【VLDB2011勉強会】

Session 18: MapReduce and Hadoop

担当：塩川浩昭（NTT）

紹介する4本の論文

1. Column-Oriented Storage Techniques for MapReduce
   Avrilia Floratou, Jignesh M. Patel(University of Wisconsin Madison), Eugene J. Shekita, Sandeep Tata (IBM)
2. Automatic Optimization for MapReduce Program
   Eaman Jahani, Michael J. Cafarella (University of Michigan), Christopher Re (University of Wisconsin Madison)
3. CoHadoop: Flexible Data Placement and Its Exploitation in Hadoop
   Mohamed Y. Eltabakh, Yuanyuan Tian, Fatma Ozcan (IBM), Rainer Gemulla (Max Planck Institut fur Informatik), Aljoscha Krettek, John McPherson (IBM)
4. Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs
   Herodotos Herodotou, Shivnath Babu (Duke Uiversity)

1. Column-Oriented Storage Techniques for MapReduce

- 背景
  - MapReduceに対してparallel DBMSのテクニックを導入したい
  - Hadoopの実装に手を入れるのは大変なのでデータフォーマットを変えるだけで対応したい
- 本論文の提案
  - 複数のデータ型から構成されるデータスキーマのcolocationを維持しつつ，列ごとに1つのファイルとして保持するデータフォーマットColumnInputFormat(CIF)を実装，高速化
  - CIFとskip listを用いた不要なColumnのdeserialize削減による処理の高速化
  - LZOとDictionary Compressed Skip Listを用いたデータ圧縮による処理の高速化

1. Column-Oriented Storage Techniques for MapReduce

- 対象とする処理
  - 複合的なデータ型を持つスキーマの処理
    - 例)URLに”ibm.com/jp”を持つ際，metadataを出力

Mapper

```
class MyMapper {
  void map (NullWritable key, Record rec) {
    String url = (String) rec.get("url");
    if (url.contains("ibm.com/jp"))
      output.collect(null,
        rec.get("metadata").get("content-type"));
  }
}
```

論文Figure1より引用

処理

データスキーマ

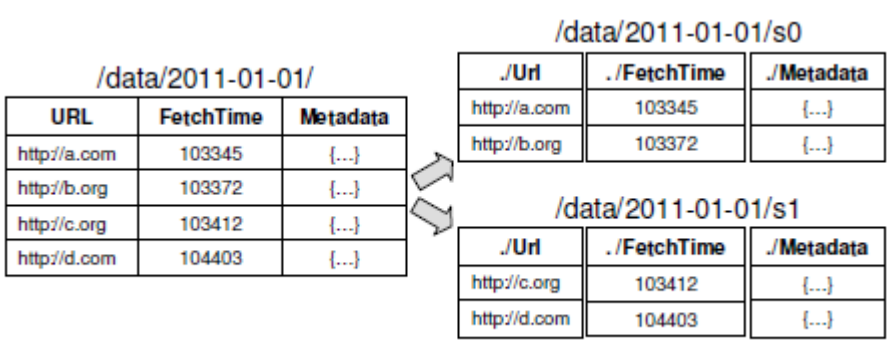
```
URLInfo {
  Utf8 url,
  Utf8 srcUrl,
  time fetchTime,
  Utf8[] inlink,
  Map<String, String> metadata,
  Map<String,String> annotations,
  byte[] content
}
```

論文Figure2より引用

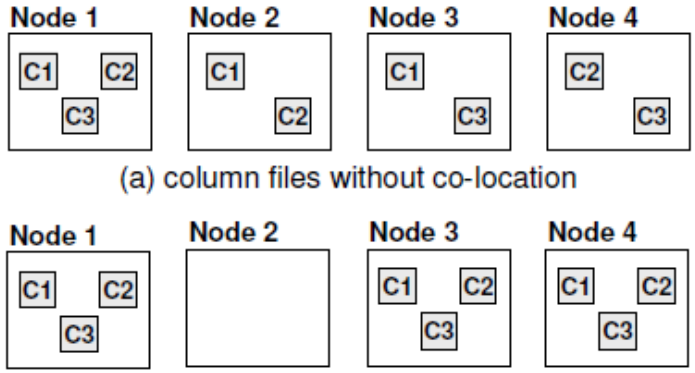
  - 処理の高速化に必要なテクニック
    - データスキーマに含まれるデータを同じデータノードで保持(colocation)
    - Binary formatでのデータの管理，serialize/deserialize処理の削減
    - データの圧縮による読み込み/書き出しデータ量の削減

