並列ストリーム処理システムにおけるDBを用いた内部状態の共有手法

杉浦　健人†　　石川　佳治†

† 名古屋大学大学院情報学研究科　〒 464-8601 愛知県名古屋市千種区不老町
E-mail: sugiura@db.is.i.nagoya-u.ac.jp, ishikawa@i.nagoya-u.ac.jp

あらまし　近年，IoT で得られるセンサデータなどを連続的に処理するストリーム処理に注目が集まっており，学術・産業両面で活発に研究開発が行われている．例えば，既存の OSS であれば Flink や Samza など，いくつかの分散並列ストリーム処理システムが提案されている．一方，そうした既存システムに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となる，入力データの偏りにより性能が悪化するなどの課題がある．そこで，本研究では各サーバ内における内部状態を DB により共有することで，部分的にシャッフリングを避けつつ入力データの偏りにも対応可能な並列ストリーム処理について検討する．

キーワード　並列ストリーム処理，内部状態共有

1　はじめに

ビッグデータ時代の今日ではデータ解析による意思決定の高速化が進んでおり，ストリーム処理による遅延削減が活発に行われている．従来のバッチ処理ではデータの取得が完了してから処理を開始するため，一番初めのタプルを取得してから最後のタプルを取得するまでの時間という回避不可能な遅延が存在する．一方，ストリーム処理では全データの取得を待たずに，タプル一つ一つ，ないし小さな単位でまとめられたバッチ（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，バッチ処理に対し全体的な処理スループットは低下するが，最終タプルを取得してから処理終了までの遅延が削減できる．

ストリーム処理を実現する既存の OSS（open source software）としては，例えば Flink [1] や Samza [2]，Storm [3]，Heron [4] など様々な分散並列ストリーム処理システムが開発されている．これらの OSS では主にシェアードナッシングアーキテクチャ及びパイプライン処理が採用されており，パイプライン上の各タスクを複数のサーバで分散処理することで性能のスケールアウトを図っている．なお，タスクは選択や変換などのようなステートレス（stateless）処理と，集約や結合などのようなステートフル（stateful）処理とに分けられる．並列処理において，キーに依存しないステートレス処理は入力ストリームを単純に並列処理できるのに対し，ステートフル処理は処理内容がキーに依存するため入力ストリームをシャッフリングなどによって処理前にグルーピングする必要がある．

一方，既存システムに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となる，入力データの偏りにより性能が悪化するなどの課題がある．Zeuch ら [5] は近年の CPU のメニーコア化を受け，メニーコア環境における既存システムのボトルネックを調査した．その結果，分散処理時におけるネットワーク越しのシャッフリングだけでなく，メモリ上でのロックフリーキューを用いたデータのシャッフリングもボトルネックとなりうることを示し，シャッフリングを排除した late local/global merge 方式の処理を提案している．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ため，データに偏りが発生したとき特定のスレッドに負荷が集中するという問題もある．

処理性能以外の面では，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見られる．例えば，あるステートフル処理を行うスレッドで障害が発生し，再処理が必要となった場合を考える．既存の OSS ではチェックポインティングや書き込みログ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ているため，そうした内部状態をリストアすることでシステム全体の内部状態を特定の時点にロールバックする．その後，Kafka [6, 7] などの入力ソースからタプルを再送，再処理することで内部状態を回復するとともに，正常時の処理へと移行する．このとき，システム全体で一貫性の取れた内部状態をロールバックしなければならないのはタプルのシャッフリングを伴うためである．ステートフル処理の前でのシャッフリングが前提となっているため，障害が発生したタスクに対してタプルを再送するとき，図 1 のように障害が発生していないタスクにも同様にタプルが再送される．障害が発生していないタスクをロールバックしていない場合，そのタスクではいくつかのタプルが重複して処理され，at-least once な処理となる．しかし，障害が発生していないタスクの内部状態は本来継続して利用可能であり，シャッフリングによって障害範囲が拡大していると言える．

