Asakusa DSLの内部

2011/02/25

あらかわ (@ashigeru)

講演者について

- 荒川 傑 (@ashigeru)
  - Ashigel Compilerの中の人
    - Asakusa DSLのデザインも担当
  - 株式会社グルージェント開発部所属
    - 2010年9月より出向中
  - 普段の仕事
    - 研究: コンパイラ、開発環境
    - 調査: Google App Engineなど
    - 教育: 授業設計、教材開発

本日のテーマ

- 第一部: Asakusa DSLの設計思想
  - どんな言語か、何を考えて作ったのか
- 第二部: Asakusa DSLのセマンティクス
  - Hadoopとどのように関連するか
- 第三部: コンパイラのコード生成
  - どのようなコードを生成しているか
- 第四部: コンパイラの最適化手法
  - どのような最適化があるか

本日の内容

- 触れること
  - DSLの位置づけ
  - DSLの設計思想
  - 言語設計における判断
  - DSLのセマンティクス
  - 実行計画作成の仕組み
  - コード生成の仕組み
  - 最適化手法
- 触れないこと
  - Asakusa Frameworkの使い方
  - Asakusa DSLの書き方

言語設計者としての主張

- プログラミング言語は作るべきでない
  - 品質がどうしても落ちる
  - 新しい概念の習得は大変
  - 言語そのものよりも周辺環境が重要
- それでもAsakusa DSLなんてものを作った
  - その価値はあったのか、という観点で

まずは何を目指してDSLを作ったのかを紹介

第一部
Asakusa DSLの設計思想

第一部: Asakusa DSLの設計思想

- Asakusa "Framework"の概要
- Hadoopを利用する障壁
- Asakusa DSL
- まとめ

第一部: Asakusa DSLの設計思想

- Asakusa "Framework"の概要
- Hadoopを利用する障壁
- Asakusa DSL
- まとめ

Asakusa "Framework"の概要

- エンタープライズ系のバッチ処理にHadoopを利用するためのフレームワーク
  - 外部システムとの連携
  - 業務トランザクションの実現
  - 複雑なMap Reduceジョブネットの管理

外部システムとの連携

- Map Reduceでない処理が含まれる
  - 計算単体で終わることはまずない
  - データを外部から取り込んだり、計算結果をほかで利用したりすることが必須
- 例:
  - フロントデータベースからの取り込み
  - 外部基幹システムからの取り込み
  - 印刷サーバーから帳票の出力

業務トランザクションの実現

- 外部システムと整合性を保ちながら処理
  - 一つ一つの処理はオンライントランザクションよりもずっと時間がかかる
- Hadoopだけではジョブ単位の整合性のみ
  - ジョブネット単位のロールバックやロールフォワードが必須
  - 上位のワークフローエンジンとの連携が必須

複雑なMap Reduceジョブネットの管理

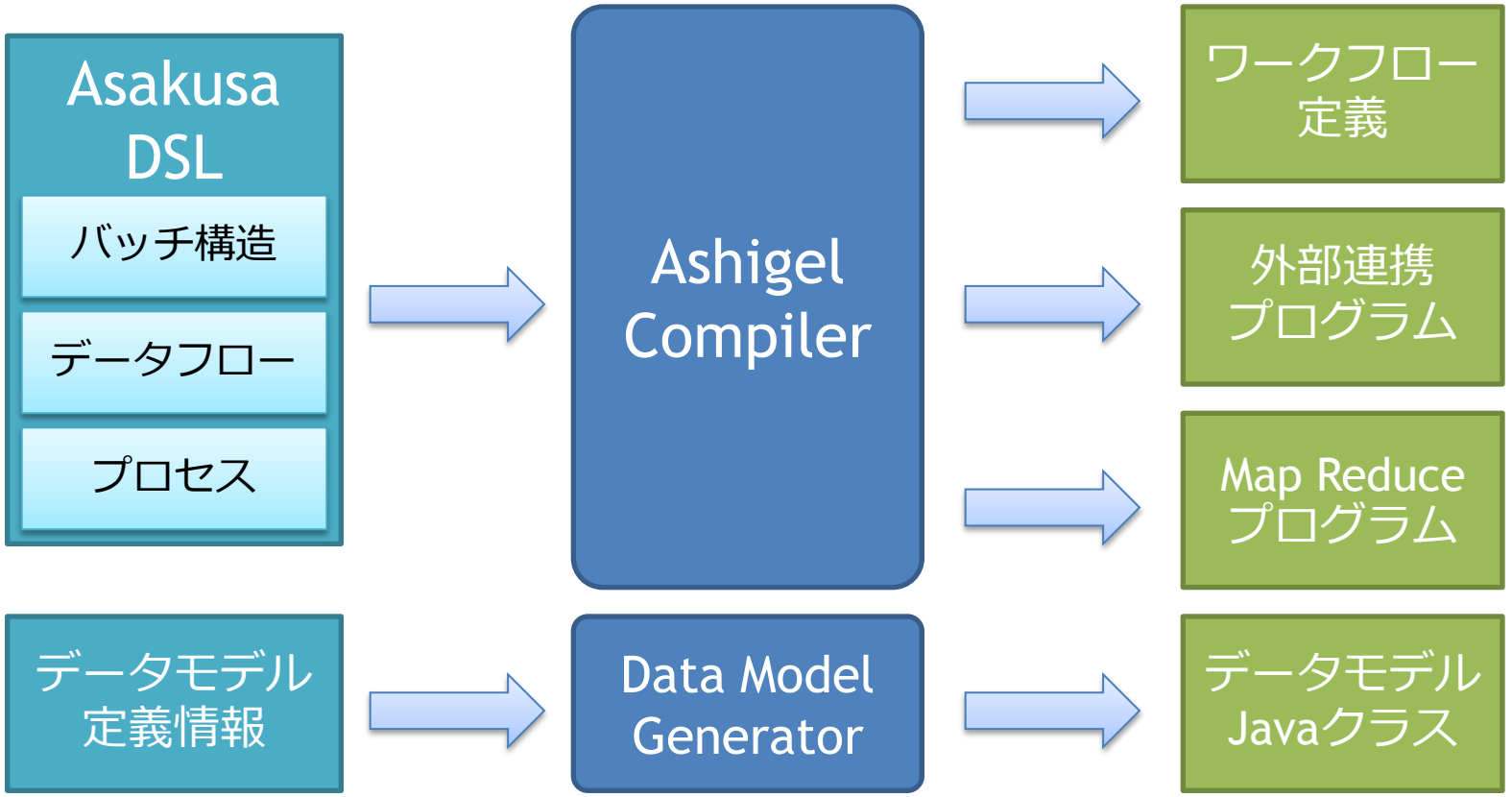
- 単体のMap Reduceジョブで処理が完結することはほぼない
  - ジョブの結果を複数の別のジョブが使う
  - 複数のジョブの結果を一つのジョブが使う
- 業務トランザクションとの兼ね合いも
  - ジョブネット中のどこで失敗しても整合性が崩れないようにする

Asakusa "Framework"のアプローチ

- 外部システムとの連携
  - 外部システムのデータをMap Reduceで処理できる
  - Map Reduceの処理結果を外部システムに送信できる
- 業務トランザクションの概念
  - DSLレベルでトランザクション境界を記述できる
  - ワークフローエンジンと連携してロングトランザクションを実現
- 複雑なMap Reduceジョブネットの管理
  - 複雑なデータフローをDSLで記述し、Map Reduceジョブ群を自動生成
  - ワークフローエンジンごとにジョブネット記述を自動生成

Asakusa "Framework"での開発

- Asakusa DSLを中心にプログラムを生成



第一部: Asakusa DSLの設計思想

- Asakusa "Framework"の概要
- Hadoopを利用する障壁
- Asakusa DSL
- まとめ

Hadoopを利用する障壁

- 設計技法の欠如
- Map Reduceの適用範囲の理解
- Hadoop特有の実装

設計技法の欠如

- どのような設計情報があれば実装を始められるのか
  - データモデル
  - データフロー
  - 構造化単位
- Map Reduceのプログラムにするにはどこまで情報が必要か
  - 取り扱うデータの転送形式、ファイル形式
  - データフロー内の「プロセス」の粒度

Map Reduceの適用範囲の理解

- どんな計算が得意か
  - どんな計算が不得意なのか
- どう書くと速いか
  - どう書くと遅いのか
- そもそもできないことは何か
- 設計、実装担当にどう理解してもらうか

Hadoop特有の実装

- ある処理を実現する際に、どのようなものを記述すればよいか
  - JOINの処理
  - グループ化
- どのような知識が必要か
  - ネットワークアルゴリズム
  - ソートアルゴリズム
  - Hadoopの裏側の挙動

Asakusa Frameworkのアプローチ

- 設計技法の欠如
  - 詳細設計と実装で両方使えるDSLを用意
    - DSLの語彙と詳細設計の語彙を一致させる
    - DSLの形式を詳細設計の成果物に寄せる
- Map Reduceの適用範囲の理解
  - Map Reduceの処理パターンに対応するDSLを用意
    - DSLで書けないことは実装できない
    - DSLで書けることは速く動くように
- Hadoop特有の実装
  - DSLを元にコンパイラが自動生成
    - DSLで書けることを制限
    - 裏側の実装を意識させない

第一部: Asakusa DSLの設計思想

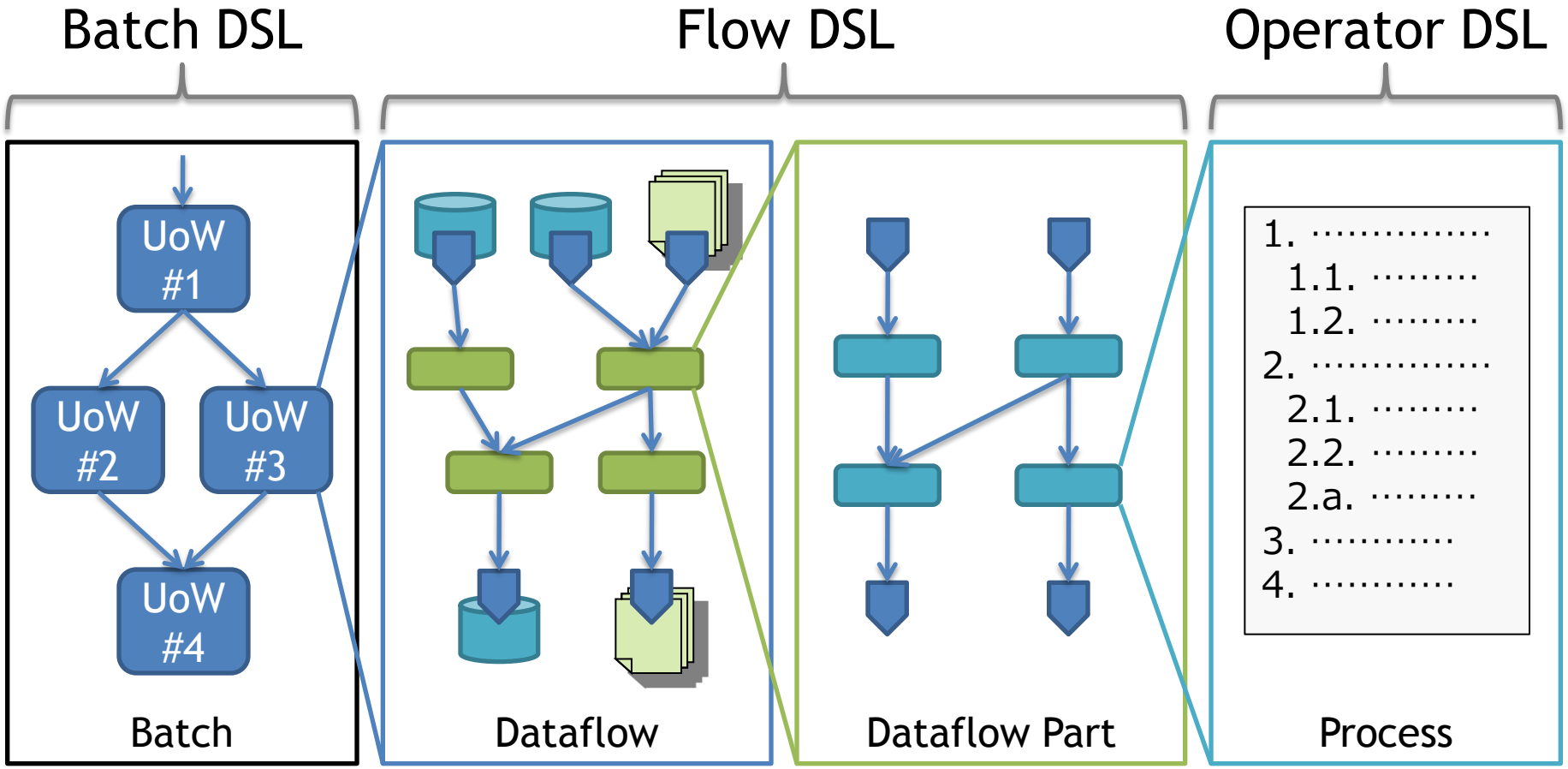
- Asakusa "Framework"の概要
- Hadoopを利用する障壁
- Asakusa DSL
- まとめ

