Load Shedding による近似計算を補完するデータストリーム処理システム

岡田 瑞穂† 鈴村 豊太郎†‡

†東京工業大学 〒152-8550 東京都目黒区大岡山 2-12-1
‡IBM 東京基礎研究所 〒135-8511 東京都江東区豊洲 5-6-52
E-mail: †okada.m.af@m.titech.ac.jp, ‡suzumura@cs.titech.ac.jp

あらまし データストリーム処理においては，Twitter ストリームに対するリアルタイム自然言語処理システムのように，大容量のストリームデータを低レイテンシで処理することが求められるようなアプリケーションがある．しかし，例えば Twitter においてツイートの量が一時的に増大するような時，入力データレートがシステムの処理能力を超えてしまい処理レイテンシが増加すると，アプリケーションのサービスレベル・アグリーメントを満たせない場合がある．そのような過負荷時に入力データの一部を削除する Load Shedding という手法があるが，レイテンシを確保する代わりに計算精度は落ちてしまい，後に別のアプリケーションで同じ計算結果を利用したい時に問題となってしまう．我々は Load Shedding によって削除されるデータと不完全な計算結果の両方をストレージに保持し，システムの処理能力に余裕があるときに前者のデータを再度読み込んで処理を施し，後者の値と集約することによって計算結果を補完する処理機構を提案する．

キーワード データストリーム処理，スケジューリング，System S，Twitter，自然言語処理

Data Stream Management System for Complementation of Approximate Calculation by Load Shedding

Mizuho OKADA† and Toyotaro SUZUMURA†‡

†Tokyo Institute of Technology 2-12-1 Ookayama, Meguro-ku, Tokyo, Japan
‡IBM Research – Tokyo 5-6-52 Toyosu, Koutou-ku, Tokyo, Japan
E-mail: †okada.m.af@titech.ac.jp, ‡suzumura@cs.titech.ac.jp

Abstract There are some applications that needs low latency to process big stream data such as real time natural language processing system on Twitter timeline. However in case that the amount of tweets on Twitter becomes very big temporarily, application can't satisfy its Service Level Agreement because the input data rate exceeds the capacity of the system. Load Shedding is a technique that sheds a part of the data in the overload situation to calculate approximate value, but there is a problem when other application later needs the result of calculation. We propose the processing mechanism that stores the shed data on storage to calculate full value later by aggregating it with the approximate value by Load Shedding.

Keyword Data Stream Processing, Scheduling, System S, Twitter, Natural Language Processing

1. はじめに

近年，工場のセンサーやインターネット等から逐次到着する大量のストリームデータをリアルタイムに高速処理するデータストリーム処理の需要が高まっている．

データストリーム処理を利用したアプリケーションでは，データが到着してから計算結果を得るまでのレイテンシをなるべく低く抑えることが求められ，リアルタイム性を保つことがサービスが成り立つための条件になる．サービスレベル・アグリーメント（Service Level Agreement，SLA）とは，そのようにアプリケーションがサービスとして満たすべき品質要件を定めたものである．しかし現実世界ではストリームデータの到着間隔は一定ではなく，時としてシステムの処理性能を超えるほど高頻度になることも考えられ，その状態はデータバーストと呼ばれる．そのような場合，アプリケーションは要求されるレイテンシ以内に処理を終えることができないオーバーロード状態に陥り，SLA を満たせない場合がある．オーバーロード時にリアルタイム性を確保するための既存手法としては Load Shedding[1]がある．Load Shedding では到着データの一部を削除して残りのデータのみでウィンドウ演

算を行う．ウィンドウ演算とは，データストリーム処理において永続的に到着するデータを一定単位で分割して処理する手法である．

例えば 1 分間のウィンドウに蓄積されたデータの平均値を求める場合は，全てのデータがなくても計算自体を行うことはできる．しかし Load Shedding では，リアルタイム性を確保する代わりにデータを削除したことによって本来の値とは異なる値が出力される．

さて，データストリーム処理によるアプリケーションの計算結果を一定時間後に別のアプリケーションで再利用したい場合がある．後者のアプリケーションがリアルタイム性よりもデータの完全性を重視するとき，システムの負荷状況を見ながら Load Shedding による不完全な計算結果を本来の計算結果に置き換える処理を行うことが考えられる．

