コンフリクトフリーなデータ型に基づくステートフルストリーム処理基盤

斎藤　貴文† 　善明　晃由†

† 株式会社サイバーエージェント

あらまし　ストリーム処理においてステートフルなデータを簡潔に処理するための解析基盤 Phalanx を提案する．ストリーム処理ではイベントの到着順序が保障されないため、更新順序を考慮する必要があるステートフルなデータのあつかいが難しい．Phalanx では CRDT のデータモデルに基づいて，ステートフルなストリーム処理を実現する．CRDT で用いられるデータ型は更新の順序を問わず結果整合性を保つことが可能なため，データの到着順序が保証されない状況でのステートフルなデータへの管理を簡素化できる．結果整合性をもつデータ型を利用することで，Phalanx ではイベントを CRDT の操作に対応付ける記述のみでステートフルなデータの変更を実現できる．

キーワード　ストリーム処理,CRDT

1. 導　　入

近年，Web サービスのデータ解析においてストリーム処理は重要視されている．ストリーム処理では入力イベントを逐次実行するため，入力データを時刻で区切った単位でまとめて処理を行うバッチ処理に比べてリアルタイム性の高い処理を行うことが可能である．

ストリーム処理ではイベントが発生した時刻と同じ順序でストリーム処理エンジンに届くという保証はない．実運用におけるストリーム処理のイベントは様々なライブラリやシステムの転送パイプラインを経てストリーム処理エンジンに届けられる．例えばパイプライン中の転送経路を並列化した場合，負荷の不均質が原因で経路間で転送速度が異なり到着順序が入れ替わる可能性がある．

転送順序が保証されないためストリーム処理で扱うことが難しかったデータとしてステートフルデータがある．ステートフルデータとは現在の状態に依存して次の状態が決まるデータである．Web サービスにおけるステートフルデータにはソーシャルゲームのユーザの所持アイテムリスト，Web サイトの PV やセッション数など多岐に渡る．

ストリーム処理でステートフルデータを更新することでリアルタイムで最新の状態を参照できる．しかしステートフルデータの変更は現在の状態に依存し，ストリーム処理ではイベントの順序が保証されないのでステートフルデータの整合性を維持するのが難しい．

本研究ではストリーム処理においてステートフルなデータ処理を簡潔に記述するための解析基盤 Phalanx を開発した．Phalanx では CRDT [1] のデータモデルを参考として更新順序が保証されない場合もステートフルデータの結果整合性を担保するデータ型を用意している．結果整合性を担保するデータ型を利用することで更新の順序を考慮することなく，イベントとステートフルデータへの操作を対応付けるのみでストリーム処理でステートフルデータの扱うことが可能になる．

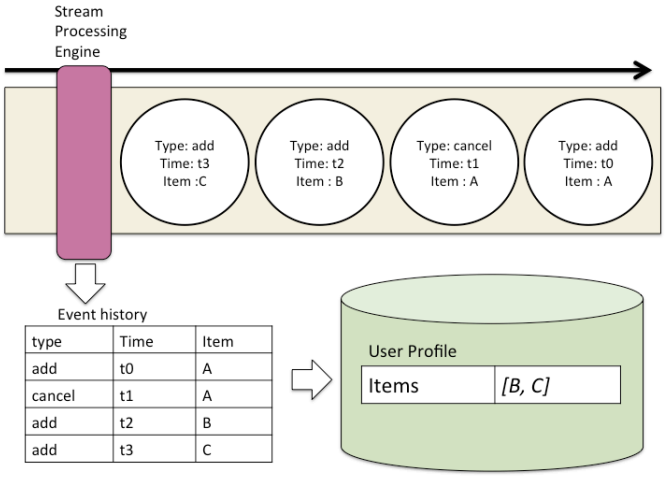

図 1　イベントの順序に変更がないときの例

本論文の構成は以下の通りである．2. 章では背景としてステートフルストリーム処理の課題について説明する．3. 章では Phalanx のシステムについて説明し，4. では Phalanx の実装について説明する．5. 章では Phalanx の適用例について紹介する．6. 章で関連研究について説明し，7. で本研究についてまとめる．

