DEIM2022 K21-1

並列データストリーム処理におけるデータベースを用いた内部状態の共有

徳増　直紀†　　杉浦　健人†　　石川　佳治†　　陸　　可鏡†

† 名古屋大学情報学研究科　〒 464–8603 愛知県名古屋市千種区不老町

Email: {tokumasu,sugiura}@db.is.i.nagoya-u.ac.jp, ishikawa@i.nagoya-u.ac.jp，lu@db.is.i.nagoya-u.ac.jp

あらまし　現在、データ解析におけるストリーム処理による遅延削減は広く行われている．しかし，Flink や Samza などの既存のストリーム処理システムでは，分散並列処理のためのシャッフリングが処理のボトルネックになる．また，他にもキーが偏った際に負荷が特定のスレッドに集中する，シャッフリングにより障害発生時の影響範囲が拡大するという課題も存在する．そこで，本研究ではシャッフリングの代わりにデータベース技術を用いて内部状態の共有を行うことで，上記の問題に対応可能なストリーム処理のアーキテクチャを提案する．

キーワード　ストリーム処理，シャッフリング，ステートフル処理

1 はじめに

現在，データ解析における意思決定の高速化が求めれられる場面も増加し，ストリーム処理による遅延削減も広く行われている．データ解析で従来用いられていた手法であるバッチ処理では，一定期間データを蓄積し，それらに対してまとめて処理を行う．そのため，最初のタプルを取得してから処理が完了するまでに一定の遅延が生じてしまう．一方でストリーム処理ではデータが貯まるのを待たずに，受け取ったタプル１つまたは小さな単位でまとめられたマイクロバッチに対して順次処理を行っていく．そのため，ストリーム処理はバッチ処理に対して全体的な処理のスループットは低下するが，タプルを取得してから処理結果を得るまでの遅延を削減できる．

一方，既存のストリーム処理システムである Apache Flink [1,2] や Samza [3]，Storm [4]，Heron [5] などは分散並列処理を前提としており，近年増加しているメニーコア CPU を効率的に利用できないという問題がある [6]．ストリーム処理を行う既存 OSS（open source software）はパイプライン処理に基づいており，処理は大きく分けてステートレス（stateless）処理とステートフル (stateful) 処理の２種類にわけられる．ステートレス処理は，選択や変換，射影等のように入力タプルに依存しない処理であり，これは単純な並列化が容易である．一方，ステートレス処理は集約や結合といったタプルが持つキーに依存する処理であり，単純には並列化ができない．そこで，既存システムではステートレス・ステートフル処理間でタプルのシャッフリングを行い，特定のキーに対するステートフル処理を一つのスレッドで行うことで並列処理を実現している．しかし，メニーコア CPU を持つ単一ノード上ではシャッフリングそのものがボトルネックとなりその性能を活かしきれないと指摘されている [6]．また，シャッフリングを用いた並列処理では，グルーピングされたデータに対する各ステートフル処理はシングルスレッドで行われるので，データに偏りが発生している場合特定のスレッドに負荷が集中してしまい性能が低下するという問題もある．

処理性能以外の面でも，シャッフリングによって障害発生時の影響範囲が拡大するという問題もある．既存システムでは，チェックポイントや書き込みログなどを用いてある特定の時点での内部状態をバックアップし，障害発生時にはそれらの情報のロールバックを行っている．しかし，入力時点ではまだキーはグルーピングされておらず，障害が発生していないタスクをロールバックしない場合，そのタスクではいくつかのタプルが重複して処理されてしまう．そのため，障害発生時には障害が発生していないタスクも含めてすべてのタスクをロールバックする必要がある．しかし，障害が発生していないタスクの内部状態などは本来そのまま利用可能であり，シャッフリングにより障害発生範囲が拡大してしまったといえる．

これらの課題に対処するために，本研究ではデータベース技術を用いて内部状態の共有を行う分散並列ストリーム処理のアーキテクチャを提案する，本稿では特に，単一ノード上での並列ストリーム処理に注目し，メニーコア CPU 上での処理性能向上を目指す．

