修　　士　　論　　文

非集約型ストレージアーキテクチャに於ける問合せ処理方式に関する研究

東京大学大学院 情報理工学系研究科
電子情報学専攻

48-206463 加藤 滉貴

指導教員　豊田 正史 教授

2022年1月27日

本論文は東京大学大学院情報理工学系研究科に
修士号授与の要件として提出した修士論文である.

概 要

近年，高速ネットワークによって結合された計算機上で二次記憶を分散して管理することを特徴とする「Disaggregated storage architecture」（非集約型ストレージアーキテクチャ）が注目されている．当該ストレージアーキテクチャ上では，データが自ノードのストレージに格納されているか，他ノードのストレージに格納されているかによって，アクセスコストが異なり，高性能なデータベースシステムを構築するためには，サーバとデータのアフィニティ問題を解決する必要がある．本研究では，当該ストレージアーキテクチャに基づいて構成された並列データベースを対象として，問合せ処理の効率を向上するための，アフィニティを考慮したデータと演算の配置方法，IO の発行方法を検討し，実装を用いた実験によりその有効性を示した．

実験の結果，リモートサーバに接続されたストレージへのリード（リモート IO）において最大で 4.1M IOPS を達成した．これはローカルストレージへのリード（ローカル IO）性能の 85%に相当し，任意のストレージへの高速なアクセスを実現したといえる．また，クエリの模擬実行を用いた実験を行い，IO と演算が共存する条件下でのクエリ処理の性能特性を明らかにした．この結果，ページサイズが 128KiB のときは，演算を任意のサーバに配置しても性能は凡そ不変であるという，安定性を確認することができた．

目 次

図 目 次

表 目 次

第1章　序論

1.1　背景

近年，データベースが取り扱うデータ量は飛躍的に増大している [1,2]．扱うべきデータの増大量に対して，データベースシステムの性能向上速度が追いついていないのが現状である．データベースは銀行口座や EC サイト，航空券予約システムなど身の回りのあらゆるところで使用されており，高速にデータベースへのクエリが処理されることは，ユーザやシステムにとって必要不可欠である．

そもそもデータベースのワークロードは，Online Analytical Processing（OLAP）と Online Transaction Processing（OLTP）に大別される [3]．OLAP とは，大量のデータを一度に複雑に処理するようなワークロードで，例えばビッグデータの分析作業などが挙げられる．OLTP とは，大量のスモールセットの小さいサイズのデータを高速に処理するようなワークロードで，例えばモバイル決済などが挙げられる．本研究では OLAP 系のワークロードを想定している．増大するビッグデータを，高速に処理出来るようにすることが本研究の方向性である．

データベースシステムの性能向上は，例えば単純にマシンをハードウェアの性能数値のより良いものに置換することで達成される．しかし，コンピュータのコンポーネントのうち例えば CPU はムーア則 [4] に則って性能向上してきたが，それも近年終焉を迎えるとの説も強い [5]．ネットワークやストレージなどの他のコンポーネントについては，そもそも CPU ほど劇的な性能向上は果たしていない．このように，マシンのハードウェア的スケールアップではその効用に限界が存在する．

また，単純にハードウェアをスペックアップするのではなくて，そもそも採用する技術スタックから変更する場合もある．例えば，近年の DRAM の高容量化，低コ

スト化に伴うインメモリデータベースシステムの登場である．近年の分散インメモリデータベースシステムに大きな貢献をもたらしたSilo [6] はその最たる例である．

一方で，データベースシステムは，そのネットワーク部についてはあまり改善が行われてこなかった．歴史的には，データベースシステムを構成する際，ネットワークに採用される技術としてはTCP/IPがデファクトスタンダードであった．インターネットにおいてもデファクトスタンダードの通信技術であるTCP/IPが，データベースシステムのようなデータセンタ内のノード間通信においても使用されることが多く，これは今尚数十年続いている傾向である．TCP/IPは幅広いアーキテクチャ上で信頼性の高い通信路を提供する一方で，パフォーマンスの観点からはデータベースシステムに対して十分に最適化されているとはいえず，しばしばネットワークがデータベースシステムにおいてボトルネックとなることが多い．実際に，ネットワークがボトルネックであるという認識のもと設計されているデータベースシステムも多い [7].

しかし，近年InfiniBandの採用などにより，データベースシステムの構成要素のパワーバランスは年々従来とは異なった様相を呈してきている．InfiniBandや100Gb Ethernetなど高速ネットワークの技術がコンバージドネットワーク [8] として採用され，データベースを構成する際に使用されることがある．例えば，コンバージドネットワークとして構成されたInfiniBandを採用したデータベースシステムにおいて，Remote Direct Memory Access (RDMA) と呼ばれる高スループット，低遅延なノード間通信を可能とする技術を導入しようという研究動向が存在する [9–25].

さて，大規模データを高速にデータベースシステムで処理するための手段として，ハードウェアのスペックアップや新しい技術スタックの採用とは，別の角度からのアプローチが存在する．それは，並列データベースシステムを構成して，負荷を複数のサーバで分散するという手法である．この並列データベースシステムのストレージアーキテクチャとして，Shared-nothing [26] とShared-storage [27] という2つがあった（2.1章参照）．しかし，前述のようにコンバージドネットワークの登場によって，ネットワークが高速化した結果，Disaggregated storage architecture（非集約型ストレージアーキテクチャ）という新しいストレージアーキテクチャ (図1.1参照)

が考案されている [28, 29].

Disaggregated storage architecture（以下，Disaggregated storage ）とは，Shared-nothing と Shared-storage の利点を併せ持つハイブリッドなアーキテクチャであるが，サーバに対してストレージの局所性が存在するため，潜在的には非常に高いパフォーマンスを持つ一方で，局所性を上手に利用しないとかえってパフォーマンスが落ちてしまうというアフィニティの問題が存在し，この問題は現在未解決である.

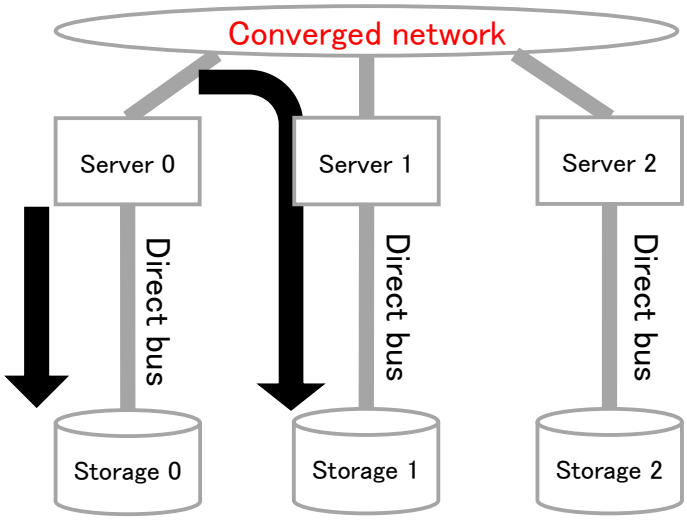

図 1.1: Disaggregated storage

1.2 本研究の目的と貢献

そこで，本研究ではコンバージドネットワークを用いた Disaggregated storage 型の並列データベースシステム上で，どのようにオペレータ，IO，データのアフィニティを調整したら最大性能を引き出せるかを検討した．これを行うために

- ローカル IO とリモート IO の性能を向上するための IO 発行方法を実験により明らかにした.

- その上で，SQL での where 句のようなオペレータをサーバに配置した上で，ローカル IO，リモート IO 発行方式を考案し性能特性を検証した．
- この際並列データベースシステムを構成する複数のサーバのうち，どのサーバにオペレータを配置するかというアフィニティの問題が存在したので，複数のクエリ処理の方式を検討し実験により効率的な方式を明らかにした．

1.3 本論文の構成

本論文の構成は以下の通りである．

2 章 並列データベースのストレージアーキテクチャ，RDMA，データベースに於けるページとレコードについて紹介する．

3 章 リモート IO やリモートクエリなど，Disaggregated storage 上での IO やクエリ処理の方法を検討する．

4 章 評価実験の測定方法，実験環境，実験結果などについて述べる．

5 章 関連研究を紹介し，本研究への応用を考察する．

6 章 本論文を総括する．

第2章　ストレージアーキテクチャと並列データベースシステム

本章ではまず，並列データベースのストレージアーキテクチャについて説明する．そこで Disaggregated storage（非集約型ストレージ）の課題について問題提議を行う．次に，本研究ではコンバージドネットワークとして InfiniBand を採用するが，その上で使用する通信プロトコルである RDMA について説明する．最後に，データベースのページという概念について説明する．

2.1　並列データベースのストレージアーキテクチャ

並列データベースをストレージアーキテクチャの視点から見ると，ストレージをサーバ間で共有するか否かという観点で，図 2.1 のように Shared-nothing， Shared-storage の 2 つのアーキテクチャがそれぞれ 1980 年代, 2000 年代から存在する [26,27]．

Shared-storage アーキテクチャでは，複数のサーバで一つのグローバルなストレージを共有しており，論理的にはシンプルである．一方で，サーバと共有ストレージ間を接続する Storage Area Network（SAN）などのネットワーク部がボトルネックになることもある．それ故にデータへのアクセス自由度は高いという利点はあるが，スケールアウトしにくいという欠点が存在する．元々は，Shared-disk アーキテクチャと呼ばれていたが，最近ではストレージにフラッシュメモリを採用することも多くなり，ディスクより一般的な用語であるストレージを使い，Shared-storage と呼ばれることも多い．

一方で，Shared-nothing アーキテクチャでは，各サーバはリモートサーバのスト

レージにはアクセスできない．それ故にデータを格納する際は，データを分割して各ストレージに保管する方法（パーティショニング）や，何らかの方法によりストレージ間で同期を取り全てのストレージにデータのコピーをもたせる方法などを取る．しかし，各サーバは専有のストレージを有するため，サーバの並列度を高めてもストレージへのアクセスがボトルネックになりにくく，スケールアウトしやすいアーキテクチャとなっている．

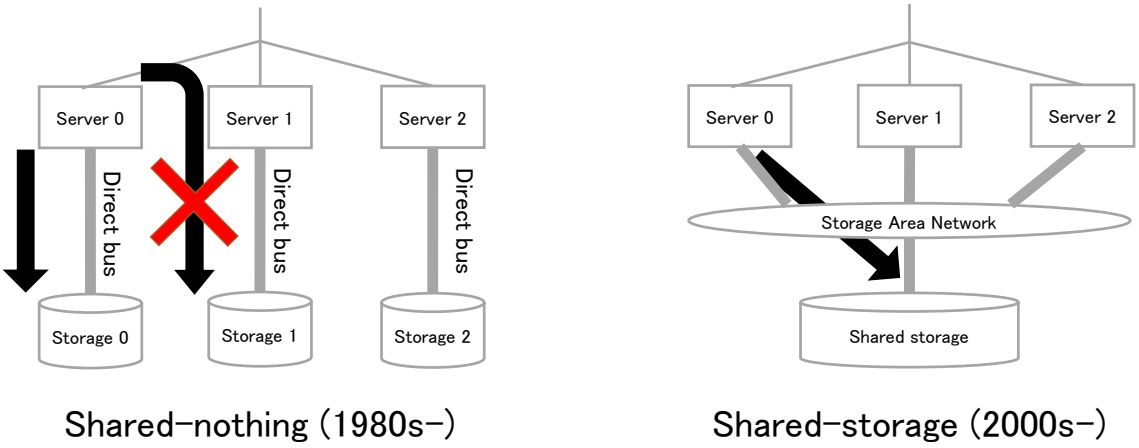

図 2.1: 既存のストレージアーキテクチャ

これらの伝統的なストレージアーキテクチャに対して，1 章で述べたように Disaggregated storage が存在する（図 2.2）．Disaggregated storage は，Shared-nothing と Shared-storage の特徴を併せ持つハイブリッドなアーキテクチャである．Shared-nothing のように専有のストレージを持ち，そのストレージへは高速でアクセスすることができる．また他サーバのストレージにもコンバージドネットワーク経由でアクセスできるため，Shared-storage のようにストレージ空間はグローバルである．このように Disaggregated storage は両者の利点を併有する．既存のストレージアーキテクチャと Disaggregated storage の比較表を表 2.1 に掲載する．

このように Disaggregated storage は多くの利点を持つが，サーバは自ストレージへと他ストレージでアクセスコストが異なる．自ストレージには Direct bus でアクセスし，他ストレージには，コンバージドネットワーク，リモートサーバ経由でよう

表 2.1: ストレージアーキテクチャの比較

	Shared-nothing	Shared-storage	Disaggregated storage
自ストレージへのアクセス	高速	低速	高速
他ストレージへのアクセス	不可	低速	やや高速
スケールアウト性	高	低	高

やく Direct bus に辿り着きストレージにアクセスするからだ．つまりサーバとストレージの間にアフィニティが存在する．そのため，潜在的には非常に高いパフォーマンスを持つストレージアーキテクチャではあるが，アフィニティを上手に調整しないとかえってパフォーマンスが落ちてしまうという問題が存在する．そしてこの問題は現在未解決である．そこで本研究では，他ストレージへの高速なアクセス方法であるリモート IO を提案し実証した．

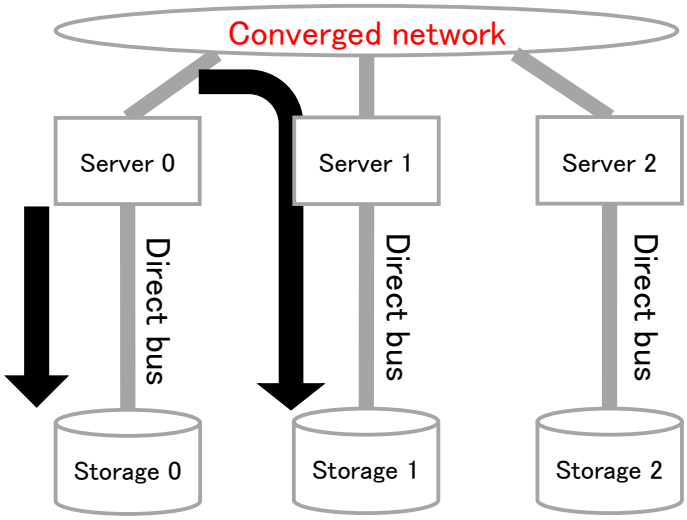

図 2.2: Disaggregated storage （再掲）

2.2 Remote Direct Memory Access (RDMA)

本研究ではコンバージドネットワークを構成する InfiniBand 上の通信プロトコルとして RDMA を採用する．Remote Direct Memory Access，通称 RDMA とはその名の通り，リモートサーバに存在するメモリ上に読み書きを行う技術である [12,19,25]．つまり，TCP/IP が提供するソケットインタフェースのようには抽象化されておらず，RDMA は通信先のメモリに書き込むところまでしか機能として提供していない．そこから先の，どのようにどのタイミングで通信先のアプリケーションがその書き換えられた内容を見るかといった処理は RDMA のユーザ側に任されている．

RDMA が低遅延な通信を提供できる理由として，ゼロコピー通信の寄与が大きい．TCP/IP では，通常ユーザメモリ空間上にあるデータは，OS のデータバッファに CPU を使用してコピーされた後に，リモートサーバへと送信される．しかし，ゼロコピー通信では，ユーザメモリ空間上にあるデータは CPU やカーネルの関与なしで，RDMA NIC (RNIC) によって直接リモートサーバへと送信されるので遅延が非常に少ない．

RDMA はそのインターフェースとして verbs API [30] を提供しており，ユーザは用途に合わせて使用プロトコルを選択することができる．verbs と呼ばれるプリミティブを RNIC が管理するキューに入れることで，制御が可能となる．この verbs には one-sided プリミティブ (e.g. READ，WRITE， ATOMIC) と two-sided プリミティブ (SEND，RECEIVE) の二種類がある．どちらも同じような機能 (i.e. 送信・受信) を提供しているが，その実装や性能には差がある．one-sided は一般により高いパフォーマンスであり，また，リモートサーバの CPU 関与すら必要としない．一方，two-sided はリモートサーバの CPU 関与は必要とするが，one-sided よりは通信を抽象化しており，シンプルで使いやすい．つまり，ただでさえ原始的なプロトコルである RDMA であるが，one-sided のほうがより原始的で低レイヤーのインターフェースであると言える．

またプリミティブという分類以外にも，IP における TCP，UDP のように RDMA にもサービスタイプという分類があり，Reliable な Connection を提供する RC や

表 2.2: RDMA のサービスタイプ．本研究では RC を採用する

	Reliable Connection（RC）	Unreliable Datagram（UD）
データロス検出	○	×
エラーリカバリ	自動で再送	ユーザが実装する必要あり
メッセージサイズ	仕様上最大 2^{31} Byte	最大 4096 Byte

Unreliable な Datagram を提供する UD などがある[1]．表 2.2 で RC と UD を比較しているが，RC のほうが機能が豊富である．本研究では実装上の簡単のため，RC を採用している．

2.3 データベースに於けるページとレコード

リレーショナルデータベースはデータを管理する際，ページという一定サイズの領域を基本単位とする．例えば有名なオープンソースのリレーショナルデータベースシステムである MySQL のページサイズはデフォルトで 16KiB である．ストレージとメモリ間の IO の際は，ページ単位でデータを転送する．つまりデフォルトの MySQL の場合，16KiB 単位で IO を発行することになる．

