GridBatch : Cloud Computing for Large-Scale Data-Intensive Batch Applications

グリッドコンピューティング　論文紹介
計算工学専攻　杉山研究室
09M38217　　重村 達哉

出典

- GridBatch: Cloud Computing for Large-Scale Data-Intensive Batch Application
  - Author :
    Huan Liu, Dan Orban
  - Source :
    2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)
  - URL : http://www.computer.org/portal/web/csdl/doi/10.1109/CCGRID.2008.30

背景

- 企業は大量の情報を現実的な時間で収集し、分析する必要がある
- 並列化によって処理速度を改善
  - 2002年頃から単一プロセッサーの処理速度の発達はあまり目覚ましいものではない
  - 今日のアプリケーションでは時間がかかり過ぎる
- 並列プログラミングは時間がかかり、ミスを起こしやすい

既存の分析アプリケーションの問題点

- 大規模データセットを扱えない
  - SQL言語
    - 扱いやすいが、非効率的な方法で計算を表現することになる
      - 効率的な手法では、データを一度スキャンするだけ
    - 自動的にクエリ処理を最適化するように発達しているがまだ性能は良くない
  - 企業のインフラは十分なバンド幅を持っているが、クラウドのインフラではバンド幅が制限されている(Amazon EC2 250Mbps)

関連研究

- MapReduce
- Microsoft Dryad
  - 柔軟性が高い
  - 全ての並列計算がグラフとして表現される
- Map-Reduce-Merge
  - MapReduceの枠組みを拡張
  - 2つのデータセットをマージできる

これまでのシステムと異なる点

- 複数のオペレータを提供
  - プログラマーが複雑な分析アプリケーションを実装する為に自由裁量で構成できる
- GFSをプログラマーがデータを局所的に最適化する為にどこにストアするかをコントロールできるように拡張
  - ネットワークのバンド幅が専用のインフラのバンド幅に比べて非常に制限されている場合に重要

サンプルアプリケーション

- 重複の特定
  - バーコードはそれぞれ違う
  - 同じバーコードのものを検出したい
- 監査目的で全てのバーコード付きの商品の把握
- 同じコンテナ内の全ての商品をスキャン
  - 別のコンテナの商品が混ざってないか確認したい
- 業務ルール違反の摘出
  - 業務ルールのリストを与え、商品が違反していないことを確認したい

The GridBatch system

- GridBatchのデータタイプ
  - テーブル
  - インデックステーブル
- レコード(列)、フィールド(行)
- レコードはそれぞれ独立
  - 並列に処理できる
- The Grid Batch System
  - The distributed file system
  - The job scheduler

The Distributed File System (DFS)

- DFS：GFSの拡張版
  - ファイル管理
- 大きなファイルは多くの小さなチャンクに分割
- 各々のチャンクは別々のノードにストアされる
- １つはネームノード、残りはデータノード
- データタイプ
  - fixed-chunk-size file
    - 64MB のチャンクに分割
    - 新しいレコードが書き込まれるときは最後のチャンクの一番後ろに付け加えられる
    - 最後のチャンクが64MBに達したら、新たなチャンクがネームノードによって割り当てられる
  - fixed-num-of-chunk file
    - C個のチャンクに分割
    - DFSクライアントが新しいファイルを作るよう指示した時は、全てのチャンクに同時に割り当てられる
- 対故障性：同じファイルを複数のノードに保持させる

The job scheduling system

- １つのマスターノードと多くのスレーブノードから成る。

- スレーブノード
  - マスターノードに割り当てられたタスクを実行

- マスターノード
  - ジョブを多くの小さなタスクに分割
  - スレーブノードにタスクを分配
  - タスクがきちんと実行されるか監視

GridBatch operators

- GoogleのMapReduce systemの拡張版
- MapReduce system
  - Map operator
    - ファイル内の全てのレコードにそれぞれ独立に適用される
    - 並列化が簡単
    - Reduce operatorで用いるkey-value pairのセットを作る
  - Reduce operator
    - 特定のキーに関連した値を取り、ユーザー定義のReduce functionを適用する
- GridBatch
  - Map, Distribute, Recurse, Join, Cartesian, Neighbor

Map operator

```
Map(Table X, Func mapFunc)
```

- ユーザー定義の関数をテーブルの全てのレコードに適用
  - ユーザーはレコードに対してどんな処理もできる
  - ユーザー定義の関数

```
mapFunc(Record x):
   // Apply necessary processing on Record
   // x to generate Record y
   ......
   EmitResult(Table Y, record y)
```