2. 背　　景

2.1 ステートフルストリーム処理について

バッチ処理でステートフルデータの変更を行う場合，実際のイベントの発生時刻とステートフルデータへの変更が反映されるまでの時刻に大きな差異が生じる．例えば日次バッチ処理によってステートフルデータを変更する場合，特定の日に発生したイベントは翌日にならなければステートフルデータに変更が反映されない．対してステートフルストリーム処理ではリアルタイムにステートフルデータの変更が可能なので最新のデータを参照することが可能になる．

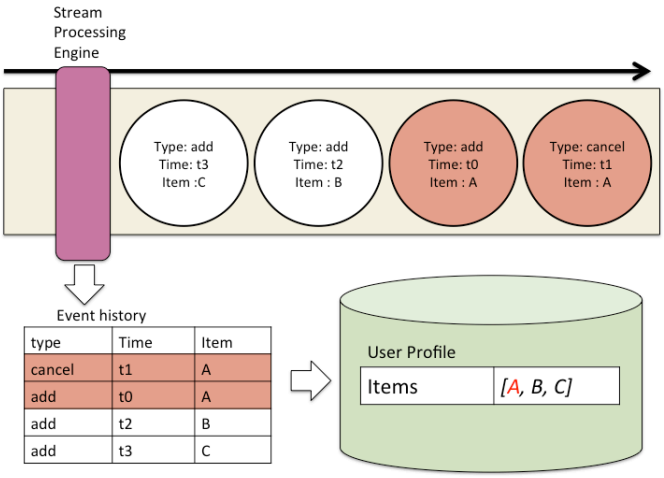

図 2　購入取消イベントが購入追加イベントより先に到達した例

しかし，ストリーム処理でステートフルデータを扱うのは難しい．これはストリーム処理ではイベントの到着順が保証されないためである。

ソーシャルゲームにおけるユーザのアイテムリストをストリーム処理で更新する例を考える．またイベントにはアイテムの購入を表すイベント (type=buy) とアイテムの購入の取り消しを表す購入取消イベント (type=cancel) の 2 種類が流れると仮定する．2 つのイベントにはアイテム名が付与される．ストリーム処理エンジンは購入イベントが到達するとアイテムリストにアイテムを追加する．購入取消イベントが到達するとアイテムリストに該当のアイテムが存在する場合アイテムをリストから削除する．図 1 はイベントが時刻順に到達した場合を表している．この時アイテムリストは正しい状態でデータストアに反映される．一方，図 2 では時刻 t_0 のアイテム購入イベントが時刻 t_1 のアイテム購入取消イベントより遅れて到達した場合を表している．この時アイテムリストには本来存在しないはずのアイテム **A** が存在する．これはアイテムの購入取消をアイテムの購入後に対して行ったはずであるのに，更新の順序が入れ替わることでアイテムが存在しない状態のリストに対してアイテムの購入取消を試みるためである．

この様にイベントの順序が入れ替わることで更新の順序も入れ替わりステートフルデータに対して誤った変更を行うことが起こりえる．

3. Phalanx のシステムデザイン

本節ではステートフルストリーム処理基盤である Phalanx のシステムについて説明する．まず 3.1 節で Phalanx のシステムの概要について説明する．3.2 節では更新の順序が保証されない場合でも結果整合性を担保するデータ型を提案する．

3.1 システムの概要

ステートフルストリーム処理を実現するためにはイベントが順不同で到達する状況で整合性を保つ必要がある．Phalanx では CRDT [1] というデータの更新の順序を問わず結果整合性を担保するデータ型を参考にしてデータ型を用意した．

CRDT(Conflict-free Replicated Data Type) は分散環境における複製の更新の衝突せずに結果整合性を担保するためのデータ型である．CRDT には Operation-based(Commutative Replicated Data Types, CmRDTs) と State-Based(Convergent Replicated Data Types, CvRDTs) の 2 種類存在する．CmRDT は複製の更新操作を可換にすることで更新が順不同であっても必ず同じ状態になる．CvRDT は参照する際に可換かつ結合的である集約関数を用いて全ての複製を結合する．集約関数は可換であるため CmRDT 同様，各複製への更新が順不同であっても整合性を担保する．

Phalanx では CRDT を参考にして更新時にステートフルデータを変更するのではなく，CmRDT のように更新操作を保存する (注1)．また，Phalanx は更新操作を集約してステートフルデータとして提供するデータ型を用意している．このデータ型によってストリーム処理における課題であるイベントの発生と到着の順序が不同であっても参照時に集約することで結果整合性を担保する．また更新時にイベントの到着順を考慮する必要がなくなるため，ステートフルデータの更新の記述を簡略化できる．