我々は以上のような状況に対応する新しい処理機構を提案する．まずリアルタイム性が求められるシステムにおいて，オーバーロード時に Load Shedding を用いてリアルタイムに計算結果を出力するが，その際削除されるべきデータをストレージに保持する．そして，処理能力に余裕がある非オーバーロード時にストレージに保持したデータを処理し，先ほど求めた近似値と集約することで全てのデータを用いた完全な計算結果を出力する．これにより，データの完全性が求められるアプリケーションも問題なく計算結果を利用することができる．評価実験では提案手法を DSMS のひとつである IBM System S[7][8]にて実装し，Twitter タイムラインに対する感情分析システム[3][6]をアプリケーションとして搭載した．

以降の節では，第 2 章でデータストリーム処理と評価実験で使用した処理系である System S について述べ，第 3 章でデータバーストと既存手法である Load Shedding について述べた後，第 4 章で問題定義を行う．第 5 章で提案手法について説明した後，第 6 章でその実装，第 7 章で評価を行う．第 8 章にて関連研究を挙げ，最後に第 9 章でまとめと今後の課題について述べる．

2. データストリーム処理と System S

2.1 データストリーム処理

データストリーム処理とは，インターネット等から止め処なく到着するストリームデータを計算対象とし，これをストレージに蓄積することなくオンメモリで逐次処理していくという計算パラダイムである．計算対象を全てストレージに蓄積してから計算するバッチ処理とは違い，リアルタイムの応答が要求される場合や，全データの蓄積が物理的に困難な処理に適している．このような手法は音声や動画のストリーミングなど一部の処理では利用されていたが，データストリーム処理はこれを抽象・汎用化し，幅広い処理に対して適用できるよう洗練された処理系としてまとめられている点が従来とは異なっている．このような処理系を DSMS / DSPS (Data Stream Management / Processing System)と呼ぶが，その一例として MIT の Borealis や IBM Research の System S[7][8]などが存在し，ここ数年活発な研究がなされている．多くの DSMS がシングルノードでの実行を前提としているが，Borealis と System S は分散環境上で実行可能である．

2.2 System S と SPADE

System S[7][8]は，データフロー図から直感的に処理を記述できる SPADE という言語と，自動性能最適化機構を持つ SPADE コンパイラ，処理基盤である SPC によって構成される．SPADE は高級な宣言的言語で，処理対象であるストリームと，処理を行うオペレータの関係をデータフローとして記述するだけでデータストリーム処理を定義でき，ノード間やプロセス間の通信や，デーモンの立ち上げなどを意識することなくプログラミングが可能である．

SPADE では多様な組み込みオペレータのほかにユーザ定義オペレータ UDOP（User Defined Operator）を持ち，複雑なデータストリーム処理に対応することが可能となっている．UDOP は実際の処理以外の通信やストリームの管理部分が書かれたスケルトンコードを自動生成するため，ユーザは処理部分のみを C++や Java で記述すればよい．これにより，汎用オペレータでは表現できない複雑な処理や操作を実現でき，UDOP 自体も他のオペレータと同様にモジュール化されるため高度な柔軟性，再利用性の恩恵を受けることができる．SPADE のオペレータの詳細については[]に詳細が記載されているため省略するが，SPADE は簡易な記述と高い柔軟性を兼ね備えており効率的なシステム開発が可能となっている．

また，SPADE の組み込みオペレータの一つに Aggregate オペレータがあり，これは DSMS におけるウィンドウ演算を可能とするものである．ウィンドウ演算とは，永続的に到着する無限長のデータを一定間隔で区切ってウィンドウと呼ばれる単位にまとめ，計算を行う．株価の一定時間ごとの値動きや，平均を求めるときなどに使われる．ウィンドウ演算に用いられるウィンドウの種類としては，一定時間で区切る時間ベースウィンドウ（Time-based Window），データが到着した個数で区切るタプルベースウィンドウ（Tuple-based Window）などがある．

