学位論文　博士(工学)

構造型ストレージ向け複数GPU間分散キャッシュの研究

平成29年度

慶應義塾大学大学院理工学研究科

森島　信

論　文　要　旨

近年の情報通信技術やセンシング技術の発展によって生成されるようになった多様な情報を効率的に保存、管理するためのシステムとして、特定用途に特化しており、高い拡張性とスループットを持つ構造型ストレージが広く用いられている。構造型ストレージは取り扱うデータの種類や用途に合わせて様々な実装が提案されているが、データ構造によって、KVS(Key-Value Store) 型、ドキュメント指向型、グラフ型の 3 つに分類できる。構造型ストレージでは、ドキュメント指向型の正規表現探索やグラフ型のグラフ探索といった特に計算量の大きいクエリを中心にボトルネックとなる。これらの処理は並列化可能であり、近年並列処理性能の向上が著しい GPU を用いることで高速化できる。しかし、構造型ストレージは、データ構造が GPU で処理することを考慮されておらず、そのまま GPU を用いて高速化するのは困難である。そこで本論文では、ドキュメント指向型、グラフ型それぞれの構造型ストレージに対して、メモリ利用効率が良く、GPU 処理に適したデータ構造のキャッシュを提案する。このキャッシュに対して正規表現探索やグラフ探索処理を行うことで、各システムの枠組みを維持しつつ、ドキュメント指向型、グラフ型ストアの高速化を達成した。GPU を用いたキャッシュ手法の問題点として、GPU のデバイスメモリによる制約があげられる。GPU のメモリ容量は、ホストメモリよりも小さく、GPU のメモリ容量を超える量のデータを扱う場合には、複数回に分けて GPU へ転送、計算を繰り返さなければならず、高速化率が大幅に減少する。この問題点を解決するために、本論文では、提案するキャッシュ機構を 10GbE で遠隔接続された複数の GPU へ拡張し、より多くの GPU による分散キャッシュとすることで、GPU のメモリ容量を超える量のデータにも対応可能にする。この際、グラフの非同期更新とドキュメントのハッシュを用いた分散によって転送と計算の重複実行をすることを提案し、転送遅延による性能低下を抑えつつ、遠隔 GPU への分散キャッシュの拡張を可能にする。

評価では、各種構造型ストレージと本提案手法の 3 台の遠隔 GPU によるキャッシュとの性能比較を行った。ドキュメント指向型の評価では、文字列の正規表現探索をドキュメント 1 件あたり 128 文字、1 千万件のドキュメントに対して実行し、提案手法はドキュメント指向型ストアに対して 75.0 倍のスループットを達成した。また、グラフ型の評価では、単一始点最短経路問題を頂点数 320 万、次数 100 のグラフに対して実行した際、提案手法はグラフ型ストアに対して 383.2 倍のスループットを達成し、本提案手法により構造型ストレージを大幅に高速化できることを示した。これらの評価では、正規表現探索に関してはクエリ実行中の GPU 間同期が発生しないため、GPU 数の増加による並列度の増加に伴ってスループットが向上した。グラフ探索では、クエリ中に同期が発生するため、単純に GPU 数を増やしただけでは性能向上を見込めないが、本論文では、同期オーバーヘッドを削減する手法を提案することで、GPU を複数用いた場合のスループットの向上を達成した。

Thesis Abstract

A Study on Distributed Cache over GPUs for Structured Storage

Structured storages are widely used for storing and managing various data made by telecommunications and sensing devices. They provide high scalability and throughput for some specific application domains. They are classified into three types by data structure: KVS (Key-Value Store), document-oriented store, and graph store. Performance bottlenecks in structured storages are related to a certain class of queries that requires high computation cost such as regular expression search on document-oriented store and graph search on graph store. These queries can be parallelized and accelerated by GPUs whose parallel processing performance is significantly improving recent years. This thesis proposes a cache structure suitable for GPU processing with high memory efficiency for each structured storage. A GPU performs regular expression search or graph search for the proposed cache so that these queries are accelerated without largely changing the original structured storage framework. The proposed acceleration method using cache can be applied to various applications using structure storages. The problem of GPU processing is a restriction of memory capacity of GPU device memory. The capacity is smaller than a host memory. When a cache size is larger than a GPU device memory, performance improvement by GPU processing drastically decreases, because multiple CPU-GPU data transfers and computations are needed. To tackle this issue, the proposed cache is extended for multiple GPUs where GPU devices are connected via 10GbE so that the caches are distributed to multiple GPUs and increase the capacity of device memory. This thesis proposes an overlapped execution of computation and data transfer by asynchronous update of graph and document distribution using a hash structure. These methods can be extended to distributed cache for remote GPUs while suppressing a performance degradation by data transfer.

The proposed distributed cache using three remote GPUs is evaluated in terms of query throughputs. A query that performs regular expression search from 10 M documents is accelerated by 75.0x compared to MongoDB. A query that performs single source shortest path search in graph store for a large graph with 3.2M nodes and 100 degree is accelerated by 383.2x compared to Neo4j. The results demonstrate that the proposed cache can accelerate these structured storages. In regular expression matching, because there is no synchronization between multiple GPUs during a query, the throughput increases as the number of GPU increases. In graph search, since synchronizations are required during queries, performance cannot be simply improved by increasing the number of GPUs. In this thesis, by proposing a method to reduce synchronization overhead, we achieved throughput improvement when using multiple GPUs.

