大域的一貫性を保証する自律分散型データ統合技術の性能分析

吉田 凌河† 伊藤 竜一† 肖 川† 鬼塚 真†

† 大阪大学大学院情報科学研究科 〒 565–0871 大阪府吹田市山田丘 1-5

E-mail: †{yoshida.ryoga,ito.ryuichi,chuanx,onizuka}@ist.osaka-u.ac.jp

あらまし データ統合には，大域的なコンポーネントに基づく中央集権型のシステムとピアに基づく自律分散型のシステムがある．自律分散型で大域的な一貫性を保証するシステムとして Dejima が挙げられる．Dejima のロッキングプロトコルとして，2 相ロック，保守的な 2 相ロック，適応的 2 相ロックが実装されている．保守的な 2 相ロックでは，特定の状況において性能が大きく下がるという特徴がある．本研究では，ライドシェアリングを想定した環境において，その性能がどれほど下がり得るのか，また Dejima の割合を調査する．実験の結果，保守的な 2 相ロックは通常の 2 相ロックを用いた場合に比べて 50%程度性能が落ちることが確認できた．Dejima における処理時間の割合では，ビューの更新に一番時間がかかっていることが確認できた．

キーワード データ統合，分散トランザクション制御，双方向変換

1 はじめに

大企業や大きな組織において，それぞれの部門でデータを管理し，他と連携することなく独自のアプリケーションやデータベースシステムを設計・構築していることは珍しくない．しかし，大企業の合併や買収，病院と薬局，ホテルと航空会社など，それらのシステムを簡単に連携させることが出来れば便利であり，その需要は大きい [1]．また，複数の研究者が独立してデータセットを作成する大規模な科学プロジェクトや，それぞれが独自のデータソースを持つ政府機関の間でのデータ統合などにおいても大変重要な役割を果たす [2]．このようなデータ統合を実現するには，データベース間のスキーマを一致させる必要があったりするなど問題が多く，多くの研究が行われている [3]．

データ統合は主にグローバルスキーマに基づく中央集権型のものとピアに基づく自律分散型のアーキテクチャに分類される．前者は主に企業での利用を想定しており，単一のグローバルスキーマを持つ大域的なコンポーネントが他のデータベースサーバーからの通信を受け付け，データを管理する [4]?[6]．一方後者は，単一のグローバルスキーマを決定することが困難な場合を想定したもので，大域的なコンポーネントを設けず，2 つのピア間での通信を元にデータをカスケード的に伝搬していくものである．しかし，実際には，この 2 つに分類できない両者のシステムを組み合わせたようなシステムも存在する．

本研究のターゲットはライドシェアリングである．ライドシェアリングとは，Uber Taxi や DiDi，Go のように運転手とユーザーを繋ぐサービスである．Uber や DiDi は複数の企業の車両情報をまとめて持っており，このような企業の連合体はアライアンスと呼ばれる．アライアンスが一括して車両情報を持ち，配車リクエストにも答えることで，ユーザーからすると，好みのアライアンスに問い合わせることで簡単に色々な企業の空き車両を見つけることができるといったメリットがあり，配車サービスを提供するプロバイダからすると，自社の車両を知ってもらう機会や使ってもらう機会が増えるといったメリットがある．

ライドシェアリングのサービスを提供するために，アライアンスは複数のサービスプロバイダのデータ統合をする必要がある．多くのアライアンスは中央集権型のアーキテクチャを使ってデータ統合を行っているが，いくつか問題がある．まず，それぞれのアライアンスやプロバイダのデータベーススキーマが一致していないため，両社のデータベースを単純に同期することによるデータ統合はできない．そのため通常プロバイダは，アライアンスから送られてくるデータに従って自身のデータベースを更新するプログラムを書く必要があり，複数のアライアンスに所属している場合，そのプログラムは複雑になる可能性がある．また，中央集権型アーキテクチャは大域的なコンポーネントに依存しており，そこに障害が発生した場合，システム全体が使用不可能になる可能性がある．一方，自律分散型アーキテクチャは，大域的なコンポーネントに依存せず，ピア同士の通信で成り立っているため，システムの一部に障害が発生したとしてもシステム全体が運用不可能になるようなことは少ないなどの理由から，ライドシェアリングにおける自律分散型アーキテクチャの利用が提案されている [7]．

しかし，Piazza [8], [9] や ORCHESTRA [10], [11] のように自律分散型アーキテクチャの多くが，システム全体でのデータの一貫性を意識しない限定的な利用を想定しており，各ピア内だけのデータの一貫性は保証するものの全てのピアにわたった大域的なデータの一貫性は保証しない．これはライドシェアリングなどのより一般的な利用を考えた場合，大きな問題となる．そこで，大域的な一貫性を保証するデータ統合アーキテクチャとして，Dejima が提案されており [1], [12]，プロトタイプも実装されている [13]．