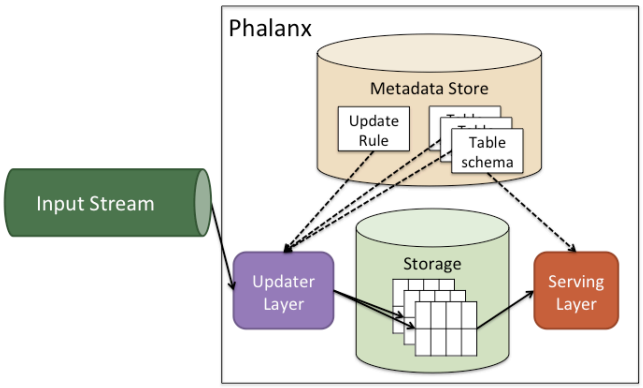

図 3　Phalanx のシステム概要

図 3 は Phalanx のシステムの概要図である．Phalanx はメタデータを保存する **Metadata Store**，入力イベントを更新操作に変換する **Updater Layer**，ステートフルデータへの更新操作データを保存する **Storage**，ストレージのデータをステートフルデータとして提供する **Serving Layer** で構成される．

まず入力ストリームのイベントは Updater Layer に渡される．イベントは発生時刻のタイムスタンプを持つ．Updater Layer は Metadata Store から **Table Schema(テーブルスキーマ)** と **Updater Rule(更新ルール)** の 2 種類のメタデータを参照して処理を行う．Phalanx ではステートフルデータはテーブルスキーマで規定される **Table(テーブル)** ごとに管理する．更新ルールは Updater Layer に届いたイベントを更新操作に変換する際に用いられる．テーブルスキーマ・更新ルールについては

(注1)：Phalanx では複製の扱いについては考えない

3.2.3 節で説明する．Updater Layer では更新ルールにもとづいてイベントを更新操作データに変換し Storage に追加する．

ステートフルデータを参照する際には Serving Layer に問い合わせる．Serving Layer ではテーブルスキーマに従って Storage の更新操作データをデータ型の集約関数をもちいて集約した結果を返す．

3.2 Phalanx のデータモデル

本節では Phalanx のデータ・モデルについて説明する．まず Phalanx のデータ型について説明する．次にデータ型を踏まえて Storage のデータ構造について説明する．最後に Phalanx のメタデータについて説明する．

3.2.1 Phalanx のデータ型

Phalanx は更新時にステートフルデータを変更するのではなく，参照時に更新操作を集約してステートフルデータを返すデータ型を用意している．

Phalanx のデータ型は CRDT を参考にしており，Phalanx のデータ型 D は以下のタプルで構成される．

$$D = (P, p_0, U, q)$$

P はステートフルデータへの変更の集合を表す．P をペイロードと呼称する．P はバージョン V と更新値 Δ の直積 $P = V \times \Delta$ とする．ペイロードのバージョン V は更新の順序を定める値である．よってペイロード P はバージョンに従って順序付けられる順序集合である．更新値 Δ はデータ型によって規定される．$p_0 = (ver_0, val_0) : ver_0 \in V, val_0 \in \Delta$ はペイロードの初期値を表す．ver_0 は V の最小値とする．U は入力イベント E をペイロード P に変換する更新メソッドの集合である．更新メソッドはデータ型によって複数存在することがある．入力イベントを $e \in E$ とするとき更新メソッド $u \in U$ は下記の様に表すことが出来る．

$$p = u(e)$$

入力イベント E はイベントの発生時刻を表すタイムスタンプ T とイベントの情報 C の直積 $E = T \times C$ として表すことができる．入力イベントはタイムスタンプ T によって順序付けられる順序集合である．更新メソッドは入力イベントとペイロードの間で単調性を満たす関数である．つまり更新メソッド u は下記の条件を満たす．

$$\forall e, e' \in E : e < e' \Rightarrow u(e) < u(e')$$

この条件を満たすため，ペイロードのバージョン V はイベントの発生時刻のタイムスタンプ T に基いた順序を守るような値となる．

q はペイロードを状態に変換する参照メソッドである．r_i は i 番目の更新におけるステートフルデータを表す．q は下記のように表す事ができる．

$$\begin{aligned} r_i &= q \cdot P_i \\ &= q \cdot \{p_0, ..., p_i\} \end{aligned}$$

