Asakusaソースコードリーディング（第二回）

ThunderGate

2011年6月23日

- 自己紹介
  - 埋金　進一
  - ウルシステムズ　株式会社　所属

  - 普段の仕事
    - Asakusaの中の人
    - ThunderGateのアーキテクチャ設計
  - つい最近まで
    - UMLaut/J-XML(XML-EDIパッケージ)の開発
  - もともとは
    - テレメータの開発

本日の内容

- なぜThunderGateを作ったのか
- TunderGateの中身の紹介
- TunderGate今後、どのように拡張していくのか

なぜThunderGateを作成したのか

データをどこに永続化するのか

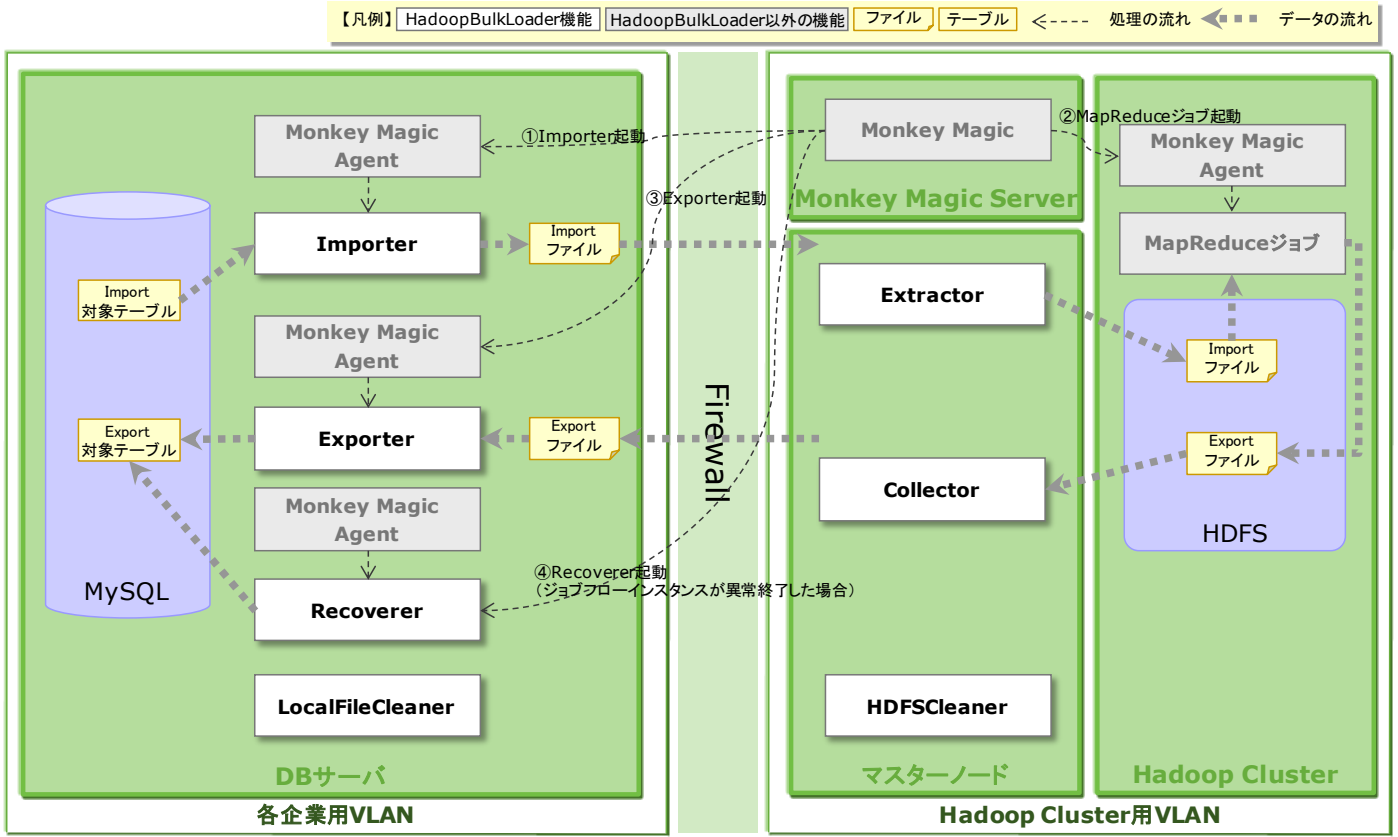
- 基幹業務をHadoopで実現するとき、Hadoopで処理する/処理したデータをどこに置くのか
  - HDFS
    - Hadoopを使うなら基本HDFS
    - 信頼性は？
      - namenodeがSPOF
      - バックアップ/リカバリ
        - ノウハウの不足
        - 壊れたとき復旧できる技術者がいるのか
  - RDBMS
    - 高い信頼性
      - ノウハウの蓄積
      - 技術者が多い
    - オンライン処理
      - 一部ACIDトランザクションが必要
    - 高機能ファイルシステム
  - KVS
    - (私には)未知数

