D-009

並列データストリーム処理システムにおける内部状態共有手法の検討

Internal State Sharing Method in Parallel Data Stream Processing System

徳増 直紀 Naoki Tokumasu
杉浦 健人 Kento Sugiura
石川 佳治 Yoshiharu ishikawa
陸 可鏡 Kejing Lu

1 はじめに

意思決定の高速化のために，データ解析におけるストリーム処理に基づいた遅延削減は広く行われている．従来のバッチ処理では一定期間蓄積したデータをまとめて処理するため，最初のタプルを取得してから処理の完了までに一定の遅延が発生する．一方ストリーム処理はデータが貯まるのを待たず，タプル一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，バッチ処理に対して全体的な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．

一方で，Flink [1] や Samza [2] など既存のストリーム処理システム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課題がある [3]．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，大別するとステートレス処理とステートフル処理に分けられる．ステートレス処理は選択や射影のように入力タプルに依存しない処理のため並列化が容易であるが，ステートフル処理は集約などタプルが持つキーに依存し単純には並列化できない．そのため，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（キーのグルーピング）を行い，あるキーに対するステートフル処理を一つのスレッドが担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 [3]．

また，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ある．既存システムはチェックポイント等を用いてある時点の内部状態をバックアップしており，障害発生時にはそれらを用いてロールバックを行う．内部状態には入力ストリームの読み出し状態も保持されているため，前段のメッセージキューからタプルを再送，再処理することで障害発生時の状態へと復帰する．ここで問題となるのは，タプル再送時のシャッフリングである．入力時点ではキーはグルーピングされていないため，メッセージキューからは全てのタプルを再送する必要があり，結果全てのタプルに対してステートレス・ステートフル両処理が行われる．つまり，障害が発生していないタスク含めて全てのタスクを一括してロールバックする必要がある．しかし，例えば障害が発生していないタスクの内部状態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範囲が拡大していると言える．

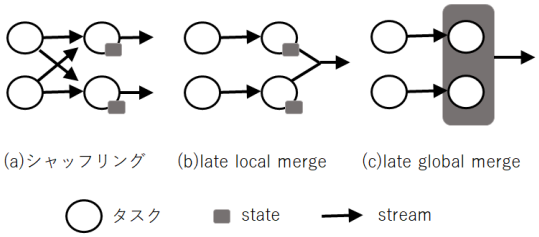

図 1 シャッフリングと late local/global merge

そこで，本研究ではデータベース技術を用いて内部状態を共有し，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検討する．特に，本稿では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

本章では，集約処理における内部状態の共有手法として Zeuch ら [3] の提案した late merge について紹介する．

まず，既存の OSS では図 1 (a) のように内部状態は共有されず，タプルはキーに基づいてシャッフリングされ各ステーフル処理の実行スレッドに分配される．これは Hadoop や Spark など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おり，十分なキー（ないしそのパーティション）さえ確保できれば優れた並列性が得られる．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ったり，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取るキューがボトルネックとなる．

上述の問題に対し，Zeuch らは late local/global merge と呼ぶ後段での集約処理のマージを提案した．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処理を行う．一方 late global merge では各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++ および 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．

3 提案手法

前述した 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

名古屋大学大学院情報学研究科 Graduate School of Informatics, Nagoya University

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，並列で構築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．既存の分散並列ストリーム処理システムである Apache Flink [1] の拡張としての実装を想定しており，基本的なアーキテクチャは同様のものを用いる．

まず，マージ方式としては 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となる．しかし，ストリーム処理は結合律などが成り立つ差分更新可能な処理が大半であり，完全な同期はその目的に対し過剰である．そこで，late local merge によって後段での非同期なマージを検討する．late local merge は前述のように結合に対し問題を持つが，本研究では Jacques-Silva ら [4] の提案した結合手法を使用しこの問題に対応する．つまり，ストリームをタプルの生成頻度が高いストリーム（main stream）と生成頻度が低い参照用のストリーム（join stream）に分けて考える．タプルの生成頻度が低いストリームを全ノードに対しブロードキャストし，各ノードでは結合用の情報（ハッシュテーブルなど）を独立に保持する．また，結合用の情報は内部状態としては永続化せず，障害発生時には必要な参照用のストリームを全再送しハッシュテーブルなどを再構築する．