Phalanx のデータ型は CRDT と同様データ型に更新メソッドや参照メソッドが用意されている．しかし CRDT と違い，Phalanx では状態を保持するのではなくペイロードを保持する．Phalanx では参照メソッドを用いてペイロードをステートフルデータに変換する．

次に CRDT [1] を参考にした Phalanx でサポートするデータ型について説明する．

Counter 型 Counter 型は数値型の更新値を合計した値を返す方である．更新操作メソッド incr を用いて Counter 型に数値を渡す．Counter 型は $D_{counter} = ((V, Z), (ver_0, 0), incr, query)$ で表すことができる．

Counter 型は下記の様に表すことができる．

```
payload: V v, Z z
  initial: ver0, 0
update: incr(V u, Z z):V v, integer x
  let v = u
  let x = z
query: query(P): integer x
  let x = ∑_{z∈P.Z} z
```

Register 型 Register 型では更新時に与えられた値を保存するための型である．Phalanx では Last Writers Wins ベースの Register(LWW-Register) を用意している．LWW-Register では最新の更新操作によって与えられた値を返す．Phalanx ではバージョンを元に最新の値を返す．

Register 型は下記の様に表すことができる．

```
payload: V v, X x
  initial: ver0, null
update: set(V u, X y):V v, X x
  let v = u
  let x = y
query: query(P): X x
  mv = ver0
  mx = null
  for p ∈ P do
    if p.v > mv
      then mx = p.x; mv = p.v;
  let x = mx
```

Set 型 Set は更新時に与えられた値を要素とする集合を保存するための型である．Phalanx では CRDT のデータモデルである G-Set と 2P-Set の 2 つの Set 型をサポートしている．

G-Set(Grow-only Set) は与えられた値を集合の要素として追加し常に単調増加する集合である．G-Set では add という更新メソッドで与えられた要素を集合を追加する．

```
payload: V v, set S
  initial: ver0, ∅
update: add(V u, X x): V v, set S
  let v = u
  let S = {x}
query: query(P): set S
   let S = ∪P.S
```

2P-Set(Two-Phase-Set) は要素の追加だけでなく削除も可能

な集合である．2P-Set は add と remove の 2 つの更新メソッドが用意されている．2P-Set 型は下記の様に表すことができる．

```
payload: V v, set AS, set RS
  initial: ver_0, ∅, ∅
update: add(V u, X x): V v, set AS, set RS
  let v = u
  let AS = {x}
  let RS = ∅
update: remove(V u, X x): V v, set AS, set RS
  let v = u
  let AS = ∅
  let RS = {x}
query: query(P): set S
  let S = ∪P.AS \ ∪P.RS
```

Map*型 Map はキーとペアになる値を保持する型である．Phalanx の Map*型では値ごとに異なるデータ型を提供する．MapCounter はキー毎に与えられた数値の合計値を保持するデータ型であり，MapRegister はキー毎に与えられた値を保持するデータ型であり，MapSet はキー毎に与えられた要素で構成される集合を保持するデータ型である．Map*型には add という更新操作メソッドを用意しており，add では Map のキーとデータ型に適した値の組を渡す．

3.2.2 Storage のデータ構造

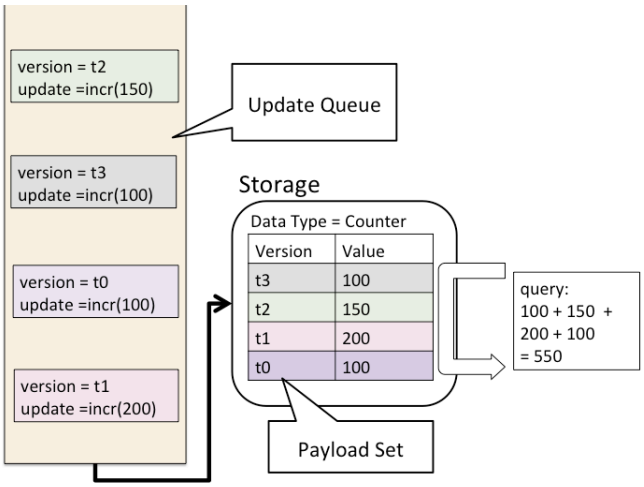

図 4　Storage のデータ構造

ここでは Storage におけるペイロードの要素のデータ構造について説明する．ペイロードはバージョンによって順序付けられる集合である．そのためペイロードの要素の Storage における配置はイベントの到着順序にかかわらずイベントの発生時刻に基づいたバージョン順に配置される．データを参照する際にはペイロードをスキャンしデータ型で規定される query メソッドを用いてステートフルデータへの変換を行う．

