クラウドコンピューティング技術

伊藤孝行

名古屋工業大学大学院産業戦略工学専攻及び情報工学科
〒 466-8555 名古屋市昭和区御器所町
School of Techno-business Administration and Dept. of Computer Science,
Gokiso, Showa-ku, Nagoya 466-8555, Japan.

マサチューセッツ工科大学スローン経営大学院集合知研究センター
Center for Collective Intelligence, Sloan School of Management,
Massachusetts Institute of Technology (MIT),
5 Cambridge Center, NE25-749A, Cambridge, Massachusetss, MA 02142, US.

科学技術振興機構 JST さきがけ
〒 332-0012　埼玉県川口市本町 4-1-8　川口センタービル
Japan Science and Technology Agency, Kawaguchi Center Building
4-1-8, Honcho, Kawaguchi-shi, Saitama 332-0012 Japan
2010 年 8 月 6 日更新

概 要

本稿では，クラウドコンピューティングの技術的特徴を解説し，具体例とその応用に関する展望を示す．クラウドコンピューティングは，可用性に重点をおき，極めて大規模なデータ集合を保持するためのデータストア手法を用いた分散型の計算機利用形態である．従来の関係データベース（RDBMS）との違いは，RDBMS では，一般に，データの（強い）一貫性（Consistency）に重きを置いているが，クラウドコンピューティングでは一貫性よりも可用性（Availability）に重きを置いている点が異なる．そのため，両者のアーキテクチャは必然的に異なる．クラウドコンピューティングでは，ペタバイト級の大規模なデータ集合を，極めて高い応答性を保ったまま操作を可能とするデータストアの技術が用いられる．クラウドコンピューティングの一例である電子商取引のサイトなどでは，応答時間が数秒遅れるだけで，可用性が損なわれ，顧客の信用を失うことで経済的損害を被る．さらに，一部のデータセンターが竜巻にあって壊滅しても，ユーザがいつも通り商品を取引できる信頼性が求められる．強い一貫性を保持するという従来の RDBMS の方針では，大規模なデータ集合を扱う場合に，以上のような可用性及び信頼性を維持できるという実際的な例が少ない．複雑な RDBMS のオペレーションを大規模に維持するためには，極めて高価なハードウェアと，高い技術を持つ技術者が必要である．一方，クラウドコンピューティングの実際例である Google や Amazon は，低価格なコンピュータと単純に管理できるソフトウェアによって，高い可用性を維持しながら，信頼性と一貫性を保持するための綿密な工夫がなされている．クラウドコンピューティングの各技術要素は，分散システムや分散データベースでの過去の研究成果を実成果に即して大規模に応用したものであり，真新しい技術が発明されたわけではない．しかし，ネットワーク帯域が大きくなり，誰でもネットにアクセスできるようになったため，極めて大規模な非定型のデータを高い可用性を保持しながら扱う必要がでてきた．これは従来の RDBMS では対処しにくい状況であり，クラウドコンピューティングのような計算機の利用形態が重要になっている．

1 クラウドコンピューティングの技術的特徴

1.1 概要

クラウドコンピューティングという言葉によって、可用性に重点をおく分散システムの技術的特徴の重要性を世に示したのは Google である．Google では，超大規模なデータをどのように扱うかについて徹底した仕組みが開発・運用されている．学術的には分散システムの研究では，３０年前から，ネットワークの障害を前提とした場合，強い一貫性と高い可用性は同時に得られないことが指摘されている [1]．現在でも，(強い）一貫性（Consistency）に重きをおくべきだとする主張（RDBMS など）と，一貫性を緩めることで，可用性（Availability）を高めることに重きをおくべきだという主張がある．

クラウドコンピューティングの基盤は分散システムであり，強い一貫性よりも可用性や分散性に焦点をあてることで，極めて規模の大きいデータ集合を極めて高い可用性で運用することを実現している．ここでは，強い一貫性を緩めた結果一貫性（Eventual Consistency）という概念が提唱されている．強い一貫性を保持しないか

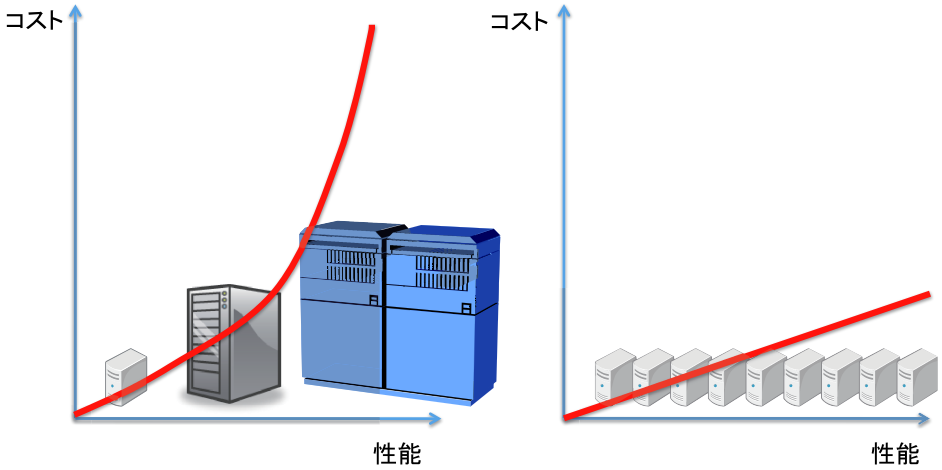


図 1: スケールアップとスケールアウト

らと言って，すぐにデータの一貫性を保証しないわけではなく，そこには実用上十分に高い可用性を高めるための様々な工夫が施されている．また，クラウドコンピューティングでは，その技術的特徴として，キーと値を対にして扱う（キーバリューストア）ことが多い．また，多くは，MapReduce というプログラミングモデルを用いている．一般に興味深い点は，Google，Amazon，マイクロソフトといった，次世代の IT 産業のリーダとなる新興企業がほぼ間違いなくクラウドコンピューティングの方式を用いている／注目していることである。

1.2 スケールアウトの有効性

クラウドコンピューティング技術で最も注目されるのはシステムのスケーラビリティである．特にクラウドコンピューティングでは，システムのスケーラビリティを高めるためにスケールアウトの方式が取られる．スケールアウトとは安価なサーバをたくさん導入することで 台数効果によって処理能力を拡大する方式である．安価なサーバであることがポイントで，Google などではコモディティ化したような安価なサーバを用いている．スケールアウトは，分散指向であり問い合わせが大量にある場合，保管するデータ量が大量にある場合などに役立つ．

一方，スケールアップは，処理能力を拡大するために，高性能・高機能・高価格な計算機を 1 つ（または少数）導入するという方式のことを言う．一般に，計算機を高性能化もしくは高機能化すると，指数関数的にコストは増加する（といわれている）．したがって，スケールアップの方式で高性能化・高機能化を行うと，費用は極めて高くなる．スケールアップは，集中指向でありデータの強い一貫性を保持する RDBMS に向いている．図 1 にスケールアウトとスケールアップに関する比較の概念図を示す．

とても身近なスケールアウトの例として，Apple の最上位機種 MacPro と、汎用機種 Mac mini（ディスプレイなし）の比較を行う（AppleStore での価格，2010 年

4月).「身近」といっても（1）のMacProは，ハイエンドサーバー並の性能を持っている．

（1）MacPro：2 x 2.93GHz Quad Core Xeon & 32GB memory 1,018,000円

（2）Mac Mini：2.66 Core 2 Duo & 4GB memory 100,000円

コストも性能も,（2）を8台購入すれば,（1）と同等の性能に近くなる．

1.3 ACIDに基づくデータベース

クラウドコンピューティングで実現されるものは，巨大なデータストアアプリケーションと言える．しかし，既存のデータベースアプリケーションとクラウドコンピューティングの違いを明確にするために，ここでは，データベースアプリケーションのトランザクションについて述べる．トランザクションとは，データに対する一つの論理的操作のことを言う．データベースの分野（特にRDBMSを扱う分野）では，トランザクションはACID基準を満たすべきだという見方と，そうでなくても良いという見方がある．まず，ACID基準について概説する．

ACID基準は文献[13]で紹介され，以下の4つの単語の頭文字による言葉である．A: Atomicity（アトミック性），C: Consistency（一貫性），I: Isolation(孤立性)，D: Durality (耐久性)．データベースの分野では，以上の4つの性質が満たされない限りデータベースは完全でないとされ，データベースは4つの性質を満たすべきであると主張される．実際はこの4つの性質を満たすことは高いハードルとなっている．文献[13]でも，いくつかの課題を上げている．

《A: Atomicity（アトミック性）》
各トランザクションは，完全な形で発生するか，全く発生しないかのどちらかであり，もし発生するなら，単一の分割不可の瞬間的な行為として発生する．トランザクションが起こっている間は，すべてのプロセスはその中間的な状態を見ることはできない．

《C: Consistency（一貫性）》
トランザクション開始と終了時において，あらかじめ与えられた整合性（不変性）を満たすことを保証しなければならない．トランザクション前にある整合性を維持していたなら，トランザクション後にも整合性を満たさなければならない．銀行のシステムにおいて，全体の預金のトータル量は，ある一つの送金のトランザクションの前と後で同じでなければならない．一方，トランザクションの途中は，整合性は維持されていないかもしれないが，トランザクションの外からは見えない．ここでいう一貫性は，強い一貫性ともいわれる．

《I: Isolation（孤立性）》
逐次処理可能性（serializable）とも言われる．すなわち，2つ以上のトランザクションが同時に動作したとき，最終的な結果はすべてのトランザクション

がある順序で逐次的に実行されたように見えなければならない．すなわち，同じデータにアクセスする二つのトランザクションはどちらかがもう片方のトランザクションが終了するのを待っている必要がある．

例として，銀行システムにおいて，２つの口座 A と B があり，次の２つのトランザクションがあるとする．

- トランザクション１：口座 A から口座 B に$10 送金する．
- トランザクション２：口座 B から口座 A に$20 送金する．

トランザクション 1 は次の二つのアクションからなる．

- アクション１－１：口座 A から$10 引く．
- アクション１－２：口座 B に$10 足す．

トランザクション２は次の二つのアクションからなる．

- アクション２－１：口座 B から$10 引く．
- アクション２－２：口座 A に$10 足す．

【良い例】内部処理がどのような順でアクション１－１，１－２，２－１，及び２－２を実行したかに関係なく，見た目として，トランザクション 1 が行われ，次にトランザクション２が行われた時，孤立性が保たれたと言う．

【失敗】内部処理として，アクション１－１，アクション２－１，アクション２－２，アクション１－２の順で行われ，最後のアクション１－２で失敗が起こった時，結果は，口座 A は$10 を引かれたままになってしまったとする．この場合，孤立性が保たれていないという．

《D: Durality （耐久性）》

一度行われたトランザクションの結果が永久に失われてはならない．多くのシステムでは，トランザクションのログが管理されており，故障が起きて復帰した時は，このログによって，システムをもとの状態に復帰させることが行われる．

