MapReduce
大規模分散データ処理の革新

Googleによる大規模クラスタ向けシンプルなプログラミングモデル

> 原論文: "MapReduce: Simplified Data Processing on Large Clusters"
> 著者: Jeffrey Dean and Sanjay Ghemawat (Google, Inc.)
> 発表: OSDI 2004

シンプルなインターフェースで大規模分散処理を実現した
Googleのビッグデータ処理基盤の技術解説

論文概要と登場背景

■ 論文の背景

- データ量の爆発的増加により大規模並列/分散処理の必要性
- 従来は開発者が分散制御や障害対応を個別実装
- 数千台規模のマシンクラスタでの処理が課題に

■ MapReduceの登場意義

- シンプルな抽象化による開発負荷の大幅削減
- 並列化・障害耐性・負荷分散を自動的に実現
- 関数型プログラミングに着想を得た明快なモデル

■ Googleでの大規模活用

- 2004年当時、毎月29,000以上のジョブが実行
- Webインデックス生成など基幹システムで使用
- 3,000TB以上の入力データ、数百台〜数千台規模での運用

MapReduceの基本概念

■ MapReduceとは

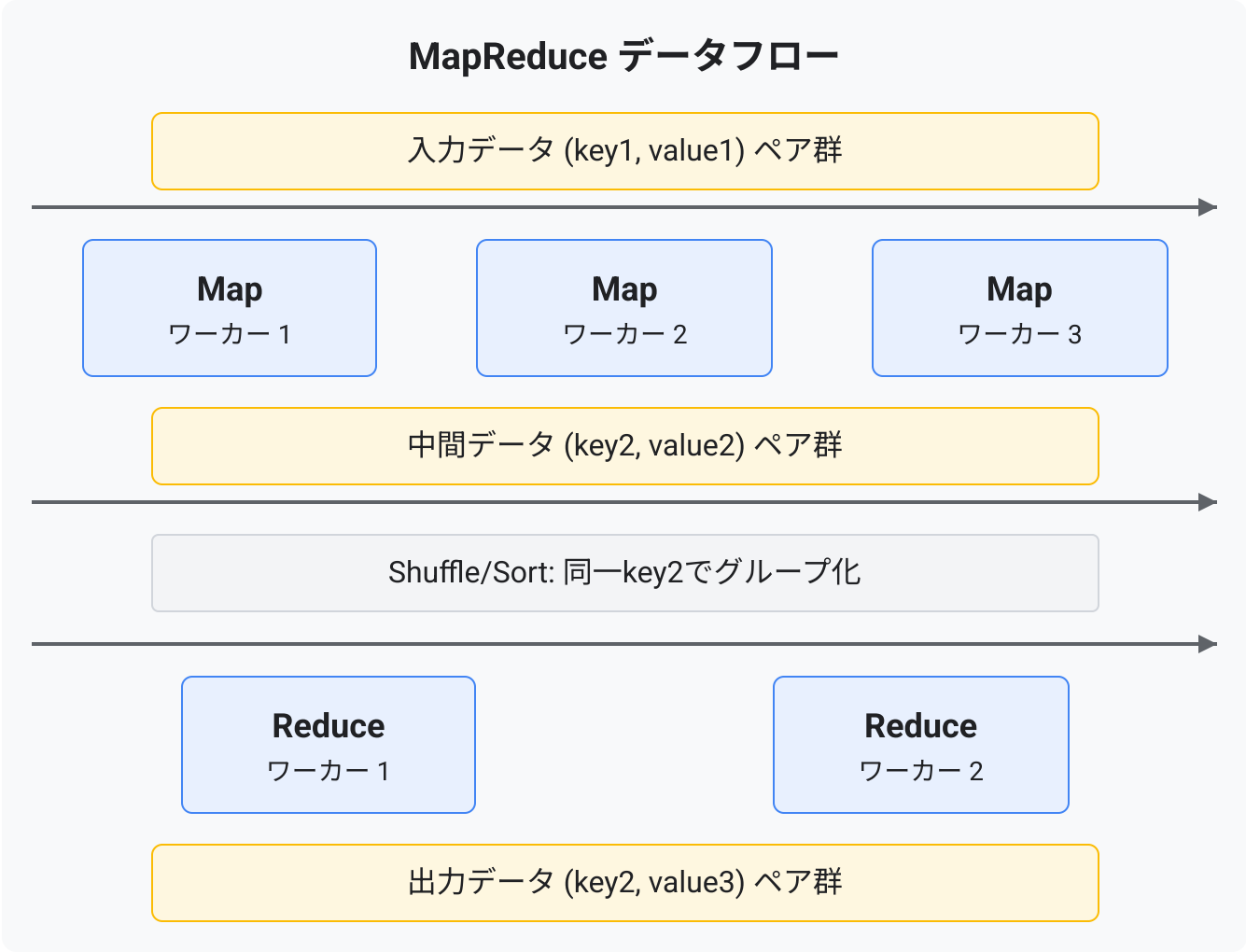
・大規模データセットの効率的な処理モデル
・関数型プログラミングに着想を得た設計
・単純な2つの関数で複雑な処理を表現