図 4 は Counter 型のデータ構造を表したものである．Update Queue の中の古い更新から順に Storage に反映される．前述の通りペイロードはバージョンによって順序が決まるため衝突することなく Storage に反映される．参照時には Storage のペイロードをスキャンし参照メソッドを適用する．

3.2.3 メタデータ

データ型で定義されているペイロード以外の情報はメタデータとして管理される．メタデータにはテーブルスキーマと更新ルールの 2 つが存在する．

テーブルスキーマは Phalanx で扱うアプリケーションのデータ構造を定義するための記述である．表 1 はソーシャルゲームにおけるユーザのステータスを Phalanx のテーブルとして表すテーブルの例である．Phalanx のテーブルスキーマはキーと 1 つ以上のカラムによって構成される．キーはステートフルデータを参照する際に使われる．カラムはステートフルデータを表す属性であり，必ず一つのデータ型と紐付かれる．Phalanx ではキーを指定することでカラムのステートフルデータを参照することができる．その際，カラム紐づくデータ型の参照メソッドに従ってペイロードをステートフルデータへと変換する．

更新ルールはストリームで流れてくるイベントをペイロードに変換する記述である．Phalanx ではカラムに紐付いたデータ型によって順序に関係なくステートフルデータへの更新を行うことが可能である．そのため入力イベントをデータ型の更新メソッドに変換するようなシンプルな更新ルールを記述するだけで更新を実現することが可能になる．

4. Phalanx の実装

本章では Phalanx の実装について説明する．

4.1 ペイロードのスナップショット

Storage ではステートフルデータへのペイロードをカラム毎に保存する．しかし全てのペイロードを Storage に保存するとデータサイズは単調に増加する上，参照時に集約するペイロードが肥大するにつれレイテンシが高くなる．時間が経つにつれ過去への変更は頻度が少なくなるため，過去分ペイロードはステートフルデータを保存しておけば良い．Phalanx の実装ではデータ型を拡張しペイロードをスナップショットに変換する手法を実現した．

まず式 (1) のデータ型を D_{snap} に拡張する．

$$D_{snap} = (P, p_0, U, q', S, m)$$

S はスナップショットを表し，m は集約メソッドを表す．スナップショット S はペイロードのバージョン V と集約値 Φ の直積 $S = V \times \Phi$ とする．Phalanx では一定の時間が経った過去のペイロードを集約してスナップショットとして保存する．ペイロード $P_n = \{p_0, ..., p_n\}$ を集約メソッドを持ちいて集約した値を s_n と表す．このとき集約メソッドは下記の条件を満たす．

$$\begin{aligned} s_i &= m \cdot \{s_k, P_{k+1,i}\} \\ &= m \cdot \{s_k, p_k + 1, ..., p_i\} (k < i) \\ s_i &= m \cdot s_i \\ s_0 &= \emptyset \end{aligned}$$

$P_{i,j} (i < j)$ はバージョン i 番目から j 番目のペイロードを表す．ペイロードとスナップショットの要素の順序はバージョン V に

Key-Column	Key	Column		
カラム名		total_price_of_items	items	last_login_date
データ型		Counter	Set	Register
レコード 1	taro	1000	[”A”, ”B”]	”2018/01/01”
レコード 2	hanako	4000	[”A”]	”2018/01/03”

表 1　Phalanx のテーブル例

よって決まる．

また参照メソッド q' はスナップショットを考慮するように下記の様に拡張する必要がある．

$$
\begin{aligned}
r_i &= q' \cdot \{s_k, P_{k+1,i}\} \\
&= q' \cdot \{s_k, p_k + 1, ..., p_i\} (k < i) \\
r_k &= q' \cdot s_k
\end{aligned}
$$

過去のステートフルデータへの参照があった場合はペイロードを全てスキャンする必要はなくスナップショットとその差分のペイロードを用いて参照メソッドを使用すれば良い．またスナップショットを作成することで過去のペイロードを参照する必要がなくなるので，過去分のペイロードを削減してデータサイズの増加を抑えることができる．