2 関連研究

既存の分散並列ストリーム処理システムの OSS として，Flink と Samza における処理方式と内部状態の保持について述べる．

Flink は，汎用なストリーム処理を分散並列環境で実行するシステムであり，耐障害性として exactly-once セマンティクスを保証する．Flink での処理は論理的にはデータフローとして記述されるとともに，物理的にサーバ上のスレッドに処理を割り当てる際にはステートレス処理を可能な限りまとめるなど，タスク間での不要なデータの送受信を省くための最適化が行われる．また，結合や集約処理の実現にはシャッフリング（一部の結合にはブロードキャスト）を用いる．各タスクの内部状態は入力ソースから定期的に与えられるチェックポイントバリアに基づき永続化が行われる．つまり，チェックポイントバリア

がタスクを通過した時点で内部状態をローカル/リモートに永続化し，チェックポイントバリアがソースに到着した時点でシステム全体のチェック取得が完了する．Exactly-once セマンティクスを保証するためのチェックポイントの取得が非同期に行われる一方，障害発生時には障害が発生していないタスクを含めシステム全体をある特定の状態に戻す必要がある．

Samza は，分散並列バッチ処理のための MapReduce フレームワークと類似した概念を，ストリーム処理において実現しようとしたシステムである．つまり，HDFS からのデータの読み出し，Hadoop による処理，HDFS への書き出しという一連の流れに対し、Kafka からのストリームの読み出し，Samza による処理，Kafka への書き出しという流れを主に想定している．高機能なメッセージングキューである Kafka 等の存在を前提とすることで，Samza における処理の記述や耐障害性保証のための回復処理の簡略化を図っている．本研究においても，Samza と同様にストリームに対して仮定を置くことで，データベースを用いた内部状態の共有を実現している．

3 準　　備

本章では，本研究における前提について述べる．

3.1 入力ストリーム

まず，本研究における入力ストリームはキーバリューデータであるとし，いくつかの前提を置く．本研究ではストリーム処理の前段に Kafka [7,8] のような Pub/Sub メッセージキューの存在を想定する．なお，以下ではストリームデータを無限のタプルの集合として定義し，ストリーム中のある要素について言及する際はタプルという単語を用いる．

(a) ストリームの事前パーティショニング

ストリームは複数のパーティションに事前に分割されており，各サーバはそれらを重複なしで各タスクに割り当てる．そして各タスクは割り当てられたパーティションのストリームデータのみを読み取る．

(b) パーティション内における順序付け

各パーティション内のタプル集合は何らかの順序（メッセージングキューへの挿入順など) で並べられたシーケンスデータとする．シーケンス中の位置を示す値をオフセットを呼び，再送を開始したい位置を示すオフセットを与えることで任意の位置から再送可能であるとする．

(c) タイムスタンプの割り当て

ストリーム中の各タプルは，パーティション内におけるオフセットとは別に，タプルの時間情報を表すタイムスタンプ（イベントタイム）を持つ．つまり，配送の遅延やネットワークの障害などにより，各タプルの持つタイムスタンプの順序とメッセージキューへの挿入順が異なる非順序ストリーム（out-of-order stream）を想定する．

3.2 処理の性質

続いて，ストリーム処理システムにおいて用いられるステートフル・ステートレス処理のそれぞれに性質について述べる．

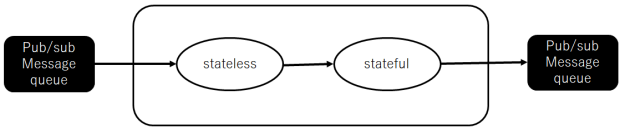

図 1　ステージ単位のストリーム処理

3.2.1 ステートレス処理

ステートレス処理は，データの内部状態（ステート）を考慮しないで，受け取ったデータのみを利用して行われる処理である．このような処理では，一つのタプルの処理が他のタプルの処理から独立している．例としては，一度に一つのタプルしか変換しないような場合や，ある条件に基づいてタプルを除外するような場合が挙げられる．ステートレス処理は，内部状態によって結果が変化しないため，別のスレッド間で処理を行ったとしても同じ結果が得られるため単純な並列化が可能である．