Distribute operator

```
Table Y = Distribute(Table X, Field i, Func
newPartitionFunc)
```

- テーブルやインデックステーブルを違うインデックスを持ったインデックステーブルに変換
- インデックステーブルはfixed-num-of –chunk DFS file としてストアされる
- newPartitionFunc

```
int[] newPartitionFunc(Index x)
```

  - レコードを書き込むチャンクを示す値を返す

Distribute operator

①マスターノードは分割したスレーブタスクをスレーブノードに割り当てる
- 効率的な局所処理の為にタスクiはチャンクiを保持するノードに割り当てられる
  - タスクi：i番目のチャンクのタスク
- スレーブタスクは並列で処理される

②それぞれのスレーブタスクが局所出力ファイルを作る

③タスクiはチャンクiの各レコードに対し実行され、newPartitionFuncによって書き込むチャンクの番号jを決定する

④チャンクjにレコードが書き込まれる

⑤スレーブタスクが完了したら、マスターノードに通知し、局所出力ファイルの位置を伝える

⑥割り当てた全てのスレーブタスクが完了したら、新たにタスクを割り当てて、②に戻る

⑦全てのタスクが完了したら、Distribute operatorを終了する

Join operator

```
Join(Table X, Table Y, Func joinFunc)
```

- インデックスフィールドが一致したら一致したレコードをマージする。
- レコードxとレコードyがマッチしたときのみjoinFuncは呼び出される

```
joinFunc(Record x, Record y)
  // Apply necessary processing on Record
  // x and y to generate Record z
  ......
  EmitResult(Table Z, record z)
```

- テーブルX,Yは同じ分割関数(Distribute operator)によって分割されている必要がある

Join operator

①マスターノードがスレーブノードにタスクを割り当てる

②タスクiはテーブルXとテーブルYのチャンクをそれぞれインデックスの昇順に並び替える

③タスクiはレコードx,yのインデックスの値i(x),i(y)をポインタする

④i(x)=i(y)の時、joinFuncが呼び出される
i(x)<i(y)の時、テーブルXのポインタを進める
i(x)>i(y)の時、テーブルYのポインタを進める

⑤全てのレコードをスキャンするまで実行する

⑥全てのタスクが完了し、マスターノードに通知されたら終了

Cartesian operator

```
Cartesian(Table X, Table Y, Func
cartesianFunc)
```

- join operatorと違って、テーブルXとテーブルYの全てのレコードが一致したときにだけユーザー定義の関数を適用する

```
cartesianFunc(Record x, Record y)
  // Apply necessary processing on Record
  // x and y to generate Record z
  ......
  EmitResult(Table Z, record z)
```

- インデックスの一致は必要ない

Recurse operator

```
Recurse(Table X, Func recurseFunc)
```

- テーブルを１つのレコードにマージする
- recurseFunc : どのようにマージするのかを定義

```
Record recurseFunc(Record x1, Record x2)
  // Apply processing on x1 and x2
  return x = x1 + x2
```

- GridBatchでは再帰的にテーブルXの全てのレコードに対して関数を適用する
- RecurseはMapReduceのReduce処理に比べて、多くのノードで並列に行えるという点でより効率的だと言える

Recurse operator

①マスターノードがそれぞれのテーブルXに対するチャンクを保持しているスレーブノードにタスクを割り当てる

②タスクiはrecurseFuncを用いてチャンクiの全てのレコードをマージする

x1とx2をマージし、次にその結果とx3をマージする。

この繰り返しを全てのレコードに対して行う

③半分のタスクでの結果を残りの半分のタスクに渡し、マージを行う

④③を繰り返し、最終的な結果が出るまで行う

マスターノードはマージする順序等を調節する

Neighbor operator

```
Neighbor(int k, Table X, Func neighborFunc)
```

- k個の近いレコードを集める
- neighborFunc : k個の引数を取る

```
neighborFunc(Record x1, Record x2)
  // report discontinuity
  if ( x1.containerID ≠ x2.containerID )
    EmitResult(Table Z, record x1)
```

