特集号
招待論文

業務系バッチ処理へのHadoop導入についての課題と解決
―Asakusa Frameworkの導入によるバッチの高速化―

神林 飛志[†1]

[†1] (株) ノーチラス・テクノロジーズ

ビッグデータの担い手として広く認識されつつあるHadoopは業務系バッチ処理に並列実行基盤として適用することが可能である．その実際のケースから，課題と解決案を提示する．Hadoopの業務系システム適用における課題は，現在のHadoopエコシステムが提供する開発環境と業務系バッチ処理の開発実態のミスマッチと，Hadoopが提供する可用性・信頼性が業務系システムの要求水準に満たないことに存する．前者はHadoopでの業務系システムのために開発フレームワークAsakusa Frameworkを利用することで，後者はRDBMSとHadoopクラスタを密結合させてシステムアーキテクチャを構築することで解決できる．

1. はじめに

本稿で論考対象とするシステム（以降，当該システムと表記）は，九州の大手小売流通会社の店舗・本部における会計データを作成・確定する処理全般を司るシステムである．既存システムの老朽化への対策と機能の向上を目指して，既存システムから新システムへ全面的にリプレースされた．リプレースの検討は2010年から開始され，2011年から開発が開始され，2013年に正式に全モジュールがリリースされ，本番で運用されている．新システムではIFRS（国際財務報告基準）の要請も考慮し，従来の原価計算の仕組みを売価還元法から個別原価法に変更している．この結果，計算量が従来の1000倍程度にふくれあがることが想定され，バッチ処理の時間を短縮することが要請されていた．

当該システムの特徴は，バックエンドのバッチ処理の高速化・大容量化に対応するためにHadoop[1]を全面的に採用したこと，Hadoop上の開発フレームワークとしてオープンソースソフトウェアであるAsakusa Framework[2]（以降，Asakusaと表記）を利用したこと，ならびに実行基盤としてクラウド環境（Amazon Web Services, 以降AWSと表記）を全面的に採用したことにある．

本稿の目的は，当該システムの概況を機能面から明確にした上で，Hadoopを採用するに至った経緯とその課題，またその課題解決の方策としてAsakusaを利用した詳細や実際のアーキテクチャを詳らかにし，業務系システムへのHadoopの適用に対する示唆を示すことである．

2. 業務系バッチ処理へのHadoop適用の課題

2.1 業務系システムのバッチ処理について

当該システムを包含する大規模な業務系システムでのバッチ処理は，大別すると以下の形態に分類することができる．

- データクレンジング処理

外部システムから当該システムに入力された，またはユーザにより登録された，取引データとマスタデータとの突き合わせを行う．不正な取引データが登録されないようにすると同時に，取引データのデータ項目の登録が不十分な場合にマスタデータから適切な補完を行い，当該システム内部で不整合が起きないようにする．通常のRDBMSでのバッチ処理では，取引データとマスタデータの結合処理によりパフォーマンスが劣化することがあり，課題となることが多い．

- 突合処理

取引データ同士の突き合わせを行い，それぞれの取引データの正当性を相互に確認する処理．突き合わせを行う両者それぞれの取引データのアンマッチを検出する必要がある．したがって，RDBMSでこの処理を実行すると，取引データ同士の結合処理がループ処理をしながら順に

データを突き合わせる大規模な完全外部結合になり，一般にパフォーマンスが出ない．

• 集計処理

取引データを一定の単位で集約し，ツリー上のデータ構造に沿って集計する処理．ツリー上の集計処理が多段の結合集計になることが多く，パフォーマンス劣化が問題になることが多い．

上記の処理はいずれも，現在のRDBMSのバッチ処理では，データを多重にループさせる結合戦略をとらざるを得ないことが多い．このとき，処理するデータサイズがメモリ上に展開できない場合は，ディスクIOが頻発し，結果としてパフォーマンスが劣化することが多い．