以上のように，ACID 基準は極めて厳しい要求事項である．オンライントランザクションなどでは，大変重要な概念である．しかし，データベースを分散するシステムでは，すべてを満たすのは大変困難であることが知られている．次の CAP 定理は，分散システムにおいて，一貫性，可用性，および分割耐性が同時に満たされないことを示した経験則の一つである．

1.4 CAP 定理

CAP 定理 [2, 12] は，理論研究でいう定理というほどのものではないが，分散システムにおける一般的な経験則の一つである（実際に形式的に定理として証明されてはい

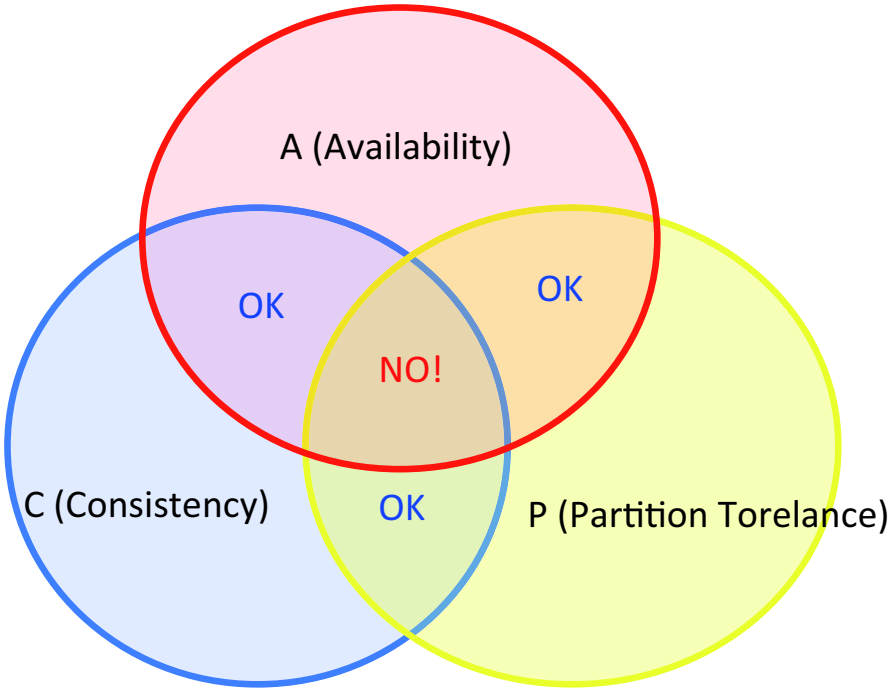


図 2: CAP 定理

る).「CAP 定理」の「C」は一貫性（Consistency),「A」は可用性（Availability），および「P」は分割耐性（Partition Tolerance）を表している．CAP 定理は，分散システムにおいて，一貫性，可用性，及び分割耐性を同時に満たすことはできないということを示した定理である（図 2）．2000 年の ACM Principles of Distributed Computing (PODC) シンポジウムの基調講演において，UC Berkeley の Eric A. Brewer 教授によって提唱された [2]．Brewer 教授は，Inktomi, inc.（現在は Yahoo!）の創業者の一人である．

一貫性（Consistency）は上で説明した通りで，ここでは，強い一貫性を意味している．

可用性（Availability）は，システムがすぐに使えることであり，システムが正常に稼働しているある瞬間に，ユーザがその機能を実行できる確率で表されることが多い．

分割耐性（Partition Tolerance）は，ネットワーク分断への耐性を意味する．例えば，二つの分割されたネットワークの各パーティションの間で，通信されるメッセージが失われる場合があっても，正しい処理が妨げられないという意味である．文献 [12] では，トータルなネットワークの故障以外の故障で，正しい処理が妨げられないこと，とも述べられている．

1.5 BASE

ACID 基準を満たすには，可用性を犠牲にしなければならない [8]．逆に可用性に注目したときの基準として，BASE が提唱されている [10]．BASE の意味は，Basically Available, Soft-state, Eventually consistent（基本的に可用性があり，ソフト状態保持をしており，結果的に一貫性がある）という意味である．

「基本的に可用性がある（Basically Available）」という意味は上で示した通り、基本的にシステムが正常に動作している瞬間にユーザがすぐにその機能を実行できるということである．

「ソフト状態を保持する（Soft-state）」という意味は，サーバがクライアントなどからの要求を受けたときに，クライアントの状態をある限られた時間のみ保持するという意味である．詳述すれば，状態保持に関しては次の 3 段階がある：状態不保持サーバ（stateless server），ソフト状態保持（soft-state），及び状態保持（stateful server）．状態不保持サーバとは，サーバはクライアントの状態をいっさい保持しない．例えば，基本的に Web サーバは状態不保持サーバである．すなわち，HTTP 要求に応答するだけのサーバである．ここでは，クッキーを用いることでクライアント側にデータを保持させたり，サーバサイドアプリケーション (Java Servler) でセッションオブジェクトを作ってソフト状態管理を実現している．

ソフト状態保持とは，状態不保持の特別な形で，サーバはクライアントの状態を保持するが，ある限られた時間のみ保持するという意味である．状態保持とは，サーバがクライアントの状態に関する情報を持続的に保持することを意味する．

サーバがクライアントの状態を保持している（状態保持）と，ファイルの更新などの効率を高めることができる．しかし，サーバが故障した場合の復旧は，故障した時点までのすべてのクライアントとの関連情報が必要である．状態不保持の場合は，サーバが故障して復旧する場合でも，クライアントの情報は何も必要ない．ソフト状態保持では，その中間的な特徴を持ち，ある限られた時間のみクライアント情報を持つので，サーバが故障しても復旧は，その限られた時間のみのクライアント情報が必要になるのみである．

「結果的に一貫性がある（Eventually consistent）」については，強い一貫性ではなく，結果として一貫性が保証されるという意味である．次の節でもう少し詳しく説明する．

1.6 一貫性（Consistency）

【強い一貫性（Strong Consistency）】 データベースの更新があった場合，その後のプロセスはすべてその更新を確実に確認できる．ACID 基準における一貫性と同義であり，RDBMS の存在意義そのものである．強い一貫性は理論的には可能だが，現実的にはネットワークの遅延がある限り，ロック制御を行って対応する必要がある．分散システムになると，ロック制御が複雑になり，かえって遅延が多くなる．その結果，システム全体としての可用性を保持するのは困難となる．

【弱い一貫性（Weak Consistency)】　データベースの更新があった場合，その後のプロセスが，その更新を確認できるとは限らない（強い一貫性以外の場合である）．

【結果一貫性（Eventual Consistency)】[22]　データベースの更新があった場合，その後のプロセスが，その更新を確認できない時期があるものの，一定の時間を経れば，必ずその更新を確認できるようになる．

最もよく結果一貫性を示していて，身近なのが，インターネットの DNS の仕組みである．DNS（Domain Name Service）というのは，インターネット上での名前（アドレス：www.u-tokyo.ac.jp など）と IP アドレスの解決を行うものである．あるひとつの DNS サーバに名前の更新を伝えると，その更新は隣接する（厳密には個々に登録してある）DNS サーバ同士に徐々に伝搬される．この状況では，違う DNS サーバで異なるデータを保持している可能性があるので，ある一つの名前に異なる IP アドレスがアサインされている可能性がある．しかし，最終的にはその更新はすべての DNS に行き渡り伝搬される事になる．これを結果一貫性と言う．

1.7　オンライントランザクション処理 (OLTP)

古典的な議論では，新幹線の座席予約システムなどに代表される既存の OLTP（On-Line Transaction Processing）では，スケールアウトでは対応しにくいといわれている．なぜなら，OLTP では，データベースの更新が頻繁に起こり，一貫性を保持するために頻繁に排他制御が必要になるためである．古典的な定義によれば，トランザクション処理は，ACID 基準を満たす必要がある．一貫性を保持するための排他制御による遅延をなるべく少なくするためには，同一サーバ機体の中でのメモリ上での処理が前提となる．スケールアウトを考えることも可能だが，スケールアウトによるデータベースの分割（Partitioning）によって，ネットワークごしの更新処理と排他処理が発生し，さらに大きな遅延が発生する可能性がある．またデータベースの分割そのものが難しい場合が多い．そのため，データの更新がリアルタイムに必要な OLTP にはスケールアウトは不向きと言われる．

OLTP で最近注目を集めているのが，東京証券市場の新システム arrowhead および arrownet である．株の売買取引処理 10ms 以下で実現できるオンラインデータベースである．証券取引情報は基本的には，頻繁に更新が起こるため，スケールアウト指向では対応できず，スケールアップ指向であり，予算は３００億円を投入しているといわれている．中心的なシステムはネットワーク越しの分散システムにはなっておらず，外側からのアクセスポイントとなるシステムとのネットワークでさえ，10GB/s のネットワークを投入している．基本的には，取引量がデータ量であるから，データのスケーラビリティ（パーティショントレランス）については，考える必要がない．すなわちデータを分割して大量に保管することを考える必要がない．

世界最大の電子商取引市場である Amazon のショッピングカートは OLTP として実装されているように思える．しかし，古典的な視点から見るとトランザクション処理とは言えない．なぜなら，ACID 基準を満たしていないためである．実際に，Amazon のショッピングカートを支える Amazon の分散システム Dynamo[8] はスケールアウ

ト指向で ACID 基準を満たすようには設計していない．ただし，文献 [8] が発表された時点で２年間以上の実働において write（書き込み更新処理）は失敗していないと言われている（このような分散システムでは，write は read よりも失敗しやすいように設計されるのが普通であった）．

Dynamo は結果一貫性（eventual consistency）に基づいている．すなわち，ある時点での，あるデータに対し，複数のバージョンの存在を許している．ただし，例えばある１つのデータに対して４つのバージョンが存在する可能性は 0.00009%であり，さらにこれらのバージョンが違うデータがあった場合にもいくつかの対処法が用意されている．スケールアップ指向の極めて高機能かつ高コストなサーバの故障確率を考えれば，Dynamo はそれに匹敵するパフォーマンスを発揮していることがわかる．以上のように Dynamo は，ACID 基準からすればトランザクション処理をしているとは言えないが，実行的には十分にショッピングカートの要件を満たしていると言える．

1.8　データ読み込みの遅延

ACID 基準，及び，強い一貫性は，理論的には望ましいが，CAP 定理が示す通り，データの読み書きに遅延が発生する限り，可用性と同時に強い一貫性を実現するのは現実的には難しい．記憶装置へのデータの書込みはもちろん，読み込みにも現実的には時間がかかり，遅延が発生するためである．具体的には，ハードディスクの容量は爆発的に増え続けてきたが，ハードディスクのシークタイムは伝統的に 9ms 〜12ms であり，ほとんど改善されていない．ハードディスクの構造は，回転する円盤の上のデータをアームで読込むという構造なので，シークタイムの改善は根本的に難しい．近年急激に需要を延ばしている SSD になるとシークタイムは改善されるが，読込む対象のデータが大きければ，それだけ時間がかかるという根本的な問題は変わらない．例えば，SSD からの読込速度および転送速度が仮に 200M バイト /s とする（現行の SSD の読込速度に準ずる）．2T バイト (=2,000,000M バイト) の記憶領域をすべて読込むのには，それでも 10,000 秒（約 3 時間）かかることになる．

従って，システムの可用性を高めるためには，記憶装置を分散配置し，分散アクセスすることが必要である．上の例で，100 台を分散配置すれば，100 秒（約２分）で読込むことができ，10,000 台を分散配置すれば，1 秒で読込むことができるかも知れない（分散配置に伴うネットワーク遅延などもあるため，線形には効果が表れない）．

1.9　ハードウェアは壊れる

可用性を高めるために，計算機を分散配置しても，次の問題は，ハードウェアの故障の問題である．例えば，理論的に 10 年に 1 度しか故障しない計算機だとしても，100,000 台分散配置すれば，１日に 1 台は故障が発生することになる．実際には，それ以上に故障が発生している．クラウドコンピューティングでは，可用性を保持す

るため分散処理をすることによって，ハードウェアは頻繁に故障することを前提にする必要がある．

古典的な研究においても，ネットワークの故障を前提にした場合，強い一貫性と高い可用性を同時に得ることが難しいことが指摘されている [1].