Asakusa DSL

- Operator DSL
  - プロセスの最小単位である「演算子」を定義
  - 「Map Reduceができること」をテンプレート化して提供
  - ほとんど普通のJavaを書ける
- Flow DSL
  - 演算子をつなぎ合わせて「データフロー」を定義
  - 作成したデータフローは演算子としてコンポーネント化
  - ホスト言語がまさかのJava
- Batch DSL
  - データフローやスクリプトを組み合わせて「バッチ」を定義
  - 処理の依存関係をシンプルに書く
  - まさかのJava

各種DSLの構造

- 層ごとに書くべきことが異なる



例: Operator DSL

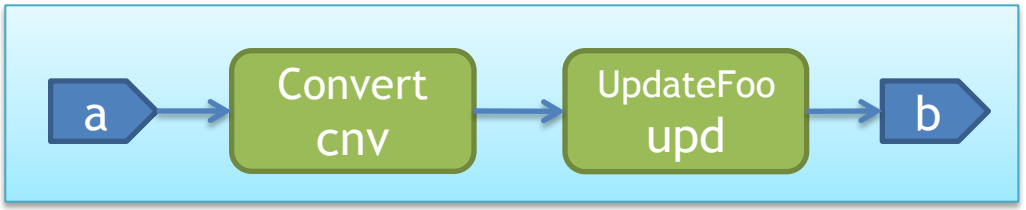
- Javaのメソッド+アノテーション

```
public abstract class ExampleOperator {
    @MasterJoinUpdate
    public void updateWithMaster(
            @Key(group = "id") ItemMst master,
            @Key(group = "itemId") PurchaseTrn tx) {
        tx.setPrice(master.getPrice());
    }
}
```

例: Flow DSL - 部品

- 入出力とデータフローを記述

```
@FlowPart
public class ExampleFlowPart extends FlowDescription {
    In<Hoge> a;
    Out<Foo> b;
    public ExampleFlowPart(In<Hoge> a, Out<Foo> b) {
        this.a = a;
        this.b = b;
    }
    @Override
    public void describe() {
        OperatorFactory f = new OperatorFactory();
        Convert cnv = f.convert(a);
        UpdateFoo upd = f.updateFoo(cnv.out);
        b.add(upd.out);
    }
}
```



例: Flow DSL - 業務トランザクション

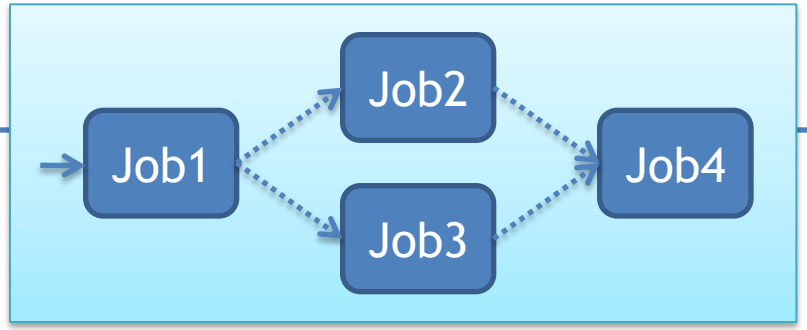
- 外部システムとの連携について追記

```
@JobFlow(name = “example”)
public class ExampleJobFlow extends FlowDescription {
    In<Hoge> in;
    Out<Hoge> out;
    public ExampleJobFlow(
            @Import(name=“hoge”, description=HogeFromDb.class)
            In<Hoge> a,
            @Export(name=“hoge”, description=HogeToFile.class)
            Out<Hoge> b) {
        this.in = a;
        this.out = b;
    }
    …
```

例: Batch DSL

- ジョブ間の依存関係を明記

```
@Batch(name = "batch.example")
public class ExampleBatch extends BatchDescription {
    @Override
    protected void describe() {
        Work job1 = run(Job1.class).soon();
        Work job2 = run(Job2.class).after(job1);
        Work job3 = run(Job3.class).after(job1);
        Work job4 = run(Job4.class).after(job2, job3);
    }
}
```



3層のDSL

- DSLごとに扱う内容がまったく違う
  - Operator DSL: 手続型
    - レコードの順序を考慮する処理を書きやすい
    - 「普通のJava」としてテストも書ける
    - いざとなれば力技で処理を書ける
  - Flow DSL: フロー型
    - 「データの流れ」をそのまま書ける
    - Map Reduceと演算子の関係を高度に隠蔽
  - Batch DSL: 依存グラフ型
    - 依存関係を書く設定ファイルのレベル

補足:レコードの順序を考慮する処理

- 並列性が高いが、グループごとの処理が煩雑
  - 伝票照合
  - 単品管理
  - 修正伝票
  - など
- 一階の関係論理では極端に書きにくい
  - ストアドプロシージャ祭り
  - PigやHiveはUDFのような補助的な仕組みで対応
- ドメインによってはこのタイプの計算が多い
  - 手続き型の計算で楽に済ませる
  - 徹底的にテストする

ホスト言語としてのJava

- コンパイラの開発工数的な話
  - 言語設計自体はパーサから書いたほうが楽
  - IDEとの連携まで考えるとホスト言語に任せたほうが楽
- スキルセットとの兼ね合い
  - 新しい言語の習得コストをとても懸念
  - SQLに寄せたり、Scalaで作ろうという案もあった
- とはいえ、いろいろ最悪
  - 型の計算を気合で書いた
  - メタプログラミングで心が折れかけた
  - Flow DSLが黒魔術

その他、気にしたこと

- 構造化技法
  - 段階的にモジュール化して、一度に把握する範囲を減らせるように
  - Asakusa DSLではデータフローを入れ子にできる
- テスト技法
  - データフローの単位でテストを実施できるように
  - 生成するMap Reduceを意識せずに論理構造だけでテスト
- IDEとの連携
  - 強い型付けで即座に結線ミスを検出
  - ポップアップされるドキュメンテーションを読みやすいように自動生成部分を設計

第一部: Asakusa DSLの設計思想

- Asakusa "Framework"の概要
- Hadoopを利用する障壁
- Asakusa DSL
- まとめ

第一部のまとめ

- Asakusa Framework
  - エンタープライズバッチをHadoopで動かす
  - Hadoopの大変なところをDSLで和らげる
- Asakusa DSL
  - 3種類のDSL
  - それぞれ異なる領域をカバーするDSL
  - いざとなれば手続き型も活用

Asakusa DSLはHadoopのどんな機能とどのように対応するのか

第二部
Asakusa DSLのセマンティクス

第二部: Asakusa DSLのセマンティクス

- Hadoop Map Reduceの表現力
- Operator DSLの表現力
- Flow DSLの表現力
- データフローの実行計画
- まとめ

第二部: Asakusa DSLのセマンティクス

- Hadoop Map Reduceの表現力
- Operator DSLの表現力
- Flow DSLの表現力
- データフローの実行計画
- まとめ

Hadoop Map Reduceの表現力

- Hadoopでどんな計算ができる？
- ここでの記法
  - 関数型を A -> B のように表記
    - A型のデータを引数にとりB型のデータを返す
  - T型のシーケンスを[T]と表記
    - List<T>のようなもの
  - 直和型を A|B|... のように表記
    - A型かB型か...のどれか
  - レコード型を{a:A, b:B,...}のように表記
    - フィールドを持つオブジェクトのようなもの
  - 組型を(A, B, ...)のように表記
    - RDBのカラム列のようなもの

補足: Javaでの直和型の表現

- インスタンスフィールドごとに要素型のデータを持つだけ
  - 該当する型以外はnullにしておく
  - Writableは型情報を落としてシリアライズ

```
public class Union implements Writable {
    public int slot;
    public Hoge slot0;
    public Foo slot1;
    public Bar slot2;
    …
}
```

「普通の」Map Reduce

- Map: A -> [S]
  - データごとに分解して別のデータを出力
- Shuffle: [S] -> [[S]]
  - データ全体をグループ分け (List<List<S>>)
- Reduce: [S] -> [B]
  - グループごとに分解して別のデータを出力

- これだけではデータフローというよりパイプライン処理程度しかできない...

MultipleInputs

- 複数の入力を一度のMap Reduceで処理
  - 入力ごとにMapperを指定できる
  - 入力ごとに異なる型を利用できる
  - Shuffleの型は揃えておく必要あり
- 少し表現力が上がった
  - Map1: A1 -> [S]
  - Map2: A2 -> [S]
  - ...

MultipleOutputs

- 複数の出力を一度のMap Reduceで行う
  - 出力ごとに名前を付けられる
  - 出力ごとに異なる型を指定できる
- 少し表現力が上がった
  - Reduce: [S] -> { b1:[B1], b2:[B2], ...}

直和型を利用したシャッフル

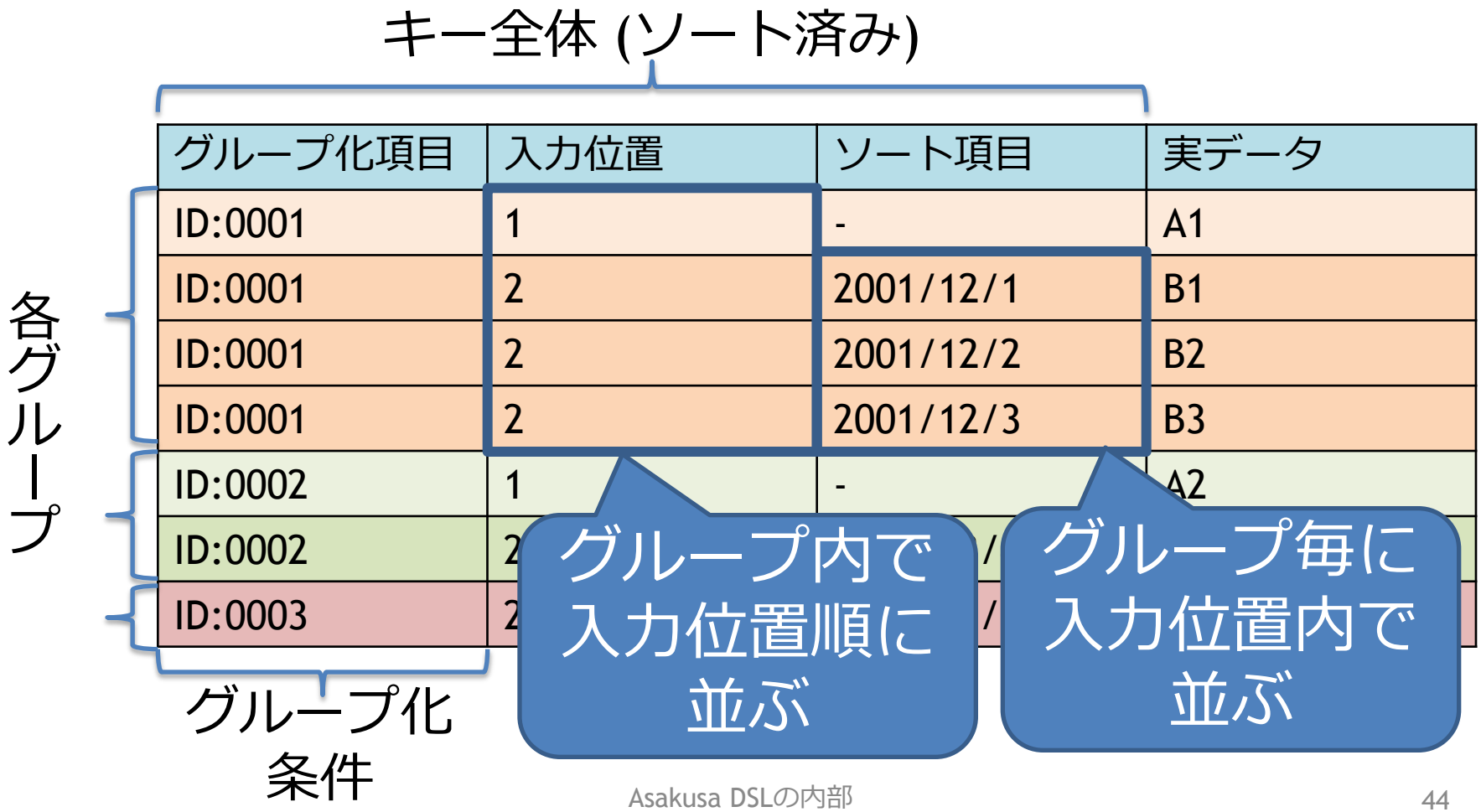
- Map: A -> [S|T|...]
  - 「SかTか...のどれか」型をシャッフルに渡す
- Shuffle: [S|T|...] -> [[S|T|...]]
  - データ全体をグループ分け
- Reduce: [S|T|...] -> [B]
  - 「SかTか...のどれか」型を元に何かを出力
- 少し表現力が上がった
  - Shuffle時に複数の型を取り扱える
    (ように見せかけられる)