1. Column-Oriented Storage Techniques for MapReduce

- ColumnInputFormat(CIF)の実装
  - Column毎に別ファイルとして管理することで，不要なColumnデータのdeserializeを回避可能にする
  - さらに，colocationのために，同じRowに属するColumnデータは同じデータノードに配置する



論文Figure4より引用

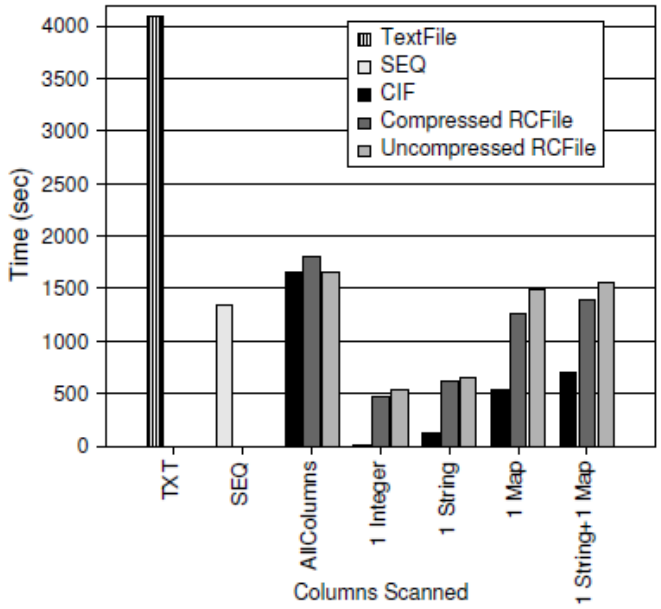


論文Figure3より引用

（その他）

- 処理に不要なColumnのスキップを高速化するためにColumn毎にSkip Listを導入
- ColumnデータはDictionary Compressed Skip Listにより圧縮

- データスキャン速度，処理速度の評価
  - CIFは必要なcolumnがcolocateされており，必要なデータのみをdeserializeするため，データの読み込みがRCFile(Hiveのデータフォーマット)よりも最大で38倍速い
  - MapReduceの処理時間はSEQに対して10～12倍速い



論文Figure7より引用

Layout	Data Read (GB)	Map Time (sec)	Map Time Ratio	Total Time (sec)	Total Time Ratio
SEQ-uncomp	6400	1416	-	1482	-
SEQ-record	3008	820	-	889	-
SEQ-block	2848	806	-	886	-
SEQ-custom	3040	754	1.0x	806	1.0x
RCFile	1113	702	1.1x	761	1.1x
RCFile-comp	102	202	3.7x	291	2.8x
CIF-ZLIB	36	12.8	59.1x	77	10.4x
CIF	96	12.4	60.8x	78	10.3x
CIF-LZO	54	12.4	61.0x	79	10.2x
CIF-SL	75	9.2	81.9x	70	11.5x
CIF-DCSL	61	7.0	107.8x	63	12.8x