これらの処理に対してHadoopを利用し，分散並列処理を利用することでパフォーマンスの劣化を防ぐことが可能である．特にHadoopの採用している処理フレームワークであるMapReduceは，独立したキーを基準にデータを分割し，そのキーの単位で突合・集約するのに適しているフレームワークである．上記のカテゴリーに属するバッチ処理は，同時に多数の独立しているキーで並列処理が可能であるという特徴を持つ．Hadoopでの分散並列処理がパフォーマンス維持にはきわめて有効である．

2.2　Hadoop 適用での課題

上記のように，業務系システムでのバッチ処理は，Hadoopで利用されているMapReduceとの親和性が高い．しかし，一方でHadoopを取り巻くエコシステムとの相性は決して良いとは言えない．

現在Hadoopを支えるエコシステム[3]，すなわちPig, Hive, Oozie, Mahoutなどの想定しているユースケースは，クラスタ分析や機械学習を利用した，レコメンデーションエンジンやそれに類する推論の仕組み（分析処理用の道具立て）に偏っている．本来MapReduceのアルゴリズムは特定の業務やユースケースに制限されるものではなく[4]，きわめて中立性の高い処理フレームワークである．しかしながらHadoopのエコシステムは，Hadoopが「ビッグデータ」の処理基盤として脚光を浴びていることに大きく影響され，前述の分析処理用の道具立てに特化している．このため，以下に列挙するように業務系のバッチ処理とは相性がきわめて悪い．

2.2.1　開発環境の課題

• フールプルーフが徹底されていない

プログラムエラーやデータ不正の結果が事後的に検出されることが多く，システムの安全設計が徹底されていない．このため利用者自身がMapReduceのパラダイムを理解し，品質の高いコードを書くことが要求される．規模の大きな業務系システムの開発のように，かかわる人間の能力にばらつきがあるようなケースでは，インテグレーションの難度が高くなり，品質が安定しない．

• 複雑な例外データフローを処理できない

Pig/Hiveでは出力を分岐する仕組みがサポートされておらず，特に例外処理を分岐させることができない．業務系バッチ処理では例外フローを準正常フローとして扱うことが要請されるため，データフローでの例外処理をサポートすることが必要になる．Pig/Hiveのような現状のHadoopのエコシステムではこのような処理フローはサポートされず，したがって業務バッチ処理を開発することが困難である．

• テストツールなどのテスト環境が不十分

前述の分析処理用システムでは，そのユースケースとして，アドホックな試行錯誤・クエリ発行を前提としていることが多い．業務系システムのように，一定の入力に対して障害の有無にかかわらず，出力が安定的・継続的に常に同じになるような高い品質を求められることは少ない．結果として，Hadoopのエコシステムで準備されるテストツールなどのテスト環境は，業務系のシステムインテグレーションに対しては不十分となっている．

2.2.2　アーキテクチャの課題

• バッチ・トランザクション処理の欠如

前述の分析処理用システムでは，複雑な処理フローを連携させて，ロングバッチを処理するケースは例外であり，一般にトランザクション処理は必要ない．このためHadoopではトランザクション処理がサポートされていない．これに対して，業務系バッチ処理は複雑な連携処理が高度に入り組んでおり，途中のバッチ処理の中断により処理全体がアボートされないようなトランザクション処理が必須になる．

• ファイルシステムの障害への対策

HadoopのファイルシステムであるHDFS(Hadoop Distributed File System)では，ファイルシステムのメタデータを管理するNameNodeが単一障害点になっている．この単一障害点がHadoopの可用性の低さの原因になっている．現状ではNameNodeのHA構成（高い可用性High

Availabilityを備えた構成）がとれるような対策がとられつつあるとはいえ，NameNodeのクラッシュによる分散クラスタ系全体への障害が起こった場合の障害対策は十分ではない．Hadoopのファイルシステムである HDFSに大きな障害が発生した場合の復旧手順については，システム全体のアーキテクチャレベルでの対策を立てておく必要がある．