また，本研究ではストリームとテーブルの結合はステートレス処理として扱う．テーブル中のタプルの更新はストリームのタプルの追加に比べて稀であるとし，ある時点でのテーブルの情報を用いてストリーム・テーブル結合を行う．

3.2.2 ステートフル処理

一方，ステートフル処理はデータの内部状態に影響を受けて結果が決まるような処理である．例えば．アプリケーションで入力データの集約やウィンドウ化を行う必要がある場合，これらはそれ以前の処理の結果を必要とするためステートフル処理である．

内部状態の情報は複数のストリームや分散したストリームにわたって同時に管理する必要があるため，ステートレス処理に比べ非常に複雑になる．アクセスの多い Web サイトのユーザーの監視などを想定した場合，データ処理システムは一度に数千ものユーザーセッションのステートを管理する必要が出てくる．その結果，ネットワーク監視のアルゴリズムの要件が増えたり，複雑になったりする．

既存 OSS ではシャッフリングなどによってステートフル処理の前にデータをグルーピングしている．ここで，シャッフリングとはタプルのキーをもとに入力データを複数のグループに分割する処理である．これにより，同じキーに対する処理はすべて同じスレッドに割り当てることが可能になる．

3.3 処 理 方 式

本研究では，ステージ単位のストリーム処理の記述を想定する．ステージ単位の処理の記述は Samza や，Spark Streaming の API 拡張である Structured Streaming で想定されている処理であり，入力ソースから受け取ったデータストリームに対して比較的単純な処理を適用し，出力シンクへ書き出すまでを 1 ステージとして考える方式である．例えば，Samza では Hadoop における HDFS からの読み出し・Hadoop 上での処理・HDFS への書き出しに対応するものとして，Kafka からの読み出し・Samza 上での処理・Kafka への書き出しを挙げている．中間データと

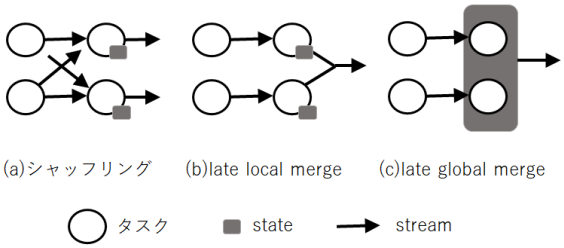

図 2　シャッフリングと late merge

なるようなストリームを適宜耐障害性を持つメッセージキューなどへ書き出すことで，1 ステージ毎のストリーム処理の記述やストリーム処理としての耐障害性保証の簡略化を目的としている．

特に，本研究では Structured Streaming と同様に，1 ステージにおける処理はいくつかのステートレス処理の組み合わせと多くても一つのステートフル処理から構成されると想定する．つまり，図 1 に示すようなストリーム処理を対象として考える．

4　Late Merge による内部状態共有

本章では，Zeuch らの提案した late merge 手法 [6] を用いた内部状態の共有について紹介する．

Zeuch らの提案した late local/global merge と，比較のために既存 OSS で用いられている手法を図 2 に示す．まず，(a) は Hadoop や Spark などの既存のバッチ処理システムで採用されている方法に準じている．この手法では内部状態の共有は行われず，タプルはキーに基づいてシャッフリングされており，各ステートフル処理の実行スレッドに分配される．この手法の利点としては，十分な種類のキーとスレッドが確保でき，偏りなく分散することが出来れば優れた並列性が得られるということが挙げられる．一方で，キーに偏りが発生した場合は特定のスレッドに処理が集中してしまうという問題がある．また，one-by-one 方式でストリーム処理を行う場合，シャッフリングされたタプルを受けとるキューがボトルネックとなる．