以下では，図 2 に基づき提案手法における処理の概要について述べる．ただし，以下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．まず，処理に結合が含まれる場合，各スレッドで参照用のストリームを受け取り結合用のハッシュテーブルを構築する．タプルの生成頻度や one-by-one 方式ないしマイクロバッチ方式の違いに応じて，各スレッド独立にハッシュテーブルを構築する方法と全スレッドで協調して構築する方法 [5] とが考えられる．次に，参照用ストリームのウォータマークが十分に先行した時点で，主のストリームからタプルを受け取り結合など全てのステートフル処理を一括で行う．各スレッドでのステートフル処理の結果はそれぞれローカルのバッファに蓄積し，専用のスレッドによって非同期にローカルのデータベースに格納する．なお，スレッドローカルのバッファは Zeuch ら [3] のようにダブルバッファなどを使用し，非同期でのデータベース書き込みが行われている間もストリーム処理は継続させる．また，ローカルのデータストアには RocksDB などの LSM（log-structured merge）木に基づくものの使用を想定しており，永続化や同時実行制御，圧縮などの実装は必要最低限とする．最後に，主となるストリームのウォータマークが十分に大きくなったとき，出力可能な時間窓の集約結果を後続のメッセージキューや分散データベースに送る．

提案する処理方式の利点として，キーの偏りによる性能の劣化の抑制，および障害発生時の影響範囲削減が挙げられる．提案手法では結合用の参照ストリームはブロードキャストする想定であり，主となるストリームは全てのノード・スレッドに対して独立に送られ，キーに偏りが生じた場合も各スレッドで並列に処理される．ただし，後段のマージ処理は追加されているため，具体的な性能への影響は実装により調査する必要がある．また，提案手法では各スレッドは互いに独立に動作しており，後段のマージ段階でのみ内部状態は干渉する．つまり，ある一つのスレッドないし一台のノードで障害が発生した場合もロールバックが必要なのは障害が発生した箇所のみとなる．

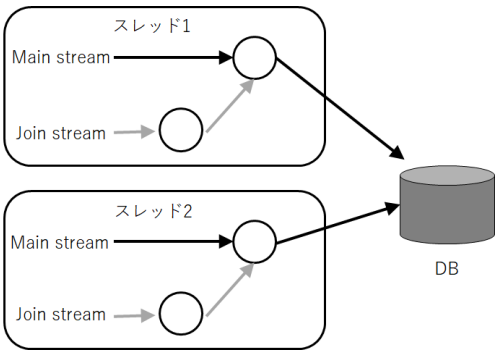

図 2　提案データフロー

4　終わりに

本稿では，並列ストリーム処理システムでシャッフリングを行わず，内部状態を共有，マージする手法を提案した．提案手法は Zeuch ら [3] の提案した late merge に基づいており，システムとしての現実的な実装を検討した．今後は本稿で述べた内容をより詳細に検討すると共に，既存 OSS である Apache Flink 上での提案手法の実装を行う予定である．

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．

参考文献

[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] 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.

[4] G. Jacques-Silva, R. Lei, L. Cheng, G. J. Chen, K. Ching, T. Hu, Y. Mei, K. Wilfong, R. Shetty, S. Yilmaz, A. Banerjee, B. Heintz, S. Iyer, and A. Jaiswal, “Providing streaming joins as a service at Facebook,” *PVLDB*, vol. 11, no. 12, pp. 1809–1821, 2018.

[5] V. Leis, P. Boncz, A. Kemper, and T. Neumann, “Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age,” in *Proc. SIGMOD*, pp. 743–754, 2014.