Secondary Sort

- Reduce時のグループをキーの一部で決められる
  - グループ化条件を指定する
  - job.setGroupingComparator()
- 残りのキーでグループ内をソートできる
  - ソート条件を指定する
  - job.setSortComparator()
- 直和型と組み合わせると…

Secondary Sort付きのShuffle

- 簡略化すると以下の通り



Secondary Sortを利用したシャッフル

- Reducerが入力を複数とっているかのように見せかける
  - 実際は直和型の要素ごとに並んでいるだけ
  - 入力の区切りごとに切り分ける
- これでまた表現力が上がった
  - Shuffle: [S|T|...] -> [([S], [T], ...)]
  - Reduce: ([S], [T], ...) -> ...

Hadoop Map Reduceの表現力

- できることのまとめ
  - 複数の入力を取れる (MultipleInputs)
  - 入力ごとに好きな型のデータを好きなだけReducerに (Map, 直和型)
  - 複数のデータを自由にグループ化して処理できる (Reduce, Secondary Sort)
  - 複数の出力を行える (MultipleOutputs)
- およそデータフローの計算で必要なことは一通りできる

Hadoop Map Reduceのモデル化

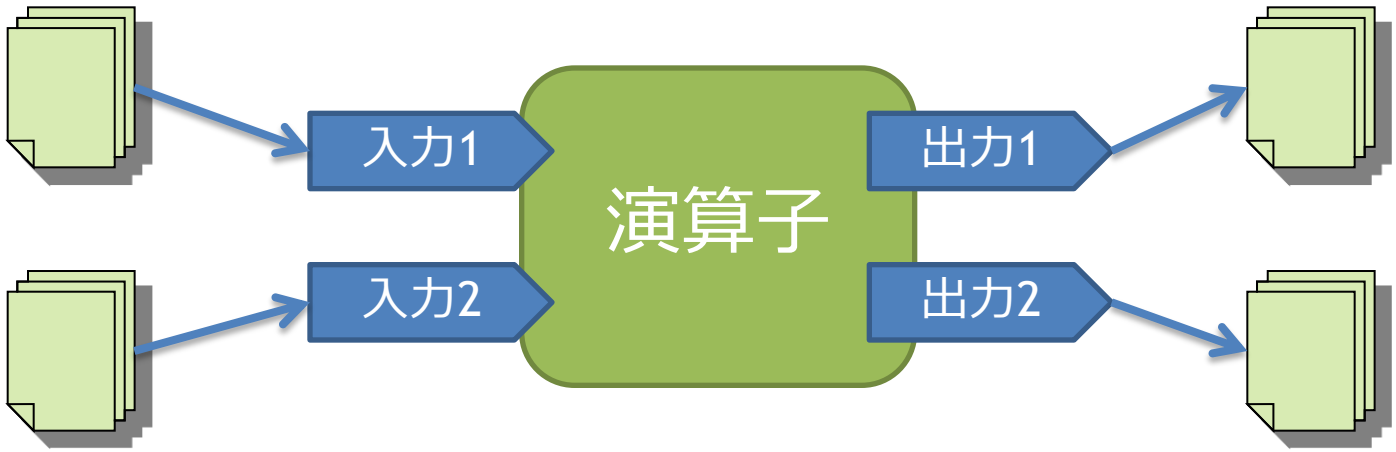
- 型の確認
  - Map1: A1 -> [(S|T|...)]
  - Map2: A2 -> [(S|T|...)]
  - ...
  - Shuffle: [(S|T|...)] -> [([S], [T], ...)]
  - Reduce:
    ([S], [T], ...) -> {b1:[B1], b2:[B2], ...}
- Asakusa DSLではこのモデルを基本にデータフローを構築
  - 上記とできるだけ同じセマンティクスに

第二部: Asakusa DSLのセマンティクス

- Hadoop Map Reduceの表現力
- Operator DSLの表現力
- Flow DSLの表現力
- データフローの実行計画
- まとめ

Operator DSLの表現力

- Operator DSLはデータフロー内の「プロセス」を記述するDSL
  - Asakusa DSLでは「演算子」と呼んでいる
  - Flow DSLで組み合わせてデータフローを構築
- 演算子はファイルを取ってファイルを生成
  - 演算子の「種類」で処理方法が変わる



演算子の種類

- 単一のレコードを入力にとる (Map系)
  - レコードごとにどの順序で処理してもよい
    - 複数のレコードにまたがる処理はできない
  - Mapperと同じセマンティクスを持つ演算子
    - Map: A -> ...
- 複数のレコードを入力にとる (Reduce系)
  - グループごとにどの順序で処理してもよい
    - 複数のグループにまたがる処理はできない
    - グループ化方法は自分で決められる
  - Reducerと同じセマンティクスを持つ演算子
    - Reduce: ([S], [T], ...) -> ...

例: 単一のレコードを入力にとる演算子

- 更新(Update): A -> A
  - 入力されたレコードを変更して出力
- 変換(Convert): A -> B
  - 入力されたレコードを別の型に変換して出力
- 分岐(Branch): A -> {u:A, v:A, ...}
  - 入力されたレコードを条件に応じて振り分けて出力
- 複製(Duplicate): A -> {u:A, v:A, ...}
  - 入力されたレコードを複製してそれぞれ出力

例: 複数のレコードを入力にとる演算子

- 結合(MasterJoin):
  (A, B) -> {joined:C, missed:B}
  - A, Bを結合してCを出力、結合に失敗したBも別途出力
- 集計(Summarize): [A] -> B
  - レコード列を集計して出力
- 合流(Confluent): ([A], [A], ...) -> [A]
  - 複数の入力をまとめてひとつに
- CoGroup:
  ([A1], [A2], ...) -> {b1:[B1], b2:[B2], ...}
  - 気合でJavaのプログラム書く

補足: CoGroup

- Reducerと同じセマンティクス
  - ([A1], [A2], ...) -> {b1:[B1], b2:[B2], ...}
- 複数のListを受け取り、なんでも返せる
  - 気合で命令型のプログラムを書く

Operator DSLのセマンティクス

- 演算子の種類を選んで命令型で記述
  - 演算子を組み合わせる限りは必ずMap Reduceに落とせるように言語を設計
  - 演算子の種類はMap系かReduce系
  - プログラムはJavaのメソッドとして書く
    - 実際には書けることに制限があるが、後述
- システム詳細設計フェーズを強く意識
  - 「適切な演算子を選ぶ」ことで設計が進む
  - 「適切な演算子を提供する」ことで設計をコントロール

第二部: Asakusa DSLのセマンティクス

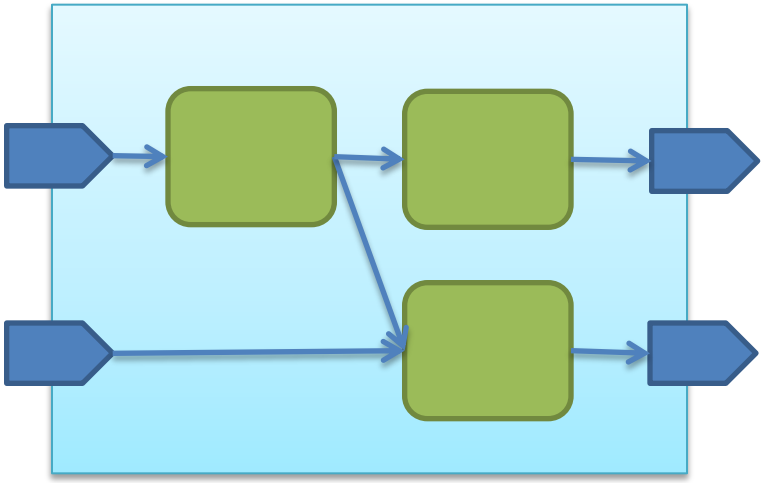
- Hadoop Map Reduceの表現力
- Operator DSLの表現力
- Flow DSLの表現力
- データフローの実行計画
- まとめ

Flow DSLの表現力

- Flow DSLはデータフローそのものを記述
  - Operator DSLで記述した「演算子」を組み合わせる
- Flow DSLからすると演算子は「語彙」
  - DFDでいう「プロセス」に該当
  - 複数の入力がある
  - 複数の出力がある
  - 入出力を繋ぎ合わせてデータフローを表現

Flow DSLと構造化

- データフローと演算子は構造が似ている
  - 複数の入力がある
  - 複数の出力がある
- データフローも演算子として部品化可能
  - Asakusa DSLでは「フロー部品」と呼んでいる
  - 回路図をイメージすると良いかも



Flow DSLと外部連携

- 入出力を外部システムに接続できる
  - データフローの処理開始前に外部システムから取り込み
  - データフローの処理完了後に外部システムに書き出し
- このようなデータフローはトランザクション単位としてマーク
  - Asakusa DSLでは「ジョブフロー」と呼ぶ
  - 途中で失敗してもロールバックできるように連携部分を作りこむ

第二部: Asakusa DSLのセマンティクス

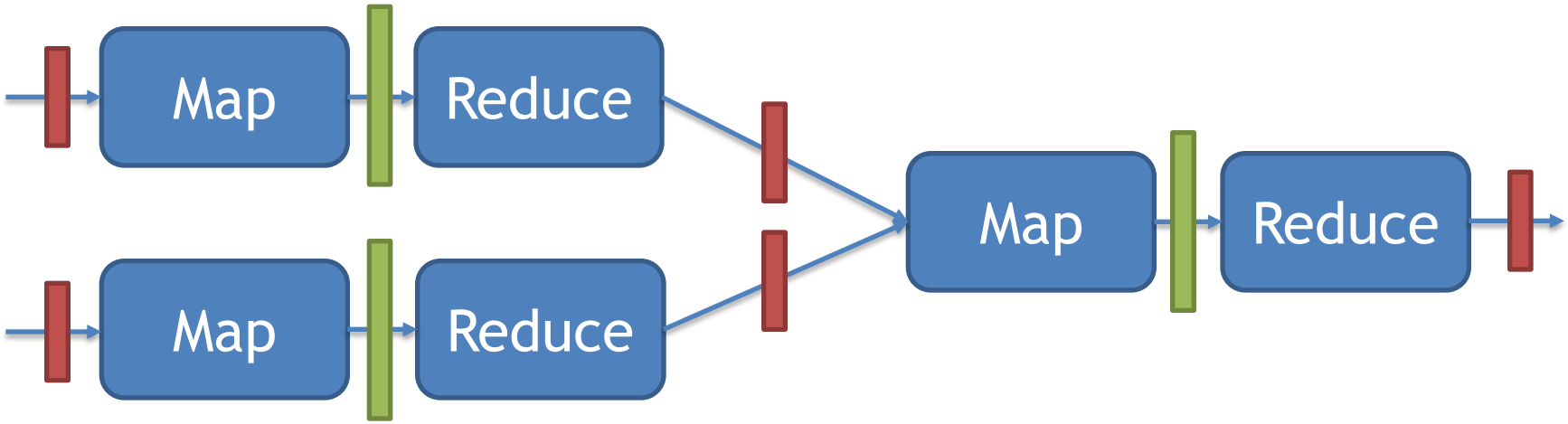
- Hadoop Map Reduceの表現力
- Operator DSLの表現力
- Flow DSLの表現力
- データフローの実行計画
- まとめ