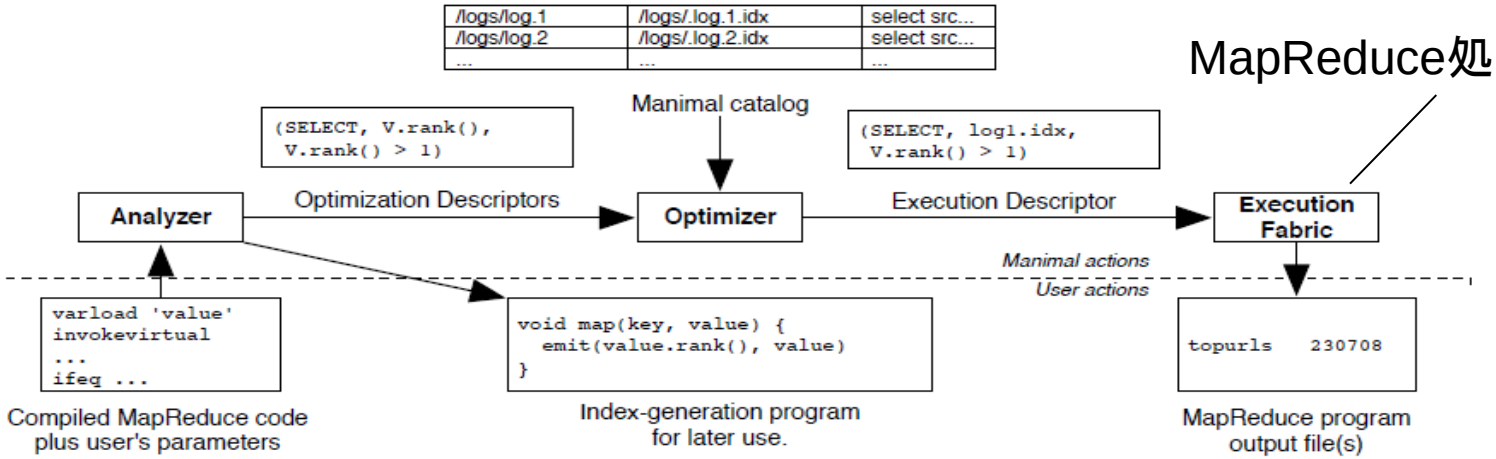
論文Table1より引用

2. Automatic Optimization for MapReduce Programs

- 概要
  - MapReduce処理はユーザがコーディングしたプログラムを実行する計算モデルであるため，DBMSで培った問合せ最適化技術が適用できず効率が悪い
  - MANIMALというプログラム自動解析システムにより，MapReduceの問合せ最適化を実現し，1,121%の処理速度向上を実現
- 本論文の貢献
  - Selection, Projection, Join処理を対象にプログラムの修正を必要としない最適化の実現
  - プログラムからSelection, Projection, Join処理を検出・抽出するアルゴリズムの提案
  - 既存ベンチマークPavloに対して，MANIMALを導入し，11倍の性能向上を実現

2. Automatic Optimization for MapReduce Programs

- MANIMAL：システム概要



論文Figure1より引用

Analyzer

- コンパイラで利用されるプログラム解析ツールを使い，＜処理の種類(selection, projection, join)，処理対象データ，処理条件＞を抽出
- 抽出したデータに応じて，処理対象データのindexを作成しておく

Optimizer

- Control flowと呼ばれる処理・状態のフローと，各処理の処理対象データから，DAGを生成
- 処理結果が正しく出力される範囲内で，DAGの統合を行い，問合せ最適化を実施

2. Automatic Optimization for MapReduce Programs

- 評価実験
  - ベンチマークPavloにおける最適化箇所の検出精度

Test	Description	Select	Project	Delta-Compression
Benchmark-1	Selection	Detected	*Undetected*	*Undetected*
Benchmark-2	Aggregation	Not Present	Detected	Detected
Benchmark-3	Join	Detected	Not Present	Detected
Benchmark-4	UDF Aggregation	*Undetected*	Not Present	Not Present

検出失敗

論文Table1より引用

【原因】・通常は使用しないクラスを用いた実装がされていた.
・今回，未対応としたJava Hashtableが含まれていた

→（イレギュラーな場合を除き）最適化箇所の抽出に成功している

  - 処理速度比較

Test	Description	Space Overhead	Hadoop	MANIMAL	Speedup
Benchmark-1	Selection	0.1%	429.78 secs	38.35 secs	**11.21**
Benchmark-2	Aggregation	20%	5,496.29 secs	1,855.65 secs	**2.96**
Benchmark-3	Join	11.7%	6,077.97 secs	903.752 secs	**6.73**
Benchmark-4	UDF Aggregation	0%	N/A	N/A	**0**