■ 2つの核となる関数

map(k1,v1) → list(k2,v2)
　入力データを中間ペアに変換
reduce(k2,list(v2)) → list(v2)
　同一キーの値をまとめて処理

■ 自動化される処理

・データの並列/分散処理
・障害復旧とリトライ
・ノード間通信とデータ転送
・スケーラビリティの確保



プログラミングモデル

■ キーバリュー型データモデル

・入力/中間/出力データを全て (key, value) ペアで表現
・型は任意、同一性比較が可能なもの
・キーの等価性でグループ化を実現

■ ユーザによる関数実装

・Map：入力を中間ペアに変換
・Reduce：同一キーの値をマージ
・関数の構成だけで分散処理を構築

■ 拡張機能

・**Combiner**：Map側で事前集約
ネットワーク転送量削減に貢献
・**Partitioner**：中間キーの分配制御
カスタムハッシュ分割などに使用

ワードカウント例

Map関数の例

```
map(String key, String value):
  // key: ドキュメント名, value: ドキュメント内容
  for each word w in value:
    EmitIntermediate(w, "1");  // 単語ごとに出現回数1を出力
```

Reduce関数の例

```
reduce(String key, Iterator values):
  // key: 単語, values: 出現回数のリスト("1","1",...)
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(key, AsString(result));  // 単語と合計出現回数を出力
```

Combiner適用例

```
// Mapタスク内で事前集約により転送量削減
combine(String key, Iterator values):
  // Reduceと同じロジックを各Mapノードでも適用
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  EmitIntermediate(key, AsString(result));
```

実行フロー

入力	Map出力	Shuffle後	最終出力
"Hello World"	(Hello, 1)	(Hello, [1,1])	(Hello, 2)
"Hello MapReduce"	(World, 1)	(World, [1])	(World, 1)
	(Hello, 1)	(MapReduce, [1])	(MapReduce, 1)
	(MapReduce, 1)		