そこで，本研究では Zeuch ら [5] の late global/local merge 方式のストリーム処理を踏襲しつつ，各サーバ内における内部状態をデータベースを用いて共有する分散並列ストリーム処理のアーキテクチャを検討する．提案アーキテクチャではステートフル処理の前のシャッフリングは行わず，ステートフル処理の途中経過をデータベースを用いて共有，マージすることで整合性の取れた処理結果の出力を行う．シャッフリングを排除することである特定のタスクで障害が発生した際の影響範囲を限定し，障害と無関係なタスクの処理継続を検討する．また，

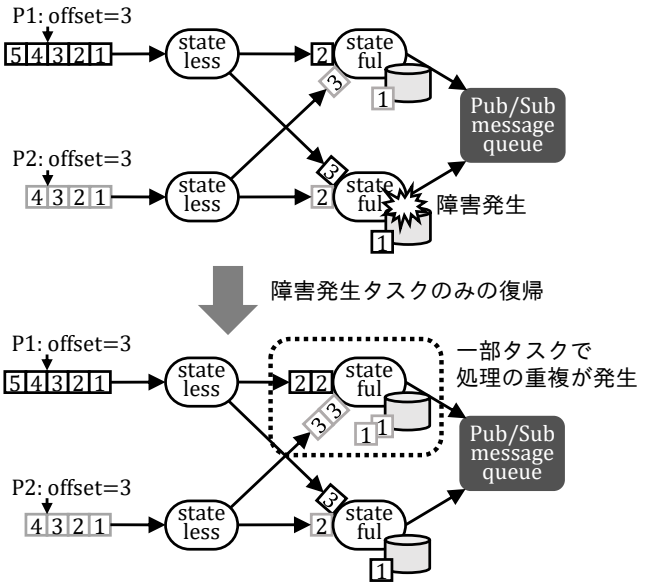

図 1　復帰時のタプル再送による重複処理の発生

ステートフル処理のためのパーティショニングを行わずデータベース上で内部状態をマージすることで，入力ストリームに偏りが発生した際の性能悪化の軽減を目指す．

2　準　　備

提案アーキテクチャについて議論するために，本研究における前提を述べる．

2.1　入力ストリーム

本研究では，入力ストリームはキーバリューデータであるとし，Samza [2] と同様にいくつかの前提を置く．端的に言えば，本研究ではストリーム処理システムの前段に Kafka [6] 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，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

入力ストリームは複数のパーティションに分割されている．各サーバはそれぞれ重複が発生しないようパーティションを割り当て，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込む．

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（e.g., メッセージングキューへの挿入順）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と呼び，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当

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タンプの順序とメッセージキューへの挿入順とが異なる非順序ストリーム（out-of-order stream）を想定する．

2.2　時間窓への分割

本研究では固定幅の時間窓 [8]，つまりタンブリングウィンドウ及びスライディングウィンドウのみを扱う．代表的な時間窓への分割としてはセッションウィンドウがあるが，本稿では扱わず，その実現は今後の課題とする．

タンブリングウィンドウでストリームを分割する場合，与えられた時間窓の幅 w を用いて重複のない時間窓を生成する．例えば，ある時刻 t から時間窓を生成する場合，以下のような時間窓の集合が生成される．

$$[t, t+w), [t+w, t+2w), [t+2w, t+3w), \ldots \tag{1}$$

スライディングウィンドウによってストリームを分割する場合，与えられた時間窓の幅 w 及び移動幅 l を用いて重複ありの時間窓を生成する．例えば，ある時刻 t から時間窓を生成する場合，以下のような時間窓の集合が生成される．

$$[t, t+w), [t+l, t+w+l), [t+2l, t+w+2l), \ldots \tag{2}$$

2.3　処理の性質

ストリーム処理システムで行われるステートフル・ステートレス処理について，前提となる性質を述べる．

2.3.1　ステートレス処理

本研究では，ステートレス処理はあるバリューデータから別のバリューデータへの 1 変数の写像として扱う．つまり，あるステートレス処理への入力ストリームのバリューの全集合を V_{in}，出力ストリームのバリューの全集合を V_{out} とするとき，ステートレス処理 f を以下の式で表す．

$$f : V_{in} \to V_{out} \tag{3}$$

具体的には，射影や選択，変換などがステートレス処理として挙げられる．

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，つまり 1 変数の写像として扱う．言い換えれば，本稿ではストリームを動的データ，テーブルを静的データとみなす．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．