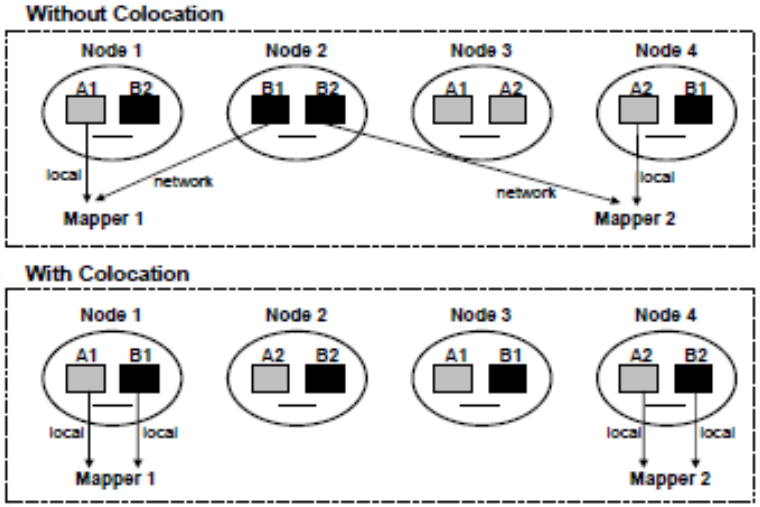
論文Table2より引用

約3～11倍の高速化を実現

3. CoHadoop: Flexible Data Placement and Its Exploitation in Hadoop

- 背景
  - HDFSはファイルをパーティションに分割し，更にパーティションを幾つかのブロックに分割するため，処理の際に複数のデータノードへのアクセスが発生し処理速度が低下する
  - CoHadoopでは（特にlog解析を対象に）処理に必要となるブロックを同じデータノードで持つ(colocating)ことにより処理速度の向上を実現

- 本論文の貢献
  - 関連するブロックをcolocatingする軽量な手法をHDFSに実装
  - Colocatingを基にlog解析処理を実装し，処理速度を向上
  - CoHadoopの障害回復，データの分散について検証



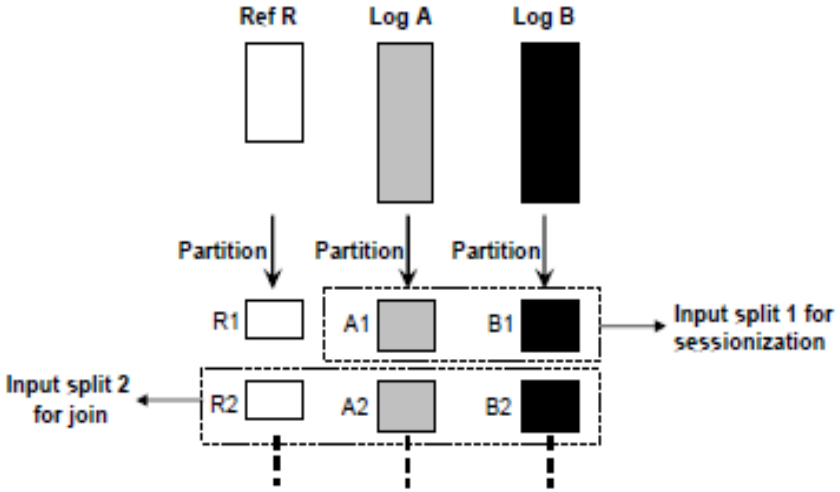
論文Figure3より引用

3. CoHadoop: Flexible Data Placement and Its Exploitation in Hadoop

- 基本アイデア

Partitioning

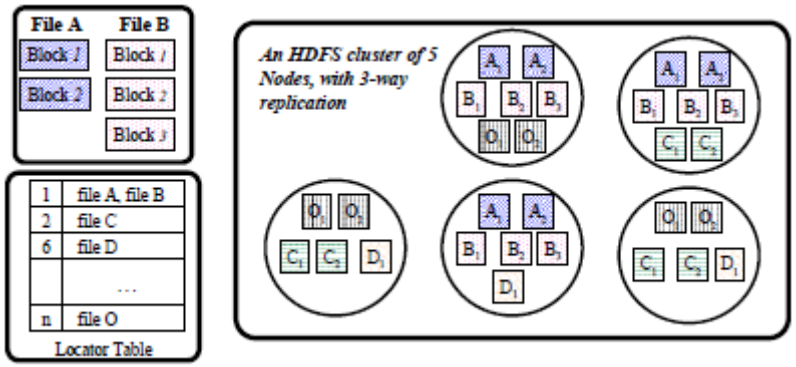
- HDFS上のファイルをkey順にソートし，一定のkeyレンジ毎にファイル分割する