また，Google の最近の報告によれば，ハードディスクの故障は，使用される環境での温度や，使用された年月には，あまり固定的な影響がみられなかったとしている．さらに，初期のスキャンにおいて，スキャンエラーがあったドライブは，そうでないドライブに比べて，60 日以内に故障する可能性が 39 倍だったとのことである [19].

1.10 分散読込みデータの結合

計算機を分散配置し，データを分割することで，可用性を保持するが，分散して読込んだデータをどのように結合するかは，古典的に難しい問題とされる．Google や Amazon などの実世界でのシステムでは，様々な工夫がなされている．下で示す Google の MapReduce はその一つで，上記の問題を抽象化したプログラミングモデルを提供している．

1.11 MapReduce

Google によって提唱された MapReduce は，巨大なデータセットを処理したり，生成するためのプログラミングモデルである [7]．ユーザは，あるキーバリューのデータを処理し，中間的なキーバリューのデータ集合を生成するようなマップ関数（Map）を定義する．そして，中間的なキーが同じデータ集合をマージするようなレデュース関数（Reduce）を定義する．2003 年に Google がウェブサーチのための転置インデックスを簡単化するために作った．それ以後，Google では，巨大なスケールのグラフの処理，テキスト処理，機械学習，統計的機械翻訳など，10,000 以上のプログラムが作られてきた．MapReduce に関しては，本レポートの具体例の章でさらに述べる．

1.12 RDBMS（リレーショナルデータベース）との違い

クラウドコンピューティングと伝統的な RDBMS（リレーショナルデータベース）は，互いに補完し合うものである．表 1 に現状における比較をした．上で議論したように，クラウドコンピューティングは対象とするデータは，サイズが極めて大きく，バッチ処理で，書き込みは一度限りで何度も読み出しが多く，構造化されていない．そして，分散処理を行うのでスケーラブルである．一方、伝統的な RDBMS は，アクセス方法がインタラクティブで，高速に読み書きの更新がおこる．可用性を保持するためには，構造化されたデータを対象とし，そのため分散処理をすることには向いていない．

表 1: RDBMS とクラウドコンピューティングの比較

比較	伝統的な RDBMS	クラウドコンピューティング
データサイズ	ギガバイト	ペタバイト以上
アクセス	インタラクティブまたはバッチ処理	バッチ処理
更新	高速に何度も読み書き	書き込みは一度で，何度も読み出し
構造	構造化されている	非構造化・準構造化
分散処理	向いていない	分散処理が前提

1.13 キーバリューストア

クラウドコンピューティングでは，データをキーとバリューの対の形で保持する．これをキーバリューストアという．RDB のような複雑なスキーマは仮定しないのが基本である．例えば，ワードカウンタ（単語の数え上げ）のプログラムなら、《単語，出現回数》という形になる．

2 並列 RDBMS との比較

MapReduce は，クラウドコンピューティングの技術特徴の中でも，Google が提唱しているプログラミングモデルであり，インパクトがある．必ずしも新しいアイデアではないため，様々な議論が生まれている．例えば，並列 RDBMS との比較は，インターネット上のブログをはじめ，最近の国際会議や学術雑誌でも行われている．並列 RDBMS の主な主張としては，『並列 RDBMS は正規化されたデータを扱う限り，データのロードに時間がかかるが，データをロードさえしてしまえば，クラウドコンピューティング（Hadoop）上の MapReduce より高速に様々な処理ができる』と言われている．

しかし，明らかに，非定型のデータが，大量に発生し蓄積する状況では，並列 RDBMS で対応するのは難しいことが分かる．データのロードが遅いのは，データを正規形として，データベースに一つ一つ INSERT する必要があるからである．さらに，現実世界のデータを扱う場合，それらのデータを SQL で扱える形に整形化すること自体が難しい．

文献 [18] では，Hadoop の MapReduce と，他の 2 つの並列 RDBMS として，DBMS-X（仮名）と Vertica を比較している．文献 [18] は通称「Comparizon Paper」として知られている．

まず，データのロードに関しては，様々なベンチマークにおいて，MapReduce の方が並列 RDBMS より早い．データをロードした後のいくつかの処理に関しては，並列 RDBMS の方が「劇的に」早いと結論付けている．例えば，Grep タスク, ペー

ジランクの高いものをフィルタするタスクなどがベンチマークとして実行され，平均として DBMS-X は MapReduce より 3.2 倍早くなり，DBMS-X より Vertica より 2.3 倍早いという結果を示している．ただし，文献 [18] の著者は MapReduce 批判派として知られているため，むしろ，「正規化されたデータの処理という MapReduce のウィークポイントに関して，既存の分散データベース技術で補完できる」という意味で捉えるのがフェアな理解と言える．

さらに文献 [20] では，著者はほぼ同じグループだが，MapReduce と並列 RDBMS は，互いに補完しあう関係にあるものである，と結論づけている．本論文では，並列データベースシステムについての利点と，MapReduce の良さについていくつかの視点からまとめらている．例えば，データベースの水平分割（horizontal partition）による分散実行は，既存の並列データベースシステムの得意な処理であることが示されている．一方で，MapReduce の長所は，Map 関数と Reduce 関数というその単純さや，コストが少ないことにもあると述べられている．さらに以下のそれぞれの応用について，MapReduce か並列 RDBMS か，どちらを応用すべきかが論じられており，以下においては MapReduce のほうが優れていると述べている：ETL(Extract Transform Load) and“read once”data sets，Complex Analysis，Semi-structured data，Quick-and-dirty analysis（使いやすさとコスト）．

さらに，文献 [20] では文献 [18] の結果をさらに精緻化した結果を延べ，Grep task，Web log task，Join task，で，並列 RDBMS の処理時間のほうが早いことを再度強調している．また，その原因として，Hadoop の MapReduce は，レコードのパージングをしていない，圧縮をしていない，パイプライン化が精緻でない（一度中間ファイルを出力しまた読み込む），タスクのスケジューリングが精緻でない，などを上げている．ただし，これらの欠点は，今後の改良でさらに精緻化される可能性はあるとしている．なぜなら，オープソースとしての Hadoop のコミュニティは，議論を続けているからである．商用の DBMS 製品についてもそのユーザビリティについてもっと改良すべきであると言及している．

文献 [7] では，もともと MapReduce の論文を発表した Google の Jeffrey Dean らが，上記 Comparizon Paper[20] や有名なブログ記事「Mapreduce:A Major Step Backwards」などの批判一つ一つに対して明確に反論している．

上記 Comparizon Paper やブログ記事は，フェアな立場から見ても，幾分，分散 RDBMS が有利な結果に偏った結論を示しているように思えるが，上記文献 [7] と同時に ACM Communications にて発表され，Comparizon paper の筆者の一人が執筆した文献 [20] では，”Most of the architectural differences discussed here are the result of the different focuses of the two classes of system.”と結論づけており，MapReduce は，ETL (Extract-Transform-Load) システムであって DBMS ではないので，Parallel RDBMS とはフォーカスの範疇が違うという結論を示している．

もともと，MapReduce は，プログラミングフレームワークであり，データストア技術ではない．Google ではむしろ，BigTable や GFS が，データストアの部分を扱う．その意味で，Comparizon Paper は，多少フォーカスのずれた比較をしている．

文献 [20] からもわかるように，MapReduce は ETL に特化したモデルであり，Google の BigTable，および GFS などのシステムと一体で扱うことで巨大なデータストア

を実現している．一方で Parallel RDBMS はデータをストアするという部分のみに特化したシステムであり，データの抽出・解析・ロードなどは対象外であり，さらに大規模システムには必須の可用性を諦めて，強い一貫性を保持しようとするシステムである．

3 具体例:Google's GFS, BigTable and MapReduce

クラウドコンピューティングの草分けの Google のシステムは，シンプルだが合理的な設計になっている．クラウドコンピューティングの仕組みを支える GFS(Google File System)[11]，GFS と連携してデータ管理の仕組みを提供する BigTable[5]，ロックシステムである Chubby[3]，および MapReduce[6] などからの構成要素からなる．

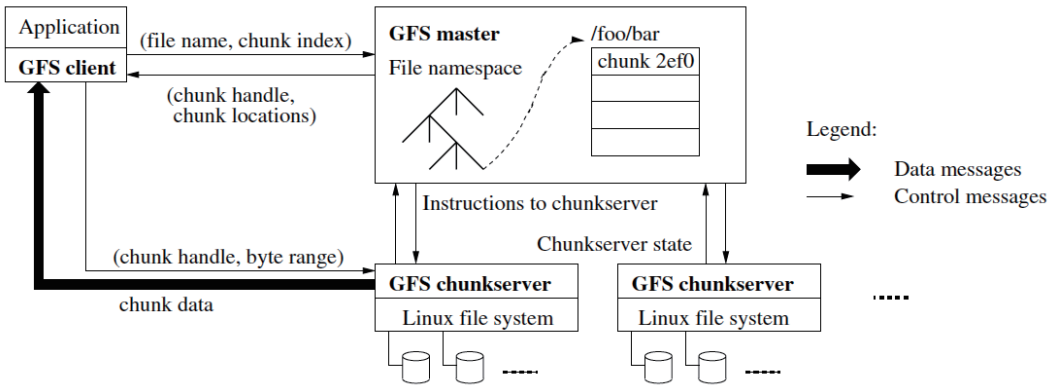
3.1 GFS：Google File System

GFS は，巨大な情報を扱うためのスケーラブルな分散ファイルシステムである．耐故障性が高く，コモディティとしてのマシンをたくさんつなげることで，高い性能を発揮する．

《設計方針》

GFS の設計方針として特徴的なのは以下である．

- システムは，多数の安価な計算機（コモディティ化した計算機）によって構成されており，良く故障する．計算機の故障は例外ではなく，通常である．
- システムは，巨大なファイルを保持している．１００ MB 程度のものが多いが、数 GB のものも一般的である．小さなファイルも扱うべきだが，小さなファイルに対して最適化はしない．
- 考えるべき負荷として主なものは以下の２つの読み取り（read）である：非常に多くのストリーミング型の読み取りと，少数のランダム読み取りである．
- また，逐次的な書込みも負荷となりえる．それは，ファイルにデータを追記する（append）形のものである．
- 多数の並行的な追記が生じるため，追記におけるアトミック性は最小限確保する．
- 大きい通信帯域を確保する．



Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, “The Google File System”, SOSP 2003 より引用.