上記の問題に対し，Zeuch らは late local/global merge と呼ぶ集約処理でのマージ手法を提案した．これらの手法はキーに基づく事前のシャッフリングは行わず，部分的に集約処理を行った後に各集約処理の結果をマージする．(b) の late local merge では集約処理を行う各タスクが並列で自身の入力ストリームに対する集約を行い，出力ストリームで追加のマージを行うことで全体の集約結果を得る．一方，(c) の late globla merge では各集約タスクが内部状態を共有しており，並列で動く集約処理の結果を適宜マージすることで全体の集約結果を得る．Zeuch らはこれらの手法を C++及び Java で実装し、既存の OSS に比べ優れた性能を達成したと報告している．

しかし，late merge は実際のシステムでの使用が十分に考慮されていない．例えば，late local merge は各ステートフル処理の結果をマージする追加処理が必要であるが，結合（直積）を含む問合せの処理方法は十分に検討されていない．仮にハッシュ結合を利用する場合，並列で構成された部分ハッシュテーブルを後段でマージした後，そのハッシュテーブルを全ノードに対しブロードキャストする，もしくはハッシュテーブルを持つスレッド（ないしノード）が全ての結合を担う必要がある．しかし，これらの検討は Zeuch らの提案では行われておらず，メニーコア CPU の活用に対する方向性の提示のみとなっている．

そこで本研究では，Zeuch らの提案に対しシステムとしてより現実的な実装方法を検討する．マージ方式としては，2 種類の late merge のうち late local merge を採用する．late global merge では集約処理のために全ノードが内部状態を共有する必要があり，分散処理を想定した場合リアルタイムでの同期が必要になる．しかし，リアルタイムでの内部状態の同期は設計が複雑になり，またその影響で性能の低下が起こることも考えられる．そのため，実装が単純で性能の向上も見込みやすい late local merge を今回は実装する．

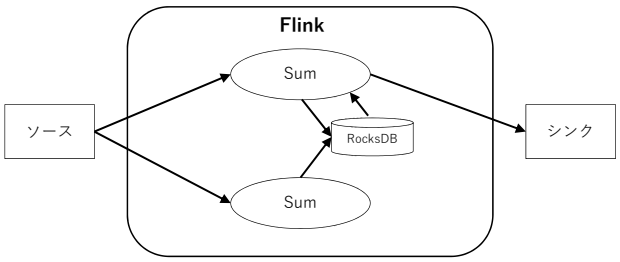

図 3　提案アーキテクチャ

5　提案手法の詳細

本研究では，既存の分散並列ストリーム処理システムである Apache Flink を基にした実装を想定し，基本的なアーキテクチャは同様のものを用いる．ただし，本研究では単一のノード上での処理を想定しており，分散化した際の検討は今後の課題である．

内部状態の共有のために，本研究では RocksDB [9] を用いる．並列で動作するスレッドの内部状態を RocksDB を用いた書き込み，読み取りによって共有し，シャッフリング無しでの整合性の維持を図る．

本研究の提案アーキテクチャを図 3 に示す．入力ソース及び出力シンクとしては Pub/Sub メッセージキューを想定する．以下では，提案アーキテクチャにおける処理の概要について述べる．

入力ソースとなるメッセージングキューで事前にパーティショニングされた入力ストリームは，Flink 上の並列スレッドに重複なく割り当てられる．そのため，入力ソースのパーティション数が処理の並列数の上限となる．各スレッドは受け取ったパー

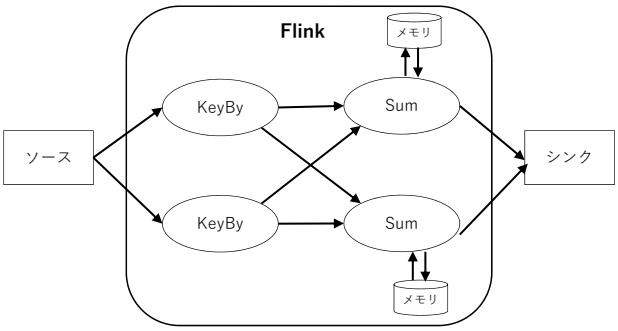

図 4　Flink のアーキテクチャ

ティションからタプルを読み取り，それに対して並列で集約処理を実行する．その後，各ワーカは集約結果を RocksDB に書き込みマージを行う．集約結果のシンクへの出力はウィンドウ単位とし，全てのワーカが特定のウィンドウに対する処理を終えた際に行う．特定のウィンドウに対する処理を行う最後のワーカが集約処理を終えたタイミングで，そのスレッドが RocksDB から集約結果を読み出し，それをシンクに出力する．