本研究では，Dejima のライドシェアリングを想定した環境における性能分析および高速化である．

2 Dejima

本章では，Dejima アーキテクチャについて説明する．

2.1 概　　要

Dejima アーキテクチャは以下の 4 つの構成要素から成る．

- ピア
- ベーステーブル
- Dejima テーブル
- Dejima グループ

ピアはサービスプロバイダなどデータを持つ主体を表し，それぞれ固有のローカルデータベースを管理する．このアーキテクチャではデータを共有したいピア同士で Dejima グループを形成する．各ピアは参加する Dejima グループごとに Dejima テーブルを定義し，同一の Dejima グループに属する他のピアと対応する Dejima テーブルを共有する．

Dejima テーブルは，データベース内にあるベーステーブルから導出されるビューである．ビューであるが，更新可能であり，Dejima テーブルの更新に対応して導出元のベーステーブルを適切に更新することができる．この更新戦略は双方向変換言語 [14], [15] を用いて定義される．つまり，ベーステーブルと Dejima テーブルは双方向の変換が可能になっている．

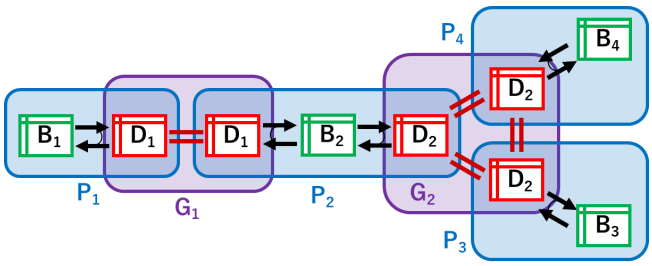

図 1　Dejima アーキテクチャの例

図 1 は Dejima アーキテクチャの一例であり，4 つのピア $P_1, \cdots, P_4$ とそれに対応するベーステーブル $B_1, \cdots, B_4$，2 つの Dejima グループ $G1, G2$ とそれに対応する Dejima テーブル D_1, D_2 が表されている．同一の Dejima グループに属するピアの対応する Dejima テーブルは同期しているため，ここで例えば P_1 の Dejima テーブル D_1 と P_2 の Dejima テーブル D_1 は全く同じ内容になっており．どちらか一方が更新されても，すぐに更新内容がもう一方に伝わり，同一の状態が保持される．

2.2 更 新 伝 搬

Dejima アーキテクチャにおける更新伝搬は次のように行われる．

（1） あるピアにおいてベーステーブルが更新される．

（2） (1) の更新によって，ベーステーブルのビューである Dejima テーブルが更新される．

（3） (2) の更新内容を，同じ Dejima テーブルを保持する他のピアに通知する．

（4） (3) で通知された更新内容をもとに自身の Dejima テーブルを更新し同期する．

（5） (4) で更新した内容をもとに，双方向変換によりベーステーブルが更新される．

（6） 以降，Dejima テーブルの更新が発生しなくなるまで (2)〜(5) を繰り返す．

2.3 ロッキングプロトコル

Dejima では，複数のピアにおいてトランザクションが並列実行され，更新が伝搬する．また，大域的なデータの一貫性を保証する必要があるため，排他制御が必要となる．Dejima では，排他制御を実現するロッキングプロトコルとして 2 相ロック [16]?[19] と保守的な 2 相ロック，適応的 2 相ロック [20] が採用されている．

2.3.1 2 相ロック

2 相ロックは，順にロックを取得していく成長相と取得したロックを順に解放していく縮退相から成る．並列に実行されるすべてのトランザクションが 2 相ロックに従うことで，すべてのデータが直列化され，他のトランザクションにデータを書き換えられることなどによるデータの不整合を避けることができる．

2.3.2 保守的な 2 相ロック

保守的な 2 相ロックは，2 相ロックにおいて，トランザクションの開始時にロックの獲得をすべて終わらせる方式である．つまり，成長相がトランザクション実行前に開始し一瞬で終わるような方式である．これにより，トランザクション実行中にロックを新たに獲得することがなくなるため，デッドロックを原因とするトランザクションのアボートを防ぐことができる．

しかし，トランザクション実行前に全てのロックを取得する必要があることから，トランザクション処理に条件分岐が含まれるなど実行時にアクセスするレコードが事前に分からないトランザクションについては，実行前にアクセスされるレコードを特定できないため，基本的に実行できない．ただし，更新が伝搬する可能性のあるレコード全てにロックをかけることで実行することは可能であるが，実際に更新が伝搬しないレコードにもロックをかけないといけないというデメリットがある．

保守的な 2 相ロックでは，トランザクション実行時の更新伝搬によりロックするべきレコードを事前に特定する必要があるが，Dejima のような自律分散型アーキテクチャにおいて，三宅らの研究 [20] では，そのロックするべきレコードの集合を Family Record Set と呼んでいる．

具体的には，あるピアのベーステーブルに新しいレコードが挿入されたとき，直接挿入されたレコードを Original Record，その挿入の更新伝搬により他のピアで生成されたレコードを Derived Record と定義し，この Original Record とそれに対応する Derived Record の集合を Family Record Set と定義する．