4.2　HBase におけるペイロードのフォーマット

Phalanx の Storage には Apache HBase [2] を用いた．HBase のデータは Cell という単位で構成される．Cell には Row, ColumnFamily, Qualifier, Timestamp, Value をフィールドとして持つ．HBase の Cell の配置は行キー (Row) と列属性群 (ColumnFamily) と列属性値 (Qualifier) で決められる，また Cell は Qualifier 値の辞書順にソートされる．Phalanx の実装では Row にキー，カラム名，スナップショット識別子の順で構成されるバイト配列を与える．スナップショット識別子は Cell に保存される値がスナップショットかどうかを判別するために用いられる．また ColumnFamily は常に単一の値を固定して使用する．そして Qualifier にバージョンを付与し Value にデータ型に合わせたペイロードの更新値もしくはスナップショットの集約値を与える．そのため Row ごとにバージョンでソートされたデータ構造を実現される．

HBase の Cell には時刻を表す Timestamp フィールドが用意されているが，Phalanx ではペイロードのバージョンを Timestamp ではなく Qualifier を用いて管理する．それはバージョンにはタイムスタンプだけでなく後述の重複除去に必要な情報を付与するためである．

4.3　ストリーム処理の到達保証

ストリーム処理を実装する際にイベントの転送の到達保証について考慮する必要がある．イベント転送の到達保証は大きく 3 つにわけられる．

- **At Most Once Semantics**
- **Exactly Once Semantics**
- **At Least Once Semantics**

At Most Once Semantics はイベントが最大で一回到達，Exactly Once Semantics はイベントが一度だけ到達，At Least Once Semantics はイベントが少なくとも一回以上到達を保証する．つまり At Most Once Semantics はイベントが転送途中で失われる可能性があり，At Least Once Semantics ではイベントが重複して届く可能性があることを示す．一番望ましいのは Exactly Once Semantics であるが，Exactly Once Semantics を実運用で実現するためには厳密な送達制御が必要となり，送達制御のオーバヘッドが性能劣化を引き起こすことが起こりえる．MillWheel [3] ではイベントに予めユニークな ID を付与し ID による重複除去の機構を設けている．Phalanx では MillWheel を参考に入力イベントの到達保証が At Least Once Semantics でも重複を除去する仕組みを設けている．Phalanx の入力イベントには必ずユニークな UUID の付与を必須と定めており，ペイロードのバージョンにはタイムスタンプに加え UUID を追記している．もし重複するイベントが Updater Layer に届いてもペイロードは同じバージョンになるため，HBase の同一の Cell として上書きされる．そのため Phalanx では参照時には重複が除去されていることを保証する．

```
# 入力データソース名
source: stream1
# 操作を行うデータの分岐
branches:
# 操作を行うイベントの条件を表すJQ
  - condition:
      ".type == \"view\""
    tables:
      # テーブル名
      - tableName: user_status
      # カラムに対する操作
        ops:
        # キーになる値をイベントから抽出するJQ
          -  key: .user_id
          # 更新メソッド名
             columnName: total_price_of_items
             # 更新メソッド名
             method: incr
             # 更新メソッドが渡す値をイベントから抽出するJQ
             paramJq: .item_price
```

図 5　更新ルールの例

4.4　更新ルールの実装

更新ルールの実装について説明する．更新ルールは入力ストリームから受け取ったイベントをペイロードに変換する．入力ストリームからは様々な種類のイベントが Updater Layer に届くと考えられるため，適切なイベントを選択して更新操作に変換できるようにする必要がある．Phalanx の更新ルールではイベントの処理を Branch という単位で分割する．Branch にはフィルタリング条件を設定することができ，条件を満たすイベ

カラム名	Hive におけるデータ型	説明
user_id	string	ユーザ ID
total	int	課金総額
last_purchase_time	string	最後に課金した時刻
by_day	map<string,int>	日毎の課金額
dt	string	日付

表 2　ユーザの課金状態を表す hive_purchase テーブル

カラム名	Hive におけるデータ型	説明
user_id	string	ユーザ ID
amount	int	課金総額
time	string	最後に課金した時刻
type	string	イベントの種類
dt	string	日付