本手法の利点としては，キーに偏りが存在した場合にパフォーマンスが低下しないということがある．既存手法では，キーに基づいたシャッフリングをおこなうため，キーに偏りが存在した場合特定のスレッドに負荷が集中し全体としてパフォーマンスは低下する．しかし，本手法ではシャッフリングを用いずに各スレッドが入力ストリームから受け取ったタプルをそのまま利用するため，キーに偏りがあった場合も各スレッドのタプル数は一定となる．そのため，特定のスレッドに負荷が集中することはなく，パフォーマンスも低下しない．

また，障害が発生した際に復旧が必要な範囲が拡大しないということもある．詳しくは次章で説明するが，既存手法は障害発生時に障害と無関係なスレッドも再処理する必要があるが，今回の手法は障害が発生したスレッドだけを再処理すればよい．

6　提案手法の耐障害性

本章では，本研究の提案アーキテクチャにおける耐障害性の保証について述べる．まず，既存の Apache Flink のアーキテクチャとそれにおける耐障害性について説明する．Flink では，入力ソースとなるメッセージングキューからパーティショニングされた入力ストリームを受け取った後，KeyBy 操作によってそれらを Key 毎にシャッフリングする．その後，ワーカは KeyBy によってキー毎にパーティショニングされたデータに対し，集約処理を実行する．その際，集約処理の内部状態は各ワーカごとにローカルのメモリ上で管理され，スレッド間での共有は行われない．そして，集約処理が終了後，各スレッドはそれぞれが集約結果をシンクに出力する．

Flink の耐障害性は，チェックポイントに基づいたものとなっている．Flink では，入力ソースでチェックポイントバリアが生成，データストリームに挿入され，そのままデータストリームの一部としてタプルとともに流れる．このチェックポイントバリアを各オペレータが受け取ると，オペレータは非同期にそれまでの集約結果をチェックポイントとして記録する．そして，チェックポイントバリアがシンクに到着すると，処理全体のチェックポイントを記録する．ここで，Flink のいずれかのスレッドで障害が起こった場合を想定する．Flink は最新のチェックポイントまでのデータを復旧するが，チェックポイント以降のそのスレッドの処理の復元には複数の入力ストリームからのタプルが必要である．結果，複数の入力ストリームからデータを再送すると複数のスレッドで再処理することになってしまい，本来障害が発生した場所以上に影響が出てしまう．

提案手法でも同様に入力ソースでチェックポイントバリアを生成し，チェックポイントバリアはストリームの一部としてタプルとともに流れていく．チェックポイントバリアをオペレータが受け取ると，オペレータはそれまでの集約結果を RocksDB にチェックポイントとして記録する．提案手法では，いずれかのスレッドで障害が起こった場合，Flink と同様に最新のチェックポイントまでのデータを復旧するが，チェックポイント以降のそのスレッドの処理の復元はそのスレッドの入力ストリームだけを再送すればよい．よって，他のスレッドはデータの再送が行われず，処理をそのまま継続することが可能であり Flink に比べ障害発生時の影響範囲を抑えることが出来る．

7　実　　験

本稿で提案したアーキテクチャを実際に実装し，評価実験を実行した．本章では，その実験と結果について述べる．

実験としては既存の Apache Flink のシャッフリングを用いたプログラムと，今回の提案手法を用いたプログラムの 2 種類を動かし，スループットを測定した．処理は単純なキー毎の総和の計算である．入力データとしてはキー毎のタプル数に偏りがなく均一になっているものと，ある特定のキーのタプル数だけ多くキーに偏りが存在する 2 種類のものを使用した．偏りのないデータとしては，10 万種類のキーを各 1000 個ずつ用意した．偏りの存在するデータとしては，10 万種類のうち 1 つのキーだけを 100 万個，それ以外のキーは各 900 個ずつ用意した．合計のタプルの数とキーの種類の数としては，どちらもそれぞれ 1 億個，10 万種類となっている．パラメータとしては，キーの偏りとスレッド数，キーの種類数を変化させた．