各 Family Record Set には固有の識別子が与えられ，同じ Family Record Set に属するレコードは同一の識別子を持つ．この識別子は，システム全体を通して一意であればなんでも良いが，実装上は Original Record のピア ID，ベーステーブル

ID，レコード ID の組み合わせで表現している．識別子は，各レコードの Lineage カラムに追加されるが，1 つである必要はなく複数の識別子を保持することができる．これを応用することで，JOIN クエリにより定義された Dejima テーブルを含んでいたとしても，ロックをかけるべきレコードを適切に特定することが可能である．

2.3.3 適用的 2 相ロック

通常の 2 相ロックと保守的な 2 相ロックにはそれぞれメリットデメリットがある．保守的な 2 相ロックを採用することで，トランザクション実行中のデッドロックによるアボートを防ぐことができる．一方，デッドロックが発生しにくい競合率の低い状況においては，事前にロックをすべきレコードを特定する処理や通信などがオーバーヘッドとなり，スループットが下がる．

適応的 2 相ロックはこの問題を解決するために提案されたロック方式で，システムの負荷に応じて通常の 2 相ロックと保守的な 2 相ロックを適応的に切り替えることでスループットの低下を防ぐ．この手法では，通常の 2 相ロックと保守的な 2 相ロックの内どちらかの方式で動作し，定期的に両者のスループットを計測し，現在実行中の手法がもう一方の手法に比べてスループットが低かった場合，ロック方式をもう一方の手法に切り替える．

3 実験設定

ライドシェアリングの実験をするにあたり，Provider-to-Provider のネットワークトポロジと Provider-to-Alliance のネットワークトポロジの 2 種類のトポロジを用意する．この 2 種類のトポロジにおいてライドシェアリングを想定したベンチマークをロッキングプロトコルを変え実行しスループットとアボート数を比較することで，ロッキングプロトコルによる特徴を調べる．また，ベンチマークを実行する際のそれぞれの処理にかかった時間の内訳を可視化することで，ボトルネックの解析も行う．

3.1 Provider-to-Provider のトポロジ

今回実験に使用するトポロジの内，1 つ目が，図 2 で表されているように，各ピアを配車サービスのプロバイダとして，調停役を設けず，プロバイダ同士でデータのやり取りをするトポロジである．このトポロジをここでは Provider-to-Provider のトポロジと呼ぶことにする．

3.1.1 スキーマ

各プロバイダのベーステーブルの定義と例，Dejima テーブルの例をそれぞれ表 1，図 3，図 4 に示す．

車両 ID は主キーになっており重複は許されない．リクエスト ID は 0，すなわち R=0 の場合，配車リクエストを受けていない状態を表す．実際にはピア数やレコード数，カラム数がもっと必要になるが，ここでは簡単のため省略している．また現在地や目的地も単に整数で表現している．

プロバイダは，自身のどの車両を共有させるかを選択出来るべきであるが Dejima アーキテクチャでは，その選択は簡単に実現できる．この例の場合，true または false の値を取る d{n}_{m} のカラムを追加することで実現している．具体的には，図 3 のプロバイダ 2 において，カラム d1_2，d2_3，d2_4 はそれぞれプロバイダ 2 が持つ Dejima テーブルに対するカラムで，この値が true なら，ベーステーブルの更新をその Dejima テーブルへ伝搬させる．false なら，伝搬させないといった設定項目のような働きをしている．

このようにプロバイダ 1 とプロバイダ 2，Dejima テーブル d1_2 のようにそれぞれのピアや Dejima テーブルは，双方向変換を適切に定義する限り異なるスキーマを持つことができる．これにより，それぞれのピアの自律性が高まる．

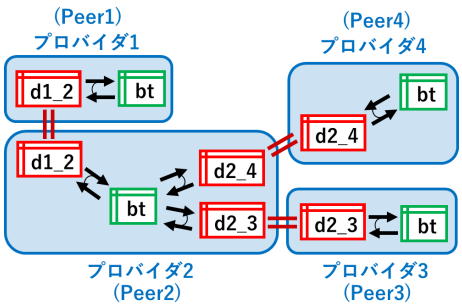

図 2 プロバイダだけから成るトポロジ

プロバイダ1のbt

V	L	D	R	d1_2	Lineage
1	8377	8377	0	true	Peer1-bt-1
2	7962	7962	0	true	Peer1-bt-2
3	4970	4970	0	true	Peer2-bt-3

プロバイダ2のbt

V	L	D	R	d1_2	d2_3	d2_4	Lineage
1	8377	8377	0	true	false	false	Peer1-bt-1
2	7962	7962	0	true	false	false	Peer1-bt-2
3	4970	4970	0	true	true	true	Peer2-bt-3
4	5867	9559	1	false	true	false	Peer3-bt-4
5	357	357	0	false	false	true	Peer4-bt-5
6	1543	2579	1	false	false	true	Peer4-bt-6