ThunderGateに求められるもの

- データはRDBMSへ永続化する
  - Hadoopで処理するデータはRDBMSからHDFSにImport
  - Hadoopが処理したデータはHFSからRDBMSへExport
  - HDFSがおかしくなったらHDFSをフォーマットして対応する
- オンライン処理、他のAsaksuaバッチが同一データにアクセス
  - 排他制御の仕組みが必要
- その他、一般的でない(かもしれない)要件
  - RDBMSはプライベートなネットワーク、Hadoopはクラウドで動作
    - RDBMSと、HDFSの間にF/Wが存在
  - 処理対象のデータは、数Gbyte程度
    - 現状のThunderGateはスケールアウトしないが、数Gbyte程度のデータであれば問題とならない
    - 数100GByte以上のデータについては今後の課題
  - ターゲットとなるRDBMSはMySQL
    - 性能を優先し、MySQL依存になることを許容する
  - ワークフローエンジンとして、MonkeyMagicを使用

TunderGateの動作概要

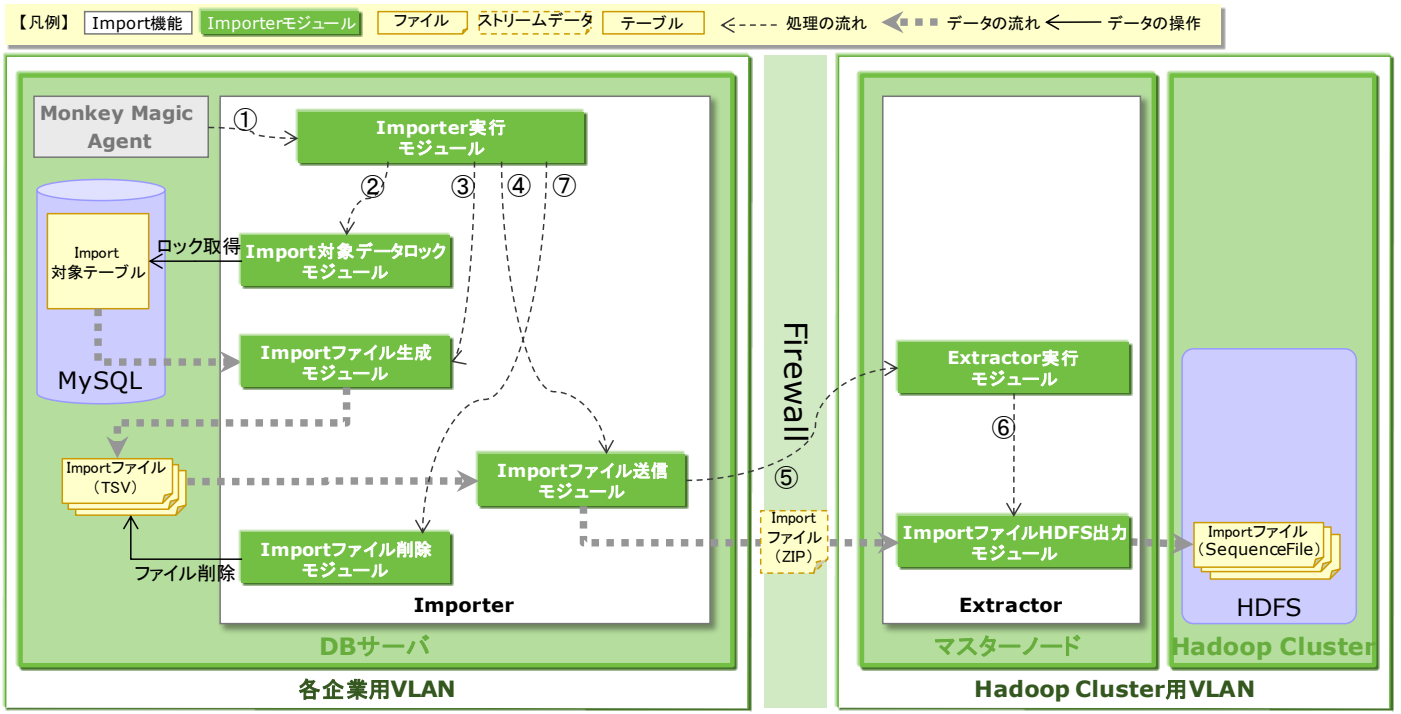
- **Import処理**
  - Import処理とは、データベース（MySQL）に格納されたデータをHDFSにコピーする処理を指す。
  - DBサーバ上のデータベースからデータをdumpし、マスターノードに転送して、HDFSにインポートする。
- **Export処理**
  - Export処理とは、HDFSに生成されたファイルをデーベース（MySQL）に格納する処理を指す。
  - マスターノード上でHDFSのファイルを取出し、DBサーバに転送して、データベースにloadする。
- **Recovery処理**
  - Recovery処理とは、異常終了したジョブフローインスタンスに対するリカバリを行い、データベースの不整合を解消する処理を指す。具体的には異常終了したジョブフローインスタンスに対してロールバック又はロールフォワードを行う。
- **Cleaning処理**
  - Cleaning処理とは、不要になったDBサーバ上のファイル及びHDFS上のファイルを削除する処理を指す。