データフローの実行計画

- Flow DSLとOperator DSLの内容から演算子グラフを構成
  - 論理的な処理の流れを表現したもの
  - Map Reduceの単位はこの時点では意識しない
- これをMap Reduceの単位に分割
  - 大したものでもないが「実行計画」と呼ぶ
  - DSLを明確に分離したのは、これをやりやすくするため
    - Flow DSL: フロー型
    - Operator DSL: 命令型

実行計画の方針

- Map Reduceの区切りを考える
  - MapとReduceの間にShuffleが必要
  - ReduceとMapの間をステージ境界と呼ぶ
- これらからMapとReduceを逆算できる

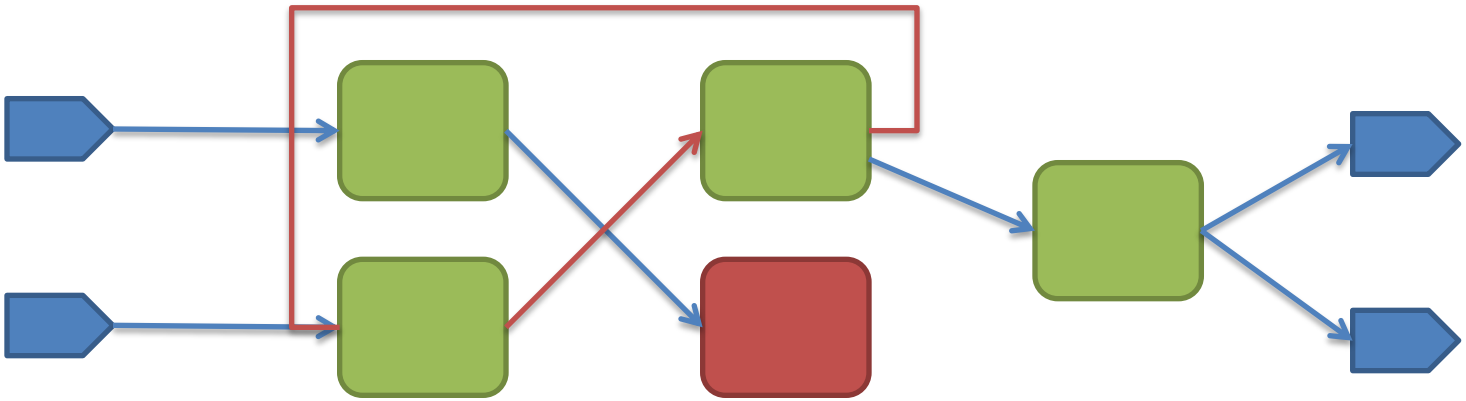


実行計画作成の流れ

- 大雑把に言えば次の流れ
  - 構造の検証
  - フロー部品の展開
  - グラフの最適化 (第3部で紹介)
  - Shuffleの抽出
  - ステージ境界の抽出
  - 恒等演算子の挿入
  - Reducerの抽出
  - Mapperの抽出
  - Map Reduceグラフの構築
  - 入出力の特定

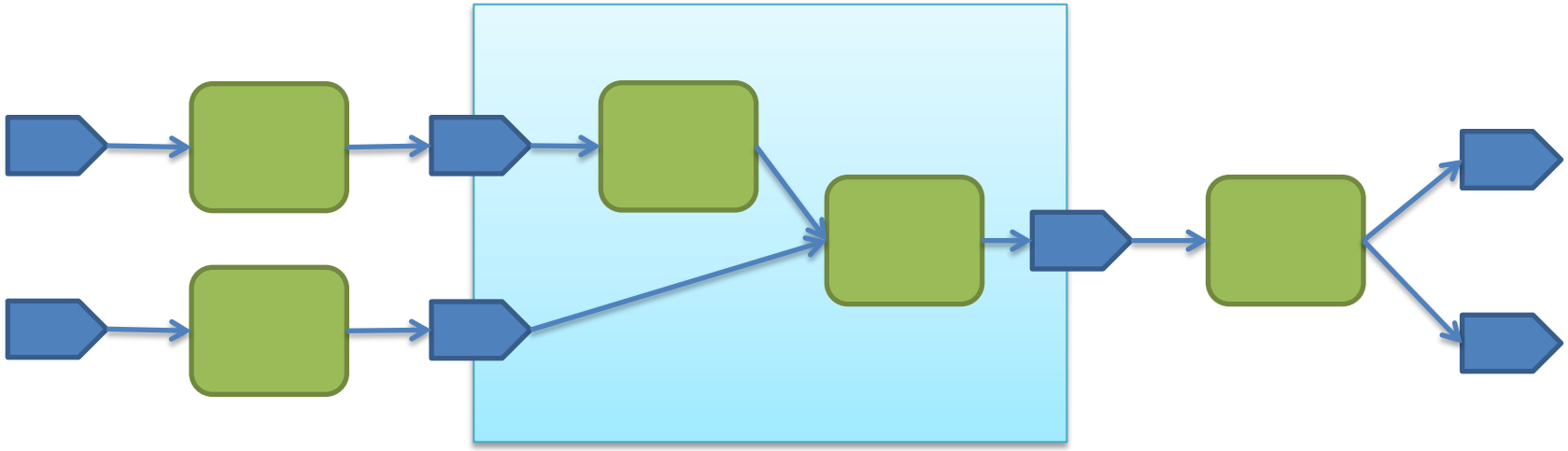
グラフ構造の検証

- グラフに循環がないこと
  - Directed Acyclic Graph (DAG) を前提にすべての処理を行う
- 未結線の演算子がないこと
  - 見つかりにくい問題なので、安全側に



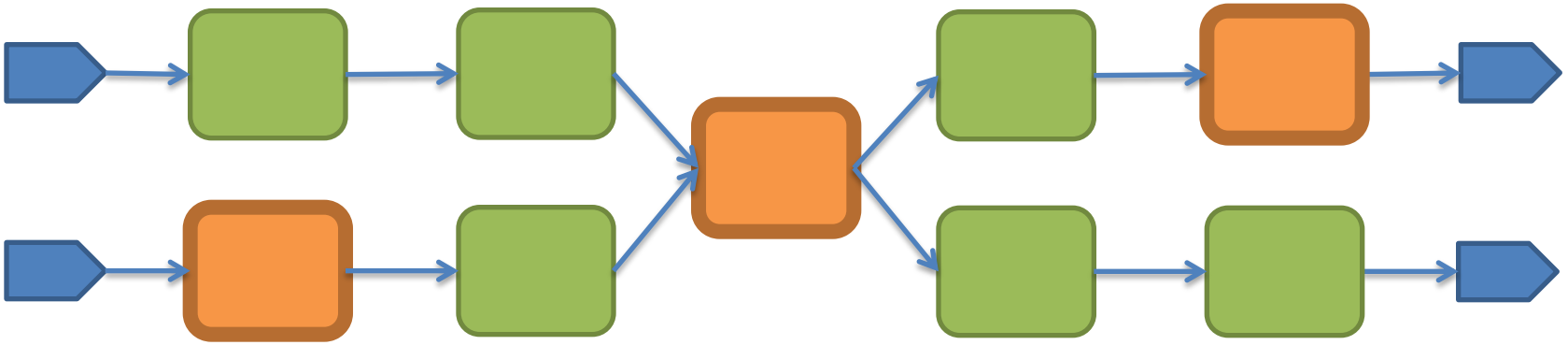
フロー部品の展開

- 構造化のために導入したフロー部品を展開
  - 階層構造を持たない巨大な演算子グラフをつくる
  - 巨大な演算子グラフで広域最適化を適用
- 実際にはフロー部品の「ワク」を外すだけ



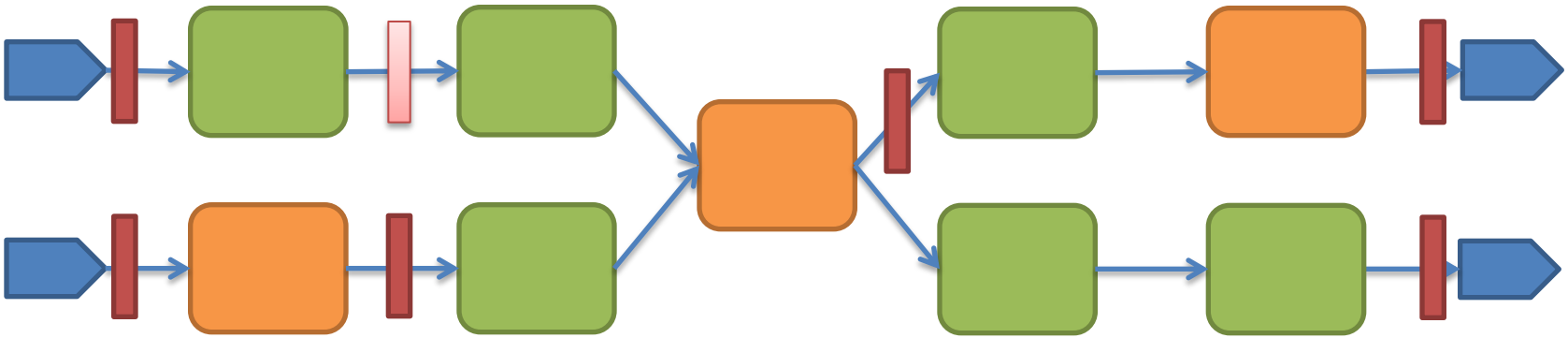
Shuffleの抽出

- MapとReduceの間に来る「Shuffleが必要な処理」を探す
  - つまり、「Reduce系演算子」を探す
  - この手前で確実にShuffleが必要



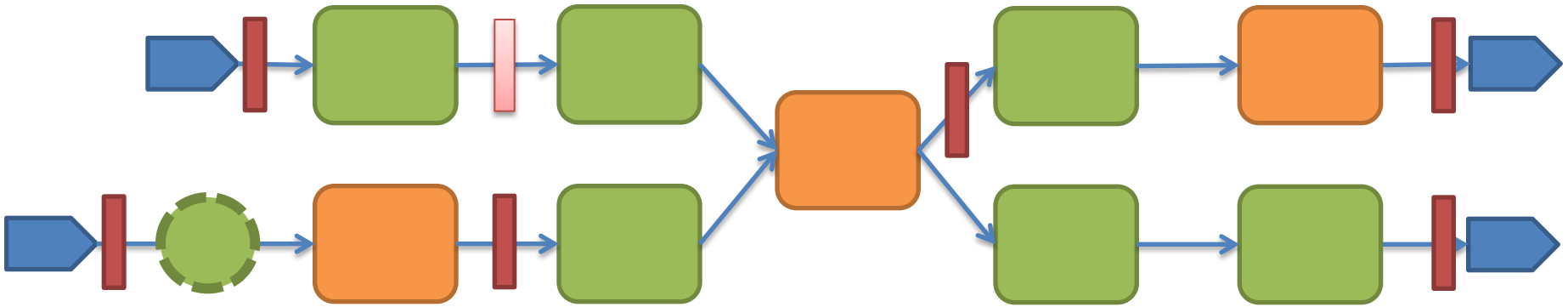
ステージ境界の抽出

- Map Reduceごとの区切りである「ステージ境界」を導入
  - フローの入力と出力は自明なステージ境界
  - ShuffleとShuffleの間になければ勝手に挿入
  - Flow DSLで明示的に挿入することも可能