3. Asakusa-Hadoopによる業務系バッチ処理の開発事例

3.1 システムに対する要件

本稿で取り上げる当該システムは以下の複数の主要モジュールから構成されており，それぞれに固有の業務サービスの提供が必要とされた．

• 売上売掛モジュール

売上確定処理と債権管理処理を行う．売上確定処理はPOSデータから上がるローレベルの取引データを日報データに処理し直すことを主目的にしている．ユーザの売上データの確定データを日次で作成する．取引データとマスタデータとの突き合わせによるデータクレンジング処理が行われる．債権管理処理はユーザの顧客ならびに取引先に対する債権の金額の確定，請求処理，入金時の債権の消し込みを行う処理からなる．債権金額の確定は日々行うが，請求・消し込みは締め処理のタイミングで確定する．消し込みバッチ処理は計上データと消し込みデータの突合処理になる．

• 仕入買掛モジュール

債務管理全般を処理する．債務は在庫管理にかかわる買掛金の処理と費用処理にかかわる未払金の処理を行う．買掛金のデータは仕入関連の取引データから生成し，未払金のデータは主として費用系のマスタデータと手入力での伝票処理から生成する．買掛金処理と未払金処理も，債務の計上データを作成し，EDIからの請求データを取り込み，計上分と突合処理を行い，必要な支払いデータの生成を行う．

• テナント管理モジュール

ユーザの店舗に出店しているテナントのフィー計算と決済処理を行うクリアリング機能を有する．フィー計算は，確定された売上預かり金から，仕入買掛モジュールで計算される費用とテナントマスタにより計算された手数料を相殺することで計算する．クリアリング処理では，マスタデータとのクレンジング処理と，計上データと決済データとの突合処理が必要になる．

• 原価計算モジュール

上記の各モジュールおよび接続された他システムからの収益データと費用データを集約し，損益計算を行う．売価還元法ベースの処理と個別原価法ベースの処理の2つの手法で損益計算を行う．損益を確定した後に，組織別・カテゴリ別の損益のツリー構成への集計処理を行う．

また，当該システムはユーザの管理会計のすべて，ならびに財務会計へのデータ生成を担うシステムであるため，数時間超に渡るシステムダウンと業務データの消失は許容されない．Hadoopクラスタの大規模障害発生時についても，速やかな復旧とデータの再構築が要求される．

要求された上記要件は，RDBMSの処理ではパフォーマンス劣化は容易に推定できるバッチ処理が多く，これらの処理を適切なパフォーマンス下で実現するためにHadoopの導入を検討することになった．ただし，Hadoopの導入に際しては，前述のHadoop適用のそれぞれの課題を事前に解決しておく必要があった．

3.2 システムの開発と実装

当該システムでは，要求された業務要件を達成し，2.2節 Hadoop適用の課題をクリアするために，開発基盤としてAsakusaを採用し，処理基盤のアーキテクチャを以下のように実装している．

3.2.1 Asakusaとは

当該システムの開発では，これらの課題のうち2.2.1項開発手法の課題を解決し，Hadoop上でバッチ処理を開発・実行するために，(株)ノーチラス・テクノロジーズが開発し，オープンソースとして公開しているAsakusaを利用した．Asakusaは，Hadoopで業務系システムを開発するためのフルスタックの開発フレームワークであり，上記の課題を克服するために開発された．

データフローとデータモデルを記述するJavaベースのDSL（Domain Specific Language）を提供しており，与えられたDSLでバッチ処理を記述すると，Asakusaのコンパイラがバッチ処理を解析し，適切なMapRuduceのプログラムを生成する．

特に以下のような特徴を持つ[5][6][7].