3. データバーストと Load Shedding

1 章で述べたように，実世界のストリーミングデー

タは到着間隔が一定ではない．例えば近年有用なストリームデータとして研究が盛んな Twitter は，2012 年に 1 日あたりの投稿数が 4 億件を超えたと発表した[4]．つまり 1 秒間あたり平均 4600 件のつぶやきが投稿されていることになる．また，2013 年に日本が新年を迎えた際の 1 秒間あたりの投稿数は最高で 33,000 件にも上り，過去最高を記録した[5]．このようなデータバースト時には，全てのデータを処理しようとするとシステムはオーバーロード状態に陥り，リアルタイム性が保てなくなってしまう．

このようなオーバーロード時にリアルタイム性を確保する手法として，Load Shedding[1]がある．Load Shedding とは，DSMS において，システムがオーバーロードを検出したときに，流れてくるデータの一部を削除して計算を行い，リアルタイム性を確保する技術である．ウィンドウ演算では，アプリケーションによっては必ずしも全てのデータが必要でない場合がある．例えばウィンドウ内のデータの平均値や総和を求めるときは，全てのデータがなくても演算を行うこと自体は可能である．しかしデータを削除したことにより，本来の計算結果との間に誤差が生じる．一般に，削除したデータが多いほど誤差は大きくなる．Load Shedding ではシステムの処理性能に収まる程度までデータを確率的に削除する方法が最も一般的な手法である．

4. 問題定義

さて，あるアプリケーションの計算結果を，後に別のアプリケーションに再利用したい場合がある．前者を（ア），後者を（イ）とし，以下にその例を挙げる．

(ア) リアルタイム性を重視するアプリケーション

Twitter のツイートを自然言語処理で分析し，ユーザが表す感情（ポジティブ or ネガティブ or ニュートラル）の値を計算するシステム[3][6]がある．1 秒間に投稿された全てのツイートをまとめ，感情を表す値を計算して 1 秒ごとに Web サイトに表示するサービスが考えられる．Web サイトを訪問したユーザに確実に最新の結果を素早く見せたいため，このサービスで最も重要なのはリアルタイム性である．計算結果は近似値を表示しても致命的にはならないため，オーバーロード時には Load Shedding によって一部のデータを削除することによる高速化が測れる．

(イ) データの完全性を重視するアプリケーション

一方で，リアルタイム性よりもデータの完全性を重視するアプリケーションがある． Twitter における例としては，株価の値動きの予測[9]である．文献[9]によると，Twitter のつぶやきは社会全体の気分を良く反映しており，それらが表す感情値は株式市場の予測まで可能であるという．この文献は，Twitter のストリームデータに対して 1 日のウィンドウを設定し，ウィンドウ内のツイートを自然言語処理で分析して求めた全体の感情値が，3 日後の米ダウ平均株価を良く予想するというものである．この研究分野では，ツイートデータが投稿されてから処理結果を利用するまでの時間が比較的長い．リアルタイム性が厳しく求められない代わりに，データの完全性に対する要求が高く，ある程度時間をかけても全てのデータを確実に処理しておきたい．

以上の（ア）（イ）において，ストリームデータに対して施したい処理が一部共通しているとき，(ア）の計算結果を後に（イ）のアプリケーションで再利用することは合理的である．しかしオーバーロードが検出されたとき，（ア）の SLA を保持するために Load Shedding を利用すると，その計算結果は完全ではないため，（イ）で再利用する際に問題が生じる．