システムアーキテクチャ

■ 全体フロー

・入力データをM分割→各Mapタスクに割当
・中間データをR分割→各Reduceタスクに割当
・全処理が終了するとジョブ完了

■ Master-Worker構成

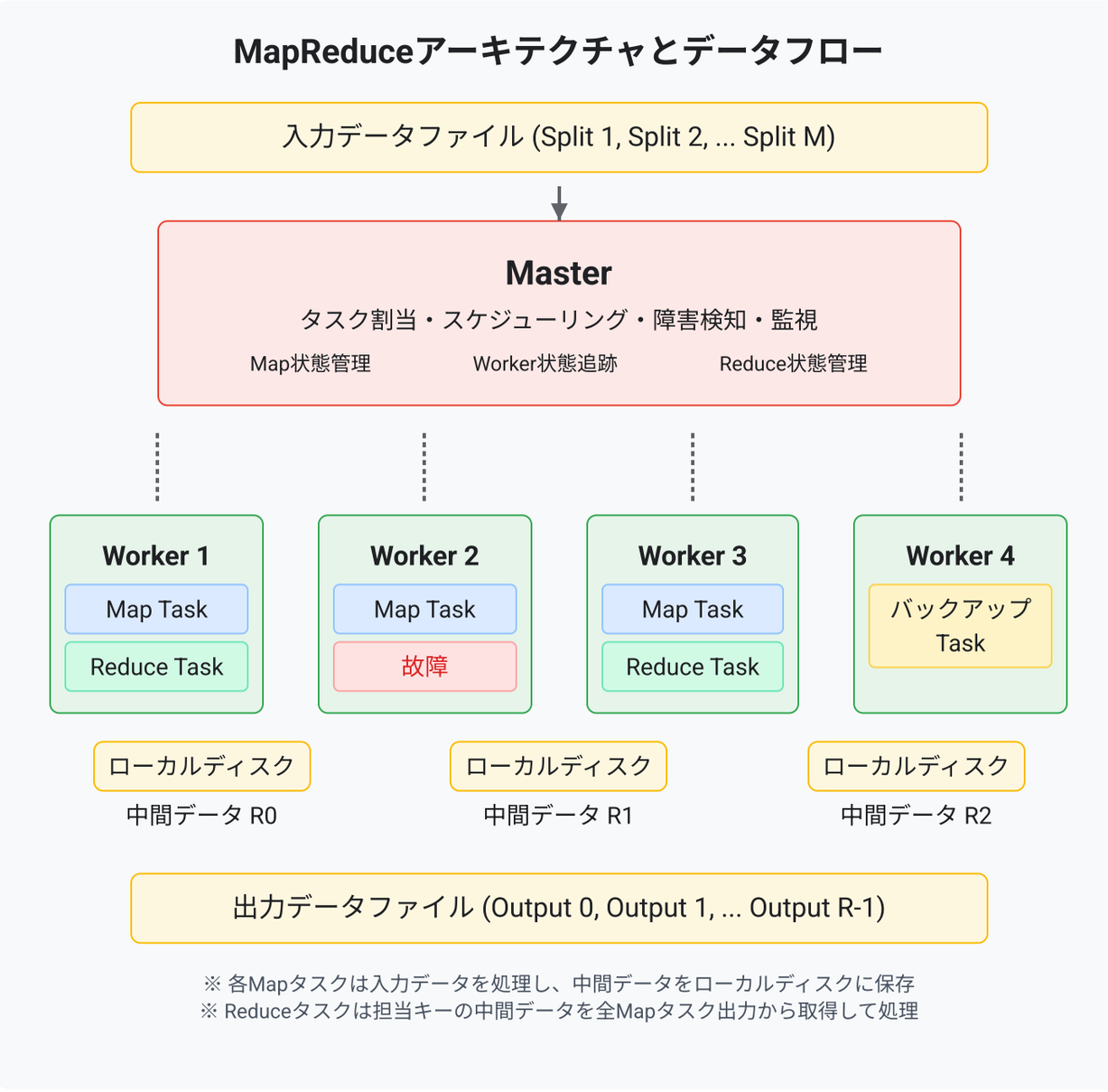
・Masterがタスク割当、進捗管理、障害検知を担当
・Workerは多数のマシンで並列実行
・同一ノードでMapとReduceを両方実行可能

■ ワーカー障害対応

・Masterが定期的にワーカーと通信
・応答のないワーカーは「故障」と判断
・故障ワーカーのタスクを再割当
・完了済みMapタスクも再実行（データ喪失対応）

■ ストラグラータスク対応

・処理が遅いタスク（ストラグラー）を検知
・同一タスクをバックアップで並列実行
・先に終了した方の結果を採用



実装の技術的詳細

■ M/Rタスク数の設定

・M/Rはワーカー数より多く設定（通常M≈16〜64MB単位）
・タスク粒度を小さくすることで動的負荷分散を実現
・一般的には M≫R（例: M=200,000、R=5,000）

■ ローカリティ最適化

・データが存在するマシンで優先的に計算処理
・GFSの複製配置を活用（入力データの3つの複製を把握）
・ネットワーク帯域の節約とI/O効率の向上
・遠距離ネットワーク転送を極力抑制

■ 障害耐性メカニズム

・定期的なping確認によるワーカー故障検知
・完了済タスクの冪等性確保（atomic renameによる）
・ストラグラー対策（遅延タスクのバックアップ実行）
・バックアップタスク：実行の最終段階で残りタスクを並行実行

技術的実装の詳細図

性能評価と実用例

■ ベンチマーク結果

・**Grep（1TB、100億レコード）**:
1764台同時稼働、**30GB/s**、完了**150秒**

・**Sort（1TB、100億レコード）**:
M=15000, R=4000、完了**891秒**
バックアップタスク無効時: **+44%**遅延
200台故障シミュレーション: **+5%**遅延