図 3: GFS のアーキテクチャ

《アーキテクチャ》

GFS のアーキテクチャの概要を図 3 に示す．GFS は，１つのマスターと複数のチャンクサーバから成り，複数のクライアントからアクセスを受ける．ファイルはチャンクとして分割される．信頼性のために，各チャンクは，３つレプリカが作成され異なるチャンクサーバに配置される．レプリカは異なるラックに配置される．災害でラックが破壊されても動くことができるようにである．マスターは，システム全体のメタデータを管理し，HeartBeat と呼ばれるメッセージを定期的にチャンクサーバに送ることで，命令を出したり，現状に関する情報を得る．

図に示すように，クライアントはチャンクサーバからデータを読み書きするが，マスターを介在してデータを読み書きすることはない．これはマスターがボトルネックになることを防ぐためである．チャンクのサイズは 64MB であり，これは一般的なファイルシステムにおけるファイルサイズよりも大きい．チャンクは，Linux ファイルとして管理されている．ファイルサイズが大きいことで，クライアントとマスターのインタラクションを減らすことができる，クライアントは続けて様々な操作ができるためネットワーク負荷を減らすことができる，メタデータのサイズを減らせる，などの特徴がある．一方で，一つだけのチャンクからなるような小さなファイルも構成され，スペースアロケーションが無駄になる場合もある．

マスターは以下のメタデータを持つ：ファイルとチャンクの名前空間，ファイルからチャンクへのマッピング，及び，各チャンクのレプリカの場所．メタデータはマスターのメモリ上におかれるため，その操作は高速にできる．マスターはチャンクとそのレプリカの関係の情報は持っていないが，スタートアップじと定期的な HeartBeat メッセージでこれらの情報を得る．初期の頃は，チャンクの情報をマスターに継続的に保持することを試みていたが，上のように，スタートアップ時と定期的な HearBeat メッセージで情報を得るだけで十分だった．

GFS の中心的存在と言えるのは，オペレーションログである．オペレーションログは，重要なメタデータの変更が記載される．これはメタデータの継続的な記録と

してだけでなく、平行処理の論理的タイムラインを決定する．マスターは，システム障害がおこった場合，オペレーションログを再生することで，現状を復帰する．オペレーションログは，複数の遠隔マシンに複製がおかれる．

《GFS の一貫性モデル》

GFS は弱い一貫性（もしくは結果一貫性）のモデルである．まず，各ファイル更新はアトミック性を保持している．マスターのオペレーションログは，オペレーションの全順序をグローバルに決定する．データ更新がおこった後の，そのファイルの領域の状態については，成功したか否か，並行更新があったかどうかに依存する．もし，ある領域への書込みが成功し，かつ他に並行的な書込みがなかった場合，その領域は一貫性がある（consistent）と言う．また，もしある領域への書込みが成功し，他に並行的な書込みがあった場合，その領域は一貫性があり，かつ定義されている（defined）と言う．GFS では，逐次的な更新が成功すると，更新されたファイル領域は定義された状態になる．一方，並行的な更新が起こった場合，定義されると保証はされないが，アプリケーション側でチェックサムなどを利用するなど，工夫している．特に書込みは追記（アペンド）が多いため，チェックサムもインクリメンタルに使うことができる．

レプリカに対して更新の一貫性を保持するために，リース（lease）を用いる．マスターは，レプリカの一つにリースを与え，それをプライマリとする。プライマリは，すべての更新に連番をつけて管理する．すべてのレプリカはこの順番に基づいて更新を適用する。

《ベンチマーク》

文献 [11] にあるベンチマークの一つを上げる．本ベンチマークは，ファイルシステムの性能を評価するものである．１マスター，２マスターレプリカ，および１６のチャンクサーバからなる小さい例での評価である．

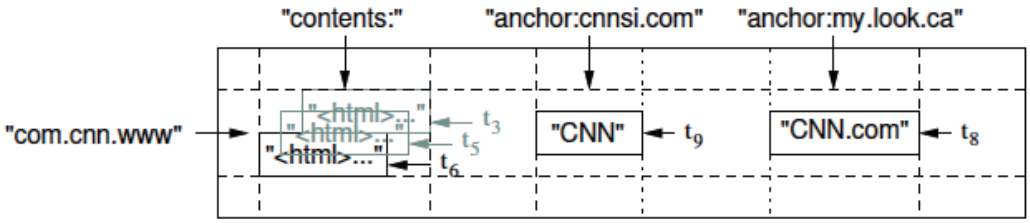
Read では，N クライアントがシステムから同時に読出しを行う．台数が増えれば増えるほど，同じチャンクサーバへアクセスするクライアントが増えるため，台数効果が理想値ほど得られていない．

Write では，N クライアントがシステムに同時に書込みを行う．読込の時よりも，コリジョンが起こるので，性能が低くなっている．

Record Append では，N クライアントがあるひとつのファイルに同時に追記をしている．チャンクサーバのネットワーク帯域が小さいために，性能は限られている．実応用では，アプリケーション側で追記すべき一つのファイルを選ぶ仕組みができれば問題ない．

3.2 Google BigTable[5]

BigTable は，多くの汎用サーバの上のペタバイト級の巨大データを扱うために設計された分散ストレージシステムである．BigTable によって，構造化してデータを扱うことができる．BigTable は，６０の Google の製品やプロジェクトで使われている：



BigTableの基本的なフォーマットを表している．行キーはcom.cnn.wwwである．ドメイン名ごとに集めると効率が良いので逆さにしている．列キーは “contents:”, “anchor:cnnsi.com”, “anchor:my.look.ca”である．列キーは「family:qualifier」という形になっており，例えば、「contents:」にはwww.cnn.comのコンテンツが入っている．「anchor:cnnsi.com」は，cnnsi.comの「CNN」というアンカーからリンクが張られていることを意味している．以上のように，データを構造化して保存することができる．

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber, “Bigtable: A Distributed Storage System for Structured Data”, OSDI2006 より引用．

図 4: BigTable のデータモデル

例えば，Google Analytics，Google Finance，Orkut,Personalized Search，Writely, Google Earth などである．BigTable では，RDB が扱うようなリレーションは扱わない．

《データモデル》

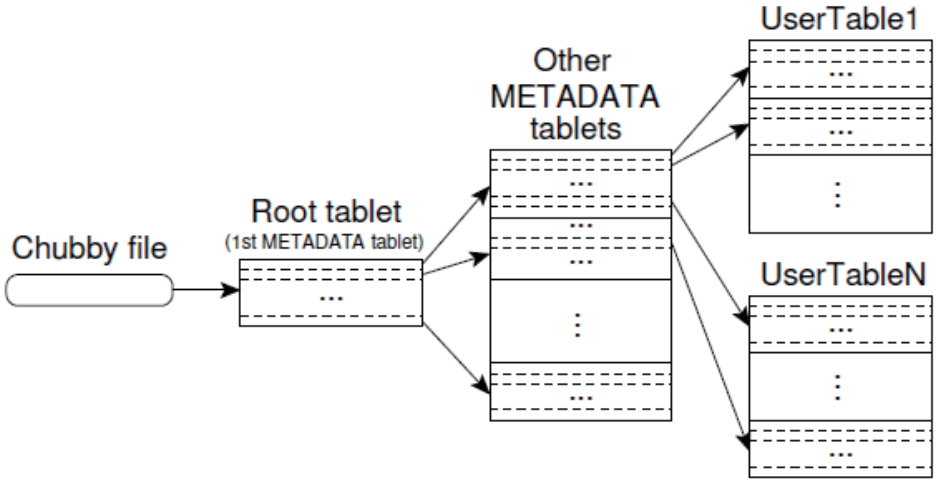
BigTable は，多次元ソートマップである，すなわち，マップは，行（row），列（column），およびタイムスタンプで定義される．行のグループをタブレットとして扱う．列キーのグループを列ファミリーとして扱う．列キーは「family:qualifier」というシンタックスを持つ．例えば，family には,「anchor」や「language」がある．例えば，図 4 の「anchor:cnnsi.com」は,「cnnsi.com のアンカー」を表している．BigTable の各セルは，タイムスタンプがつけられており，複数のバージョンを持つことができる．

《タブレット》

BigTable は，クライアントへのリンクに関するライブラリ，1 つのマスターサーバ，および複数のタブレットサーバから構成される．BigTable はデータやログを，Google File System を使って管理している．

タブレットの位置は，図 5 のような B+Tree に似た階層構造で管理されている．Chubby が，Root タブレットの場所を持っていて，Root Tablet が METADATA を持つタブレットの集合の場所を持っていて，METADATA に行キーに基づいたタブレットの場所が示されている．

タブレットは，複数の SSTable によって構成される．SSTable は，キーバリューに関するソート済み不変マップ（読み込み専用）である．SSTable は 64k ブロックがいくつか、とそのブロックに関する index を持つ．SSTable が生成されると，そ



Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber, “Bigtable: A Distributed Storage System for Structured Data”, OSDI2006 より引用.

図 5: タブレットの階層図

の SSTable への index のみがメモリにアップロードされ，ディスクへのアクセスが最小限になるように工夫されている．

各タブレットはタブレットサーバによって管理される．マスターサーバは，タブレットサーバが中断しているか動作しているか，どのタブレットが割り当てられているかなどを管理し，タブレットの割り当てなども行う．タブレットサーバの状態管理は，Chubby ファイルシステムを用いて行われ，タブレットサーバ故障時などは，マスターは Chubby ファイルシステムに基づいて，タブレットの再度割り当てなどを行う．

《Memtable，コンパクション》

BigTable への更新は，すべてログとして残され，メモリ上の memtable によって管理される．メモリ上の memtable が大きくなると，SSTable として変換し GFS に書き込む（マイナーコンパクション）．また，SSTable が増えるとそれを一つの SSTable にマージする（メジャーコンパクション）．

《Chubby[3]》

Chubby[3] は，分散ロックサービスである．Chubby は，５つのレプリカから構成され，その中の１つが他の４つからマスターとして選ばれ，実際にマスターとして，外部からの要求を受け付ける．これらのレプリカの多数が故障せず，かつ互いに通信ができれば，Chubby は動作し続ける．

《パフォーマンス評価》

図 6 に示すように，BigTable に標準装備されている Scanner API を使った，scan

Experiment	# of Tablet Servers			
	1	50	250	500
random reads	1212	593	479	241
random reads (mem)	10811	8511	8000	6250
random writes	8850	3745	3425	2000
sequential reads	4425	2463	2625	2469
sequential writes	8547	3623	2451	1905
scans	15385	10526	9524	7843

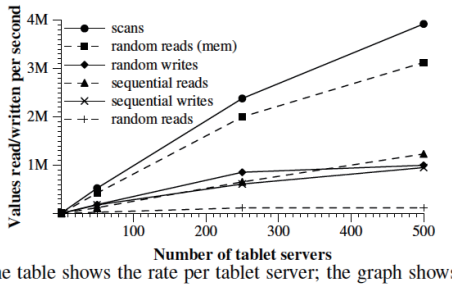


Figure 6: Number of 1000-byte values read/written per second. The table shows the rate per tablet server; the graph shows the aggregate rate.