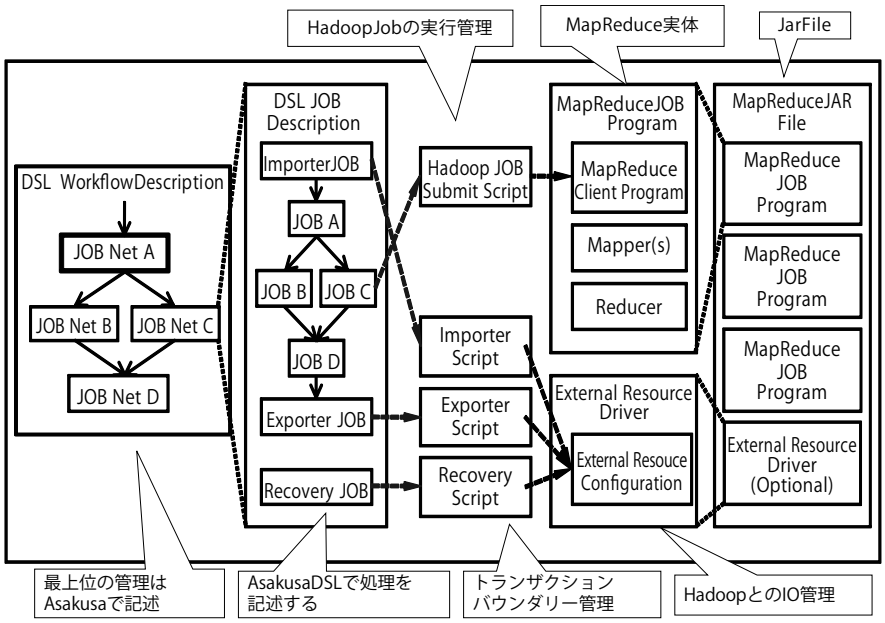

図 1　Asakusa 実行時構成

• フールプルーフの徹底

フレームワークの設計思想として，設計コストに見合う実装品質をフレームワークで確保するという方針を持っている．設計が事前にしっかりされているのであれば, 実装選択の余地がほとんど残らないようにしている．この結果，実装時の柔軟性には欠けるが，結果として，Hadoopを必ずしも熟知していないエンジニアであっても設計がしっかりしているのであれば，一定の品質のコードが書けるようにしてある．DSLのコンパイラがデータモデル／データフローについて強い型付けを行っており，コンパイル時点に静的なチェックをかけてプログラムの品質を担保している．このため，大規模バッチ処理のインテグレーションのようにチームや大量の人員で開発をする場合に，高品質のバッチ処理を大量に生産することが可能になっている．

• 例外処理フローのサポートと実行計画の最適化の強化

例外処理を中心とした複雑な処理分岐をサポートしている．データフローを構成する各処理が複数の入力をとり，複数の出力を行うことが可能になっている．このため，多岐にわたる例外処理を記述し，統合処理することが可能である．また，記述された複雑なフローの実行コストを低減する，ということに焦点をあてて実行処理の最適化を行っている．

複雑な実行フローをそのままHadoopで実行する場合，MapReduceのステージ数が極端に増加して，実行時間が延び，効率が著しく低減する．このためコンパイラが，処理の合成が可能な処理はコンパイル時にステージ合成を行い，MapReduceの段数を減らしている．

• テストツールの充実

バッチ処理のテストとして，テストツールを充実させている．検査データの管理を，オフィスツールであるExcelで行うことが可能になっている．データモデルとデータフローを記述すると，テストデータを投入することが可能なテストモックをAsakusaが自動生成する．これらの仕組みを利用してExcel上でテストデータを管理できるようになっている．

3.2.2　実装したアーキテクチャ

3.2.2.1 全体構成

当該システムの構成は，フロントエンドのシステムとバックエンドのシステムに大きく分けられる．

フロントエンドでは，画面入出力の制御と他システムから当該システムへのデータの受け渡しの制御を行い，オンライン処理を実行する．バックエンドでは，フロントエンドから受け取ったデータに対して必要なバッチ処理を施し，フロントエンドへデータを渡し戻す機能を有する．

• フロントエンドの構成

エンドユーザに対するフロントエンドの入出力はブラウザで処理する．入出力のブラウザは複雑な業務データを処理するため，リッチクライアント的な作りとなっており，開発および実行はJavaScriptを利用している．通