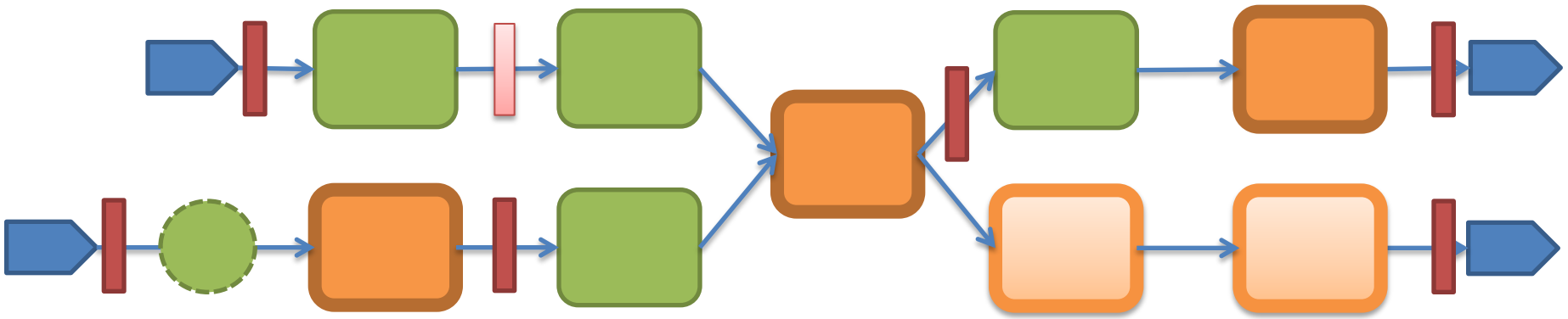
恒等演算子の挿入

- ステージ境界の直後にShuffleが来る場合に、恒等演算子を挿入
  - 「入力をそのまま出力する」演算子
  - これを入れないと、Mapperがないジョブが出来上がってしまう



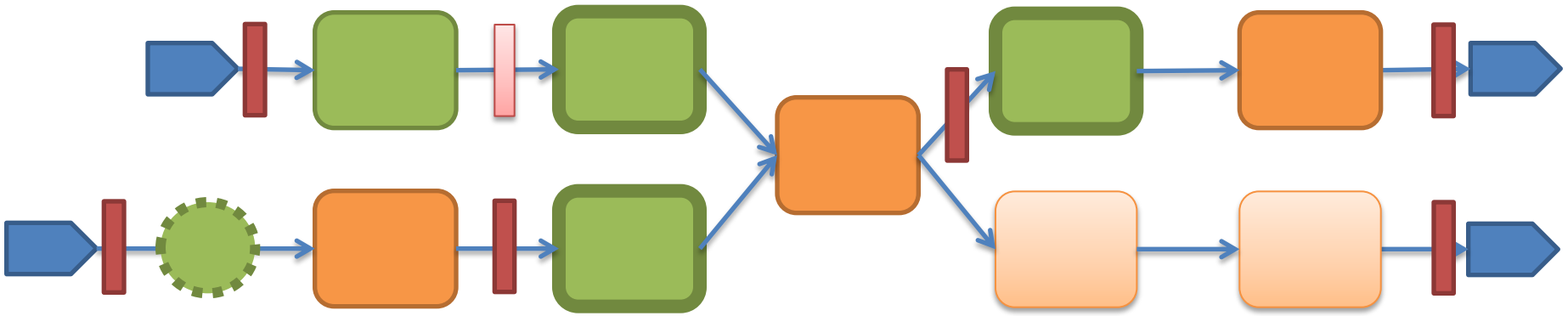
Reducerの抽出

- Shuffleからステージ境界までがReducerに
  - 逆方向でたどると、Reducerが分断されてグループ化に失敗



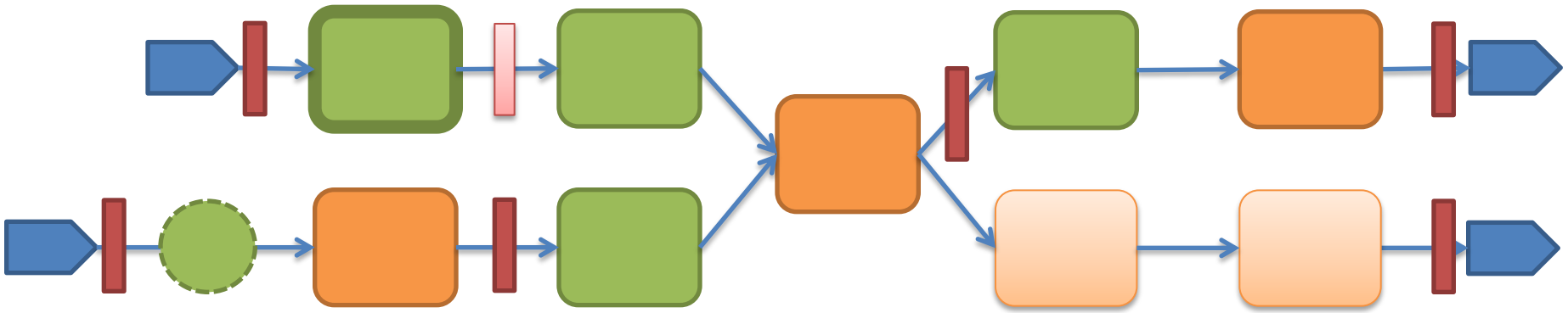
Mapperの抽出 (1)

- Shuffleから逆方向のステージ境界までがMapper
- MultipleInputsで同一ジョブに入れる



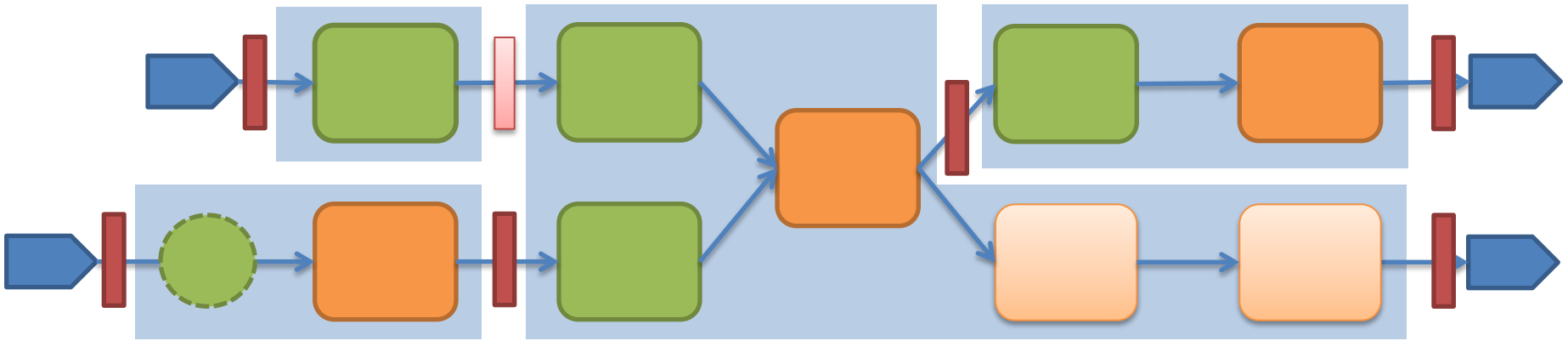
Mapperの抽出 (2)

- ステージ境界から直接ステージ境界にたどり着くものもMapper
  - Reducerがないジョブが出来上がる



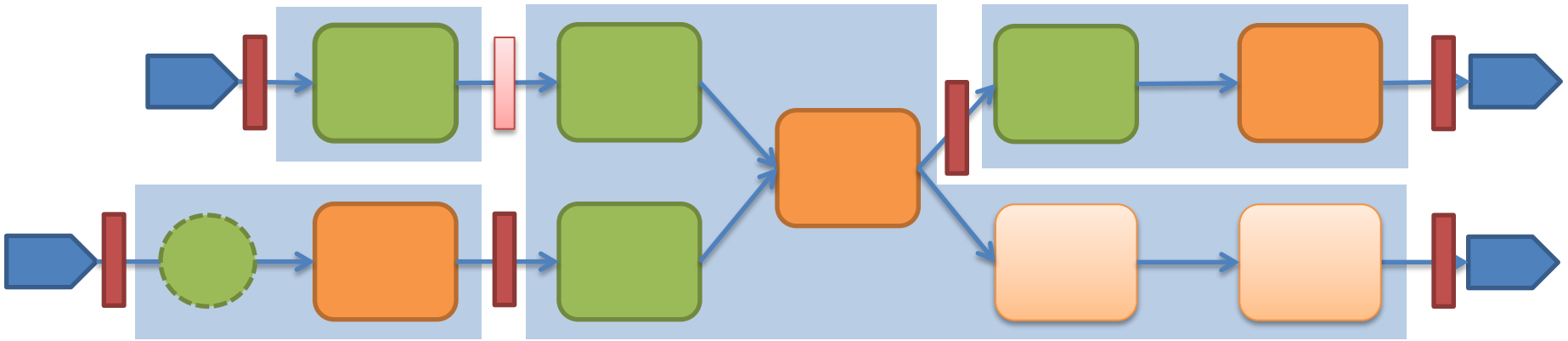
Map Reduceグラフの構築

- ステージ境界からステージ境界までを抽出
  - Reducerとそれに先行するすべてのMapper
  - Reducerが後続しないMapper
- 元の構造と合わせてMap Reduce単位のグラフを構築



入出力の特定

- データフローの入力に直結するジョブは、その入力をそのまま使う
- ほかのジョブの出力に直結するジョブは、その出力を入力に使う

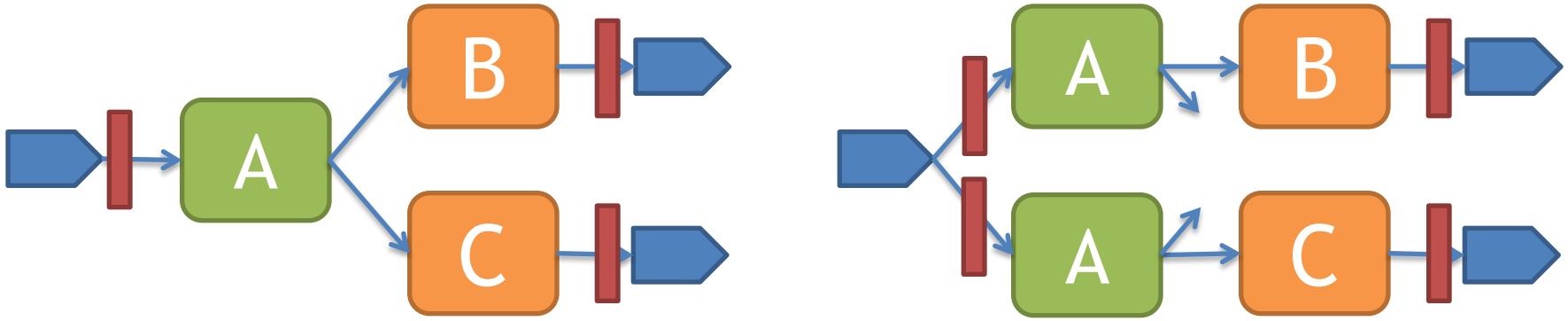


実行計画

- データフローをMap Reduce単位に割当て
  - Map Reduceそのものではなく、Shuffleやステージなどの境界を基準に考える
  - 境界に含まれるものがMapperやReducerに
- 最適化の余地がかなりある
  - Shuffleの削減
  - ステージ境界を適切に挿入
  - MapperやReducerの合成

Discussion: 分断された処理

- 演算子の出力の一部を利用する場合も
  - 内部的には演算子を多重化させて対応
  - 演算子が関数的性質を持てばセマンティクスは崩れない
    - 演算子は出力が入力によってのみ決まること
    - これはOperator DSLの制約 (緩和可能)



第二部: Asakusa DSLのセマンティクス

- Hadoop Map Reduceの表現力
- Operator DSLの表現力
- Flow DSLの表現力
- データフローの実行計画
- まとめ

第二部のまとめ

- Operator DSLはHadoopのMapやReduce処理と同じ表現力をもつように
  - 単一のレコードをとる: Map系
  - 複数のレコードをとる: Reduce系
- Flow DSLは演算子を組み合わせてデータフローを構築
  - データフロー自体も演算子として使える
  - 業務トランザクション境界もここで記述できる
- データフローは実行計画でMap Reduce単位に分割
  - 境界を中心に考えてMapperとReducerを計算
  - 紹介したアルゴリズムをほぼそのまま利用