ページにはレコードが複数格納されている．レコードとはリレーショナルデータベースのあるテーブルにおける，一組の値の連なりのことである．ページを IO によってメモリ上に展開したら，ページをパースしてレコード単位に分解することでデータを論理的に読み込むことが出来る．

[1]Reliable Datagram や Unreliable Connection というサービスタイプも存在するが，実際に頻繁に使われるのは RC と UD である．

第3章　Disaggregated storage architectureに於ける効率的なクエリ処理方式

本章ではまず，RDMA を用いたリモート IO の方法を提案する．リモート IO に於いて，RDMA とローカル IO をブリッジングする方式やブリッジングの際のスレッド管理手法について論じる．次に Disaggregated storage（非集約型ストレージ）上でクエリ処理を行うための 3 つの枠組みを検討する．これは Disaggregated storage に存在するデータとオペレータのアフィニティの問題に対するものである．

3.1　RDMA を用いたリモート IO

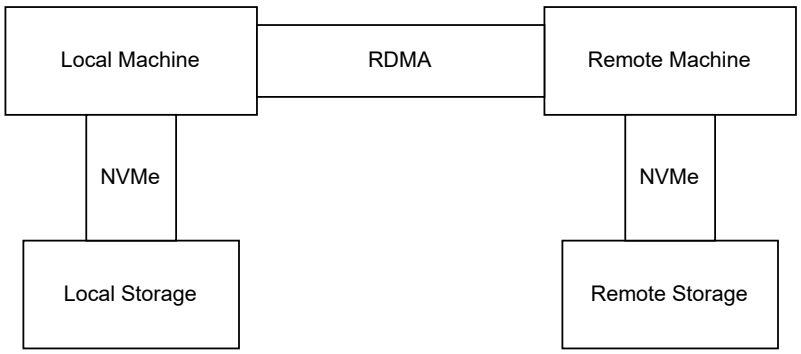

図 3.1: リモート IO で想定する並列データベースのアーキテクチャ

本研究で提案するリモート IO の概要を説明する．図 3.1 のように，2 台のサーバがあったときに，片方からもう一方のストレージにアクセスを行うことが本研究に

おけるリモート IO の想定である．図 1.1 のようにサーバが 3 台以上存在するときも，任意のサーバ間をネットワーク接続することにより，結局図 3.1 のような 2 サーバ間のリモート IO の状況に帰着させることができる．RDMA を介してリモート IO を行う方法であるが，リモートサーバに RDMA を介して IO 命令を発行し，リモートサーバ上のプロセスがその命令を受けてさらに自サーバのストレージ（つまり Remote Storage) に IO 命令を発行する．そして結果を受け取ったリモートサーバ上のプロセスは，往路と同じように RDMA を介して結果をローカルサーバに送り返すことで IO が実現される．

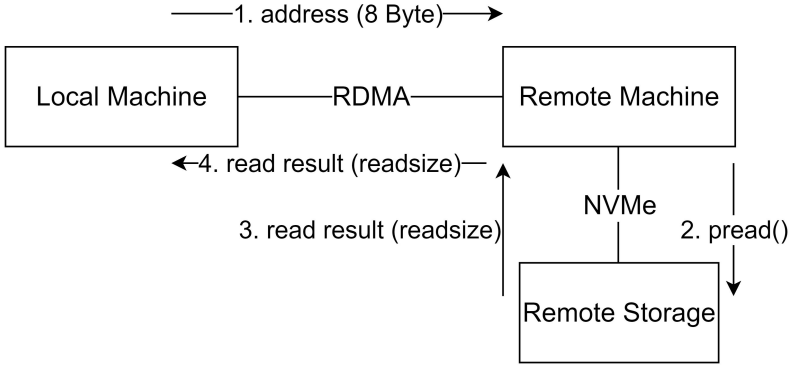

図 3.2: リモート IO のフロー

リモート IO における具体的なメッセージ内容とそのサイズは図 3.2 のとおりである．ローカルサーバにおいて，読み出したい Remote Storage のアドレスが決定したら，そのアドレス値を RDMA を用いてリモートサーバに送信する．アドレス値を受けとったリモートサーバのプログラムは，pread システムコールを用いて Remote Storage 上のデータを取得する．取得したデータはメモリ上に展開されるため，この領域をそのまま RDMA でローカルサーバに送信することで，ローカルサーバは Remote Storage のデータを取得できリモート IO が完了する．なおこの一連のフローは，RDMA で通信する際に張るコネクションやリモートサーバが用意する IO スレッドを多重化することにより，フロー自体も性能のため多重化する．

このように，リモート IO の系を考えるとその性能を最大化するためには，IO を

行うスレッド数，RDMA を用いて張るコネクション数など考慮すべき事柄が多くある．本研究の実験では，これらパラメータの様々な組み合わせに対して性能評価を行うことで，最適な RDMA の使用方法と最大性能を明らかにする．

3.2 リモートサーバの IO スレッド管理

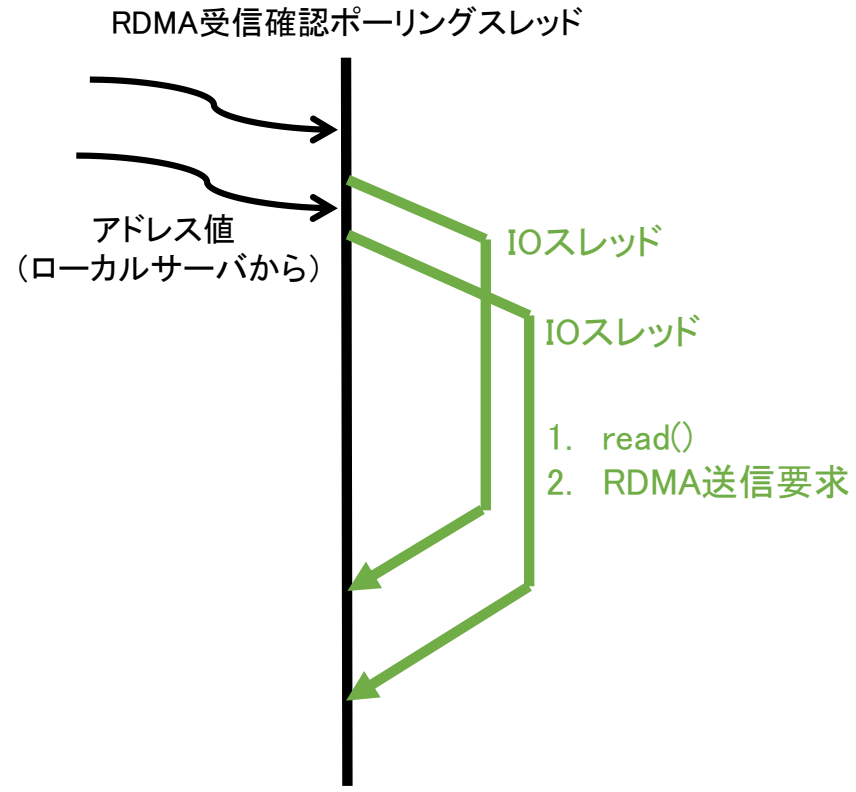

図 3.3: リモート IO の動的 IO スレッド管理手法

リモートサーバはリモート IO の際，RDMA により読むべきアドレス値を逐次受信している．RDMA の受信確認はポーリングにより行うが，そのポーリングを行う while ループの中で，IO を発行するとブロックされてしまう．それを防ぐため，IO スレッドは RDMA 受信確認ポーリングスレッドとは別に用意する．その方式は，動

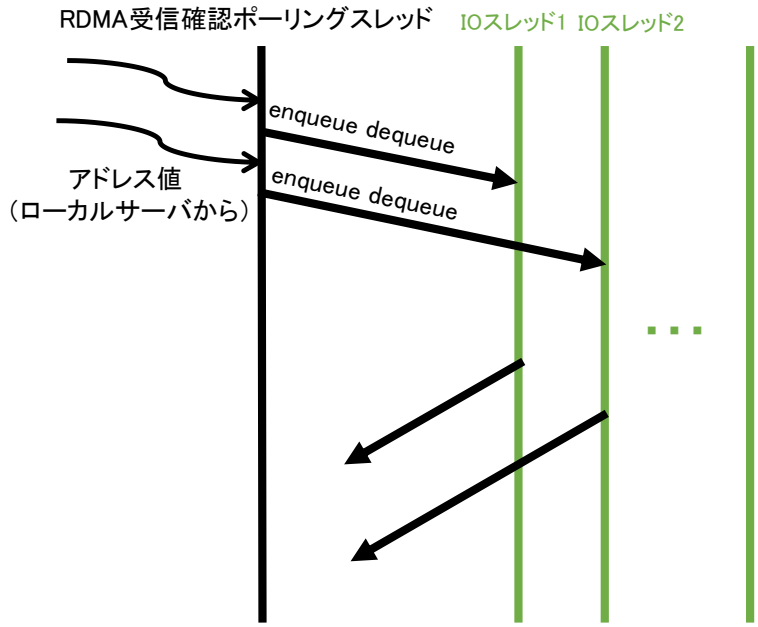

図 3.4: リモート IO の静的 IO スレッド管理手法

的にスレッドを生成するものと，静的にスレッドを用意しておくものの 2 つを考案した．

動的に IO スレッドを生成する方式においては，図 3.3 のように RDMA によりアドレス値を受信したら，pthread_create() により IO スレッドを新たに生成する．その IO スレッドの中で read() 等の IO 命令を発行する．

静的に IO スレッドを用意しておく方式では，図 3.4 のようにワークロードの開始前に予め IO スレッドを用意しておき，RDMA によりアドレス値を受信するたびに，シグナルを用いてスレッド間通信を行うことで，RDMA ポーリングスレッドは IO スレッドに IO 命令を委譲する．

3.3 Disaggregated storage 上でのクエリ処理

データベースでクエリ処理をする際は，IO を発行してページ単位でデータを取得した後，ページをパースしてレコードに分解する．レコードから必要なアトリビュー

トのデータを取得し，適宜演算を行う．

Disaggregated storage でクエリ処理するケースを考えると，データをどのサーバ（i.e. ローカルサーバ，リモートサーバ）でオペレータの演算を行うかという自由度が存在する．そこでこれらの状況を整理し，3 つの方式を検討した．

3.3.1 ローカルクエリ

データが存在するストレージに直接接続されているサーバで演算を行う．つまりローカルストレージのデータをローカルサーバで処理する．コンバージドネットワークを使用しないので，最もシンプルな方式であると言える．RDMA によるデータ転送をする必要がないので，低コストでパフォーマンスが良いことが期待される．フローを図 3.5 に示す．

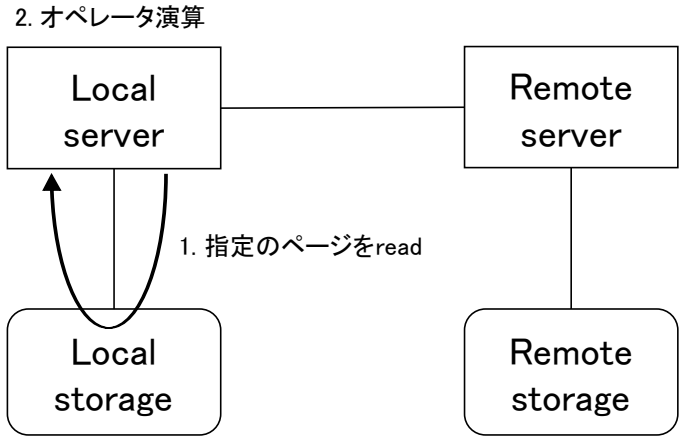

図 3.5: ローカルクエリのフロー

3.3.2 リモートクエリ（ローカル処理）

リモート IO によって，ページをリモートストレージからローカルサーバへと転送したあと，ローカルサーバで演算を行う．つまりリモートストレージのページを

ローカルサーバで処理する．リモート IO のフローの最後に演算処理を付け加える形となる．フローを図 3.6 に示す．

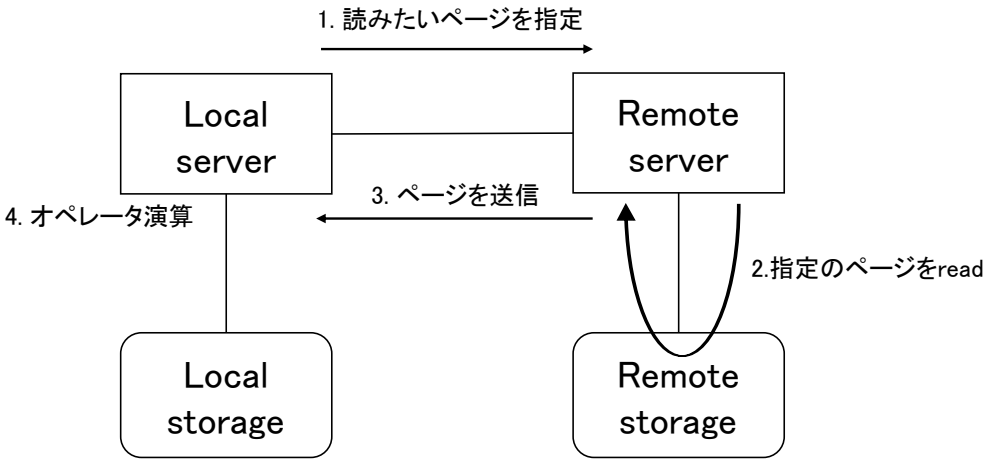

図 3.6: リモートクエリ（ローカル処理）のフロー

3.3.3 リモートクエリ（リモート処理）

リモートストレージのページをローカルサーバへと転送する際，途中のリモートサーバで演算を行う．演算が SQL の where 句のような処理の場合，演算結果のレコード数は演算前のものより減少する場合がある．そのような場合，リモートクエリ（リモート処理）方式では，演算結果の（減少した）レコードのみを RDMA によってローカルサーバに送り返す．

リモートサーバからローカルサーバへのデータ転送量がリモートクエリ（ローカル処理）と比較して減少するので，効率的な方式であることが期待される．フローを図 3.7 に示す．

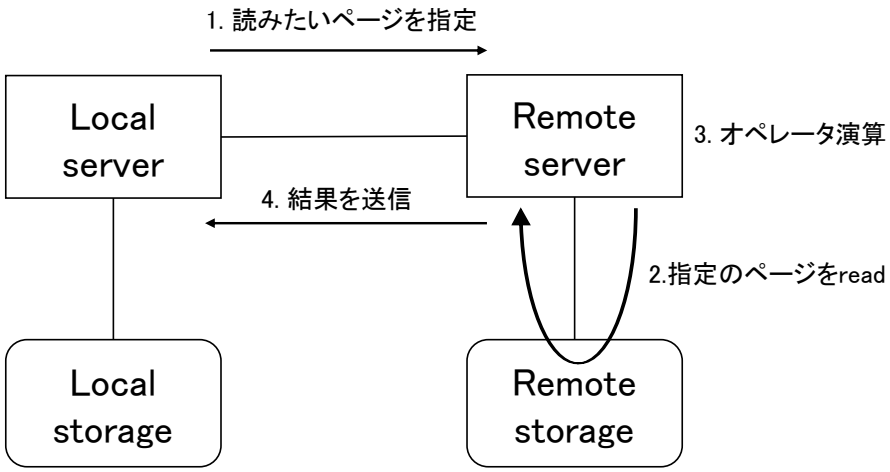

図 3.7: リモートクエリ（リモート処理）のフロー

第4章　評価実験

本章では，3 章で提案した手法の評価実験の結果を示す．最初にリモート IO の性能特性や効率的な方式を示す．次にクエリの模擬実行による Disaggregated storage（非集約型ストレージ）上でのクエリ処理の実験結果を示す．

4.1　実験概要

本研究で行う実験は大別すると 2 つにわけられる．一つ目は Disaggregated storage 環境下で IO を効率良く行う方法を明らかにする実験である．もう一つはオペレータと IO を組み合わせたクエリ処理を効率良く行う方式を明らかにする実験である．

4.1.1　IO マイクロベンチマーク

リード要求を短期間に大量に発行する実験を行う．ワークロードとしてはランダムリードや 4.4.5 節で説明する IO リプレイである．リードサイズは固定長である．

一つのリモート IO を発行してその結果が返ってくるのを待ってから，次のリモート IO を行うのではなく，結果が返ってくるのを待たずに次々にリモート IO を発行し，非同期に結果が返ってくるようにしており，多重的に IO を発行することで，リモート IO のレイテンシを隠蔽し IOPS の向上を測っている．また実験に使用したプログラムでは，性能向上のためコネクションを複数張れるようにした．コネクションとはサーバ間で RDMA 通信を行うときに張るもので，一つのコネクションに送信と受信をそれぞれ受け持つ 2 つの専属のスレッドがローカルサーバとリモートサーバでそれぞれ付くようにプログラムした．

リモート IO は，図 4.1 のように，ローカル IO と RDMA 通信部に分解することが出来る．リモート IO はこれらをブリッジングしたものであるので，当然性能最大値は 2 つのうち性能が低い方に律速されるはずである．そこでローカル IO と RDMA 通信をそれぞれ独立にマイクロベンチマークとして性能測定することで，リモート IO の理論的性能最大値を先に求めた．