論文Figure2より引用

Colocating

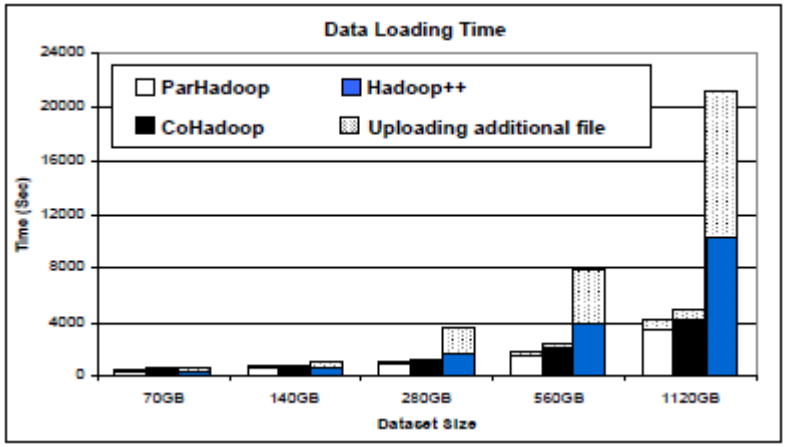
- 1つの処理に必要となるファイル集合にそれぞれlocatorを割り当てる
- Locatorが同じファイルはベストエフォートで同じデータノードに配置



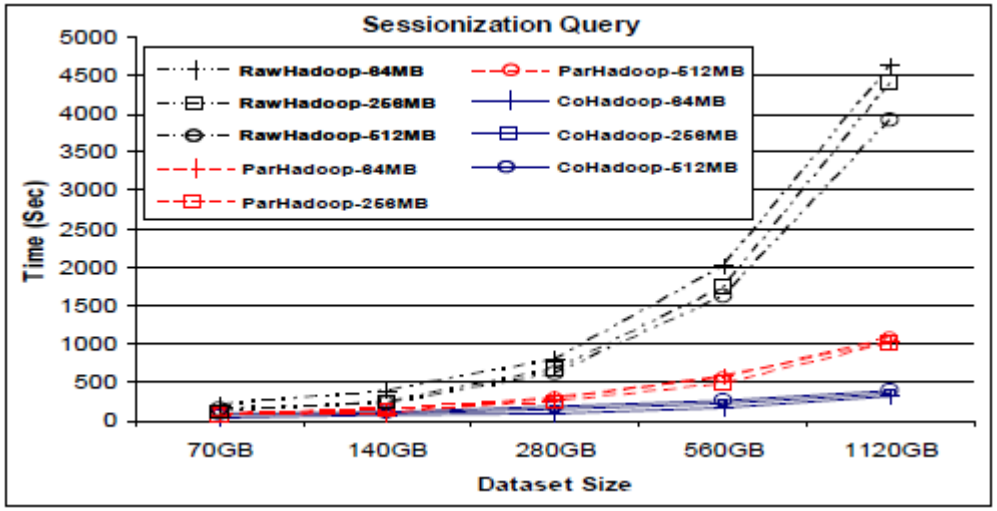
論文Figure1より引用

3. CoHadoop: Flexible Data Placement and Its Exploitation in Hadoop

- データロード時間と処理時間の評価
  - リモートからのデータロードがないためHadoop++よりも高速に読み込み
  - ParHadoopがローカルにかデータを配置しない一方で，CoHadoopはリモートにも配置するため，CoHadoopのロード時間がわずかに遅い
  - CoHadoopはデータが大きいほどデータの処理時間が相対的に速くなる（RawHadoopよりも約93%減）