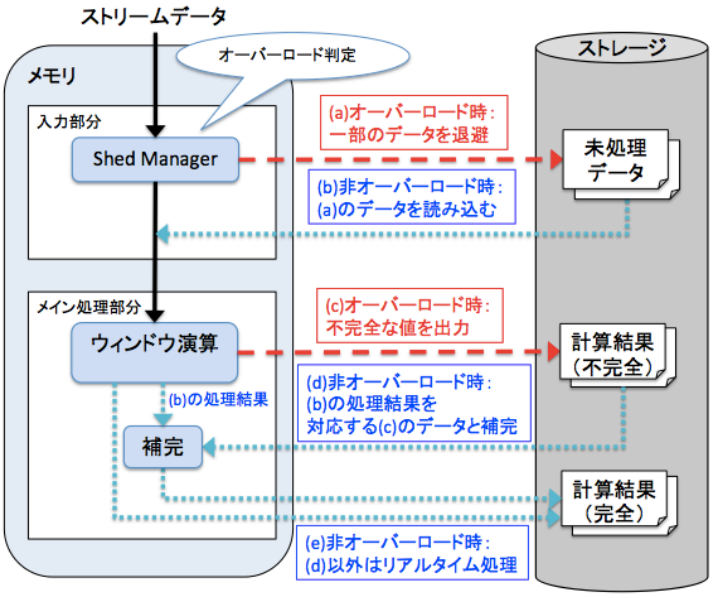

図 1 提案手法の概要

5. 提案手法

本稿では，第 4 章で述べた問題点を解決するための処理機構を提案する．具体的には，オーバーロード時に Load Shedding により削除されるデータを一時的にストレージに保持し，ひとまずは不完全なデータでリアルタイムに計算結果を出力することにより，第 4 章（ア）のシステムの SLA を保持する．後にオーバーロード状態が収まったら，ストレージに退避させたデータを読み込んで処理し，先ほど出力した計算結果と集約して完全な計算結果を得る．この手法により，あとで（イ）のアプリケーションが（ア）の計算結果を利用したいときに，完全な計算結果を参照することが可能となる．

図 1. の提案手法の概要図を用いて詳細を説明する. Shed Manager は我々が実装した DSMS のオペレータで, 入力データの到着間隔に応じてオーバーロード状態か否かの判定を行い, 状況に応じて複数のストリームにデータを送信する役割を担う. 以降, システムの挙動をオーバーロード時, 非オーバーロード時の二通りの状態に分けて説明する. 以下で述べる(a)~(e)は, 図 1. の(a)~(e)にそれぞれ対応している.

5.1.オーバーロード時

(a) システムの処理能力を超える分のデータをストレージに書き込む. このとき, データをリアルタイムに処理するかストレージに書き込むかの判定は確率的に行う. 実際の実装ではストレージに書き込むのは Shed Manager とは別のプロセスであるため, Shed Manager は単にデータを送信する役割を果たす.

(c) 全入力データのうち, (a)によって間引かれたデータ以外のデータを用いてウィンドウ演算を行い, ファイルに出力する. これは不完全な計算結果であり, 精度は低い.

5.2.非オーバーロード時

(b) (a)によってストレージに退避されたデータを連続的に読み込む. このとき, システムの処理能力を使い切る程度の頻度で読み込むと効率が良い.

(d) (b)のデータを処理し, (c)の不完全な計算結果と集約することにより, 計算精度を補完する.

(e) ネットワークから到着するデータを逐次処理し, Load Shedding が適用されない通常のデータストリーム処理を行う.

5.3.適用範囲

この手法は, 結合法則が成り立つようなウィンドウ演算のもとで効力を発揮する. 結合法則とは, 集合 S に二項演算*が定義されているとき, S の任意の元 a, b, c に関して以下の式が成り立つことである.

$$(a * b) * c = a * (b * c)$$

つまり, Load Shedding によって 2 分されたデータ集合において, それぞれに対する演算結果同士を演算した結果が, 完全なデータに対して演算を施した結果と等しくなるような状況で初めて, 計算精度が補完されるのである. 例としては, データの総和を求める演算がある. 逆に, 計算結果がデータの処理順序に依存するような場合は適用することができない.

6. 実装

System S と SPADE を用いて提案手法を実装した. 提案するシステムに載せるアプリケーションとして, Twitter タイムラインの感情分類器を採用した. 時間ベースウィンドウを設け, ウィンドウごとに分類結果を求める.

6.1 テキスト分類と単純ベイズ分類器

テキスト分類とは, 文書を予め与えられたいくつかのクラスに自動分類する技術である. 今回採用した単純ベイズ分類器(Naive Bayes Classifier)は最も標準的なテキスト分類器の一つで, Twitter に適用した研究も多くあり[3][6], 高精度に加え実装も簡単である.