表 3　ユーザの行動イベントの履歴テーブル input_purchase_events

ントのみ更新操作に変換することが可能になる．

図 5 は更新ルールを YAML 形式で記述した時の例である．一つの入力ストリームから取得したイベントを user_status テーブルへの更新操作に変換する．この時入力イベントは JSON フォーマットであることを仮定している．この更新ルール中の condition のような条件文や paramJq のようなイベントから更新値を抽出する際に JQ [4] という JSON を操作用 DSL を利用している．

5. 適 用 事 例

本章では Phalanx を当社で活用した適用事例について説明する．当社では様々な Web サービスを運営している．そのうち弊社で提供する SNS の課金テーブルを例にあげる．

ある SNS ではユーザの課金状態データを Hive テーブルとして管理していた．表 2 はユーザの課金状態を Hive で管理するテーブル hive_purchase のスキーマである．hive_purchase を更新する場合，ユーザの行動イベントを保存する Hive テーブル input_purchase_event を用いる。表 3 は input_purchase_event のテーブルスキーマである．(簡略化のため input_purchase_event のレコードには重複はないと仮定する．) 図 6 は 2018/01/02 の input_purchase_event のデータを用いて hive_purchase の 2018/01/02 分を更新する際の HiveQL クエリである．hive_purchase には日次でデータの更新を行うためユーザの課金情報は翌日にならなければ参照できなかった．mergemap 関数は第一引数の Map に第二引数の Map のエントリを上書き追加する UDF である．

HiveQL のような SQL ライクな言語でステートフルデータへの変更を行う場合は更新クエリが複雑化する．まず SQL クエリでは前状態の抽出 (行 17-27)，新規データの集約 (行 29-38)，両者のマージ (行 4-15) の三段階を記述する必要がある．

そして SNS のユーザの課金情報テーブルを Hive から Phalanx で置き換えた．表 4 は Phalanx の課金情報テーブル phalanx_purchase のテーブルスキーマである．phalanx_purchase ではユーザごとに課金情報を管理するためユーザ ID がキーとなる．

```
1  insert overwrite table
2    hive_purchase partition(dt = "2018-01-02")
3  select
4      coalesce(prev.user_id, today.user_id) as user_id,
5      coalesce(prev.total, 0) + coalesce(today.amount, 0)
6        as total,
7      coalesce(today.latest_time, prev.last_purchase_time)
8        as last_purchase_time,
9      if(today.user_id is null,
10          prev.by_day,
11          mergemap(
12               coalesce(prev.by_day, map("2017-01-02", 0)),
13               map("2018-01-02", cast(today.amount as int))
14          )
15     ) as by_day
16 from
17 (
18     select
19        user_id,
20        total,
21        last_purchase_time,
22        by_day
23     from
24        hive_purchase
25     where
26         dt = "2018-01-01"
27 ) prev
28 full outer join
29 (
30     select
31         user_id,
32         sum(cast(amount as int)) as amount,
33         max(time) as latest_time
34     from
35         input_purchase_events
36     where dt = "2018-01-02" and type = "purchase"
37     group by user_id
38 ) today
39 on  prev.user_id = today.user_id
```

図 6　2018/01/02 分のデータを hive_purchase に追記するための HiveQL クエリ

Key	Value	
	カラム名	Phalanx のデータ型
	total	Counter
	last_purchase_time	Register
	by_day	MapCounter

表 4　phalanx_purchase のテーブルスキーマ

```
{
  "time":"2018-01-01T00:00:00.000+09:00",
  "uuid":"92ed7a7f-6e56-41e8-bdfa-fa2c8a8b0e1b",
  "type":"purchase",
  "amount":1000,
  "user_id":"etQnYWPZxJEiDYwKYPpX"
}
```

図 7　課金情報のイベント例

図 7 は SNS でユーザが課金したときに生じるイベントログを簡略化した例である．type はイベントログのタイプである．課金情報テーブルでは type の値が「purchase」になるイベントだけが更新の対象となる．user_id はユーザ ID，amount は課金額を表す

図 8 は phalanx_purchase の課金情報の更新ルールである．

Phalanx ではデータ型によって前の状態を考慮する必要はな

```
source: stream1
branches:
  - condition:
      ".type == \"purchase\""
    tables:
      - tableName: user_purchase_status
        ops:
          - key: .user_id
            columnName: last_purchase_time
            method: set
            paramJq: .time
          - key: .user_id
            columnName: total
            method: incr
            paramJq: .amount
          - key: .user_id
            columnName: by_day
            method: add
            paramJq: "{(.time[:10]):.amount}"
```