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber, "Bigtable: A Distributed Storage System for Structured Data", OSDI2006 より引用.

図 6: BigTable のパフォーマンス評価

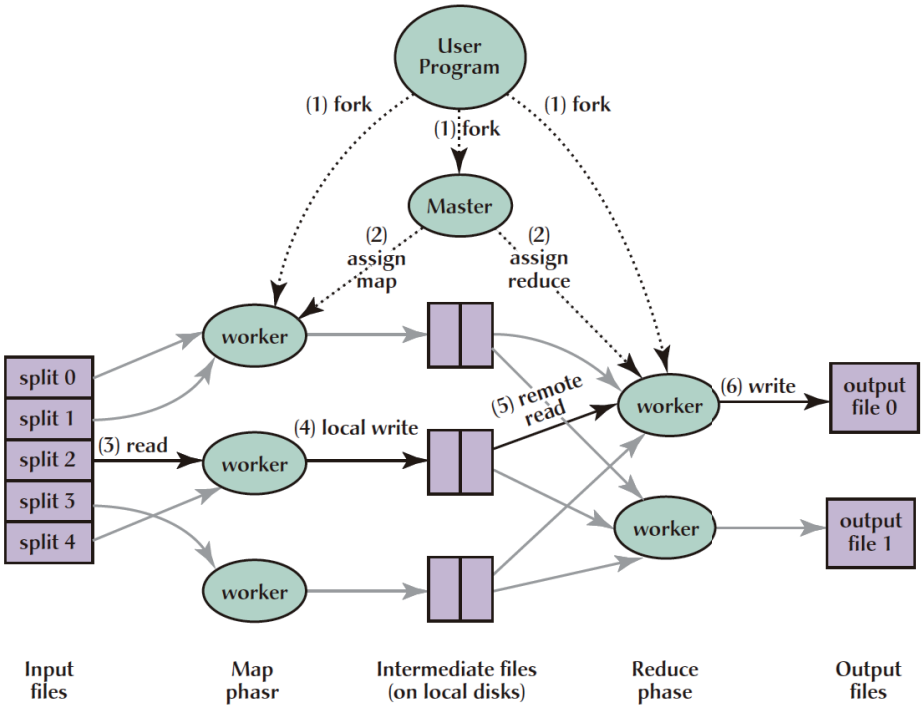
が最も早く，台数効果もえられていることがわかる．また，SSTable をすべてメモリにアップロードした上での random read(mem) も, 良いパフォーマンスを示している．書き込みは一般的に遅い．また，random read は，GFS へのアクセスがその都度必要な上に，random なアクセスが必要なので遅い．

3.3 MapReduce

Google によって提唱された MapReduce は，巨大なデータセットを処理したり，生成するための抽象化されたプログラミングモデルである [6]．Google は，MapReduce という抽象化モデルを考案するまでに，大規模で複雑な計算をするためのさまざまな並列分散処理のプログラムを開発してきた．その中で，複雑な計算をシンプルに表現するために，LISP や関数型言語の命令である map と reduce を使って，MapReduce として実現した．MapReduce では，ユーザは，大量のデータを処理し，中間的なキーバリューのデータ集合を生成するようなマップ関数（Map）を定義する．そして，中間的なキーが同じデータ集合をマージするようなレデュース関数（Reduce）を定義する．2003 年に Google がウェブサーチのための転置インデックスを簡単化するために作った．それ以後，Google では，巨大なスケールのグラフの処理，テキスト処理，機械学習，統計的機械翻訳など，10,000 以上のプログラムが作られてきた．

図 7 に MapReduce の実行フローを示す（論文より引用）．MapReduce の一連のオペレーションは以下の通りである（Google で実装されている MapReduce についての解説だが，Hadoop MapReduce など他のオープソースの実装でもほぼ同様と考えられる）．

【1】 MapReduce ライブラリが入力されたファイルを 16-64MB 程度の M 個の塊（split）に分割する．複数のプロセスが生成される（fork）．



Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004より引用.

図 7: MapReduce の実行フロー

【2】 プロセスは，1つマスターと複数のワーカーとされる．マスターはワーカー達に M 個の Map タスクと R 個の Reduce タスクを割り当てる．一つのワーカーには，一つの Map タスクか一つの Reduce タスクを割り当てる。

【3】 Map タスクを割り当てられたワーカーは，入力情報の分割されたものの内容を読みとる．キーバリューペアをその入力情報からパースし，そのキーバリューをユーザが定義した Map 関数に渡す．この Map 関数によって，中間キーバリューペアが生成され，ローカルのメモリにバッファされる．

【4】 メモリにバッファされたペアは，R 個の領域にパーティショニングされながらローカルディスクに定期的に書き込まれる．ペアがどこに書き込まれたかという情報は，マスターに知らされ，マスターは reduce ワーカーにこの情報を知らせる．

【5】 reduce ワーカーの一つが，ペアを書き込まれた場所を知らされると，この reduce ワーカーは remote procedure call を使って，map ワーカーのローカルディスクからバッファされたデータを読む．Reduce ワーカー 1 つが，そのパーティションのすべての中間データを読み取ったら，中間キーをすべてソートし，出現するすべてのキーがグループ化されるようにする。一般には、異なる多くのキーが同じ reduce タスクに map されるので，ソーティングは必要である．

【6】 Reduce ワーカーは，各中間キーとそのバリューを，ユーザの reduce 関数に渡す．Reduce 関数の出力は，そのパーティションにある出力ファイルに追加（append）される．

【7】 すべての map タスクと reduce タスクが完了したら，マスターはユーザプログラムに制御を戻す．

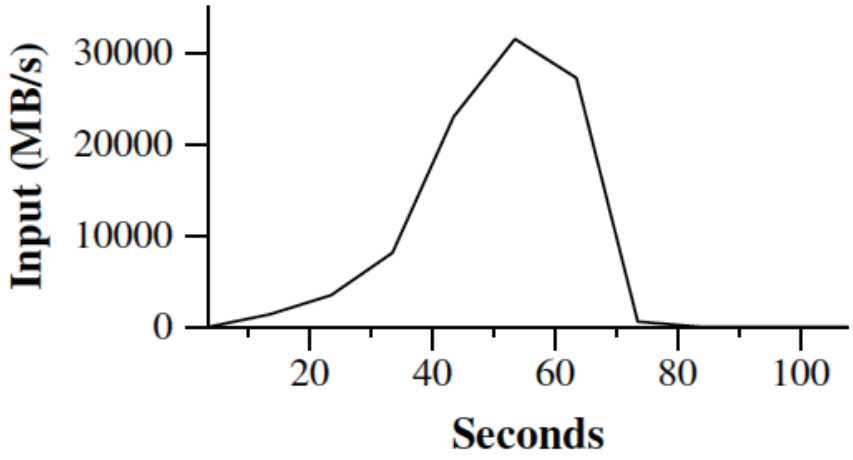
完了後は，この MapReduce の実行の出力は，R 個の出力ファイルとして利用可能である．一般には，これらの R 個の出力ファイルは，1 つにまとめるよりも，他の MapReduce タスクに引き継がれる場合が多い．

マシンの故障に対する対応は，map ワーカー A が故障した場合，他の map ワーカー B を立ち上げ，それまでに A が出力した中間データを読み取った reduce ワーカーは，ワーカー B からの出力を読むような対応をする．マスターの故障は，多くはないので，マスターが故障した場合は，その MapReduce タスクをすべて中止する．

性能評価実験として，10^{10}100 バイトのファイルに対する MapReduce を使った (分散)grep がある．6 年前になる 2004 年に発表された論文で，全体の計算はおよそ 150 秒で終了している．ピーク時には，1764 個のワーカーが 30GB/s で処理を進めている．

MapReduce の特徴は以下の 3 点である．

- 並列分散システムに慣れていないプログラマーでも，モデルが使いやすく，並列化，耐故障性，負荷分散などを隠蔽できる．



Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004より引用.

図 8: MapReduce による Grep の性能評価

- 多くの問題が MapReduce の計算として表現しやすい.
- 何千台ものマシンからなる巨大なクラスターにスケールするように MapReduce を実装したので，Google 社が出会った多くの巨大な計算問題に対して効率的で適合した.

3.4 オープソース実装: HADOOP

HADOOP は，Google の MapReduce，GFS，BigTable のオープソースの実装であり，Yahoo などで実際に使われている.

4 具体例：Amazon Dynamo

4.1 概要

Amazon の Dynamo[8] は，Amazon の様々なアプリケーションのプラットフォームとなっている大規模なキーバリューストアである．例えば，ショッピングカート，ベストセラーリスト、顧客のプリファレンス管理，セッション管理，などで使われており，高い信頼性，スケーラビリティ，及び可用性を保っている．一般的なキーバリューストアでは，任意の形のデータを 1 度書き込んで，なんども読み込む（Read）だけ形のデータストアが得意である．しかし，Dynamo の特徴は，ユーザからの Write の要求をなるべくリジェクトしないように実現されていることである．Dynamo は

分散システム分野の基本技術である，consitent hashing, 投票に基づく一貫性管理，gossip に基づく故障発見，メンバーシッププロトコルなどを採用している．

4.2 背景：RDBMS は使わない

Amazon では，推薦機能，受注機能から，不正発見まで，何百ものさまざまなサービスを動かすために，何万台ものサーバを世界に向けて稼働させている．古典的には，このようなプロダクションシステムはリレーショナルデータベース（RDBMS）を用いて，状態の管理などを行ってきた．しかし，扱うデータのほとんどは，主キーとその値という（キーバリュー）単純な形式であり，古典的な RDBMS が提供するような複雑なオペレーションは必要としていなかった．さらに，古典的な RDBMS では複雑なオペレーションを可能にするために極めて高価なハードウェアと，高い技術を持った技術者が必要であった．リレーショナルデータベースが提供するソルーションは，理想からほど遠いのである．RDBMS の分野でも多くの発展があるが，データベースをスケールアウトし，効果的に負荷分散することは，いまだに簡単ではない．文献 [8] ではさらに以下のように述べている．「Amazon での経験から，ACID を保証するデータストアは，プアな可用性を提供しがちであり，このことは，古くから産業界からも学術界でも了解されていることである（文献 [10]）.」

4.3 99.9 パーセンタイルにおける SLA の実現

Amazon Dynamo の使用は，Service Level Agreement（SLA）に基づくものである．SLA は，顧客との間で形式的に交わされた契約であり，サービスを提供するシステム関連のいくつかの特徴に関して合意を得たものである．最も良く行われるのは，ある API に対して顧客からどの程度の要求が送付されるかという分散，と，それに対するサービスの遅延の期待値に対して具体的に合意を得るのである．例えば，「クライアントの負荷がピーク時，毎秒 500 リクエストの場合に，そのリクエストの 99.9%は 300 m s 以内で反応を返す」というものが想定できる（当時の Amazon での実際の SLA のようである）.