単純ベイズ分類器が, 5.3.で述べたような提案手法の適用範囲にあることを述べる. 単純ベイズ分類器は, 特徴変数$f_1, f_2, \cdots f_n$から成る文書 doc が与えられた際に, それが属するクラス class を以下の式で出力する.

$$class = argmax_{class}\, p(class|doc)$$

ベイズの定理より,

$$p(class|doc) = \frac{p(doc|class)p(class)}{p(doc)}$$

分母の$p(doc)$はクラスに依存しないため無視できるので,

$$p(class|doc) \propto p(class)p(doc|class)$$

ここで文書 doc において特徴変数$f_1, f_2, \cdots f_n$の独立性を仮定すると,

$$p(doc|class) = \prod_{i=1}^{n} p(f_i|class)$$

となり, 結局

$$class = argmax_{class}\, p\,(class) \prod_{i=1}^{n} p(f_i|class)$$

が導き出される. 実際にはアンダーフローを防ぐために対数をとる.

$$class = argmax_{class} \left(\log\, p\,(class) + \sum_{i=1}^{n} p(f_i|class) \right)$$

これは総和の形になっているため結合法則が成り立ち, 5.3.より提案手法が適用できる.

今回は文献[3][6]を参考にして, Wikipedia の顔文字リスト[10]をクエリに Positive, Negative, Neutral の 3 つのクラスについてそれぞれ 30 万件ずつの訓練データを集めた. 特徴変数は Bigram を採用した.

6.2 データフロー図

次図 2 に System S のデータフロー図を示す. 図中の丸角の四角はオペレータ（プロセス）を表し, 矢印はデータの流れ（データストリーム）を表す. 実装に用いたオペレータは, Source（データの入力）, Sink（データの出力）, Split（データの分割）, Barrier（ストリームの同期）, Aggregate（ウィンドウ演算）の 5 つの組み込みオペレータと, 6 種類の UDOP（ユーザ定義オペレータ）である. また, 必要に応じて柔軟にファ