2.3.2　ステートフル処理

本稿では，ステートフル処理として時間窓単位の集約演算のみを想定する．つまり，集約対象となるバリューデータ V において，結合律及び交換律が成り立つ 2 変数の写像 f を用いた集約処理を扱う．

$$f : V \times V \to V \tag{4}$$

$$\forall x, y, z \in V, f(f(x, y), z) = f(x, f(y, z)) \tag{5}$$

$$\forall x, y \in V, f(x, y) = f(y, x) \tag{6}$$

集約演算では，この性質に基づきある入力ストリームのバリューの集合を一つのバリューへと集約する．つまり，2 変数の写像

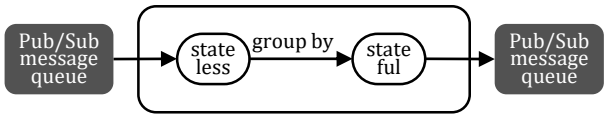

図 2　ステージ単位のストリーム処理

ではあるが，入力ストリームは一つである．具体的には，整数における加算や乗算に基づく集約処理を想定する．

上述したとおり，本研究では結合律及び交換律が成り立つことをストリーム処理における集約処理の条件として考える．これらの性質は非順序ストリームに対してスライシング手法 [9,10] を適用するための性質として挙げられているが，同時にインクリメンタル処理の適用の可否にも直結している．例として，行列の積を行う集約処理を考える．行列の積は結合律しか成り立たないため，演算を適用する順番によって最終的な計算結果が異なる．つまり非順序ストリームにおいては，時間窓内のタプルが全て揃うのを待ち，タイムスタンプに基づいてソートを行わなければ適切な結果が得られない．ソート部分をインクリメンタルに処理するとしても，集約部分はバッチ処理となってしまい，ストリーム処理を適用する意義が薄れる．そのため，本研究では結合律・交換律が成立しない集約処理は対象外（バッチ処理の範疇）とする．

なお，代表的なステートフル処理としては他にストリーム同士の結合（直積）も挙げられるが，本稿では扱わずその実現は今後の課題とする．

2.4 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合せと高々一つのステートフル処理から記述されると想定する．つまり，図 2 に示すような概念図で表されるストリーム処理を対象として考える．

3　提案アーキテクチャ

本研究で提案するアーキテクチャを図 3 に示す．基本的な構成要素は既存の分散並列ストリーム処理システムである Flink や Samza などと同様であり，処理を行うサーバ群をクラスタ

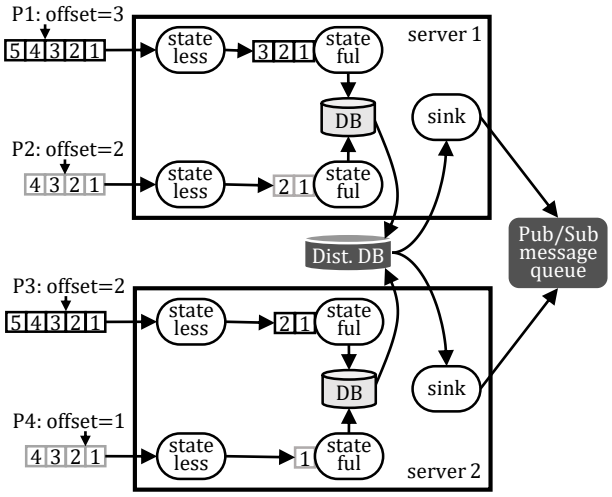

図 3　データベースによる内部状態の共有・マージを行うアーキテクチャ

として持ち，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．ただし，耐障害性保証のためのバックアップストレージとしては分散データベースを使用し，各サーバ上ではローカルストレージとしてインメモリデータベースを持つ．以下では，提案アーキテクチャにおける処理の概観について述べる．