ここまでくればあとは力技

第三部
コンパイラのコード生成

第三部: コンパイラのコード生成

- Flow DSLのコード生成
- Operator DSLのコード生成
- バッチ全体のコード生成
- まとめ

第三部: コンパイラのコード生成

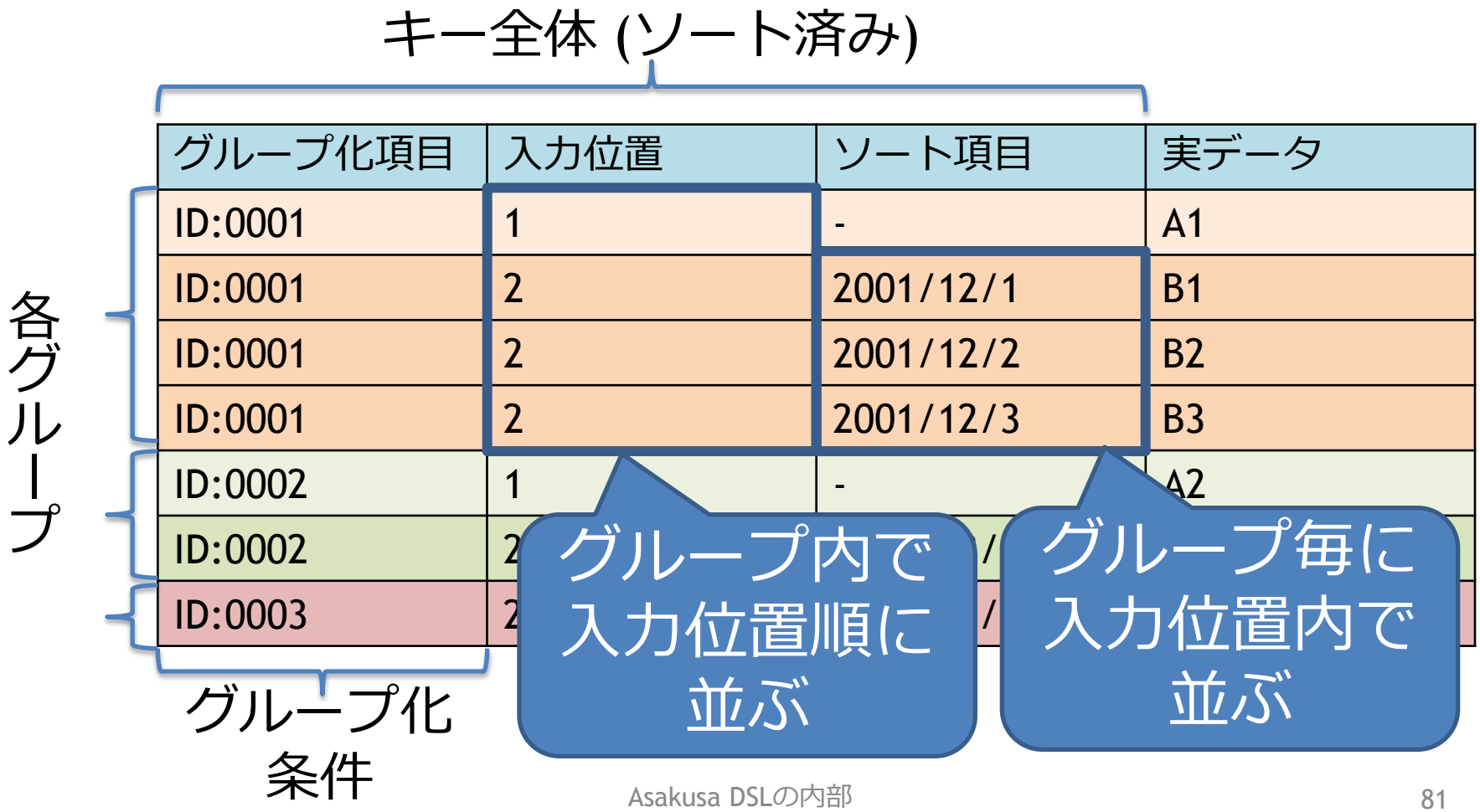
- Flow DSLのコード生成
- Operator DSLのコード生成
- バッチ全体のコード生成
- まとめ

Flow DSLのコード生成

- Map Reduce単位に区切ったので、それ用にプログラムを生成
  - Shuffle Key
  - Shuffle Value
  - Secondary Sort関連
  - Mapper
  - Reducer
  - Client

補足: Shuffleの構造

- 簡略化すると以下の通り



Shuffle Keyの生成

- Asakusa DSLでは「キー」をほとんど意識しない
  - コード生成時に適切なキーを生成
- 後続するReduce系演算子の入力を元にクラスを生成
  - グループ化条件
  - 入力位置
  - ソート順序

Shuffle Valueの生成

- シャッフル時に転送される値のクラスを生成
  - 直和型をJavaで表現

```
public class Union implements Writable {
    public int slot;
    public Hoge slot0;
    public Foo slot1;
    public Bar slot2;
    ...
}
```

Secondary Sort関連の生成

- Shuffle KeyをもとにSecondary Sort用のクラスを生成
  - グループ化のためのComparator
    - グループ化にかかわるキーのみを比較
  - ソートのためのComparator
    - Shuffle Keyのすべてを比較
  - Partitioner
    - グループ化にかかわるキーのみからハッシュ値を算出

Mapperの生成

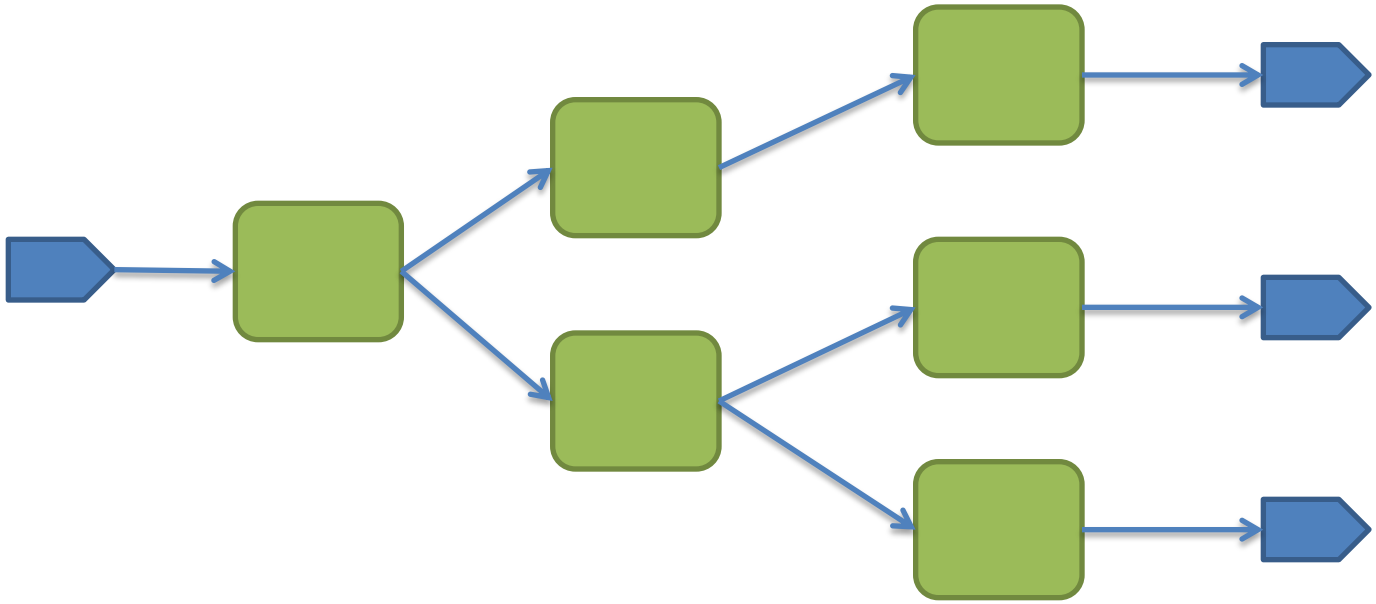
- 演算子を組み合わせてMapper処理を実現
  - Operator DSLで書いたJavaのメソッドを起動するコードを生成
- さらに出力前にShuffle Keyを生成して、Shuffleフェーズに送る
  - 必要な項目をデータから抽出して、キー項目を計算する

Fragment Graph

- 内部的には、Mapper部分の演算子グラフから「Fragment」というものを生成
  - Operator DSLのメソッド呼び出しを組み合わせて処理を実現
  - (コンパイラでいうBasic Blockと同じ概念で分割)
- Fragmentを組み合わせて小さなデータフローを構築
  - 出現する演算子はMap系のみ
  - このデータフローにデータを一つ渡すと、Mapper.map()の1回分の処理を行う
  - Mapperではこれにデータを流すだけ

補足: Fragment Graphの実装

- 次々とJavaのメソッドを起動するだけ
  - Call Treeのようになる
  - 最後に「出力」のメソッドを起動



Reducerの生成

- Shuffleフェーズで作成されたグループの入力ごとにリストを作成して、Reduce系演算子を処理
  - 実際にはOperator DSLの該当するメソッドに処理を移譲する
  - 逐次処理が可能であればリストを作成しないで処理することも
  - 後続に別の演算子があれば、出力の前にFragment Graphでさらにデータを加工

補足: Reducerの入力

- 入力位置ごとにリストを作成
  - 入力が空の場合には空のリストを作成
  - グループ化項目ごとに演算子を実行

グループ化項目	入力位置	ソート項目	実データ
ID:0001	1	-	A1
ID:0001	2	2001/12/1	B1
ID:0001	2	2001/12/2	B2
ID:0001	2	2001/12/3	B3
ID:0002	1	-	A2
ID:0002	2	2001/12/4	B4
ID:0003	2	2001/12/5	B5
ID:0003	2	2001/12/6	B6

Clientの生成

- Map Reduceジョブをキックするクライアントプログラムを作成
  - ここまでに作成したMapper, Reducer, Shuffle情報を渡す
  - job.xmlを作るという手もあるが未実装

Flow DSLのコード生成のまとめ

- ここまでくれば力技の世界
  - 実行計画を元にセマンティクスを崩さなければいい
  - 半月くらい地味なメタプログラミング
- Map Reduceを生で書きたくなくなる
  - Secondary Sortで数回心が折れかけた
  - ちゃんとシリアライザから設計すべき

第三部: コンパイラのコード生成

- Flow DSLのコード生成
- Operator DSLのコード生成
- バッチ全体のコード生成
- まとめ

Operator DSLのコード生成

- Operator DSLとFlow DSLに大きなギャップ
  - Operator DSLではJavaのメソッドを定義
  - Flow DSLでは演算子を組み合わせてデータフローを定義
- Javaらしくないことを実現するために、メタプログラミング
  - Javaのメソッドから演算子の構造を定義
  - 演算子の構造をFlow DSLで使える「語彙」にする

JSR-269: 注釈プロセッサ

- Operator DSLで記述したJavaのメソッドをJavaコンパイル時に自動的に解析
  - Javaコンパイラのプラグインとして提供
  - コンパイルのたびにコードを自動生成
  - スパイラルに生成コードもコンパイル
- IDEと相性が良い
  - オートビルド時に自動生成が走る
  - Operator DSLのエラーをIDE上に表示

演算子ファクトリ

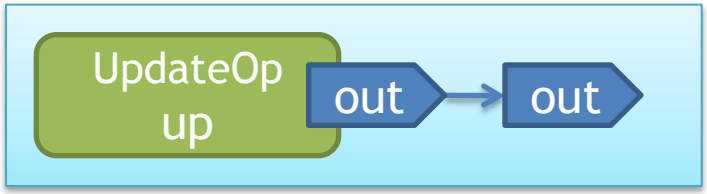
- Flow DSLで演算子を表すオブジェクトを生成するファクトリ
  - Operator DSLから自動生成される
  - フローへの入力や、ほかの演算子の出力を引数にとるメソッドを提供


```
In<Hoge> in;
…
@Override public void describe() {
    OperatorFactory factory = new OperatorFactory();
    // 入力inをUpdate演算子upの入力へ
    UpdateOp up = factory.updateOp(in);
    …
```

演算子クラス

- Flow DSLで取り扱う「演算子」のデータ構造
  - Operator DSLから自動生成される
  - 演算子の出力をフィールドとして宣言

```
In<Hoge> in;
Out<Hoge> out;
…
@Override public void describe() {
    OperatorFactory factory = new OperatorFactory();
    UpdateOp up = factory.updateOp(in);
    // Update演算子upの結果を出力outに追加
    out.add(up.out);
}
```