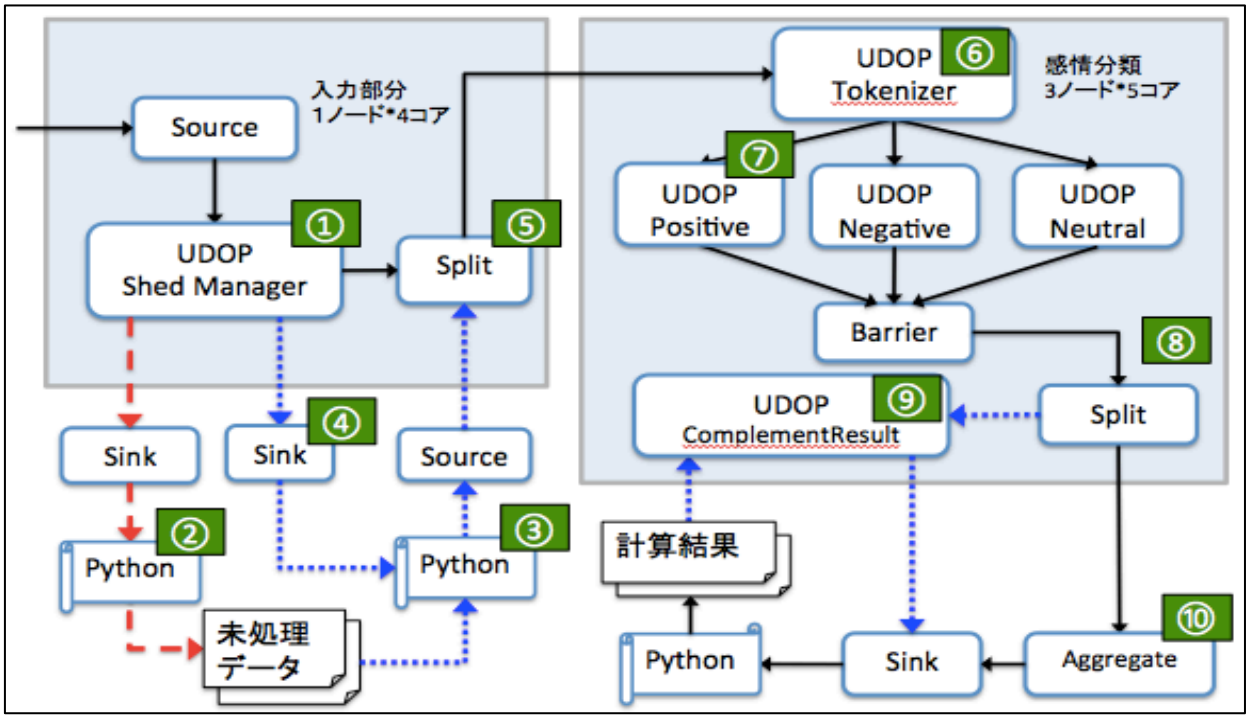

図 2 データフロー図

イルを読み書きするために Python コードを用いている．

図中の各オペレータと Python の動作について説明する．まず，Source オペレータから読まれたストリームデータは①Shed Manager に流れ込む．5 章で述べた通り，Shed Manager はオーバーロード判定とデータの分割送信を行う．今回におけるオーバーロードの定義としては，あらかじめシステムの最大スループットを計測しておき，それを上回る頻度でデータが到着したらオーバーロードを検出することとする．オーバーロード判定は 10 秒ごとに行う．また，データがどのウィンドウに属するかを示すウィンドウ id をデータに付与する．これにより，後に計算結果を補完する場合に未処理データに対応する計算済データを特定できる．②はオーバーロード時のデータの流れであり，図 1．の（a）に該当する．今回実験で使用した System S バージョン 1.2 では，複数ファイルにデータを書き込むためのオペレータが用意されていなかったので，Sink オペレータの TCP 通信により Python に処理を任せる．これによってストレージに保存された未処理データは，非オーバーロード時に③の Python によって System S に送信される．④の Sink オペレータは③の Python に対して，現在オーバーロード状態であるかどうかを示す情報を送っている．⑤の Split オペレータは，ノード間通信のためにストリームを分割する．今回の実験では入力部分が 1 ノード*4 コアなのに対して，アプリケーションの処理である感情分類の部分は 3 ノード*5 コアで処理している．Split オペレータはラウンドロビン方式で入力ストリームを各コアに送信する役割を担う．⑥の UDOP では，テキストデータに対して自然言語処理を行う．文献[3]を参考に，以下の操作を行う．

- URL，@ユーザ名，RT，#ハッシュタグ等の記号を取り除く
- 句読点や空白記号を用いて単語に分割(Tokenize)する．
- Bigram を作る．Bigram とは連続する 2 単語を一つの特徴変数として扱うことである．この際，not は前後の単語と繋げる．例えば”I do not like fish”という文は，bigram にすると”I do+not”，”do+not like”，”not+line fish”になる．

⑦で並列に並んでいる 3 つの UDOP は，それぞれ Positive，Negative，Neutral の訓練データを 30 万件ずつ保持しており，入力テキストがそれぞれのクラスに属する確率を 6.1 章で示した方法で計算する．そして，⑧の Split オペレータでは，流れてきたデータが，ストレージから読み出されたデータなのか，リアルタイムに処理されてきたデータなのかによって行き先を分岐させる．その判定のための Boolean 型のデータは予め持たせてある．前者の場合，データは⑨の UDOP に送られる．⑨は図 1. における補完モジュールに対応し，入力データのウィンドウ id と同じファイルを読み込み，2 つの計算結果の和をとって Sink オペレータに出力する．一方，リアルタイムに処理されてきたデータは⑧によって⑩へ送られる．Aggregate はウィンドウ演算を可能にするオペレータで，今回はウィンドウ幅を 1 秒に設定して 25 分間実験を行った．

7. 評価

この章では前章で述べた実装に対する評価を行う．データの流れとアプリケーションの精度の変化の 2 点に関して評価する．

7.1 実験環境

測定にはノードを 4 台使用した．環境は全ノード共通で，CPU Intel Xeon（2.93GH，12 コア）Mem 48GB，ソフトウェア構成は CentOS 5.6 2.6.18-238.el5 AMD64，gcc 4.1.2，python3.3.0 を用いる．そのうち 1 台のノードに図 2 の入力部分のオペレータを配置し，残りの 4 台に感情分類器のオペレータを配置する．

7.2 実験時のパラメータ

入力データは，Source オペレータに対して Python が通信で流し込む．この際，故意に大量のデータを送ろうとしても，System S は接続をブロックしてしまい，オーバーロードの状況を作れなかった．そこで今回は，システムの処理能力を実際よりも低く仮定し，本来は処理しきれる程度のデータバーストをオーバーロードと仮定することにより実験を行う．事前の検証により，我々のシステムでは最大秒間 23000 件程度の入力を処理しきれることがわかった．そこで，システムの処理能力の限界を 15000 件と仮定して実験を行う．