一般にこういったパフォーマンスに基づく SLA を得る時，既存の企業では，平均，中間値，期待値，などが用いられてきた．しかし，Amazon では，大半のカスタマーというだけではなく，「すべての」カスタマーによりよい体験（experience）をしてもらうために，９９．９パーセンタイルに基づく指標値によって SLA を定義している（99.9 パーセンタイルとは，99.9 番目の値を指標とする，という意味である．例えば，１０００件であればその上位から数えて９９９番目.）.

4.4 データの複製手法について

複製（Replicated）DB の先駆的研究が指摘しているように，ネットワークの故障の可能性を考えた時，強い一貫性と高度な可用性は同時に得られない [1]．すなわち，

表 2: Dynamo で使われている技術のまとめ

Problem	**Technique**	**Advantage**
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information

サーバやネットワークの故障を前提として可用性を高めるには，楽観的な複製技術を使うことができる．楽観的な複製技術では，更新（change）は，バックグランドで並行的に伝搬していくように進む．ここでは，更新が矛盾（conflict）したのを，いつ（when）、だれが（who）解決するかが重要となる．When に関して，既存の研究 [14] では多くが，読み込みの時ではなく，書き込みの時に矛盾を解決する．一方 Dynamo では，ユーザが「必ず書き込める（always writable）」を実現するために，書き込みの時よりは，読み込みの時に矛盾を解消する．誰が（who）解決するか（「どのように解決するか」と言っても良いと思われる）は,「最後の書込みが勝利する」（文献 [21]）という原則を採用し，アプリケーションがその原則を調整することもできる仕組みである．

4.5 Dynamo の要求事項

表 2 に，Amazon が直面した問題と，それを解決するための Dynamo が使っている要素技術を示す．また，Dynamo の主な要求事項を以下に上げる。

- 「いつも書き込める（always writable）」にする．

- すべてのノードは信頼できるものとする．Dynamo を直接外部から使うことは想定していない．
- 階層型の名前空間（既存のファイルシステムのような名前空間）や複雑な関係スキーマ（既存の RDB）を使うことは想定しない．
- 99.9%の書込みと読込みの操作は，2-300 ミリ秒以内で応答する．

4.6 Dynamo のアーキテクチャ

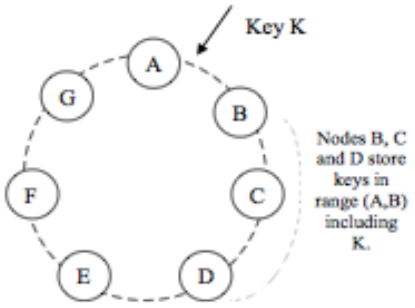
インターフェース

Dynamo はキーバリューストア（KVS:Key Value Store）であるため，get(key) と put(key, context, object) の 2 種類の操作をサポートする．get(key) では，key に関連する値 object (value) を返す．put(key,context,object) では，context に基づいて，key と object をストアする．Context とは，object に関する付加情報である．

分割アルゴリズム:Partitioning Algorithm

漸次的な（インクリメンタルな）スケーラビリティを確保しながら，複数のサーバ（ノード）に動的にデータを分割するために，Dynamo では consistent hashing が使われている．Consistent hashing[15] では，ハッシュ関数の出力であるハッシュ値の範囲がある 1 つの「リング」上にマッピングされる．各ノード（サーバ）は，ランダムに（ハッシュ）値が割り振られ，それがリング上の位置になる．データ要素は，そのキーから得られたハッシュ値が表すリング上の位置が与えられる．あるデータ要素をストアするノードは，そのデータ要素の位置から時計回りに回って最初に出逢うノードである．一般には，ノード（サーバ）の故障や追加があっても，影響があるのはその近隣のノードのみとなる点が特徴である．Dynamo ではさらに，各ノードにデータが適度に分散されるように，各ノードは仮想の位置を複数持つ．すなわち，1 つのノードがリング状に複数の位置を持つことになる．これによって，3 つの長所が得られる．図 9 に例を示す．

- あるノードが故障した場合，そのノードが担当していたデータは，いくつかのノードに分散される（オリジナルの consistent hashing だと，時計回りで後ろの 1 つのノードに集中してしまう）．
- 新しいノードが追加された場合，そのノードは，他の複数のノードからそれぞれ同等の負荷を軽減できる（オリジナルの consistent hashing だと，時計回りで後ろの 1 つのノードのみ負荷が軽減する）．
- 仮想ノードの数は，実際のインフラの状況などや容量などに基づいて調整することが可能である．



DeCandia, G., D. Hastorun, et al. (2007). Dynamo: amazon's highly available key-value store. SIGOPS Oper. Syst. Rev., ACM. 41: 205-220.より引用.

図 9: ConsistentHashing の例

複製（レプリケーション）

可用性と耐久性を得るために，Dynamo では，データをノード間で複製する．パラメータ N は，Dynamo の中での複製数だとする．具体的には，N=3 だとすると上記のリングにおいて，あるノードは自分が担当するデータを，時計回りで後ろの２つ（N-1）にも複製し担当させる．

あるキーに対して責任を持つノードのリストを，そのキーに対する preference list と呼ぶ．

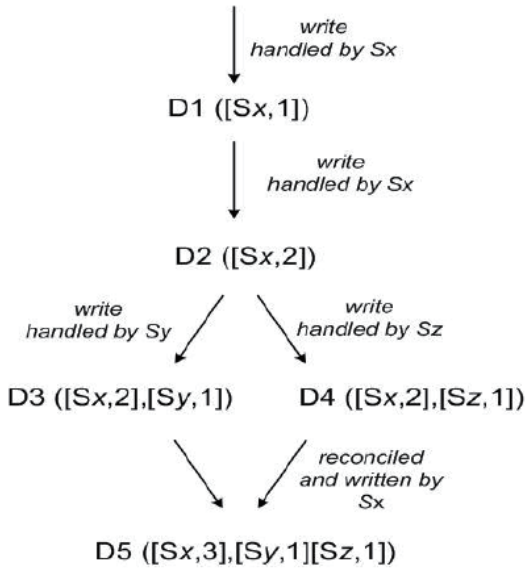
データバージョニング

Dynamo は結果一貫性を保持するという立場なので，更新が非同期に伝搬することを許している．ネットワークや計算機の故障があることを前提にした場合，更新がすべてに伝わることは保証できないが，そのような状況でも，動作することができるように構築できるアプリケーションがある．例えば，ショッピングカートである．

ショッピングカートでは「Add to Cart」などのアクションは忘れられてはならない．そこで，ユーザのアクションはすべて新規かつ不変のデータとして扱われる．すると，複数のバージョンのショッピングカートが存在し得る．システムはその後．複数のバージョンを一致（reconcile）する必要がある．そこで，まず，dynamo 上のアプリケーションを構築する場合は，データにいくつかのバージョンが存在し得ることを明示的に理解する必要がある．

Dynamo では，複数のバージョンを一致させるための方策として，vector clock[16] を使っている．Vector clock は，（ノード，カウンター）というペアのリストであって，すべてのデータのすべてのバージョンに関連する．

図 10 に Vector クロックの例を示す．あるクライアントが新しいオブジェクトを書き込んだとする．この書込みをノード（サーバ）Sx が処理したとすると，時刻を表すカウントを 1 つ進める．現在，システムはオブジェクト D1 とそのクロック [(Sx,1)] を持つ．クライアントがそのオブジェクトになんらかの更新を加えたとする．この更新をノード Sx がさらに処理し，時刻も 1 つ進むので，システムはオブジェクト D2



DeCandia, G., D. Hastorun, et al. (2007). Dynamo: amazon's highly available key-value store. SIGOPS Oper. Syst. Rev., ACM. 41: 205-220.より引用.

図 10: Version 管理の例

とそのクロック [(Sx,2)] を持つ．D2 は D1 を継承していて，D1 の他のレプリカも存在している可能性もある．同じクライアントが再度そのオブジェクトに更新を加えたとし，ノード Sy が処理したとする．システムは D3 とそのクロック [(Sx,2),(Sy,1)] を持つ事になる．一方で，異なるクライアントがそのオブジェクトを読み込み更新を加えようとしたとする．そこで，異なるノード Sz がそれを処理したとする．システムは D4 をもち，そのクロック [(Sx,2),(Sz,1)] を持つ事になる．ここで、あるクライアントが D3 と D4 を読込んだとする（両方のオブジェクトが，読込みによって見つかるものとする）．この読込みの文脈は，[(Sx,2),(Sy,1),(Sz,1)] となる．このクライアントが書込みを行うと，Sx が処理したとすると，システムは D5 を持ち，クロックは [(Sx,3),(Sy,1),(Sz,1)] となる．

Get() と Put() の実行における Quorum(定足数) メカニズム

前に記述した通り，Dynamo への書込みと読込みは，Get(key) と Put(key,context,object) 命令によって実現されているが，各 key に対する preference list にある（時計回りで）上から N 個のヘルシーなノードによって処理される．ヘルシーなノードというのは，故障などが起こっていないノードを意味する．ここでは，Quorum(定足数) メカニズムが使われている．R は，読込み命令が成功するための最小数．W は，書込み命令が成功するための最小数を表す．ここで，$R + W\ N$ であり，一般に R と W はそれぞれ，N より小さく設定される．

故障の扱い：Hinted Handoff

例えば，ノード A が一時的に故障した場合，ノード A にあるはずだったデータのレプリカは，ノード D に送付される．D に送られたレプリカは，そのメタデータの中にヒントを持っており，ヒントにはこのレプリカがどのノードに受け取られるべきかが示される．このヒントを持つレプリカを受け取ったノードは，ローカルなデータベースにそのレプリカを保持し，定期的にスキャンする．そして，ノード A が復旧したことがわかったら，ノード D はレプリカをノード A に送る．送信が成功するとノード D はレプリカを消す．このヒントを持つレプリカ（Hinted Handoff）によって，前出の Quorum プロトコルは，ノードの一時的な故障によっては失敗しない．

また，Dynamo では，データセンターをまたいでレプリカを保持し，データセンターをまたいだ preference list を持つことで，高い可用性と耐故障性を保っている．

恒久的な故障の扱い：Handling Permanent Failure

Hinted Handoff は，一時的な故障については対応できるが，ヒントを持つレプリカが戻ってこない場合などは，対処できない．そのような場合は Merkle 木 [17] によるハッシュツリーを使って，レプリカをシンクロナイズする方法を提供している．