- 時間の削減になる

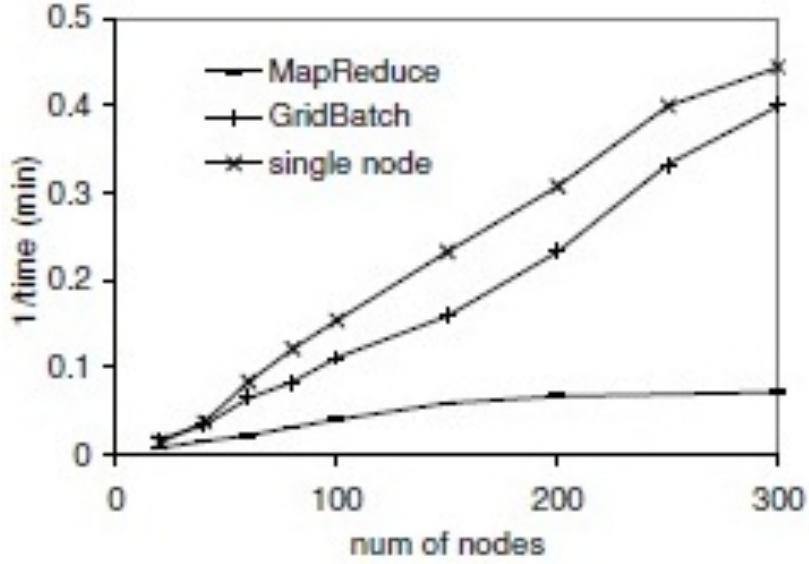
実験環境

- 比較
  - GridBatch
  - MapReduce
- Amazon EC2 Cloud で実験を行う
  - 1.7Ghz x86 processor
  - 1.75GB of RAM
  - 160GB of local disk
  - 250Mb/s of network bandwidth

Duplicate detection : 設定

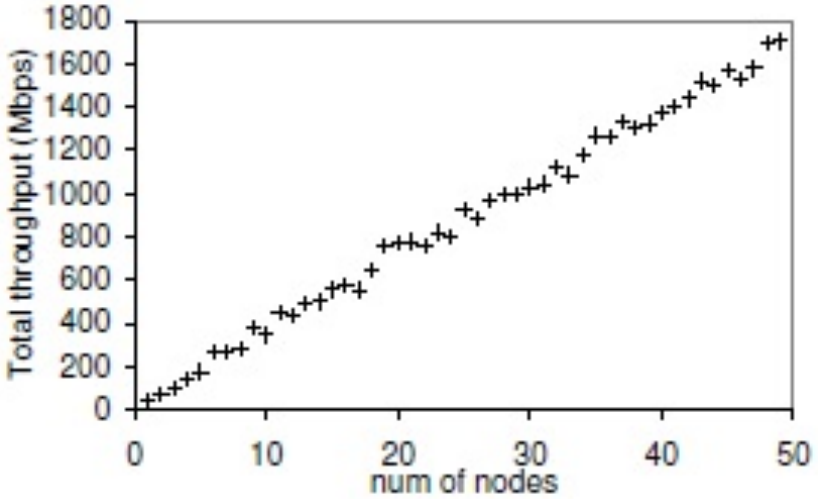
- データ
  - 250GB
  - 各レコードは約170bytes
  - 性能目標：5分以内に結果を報告する
- GridBatch
  - データセットをテーブルとして読み込む
    - インデックスはバーコード
  - Join operator
    - テーブルをそれ自身でjoinする
  - Neighbor operator
    - k=2
    - 2つのレコードが同一のバーコードだったらそのレコードを出力する
- MapReduce
  - fixed-chunk-size fileを使用
  - Map function : バーコードをキーとし、値として 1 を抜き出す
  - Reduce function : 発生回数をカウント、
    2回以上出てきたバーコードを報告

Duplicate detection : 結果



- MapReduce : 150ノード以降早くならなくなる
  - ネットワークのバンド幅の限界に達している
  - 1000ノードを用いても性能目標に到達しない
- GridBatch :
  - 全ての処理を行うのに150ノードで4.3分
  - 300ノードを用いれば、更に厳しい性能目標を達成できる

Data write throughput : 結果



- GridBatch : 100データノード
- 50DFSクライアントで1.7Gbpsの処理が出来ている
  - 保証されているバンド幅は250Mbps
- DFSクライアントは異なるデータノードに書き込んでいる

結論

- データ量が大きくなってきている
  - 現実的な時間の中で単一プロセッサーで処理するには難しい
  - 並列処理が必要

- GridBatch : MapReduceの拡張
  - クラウドインフラで性能が良いだけでなく、並列プログラミングを簡単にかける
  - 従来のMapReduceよりも高い性能を示した
  - Recurse operator, Neighbor operator等を使うことで保証されているバンド幅以上の処理が行える

意見

- ユーザーが定義する必要のある部分が多く、様々なアプリケーションに応用できるといえるが、反対に自由性が高く、扱いにくそう

- Neighbor operatorの近傍を定めるのに、一般的には距離を定義できないインデックスの方が多いのではないかと思う