行している．図 7 の実験結果より，登録する Task が 3 個以下の場合はメモリ量が十分にあるため差分計算を用いるストリーム処理方式で行った方がバッチ処理方式よりレイテンシが 2〜3 倍程度抑えられているが，登録する Task が 5 個の時は，ストリーム処理方式ではメモリ量が足りずにバッチ処理方式よりレイテンシが 1.2 倍程大きくかかっている．JsFlow ではメモリ量が十分ある場合は全てストリーム処理方式で行い，不足した場合はバッチ処理方式を用いてメモリ使用量を使用可能なメモリ量に抑えているため，レイテンシが短い処理方式と同等かそれ以下のレイテンシで実行できていることが分かる．

5.3.2　実験 2：入力データ集合に重複がない Task 群の比較

本実験でも 5.3.1 節と同様に図 6 の Task において各情報源にストリームデータを設定し，各入力演算に `window` 演算を設定する．ただし，Task 中の各 `window` 演算のウィンドウ長を 10 分・実行間隔を 10 分と設定して入力データ集合の重複を排除している．設定が同一の Task を 1，3，5 個登録し，各 Task を一律ストリーム処理方式で行った場合と一律バッチ処理方式で行った場合，及び提案処理手法で処理方式の選択を行った場合での Task のレイテンシの平均値を比較する．

実験結果を図 8 に示す．図 8 ではウィンドウ長を 10 分・実行間隔を 10 分に設定した場合での各方式のレイテンシを比較している．入力データ集合の重複がないので JsFlow では全ての Task をメモリ使用量が少ないバッチ処理方式で実行を行っている．図 8 の実験結果より，登録する Task が 1 つの場合はレイテンシはほとんど同一であるが，登録する Task が 3 つ以上の場合はストリーム処理方式を行うのに十分なメモリを確保できずにバッチ処理方式に比べてレイテンシが 1.5 倍程度大きくかかっている．JsFlow ではバッチ処理方式に Task を振り分けており，適切な処理方式を選択できている．

5.3.3　実験 3：入力データ設定が異なる Task 群の比較

本実験では図 6 の Task の入力演算子を表 6 ように設定した計 8 個の Task を登録する．情報源 1 にはストリームデータを設定し，入力演算 1 には全て `window` 演算を用いている．ウィンドウ長を 10 分あるいは 100 秒に設定し，入力演算 1 における Task 1〜6 の実行間隔を 1 分あるいは 10 秒とし，Task 7,8 の実行間隔をその Task のウィンドウ長と同値とした．情報源 2 には各マシンに分散配置したファイルかストリームデータを設定し，Task 1〜4 の入力演算 2 には `read` 演算，Task 5〜8 の入力演算 2 には `window` 演算を用いる．Task 1,2 で読み込むファイル中のレコード数を 100K，Task 3,4 で読み込むファイル中のレコード数を 6M で用意し，Task 5,6 でのウィンドウ長を 10 分，Task 7,8 でのウィンドウ長を 100 秒で設定した．入力演算 2 における Task 1〜4 の実行タイミングは登録時とし，Task 5,6 での実行間隔を 1 分，Task 7,8 での実行間隔を 10 秒で設定した．なお，一方の入力演算が実行条件を満たした場合には，もう一方の入力演算も実行される．

各 Task を一律ストリーム処理方式で行った場合と一律バッチ処理方式で行った場合，及び提案処理手法で処理方式の選択を行った場合での Task のレイテンシの平均値を比較する．

表 6　実験 3 の各 Task の入力演算子設定

Task 数	入力演算子 1	ウィンドウ長 1	実行間隔 1	入力演算子 2	レコード数 2 / ウィンドウ長 2	実行間隔 2 / 実行タイミング
Task 1	window	10 minutes	1 minute	read	100K records	once
Task 2	window	100 seconds	10 seconds	read	100K records	once
Task 3	window	10 minutes	1 minute	read	6M records	once
Task 4	window	100 seconds	10 seconds	read	6M records	once
Task 5	window	10 minutes	1 minute	window	10 minutes	1 minute
Task 6	window	100 seconds	10 seconds	window	10 minutes	1 minute
Task 7	window	10 minutes	10 minutes	window	100 seconds	10 seconds
Task 8	window	100 seconds	100 seconds	window	100 seconds	10 seconds