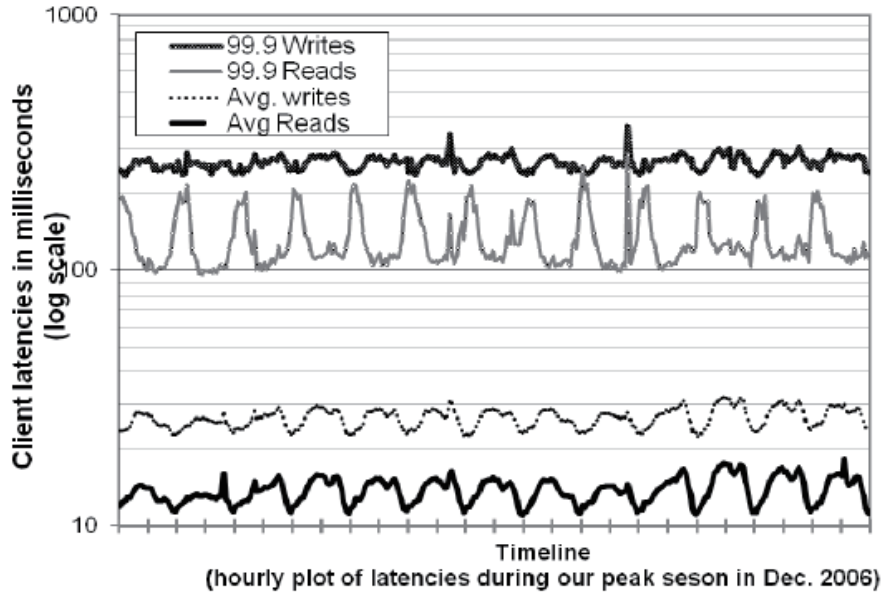
メンバーシップと故障発見

Dynamo では，ノードの停止・故障は一時的なものであるが，たまに長くかかるものもあるが，恒久的な故障，再バランス調整，再分割処理などに発展するべきではない．そこで，Dynamo リングからノードを追加・削除するための明示的なメカニズムが提供されている．管理者はコマンドラインツールやウェブブラウザを使って，Dynamo ノードをリングに付加したり削除したりすることができる．そして，メンバーシップに変更があったことを通知し，履歴を残す．メンバーシップの情報は，gossip ベースのプロトコルで，結果一貫性を保ちながら，伝搬される．(DNS のような感じ)．

通信の試みの失敗を回避するためには，故障発見の純粋にローカルな概念で十分である：すなわち，もし，ノード B がノード A のメッセージに反応しなかったら，ノード A はノード B が故障していると考える（それがたとえ，ノード B がノード C のメッセージに応答がある場合もである）．Dynamo のリングでは，安定した割合の通信があるため，ノード A はノード B がメッセージに反応できないとき，すぐさまノード B が反応しないということを発見できる．ノード A は，B にきた要求に、サービスを返すために他のノードを使う．A は B に定期的に問い合わせ B が復旧したかをチェックする．

チューニング

実装はすべて，Java でおこなわれている．Dynamo は様々な応用ごとにチューニングしたインスタンスを作ることが可能である．例えば，Dynamo は，N，R，及び W という値を調整することで，性能，可用性，耐久性に関して，チューニングすることができる．例えば，N は耐久性に関連する．Dynamo では N=3 が使われている．



DeCandia, G., D. Hastorun, et al. (2007). Dynamo: amazon's highly available key-value store. SIGOPS Oper. Syst. Rev., ACM. 41: 205-220.より引用.

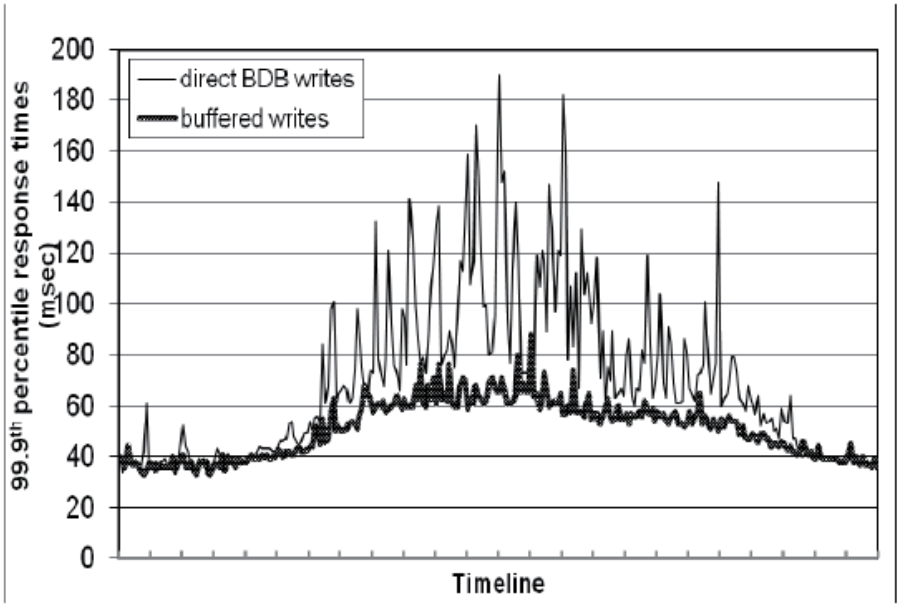
図 11: クライアントの遅延の 99.9 パーセンタイルと平均

W と R は，可用性，耐久性，および一貫性に関係する．一般的な設定は（N,R,W）=(3,2,2) である。

4.7 Dynamo の性能評価

図 11 は，Amazon が 1 年で最も忙しい時期１２月（２００６年）の３０日間の，Dynamo の読込みと書込みの遅延の平均と 99.9 パーセンタイルである．図の通り日周のパターンがあらわれている．さらに，書込みの遅延は，読込みの遅延よりも大きい，なぜなら，書込み操作は耐久性確保のために必ずディスクアクセスを伴なうためであるからである．また 99.9 パーセンタイルの遅延は 200ms 程度で，平均より遅い．これは，99.9 パーセンタイルの遅延は，リクエストの負荷は様々なものがあり，オブジェクトのサイズなど様々な要因が影響するためである．

Dynamo ではさらなる最適化として，書込み処理において，バッファを使って，バッファに書き込み処理を貯めておいてから，write thread により定期的に書込みが行われる．図 12 に示す通り反応時間は，バッファありのバージョンのほうが 5 倍ほど向上されている．一方，バッファによる弊害は，サーバが故障した場合に，バッファに溜まっていた書き込み処理も失われ，耐久性（「一度行われたトランザクションの結果が影響に失われてはならない」）が失われる．これは，耐久性と性能のトレードオフである．そこで, Dynamo では，書き込みにおいて，coordinator が書き込む先のノードを選ぶ時に，N 個の中から 1 つのノードを選び，そのノードは「耐久性のある書き込み」を行う（耐久性のある書き込みとは，バッファリングしないことだ



DeCandia, G., D. Hastorun, et al. (2007). Dynamo: amazon's highly available key-value store. SIGOPS Oper. Syst. Rev., ACM. 41: 205-220.より引用.

図 12: 24 時間でのバッファあり書込みとバッファなし書込みの比較

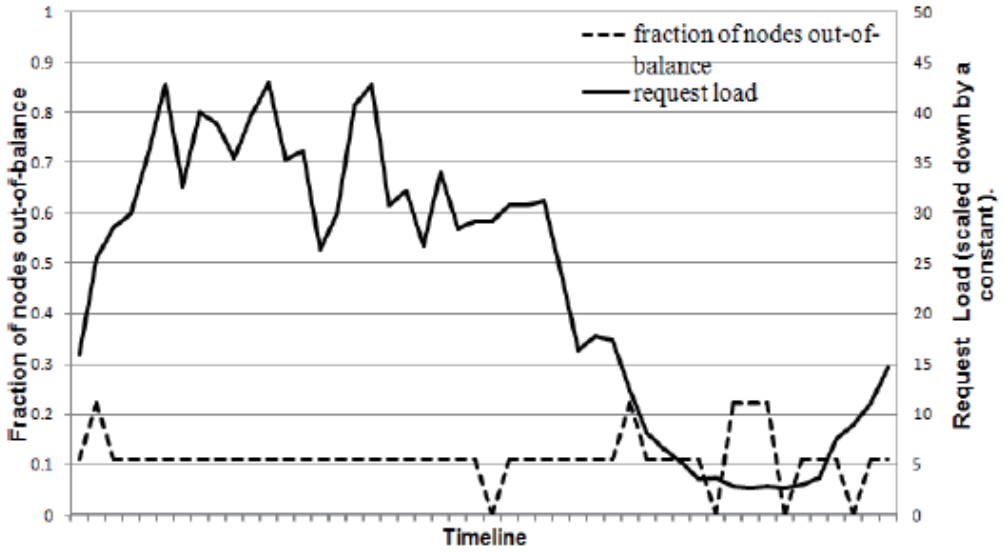
と思われる）．Coordinator は，書き込みの時には W 個の返答のみを受け付けるので，この方式は性能に影響しない．

図 13 は，リクエストの数に対して，負荷が不均衡になったと思われるノードの割合を，２４時間を３０分毎に示した図である．ここで，あるノードに関して平均的な負荷から１５％以上負荷が偏った場合，負荷が不均衡になったという．図に示すとおり，Dynamo では負荷が増えるほど不均衡は減る．また，負荷が高い場合は，不均衡の割合は１０％程度，負荷が低い場合は不均衡の割合は大きくて２０％である．これは，負荷が高い場合は，まんべんなくキーに対するアクセスが発生するため負荷の不均衡は減るが，負荷が低い場合は人気のあるキーへのアクセスの偏りが発生し，負荷の不均衡が増えるためである．

図 14 に，consistent hashing における，パーティション戦略（partition strategy）を３つ示す．ここでトークンとは，サーバ（ノード）の仮想的な場所を示す．Dynamo では，１つのサーバはいくつかの仮想的な場所を持っている．１つ目（strategy 1）は，ランダムな数のトークンを置き，各パーティションの幅を各トークンの値で決める（初期の方式）．２つ目（strategy 2）は，ランダムな数のトークンを，同じ幅のパーティションに配置する．３つ目（strategy 3）は，Q／S 個のトークンを，同じ幅のパーティションに配置する．Q はパーティションの数，S はノード（サーバ）の数である．図 15 に示すように３つ目の戦略が最も負荷分散効率が良い．