入力ソースとなるメッセージキューでパーティショニングされた入力ストリームは各サーバ上の並列タスクに重複なく割り当てられ，各タスクはそれぞれ非同期にストリームをプルする．つまり，入力ソースのパーティション数が処理の並列度の上限となる．

各並列タスクは読み込んだタプルに対してステートレス処理及びステートフル処理を連続して適用し，部分的な集約結果をまずローカルのインメモリデータベース上で共有，マージする（ローカルチェックポイント）．その後，各サーバのインメモリデータベース上の内部状態を分散データベースへ送り，分散データベース上で同様にマージ処理を行う（グローバルチェックポイント）．ウォーターマーク [8] などによりある時間窓に対する処理結果が出力可能であると判定したとき，一つ以上の出力タスクが分散データベース上のマージ結果を読み取り後段のメッセージキューへと結果を出力する．ただし，もしキーによるパーティショニングが事前に許される環境であればメッセージキューの時点でキー毎のパーティショニングを行い，サーバ間でのキーの共有が発生しないようにする．つまり，各ストリーム処理はサーバ単位で完全に分離した状態で行われ，サーバ内でのインメモリデータベースを用いた内部状態の共有・マージのみを実行し，分散データベースは内部状態のバックアップのためのみに用いる．

入力タプルを受け取るタスクは Flink と同様に定期的にチェックポイントを取得するためのチェックポイントバリアを生成し，各並列タスクに割り当てられたパーティションのオフセット位置を把握する．上述したローカルのインメモリデータベース上での内部状態の共有はチェックポイントバリアが到達したタ

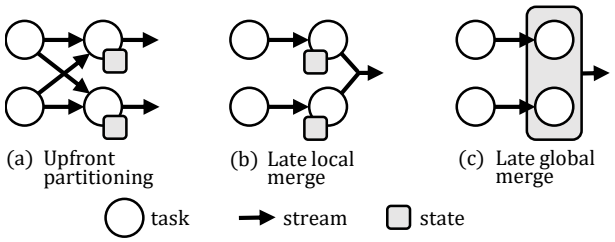

図 4　ストリーム処理における並列化戦略 [5]

イミングで行われ，把握したオフセット位置までの内部状態をローカルチェックポイントとして書き出す．一方，グローバルチェックポイントの取得はローカルチェックポイントの取得とは非同期で行われ，任意のタイミングで各サーバのローカルチェックポイントを分散データベースへと書き出す．

以下では，インメモリデータベース及び分散データベースを用いた内部状態の共有について具体的に述べる．

4　データベースを用いた内部状態の共有

本章ではステートフル処理，つまり集約処理における内部状態の共有について述べる．

本研究では，Zeuch ら [5] の提案する late global merge 方式を基に，集約処理における内部状態の共有を検討する．比較のため，Zeuch らによる 3 種類の並列処理の戦略を図 4 に示す．Upfront partitioning はキーに基づくシャッフリングを用いてデータの分割を行う，既存の OSS で主に使用されている戦略である．集約処理を行う後段の各タスクの入力ストリームにはあるキーを持つタプルが過不足なく存在し，各タスクはキー毎に集約処理を実行できる．Late local/global merge で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．Late local merge では集約処理を行う各タスクは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処理を行うことで全体の集約結果を得る．一方，late global merge では各集約タスクは内部状態を共有し，並列で行われる部分集約結果を適宜マージすることで全体の集約結果を得る．

Late global merge を実現する方法として，具体的には内部状態をデータベース上のテーブルで表し，マージ処理をトランザクションとして記述することでこの実現を図る．