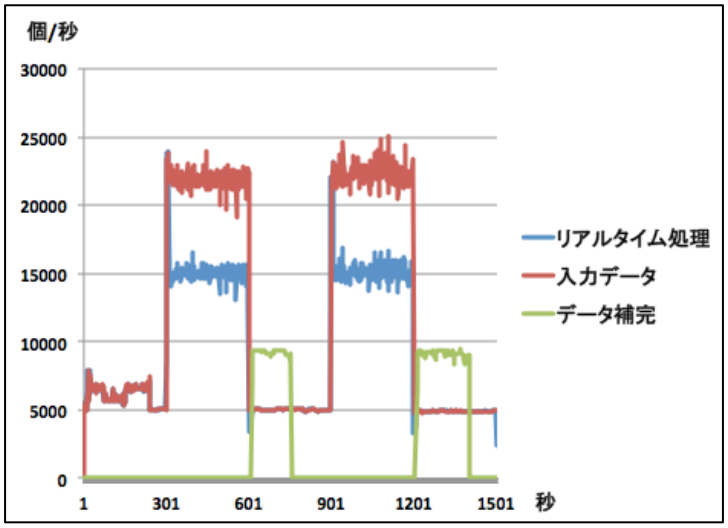

図 3 データの流れ

7.3 データの流れ

図 3 に，実験における各ストリームのデータレートを示した．実験は 25 分行った．図 3 の赤いグラフがシステムに入力されるデータである．入力データは，5 分ごとにデータバーストと非データバーストの状態を繰り返す．青いグラフは入力データに対するリアルタイム処理の速さの動きである．この実験ではシステムの処理限界を秒間 15000 と定めたため，データバースト時には秒間 15000 を超えた分のデータは Shed Manager によりストレージに保持されリアルタイムには処理されない．最後に緑のグラフは，提案手法によってディスクに保持されたデータの流れであり，非データバースト時にシステムに読み込まれるときの速さを示す．このデータが読まれる場合，リアルタイム処理の処理速度とデータ補完の処理速度を足したものがほぼ処理限界の 15000 なっていることがわかるだろう．

7.4 アプリケーションの精度の変化

この章では，6.2 章で実装した単純ベイズ分類器の精度に関する評価を行う．

表 1：分類器の精度（3 クラス）

Positive	Negative	Neutral
90.0%	86.7%	2.5%

表 2：分類器の精度（2 クラス）

Positive	Negative
93.3%	90.0%

7.4.1 単純ベイズ分類器の精度

まず，実装した分類器の精度を評価する．テストデータは各クラス 30 件ずつ人手でラベル付けを行い，ラベル通りに分類できた割合を各クラスごとに測った．表 1 に 3 クラスにおける実験結果を載せた．Positive，Negative ではうまく分類できている一方で Neutral の精度が極めて低い結果となったが，これは一般に Neutral という感情の定義，分類がとても難しいことに起因する．そこで Neutral を抜いた 2 クラスで評価を行ったところ，表 2 のような結果になった．

7.4.2 データを落とした場合の精度の変化

7.4.1 章の分類精度を踏まえた上で，ウィンドウ演算を行った時に一部のデータを削除した場合に分類結果がどの程度変化するのかを検証する．対象となるウィンドウは 7.3 章で示したデータのうちデータバーストに該当する全 600 個のウィンドウで，ウィンドウ内の平均データ数は 23240 個である．それぞれのウィンドウにおいて，全てのデータを用いた場合の分類結果を正解とみなし，一部のデータを削除した場合の分類結果と比較する．同じクラスに分類できていたら正解，そうでなければ不正解とし，600 個のウィンドウそれぞれにおいて正誤を確かめ正解率を計算する．その際，データを削除する割合を様々な値に変えることで，削除割合と分類結果との関係を調べる．3 クラス，2 クラスの場合の結果をそれぞれ表 3，表 4 に示す．削除率を上げると正解率が下がる様子が確認できた．つまり，ストレージに退避させるデータの割合が多いほど本来の計算結果から乖離していくので，その分提案手法によって計算結果を補完する意義が高まる．