プロバイダ3のbt

V	L	D	R	d2_3	Lineage
3	4970	4970	0	true	Peer2-bt-3
4	5867	9559	1	true	Peer3-bt-4

プロバイダ4のbt

V	L	D	R	d2_4	Lineage
3	4970	4970	0	true	Peer2-bt-3
5	357	357	0	true	Peer4-bt-5
6	1543	2579	1	true	Peer4-bt-6

図 3 図 2 におけるベーステーブルの例

3.1.2 データの更新伝搬

具体的に図 3 について，いくつかのデータの動きを説明する．プロバイダ 1 で車両 ID が 1 のレコード，すなわち V=1 のレ

d1_2

V	L	D	R	Lineage
1	8377	8377	0	Peer1-bt-1
2	7962	7962	0	Peer1-bt-2
3	4970	4970	0	Peer2-bt-3

d2_3

V	L	D	R	Lineage
3	4970	4970	0	Peer2-bt-3
4	5867	9559	1	Peer3-bt-4

d2_4

V	L	D	R	Lineage
3	4970	4970	0	Peer2-bt-3
5	357	357	0	Peer4-bt-5
6	1543	2579	1	Peer4-bt-6

図 4　図 2 における Dejima テーブルの例

コードを見てみると，R=0，つまり配車リクエストを受けていないことが分かる．配車依頼を受けていないときは現在地 L と目的地 D が一致するようにしており，この場合 L=D=8377 となっている．d1_2 のカラムを見てみると，値が true となっていることから，このレコードにおけるデータ更新は Dejima テーブル d1_2 に伝搬することが分かる．このレコードは隣のピアであるプロバイダ 2 にだけ共有されている．それより遠くのピアには伝搬されないように，プロバイダ 2 での d2_3 と d2_3 は false になっている．

表 1　図 2 におけるベーステーブルの定義

カラム名	説明	型
V	車両 ID	int
L	現在地	int
D	目的地	int
R	リクエスト ID	int
d{n}_{m}	ビュー d{n}_{m} とのデータ共有の有無	string
Lineage	所属する Dejima グループの集合	string

ここで例えば，プロバイダ 2 において V=1 の車両に配車リクエストを行いたいとき，カラム R とカラム D にそれぞれリクエスト ID，目的地を設定する．データが更新されると，BIRDS がその更新を検知し，d{n}_{m} が true になっている Dejima テーブル，ここでは d1_2 に更新が伝搬する．d1_2 はそのデータ更新を受けてプロバイダ 1 の V=1 のレコードも同じように更新する．

ある車両データは Dejima テーブルを介して隣のピアにカスケード的に伝わっていくが，あまりにも離れたプロバイダにまで車両データを共有したとしても，そのプロバイダから配車依頼を受ける可能性は低いうえ，それぞれのプロバイダが保持しなければならないデータ量も増えるため，隣何ピアまでデータを共有するかを選択出来るようにしている．

3.1.3　ベンチマーク

ユーザは配車依頼をしたいとき，最寄りのプロバイダに問い合わせをする．問い合わせを受けたプロバイダは，自身のデータベースで車両情報を確認し，基本的に自社の車両を優先して配車する．しかし，他のプロバイダのデータも持っているため，自社の車両が何らかの理由で配車できない場合などに，そのプロバイダが保持する他のプロバイダの車両情報を元にそのプロバイダに配車依頼をする．また，車両は定期的に現在地をプロバイダに送信することで，より適切な配車の提供などを実現することとする．

そこで本実験では，以上の設定を想定して次の 3 種類のトランザクションを用意した．

- Update: 車両の現在地 L を更新するトランザクション．
- Read: あるレコードを読み出すトランザクション．
- Request: R と D を変更する配車リクエストのトランザクション．

Update トランザクションにおいて，車両の現在地はそれほどリアルタイム性が求められないことがある．そのような場合，複数の車両の位置情報をまとめて更新する可能性があるため，このトランザクションは一度に複数レコードを処理することが可能である．また，Read トランザクションは，プロバイダやユーザが車両の情報を確認するために用いられるトランザクションであるが，このトランザクションは性質上 1 車両ずつではなく，複数の車両情報をまとめて参照する場合が多いと考えられるため，このトランザクションも一度に複数レコードを処理することが可能である．

実際には車両の追加や削除，d{n}_{m} を変更することによる車両の共有の変更処理なども存在するが，今回は簡単のため省略した．また，デッドロックを解消する方式として No-Wait 方式を採用した．

今回のベンチマークでは同時並列に各ピアにおいて次の手順で処理を実行する．

（1） 自身が持つ全てのレコードの内，複数のレコードを同確率で選択する．

（2） 選択したレコード群に対し，Update 70%，Read 25%，Request 5%の確率でトランザクションを選択する．

（3） 選択したレコード群に対し，選択したトランザクションを発行する．

（4） トランザクション処理が終了すれば，すぐさま (1) に戻る．

3.2　Provider-to-Alliance のトポロジ

Provider-to-Provider のトポロジでは，同じレコードが周囲のプロバイダにも共有され，無駄が大きいことや，更新が末端まで伝わる時間が長い，データ更新の際のコストが大きいなどの問題がある．そこでアライアンスを設け，中央集権型のアーキテクチャの要素も取り入れたトポロジ [7] を考える．