まず，内部状態を共有するためのテーブル構造について述べる．本稿で想定するステートフル処理は各時間窓に対して実行されるため，各集約結果は時間窓を区別するための時間窓 ID とタプルに付随するキーとを用いて一意に識別できる．例えば，$w = 10$ かつ $l = 5$ のスライディングウィンドウにおける各時間窓各キーに対する集約結果は表 1 のように表せる．なお，表 1 ではわかりやすさのために時間窓の ID を区間（開始時刻は開区間，終了時刻は閉区間）で表しているが，実際には一意の ID を振り分けるか時間窓毎にテーブルを用意するなどの方法で実装する．

上述のようなテーブルを用意したとき，集約処理のための内部状態の更新はトランザクション処理として表せる．集約処理のための 2 項演算を f，ある時点における部分的な集約結果を $o_{prev} \in V$，新たに入力されたタプル中の集約対象の値を $v \in V$ とすると，新たな集約結果 $o_{new} \in V$ を得るためのインクリメンタルな処理は以下の式で表せる．

$$o_{new} = f(o_{prev}, v) \tag{7}$$

2.3.2 項で述べたとおり，本研究では結合律及び交換律が成り立つ 2 項演算を基にした集約処理を扱うため，入力ストリームを集約する際の 2 項演算 f の適用順は最終結果に影響を与えない．つまり，この処理を複数のタスクで並列で実行する際は，データベース上で共有された o_{prev} を読み込み計算結果である o_{new} を書き出すトランザクション処理として表せる．

したがって，内部状態を共有，マージするための一連の処理はデータベース上で実現でき，同時実行制御や永続化などの既存技術を応用できる．Zeuch ら [5] は late local/global merge を実現するためにロックフリーキューを用いて時間窓集約用の alternating window buffers を実装しているが，これは状態の共有とマージのみを目的としており，状態の永続化は考慮していない．実際のストリーム処理システムでは耐障害性保証のために内部状態を永続化する必要があるため，本研究では共有・マージ処理と耐障害性保証のための永続化をデータベース上で実現することで，実装の効率化を図る．ただし，集約処理のための内部状態の共有・マージは競合が発生しやすい処理であるため，処理性能向上のためのスケジューリングは今後の課題である．

表 1　$w = 10$ かつ $l = 5$ のスライディングウィンドウにおける内部状態 o の共有テーブルの例

window id	key	value
[0, 10)	0	$o_0^{[0,10)}$
[0, 10)	1	$o_1^{[0,10)}$
	...	
[5, 15)	0	$o_0^{[5,15)}$
[5, 15)	1	$o_1^{[5,15)}$
	...	

5　耐障害性保証の方針

提案システムでは，インメモリデータベースを用いたローカルチェックポイントの管理及び分散データベースを用いたグローバルチェックポイントの管理を行う．インメモリデータベースを用いたローカルチェックポイントは分散して処理を行う各サーバ内で保持され，各サーバに割り当てられたパーティションのある時点（オフセット）までのステートフル処理の結果を持つ．一方，グローバルチェックポイントはそれらローカルチェックポイントを分散データベース上でひとまとめにしたものであり，分散処理されたステートフル処理全体の結果を持つ．

5.1　ローカルチェックポイントの取得

ローカルチェックポイントは，インメモリデータベース上での WAL（write ahead log）フラッシュのタイミングを調整する

ことで処理効率の向上と内部状態の永続化を実現する．上述したように，各サーバ内における内部状態の更新はトランザクションとして記述され，マルチスレッドによる並列処理の際の更新の原子性・一貫性・独立性を保証する．一方で，トランザクションを実行するたびに内部状態の更新結果を永続化する場合ストレージへの I/O が多数発生し，処理のスループットの低下が予想される．そこで，3 章で述べたチェックポイントバリアを受け取ったタイミングでのみ WAL をストレージへ書き出すことで，ストレージ I/O を削減する．