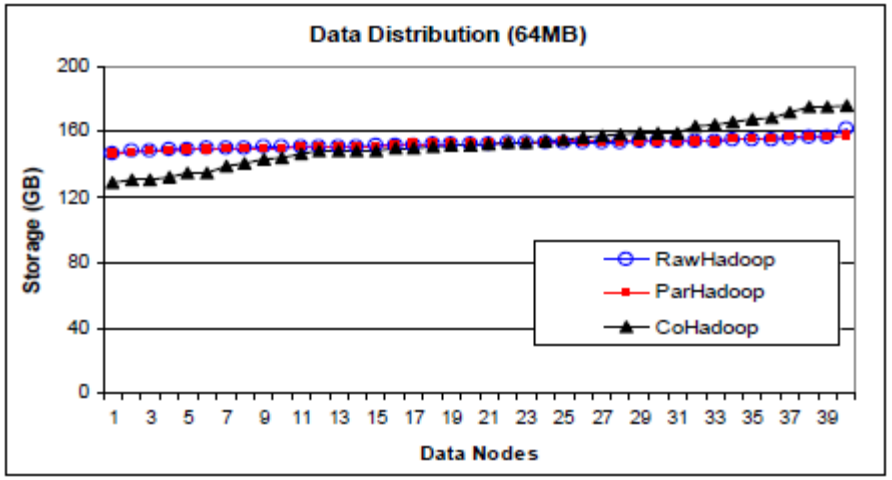
論文Figure4より引用



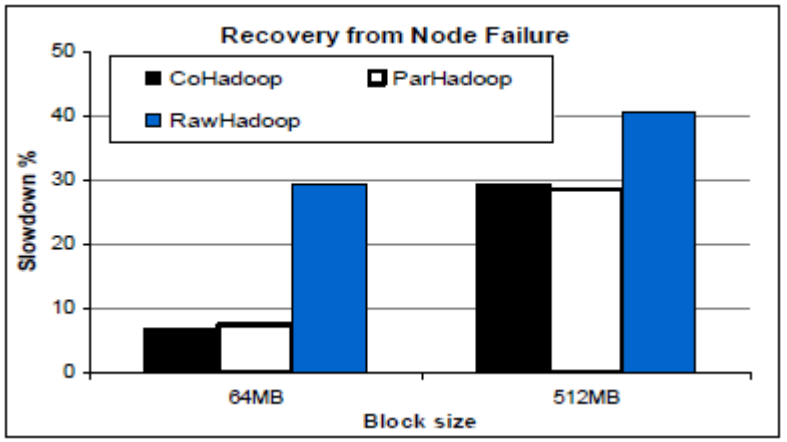
論文Figure5より引用

3. CoHadoop: Flexible Data Placement and Its Exploitation in Hadoop

- データ配置のばらつきとリカバリ時間の検証
  - CoHadoopは可能な限りcolocateするため，データノード毎に配置されるデータ量にばらつきが生じる
  - しかしながら，障害発生時のリカバリ時間は最小となる



論文Figure7より引用



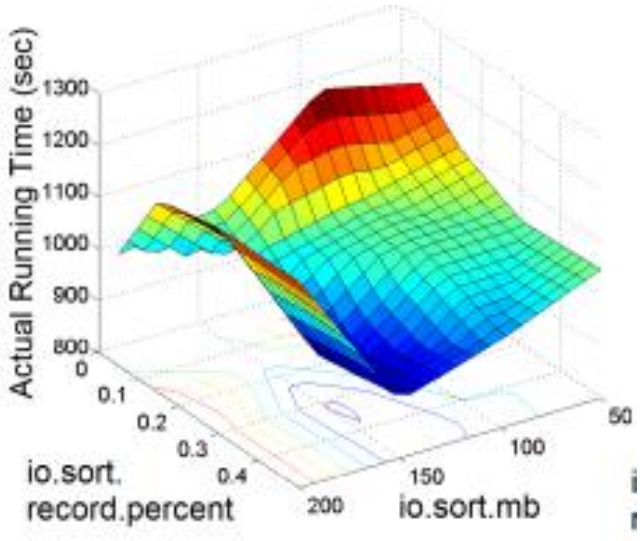
論文Figure8より引用

4. Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs

- 背景
  - MapReduce実行に必要なパラメータの数が膨大(190個以上)
  - MapReduceの性能はパラメータにより複雑に変化
  - Cost-based optimizationをMapReduceでも自動チューニングを実現できるようにしたい
- 本論文の提案
  - Profiler, What-if Engine, Cost-based Optimizer(CBO)の導入
    - **Profiler:** MapReduceのプログラムから動的に統計情報を取得
    - **What-if Engine:** パラメータ変更時の性能を推定
    - **CBO:** 最適なパラメータの探索