このトポロジは，図 5 のように調停役であるアライアンスを設け，中央集権型アーキテクチャのような使い方をする方式である．プロバイダは所属したいアライアンスを選択し，車両情報を共有する．アライアンスは共有された車両情報を元にユーザのリクエストを受ける．

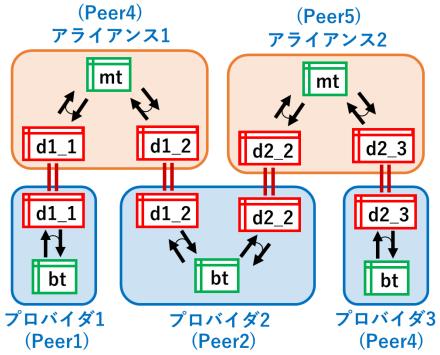

図 5　プロバイダとアライアンスから成るトポロジ

それぞれのベーステーブルと Dejima テーブルの例を図 6, 図 7 にそれぞれ示す．このトポロジでも前述のトポロジと同じように，どのアライアンスと車両情報を共有したいかを車両ごとに選択出来るようになっている．例えば AL1 はアライアンス 1 とデータを共有するかどうかのカラムである．アライアンスのベーステーブルにある P カラムは，どのプロバイダのデータかを表すものであり，P=1 ならプロバイダ 1 のデータであることを表す．このトポロジでは，Provider-to-Provider のように隣のピアにどんどん伝搬していくことはなく，アライアンスに伝搬したあとそれ以上伝搬することはない．

アライアンス1のmt

V	L	D	R	P	Lineage
1	8377	8377	0	1	Peer1-bt-1
3	4970	4970	0	2	Peer2-bt-3
5	357	357	0	2	Peer2-bt-5

アライアンス2のmt

V	L	D	R	P	Lineage
4	5867	9559	1	2	Peer2-bt-4
6	1543	2579	1	3	Peer3-bt-6

プロバイダ1のbt

V	L	D	R	AL1	Lineage
1	8377	8377	0	true	Peer1-bt-1
2	7962	7962	0	false	Peer1-bt-2

プロバイダ2のbt

V	L	D	R	AL1	AL2	Lineage
3	4970	4970	0	true	false	Peer2-bt-3
4	5867	9559	1	false	true	Peer2-bt-4
5	357	357	0	true	true	Peer2-bt-5

プロバイダ3のbt

V	L	D	R	AL2	Lineage
6	1543	2579	1	true	Peer3-bt-6

図 6　図 5 におけるベーステーブルの例

このトポロジでのベンチマークは Provider-to-Provider のときとトランザクションの選択確率が次のようになる以外は同じである．

- アライアンス: Read 80%，Request 20%
- プロバイダ: Read 20%，Update 80%

d1_1

V	L	D	R	Lineage
1	8377	8377	0	Peer1-bt-1

d1_2

V	L	D	R	Lineage
3	4970	4970	0	Peer2-bt-3
5	357	357	0	Peer2-bt-5

d2_2

V	L	D	R	P	Lineage
4	5867	9559	1	2	Peer2-bt-4

d2_3

V	L	D	R	P	Lineage
6	1543	2579	1	3	Peer3-bt-6

図 7　図 5 における Dejima テーブルの例

3.3　処理時間の内訳

Dejima の性能分析やボトルネックの調査のために，今回作成したベンチマーク実行の際にどの処理にどれだけの時間がかかったのかの内訳を調べる実験を行った．ただし，ここではアボートとなったトランザクションは対象とせず，最終的にコミットしたトランザクションのみを調査対象とする．

それぞれの処理の計測には Python の time.perf_counter 関数を使い，処理前と処理後の時間を計測し，差分を取ることで求めた．

今回の調査で対象とした処理とその内容を表 2 に示す．

表 2　調査対象とした処理とその内容

ビューの更新	更新可能ビューを更新する処理
ビューへの伝搬	ベーステーブル更新をビューへ伝搬する処理
ベーステーブルの更新	ベーステーブルを更新する処理
通信	ピア間で制御用の通信をする処理
ロック	ロックを獲得する処理
xid の取得	トランザクション ID の取得処理

4　実験結果と考察

4.1　Provider-to-Provider のトポロジ

今回作成したライドシェアリング用のベンチマークにおいて，次の条件設定で 1 トランザクションあたりに選択するレコード数を 1~10 で変化させた場合の各手法におけるアボート数とスループットの相対値をそれぞれ図 8，図 9 に示す．ここでスループットとは，1 秒あたりのコミット数のことである．