まず，提案手法と既存の Apache Flink を利用したプログラムそれぞれに対し，キーに偏りのないデータセットを用いて，スレッド数を 4 から 60 まで 4 刻みで変化させて計測を行った．その結果が図 5 である．なお．WithRocksDB が今回の提案手法を用いたプログラムの結果を表し，WithoutRocksDB が既存の Flink を用いたプログラムの結果を表す．

結果としては，どちらのプログラムもスレッド数を増やすとスループットは向上していき，30 前後からほとんど変わらなくなっている．これは，今回の実験では計測時間は Flink がプログラムを読み込む時間なども含まれているため，並列数を増

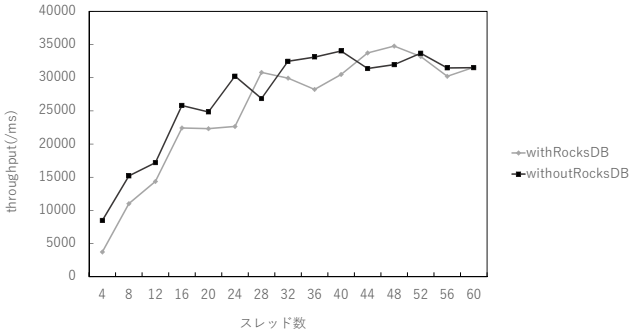

図 5　キーに偏りがないデータセットを用いた際のスループット

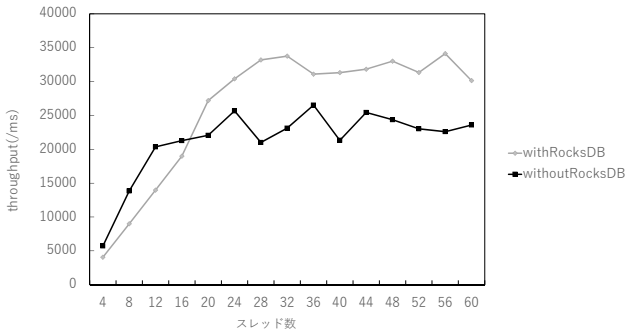

図 6　キーに偏りが存在するデータセットを用いた際のスループット

やしてもその時間は削減できないからだと思われる，２つのプログラムの結果を比較すると，全体として提案手法の方がスループットが少し低くなっている．理由として，提案手法では RocksDB への書き込みや読み出し，マージといった処理を行っており，これが Flink の KeyBy のよるシャッフリングよりも時間がかかっていることが考えられる．

続いて，先程と同様の実験をキーに偏りのないデータセットの代わりにキーに偏りの存在するデータセットを用いて計測を行った．その結果が図 6 である．２つのプログラムの結果を比較すると，全体として提案手法の方がスループットが高くなっている．理由として，Flink を用いたプログラムではキーに偏りがあるとシャッフリングにより特定のスレッドにタプルが集中してしまい，そのスレッドで処理に時間がかかるからである．一方，提案手法ではキーに偏りがあっても各スレッドの処理するタプルの数は均等であるため，処理の時間はキーに偏りがない場合と変わらない．スレッド数が少ない時は Flink を用いたプログラムの方がスループットは高くなっているが．これはおそらくスレッド数が少ない場合はそもそも各スレッドのタプル数が多いため，極端に偏りがある場合を除き，負荷が集中しないからだと思われる．