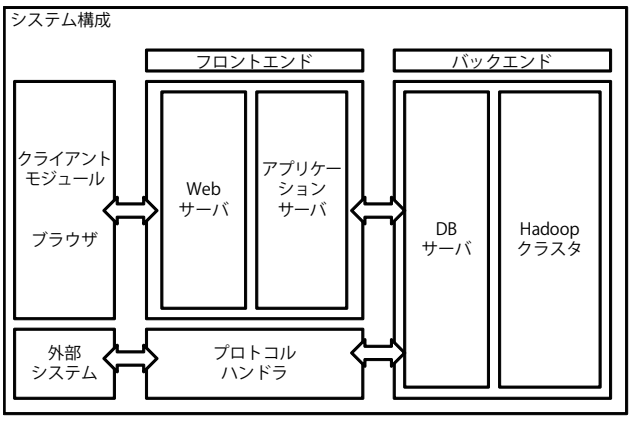

図 2　システム全体構成

常のオープン系のWeb処理のアーキテクチャを採用し，Webサーバとアプリケーションサーバで構成している．

• バックエンドの構成

バックエンドはDBサーバとHadoopクラスタで構成しており，DBサーバはRDBMSを利用している．DBサーバがバッチ実行に必要なデータを後段のHadoopクラスタに転送し，また，Hadoopクラスタ上での処理結果をDBサーバに書き戻しする処理を行う．また，DBサーバ上でフロントとバックエンドのデータ通信制御を行い，バッチ処理のトランザクション管理を行うと同時に，Hadoopクラスタへのアクセス制御も行っている．システム内部のデータ通信がすべてDBサーバを経由することになるため，データの保全をDBサーバで行うことが可能になる一方で，DBサーバのIOがシステム全体のボトルネックになる可能性がある．

3.2.2.2 アーキテクチャの課題の解決

当該システムでは，2.2.2項アーキテクチャの課題を解決するために以下の仕組みを実装した．

• DBサーバにおけるバッチトランザクション制御の導入

バッチ処理の失敗時のリトライのコストを最小限にするため，バッチ処理中のロック制御とリトライのための明示的チェックポイントを実装した．Asakusaでは，HadoopクラスタのNameノードのHA構成の信頼性が低いことに起因する可用性の課題を認識した上で，生成データをDBサーバに書き戻すことによって，バッチ処理のチェックポイントを実現している．バッチ処理が途中で異常終了した場合でも，チェックポイントからデータを復旧し，バッチ処理を再試行することが可能である．

また，このロック制御により，バッチ処理とオンライン処理の間の，データに対する排他制御を行う．トランザクションの排他制御は，処理一般をオンライン処理とバッチ処理に分けて，バッチ処理中は書き込みロックをとる仕組みにした．オンライン処理は，バッチ処理が行われていない場合は，読み込み・書き込みが可能であるが，バッチ実行中は書き込みは許可されない．ロック取得の優先権はバッチ処理にある．オンライン処理のロック取得をバッチ処理にオーバーライドされた場合は，オンライン処理は書き込み時に失敗する．この排他制御により，バッチのリトライの冪等性の確保している．

• システム全体の可用性の確保はRDBMSで行う

当該システムでHadoopクラスタの大規模障害発生時には，Hadoopクラスタ上のデータの復旧をあきらめ，すべてのデータをRDBMSから再構築する手法により，システム全体の可用性を確保する．HadoopクラスタよりもRDBMSの方がシステムの信頼性・可用性が高い．データを一度すべてRDBMSに格納し，システム障害時のデータ消失を防ぐとともに，Hadoopクラスタに大規模障害が発生してもシステム全体が復旧不能に陥らないようにした．

上記の課題解決は，DBサーバとバッチ処理を実行するHadoopクラスタを密結合することによって実現したものである．しかし，その代償として，バッチ処理されるすべてのデータがRDBMSに必ず保存され，その結果，バッチ実行時にRDBMSからHadoopクラスタへ大量のデータ転送が起こるというボトルネックが発生する．このボトルネックを解消するために，Hadoopクラスタ上にRDBMSデータのキャッシュデータを作成し，更新処理のかかったデータのみをDBサーバからロードする構成とした．キャッシュデータは，バッチ・アプリケーションから透過的にアクセスすることが可能であり，キャッシュの有無をアプリケーションが意識することはない．