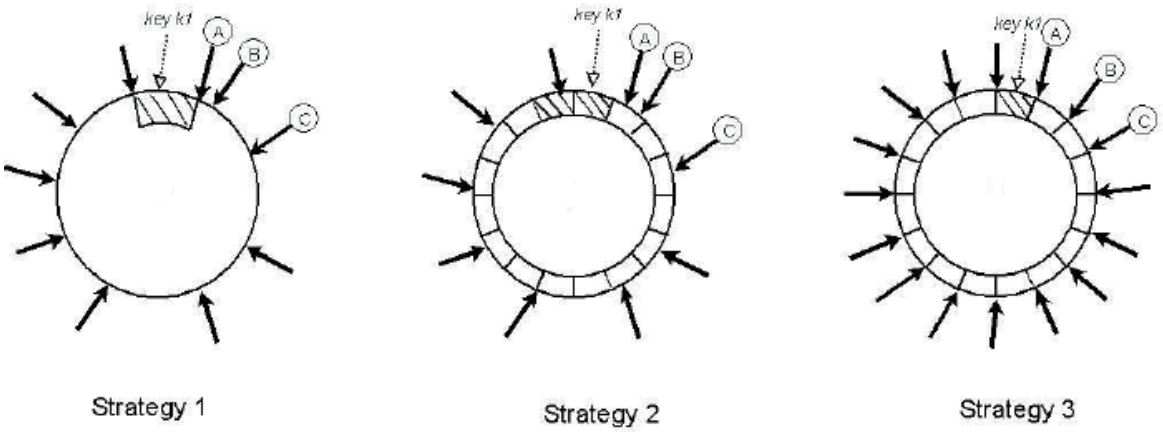
バージョンの不一致

Dynamo は，一貫性を犠牲にして可用性を高めている．ここでは，一貫性を犠牲にしたことにより生ずる，あるデータの不一致バージョンの数について実験結果を



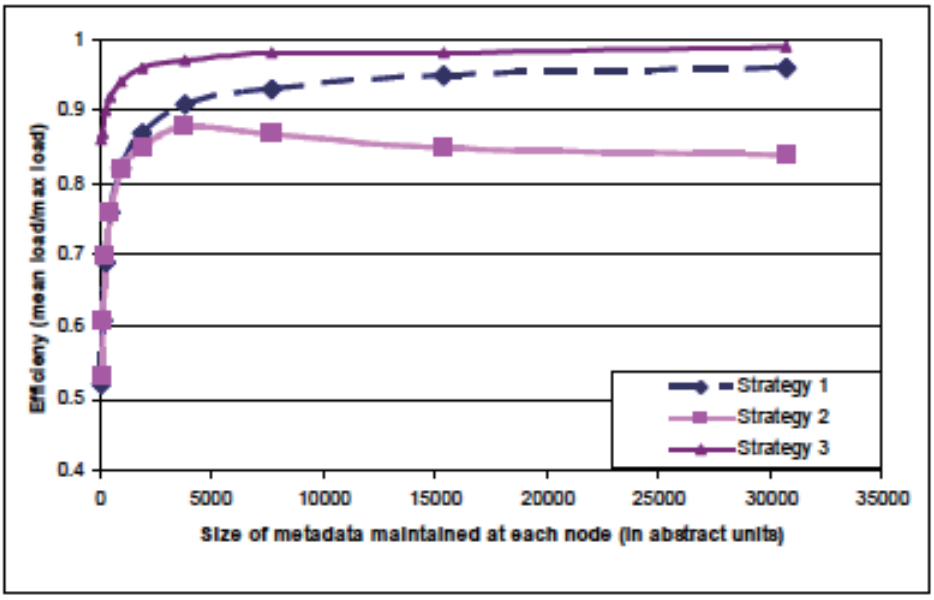
DeCandia, G., D. Hastorun, et al. (2007). Dynamo: amazon's highly available key-value store. SIGOPS Oper. Syst. Rev., ACM. 41: 205-220.より引用.

図 13: 負荷の不均衡に関する実験結果



DeCandia, G., D. Hastorun, et al. (2007). Dynamo: amazon's highly available key-value store. SIGOPS Oper. Syst. Rev., ACM. 41: 205-220.より引用.

図 14: Consistent Hashing における分割戦略



DeCandia, G., D. Hastorun, et al. (2007). Dynamo: amazon's highly available key-value store. SIGOPS Oper. Syst. Rev., ACM. 41: 205-220.より引用.

図 15: パーティション戦略の負荷分散効率（load balancing efficiency）の比較

示す．バージョンの不一致が生じるのは，二つ理由がある．１つは，システムに何らかの障害，故障が起きたとき．もう一つは，ある１つのデータに多数の並行的な書き込みがあり，複数のノードが並行にそれを処理したとき，である．

実験では，ショッピングカートサービスにおいて，ある２４時間でのバージョンの不一致の数をカウントした．全リクエストのうち，99.94%は１つのバージョンしかなかった（一貫性が保たれている）．0.00057%が２つのバージョンを持っていた．0.00047%が３つのバージョン，0.00009%が４つのバージョンを持っていた．以上より，バージョンの不一致が発生すること自体が少ないことがわかる．

また，経験的に，バージョンの不一致の原因は，システムの障害や故障よりも，並行的な書き込みの数の増加によることが多い．並行的な書き込みは，人間によるものよりも，ロボットによるもののほうが多いようである．

4.8　Dynamo のオープソース実装

Twitter などで使われている Apache Cassandra は，Dynamo の分散システムとしての設計をベースに実装されている．

5　利用者側からのクラウドコンピューティング

本章は，クラウドコンピューティングの一般的な説明で，読み飛ばしていただいても構わない．

5.1 利用形態による違い

クラウドコンピューティングによって，計算機を所有から利用することになる．サービスの提供の方法として，以下の３つに分類される．しかし，この分類は，クラウドコンピューティングだからというよりは，その前からデータセンター業務における，サーバハウジング，サーバホスティング，レンタルサーバ，ASP（アプリケーションサービスプロバイダ），など，古くからあった概念を，クラウド技術を使った場合に，まとめ直したものであると考えられる．

- SaaS（Software as a Service）：カスタマイズした，もしくは，消費者向けソフトウェアをサービスとして提供する．
- PaaS（Platform as a Service）：ツール，DB, インターフェースなどをサービスとして提供する．
- IaaS（Infrastructure as a Service）：主に仮想化したサーバ機能を提供する．

5.2 ユーティリティコンピューティング

クラウドコンピューティングという言葉が，Google によって提唱される前から，ユーティリティコンピューティングという言葉があった．ユーティリティコンピューティングは，コンピューティング資源が，電力，水道，電話回線のように，誰もが使えるコモディティとして使えるような状況を言っている．ユーティリティコンピューティングでは，クラウドコンピューティングでいう一貫性を緩めた分散データベースような議論はしない．クラウドコンピューティングは，ユーティリティコンピューティングという理想を，一貫性を緩めた分散システムという技術的要素によって，実現したものであるとも言える．ニコラス・カーはその著書 [4] の中で，現在の情報ネットワークの発展を，過去の電力網の発展に似ているものとして，エジソンの General Electronics 社の発展を比較し紹介している．すなわち，個々に自家発電機を所有していた時代から，巨大な火力発電施設を共有し利用する仕組みになった歴史は，現在の計算機の所有から利用への変化に大変似ている．当初も本当に安定して電力が供給されるのかという信頼性に対する心配は大きかった．

また，インターネットの聡明期には，それまでは電話の全２重通信で通信路固定あったのが，パケットによる通信経路の非固定化が行われるようになった．パケットがきちんと相手に届くのかという意味で通信に対する信頼性が大変疑われたが，現在は電話（携帯電話）とインターネットは，相互補完的なものとして発展している．

5.3 プライベートクラウド

「プライベートクラウド」は，クラウドコンピューティングとは異なる言葉である．大規模な可用性がクラウドコンピューティングの特徴だとすれば，企業内で，エキサバイト級のデータセンターを，可用性を最大化しながら運用する必要性が現状

では少ないように思われる．むしろ，企業内での様々なプライバシを保ったまま，クラウドコンピューティングの技術的特徴を利用したサーバシステムである．

6　クリニカルデータ応用への展望

医療データはこれまで，プライバシーの問題などから，広域的に共有されることが少なかった．一方で近年様々な医療データが大量に得られる仕組みが構築されつつある．今後，膨大な医療データを有効に活用するためには，クラウドコンピューティングの技術を用いることで，効率的にデータを蓄積することが重要である．

多くのクリニカルデータは，あるデータオブジェクトが頻繁に書き換えられるような性質を持っていない．すなわち，古典的なRDBMSのモデルは必ずしもベターな選択ではない．ある時刻に発生したクリニカルデータは，一度書き留めておけば，あとはそのデータを用いた解析や病気の予防や推測のために読み出すのみである．分散データベース上で瞬間的に不一致があったとしても，結果的に整合性を保つことができるなら，ほとんど問題はないと言える．これはまさにクラウドコンピューティング技術を用いるべき応用分野である．

法律の問題はあるものの，今後の医療は，医療行為や介護行為を行うと同時にすべてのデータが蓄えられ，その後の病気の発生予測や予防に役立てることができる．このようなデータにはクラウドコンピューティング技術が適している．

7　環境やグリーンITとの関連

クラウドコンピューティング技術のもう一つの側面として，大量のサーバを大人数で共有するという面がある．直感的にはこれによって，多くの企業や個人が自分でサーバを稼働させた場合と比較して，大幅にサーバのアイドル時間を減らすことが可能になる．

計算機のアイドル時間を使って計算を行うというアイデアは，グリッドコンピューティングによって確立されている．もっとも有名なプロジェクトの一つとして，SETI@Homeがある．宇宙から地球に到達した大量の電波のデータを，インターネット上に接続された家庭用のコンピュータのアイドル時間を使って解析するというプロジェクトである．

また，Googleなどのクラウドコンピューティングの先駆的な企業では，大量の計算機をいかにして低コストで稼働させるかという問題に関して，さまざまなシミュレーションを用いて研究している．例えば，文献[9]では，大規模なデータセンターにおける消費電力量の予測に関する方式に関してシミュレーション結果をもとに議論している．

8 まとめ

本レポートでは，クラウドコンピューティングの技術的側面を概観した．本レポートで紹介した通り，ACM 関係の国際会議や雑誌上で，並列 RDBMS とクラウドコンピューティング技術の論争が展開されているが，クラウドコンピューティング技術は，これまでの並列 RDBMS や RDBMS とは，異なる目的のために，実用上のニーズに基づいて発展してきている．すなわち，強い一貫性を保持するのではなく，結果的な一貫性を保持するという方針で，既存の並列 RDBMS や RDBMS ではなし得なかった，極めて大規模なデータを管理保持し，高い可用性を実現することが可能になっているのである．

もちろん，結果的な一貫性の保持という観点からは，Amazon の Dynamo での議論がある通り，あるデータに関して複数のバージョンがシステム内に存在する瞬間もあり得る．しかし，Amazon の Dynamo のように，多くの技術的な工夫によって，結果的な一貫性に起因する問題点は，実用上無視できるほど小さくなっていることが,「現実的に」示されている．Amazon のショッピングカートや Internet の基本的な仕組みである DNS(Domain Name Service）などがそれにあたる．

並列 RDBMS や RDBMS の長所と，クラウドコンピューティング技術の長所は，相反するものではなく，相補的なものである．どちらが勝っているとかどちらに欠点があるという議論は，クラウドコンピューティング技術の今後の大きな可能性から考えれば，取るに足りないものである．

参考文献

[1] Philip A. Bernstein and Nathan Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Trans. Database Syst.*, 9(4):596–615, 1984.

[2] Eric A. Brewer. Towards robust distributed systems (invited talk), 2000.

[3] Mike Burrows. The chubby lock service for loosely-coupled distributed systems, 2006.

[4] Nicholas Carr. *The Big Switch: Rewiring The World, From Edison To Google*. W. W. Norton & Company, Inc., New York, 2008.

[5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2), 2008.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.

[8] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store, 2007.

[9] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer, 2007.

[10] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services, 1997.

[11] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[12] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[13] Jim Gray. The transaction concept: virtues and limitations (invited paper), 1981.

[14] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution, 1996.

[15] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web, 1997.

[16] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[17] Ralph C. Merkle. A digital signature based on a conventional encryption function, 1988.

[18] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis, 2009.

[19] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz Andre Barroso. Failure trends in a large disk drive population, 2007.

[20] Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. Mapreduce and parallel dbmss: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.

[21] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, 1979.

[22] Werner Vogels. Eventually consistent. *Queue*, 6(6):14–19, 2008.