- ピア数: 15
- 各ピアに追加するレコード数: 10
- 実行時間: 1200 秒
- 伝搬範囲: 3 ホップ

図 8 より，1 トランザクションあたりに選択するレコード数が増えると，どの手法においてもアボート数が増えることが分

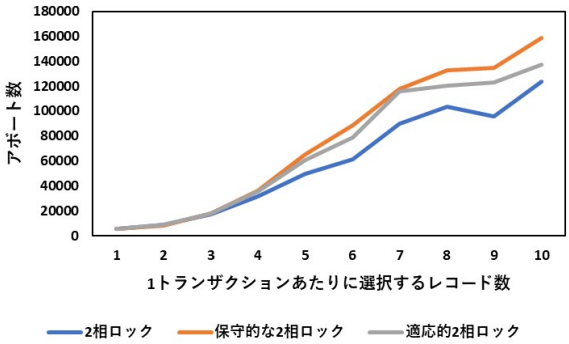

図 8　Provider-to-Provider において，1 トランザクションあたりに選択するレコード数を変化させたときのアボート数

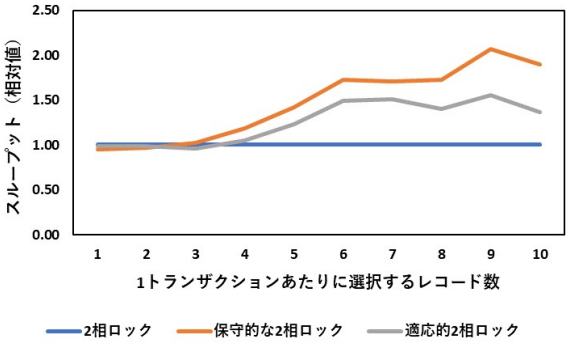

図 9　Provider-to-Provider において，1 トランザクションあたりに選択するレコード数を変化させたときのスループット

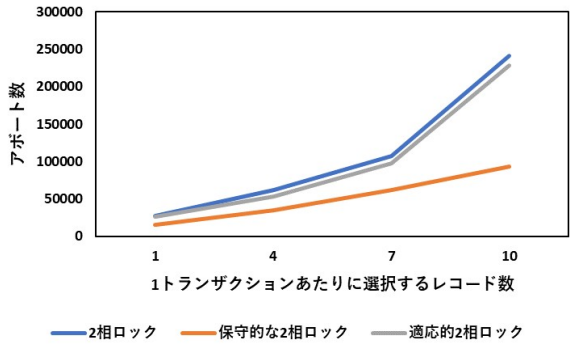

図 10　Provider-to-Alliance において，1 トランザクションあたりに選択するレコード数を変化させたときのアボート数

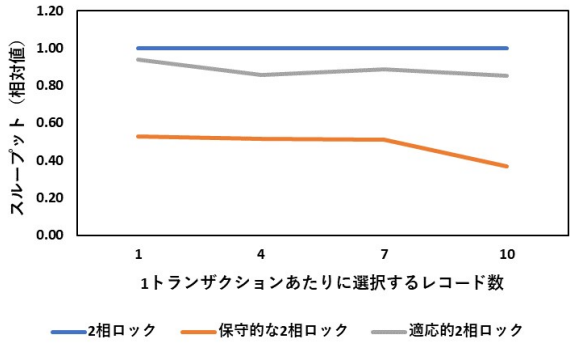

図 11　Provider-to-Alliance において，1 トランザクションあたりに選択するレコード数を変化させたときのスループット

かる．これは 1 トランザクションあたりに選択するレコード数が増えたことで，他のトランザクションとのコンフリクトが発生しやすくなったからだと考えられる．

一方，図 9 より 1 トランザクションあたりに選択するレコード数が増えるにつれて，保守的な 2 相ロックが通常の 2 相ロックに比べてスループットがどんどん良くなっている．これは，1 トランザクションあたりに選択するレコード数が増えるにつれアボート数も増えることにより，アボート時のコストが小さい保守的な 2 相ロックが有利になったからだと考えられる．

また，適応的 2 相ロックは，定期的に両者のスループットをテストし，より良い手法に適宜切り替えるため，2 相ロックより良い性能を達成している．

4.2　Provider-to-Alliance のトポロジ

次の条件設定で 1 トランザクションあたりに選択するレコード数を 1~10 で変化させた場合の各手法におけるアボート数とスループットの相対値をそれぞれ図 10，図 11 に示す．

- アライアンス数: 7
- プロバイダ数: 8
- 各プロバイダに追加するレコード数: 10
- 実行時間: 1200 秒

図 10 より，1 トランザクションあたりに選択するレコード数が増えることで，Provider-to-Provider のときと同じようにトランザクションのコンフリクトが増え，アボート数が増えることが読み取れる．