ThunderGateの中身の紹介

Import処理概要

① MonkeyMagicから Importer起動
② ロックを取得
③ MySQLの select … info outfile filename を使用してTSVファイルを作成
④ sshで、Extractorを実行
⑤ TSVファイルを読み込み、zipフォーマットでExtractorへ流し込む
⑥ TSVファイルをシーケンスファイルに変換しHDFSに書き込む

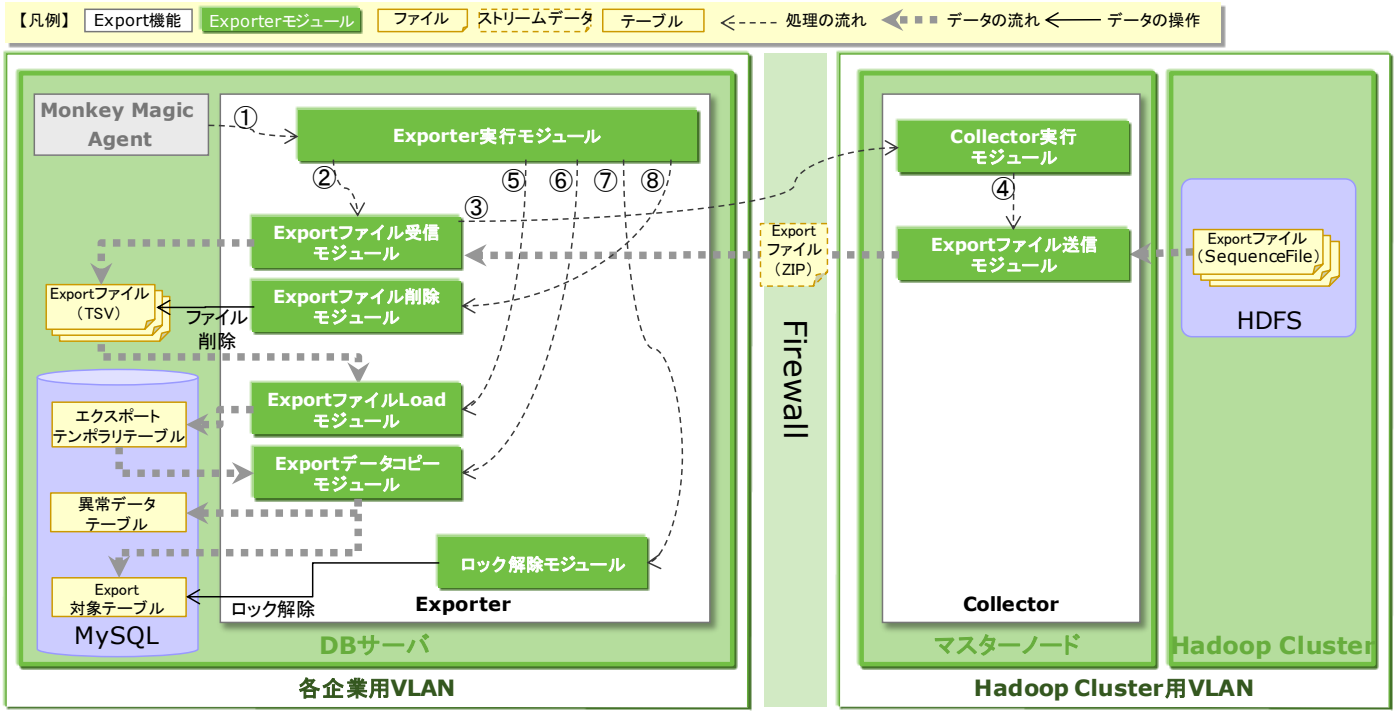


Import処理について思うこといろいろ

- sshによる認証は必要だが、暗号化は必ずしも必要ない
  - sshによる暗号化のオーバヘッドが大きく、Gigabit Etherの帯域を使い切れないのでなんとかしたい
- zip形式を採用したのは、データ圧縮による性能向上と、複数のTSVファイルを手っ取り早く、Extractorへ送るため
  - データ圧縮は、100MbpsのLANでは効果あり、1GbpsのLANでは圧縮のオーバヘッドが大きく意味がない
    - ThunderGateのデフォルトでは圧縮を行わない
    - 他の圧縮アルゴリズムを試してみたい
- TSV => シーケンスファイル変換をExtractor側に置いたのは、DBサーバ側のリソースが貴重と考えたから
  - 他がボトルネックなので、あまり関係なかった
- Extractorが分散処理をしないのは、DB側がボトルネックになるから
  - 今のところは

Exporterの処理概要

① MonkeyMagicから Expoterを起動
②~③ sshを使用してcollectorを起動
④ CollectorがHDFS上のSeqeunceファイルをTSVファイルに変換し、zipフォーマットでExporterに送り込み、DBサーバ上にTSVファイルを生成する
⑤ mysqlの load data in file … コマンドでワークテーブルに取り込む
⑥ ワークテーブルの内容でExport対象テーブルに書き込む
このとき、重複レコードを検出し重複レコードを以上データテーブルに書き込む
⑦ ロックを解除する