4. システム開発・運用から得られた知見

4.1 Hadoopは業務バッチ処理との親和性が高い

業務系バッチ処理とHadoopの親和性が非常に高いといえる．与えられた業務処理でのパフォーマンス劣化は観察されなかった．これは業務系のバッチ処理が，締め処理や全件データの参照または更新が多いこと，ならびに各データ行にかかわる処理同士の依存関係が低いケースが多いことによる．全件データに一度にアクセスし，かつそのアクセスのロジックの並列性が高いMapReduceは，業務バッチ処理の実行に適しており，その実行基盤としてHadoopはきわめて有効である．500GBから数TB程度のデータサイズを処理するのに従来5～20時間程度の時間を要した複数の会計バッチ処理を，30分～3時間程度で処理できるようになった．

4.2 現在主流のHadoopのエコシステムでは業務系バッチ処理の大規模開発は難しい

現在のHadoopのエコシステムは，クラスタ分析や機械学習を利用した，レコメンデーションエンジンや推論の仕組みとしての分析処理用途に特化しており，業務系バッチ処理には向いていない．現在のHadoopエコシステムが暗黙に想定している開発・運用要件と，業務系シ

ステムの開発で要求される開発・運用要件の違いを分析し，その違いを吸収する仕組みが必要である．その主たる違いとして，開発体制の違い，扱うデータ種類の数の多寡，処理の内容とそのフロー制御の複雑さ，システム開発・運用への品質要求，が挙げられる．Hadoopを業務系向けに利用できるようにするAsakusaのようなフレームワークがなければ，Hadoopの業務系適用は困難であると思われる．

4.3 Hadoopクラスタ外にあるDBからデータをクラスタに分散させる，またクラスタからDBに書き戻すコストが高い．

4.3.1 RDBMSからHadoopクラスタへの転送は，構成アーキテクチャの要素技術に大きく影響される

• RDBMSからのデータダンプの手法

データダンプの手法とそのパフォーマンスは，各RDBMSの製品によって大きく異なる．各RDBMSごとにデータダンプの手法や提供されているツール，そのパラメータ設定が異なるため，採用するRDBMSごとに最適なデータダンプ戦略を採用することが必要になる．たとえば，強い最適化が施される製品では，単純なSQLをjdbc経由で処理しても最適なパフォーマンスが得られる．また，最適化がほとんど施されない製品では，テーブルデータをSQLを介さずにダンプするツールを利用し，複数のプロセスでテーブルを並列にダンプし，ストリーミング的にネットワークに渡すという戦略を選択する．

• HDFSへの分散コピーの利用

RDBMSからダンプしたデータのHadoopクラスタへの展開は，単純なデータコピーではなく，分散コピーの手法を利用することにより，パフォーマンスが向上する．通常はHadoopエコシステムで提供されているSqoopを利用することが有効である．また，クラウド上で独自のHadoopを提供しているようなケースでは，独自の分散コピーのツールが提供されることがあるので，これを利用する．

• ハードウェアや環境のパフォーマンス

N/Wのスループットとデータのダンプ元のRDBMSのIOパフォーマンスは，データのクラスタへの転送に大きく影響する．それぞれがボトルネックになることがあるので，両者のパフォーマンスを見ながら，よりスループットの上がる環境を導入することが望ましい．プロジェクト経験では，RDBMSのIOパフォーマンスが最初のボトルネックになることが多かった．

4.3.2 Hadoopクラスタ上に，キャッシュ機構を実装することは転送量を削減するという点で効果がある．

Hadoopクラスタ上にバッチ処理で必要なデータを一時的にキャッシュすることで，RDBMSからHDFSへのデータ転送量を大幅に削減する事が可能である．バッチ処理をHadoop上でキックするときに，必要なデータのタイムスタンプを識別して，キャッシュされたHDFS上のデータとRDBMS上のデータを比較し，更新がない場合は，キャッシュされたデータのみでバッチ処理を実行する．この手法は，HadoopクラスタとRDBMSとのデータ転送量を減らすことにつながり，バッチ実行のオーバヘッドを削減できる．