■ 運用実績（2004年8月）

月間実行ジョブ数: **29,423**
平均ジョブ完了時間: **634**秒
平均ワーカー数: **157**台/ジョブ
平均ワーカー故障数: **1.2**/ジョブ

■ 主な実用例

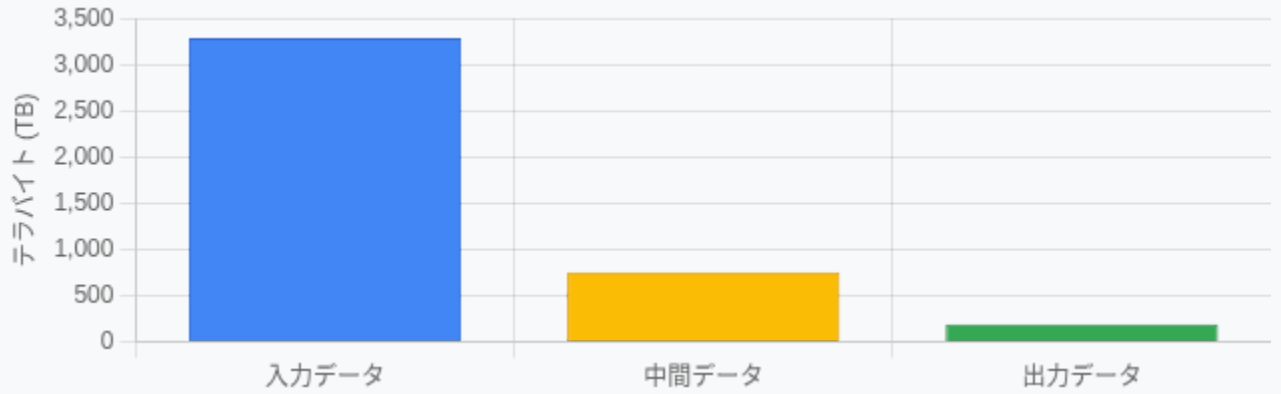
Google検索インデックス作成
5〜10段階のMapReduceチェーン処理
入力20TB超のデータを複数フェーズで処理

その他の活用事例
・機械学習モデルの構築
・クラスタリング、データマイニング
・Google Zeitgeistなど統計データ抽出
・大規模グラフ計算、リンク解析

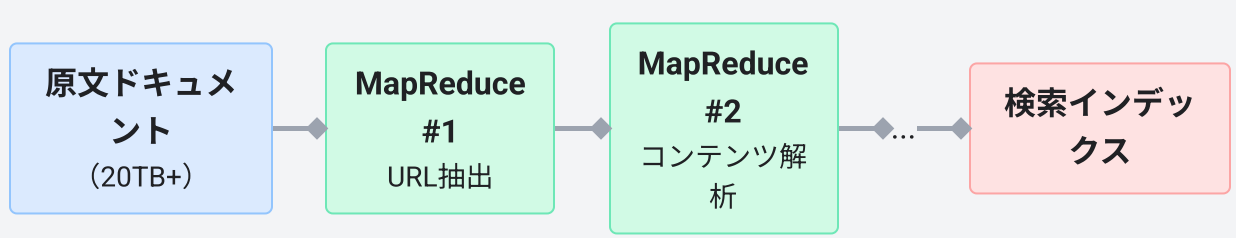
処理性能と運用データ

1TBデータ処理速度比較

MapReduce処理データ量（2004年8月）



Google検索インデックス作成パイプライン



5〜10段階のMapReduceを連鎖的に適用し大規模データを効率処理

まとめと意義

■ MapReduceモデルの強み

- シンプルな制約が並列処理効率とスケーラビリティを実現
- 隠蔽された複雑性（分散制御・障害対応・負荷分散）
- 柔軟性と表現力の両立（多様な実問題への適用）

■ 他システムとの比較優位性

- MPI等と比較し、障害耐性とスケーラビリティで優位
- 大規模分散処理に特化した設計と最適化
- 専門知識なしで並列分散処理を実装できる民主化

■ 分散処理技術への影響

- 制約によって処理効率と保守性を両立するモデルの先駆け
- 現代のHadoop、Spark等の分散処理フレームワークに影響
- ビッグデータ処理技術の基礎となる概念と設計思想