入力

入力

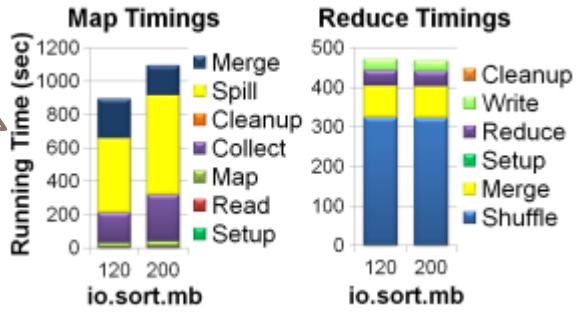


論文Figure2(a)より引用

4. Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs

- **Profiler：MapReduce処理に関する情報の抽出**
  1. 解析ツールBtraceを用いてMapReduceプログラムのbyte codeに対してECAルールのアクションを関連付ける
  2. Map, Reduce処理をより詳細なフェーズに分解し，以下の4カテゴリについてキャプチャする
     - **Dataflow fields:** 各フェーズで処理されたバイト数，レコード数など
     - **Cost fields:** 各フェーズの処理時間，CPU使用量，I/Oコストなど
     - **Dataflow Statics fields:** Dataflow fieldsに関する統計（平均，合計など）
     - **Cost Statics fields:** Cost fieldsに関する統計（平均，合計など）

MapのSpill処理の時間が増加してる



論文Figure3より引用

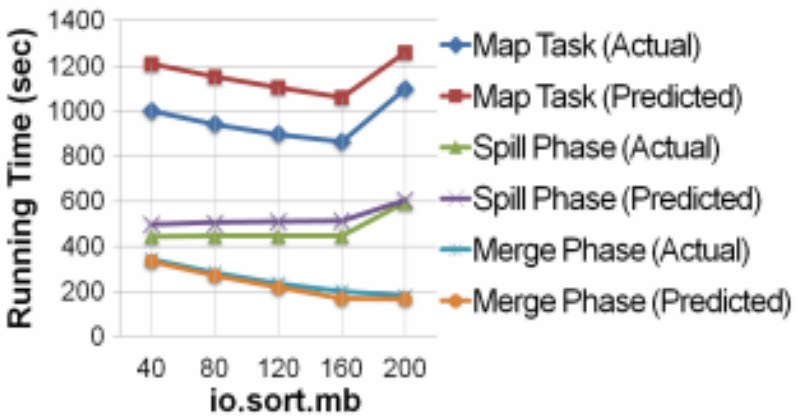
Information in Job Profile	*io.sort.mb*	
	120	200
Number of spills	12	8
Number of merge rounds	2	1
Combiner selectivity (size)	0.70	0.67
Combiner selectivity (records)	0.59	0.56
Map output compression ratio	0.39	0.39
File bytes read in map task	133 MB	102 MB
File bytes written in map task	216 MB	185 MB

Spill数減少 I/Oの低下

論文Figure4より引用

4. Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs

- **What-if Engine: Profilerの結果を利用して，パラメータ変更時の性能変化を推定**
  - Dataflow (Statics) fieldはデータ量に比例すると仮定することで推定
  - Cost (Statics) fieldは処理に利用する計算機が対照的な性能を持つと仮定して，I/OコストやCPU使用量を推定
    - 計算機が非対称な性能を持つ場合は他の処理を実行した際の統計を利用



論文Figure5より引用

- **Cost-based Optimizer: What-if Engineにクエリを投げ，コストを最小化するパラメータを探索**
  - 設定が必要な全パラメータcを部分パラメータc^iに分けて分割統治法で解く
  - GriddingやRecursive Random Searchにより部分パラメータc^iの変数を離散化し，探索点を減らすことで最適化処理を高速化