しかし，図 11 から，今回の実験ではアボート数が少ない場合でも多い場合でも，保守的な 2 相ロックの方が性能が悪いことが読み取れる．このトポロジでは，プロバイダやアライアンスでのデータ更新が高々隣のピアにまでにしか伝搬しないのに，保守的な 2 相ロックでは全ピアに対して事前にロックをとっており，そのオーバーヘッドが大きいからだと考えられる．この実験での 2 相ロックの結果は，保守的な 2 相ロックにおいて適切な事前ロックを行った場合と本質的な差はない．そのため，保守的な 2 相ロックにおいて，適切な事前ロックを取ることが出来れば，今回のような実験設定において 50%程度の性能改善が見込めることが示せた．

4.3　処理時間の内訳

Provider-to-Alliance のトポロジにおいて，次の条件設定で実験したときのコミットしたトランザクションにおける処理時間の内訳を図 12，図 13 に示す．図 12 が 2 相ロックにおける結果，図 13 が保守的な 2 相ロックにおける結果である．

- アライアンス数: 7
- プロバイダ数: 8
- 各プロバイダに追加するレコード数: 10
- 1 トランザクションあたりに選択するレコード数: 5
- 実行時間: 1200 秒

この結果によるとビューの更新に一番時間がかかっているこ

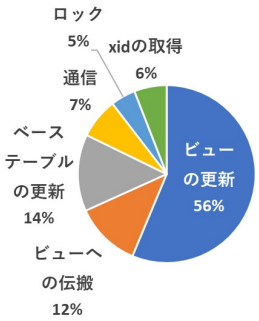

図 12　2 相ロックの結果

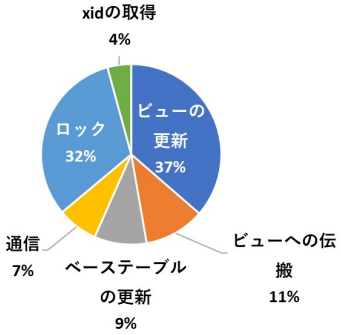

図 13　保守的な 2 相ロックの結果

とが分かる．Dejima アーキテクチャにおいて，ビューの更新は BIRDS により行われているが，BIRDS がビューの更新を行う際，Datalog の定義ファイルに従って導出元のベーステーブルの更新内容を求める．その後その更新情報を元にベーステーブルを更新することにより，ビューも適切に書き変わる．ベーステーブルの更新時間はビューの更新時間に比べて極めて小さいため，主にビューの更新内容からベーステーブルの更新内容を求める処理に時間がかかっていていると考えられる．

また，保守的な 2 相ロックにおいて 2 番目に処理時間がかかっている処理がロックである．これは，本来必要のないピアにまでロックを獲得しにいっているオーバーヘッドが原因であると考えられる．そのため，事前ロック命令を全ピアに送るのではなく，Update Peer Scope と呼ばれる更新が伝搬する可能性があるピアに限定する仕組みを導入したり，HTTP/2 を使用した gRPC [21], [22] を用いることなどにより改善可能だと思われる．

5　ま　と　め

本研究では，自律分散型データ統合システムである Dejima を使って，ライドシェアリングを想定して応用実験を行った．実験では，プロバイダ同士が直接繋がる Provider-to-Provider のトポロジと複数のプロバイダのデータをいくつかのアライアンスがまとめる Provider-to-Alliance のトポロジの 2 つを想定して性能分析を行った．

実験の結果，Provider-to-Provider のトポロジでは，1 トランザクションあたりに選択するレコード数が増え，コンフリクトが起きやすくなると，事前ロックによるトランザクション実行中のアボートを防ぐことで，保守的な 2 相ロックの性能が通常の 2 相ロックに比べて優位になることを確認した．Provider-to-Alliance のトポロジでは，データ更新が伝搬しないピアに対しても事前ロックを取ることで，保守的な 2 相ロックが通常の 2 相ロックより 50%程度スループットが落ちることを確認した．処理時間の内訳から，ビューの更新に最も時間がかかっていること，保守的な 2 相ロックではロック処理にかかる時間の割合が大きいことを確認した．性能分析の結果，高速化すべき点として事前ロックアルゴリズムの改善や通信プロトコルの変更などが挙げられた．

最後に今後の課題について述べる．本実験では，ピア数 15 など小さい規模でしか実験が出来ていない．しかし，1 都道府県内だけでシステムを運用するとしても 1 都道府県内に少なくとも 60~100，多ければ 1000~1200 のタクシー事業者があるため，より実用的なユースケースを考えて，ピア数 100 などより大きい規模での実験が必要であると考えている．

また，保守的な 2 相ロックにおいて，ロックのオーバーヘッドが大きいことから，ロックの獲得を更新が伝搬する可能性があるピアである Update Peer Scope に限定する実装をする必要がある．Update Peer Scope に事前ロック要求を限定する仕組みの実現には，他のピアの双方向変換の定義ファイルを事前に集めておき，述語ロックベースでロック範囲を特定するという方法や，Ultraverse [23] と呼ばれる更新命令に対して影響を受けるクエリ群を特定する技術を使う方法が考えられる．他には，レコードへの書き込みのオーバーヘッドが大きくなりそうではあるが，各レコードごとに過去に伝搬したピア集合を追加で記録しておくことで，更新伝搬する可能性のあるピア集合を限定するという方法も考えられる．