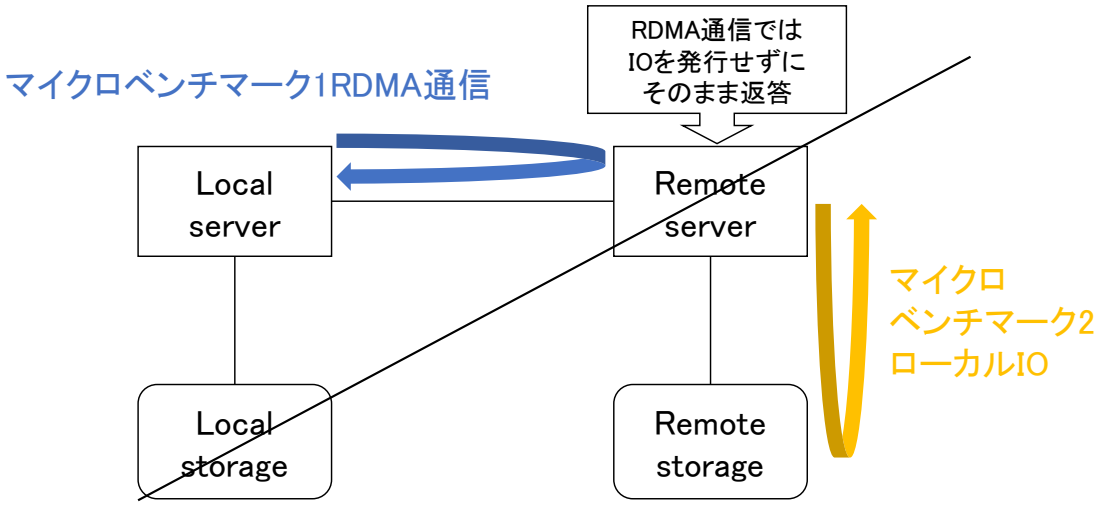

図 4.1: リモート IO を 2 つに分割したマイクロベンチマーク

リモート IO の場合測定に際し，以下のようなパラメータが存在する．

- RDMA のコネクション数
- コネクション当たりの IO スレッド数
- リードサイズ

これらの様々な組み合わせに対して実験を行った．

4.1.2 IO リプレイによるクエリの模擬実行

ローカル IO，リモート IO の上にオペレータを組み合わせたワークロードを行う．具体的には IO を発行して取得したデータ（レコード）を SQL の where 句に相当する

演算で処理することを行う．つまり擬似的にクエリ処理を行う．where 句の演算は CPU を主に消費するので，これにより IO のときよりもサーバの CPU 資源が消費されることになる．このような状況下で性能測定を行うことで，Disaggregated storage 上に構成された並列データベースに於いて，クエリ処理の性能特性を調査し，高速なクエリ処理の方式を実証する．

本ワークロードではローカル IO，リモート IO を用いてページ単位でストレージにアクセスする．ページにはデータベースのレコードが複数並んでおり，ページをパースしレコードを逐次 where 句にかけることでクエリを模擬実行する．

また，リモート IO の実験と同様に測定に際し，以下のようなパラメータが存在する．

- RDMA のコネクション数
- コネクション当たりの IO スレッド数
- ページサイズ（データベースのアクセス単位はページと呼ぶのでそれに従って呼称を変えているが，リードサイズと実装的には同一）

これらの様々な組み合わせに対して実験を行った．

4.2 実験環境

本研究で使用した実験環境について示す．実験に使用したプログラムは全て C 言語で記述した．

本研究の実験環境は 2 台のサーバから構成される．これらのサーバは RDMA を提供するネットワークチャネルのデファクトスタンダードとなっている InfiniBand によって接続される．両サーバの仕様は同一である．詳しい環境を表 4.1 に示す．

InfiniBand 上で RDMA を使用する際は，libibverbs や librdmacm のようなライブラリが提供する API を使用する．これら 2 つのライブラリの違いとしては，librdmacm は libibverbs のラッパーであり，より簡易なインタフェースを提供している．本実

表 4.1: 実験環境

サーバ	Supermicro Super Server
CPU	Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz
メモリ	128GiB
SSD	Intel SSDPE2MD800G4 ×10
HCA	Mellanox ConnectX-3
InfiniBand	56Gbps
OS	CentOS Linux release 7.6.1810 (Core)
OS のページサイズ	4KiB

験では最適化を行うため，より低レイヤであり直接 API を使用できる libibverbs を用いて実装を行う．

またストレージには NVMe 接続された SSD を用いる．ストレージへのアクセス速度が遅いとシステムのボトルネックがストレージになってしまい，正しく RDMA の性能を測ることができなくなってしまう．そこで本研究では，十分に性能が見込めるストレージ構成をとっている [31]．SSD にアクセスする方法として，ストレージに紐づく 10 個のデバイスファイルを O_DIRECT フラグを有効にしてオープンし，それらにラウンドロビン方式で pread 命令を発行する．ここで O_DIRECT フラグとは OS によるキャッシングを防ぐもので，より正確な IO の性能を測定することができる．

4.3 TPC-H

本研究では，Disaggregated storage 上で IO やクエリ処理を行う評価実験の際の実践的なワークロードとして，ランダムリード以外に TPC-H [32] を採用した．使用した Revision は 3.0.0 である．これは Transaction Processing Performance Council が運営するデータベースのベンチマークであり，デファクトスタンダードとして広くデータベースシステムの性能評価に用いられている [33]．TPC-H は Online Analytical Processing(OLAP) という特性のワークロードであり，大量のデータを分析のため

に一度に処理する意思決定支援ベンチマークである．

4.4 実験の方法と結果（IO マイクロベンチマーク）

まず最初に，ワークロードとしてランダムリードを採用し，RDMA 通信（4.4.1 節参照），ローカル IO（4.4.2 節参照），リモート IO（4.4.3 節参照）の各方式で実験的考察を行った．リモート IO が本実験の主目的である．

4.4.1 RDMA 通信の性能測定

リモート IO では，リモートサーバは RDMA によってリード要求のアドレス値を受信し，そのまま IO へと中継する．しかし本マイクロベンチマークでは IO を発行せず，IO を発行したのものと仮定してダミーデータを即座に送りかえす．これにより RDMA 側の通信性能を知ることが出来る．

RDMA 通信の性能測定実験においては，リードサイズが 512B のときの測定の際にコア固定を行っている．コア固定とは sched_setaffinity() によって，各スレッドを特定コアに張り付けることで，OS によるコア移動に伴う無駄なオーバーヘッドを取り除いたものである．

マルチスレッド化し，複数の QP と呼ばれるコネクションを張り，それらで同時多重的にランダムリードを行った．結果はそれぞれリードサイズ別に，図 4.2, 図 4.3, 図 4.4, 図 4.5 である．コネクション数 n に対して 2n+1(2n 個の RDMA の送信・受信スレッドと 1 個のワークロードを管理するメインスレッド) が走っている．[1]．

リードサイズが 512B のときは，図 4.2 のようにコネクション数の増加に伴って最初は OPS が増加しているが，11 から性能が低下しその後は停滞している．これは実験サーバの物理コアが 1 ソケットあたり 22 個であることに対し，11 コネクションのときには，23 個のスレッドがそれぞれコアに固定されることにより，スレッドがソケットを跨ぐ状況が発生しているためと考えられる．コネクション数が 10 のと

[1]物理コア数を超えるスレッド数は固定することができないので，コア固定のグラフは途中で途切れている．

きに，OPS は最大値である 6M 弱に達しており InfiniBand が提供する RDMA が非常に高速なことがわかる．一方で，それ以外のリードサイズのときはコネクション数によって，特に性能は変化していない．

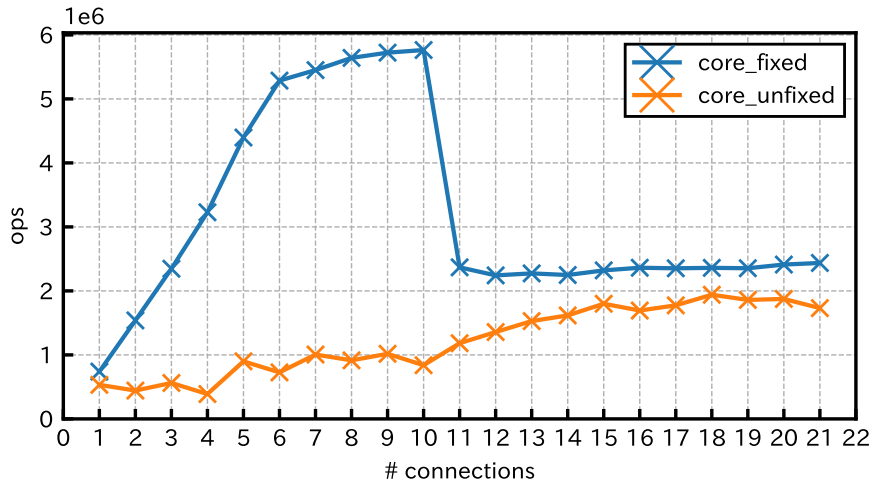

図 4.2: 512B RDMA 通信

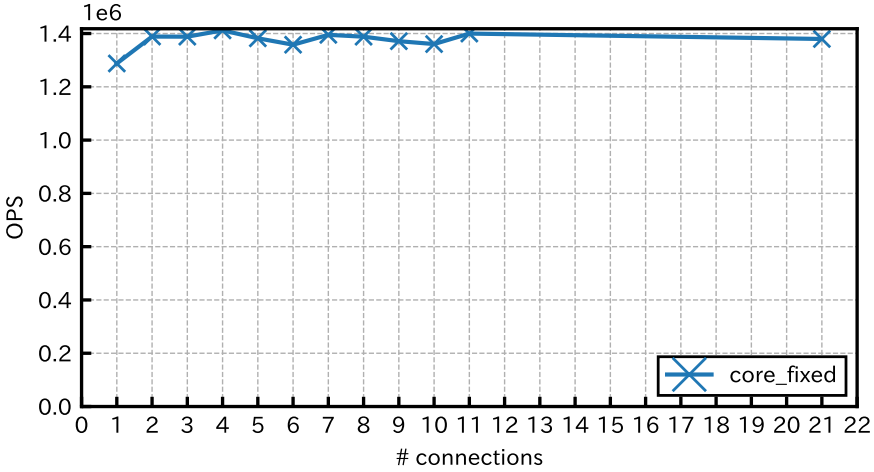

図 4.3: 4KiB RDMA 通信

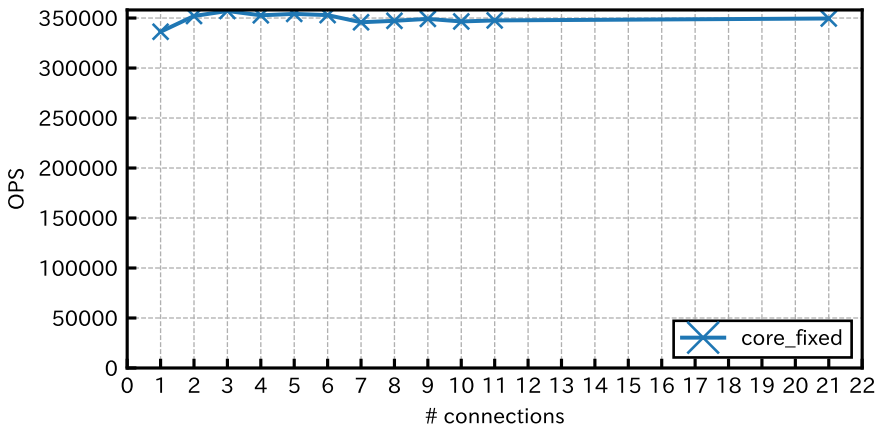

図 4.4: 16KiB RDMA 通信

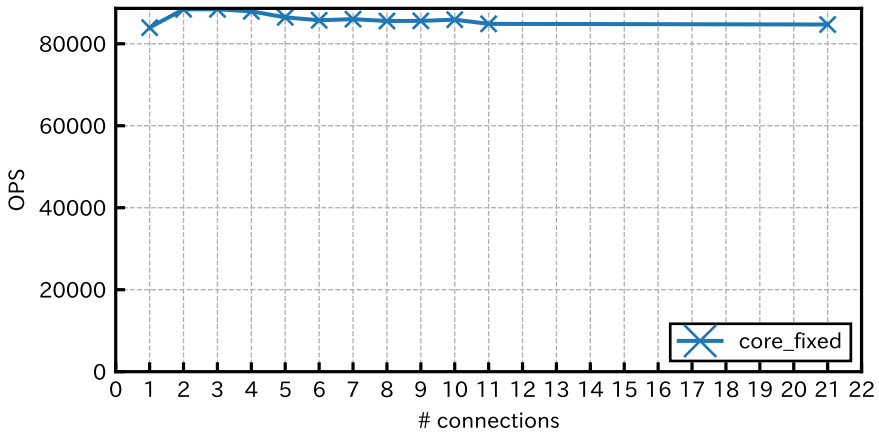

図 4.5: 64KiB RDMA 通信

4.4.2 ローカル IO の性能測定

リモート IO を構成する要素であるローカル IO にてランダムリードを行う実験を行った．つまりこの実験ではサーバは 1 台しか使用せず，RDMA 等のネットワークプロトコルは一切使用していない．実験設定としては，スレッド数を独立変数として設定して実験した．n スレッドを立ち上げ，その各スレッドで乱数生成→その乱数をアドレス値として read を行う，というループを繰り返すものである．マルチス

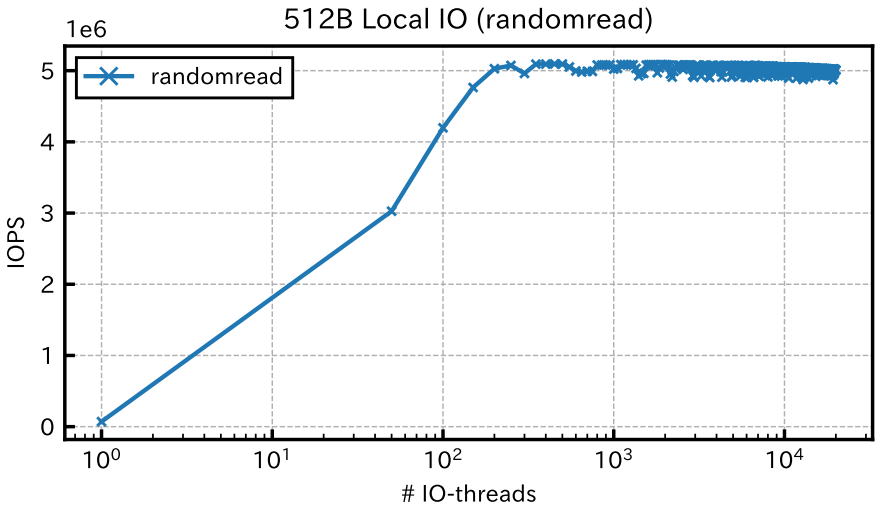

図 4.6: 512B ローカル IO

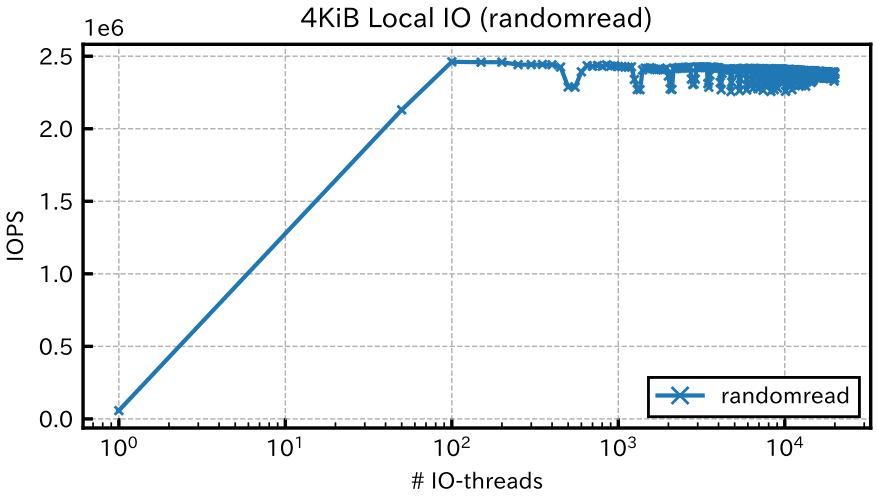

図 4.7: 4KiB ローカル IO

レッドにした理由は，read 命令を発行したあと，その結果が返ってくるまでのブロッキング時間をマルチスレッドにより隠蔽出来る等の理由で高性能化が見込めるからである．ローカル IO はリモート IO に対するベースラインという位置付けもある．