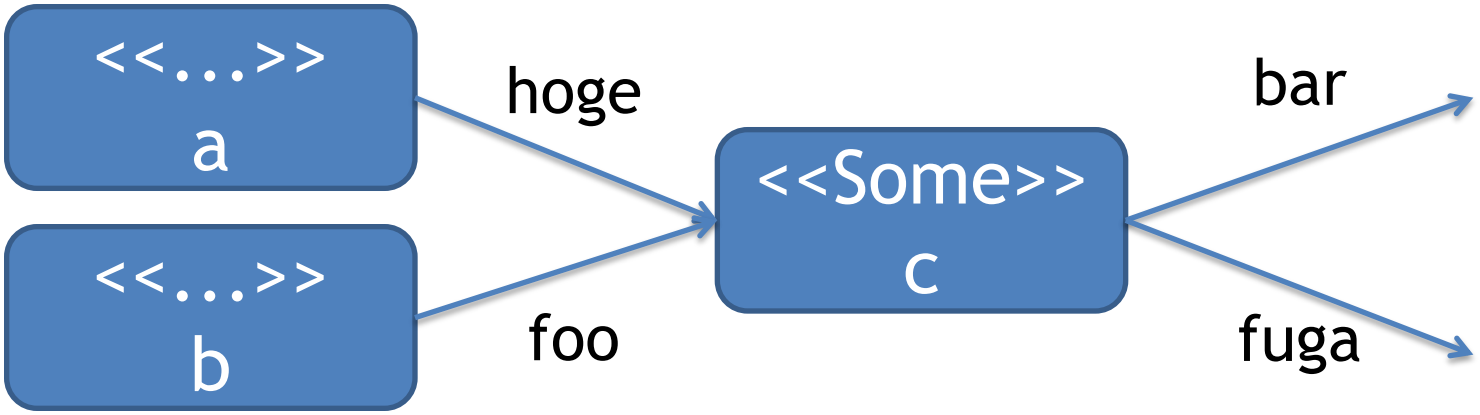
演算子の「語彙」に関する設計意図

- データフロー中の「プロセス」をモデル化
  - プロセスの入力を引数にとって、モデルオブジェクトを生成
  - モデルオブジェクトはプロセスの出力を提供
- DFDがないと書く気になれない
  - DFDとの対応付けはとてもやりやすい (はず)

余談: Java

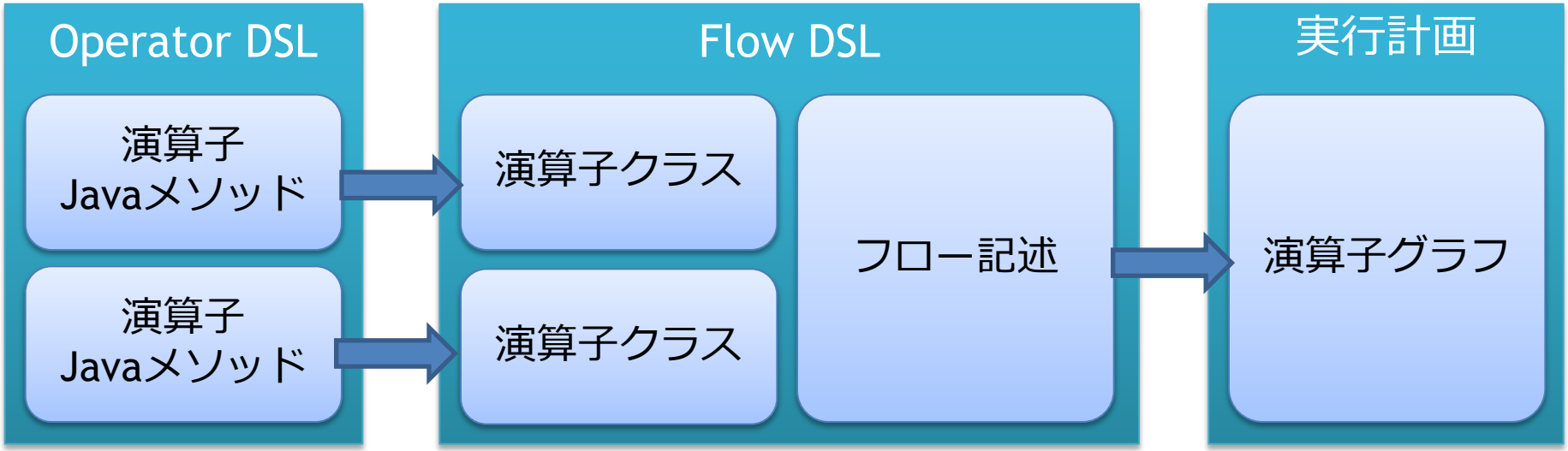
- ホスト言語にJavaを選んで微妙に後悔
  - とはいえ、消去法でこうなった
  - LLに型システム入れるとかでもよかったかも

```
Some c = namespace.some(a.hoge, b.foo);
... namespace.other(c.bar, c.fuga);
```



演算子クラスの位置づけ

- 演算子グラフを構築するためのラッパー
  - 最終成果物で一切利用しない
  - 実行計画を立てるためのデータを入力する「ユーティリティ」
  - IDEにうまくヒントを与えるためだけに存在する



Operator DSLのコード生成のまとめ

- Operator DSLのコードを元に:
  - 演算子クラスを生成
  - 演算子ファクトリーを生成
- Flow DSLではこれらを「語彙」としてプログラムを記述
  - 演算子を結線してデータフローを記述
  - 裏側では演算子グラフを構築している
- 最終成果物にならない
  - 将来的になくなる予感はしている
  - 「絵から自動生成しないんですかね？」と100万回言われた

第三部: コンパイラのコード生成

- Flow DSLのコード生成
- Operator DSLのコード生成
- バッチ全体のコード生成
- まとめ

バッチ全体のコード生成

- 他システムとの連携
- 業務トランザクション
- ワークフロー記述の生成

他システムとの連携

- Flow DSLでは他システムと連携するための情報を記載できる
  - インポーター記述
    - 他システムから何をどのようにインポートするかをDSLで記述
    - ジョブフローの入力に設定すると、他システムからデータを取り込んでジョブの入力にする
  - エクスポーター記述
    - 他システムへ何をどのようにエクスポートするかをDSLで記述
    - ジョブフローの出力に設定すると、その出力結果が外部システムに送られる

例: インポーター記述

- データのインポート方法をJavaで記述
  - 対象システムごとに記述内容は異なる

```
public class HogeFromDb extends DbImporterDescription {
    @Override public Class<?> getModelType() {
        return Hoge.class;
    }
    @Override public String getWhere() {
        return "VALUE > 0";
    }
    @Override public LockType getLockType() {
        return LockType.TABLE;
    }
}
```

例: エクスポーター記述

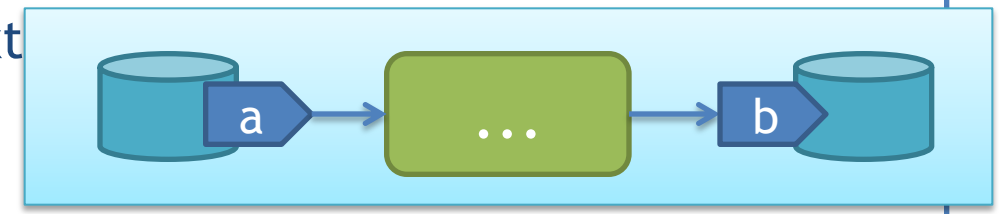
- データのエクスポート方法をJavaで記述
  - 対象システムごとに記述内容は異なる

```
public class HogeToFile extends FileExporterDescription {
    @Override public Class<?> getModelType() {
        return MockHoge.class;
    }
    @Override public String getPathPrefix() {
        return "example/export/hoge-*";
    }
}
```

業務トランザクション

- インポートやエクスポートを行うデータフローは、業務トランザクション処理
  - 失敗したらロールバック処理を起動

```
@JobFlow(name = “example”)
public class ExampleJobFlow ext
    In<Hoge> in;
    Out<Hoge> out;
    public ExampleJobFlow(
            @Import(name=“hoge”, description=HogeFromDb.class)
            In<Hoge> a,
            @Export(name=“hoge”, description=HogeToDb.class)
            Out<Hoge> b) {
    …
```



外部システムとの連携のコード生成

- インポーター記述やエクスポーター記述はコンパイラプラグインで処理
  - 適切なプラグインが入っていないとコンパイルに失敗
- 次の情報を生成
  - インポートやエクスポート時に行われる処理の設定情報
  - インポート処理を起動するコマンドライン
  - エクスポート処理を起動するコマンドライン
  - ロールバック処理を起動するコマンドライン

Batch DSLの解析

- ジョブフローや、ほかのスクリプトを一連のバッチとしてまとめ上げる
  - 「エンドユーザーが認識する処理の単位」を記述
  - 内部的には、バッチに含まれるジョブフローをそれぞれコンパイルしている

```
@Batch(name = "batch.example")
public class ExampleBatch extends BatchDescription {
    @Override
    protected void describe() {
        Work script = runScript("daily-ftp.sh").soon();
        Work jobNet1 = run(JobNet1.class).after(script);
        Work jobNet2 = run(JobNet2.class).after(script);
    }
}
```

ワークフロー記述の生成

- ここまでに作成したMap Reduceジョブや、他システムとの連携方法を一連のワークフローとして出力
  - ほかのワークフローエンジンが一連のジョブを実行するための情報を生成
  - Asakusa Frameworkはワークフローエンジンを内蔵していない

標準のワークフロー記述

- バッチに含まれる処理を順に実行するシェルスクリプト
  - 依存関係のグラフをトポロジカルソートして順に起動するだけ
- 負荷テスト用に少し試すなどの用途向け
  - 実運用には向いていない
  - あえて「experimental.sh」という名前

ワークフローの構造

- バッチ: エンドユーザーの認識単位
  - ジョブフロー: 業務トランザクション
    - イニシャライズ: Hadoopクラスタの初期化
    - インポート: 外部システムからの取り込み
    - プロローグ: 外部システムから取り込んだデータを加工
    - ステージグラフ: Map Reduce処理
    - エピローグ: 外部システム向けに出力データを加工
    - エクスポート: 外部システムへの書き出し
    - ファイナライズ: 不要なデータのクリーンナップ
    - リカバリ: 業務トランザクションのロールバック/ロールフォワード
  - コマンドラインの起動

ワークフロー記述の種類

- コンパイラプラグインで生成するワークフロー記述の種類が増える
  - JP1向けとかあってもいいかも
- Oozieなどを標準にしたほうがよいか
  - 現在の標準は例のシェルスクリプト
  - job.xmlを生成するようにすればできそう

バッチ全体のコード生成のまとめ

- 外部システムとの連携はFlow DSLに
  - インポーター記述
  - エクスポーター記述
  - 自動的に業務トランザクション単位になる
- コンパイラの最終成果物はワークフロー記述
  - ここまでに生成したコードがワークフロー処理としてまとめられる

第三部: コンパイラのコード生成

- Flow DSLのコード生成
- Operator DSLのコード生成
- バッチ全体のコード生成
- まとめ

第三部のまとめ

- Flow DSL
  - 実行計画を元にMap Reduceジョブを生成
  - Shuffleなどの情報も自動的に生成
- Operator DSL
  - Flow DSLの「語彙」を生成
  - 何とかしたいところ
- ワークフロー記述
  - 外部連携や業務トランザクションもDSLで
  - コンパイラは最終的にワークフロー処理を行うための記述を生成