4.3.3 Hadoopクラスタ上のデータを書き戻すコストの削減は困難である．

Hadoopで生成されたデータを，元のRDBMSへ書き戻す処理はRDBMSの構成によっては非常にコストがかかる．特に，複雑なインデックスを張っているDBにデータを書き戻す場合は，スループットが著しく落ちる．これはHadoopの問題ではなく，むしろRDBMSでの大量データの更新処理のパフォーマンス劣化の問題に還元されるため，本稿の範囲を逸脱するが，問題点として指摘しておく．当該システムでは，インデックスを張ったRDBMSに直接データ書き戻す手法と，いったんRDBMS上のデータをトランケートして，データをバルクロードし，その後インデックスを再構築する手法とをそれぞれ比較して，よりパフォーマンスの高い手法を採用している．

5. おわりに

業務系のバッチ処理へのHadoop適用はきわめて有用であることが示唆される一方で，その導入手法については，さまざまな工夫が必要であることも分かった．この問題に対して，現在のHadoopエコシステムは業務系バッチ処理に適していないため，Asakusaのような別のフレームワークを利用する必要がある．

また，全体のアーキテクチャをどう構成するかについても検討する余地がある．特に本稿で考察したシステムでは，RDBMSとHadoopクラスタを密結合するアーキテクチャを採用し，システム全体の可用性を上げることができた．これにより，アプリケーションレイヤから見たデータの取り回しの容易性や，分散クラスタの対障害性

の確保といったメリットが簡易に享受できる．その一方で，RDBMSとHadoopクラスタ間のデータ転送において，ボトルネックが生じやすいというデメリットが強く出る．

このデメリットは，ハードウェアによる解決やRDBMSの構成を変更させることで軽減させることは可能ではあるが，根本的な解決にはならない．データ転送ボトルネックを根本的に解消するには，そもそもRDBMSを使わずに業務系の処理をすべて分散クラスタ内部で行ってしまえばよい．しかし，現状の分散クラスタの各種製品の品質や可用性・運用容易性は業務系の処理が行えるRDBMSの水準までには至っていない．これが今後の検討課題として残るであろう．

参考文献

1）トム・ホワイト（著），玉川竜司，兼田聖士（翻訳）：Hadoop，オライリージャパン（2013）．
2）Asakusa Framework : Asakusa Framework 入門，http://asakusafw.s3.amazonaws.com/documents/latest/release/ja/html/introduction/index.html
3）太田一樹，下垣　徹，山下真一，猿田浩輔，藤井達朗，濱野賢一朗：Hadoop 徹底入門，翔泳社（2013）．
4）Lin, J. and Dyer, C.（著），玉川竜司（翻訳）：Hadoop MapReduce デザインパターン，オライリージャパン（2011）．
5）神林飛志：開発・運用フレームワーク Asaksua, ASCII. technologies 2011 年 1 月号，アスキー・メディアワークス（2010）．
6）神林飛志:Hadoop 向け基幹バッチ分散処理ソフト「Asakusa」の全貌，ITPro，http://itpro.nikkeibp.co.jp/article/COLUMN/20110322/358560/（2011）.
7）神林飛志，須賀秀和：Haddop で業務バッチ処理，Web+DB Press Vol.67, pp.112-119, 技術評論社（2012）．

神林 飛志（非会員）kambayashi@nautilus-technologies.com
1970 年 5 月 2 日生．1994 年 4 月中央監査法人入所．1998 年 5 月公認会計士登録．同年 6 月（株）カスミ入社．2000 年 5 月同社取締役．同年 10 月ウルシステムズ取締役就任．2011 年 10 月（株）ノーチラス・テクノロジーズ代表取締役副社長就任．2012 年 4 月（株）ノーチラス・テクノロジーズ代表取締役社長就任．

投稿受付：2013 年 12 月 24 日
編集担当：福島俊一（日本電気（株））