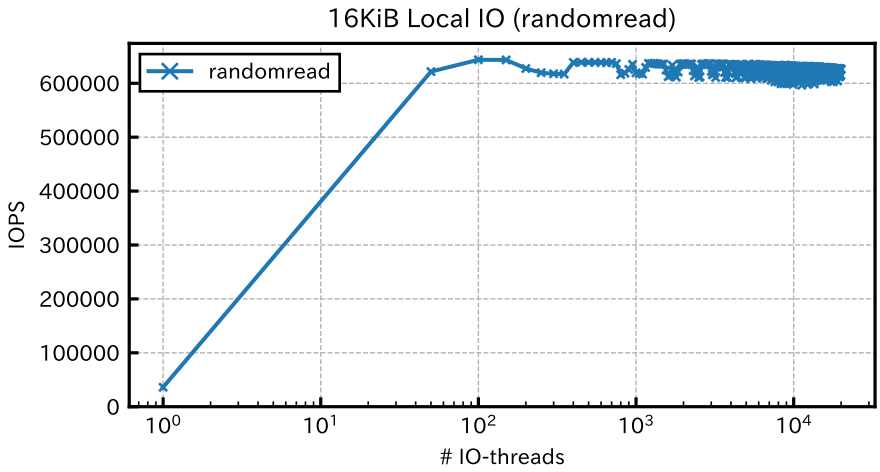

図 4.8: 16KiB ローカル IO

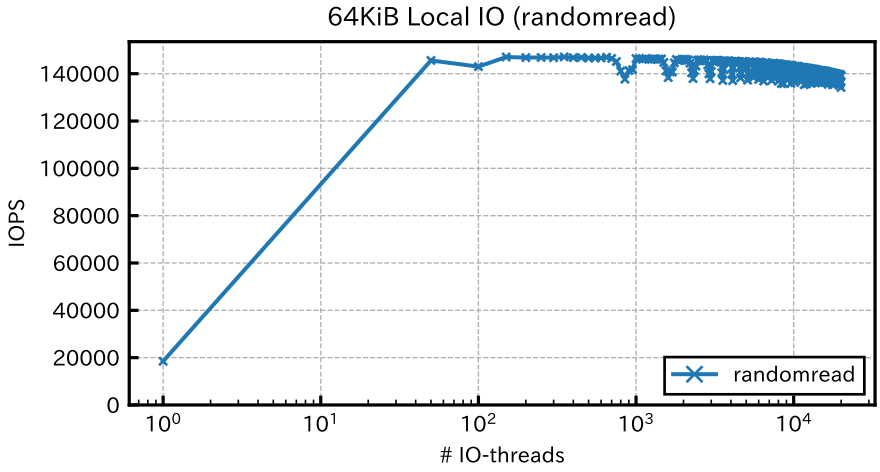

図 4.9: 64KiB ローカル IO

リードサイズは 512B，4KiB，16KiB，64KiB のそれぞれで実験を行った．結果はリードサイズ別にそれぞれ，図 4.6, 図 4.7, 図 4.8, 図 4.9 である．いずれの結果でも，スレッド数が大体 100 付近で性能が頭打ちしていることがわかる．

4.4.3 リモート IO の性能測定

RDMA のコネクション数とその各コネクションが生成する IO スレッド数の様々な組み合わせのパラメータで，リモート IO のランダムリードを行った．3.2 節で述べたように，リモートサーバでの IO スレッド管理方法は動的手法と静的手法の二つがあるのでそれぞれの結果を掲載する．

4.4.3.1 動的 IO スレッド方式

まずコネクション数を変化させて IOPS を測定した．RDMA 通信の時と同様に，コネクション数 n に対して 2n+1(2n 個の RDMA の送信・受信スレッドと 1 個のワークロードを管理するメインスレッド) が走っている．512B ランダムリードを行った結果は，図 4.10 である．

結果はコネクション数の増加に対して IOPS は単調減少でありコネクション数 1 のときに，70K IOPS 程度となっている．コアを固定して場合は劇的に性能が低下し，数百 IOPS 程度で推移している．この理由は，ボトルネックがリモートストレージに対する IO にあるため，コネクションを増やして RDMA 用のスレッドを無駄に増やすよりも，IO を最大限行えるよう可能な限りリモートサーバで走る RDMA のためのスレッド数を減らしておいたほうが良いからだと考えられる．IO スレッドを生成した後，生成した IO スレッドに，CPU 時間が中々渡らないため，IO スレッドが全然実行されていないことが考えられる．これは，RDMA の受信のためのポーリングでは CPU 時間を要求する一方，read 命令は割込であり，ポーリングほど CPU 時間を要求しないため，OS スケジューラが適切に CPU 時間を配分できなくなっている状況により引き起こされているのではないかと理由付けをすることができる．コア固定を行うとただでさえ CPU が渡らない状況であるのに，余計 CPU が渡らなくなりいっそう性能が低下していると考えられる．

CPU が適切に使われていないか確かめるため，このワークロード実行中の CPU 利用率を監視した．コネクション数 1 で 512B のランダムリモート IO（動的 IO スレッド手法）を行いコアごとの CPU 使用率の時間変化を測定した．結果は，図 4.11

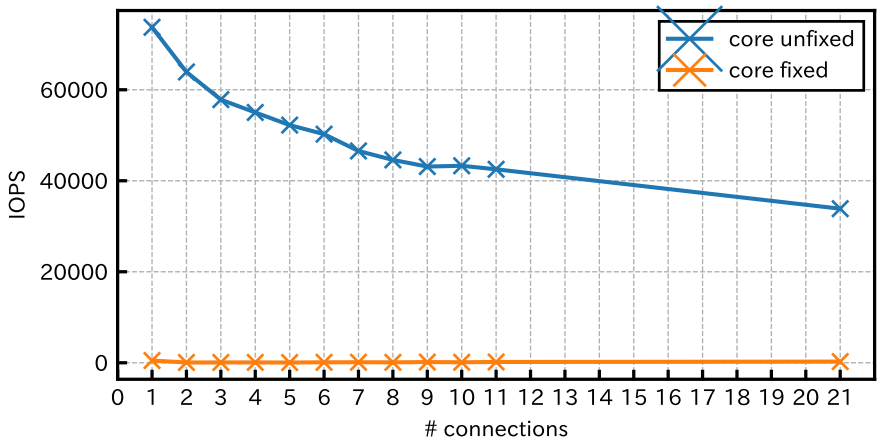

図 4.10: 512KiB リモート IO（動的 IO スレッド方式）

である．上から二つのコアは RDMA 通信に使用されているので CPU 使用率が常に高い．IO スレッドはこれら 2 つのコア以外の任意のコアで本来実行されるはずだが，実際は少数の偏ったコアで実行されていることがわかる．

4.4.3.2 静的 IO スレッド方式

次に，静的 IO スレッド方式でリモート IO の性能測定を行った．[2]

静的 IO スレッド方式ではパラメータとして RDMA のコネクション数と各コネクション当たりの IO スレッド数の 2 つがある．まずはリードサイズ 512B で，どちらか一方のパラメータを固定して，もう片方のパラメータを振って実験した．

RDMA のコネクション数を固定した結果は図 4.12 である．RDMA のコネクション数を 10 に固定して，系全体の IO スレッド数を横軸とした．系全体の IO スレッド数とは RDMA のコネクション数とコネクション当たりの IO スレッド数をかけ合わせた値であり，リモートサーバ上に存在する全ての IO スレッドの数である．参考としてリモート IO を構成する要素であるローカル IO と RDMA 通信の結果も掲載している．原理上これらよりリモート IO の性能が上回ることはない．性能ピーク

[2] コア固定を行った場合と，行わない場合両方で計測したが，行わない場合のほうが常に性能が良かったため，コア固定を行わない場合のみ掲載する．

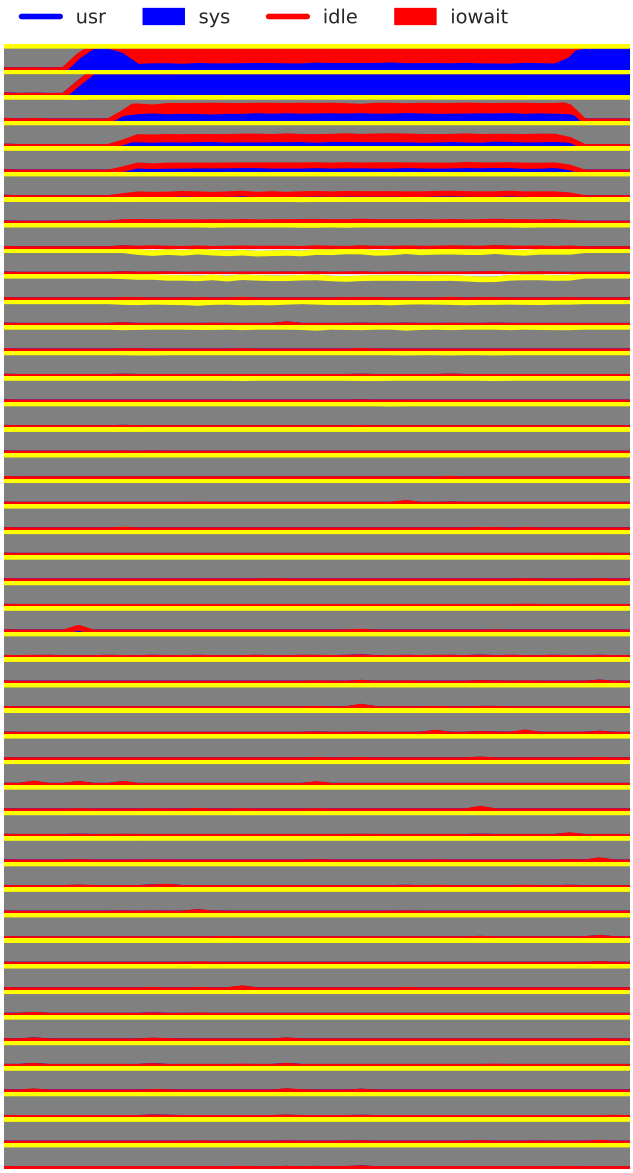

図 4.11: リモート IO（動的 IO スレッド方式）実行中の CPU 利用率の時間変化．横軸は時間軸である．縦軸は各コアの CPU 使用率を示す．

が IO スレッド数 100 程度の箇所に存在する．これは，ローカル IO の性能が IO スレッド数 100 から 200 付近でほぼ飽和する一方で，多すぎる IO スレッド数はオー

バーヘッドとなり，RDMA 通信のパフォーマンスを阻害し得るので，両者の性能バランスのよい IO スレッド数 100 付近でピークをとっていると考えられる．しかし，ピークでも理論的性能最大値（この場合ローカル IO の性能値）との間には大きくギャップがある．

逆に，コネクション当たりの IO スレッド数を固定した結果は図 4.13 である．コネクション当たりの IO スレッド数を 100 に固定して，コネクション数を横軸とした．コネクション数を増やせば増やすほど性能が向上しており，コネクション数 100 付近では，ほぼ理論的性能最大値に達しており，理想的な性能に達していることがわかる．

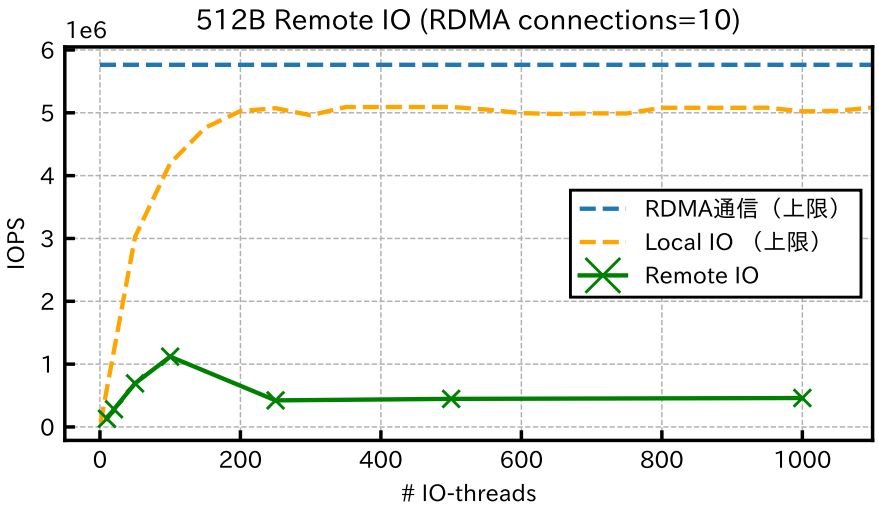

図 4.12: 512B リモート IO（静的 IO スレッド方式）（RDMA コネクションを 10 に固定）

次に，2 つのパラメータのどちらかを固定することなく，様々なパラメータで実験を行った．そして，横軸を系全体の IO スレッド数，縦軸を IOPS としてプロットした．ここで，本来の実験パラメータ空間としては，RDMA のコネクション数，各コネクション当たりの IO スレッド数，（測定結果として）IOPS の 3 次元空間であるが，RDMA のコネクション数とコネクション当たりの IO スレッド数を掛け合わ

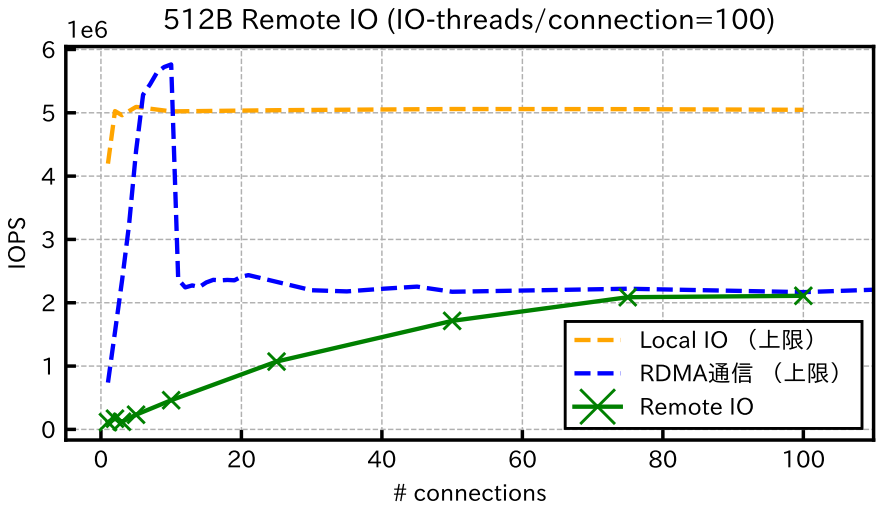

図 4.13: 512B リモート IO（静的 IO スレッド方式）（コネクション当たりの IO スレッド数を 100 に固定）

せた値を，「系全体の IO スレッド数」として，新たな軸を生成し実験空間を 2 次元に射影したグラフになる．[3]ローカル IO のときと同じくスレッド数が 100 に近い領域では，性能が頭打ちになっていることが見受けられる．

リードサイズはローカル IO と同じく，512B，4KiB，16KiB，64KiB のそれぞれで実験を行った．結果はリードサイズ別にそれぞれ，図 4.14, 図 4.15, 図 4.16, 図 4.17 である．512B の結果（図 4.14）では，IO スレッド数が 1000 付近でピーク性能に達している一方で，同じ IO スレッド数でも性能にバラツキがある．これは，様々なコネクション数の結果を一括してプロットしているためである．このことは，コネクション数によって性能が大きく変わるということを示しており，リモート IO を用いてパフォーマンスを追求する際は，パラメータに細心の調整が必要なことがわかる．なお，パラメータの絶対値は動作環境に依存するところが大きいと考えられ，Disaggregated storage を構成するサーバごとに，調整する必要があると考えられる．

[3]例えば，コネクション数=20，コネクション当たりの IO スレッド数=5 のときは系全体の IO スレッド数は 100 となる．また，コネクション数=25，コネクション当たりの IO スレッド数=4 でも同じく系全体の IO スレッド数は 100 となる．それ故に同じ横軸の値でも複数の測定点が存在する．

4KiB の結果は同様のグラフの形状を呈している図 4.15．16KiB と 64KiB の結果（図 4.16，図 4.17）では、系全体の IO スレッド数によってほぼ一意に性能が定まっている．IO スレッド数をある程度確保しておけば，ほぼ最大性能値を達成できるということであり，リードサイズが 16KiB や 64KiB のときは系としてよりロバストであるといえる．

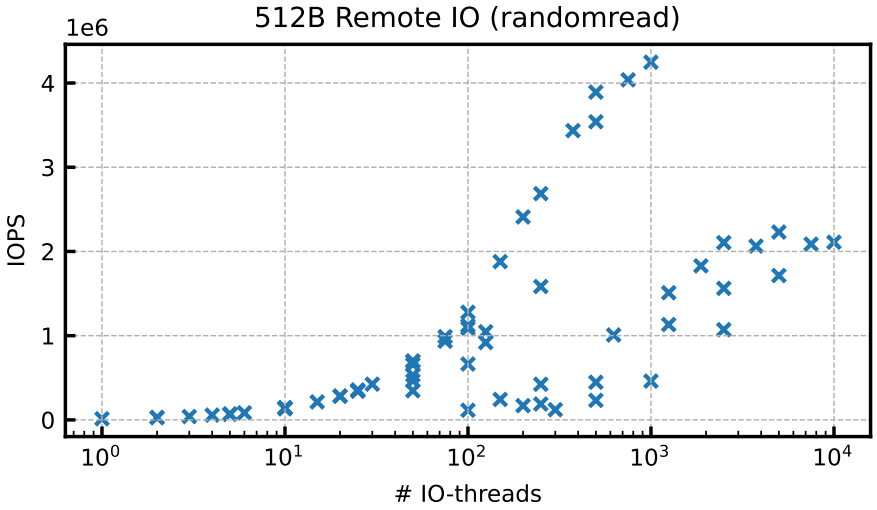

図 4.14: 512B リモート IO（静的 IO スレッド方式）

今後，本論文では特筆しない限り，性能の観点から，リモート IO は静的 IO スレッド方式を用いているものとする．

4.4.4 RDMA 通信，ローカル IO，リモート IO のまとめ

リモート IO と，リモート IO を構成するローカル IO と RDMA 通信の性能測定を行い考察を行った．最後に，リードサイズ別の各方式の最大性能値を纏めたものが，図 4.18 である．ここでは IOPS ではなくスループットによる比較であることに注意されたい．リードサイズが 512B のときは，4KiB 以上のときと比較して性能が劣っている．これは，実験環境の OS のページサイズ設定が 4KiB であることにより，512B リードは内部的には 4KiB リードに変換されていたためであると考えられる．