一方で，今回の結果においては正解率についてそれほど大きな違いが現れなかったことも確認できた．つまりデータのどの部分を抜き出してもその特徴はほぼ変わらなかった．この結果の原因としては，データ数がかなり多いことや，そもそもデータ全体の特徴とし

表 3：データの削除率と正解率（3 クラス）

削除率（%）	0	10	20	30	40	50	60	70	80	90
正解率（%）	100	99.5	99.0	98.0	97.0	96.7	96.5	95.0	91.3	89.0

表 4：データの削除率と正解率（2 クラス）

削除率（%）	0	10	20	30	40	50	60	70	80	90
正解率（%）	100	99.6	99.1	97.8	97.6	96.8	96.5	93.0	92.3	85.7

て各感情クラスの占める比率が決まっていることが考えられる．

8. 関連研究

Barzan ら[12]は Load Shedding を用いながらオペレータの分割を行なっている．しかしこれは，削除したデータを保持して再度処理する機構については触れていない．

また，データバースト時にクラウド環境に処理を委譲することで，レイテンシを抑える手法[11]がある．この手法は全てのデータを間引かずに処理できるが，経済的なコストが発生してしまうことにより，本研究とは趣が異なると言える．

Nishii ら[13]は音声認識をストリーム処理に適用した手法を提案しており，入力データレートに応じてビーム幅と呼ばれるパラメータを変更することで，本手法と同様に精度を下げつつスループットを確保する手法である．

9. まとめと今後の課題

本稿の貢献として，システムがオーバーロード時に Load Shedding によって本来削除されていたデータをストレージに保持し，処理に余裕があるときに再度読み込むことで完全なデータによる計算結果を補完する新しい処理機構を提案した．

今後の展望として，アプリケーションの適用範囲についてより深く考える必要がある．今回の実験で使用したデータの流れはごく単純なものであったが，様々なデータのパターンに関して提案手法がどのような振る舞いを見せるかを調べる必要があると共に，提案手法が最も役に立つ状況，さらにその逆の条件について明らかにしたい．加えて，搭載するアプリケーションの精度に関して，より深い考察をすることも今後の課題である．

謝辞

本研究の一部は科学研究費補助金・挑戦的萌芽研究（課題番号:22650017） の助成によって行われた．

参 考 文 献

[1] N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack and M. Stonebraker. Load Shedding in a Data Stream Manager. In Proc. VLDB, 2003.

[2] Babcock, B., Babu, S., Datar, M., Motwani, R. and Thomas., D, Operator scheduling in data stream systems, The VLDB Journal, Vol.13, pp.333-353, 2004.

[3] A. Pak, P. Paroubek, Twitter as a corpus for sentiment analysis and opinion mining, Proceedings of the Seventh conference on International Language Resources and Evaluation (LREC'10), European Language Resources Association (ELRA), 2010, pp. 19–21.

[4] https://twitter.com/TwitterAds/statuses/21086778236 1948161?tw_i=21086778236194816 1&tw_e=details &tw_p=tweetembed

[5] http://www.itmedia.co.jp/news/articles/1301/04/news 014.html

[6] Alec Go, Lei Huang, and Richa Bhayani. 2009. Twitter sentiment analysis. Final Projects from CS224N for Spring 2008/2009 at The Stanford Natural Language Processing Group.

[7] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, K.-L. Wu and L. Fleischer, "SODA: An Optimizing Scheduler for Large-Scale Stream-Based Distributed Computer Systems", Middleware, 2008.

[8] B. Gedik, H. Andrade and K.-L. Wu, "A Code Generation Approach to Optimizing High-Performance Distributed Data Stream Processing", In Proc. USENIX, pages 847-856, 2009.

[9] Bollen, J.; Mao, H.; and Zeng, X.-J. 2010. Twitter mood predicts the stock market. Journal of Computational Science 2(1):1–8.

[10] http://en.wikipedia.org/wiki/List_of_emoticons

[11] Atsushi Ishii and Toyotaro Suzumura, "Elastic Stream Computing with Cloud", IEEE CLOUD 2011 (The 4th International Conference on Cloud Computing), Research Track

[12] Barzan Mozafari, et al., Optimal Load Shedding with Aggregates and Mining, ICDE 2010

[13] Shunsuke Nishii and Toyotaro Suzumura, Highly Scalable Speech Recognition System with Stream Computing, DASFAA 2012