Export処理について補足

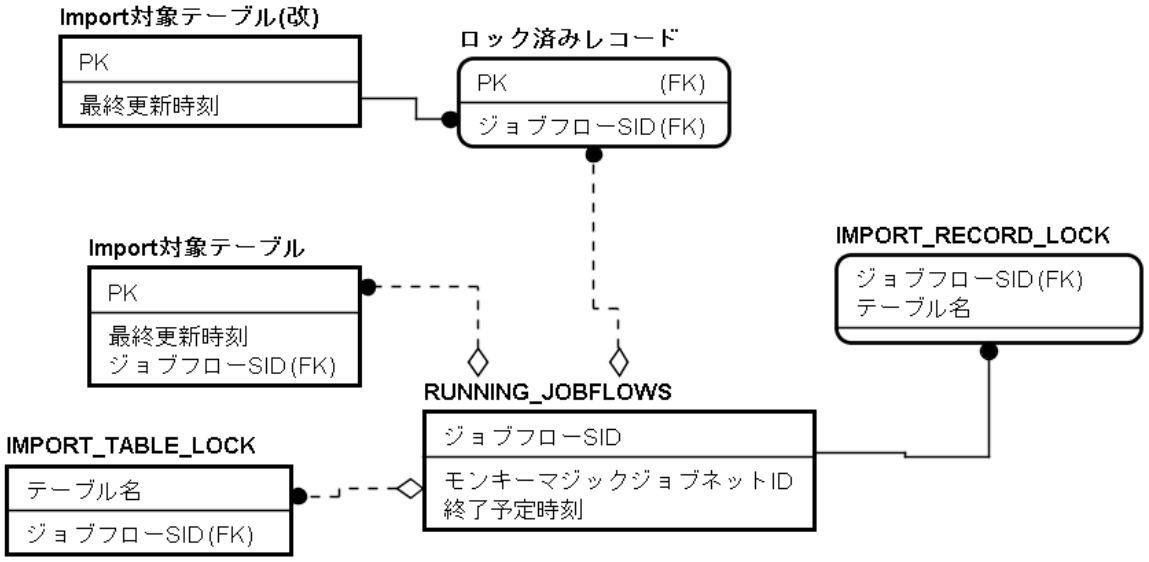
- Importer/Exporterは、MySQLとHDFS間のデータの受け渡しを行い、業務ロジックは動かさないのが基本
  - 重複チェック機能は例外的にExporterで実現している。パフォーマンスを得るため、例外的にExporterで実現。
- TSVファイルが大きすぎると、MySQLのリソースが不足するので、ある程度の大きさで分割する
  - 分割サイズはパラメータで設定可能、極端に大きい値、極端に小さな値を指定しない限り、性能に大きな影響はない。

TunderGateのトランザクションの考え方

- RDBMSのテーブルロック、レコードロックを使用しない
  - 長時間処理がブロックされる可能性があり、ロングトランザクションに不向き
  - RDBMSのリソースの枯渇
- 基本的にはバッチの処理中のデータにフラグを立て、他のバッチ、オンライントランザクションはフラグが立ったデータにアクセスしない
- テーブルロックと、レコードロックをサポート
- 楽観ロックが適用可能
- ロールバック/ロールフォワードが可能
  - Recoverer
    - ThunderGateおよび、Hadoopの処理が何らかの問題でAbortした場合にトランザクションをロールバック/ロールフォワードする
    - ロックの解放
    - ExporterによるRDBMS上のデータ更新が始まっている場合、ロールフォワードにすることにより中途半端な状況を解消する
    - HDFS上のデータは単なるキャッシュに過ぎないので、クリーニングするのみ

ロックの実現方法

- 実行中のAsakusaバッチを管理する(RUNNING_JOBFLOWS)
- レコードロック
  - Import対象のテーブルに、ロック用のフィールド(ジョブフローSID)を用意