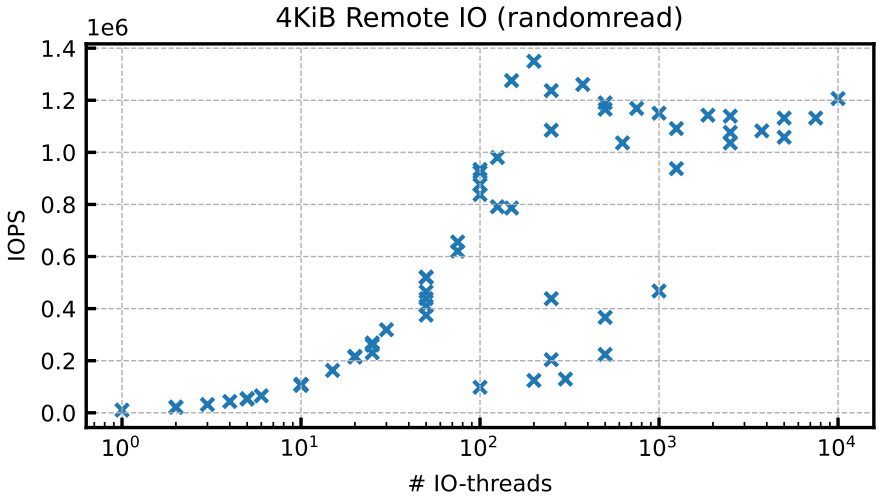

図 4.15: 4KiB リモート IO（静的 IO スレッド方式）

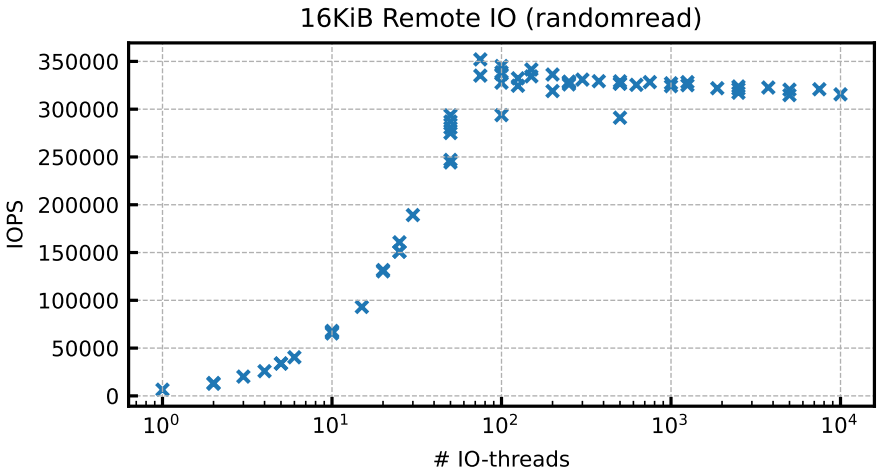

図 4.16: 16KiB リモート IO（静的 IO スレッド方式）

本実験の主題であるリモート IO について着目すると，4KiB 以上のリードサイズでは，RDMA 通信がボトルネックとなっていることがわかり，リモート IO でもほぼ同様の性能を達成している．原理上これよりスループットの高いアクセスは，RDMA 通信の方式から抜本的に変更しないと原理上不可能であり，理想的な結果が得られ

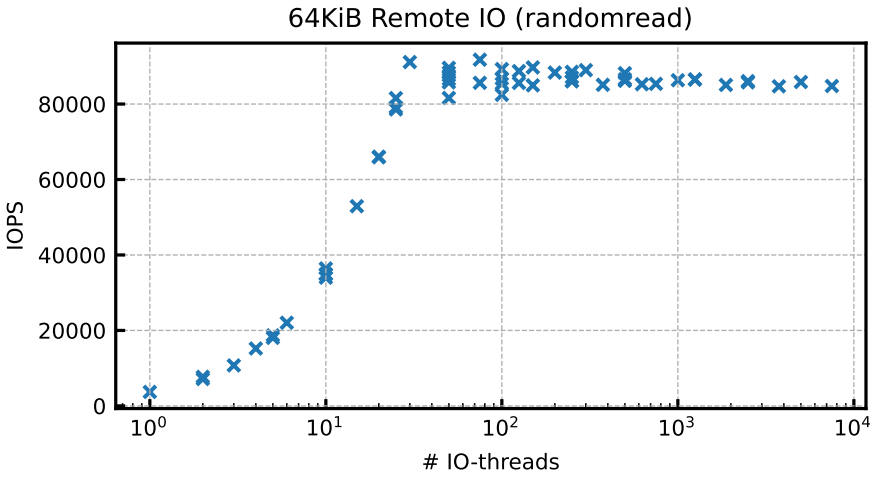

図 4.17: 64KiB リモート IO（静的 IO スレッド方式）

たといえる．

Disaggregated storage を考えたときに，ローカル IO とリモート IO のアクセスコストの差が重要になる．そこでリモート IO がローカル IO と比較してどれだけの IOPS を達成したかを，表 4.2 に纏めた．512B では理論的性能最大値（この場合ローカル IO の 5.1M IOPS）の 85%である 4.3M IOPS を達成している．これはリモートストレージへのアクセスであることを考えると非常に高速である．

表 4.2: リードサイズ別のローカル IO，リモート IO の IOPS 比較

リードサイズ	ローカル IO	リモート IO	$\frac{\text{リモート } IO}{\text{ローカル } IO}$
512 B	5.1 M	4.3 M	85%
4 KiB	2.5 M	1.4 M	55%
16 KiB	0.64 M	0.35 M	55%
64 KiB	0.15 M	9.3 M	62%

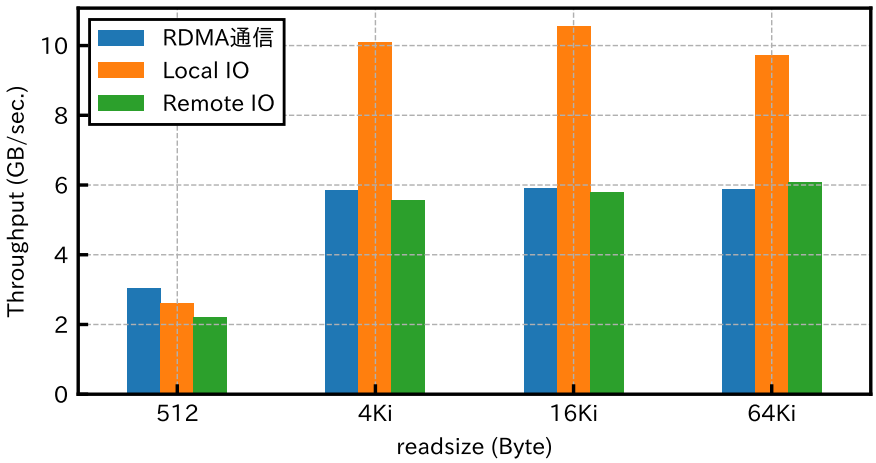

図 4.18: リードサイズ別の各方式のランダムリード性能最大値

4.4.5 IO リプレイ

4.4.5.1 実験設定

SQL クエリを実行したときの IO ログ（どのアドレスを読んだか）の履歴の通りに IO を発行すること（以下 IO リプレイと呼ぶ）で，擬似的にクエリの実行性能を測定することができる．OLAP 系ベンチマークである TPC-H（4.3 節参照）のクエリ 3 を擬似実行することで，ランダムリードとは異なる実際のクエリの IO パターンでの性能測定も行った．なお用意した IO ログは 16KiB 単位でのアクセスだったので本実験では明記しない限り 16KiB アクセスを行っている．

クエリ 3 の内容は，ソースコード 4.1 である．IO リプレイでは IO を再現するだけで，select，where，groupby，orderby などの演算は行わない．

ソースコード 4.1: TPC-H クエリ 3

```
1  select
2      l_orderkey,
3      sum(l_extendedprice*(1-l_discount)) as revenue,
4      o_orderdate,
5      o_shippriority
6  from
7      customer,
8      orders,
```

```
9       lineitem
10  where
11      c_mktsegment = '[SEGMENT]'
12      and c_custkey = o_custkey
13      and l_orderkey = o_orderkey
14      and o_orderdate < date '[DATE]'
15      and l_shipdate > date '[DATE]'
16  group by
17      l_orderkey,
18      o_orderdate,
19      o_shippriority
20  order by
21      revenue desc,
22      o_orderdate;
```

まず IO スレッド数を 10000 に固定してローカル IO とリモート IO のそれぞれで TPC-H クエリ 3 の IO リプレイを行った．結果は図 4.19 である．表 4.2 によればリードサイズが 16KiB のときランダムリードにおいて，リモート IO の IOPS はローカル IO の IOPS の 55%であるので，実行時間は高々約 2 倍になるはずだが，実際はもっと長くなっている．これは，リモート IO では RDMA にキャッシュ機構がないために，ローカル IO と比較してキャッシュが効かず，IO アクセスパターンに局所性がある IO リプレイでは，性能差がランダムリードのときより開いているからであると考えられる．

そこでより詳しく，IO リプレイでの性能特性を見るために，ローカル IO，リモート IO のそれぞれで IO スレッド数を変化させて挙動を確かめた．

4.4.5.2 IO リプレイ（ローカル IO）の性能測定

ローカル IO の結果はリードサイズ別にそれぞれ図 4.20, 図 4.21, 図 4.22, 図 4.23 である．参考としてランダムリードの結果も併せて載せる．やはりローカル IO では IO リプレイのほうが大幅に性能が良い．これは IO リプレイでは IO アクセスに局所性があり，キャッシュヒット率がランダムリードに比べて向上するためであるためと考えられる．

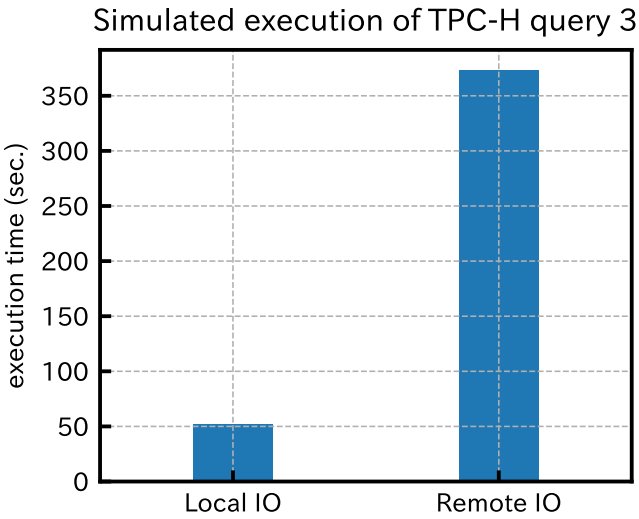

図 4.19: 16KiB での IO リプレイのローカル IO とリモート IO の実行時間の比較

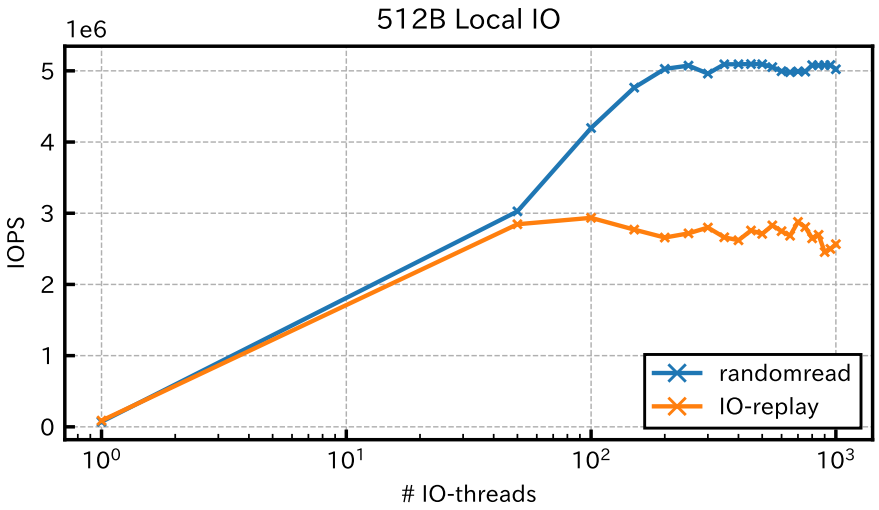

図 4.20: ランダムリードと IO リプレイに於ける 512B ローカル IO

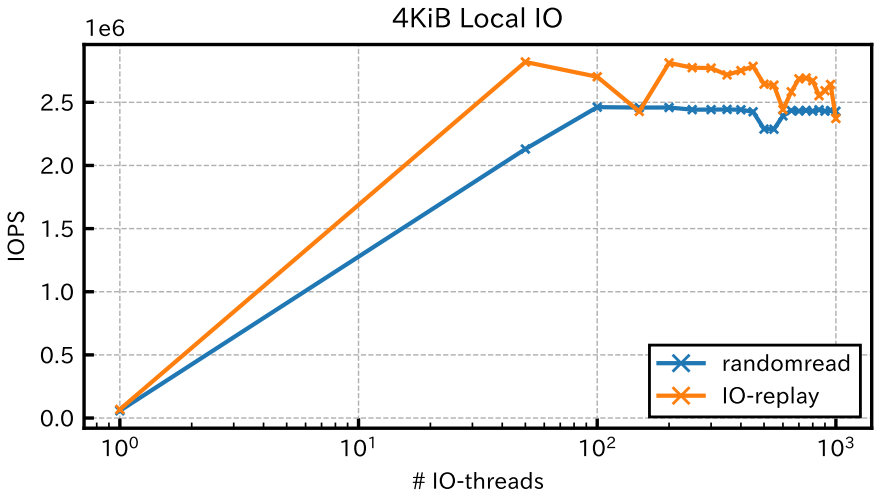

図 4.21: ランダムリードと IO リプレイに於ける 4KiB ローカル IO

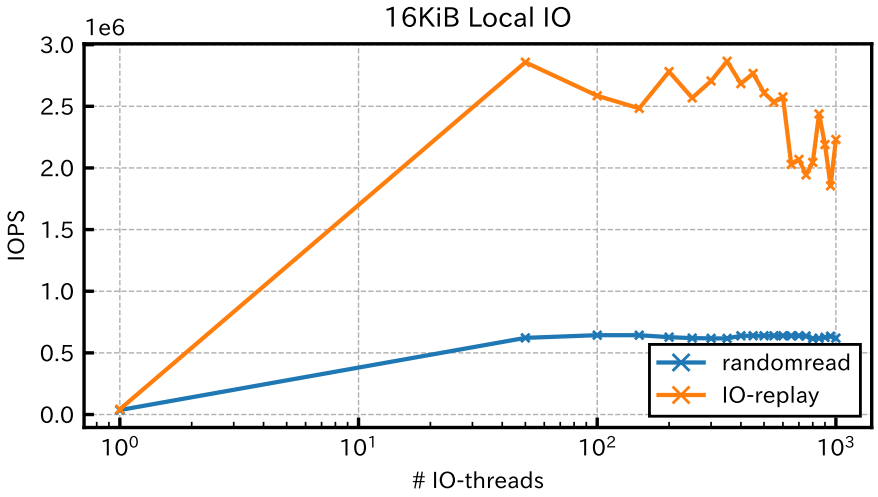

図 4.22: ランダムリードと IO リプレイに於ける 16KiB ローカル IO

4.4.5.3 IO リプレイ（リモート IO）の性能測定

リモート IO の結果はリードサイズ別にそれぞれ図 4.24, 図 4.25, 図 4.26, 図 4.27 である．横軸は軽全体のスレッド数である．16KiB と 64KiB の測定結果で多少アノ

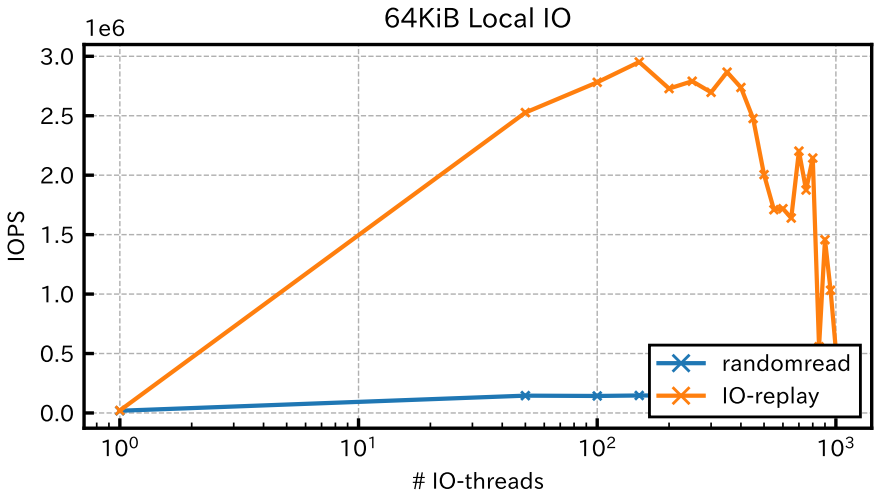

図 4.23: ランダムリードと IO リプレイに於ける 64KiB ローカル IO

マリーが見られることを除けば，ランダムリードのとき（図 4.14，図 4.15，図 4.16，図 4.17）とグラフの概形は同一である．しかし性能の絶対値は，ローカル IO と比較すると低下しており，前述の通りキャッシュがないことが原因になっていると考えられる．