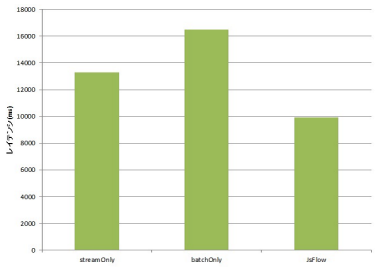

図 9　実験 3 の Task 群の各方式でのレイテンシ比較

実験結果を図 9 に示す．JsFlow の処理方式の割り当てにおいて全てバッチ処理方式で行った場合には 3.0 GB のメモリを必要とし，全てストリーム処理方式で行う場合は 6.2 GB のメモリを必要とする．そのため使用可能メモリ量である 4.0 GB になるまで推定処理時間の減少が大きい Task から処理方式を変更し，Task 1，2，5，6，8 はストリーム処理方式で実行し Task 3，4，7 はバッチ処理方式で実行した．図 9 の実験結果が示すように, JsFlow を用いて Task の割り当てを行った場合にレイテンシが一番抑えられている．これは， 全ての Task をバッチ処理方式で行った場合は差分計算を用いないために入力データの重複部分の再計算によってストリーム処理方式より大きく遅れており，全ての Task をストリーム処理方式で行った場合は十分なメモリ量が確保できていないため，処理の途中でディスク I/O による遅延が発生しているからだと考えられる．以上より，JsFlow による処理の割り当ては有効であると考えられる．

6. ま と め

本研究では，ストリーム処理方式とバッチ処理方式を統合的に実行可能な処理基盤 JsFlow の開発を行った．本稿では，JSON を用いてストリームデータと蓄積データを JSON レコード集合として抽象化し，処理を JSON レコード集合に対する変換のデータフローで記述可能にすることで，両処理方式を統一的に扱えるプログラミングモデルを提供した．また，各処理方式で実行するのに必要なメモリ使用量と計算量を推定し，使用可能なメモリ量に応じて処理時間が短くなるように処理記述の処理方式を自動的に選択するアルゴリズムを提案した．これにより，大規模データ処理処理を簡潔なプログラミングモデルを用いて容易に実行することを可能にした．評価実験により，学習・実装コストの課題の削減と処理の振り分けの有効性を示した．

今後の課題として処理方式の動的変更が考えられる．本研究で提供する処理基盤が行っている処理方式の割り当ては登録時に一度だけ行い，一度割り当てられた処理方式を動的に変更することは対象としていない．しかし，実際に処理の運用を行う際には，処理を新規に追加する場合や処理の情報源となるストリームデータの入力レートが変わることで，割り当てられた処理方式を変更した方が効率良く処理を行える可能性がある．そのため，処理が追加されたタイミングや処理基盤の起動後からの経過時間に応じて処理方式を再度割り当てることに検討の余地がある．

7. 謝　　辞

本研究の一部は，科研費・基盤研究 B(26280037)，及び国立研究開発法人情報通信研究機構の委託研究 (APD28151) の助成を受けたものである．

文　　献

[1] K. Shvachko, H. Kuang, S. Radia, R Chansler. The Hadoop Distributed File System. In MSST, pages 1-10, 2010.

[2] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In HotCloud, pages 10-10, 2010.

[3] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm at Twitter. In SIGMOD, 2014.

[4] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In USENIX, pages 10-10. 2012.

[5] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoc. Foundations of JSON Schema. In WWW, pages 263-273, 2016.

[6] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernandez-Moctezuma, R. Lax, S. McVeety, D. Mills, ´ F. Perry, E. Schmidt, 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.

[7] 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.

[8] K.S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Eltabakh, C.-C. Kanne, F. Ozcan, E.J. Shekita. Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. In VLDB, pages. 1272-1283, 2011.

[9] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, Apache flink: Stream and batch processing in a single engine, In Data Engineering, pages 28-38, 2015.

[10] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel data processing with mapreduce: A survey. In SIGMOD, pages 11-20, 2011.

[11] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernandez-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, 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, pages 1792-1803, 2015.