ここまでの過程で、どのような最適化が考えられるか

第四部
コンパイラの最適化手法

第四部: コンパイラの最適化手法

- 最適化手法の紹介
- Asakusa Frameworkの制限
- まとめ

第四部: コンパイラの最適化手法

- 最適化手法の紹介
- Asakusa Frameworkの制限
- まとめ

注意

- 未実装なもの含めて紹介します
  - 調査時間が足りてないので、過不足あればご指摘ください

最適化手法の紹介

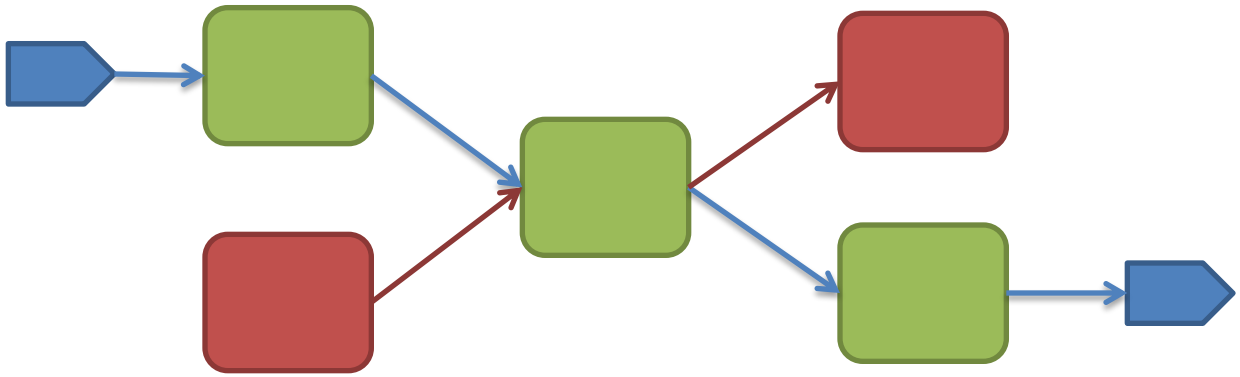
- 死流の除去
- 合流の除去
- ステージの合成
- Map side Join
- 出力の合成
- Shuffleの合成
- プロファイルベースの最適化

最適化のポイント

- Hadoop税が高い
  - ジョブごとに35秒くらい最低かかる
  - いかにジョブを減らせるか
    - 実行計画から逆算するとShuffleを減らせばよい
- 細かいファイルの取り扱いが遅い
  - Hadoopは64MB~128MBのブロックで処理
  - 小さいファイルはオーバーヘッドが相対的に大きい
- その他、分散処理で効くもの
  - 転送量を減らす
  - キーの偏りを減らす

死流の除去

- 以下は死流としてマーク
  - フローの入力から辿り着けないパス
    - 入力が来ない限り演算子の処理は発火しない
  - 副作用やフローの出力に辿り着けないパス
    - 結果が観測されないなら存在しないのと同じ

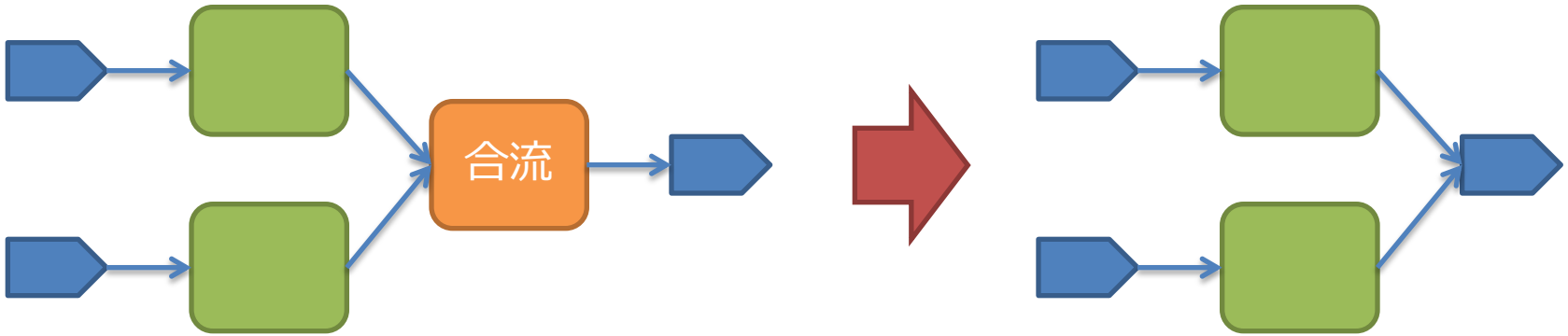


死流の除去の意図

- フロー部品の一部だけを使える
  - フロー部品は部品化されたデータフロー
  - 一部の出力が不要な場合、等
  - 最終結果に影響する演算子のみが発火

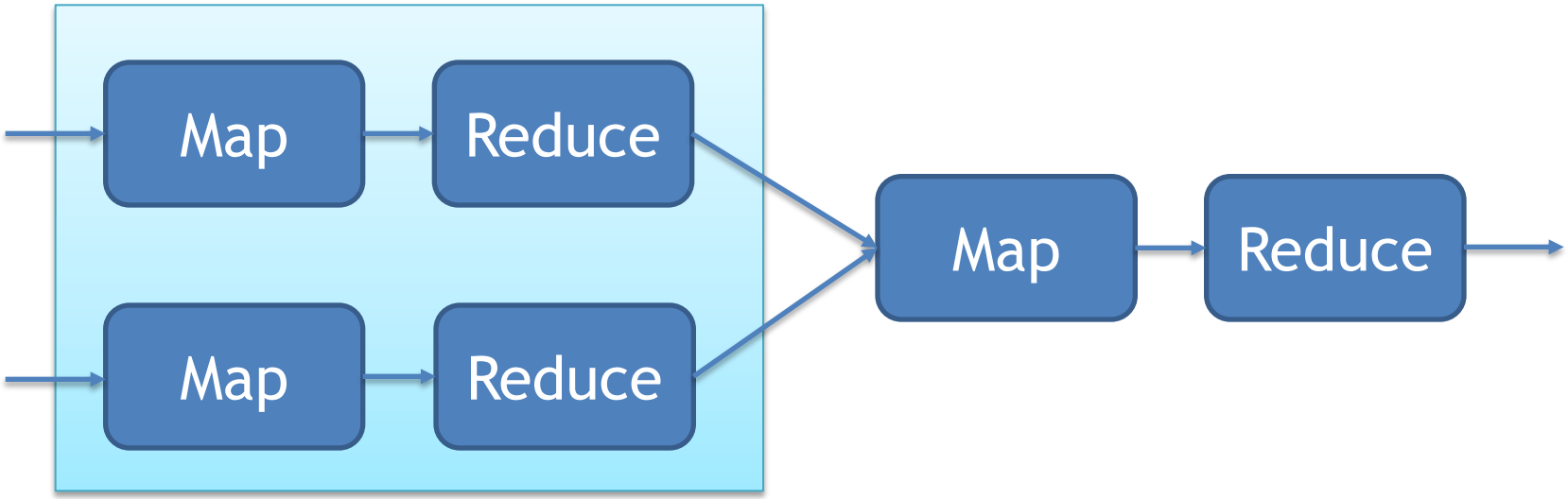
合流の除去

- 合流はわざわざReducerでやることもない
  - Mapperに個別に渡せば処理してくれる
  - Shuffleに個別に渡せば処理してくれる
- Shuffleの回数を減らせる



ステージの合成

- 依存関係のないMap Reduce処理は一つのジョブで実行できる
  - DAG上で先行しない、かつ後続しない
  - DAGの直径までステージ数を減らせる



ステージ合成のポイント

- 複数の処理を同時にShuffleすることで、キーの偏りを軽減
  - 個々のキー種が少なくても、同時に処理すればその分だけキー種が増える
- 一番遅いジョブに引きずられるので、危険な最適化
  - ダイナミックスケジューリングしたほうが普通は速くなる
  - とある理由により現在はデフォルトでONになっている

Map side Join

- 結合対象のデータが十分に小さければ、Reducerでやらなくてもよい
  - Mapperに小さいほうのデータを先に配布
  - 「サイドデータ」として演算子そのものに埋め込まれているようなセマンティクスに
  - 大きいほうのデータだけの入力になり、Map系演算子として取り扱える
- Shuffleの回数を減らすため、Hadoop的にはものすごく効く

出力の合成

- 複数の出力を一つにまとめる
  - セマンティクスの中で何をしてもよい
  - ファイル数が減り、ファイルの粒度が大きくなるため、Hadoop的に効く
- アプローチ
  - 次段の入力が一致する出力を併合
  - ステージを合成
  - ステージ境界の位置を調整
  - 分岐演算子の位置を組み替え

Shuffleの合成

- 並行するReducerが似た条件のShuffleを行っていたら、1回分にまとめる
  - Shuffleの転送量が1回分になる
- 現在の内部表現だと少し遠い
  - Shuffleフェーズを個別の内部表現として扱う必要がある
  - データフローのアセンブリ言語を作りたい

プロファイルベースの最適化

- サンプルデータを流して実行プロファイル情報を作成
- プロファイル情報を元に実行計画を修正
- …ということをやりたいな、というレベル

第四部: コンパイラの最適化手法

- 最適化手法の紹介
- Asakusa Frameworkの制限
- まとめ

Asakusa Frameworkの制限

- 命令型のプログラムを含む
- 静的スケジューリング
- 外部システムとの疎結合性
- ワークフローエンジンとの疎結合性

命令型のプログラム

- Operator DSLはJavaのメソッドを書ける
  - ほとんど自由に命令型で記述可能
- 解析が難しいため、広域最適化に制限
  - かなりここはどうするか迷った
  - 最終的には、「開発容易性」を優先
  - 宣言的な部分はほぼFlow DSLのみ

静的スケジューリング

- コンパイル時に実行計画を確定させてしまう
  - クラスタの状況に応じた動的な最適化が困難
  - 理想的には実行直前に１段分の実行計画を立てる
    - 例:データサイズを動的に観測して結合アルゴリズムを選択
- 障害追跡を考慮した選択
  - 速度を犠牲にしても、障害を追跡しやすい方向に
  - 動的実行計画を実現しつつ障害追跡をやれるかも
    - DSLでかぶせてあるので、同じコードでそのまま速くなる可能性も

外部システムとの疎結合性

- 外部システムと疎結合なため、細かい制御が困難
  - 痛いのはsemi-joinを利用するのが厳しいこと
- これはHadoopクラスタと外部システムが完全に隔離されているという前提
  - Hadoopは直接外部システムに問い合わせを行えず、ワークフローエンジンが制御する
  - 密結合を前提にした機構も作りたいところ
    - HBaseなどをクラスタ内に配置する、など

ワークフローエンジンとの疎結合性

- ワークフローエンジンと連携しにくいため、クラスタ全体のリソース情報を利用した最適化を行いにくい
  - OS不在でバッチ処理を行うのと似た辛さ
- Hadoop内部のリソース管理機構と連携したワークフローエンジンがない
  - 外部システムとの連携まで視野に入れると、現状は妥当ではある
  - Hadoop内部のキューをAsakusa Frameworkで提供することも視野に入れたい

第四部: コンパイラの最適化手法

- 最適化手法の紹介
- Asakusa Frameworkの制限
- まとめ

第四部のまとめ

- 各種最適化
  - 現在はジョブ数を減らす方向を中心に作業中
  - 将来的には転送量等までちゃんとやりたい
- Asakusa Frameworkの制限
  - 開発容易性のために最適化を制限
  - DSLのセマンティクスを崩さずに、将来はもう少し無茶なものを入れたい

ここまで来るのか微妙...

全体のまとめ

まとめ

- Asakusa DSLの設計思想について紹介
  - エンタープライズ分野のバッチ処理をHadoopで
- Asakusa DSLとMap Reduceとの関係について紹介
  - Map Reduceはそもそもどんなことが可能か
  - それに対してDSLはどんなことが可能か
  - DSLがどうやってHadoopのプログラムになるか
- 最適化や今後の展望についても紹介
  - まだまだやれることはたくさん

OSS化について

- コード整理中
  - 実行計画周りは数か月以内に刷新予定
    - データフローアセンブリ言語の導入
    - マルチホスト言語対応
- ドキュメント整備中
  - 今日のスライドが一番詳しいところもあるレベル
  - 一部プロトタイプとして先行実装している感じ
- ３月中に出す
  - 予定で線を引いてます
  - 詳しくは主催者まで

おわり

- 第一部: Asakusa DSLの設計思想
  - どんな言語か、何を考えて作ったのか
- 第二部: Asakusa DSLのセマンティクス
  - Hadoopとどのように関連するか
- 第三部: コンパイラのコード生成
  - どのようなコードを生成しているか
- 第四部: コンパイラの最適化手法
  - どのような最適化があるか