コネクションあたりの IO スレッド数を固定して，コネクション数を変化させてプロットすると，リードサイズ別に図 4.28, 図 4.29, 図 4.30, 図 4.31 のようになる．リモート IO では，IO リプレイのほうがランダムリードとして比較して，性能が優位に良いという現象は見られない．

特に図 4.28 の 512B のケースでは，本来局所性が存在する IO リプレイのほうが性能がランダムリードを上回るはずであるが，実際はその半分程度まで最大性能が落ち込んでしまっている．これは，IO リプレイを行う際にログの配列にアクセスにするほうが，ランダムリードで乱数を生成するより，コストがかかってしまっているからであると考えられる．

それ以外のリードサイズではランダムリードと IO リプレイでほぼ性能は同一であり，リモート IO ではキャッシュが効かないということを考えると，妥当な結果で

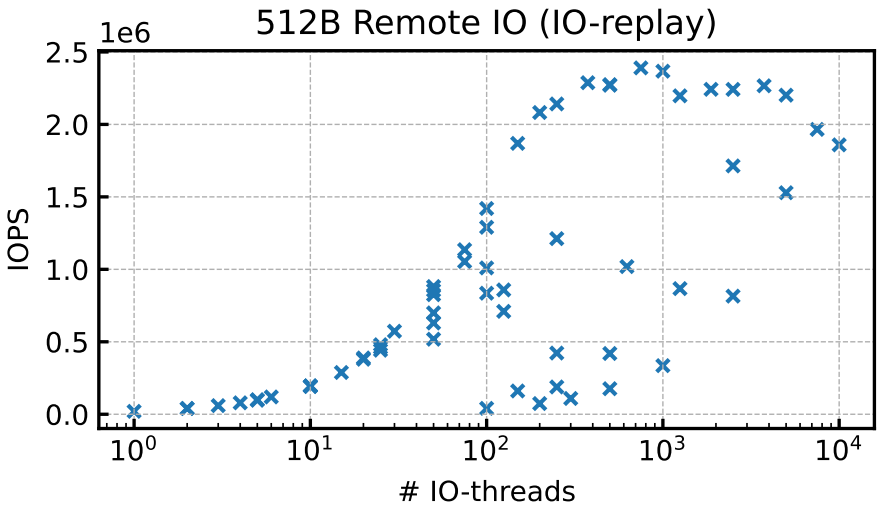

図 4.24: 512B リモート IO（IO リプレイ）

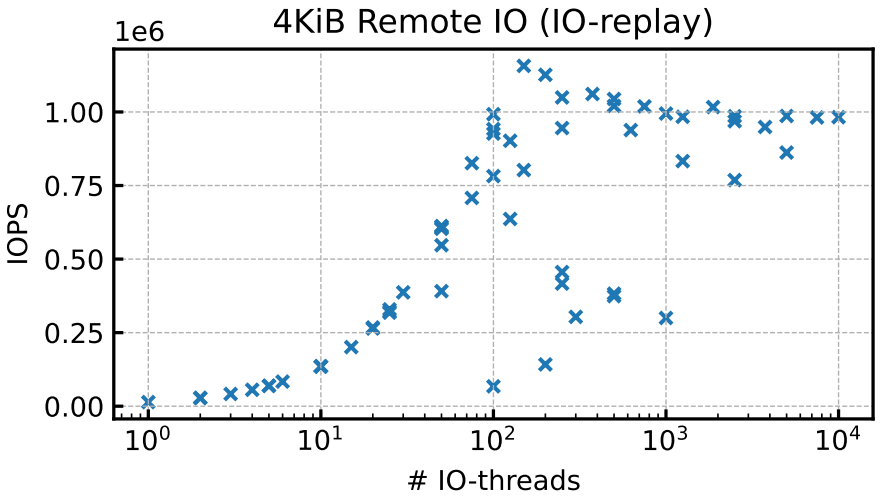

図 4.25: 4KiB リモート IO（IO リプレイ）

ある．

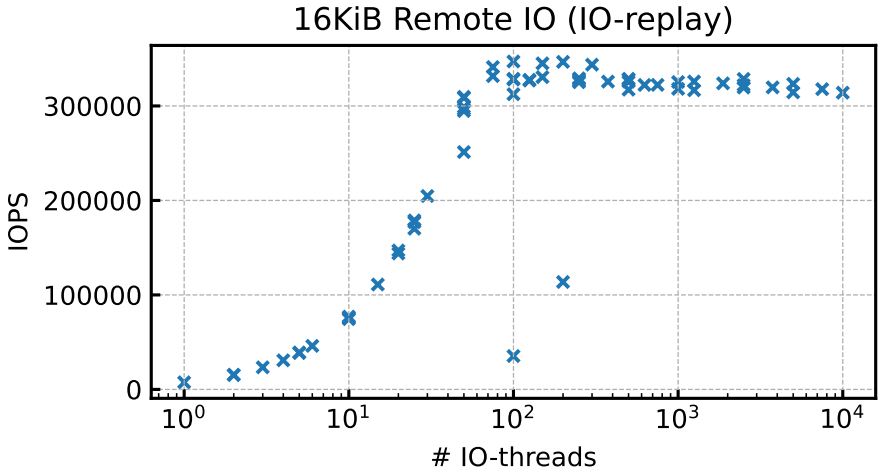

図 4.26: 16KiB リモート IO（IO リプレイ）

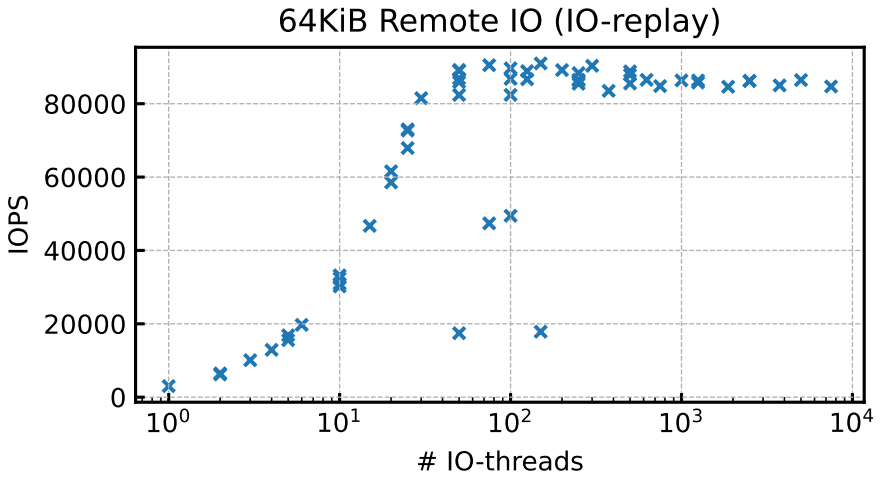

図 4.27: 64KiB リモート IO（IO リプレイ）

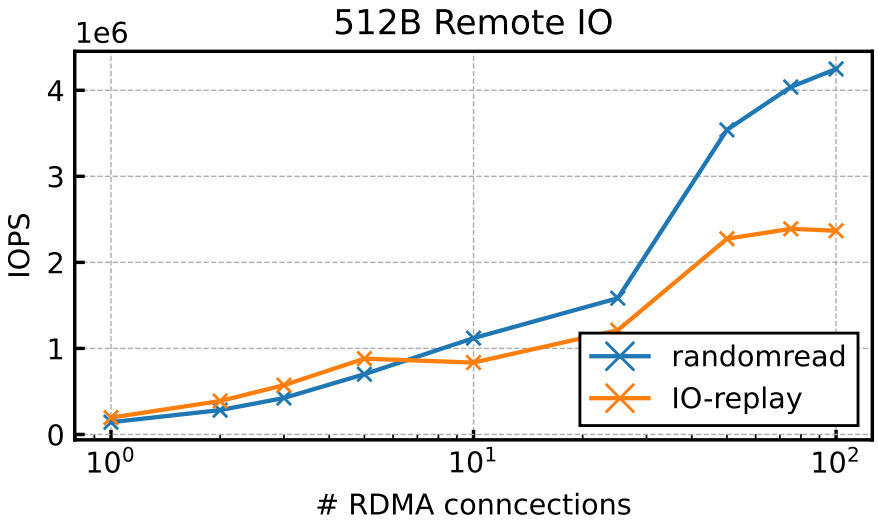

図 4.28: ランダムリードと IO リプレイに於ける 512B リモート IO

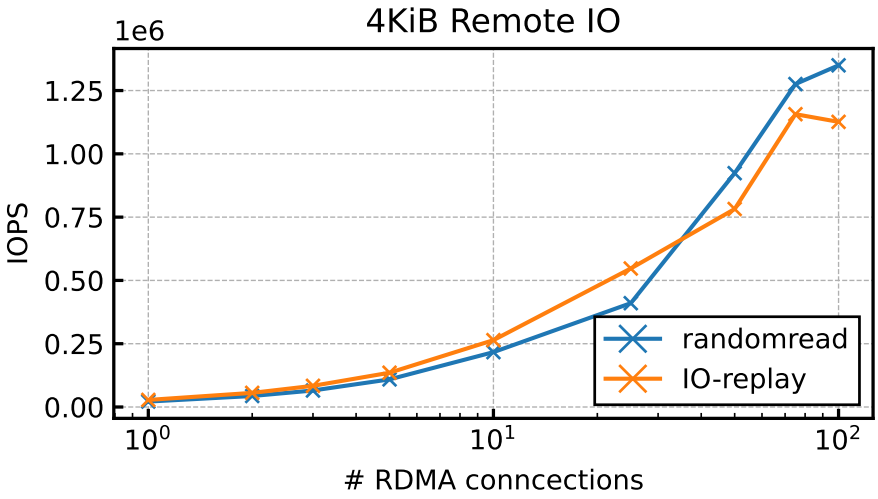

図 4.29: ランダムリードと IO リプレイに於ける 4KiB リモート IO

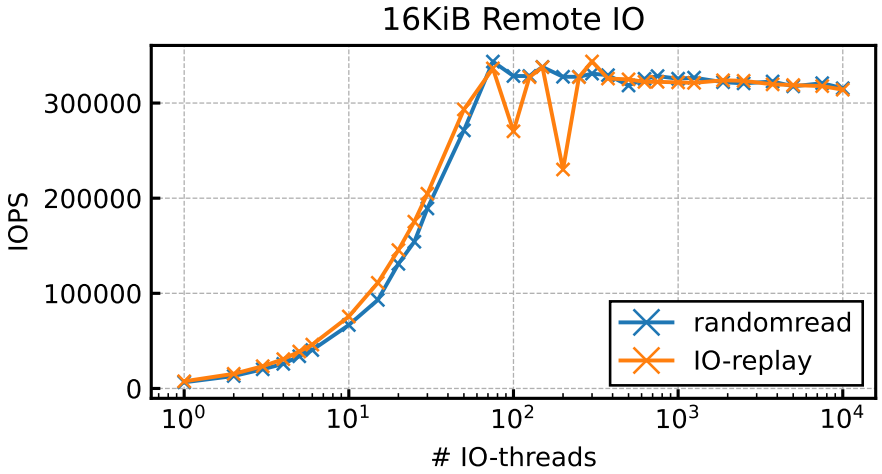

図 4.30: ランダムリードと IO リプレイに於ける 16KiB リモート IO

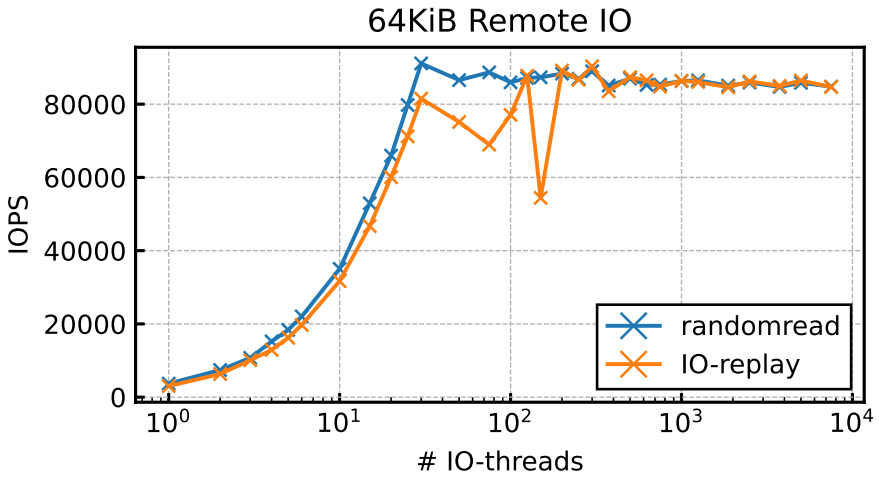

図 4.31: ランダムリードと IO リプレイに於ける 64KiB リモート IO

4.5 実験の方法と結果（IO リプレイによるクエリの模擬実行）

4.5.1 実験設定

実験用のデータセットとして 4.3 節で紹介した TPC-H を使用した．データ生成には TPC-H に付属するツールである dbgen によって生成した．dbgen では Scale Factor (SF) と呼ばれる値を設定することで，生成するデータ量を調整することが出来，本研究では SF=1 に設定した．

TPC-H では本来複数のテーブルが定義されているが, 今回は用途に合わせて lineitem テーブルを使用した．lineitem テーブルの定義を表 4.3 に示す [32]．なお SF=1 で生成された lineitem は表 4.4 のとおりである．

ソースコード 4.2: TPC-H クエリ 1

```
1  select
2      l_returnflag,
3      l_linestatus,
4      sum(l_quantity) as sum_qty,
5      sum(l_extendedprice) as sum_base_price,
6      sum(l_extendedprice*(1-l_discount)) as sum_disc_price,
7      sum(l_extendedprice*(1-l_discount)*(1+l_tax)) as sum_charge,
8      avg(l_quantity) as avg_qty,
9      avg(l_extendedprice) as avg_price,
10     avg(l_discount) as avg_disc,
11     count(*) as count_order
12 from
13     lineitem
14 where
15     l_shipdate <= date '1998-12-01' - interval '120' day (3)
16 group by
17     l_returnflag,
18     l_linestatus
19 order by
20     l_returnflag,
21     l_linestatus;
```

また，TPC-H ではベンチマークのためのクエリも複数用意されている．lineitem テーブルのみを使用するクエリである，クエリ 1（ソースコード 4.2），クエリ 6（ソースコード 4.3），クエリ 15（ソースコード 4.4）を使用した．クエリ 15 に関しては lineitem テーブルのみを使用するのはサブクエリであったのでサブクエリのみの実

表 4.3: TPC-H の lineitem テーブルのカラムとデータ型．データベース上でのデータ型は仕様書で定義されている要件．C 言語上でのデータ型は実装に際し，使用したデータ型

カラム名	データベース上でのデータ型	C 言語上でのデータ型
L_ORDERKEY	identifier	unsigned int
L_PARTKEY	identifier	unsigned int
L_SUPPKEY	identifier	unsigned int
L_LINENUMBER	integer	unsigned short
L_QUANTITY	decimal	unsigned short
L_EXTENDEDPRICE	decimal	float
L_DISCOUNT	decimal	float
L_TAX	decimal	float
L_RETURNFLAX	fixed text, size 1	char
L_LINESTATUS	fixed text, size 1	char
L_SHIPDATE	date	char[11]
L_COMMITDATE	date	char[11]
L_RECIPTDATE	date	char[11]
L_SHIPINSTRUCT	fixed text, size 25	char[26]
L_SHIPMODE	fixed text, size 10	char[11]
L_COMMENT	variable text size 44	char[45]

表 4.4: dbgen によって生成された lineitem

Scale Factor	1
レコード数	6001215
サイズ	759863287B

行となっている．なお，select，groupby，orderby のような演算は今回は行っておらず，where 句のみの演算である．

ソースコード 4.3: TPC-H クエリ 6

```
1  select
2      sum(l_extendedprice*l_discount) as revenue
3  from
4      lineitem
```

```
5     where
6         l_shipdate >= date '1995-01-01'
7         and l_shipdate < date '1995-01-01' + interval '1' year
8         and l_discount between 0.05 - 0.01 and 0.05 + 0.01
9         and l_quantity < 24;
```

ソースコード 4.4: TPC-H クエリ 15 のサブクエリ（以下，単にクエリ 15 と呼ぶ）

```
1     select
2         l_suppkey,
3         sum(l_extendedprice * (1 - l_discount))
4     from
5         lineitem
6     where
7         l_shipdate >= date '1993-04-01'
8         and l_shipdate < date '1993-04-01' + interval '3' month
9     group by
10        l_suppkey;
```

なお，予めクエリを実行し，各クエリの選択率を求めたところ表 4.5 のようになった．選択率が取りうる値は [0,1] であるので，多種の選択率が存在しており実験設定として妥当であることがわかる．

表 4.5: TPC-H の各クエリの選択率

クエリ	選択率
1	0.97
6	0.013
15	0.15

4.5.2 クエリ 1 の性能測定

選択率が 0.97 であるクエリ 1 を実行した結果を示す．

まず，系全体の IO スレッド数と性能の関係を図 4.32 に示す．縦軸は実行に要した時間であり，短いほど性能が良いといえる．ローカルクエリとリモートクエリ（リモート処理）では IO スレッドの数によらず，比較的安定して高性能を達成していることがわかる．一方，リモートクエリ（ローカル処理）では IO スレッド数が少ない