  - バッチ実行中は、ジョブフローSIDに、バッチを識別するIDをセットして、当該レコードがAsakusaバッチで実行中であることを示す。
  - オンライントランザクションは、最終更新時刻を使用した楽観ロックにより、バッチ実行後の値をバッチ実行前の値で上書きすることを抑止する。
  - 実際には、速度向上のため別のフィールドを用意するのではなく別のテーブル(ロック済みレコードテーブル)を使用している。
  - IMPORT_RECORD_LOCKテーブルは、行ロックの対象となっているテーブルを高速に取得するために使用する
- テーブルロック
  - ロックされたテーブルを記録するテーブルを用意(IMPORT_TABLE_LOCK)



ロールバック/ロールフォワード

- Recovererを実行すると、処理状況に応じてロールバック/ロールフォワードを実行し、トランザクションの中途半端な状況を解消する
- Asakusaバッチのロールバック
  - Exporterの処理が始まるまでは、ロールバックするAsakusaバッチに関連するレコードを削除するだけでよい。
  - バッチがロールバックできずに終了してしまうケース
    - 実行中でないバッチに関連するレコードを削除する機能を用意
      - Recovererの一機能として実装
- Export処理が中途半端な状態にしてはいけない
  - RDBMSのトランザクションで実現するのが簡単だが、Export対象データが大きいときは無理
  - Exporterは、直接Export対象テーブルを更新するのではなく、一度ワークテーブルにデータを格納し、全てのEexport対象データをワークテーブルに格納した後、ワークテーブルの値を使用して、更新対象のテーブルの更新を開始する。
  - 一度、更新対象のテーブルの更新を始めたら、ロールバックはできなくなるので、ロールフォワードで対応する。