今回は困難性から言及しなかったが，Dejima の一番のボトルネックとなっているのがビューの更新であるため，BIRDS においてビューの更新戦略から高速に動作する SQL ファイルを出力することが出来るようになれば大幅な性能向上が見込めると思われるため取り組む価値は大いにあると考えている．

文　献

[1] Yasuhito Asano, Dennis-Florian Herr, Yasunori Ishihara, Hiroyuki Kato, Keisuke Nakano, Makoto Onizuka, and Yuya Sasaki. Flexible framework for data integration and update propagation: System aspect. In *BigComp*, pp. 1–5. IEEE, 2019.

[2] Alon Y. Halevy, Anand Rajaraman, and Joann J. Ordille. Data integration: The teenage years. In *VLDB*, pp. 9–16. ACM, 2006.

[3] AnHai Doan, Alon Y. Halevy, and Zachary G. Ives. *Principles of Data Integration*. Morgan Kaufmann, 2012.

[4] Maurizio Lenzerini. Data integration: A theoretical perspective. In *PODS*, pp. 233–246. ACM, 2002.

[5] Alon Y. Halevy. Answering queries using views: A survey. *VLDB J.*, Vol. 10, No. 4, pp. 270–294, 2001.

[6] Jeffrey D. Ullman. Information integration using logical views. In *ICDT*, pp. 19–40. Springer, 1997.

[7] Yasuhito Asano, Zhenjiang Hu, Yasunori Ishihara, Hiroyuki Kato, Makoto Onizuka, and Masatoshi Yoshikawa. Control-

ling and sharing distributed data for implementing service alliance. In *BigComp*, pp. 1–4. IEEE, 2019.

[8] Alon Y. Halevy, Zachary G. Ives, Peter Mork, and Igor Tatarinov. Piazza: data management infrastructure for semantic web applications. In *WWW*, pp. 556–567. ACM, 2003.

[9] Alon Y. Halevy, Zachary G. Ives, Jayant Madhavan, Peter Mork, Dan Suciu, and Igor Tatarinov. The piazza peer data management system. *IEEE Trans. Knowl. Data Eng.*, Vol. 16, No. 7, pp. 787–798, 2004.

[10] Grigoris Karvounarakis, Todd J. Green, Zachary G. Ives, and Val Tannen. Collaborative data sharing via update exchange and provenance. *ACM Trans. Database Syst.*, Vol. 38, No. 3, p. 19, 2013.

[11] Zachary G. Ives, Nitin Khandelwal, Aneesh Kapur, and Murat Cakir. ORCHESTRA: rapid, collaborative sharing of dynamic data. In *CIDR*, pp. 107–118, 2005.

[12] Yasuhito Asano, Soichiro Hidaka, Zhenjiang Hu, Yasunori Ishihara, Hiroyuki Kato, Hsiang-Shang Ko, Keisuke Nakano, Makoto Onizuka, Yuya Sasaki, Toshiyuki Shimizu, Van-Dang Tran, Kanae Tsushima, and Masatoshi Yoshikawa. Making view update strategies programmable - toward controlling and sharing distributed data. *CoRR*, Vol. abs/1809.10357, , 2018.

[13] Onizuka Lab. Dejima prototype. `https://github.com/OnizukaLab/dejima-prototype`, 2022.

[14] Van-Dang Tran, Hiroyuki Kato, and Zhenjiang Hu. Programmable view update strategies on relations. *PVLDB*, Vol. 13, No. 5, pp. 726–739, 2020.

[15] Van-Dang Tran, Hiroyuki Kato, and Zhenjiang Hu. Birds – bidirectional transformation for relational view update datalog-based strategies. `https://dangtv.github.io/BIRDS/`, 2020.

[16] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[17] Christos H. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, 1986.

[18] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, Vol. 19, No. 11, pp. 624–633, 1976.

[19] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger. Granularity of locks in a large shared data base. In *VLDB*, pp. 428–451. ACM, 1975.

[20] 三宅康太, 佐々木勇和, 肖川, 鬼塚真. 統合型データベースにおける適応的 2 相ロックに基づく分散トランザクション制御. In *DEIM*, pp. 1–8, 2022.

[21] Xingwei Wang, Hong Zhao, and Jiakeng Zhu. GRPC: A communication cooperation mechanism in distributed systems. *ACM SIGOPS Oper. Syst. Rev.*, Vol. 27, No. 3, pp. 75–86, 1993.

[22] Open Source. grpc. `https://github.com/grpc/grpc`, 2023.

[23] Ronny Ko, Chuan Xiao, Makoto Onizuka, Yihe Huang, and Zhiqiang Lin. Ultraverse: Efficient retroactive operation for attack recovery in database systems and web frameworks. *CoRR*, Vol. abs/2211.05327, , 2022.