領域でこそ安定して高性能だが，IO スレッド数が増加すると性能が悪いケースが多くなっていることがわかる．

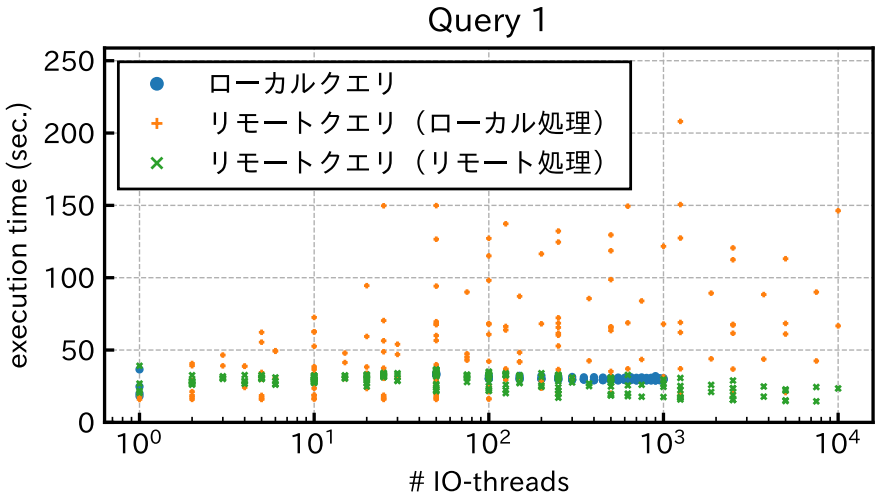

図 4.32: クエリ 1 の IO スレッド数と実行時間

次に，RDMA コネクション数と性能の関係を図 4.33 に示す．ローカルクエリでは RDMA を使用しないので，このグラフには示されない．リモートクエリ（リモート処理）では，IO スレッド数のときと同様に，RDMA コネクション数によらず，安定して高性能を達成している．対して，リモートクエリ（ローカル処理）では RDMA コネクション数が増加するほど性能が悪化している．

次に，ページサイズ別で，最良の実行時間であったケースを方式ごとに図 4.34 で比較した．（それぞれの場合でベストなケースを抽出しているので，それぞれ IO スレッド数と RDMA コネクション数は異なる．）ページサイズを大きめに取るほど各方式ごとの性能差が小さくなることがわかる．ページサイズが 128KiB の場合は，性能は各方式でほぼ同一で，クエリをどのサーバで処理するかはほとんど気にしなくてよいと言える．ページサイズ 4KiB のとき以外は，リモートクエリ（リモート処理）の性能が最良である．直感的には RDMA によるデータ転送を必要としないローカルクエリの性能が最も良さそうであるが，実際は真逆である．

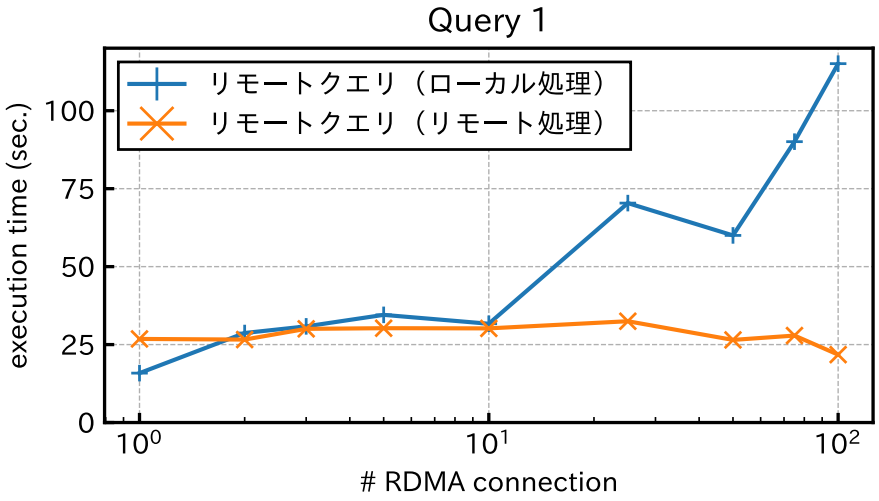

図 4.33: クエリ 1 の RDMA コネクション数と実行時間

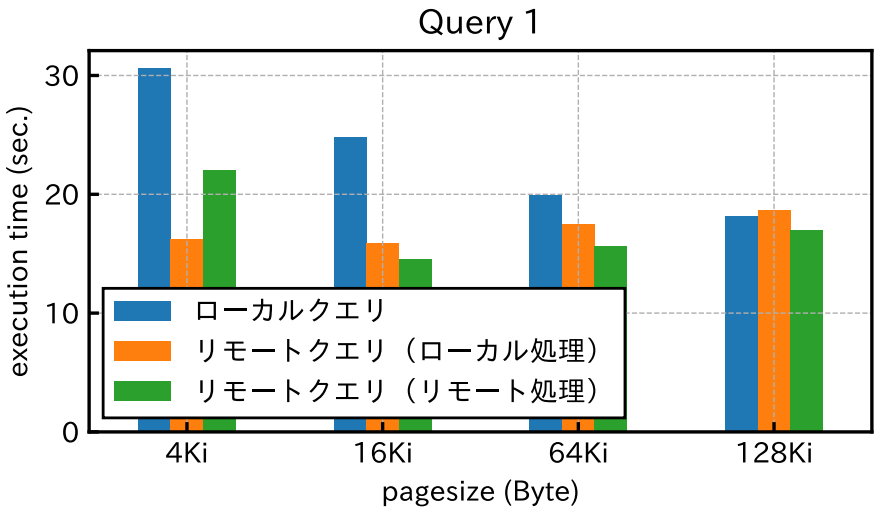

図 4.34: クエリ 1 のページサイズと最短実行時間

この理由を探るため，ページサイズが 16KiB のときの最良ケースの測定パラメータと結果を表 4.6 に示した．[4]性能が良いリモートクエリ（リモート処理）ではワーク

[4]CPU（システム）使用率は全てのケースの両サーバで常に 1%程度になっており低かったので掲載を省略した．

ロード実行中の CPU 使用率が両サーバで 100%になっており，しっかり両サーバの全コアが使い切られていることがわかる．一方，ローカルクエリ，リモートクエリ（ローカル処理）では CPU 使用率が低く，あまりコアが使われていないことがわかる．CPU 使用率が低いので IO バウンドかネットワークバウンドのワークロードであると仮定を置く．しかし，表 4.2 を見ると，ローカル IO で 0.64M IOPS，リモート IO でも 0.35M IOPS でているので，IO やネットワークがボトルネックになっているとは考えにくい．このことから，4.4.3.1 節の実験でも見られた，CPU 負荷と割り込み（IO は割り込み命令である）が共存していると適切に CPU 時間がスレッド間で渡らない現象が発生している可能性が高いと言える．

表 4.6: クエリ 1 の最良結果時の測定パラメータと測定結果．ページサイズは 16KiB

	ローカルクエリ	リモートクエリ（ローカル処理）	リモートクエリ（リモート処理）
RDMA コネクション数		1	75
IO スレッド数（コネクションあたり）	1	1	100
IOPS	2206	3442	4268
ローカルサーバの CPU（ユーザ）使用率	0.6%	5%	100%
リモートサーバの CPU（ユーザ）使用率		6%	100%
実行時間（sec.）	24.8	15.8	14.5
スループット（MB/sec.）	36	56	70

これらの結果を総合して考えると，ローカルクエリ，リモートクエリ（ローカル処理）ではマルチスレッドで多重化せずに，シングルスレッドで IO スレッド数，RDMA コネクション数ともに 1 にしておくのが性能上良いといえる．リモートクエリ（リモート処理）ではベストな性能を求める場合，多重化したほうがよいが，多重化しなくてもそれほど性能に大差はないといえる．

4.5.3 クエリ 6 の性能測定

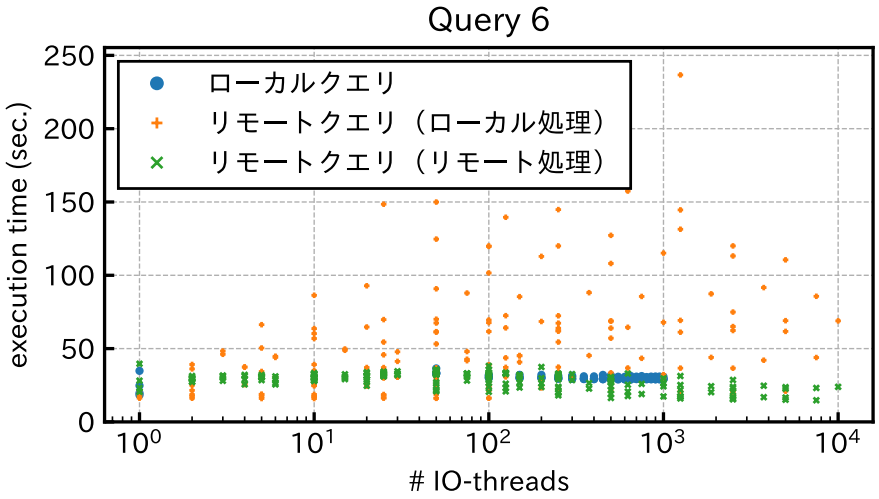

図 4.35: クエリ 6 の IO スレッド数と実行時間

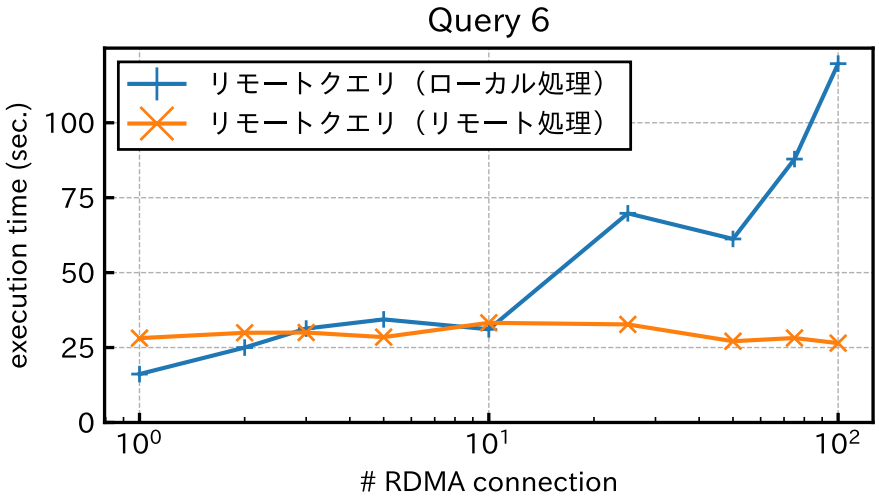

図 4.36: クエリ 6 の RDMA コネクション数と実行時間

選択率が 0.013 であるクエリ 6 を実行した結果を 4.5.2 節と同様に図 4.35，図 4.36，図 4.37 に示す．結果はクエリ 1 のときとほぼ同一であった．選択率が低いので特に

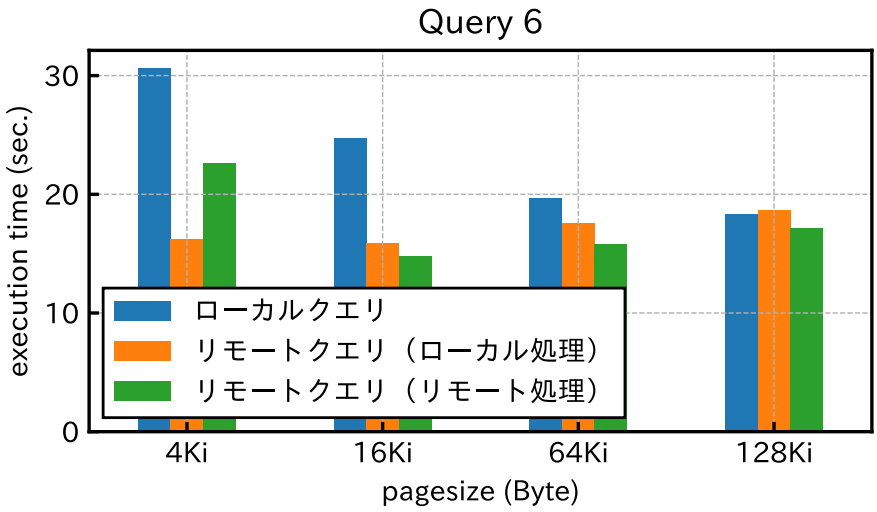

図 4.37: クエリ 6 のページサイズと最短実行時間

リモートクエリ（リモート処理）の性能に変化が見られるかと期待されたが，変化は特になかった．

4.5.4 クエリ 15 の性能測定

最後に選択率が 0.15 と中程度であるクエリ 15 を実行した結果を 4.5.2 節と同様に図 4.38，図 4.39，図 4.40 に示す．結果はやはりクエリ 1 のときとほぼ同一であった．Disaggregated storage 上でのクエリ処理の性能は選択率によらないと結論づけられる．

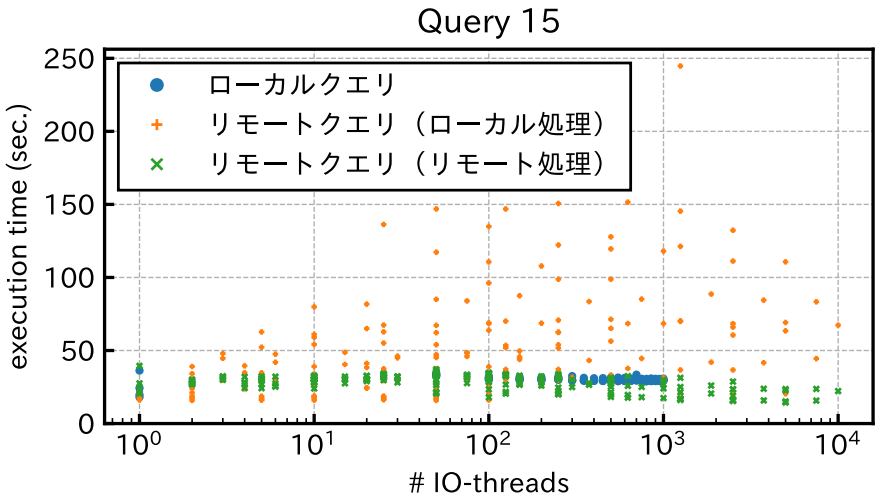

図 4.38: クエリ 15 の IO スレッド数と実行時間

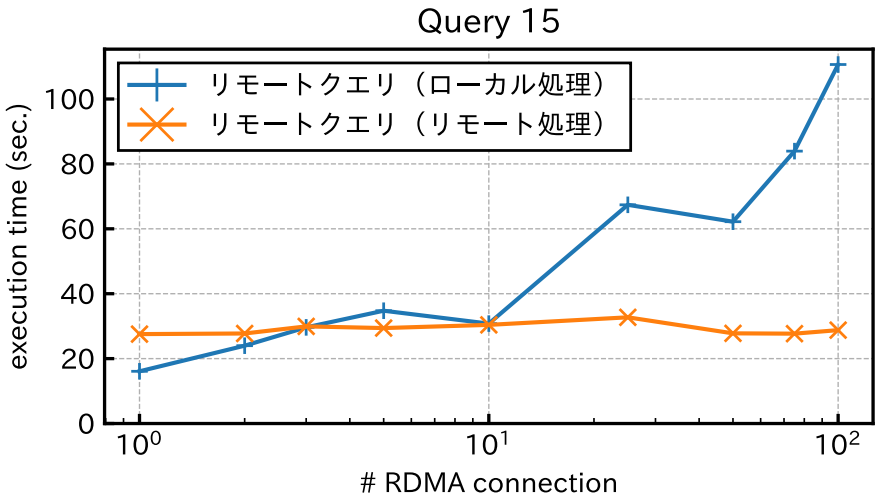

図 4.39: クエリ 15 の RDMA コネクション数と実行時間

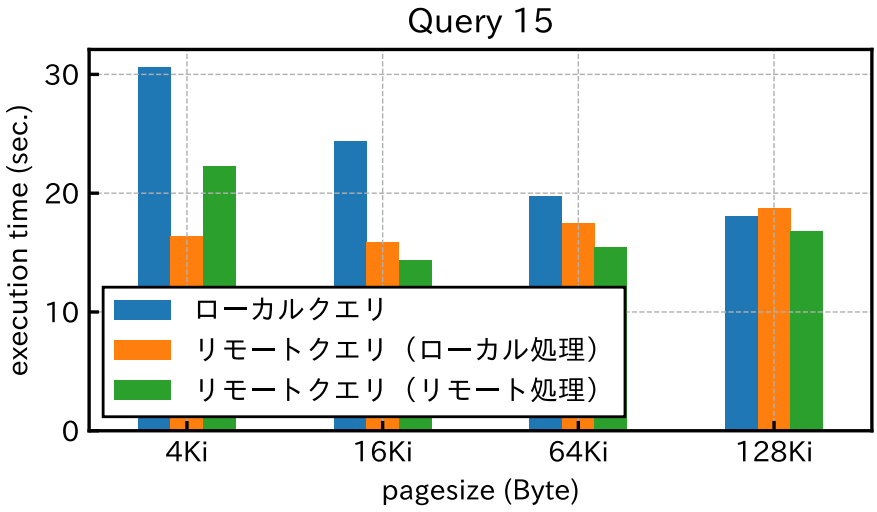

図 4.40: クエリ 15 のページサイズと最短実行時間

第5章　関連研究

1章で述べたように，RDMAを用いたリモートIOについての研究は現状なされていないが，RDMAを用いた通信（メモリ間通信）については，どのようにデータベースシステムに応用できるか数多くの研究がなされている．