図 8　課金情報テーブルの更新ルール

いため，HiveQL における前状態の抽出は必要なくなる．またデータ型によって定義される参照メソッドを使うため新規データと前状態のマージも不要となる．そして新規更新分を集約する必要も無いため，Phalanx の更新ルールではイベントごとのペイロードへの変換についてのみ記述すれば良くなる．

6. 関 連 研 究

ステートフルストリーム処理を行う際にはラムダアーキテクチャ [5] とカッパアーキテクチャ [6] という 2 つのアーキテクチャが広く知られている．ラムダアーキテクチャはバッチ処理を行うバッチレイヤ，バッチ処理の結果をビューに変換するサービングレイヤ、ストリーム処理を行うスピードレイヤの 3 つのレイヤに分かれる．ラムダアーキテクチャではサービングレイヤの結果にストリーム処理の結果を差分として追記することでリアルタイムにステートフルなデータを参照することを可能にする．しかしラムダアーキテクチャではバッチレイヤとスピードレイヤの 2 つの更新処理を実行するレイヤをそれぞれ実装する必要があるのと，バッチ処理とストリーム処理の結果を統合可能なビューを管理しなければならないので実装と運用におけるコストが高い．ラムダアーキテクチャの問題を解決するアーキテクチャとしてカッパアーキテクチャが提案された．カッパアーキテクチャはラムダアーキテクチャのバッチレイヤを取り除き，データ処理はスピードレイヤに任せサービング DB と呼ばれるデータストアに全ての結果を反映する．Phalanx は Updater Layer をスピードレイヤ，Storage と Serving Layer をサービング DB とみなすことでカッパアーキテクチャの一種とみなすこともできる．しかし既存のカッパアーキテクチャのの実装例 [7] では本論文で述べた様なストリーム処理をステートフルデータに変換する手法について議論はされていない．

既存のステートフルストリーム処理はストリーム処理の耐障害性を担保するためのスナップショットでの利用目的で導入されるケースが多い．Akidau らの研究 [8] ではマーカーと呼ばれる特殊なイベントを入力ストリームに追加する．マーカーを基準としたエポック単位でストリーム処理を行うことで，処理パイプラインの各タスクにおけるスナップショットの一貫性を保つ方法を提案している．また Noghabi らの研究 [9] ではストリーム処理エンジンで行った処理をローカルの DB にチェンジログとして残すことで定常処理と並行してローカル DB から復旧処理を実現する手法を提案している．Phalanx でのステートフルデータをストリーム処理エンジンのスナップショットではなく，ストリーム処理の結果として扱う点でこれらの研究とは異なる．

また状態に対する変更処理をモノイドに限定するようなリアルタイム処理システム [10] も存在する．モノイドは二項演算子と単位元のみで構成されるため，CRDT のような複雑な処理を行うことが難しいと考えられる．

7. ま と め

本研究ではステートフルデータをストリーム処理で扱うための解析基盤 Phalanx を開発した．Phalanx ではイベントの到着順が保証されないストリーム処理においてステートフルデータの変更を可能にするために更新時にステートフルデータを変更するのではなく参照時に更新操作を集約しステートフルデータに変換するデータ型を用意した．このデータ型では結果整合性を保証する．Phalanx のデータ型は順不同で更新が可能であるのでステートフルデータの変更をイベントとデータ型への操作を対応するだけの平易な更新ルールを記述するのみで実現可能にした．また Phalanx を実装し，SNS における適用例について述べた．

今後の課題として今回紹介した以外のデータ型の実装がある．また，現在はテーブルのデータは単一の文字列型のキーでしか参照できないが複合キーへの拡張も計画している．

文　献

[1] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] Apache Software Foundation. Hbase.

[3] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.*, 6(11):1033–1044, August 2013.

[4] Stephen Dolan. Jq.

[5] Nathan Marz and James Warren. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2015.

[6] Jay Kreps. Questioning the lambda architecture, 2014-07-02.

[7] Nicolas Seyvet and Ignacio Mulas Viela. Applying the kappa architecture in the telco industry.

[8] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernndez-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt,

and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8:1792–1803, 2015.

[9] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. Samza: Stateful scalable stream processing at linkedin. *Proc. VLDB Endow.*, 10(12):1634–1645, August 2017.

[10] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. Realtime data processing at facebook. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1087–1098, New York, NY, USA, 2016. ACM.