具体的には，ローカルチェックポイントの取得状況をステートマシンで表し，WAL フラッシュのタイミングを決定する．まず，開始状態ではいずれのステートフルタスクもチェックポイントバリアは受け取っていない状態から始まる．その後，いずれかのステートフルタスクがチェックポイントバリアを受け取ることでチェックポイント取得中状態へと移る．このとき，チェックポイントバリアを受け取ったタスクは内部状態を更新するテーブルを切り替え，それ以降そのタスクがチェックポイントの対象となったオフセット位置までの内部状態を更新しないようにする．つまり，チェックポイントバリアを受け取ったタスクは切り替えた先のテーブルを用いて処理を継続するため，チェックポイントの取得は各タスクで非同期に行われる．全てのタスクがチェックポイントバリアを受け取った時点でチェックポイントの取得完了状態へ移り，グローバルチェックポイントのための分散データベースへの書き出しを実行し，元の状態へと戻る．

インメモリデータベースを用いて内部状態を管理する利点は，インメモリデータベース上の内部状態が常にストリーム処理に関して一貫性の取れたものであることを保証できる点である．この一貫性は，各並列タスクがインメモリデータベースを更新する際，その差分を計算するのに使用した各ストリームパーティションのオフセット位置も同じトランザクション内で更新することで保証できる．つまり，仮にサーバ内で何らかの原因によって並列タスクを実行するスレッドに障害が起きたとしても，代替スレッドはインメモリデータベース上のデータから各ストリームパーティションの次の読み出しオフセットの位置を把握し，処理を継続できる．また，各並列タスクが読み出すパーティションは決まっているため，他の並列タスクは他スレッドの障害を無視して処理を継続できる．

5.2 グローバルチェックポイントの取得

グローバルチェックポイントは各サーバから書き出されたローカルチェックポイントをマージすることで生成する．マージ操作は結果の出力時，もしくは適宜分散データベース上で実行される．なお，マージ操作自体はインメモリデータベース上で行うものと同様である．

ただし，前述したようにサーバ単位でのキーのパーティショニングが行われている場合は，インメモリデータベース上の時点でマージ操作は完了しているため，分散データベース上でのマージは実行されない．つまり，サーバ障害発生時のためのバックアップとしてのみ分散データベースは利用される．

表 2　測 定 環 境

item	value
OS	Ubuntu 18.04 LTS
CPU	Intel(R) Xeon(R) Gold 6262V CPU ×2
	(cores: physical 48, logical 96)
RAM	224GB
JVM	OracleJDK 1.8.0_231
Flink	ver. 1.9.1 (stand alone mode)

6 予 備 実 験

本章では，提案するインメモリデータベースを用いた並列ストリーム処理に関して，予備実験としてメニーコア環境での既存 OSS の性能を調査し，提案システムの一部を実装した試作を用いて評価する．実験に用いたサーバのスペックなどを表 2 に示す．

検証用のデータはシミュレーションによりミニベンチマークを作成した．ストリーム中の各タプルはタイムスタンプ，キー，バリューの三つ組であり，以下の実験では 1 億タプルのストリームを評価に使用した．キーには Zipf の法則に基づき偏りを与えており，パラメータは 0.2 刻みで 1.2 から 2.0 まで変化させた．なお，以下ではキーの分布が均一であることをパラメータ 1.0 で表す．各バリューは 0 から 100 の間で乱数を生成した．

処理は単純なステートレス処理とステートフル処理の連結である．ステートレス処理では入力されたタプル（文字列）をパースし，ステートフル処理ではキー毎にバリューの総和を計算した．

提案するデータベースを用いた内部状態の共有について，Flink の API を用いる形で試作し，その性能を評価する．共有用のデータベースにはインメモリのキーバリューストアである RocksDB [11] を使用し，耐障害性は考慮せずに試作を行った．つまり，各タスクで並列に処理される時間窓の集約結果はそれぞれ一度だけデータベースに書き出され，全てのタスクの処理結果がマージされた時点で出力される．なお，今回の実験ではタンブリングウィンドウでかつ一つの大きな時間窓のみを使用している．