例えば分散データベースシステムを構成するサーバ間でRDMA通信する際（トランザクション処理の際）に，RDMAが提供する2つのプリミティブのうち，one-sidedプリミティブとtwo-sidedプリミティブのどちらを使用すればよいかという点で活発な議論がなされている．FaRM [23]という手法では，主にone-sidedのみを使用しており，DrTM+R [24]では，完全にone-sidedのみを使用している．一方，FaSST [25]という手法では，two-sidedのみを使用している．さらにDrTM+H [19]では，分散トランザクション処理のアルゴリズムにOptimistic Concurrency Control(OCC)を採用し，OCCが4つのフェーズから成立つことに注目し，各フェーズで行われる通信の特性を考慮した上で，2つのプリミティブをハイブリッドに使い分けるという手法をとっている．本研究では，ランダムリードやTPC-Hなど汎用性が高いワークロードであったため，このような最適化の出番はなかったが，実際に特定の並列データベースシステムにRDMAのリモートIOを組み込む際は，DrTM+H [19]のような，通信内容まで考慮した最適化をかけることで，さらなる性能向上が期待できる．

Fentらはデータベースシステムのサーバとクライアント間の通信に着目した [14]．その通信路上では，SQLクエリとその結果が転送されるが，ここにRDMAを導入し大幅なパフォーマンス向上に成功している．Fentらが RDMAをデータベースシステムに導入するにあたって示した効率的なデータ構造は，IOを発行せずサーバ同

士のメモリ間の通信用途であったが，本研究のようにIOまで行う際も参考にすることが出来る．

また，RDMAを採用した分散データベースシステムではノード間通信が高速低遅延になり，アルゴリズムの前提条件が崩れるということがしばしばある．これに対し，ノード間通信を極力減らそうとする従来のアルゴリズムではなく，新たなアルゴリズムを考えてRDMAの力を最大限使い切るという研究がなされている [9–13]．例えば [34] の研究では，従来最優先事項であったノード間通信を避けるということを辞め，データレコードに対して競合が発生する時間を最小化するChillerというアルゴリズムを提案することで，パフォーマンスが2倍向上することを示した．これらの研究で対象とされているのはサーバ間通信でありメモリ上で完結する故に，ストレージまでIOを発行する本研究には直接活かすことは出来ない．しかし，本研究でリモートIOのコストが下がった結果，ローカルIOとのアクセスコストの差が減少し，Disaggregated storage（非集約型ストレージ）上でのIOのアクセスコストのバランスが従来と比べて変化した．今後これらの研究のように，変化したパワーバランスの上でアルゴリズムを再設計することで，さらなる発展が望めると考えられる．

第6章　結論

Disaggregated storage architecture（非集約型ストレージアーキテクチャ）上に並列データベースを構成する際，データがどのストレージに格納されているかによってアクセスコストが異なることがパフォーマンスを追求する上で課題であった．本研究では，まず潜在的最大性能を引き出すIOの発行方法を実験により明らかにした．特にリモートIOの場合では，RDMAとIOをリモートサーバで高速にブリッジングする必要があったがこれを達成するシステムソフトウェアを考案した．これにより，ローカルのストレージだけでなく，リモートストレージにも高速でアクセス出来るようになり，最大でローカルIOの場合5.1M IOPS，リモートIOの場合4.3M IOPSを達成し，系全体でIOのコストが低下した．さらに，ローカルIOとリモートIOのアクセスコストの差が小さくなり，特にランダムリードの場合リモートIOのIOPSはローカルIOのIOPSの半分以上になった．これにより，自由にデータをパーティショニングしても比較的高性能で任意のデータにアクセスできるようになった．

また，IOリプレイでローカルIO，リモートIOを発行する際には，両者の性能差がランダムリードより開くことがわかった．これはRDMAはキャッシュを提供しないため，IOリプレイのように，データのアクセスパターンに局所性があるワークロードであっても，キャッシュが効かず局所性の恩恵を受けないからである．今後の課題としてリモートIOにキャッシュ機構をもたせることが挙げられる．

なお本研究により得られたローカルIO，リモートIOの知見はデータベースシステム以外にも，ローカルIO，リモートIOを行うアプリケーションであればどのようなものでも適用することができ，貢献範囲は広い．

次に，これらの明らかになったIOの性能を考慮した上で，オペレータをどのようにサーバに配置してクエリ処理をするか3つの方式を提案し，実装を用いた実験を行った．リモートクエリ（リモート処理）ではローカルサーバ，リモートサーバのCPUを使い切る事ができる一方で，ローカルクエリやリモートクエリ（ローカル処理）ではCPUを使い切る事ができず，実験の結果直感に反して，多くの条件下でリモートクエリ（リモート処理）の性能がローカルクエリの性能を上回ることが明らかになった．また，ローカルクエリ，リモートクエリ（リモート処理）ではパラメータによらず性能が安定している一方で，リモートクエリ（ローカル処理）ではパラメータによっては性能が著しく低下することが明らかになった．そして，クエリのwhere句の選択率は，クエリ処理の性能にほとんど影響を与えないこともわかった．また，ページサイズが128KiBのときにはほとんどローカルクエリ，リモートクエリ（ローカル処理），リモートクエリ（リモート処理）の3方式で性能に違いがないことがわかり，クエリ処理はどのサーバで行ってもよく，非常にフレキシビリティが高いストレージアーキテクチャが実現したと言える．

今後の課題として，本研究ではリードのみを対象としていたので，ライトの方式についても検討することが挙げられる．ただし，ライトはリードと異なり，メモリ上のバッファに一時的に書き込んでおくなどの技法が存在するため，より問題は複雑になる．

また，本研究では様々な測定条件で試行することにより最適なRDMAの使用方法を探索したが，本研究では考えなかった未探索の測定条件もある．例えばRDMAを使用する際には，非常に多くのAPIが提供されているが，本研究ではそのうち代表的なものしか使用していない．また，RDMAで転送するメッセージのデータ構造は開発者が任意に決めることが出来，より効率的な実装方法が存在する可能性もある．今後，このような測定条件についてもさらなる探索がなされることも期待される．

謝辞

本研究に取り組むにあたり，多くの方々から多大なるご尽力をいただきました．私は学部ではコンピュータを用いた群知能のシミュレーションを研究しましたが，修士ではシステムソフトウェアの研究と分野を変えました．また，COVID-19 により修士課程の殆どをオンラインで過ごすという未曾有の事態でした．このような条件下でも，こうして修士論文を完成させることが出来たのは，研究室の皆様のご指導と素晴らしい研究環境があったからこそに他なりません．

まず，指導教員である豊田正史教授に深謝申し上げます．先生には，全体ミーティングなどでいつも非常に的確で参考になるアドバイスを頂いたり，研究室のサーバ環境などの手続きをしていただきました．先生のご協力がなければ研究を行うことは決して出来ませんでした．重ねてお礼申し上げます．

そして，私の選択した研究分野が最も近かった故に，最も研究を直接的に見ていただいた合田和生准教授に心より御礼申し上げます．そもそも私が修士から分野を変えたのは，先生がご紹介するデータベースシステムの世界に非常に惹かれたことが大きな理由の一つであり，先生のご存在なしでは私がこの分野に飛び込むこともなかったと思います．先生には，研究ネタ，研究の思考法，論文の読み方，実験・考察の仕方など研究を遂行する上で不可欠な事からスライドの作成，発表の技法などアウトプットする上で極めて重要なスキル，さらには，コンピュータの様々なご知見や就職・人生相談など研究外のことまで幅広くご教授いただきました．また，研究を中々進めることが出来なかったときも優しくご指導いただいたり，深夜遅くまで発表練習を見ていただいたり，休日にすぐ研究相談に乗っていただいたりと，ここには書ききれないほど本当にご厚意に預かりました．重ねて深く感謝の意を表し

ます．

早水悠登特任助教，特任研究員の小沢健史さん，川道亮治さん，堀田雅也さん，博士課程の吉岡弘隆さん，高田実佳さんには輪読会を始めとして大変お世話になりました．専門的なご見地から非常に詳しい意見を多くいただき大変参考になりました．特に共著者に入っていただいた小沢さんには，IO リプレイのログをご提供いただいたり，ミーティングで結果の考察をご一緒にしていただいたりしました．また，堀田さんには，実験用のサーバを管理していただいて困難なトラブルシューティングの際にもお助けいただきました．また，皆様の豊富なご経験とご知見によるお話は，常にとても刺激的で興味深いものばかりでした．これだけコンピュータのエキスパートの方々に囲まれた環境は人生で初めてで，この世界の奥深さを知ることができ，非常に貴重な経験をさせていただきました．皆様のご支援なしでは研究を進めることは出来ませんでした，心よりお礼申し上げます．

吉永直樹准教授，梅本和俊助教には，全体ミーティングを始めとして研究やスライド作成法などについて多くのご指導をいただき大変お世話になりました．心よりお礼申し上げます．この研究室は合同研究室であり，研究分野が遠い方からも頻繁に研究を見ていただく機会があったので，とても勉強になる良い環境でした．

喜連川優特別教授には，非常に潤沢な計算資源を始めとして最高の研究環境をご用意していただきました．このような環境で研究を行うことが出来て幸せでした．心よりお礼申し上げます．

秘書・経理の井崎葉子さん，周佐亜樹さんには，事務手続きなどでお世話になりました，感謝申し上げます。私の入学のタイミングは，COVID-19 によるオンライン化とほぼ同時期でしたが，皆様の素早いサポートのおかげで迅速に事務手続きを行うことが出来ました．

研究室の先輩方には，研究のことから研究室のことまで幅広くお世話になりました．先輩方の研究はとてもレベルが高く，いつも驚嘆しながらご発表を聞かせていただきました．

また，同期の皆さんにも大変お世話になりました．授業から研究のことまで色々教えていただいたり，定期的にオンライン飲み会をしたりと，同期の皆さんとの交

流は，修士生活の癒やしでした．結局全員と直接お会いすることは出来ませんでしたが，いつか COVID-19 が収束したら是非お会いしましょう．

また，研究室の後輩の皆さんにもお世話になりました．特に分野が近い木村元紀君にはお世話になりました．非常に知識が豊富でコンピュータに詳しいのでいつも感心させられてばかりで楽しいひとときでした．皆さんの益々のご活躍を期待しています．

この研究室で 18 年間に渡る学生生活の最後を迎えられたことは非常に幸運でした．この研究室での学びと経験は私の今後の人生において非常に役立つ有益なものでした．このような機会をご提供いただいた研究室の皆様に重ねて感謝いたします．

最後に，常に精神的な支えとなってくれた友人，遠くからいつも応援し続けてくれた両親，祖父母に感謝の意を表します．

2022 年 1 月 27 日

参考文献

[1] Alfredo Cuzzocrea, Ladjel Bellatreche, and Il-Yeol Song. Data warehousing and olap over big data: current challenges and future research directions. In *Proceedings of the sixteenth international workshop on Data warehousing and OLAP*, pp. 67–70, 2013.

[2] Hakan Özköse, Emin Sertaç Arı, and Cevriye Gencer. Yesterday, today and tomorrow of big data. *Procedia-Social and Behavioral Sciences*, Vol. 195, pp. 1042–1050, 2015.

[3] Jana Giceva and Mohammad Sadoghi. Hybrid oltp and olap., 2019.

[4] R.R. Schaller. Moore's law: past, present and future. *IEEE Spectrum*, Vol. 34, No. 6, pp. 52–59, 1997.

[5] Thomas N. Theis and H.-S. Philip Wong. The end of moore's law: A new beginning for information technology. *Computing in Science Engineering*, Vol. 19, No. 2, pp. 41–50, 2017.

[6] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. pp. 18–32, 2013.

[7] Shivnath Babu and Herodotos Herodotou. Massively parallel databases and mapreduce systems. *Foundations and Trends® in Databases*, 2013.

[8] Francisco J Hens and José M Caballero. *Triple Play: Building the converged network for IP, VoIP and IPTV*, Vol. 3. John Wiley & Sons, 2008.

[9] Dong Young Yoon, Mosharaf Chowdhury, and Barzan Mozafari. Distributed lock management with rdma: Decentralization without starvation. pp. 1571–1586. Association for Computing Machinery, 5 2018.

[10] Georgios Chatzopoulos, Aleksandar Dragojević, and Rachid Guerraoui. Spade: Tuning scale-out oltp on modern rdma clusters. pp. 80–93. Association for Computing Machinery, Inc, 11 2018.

[11] Claude Barthels, Ingo Mu ¨ ller, Konstantin Taranov, Gustavo Alonso, and Torsten Hoefler. Strong consistency is not hard to get: Twophase locking and twophase commit on thousands of cores. Vol. 12, pp. 2325–2338. VLDB Endowment, 2020.

[12] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The end of slow networks: It's time for a redesign, 2016.

[13] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The end of a myth: Distributed transactions can scale, 2017.

[14] Philipp Fent, Alexander van Renen, Andreas Kipf, Viktor Leis, Thomas Neumann, Alfons Kemper, and Friedrich-Schiller-Universität Jena. Low-latency communication for fast dbms using rdma and shared memory, 2020.

[15] Erfan Zamanian, Xiangyao Yu, Michael Stonebraker, and Tim Kraska. Rethinking database high availability with rdma networks. Vol. 12, pp. 1637–1650. VLDB Endowment, 2018.

[16] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chenz, Beng Chin Ooi, Kian Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient distributed memory management with rdma and caching. Vol. 11, pp. 1604–1617. Association for Computing Machinery, 2018.

[17] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. pp. 137–152. Association for Computing Machinery, Inc, 10 2017.

[18] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. Designing distributed tree-based index structures for fast rdma-capable networks. pp. 741–758. Association for Computing Machinery, 6 2019.

[19] Xingda Wei, Zhiyuan Dong, Rong Chen, Haibo Chen, and Shanghai Jiao Tong. *Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better!* 2018.

[20] Chao Wang, Kezhao Huang, and Xuehai Qian. Comprehensive framework of rdma-enabled concurrency control protocols. 2 2020.

[21] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. Vol. 44, pp. 295–306. Association for Computing Machinery, 2 2015.

[22] Christopher Mitchell, Yifeng Geng, Jinyang Li, and New York University. Using one-sided rdma reads to build a fast, cpu-efficient key-value store, 2013.

[23] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th symposium on operating systems principles*, pp. 54–70, 2015.

[24] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems*, pp. 1–17, 2016.

[25] Anuj Kalia, Michael Kaminsky, and David G Andersen. *FaSST: Fast, Scalable and Simple Distributed Transactions with Two-sided (RDMA) Datagram RPCs*. USENIX Association, 2016.

[26] Michael Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, Vol. 9, No. 1, pp. 4–9, 1986.

[27] Erhard Rahm. Parallel query processing in shared disk database systems. *ACM SIGMOD Record*, Vol. 22, No. 4, pp. 32–37, 1993.

[28] Glenn K Lockwood, Damian Hazen, Quincey Koziol, R Shane Canon, Katie Antypas, Jan Balewski, Nicholas Balthaser, Wahid Bhimji, James Botts, Jeff Broughton, et al. Storage 2020: A vision for the future of hpc storage. 2017.

[29] Sergey Legtchenko, Hugh Williams, Kaveh Razavi, Austin Donnelly, Richard Black, Andrew Douglas, Nathanael Cheriere, Daniel Fryer, Kai Mast, Angela Demke Brown, Ana Klimovic, Andy Slowey, and Antony Rowstron. Understanding Rack-Scale disaggregated storage. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, July 2017. USENIX Association.

[30] Gregory Kerr. Dissecting a small infiniband application using the verbs api. *arXiv preprint arXiv:1105.1827*, 2011.

[31] Kazuo Goda and Masaru Kitsuregawa. The history of storage systems. *Proceedings of the IEEE*, Vol. 100, No. Special Centennial Issue, pp. 1433–1440, 2012.

[32] Transaction Processing Performance Council (TPC). Tpc benchmarktm ™h (decision support) standard specification revision 3.0.0. `http://tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.0.pdf`, 2021.

[33] Melyssa Barata, Jorge Bernardino, and Pedro Furtado. Ycsb and tpc-h: Big data and decision support benchmarks. In *2014 IEEE International Congress on Big Data*, pp. 800–801, 2014.

[34] Erfan Zamanian, Julian Shun, Carsten Binnig, and Tim Kraska. Chiller: Contention-centric transaction execution and data partitioning for modern networks. pp. 511–526. Association for Computing Machinery, 6 2020.

発表文献

国内会議

1. 加藤 滉貴，小沢 健史，合田 和生，喜連川 優.
   RDMA を用いたリモート入出力性能の実験的考察
   第 13 回データ工学と情報マネジメントに関するフォーラム（DEIM2021），B25-3，2021.

2. 加藤 滉貴，小沢 健史，合田 和生，喜連川 優.
   並列データベースシステムに於ける RDMA を用いたリモート入出力性能の検討
   信学技報（データ工学研究会），vol. 121，no. 314，DE2021-17，pp. 13-18，2021.

3. 加藤 滉貴，小沢 健史，合田 和生，喜連川 優.
   並列データベースシステムに於ける RDMA を用いたリモート入出力性能の測定と問合せ処理への影響
   第 14 回データ工学と情報マネジメントに関するフォーラム（DEIM2022），2022（予定）.