続いて，キーの種類数を変化させて２つのプログラムで実験

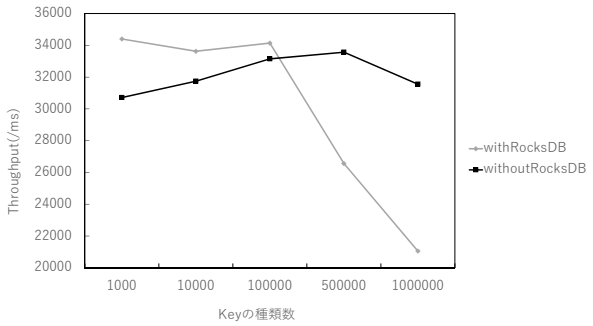

図 7　キーの種類数を変化させた際のスループット

を行った．この際，スレッド数は 60，キーの偏りはないものを扱う．その結果が図 7 のようになる．キーの種類数が多いとき，提案手法はスループットが低下している．一方，Flink を用いたプログラムの方はキーの種類数を増やしてもスループットは変わらない．これは，提案手法では RocksDB を用いているため，キーの種類数が多いと RocksDB で作成されるテーブルが大きくなり，マージや読み取りの際のコストが大きくなるからだと考えられる．

今回の実験の結果，提案手法はキーに偏りが存在する場合，既存手法よりも優れたパフォーマンスを発揮できることがわかった．一方で，キーに偏りが存在しない，あるいは少ない場合は，RocksDB への書き込み，読み取り，マージといった処理がボトルネックとなり，既存手法に比べ性能が低下してしまっていた．これは，キーの種類が多い場合これはより顕著に現れていた．このことから，本手法はキーが偏るようなユースケースに対しては有用であると考えられる．また，本論文では単一のノード上での処理を想定していたが，実際は複数のノードでの内部状態の共有などを考える必要があるため今後の課題である．

8　おわりに

本研究では，分散並列ストリーム処理システムにおける，シャッフリングをせず代わりにデータベースを用いて内部状態を共有する手法について提案した．本手法は，Zeuch らの提案した late local merge に基づいており，システムとして現実的な実装を検討した．また，実際に実装を行い，それに対して性能評価の実験も実施した．今後の課題としては，今回の実装では単一ノード上での処理を想定していたため，複数サーバを想定した実装などが挙げられる．

謝　　辞

本研究は JSPS 科研費（16H01722，20K19804）の助成，および国立研究開発法人新エネルギー・産業技術総合開発機構（NEDO）の委託業務（JPNP16007）から得られた結果による．

文　献

[1] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State management in Apache Flink®: consistent stateful distributed stream processing," *PVLDB*, vol. 10, no. 12, pp. 1718–1729, 2017.

[2] Apache Flink: Stateful Computations over Data Streams: `https://flink.apache.org/` (accessed Feb. 9, 2022).

[3] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: stateful scalable stream processing at LinkedIn," *PVLDB*, vol. 10, no. 12, pp. 1634–1645, 2017.

[4] A. Toshniwal, J. Donham, N. Bhagat, S. Mittal, D. Ryaboy, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, and M. Fu, "Storm @Twitter," in *Proc. SIGMOD*, pp. 147–156, ACM Press, 2014.

[5] M. Fu, A. Agrawal, A. Floratou, B. Graham, A. Jorgensen, M. Li, N. Lu, K. Ramasamy, S. Rao, and C. Wang, "Twitter Heron: Towards extensible streaming engines," in *Proc. ICDE*, pp. 1165–1172, 2017.

[6] S. Zeuch, B. D. Monte, J. Karimov, C. Lutz, M. Renz, J. Traub, S. Breß, T. Rabl, and V. Markl, "Analyzing efficient stream processing on modern hardware," *PVLDB*, vol. 12, no. 5, pp. 516–530, 2019.

[7] J. Kreps, N. Narkhede, and J. Rao, "Kafka: a distributed messaging system for log processing," in *Proc. International Workshop on Networking Meets Databases (NetDB)*, pp. 1–7, 2011.

[8] G. Wang, J. Koshy, S. Subramanian, K. Paramasivam, M. Zadeh, N. Narkhede, J. Rao, J. Kreps, and J. Stein, "Building a replicated logging system with Apache Kafka," *PVLDB*, vol. 8, no. 12, pp. 1654–1655, 2015.

[9] RocksDB | A persistent key-value store | RocksDB: `https://rocksdb.org/` (accessed Jan. 9, 2020).