まず，インメモリデータベースによる共有を用いず，シャッフリングを用いた際の実験結果を図 5 に示す．並列度を上げることでスループットが向上するが，その増加は 24 程度で止まっている．ただし，今回の実験では単純な処理しか実行していないため実行時間が極めて短く，今後時間窓などを組み合わせたより複雑な処理を通じて傾向を調査する必要がある．また，キーの偏りを変化させることでスループットは減少している．偏りが 1.4 以上の場合は並列度 8 の時点でスループットの増加が止まっており，キーのグループ化後の処理が上手く並列に動作していないことが確認できる．

一方で，インメモリデータベースを用いて内部状態を共有した場合の実験結果を図 1 に示す．並列度をあげた際に 24 並列程度でスループットの増加が頭打ちとなるのは同様だが，一方

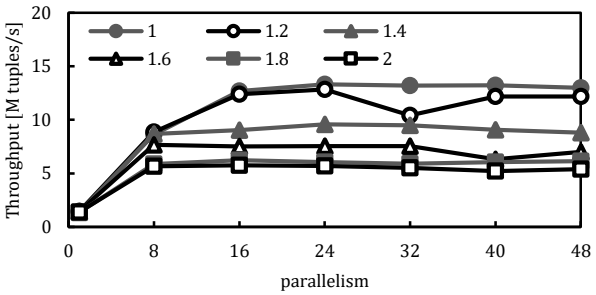

図 5　従来方式におけるキーの偏りに応じたスループット

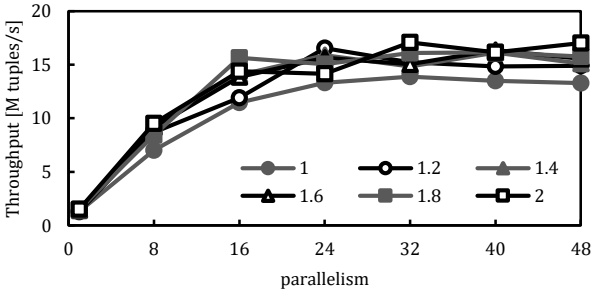

図 6　提案方式におけるキーの偏りに応じたスループット

でキーに偏りを与えた際のスループットは改善している．むしろ，偏りがない場合よりもスループットが向上した．この点に関しては，キーに偏りが生まれることで内部状態として生成，維持しなければならないインスタンスの数が減少したことが原因だと推測できる．

7　関 連 研 究

代表的な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 [1] 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共に，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 [2] 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 など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．

8　お わ り に

本稿では，分散並列ストリーム処理において，シャッフリングを行わずにデータベースを用いて内部状態を共有，マージする方針を提案した．シャッフリングによる障害発生時の影響範囲拡大を抑制するとともに，入力ストリームに偏りが発生した際の処理性能の維持について検討し，試作によってその性能を評価した．

今後は，データベースへの読み書きを効率化するためのスケジューリング，ストリーム同士の結合を効率的に行うための内部状態の共有方法，耐障害性を保証するための具体的なデータベースとの連携方法，及び障害からの回復プロセスについて検討する予定である．

謝　　辞

本研究は JSPS 科研費（JP16H01722，JP19K21530）の助成及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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] T. Akidau, S. Chernyak, and R. Lax, *Streaming Systems: The What, Where, When, and how of Large-scale Data Processing*. O’Reilly Media, Incorporated, 2018.

[9] J. Traub, P. M. Grulich, A. Rodriguez Cuellar, S. Bress, A. Katsifodimos, T. Rabl, and V. Markl, “Scotty: Efficient window aggregation

for out-of-order stream processing," in *Proc. ICDE*, pp. 1300–1303, 2018.

[10] J. Traub, P. Grulich, A. R. Cuéllar, S. Breß, A. Katsifodimos, T. Rabl, and V. Markl, "Efficient window aggregation with general stream slicing," in *Proc. EDBT*, 2019.

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