TunderGate今後、どのように拡張していくのか

今後の拡張について - ThunderGateの高速化

- 現状のThunderGateの処理性能が不十分
  - 実績値
    - 20MByte/Sec程度
  - ThunderGateの処理中でボトルネックとなりそうな箇所の処理速度
    - MySQLのLoad/Dump　　30MByte/Sec～70MByte/Sec
    - SSHによる転送　　50MByte/Sec
    - Gigabit Ether　　100MByte/Sec
- Import/Export処理が、バッチ処理全体のボトルネックになる
  - MapReduceジョブはスケールアウト可能なのに対して、現行のThunderGateはスケールアウトできない
  - 現行のアーキテクチャでは、DBサーバのI/O性能がボトルネックとなり、スケールアウトしない
- 以降のスライドで、ThunderGateの高速化の方法について述べる
  - Step-1
    - DBサーバのI/Oボトルネックを目指す高速化
  - Step-2
    - キャッシュによる高速化
  - Step-3
    - MySQLのクラスタによる高速化

DBサーバのI/Oボトルネックを目指す高速化(1/4)

- 処理の並列化
  - ThunderGateでは、本質的に並列処理可能な処理が逐次処理されている箇所が幾つかあるため、処理の並列化により高速化が可能である。
  - DBのダンプファイルの扱い
    - 次の二つの処理が逐次実行される
      - Select ～into outfile ～によるダンプファイルの作成
      - ダンプファイルの取り込み
    - 高速化その0
      - 大きなダンプファイルを1つ作成するのではなく、複数のダンプファイルを作成し、作成が終わったダンプファイルを順次処理する
      - ThunderGate設計時に考えていた方式、あと二つの処理の方が良い
    - 高速化その1
      - ダンプファイルの書き込み完了を待たずに、ダンプファイルの取り込みを実行する。
      - ダンプファイルをポーリングし、増加分を転送する。
      - ダンプ処理が終了した時点のダンプファイルを転送し終わったら転送処理を終了する。
    - 高速化その2
      - そもそも、ダンプファイルを介した処理をしない。Sqoopがダンプファイルを使用しないで、Import処理をしているようなので、Sqoopと同じ仕組みを取り入れる。

DBサーバのI/Oボトルネックを目指す高速化(2/4)

- sshによるファイル転送の高速化
  - もともと、Hadoop Clusterがクラウド上にあるという要件からsshを使用するようになった
  - sshの認証と暗号化のうち、必要なのは認証であり、暗号化は不要
    - Hadoopの通信は暗号化されていないので、暗号化することによるセキュリティへの寄与は小さい
    - 暗号化処理はCPUリソース消費が無視できない
      - 最新のCPU、1Coreで50MByte/Sec程度の処理能力
  - 高速化その0
    - ImporterとExtractor/ExporterとCollectorを同一サーバ(いっそのこと同一プロセスで)動かす
  - 高速化その1
    - sshによるファイル転送を並列実行する
      - DBサーバと、転送先のサーバが両方ともマルチコアでなければならない
      - DBサーバのCPUを浪費して良いのかという問題が残る
  - 高速化その2
    - sshの使用を止める
      - http
      - Raw socketを使用

DBサーバのI/Oボトルネックを目指す高速化(3/4)

- HDFSの書き込みの分散
  - 現状の仕様では、1ノードでHDFSへ書き込んでいる
  - 複数のノードでHDFSに書き込むことにより、HDFSへの書き込みがボトルネックとなることを防ぐ
- データ圧縮
  - ファイルに書き出すデータの圧縮、ネットワークを通すデータの圧縮によるI/Oを減らす
  - CPUリソースの消費量とのトレードオフがある
  - 使用する圧縮アルゴリズムに依存、LZOが良いらしい
  - ThunderGateのパフォーマンステストでは、データ圧縮は逆効果
    - 使用した圧縮アルゴリズがbzip, LZOだと結果が変わったかも
    - データを圧縮するスレッドと、I/Oをするスレッドを別スレッドにすれば、結果が違ったかもしれない。
- MySQLの処理の高速化
  - いろいろ考えられるが、この資料の対象外

DBサーバのI/Oボトルネックを目指す高速化(4/4)

- Exportファイルの取り込み
  - 次の三つの処理が逐次実行される
    - ①Exportファイルの作成
    - ②Exportファイルのテンポラリテーブルへの Load
      - MySQLのLoad data in file文を使用
    - ③テンポラリテーブルの内容でトランザクションテーブルを更新
      - INSERT ～ SELECT ～/ UPDATE ～ SELECT ～
  - ①、②については、DBのダンプファイルの扱いと同様に並列化可能
  - トランザクショナルに処理する必要が無い場合は③の処理も並列化可能
    - 1つのExportファイルのテンポラリテーブルへのLoadが終わった時点で、他のExportファイルのLoadを待たずにトランザクションテーブルの更新を始まる
    - 処理のロールバック、ロールフォワードができなくなるので、トランザクショナルな処理が不要な場合のみに適用可能
- SQL文の同時実行
  - Select ～ into outfile ～, Load data in file , Insert ～ select ～, Update ～ SELECT ～ を複数同時実行する
  - 以前の検証では、逆効果だった(ちなみにOracleは4並列ぐらいが一番性能が良かった)が、MySQLのチューニング次第では効果が得られる可能性がある
  - 基本的に、I/Oボトルネックではなく、CPUボトルネックの場合に効果が出る。

キャッシュによる最適化

- バッチの実行前に予めデータをHadoop Clusterに置いておくことにより高速化する
  - Hadoopバッチ実行時には、差分データのみをHadoop Clusterにコピーする
  - 次のようなケースで効果的
    - 1日に更新される量はごく僅かだが、データ全体は巨大なマスタデータ
    - 過去1年分の実績データを処理する日時バッチ
      - 新規に発生した実績データは全体の1/365
- ポイント
  - Hadoop Cluster上にどのようにデータを置くのか
    - シーケンスファイル
      - MapReduceで差分データをマージする
    - HBase
      - 単にThunderGateのキャッシュと考えた場合、シーケンスファイルに対する優位点はないが、JOINの高速化など、ThunderGate以外の処理でも恩恵が得られる。
  - 差分情報をどのように作成するのか
    - 新規作成データ、更新データについては、最終更新日と前回のキャッシュの更新日時を比較すれば良い
    - 削除データの扱い
      - 物理削除への対応は難しいので、基本的に論理削除を使用する
      - 論理削除データの物理削除に先立って、論理削除データをキャッシュから削除して、キャッシュとDBデータの整合性を確保する必要がある。
        - この処理は上手く作らないと非常に重たくなる
  - トランザクション
    - ThundarGateのトランザクション管理の仕組みを維持したまま、キャッシュの仕組みを導入するのは難しそう
      - 詳細は要検討
      - 当面は、キャッシュ可能なデータはTundarGateのトランザクションのスコープに含めないことで良いと思う

MySQLのクラスタによる高速化(1/2)

- DBサーバがボトルネックになるなら、MySQLのインスタンスを増やしてしまおうという発想
  - 更新可能なMySQLインスタンス(マスタ)を1つだけ用意し、複数の参照専用のレプリカを用意する
  - Import時には、各レプリカからもImportを行うことにより処理を分散できる
  - Select分の実行結果を分散して取り出すのは意外と難しいため、要検討
- ThunderGateのトランザクション管理と容易に共存可能
  - ただし、今度はThunderGateのトランザクション管理のための、DB更新処理がボトルネックになってしまう

MySQLのクラスタによる高速化(2/2)

- 要検討項目
  - MySQLのレプリケーションがボトルネックにならないか
  - レプリカが完全な複製を持っていることをどう保証するのか
  - レプリカが壊れたときの復旧/レプリカの復旧に時間がかからないか