目 次

図 目 次

表 目 次

第 1 章

緒論

1.1 背景

近年、携帯機器の普及や発展途上国への情報機器の普及に伴い、人々が生み出す情報量は年々指数関数的に増加している。さらに、各種センサー、家電、自動車等、これまでは情報通信を行わなかった各種機器に通信機能を持たせ、インターネットに接続する IoT (Internet of Things) [2] の普及も進んでいる。これらの物が自動的に生成する情報量も増加傾向にあり、それらの情報の種類もグラフ構造、画像、動画といった多様な情報となっている。また、情報量の増加傾向は今後も継続すると考えられており、IDC 社は、世界のデータ量は 2013 年では、4.4ZB であったが、2020 年には 10 倍の 44ZB に拡大すると予測している [3]。このような情報量の増加によって、今までは扱ってこなかったような膨大な量の情報や様々なデータ構造の情報を処理する必要が生じ、このような情報を処理するシステムの重要性が高まっている。

現在、これらの情報を蓄積、処理するシステムの基盤として用いられているのは、RDBMS (Relational DataBase Management System) というデータベースシステムである [4]。RDBMS はデータを n 項関係で表現、管理するもので、SQL という問い合わせ言語を用いて柔軟なデータベース管理を行う事ができる。また、データベースの更新の際に、データの不可分性、一貫性、独立性、永続性を保証するトランザクション処理を行うことで、データの整合性を厳密に保っている。これらの特徴は、特に、金銭の授受等のデータの厳密な整合性を必要とする用途において重要な特徴であり、現在の社会の基盤となる様々なシステムに利用されている。

一方で、近年普及が著しく進んだ SNS(Social Networking Service) [5] 等のアプリケーションにおいては、データの整合性よりも大量のデータを処理するための水平拡張性や大量のクエリを処理できるスループットが要求される。このようなアプリケーションに対応するために、RDBMS に加えて、構造型ストレージと呼ばれるシステムがデータの管理、運用に用いられている。構造型ストレージは、対象とするアプリケーションの範囲を絞り、アプリケーションの特徴に合わせて機能を単純な読み書きのみに絞ったり、データの整合性を緩めることで、対象とするアプリケーションにおいて RDBMS を越えるスループットを達成している [6][7][8]。構造型ストレージは、RDBMS のように汎用性に優れていないため、その使用用途に応じて、様々な実装が存在するが、データ構造によって、KVS (Key-Value Store) 型、ドキュメント指向型、グラフ型の 3 種類に分類できる。それぞれの構造型ストレージは、特定の用途のために単独で用いられることもあるが、現在、異なる複数種のデータ構造の構造型ストレージを組み合わせて用いる事で、より汎用的にビックデータ処理に利用するポリグロット永続化という利用方法が注目されている [9]。

計算機科学において大きな関心事の一つに、トランジスタの微細化の限界という問題があげられる。これまでの計算機科学は、トランジスタの指数関数的な微細化と共にあり、それにともなって CPU の性能も指数関数的に向上しつづけてきた。しかし、トランジスタの微細化は、近い将来、分子の大きさという物理的な壁によって限界を迎えると考えられている。また、物理的な壁に

直面していない現在においても、CPU の冷却能力に起因する電力の壁によって CPU の動作周波数の向上は頭打ちとなっており、複数のコアによる並列処理によって性能向上を図っている [10]。このような現状において、システムの性能を向上させる手法として注目を集めているのが、アクセラレータの利用である。アクセラレータは CPU と比較すると一部の用途に特化しているが、その用途においては高い性能を発揮できる。そのようなアクセラレータの一つとして、画像処理に特化した GPU(Graphics Processing Unit) があげられる。GPU は、画像を構成する多数のピクセルを並列に処理するため、多数のコアを有しており、それらを活かした並列処理に適したアクセラレータである。そのため、画像処理以外にも、並列処理によって性能向上可能な用途に応用することができ、様々な用途に GPU を用いる GPU による汎用計算 (GPGPU:General-purpose computing on GPU) として用いられている。また、GPU 以外のアクセラレータとしては、特定の用途に合わせて設計される回路である ASIC(Application Specific Integrated Circuit) や、製造後に構成を設定できる回路である FPGA(Field-Programmable Gate Array) があげられる。今後は、従来の CPU のみを用いたシステムでは、性能向上率が微細化の限界に伴って低下することが予測され、このような様々なアクセラレータと CPU を組み合わせて利用することによって性能向上を図る必要がある。

このようなアクセラレータを導入する流れは、構造型ストレージの導入の流れと一致した考え方によってもたらされている。特定の処理に適したアクセラレータを用いるのと同様に、構造型ストレージは用途に特化したデータ構造や機能の絞り込みを行っている。また、特定の処理に適した様々なアクセラレータと CPU を併用してシステム全体の性能向上を図るという手法と、用途に特化した構造型ストレージを組み合わせるポリグロット永続化の手法も共通した考え方であり、特定用途に特化するものを組み合わせてシステムの高速化を行う手法が現在の計算機科学において重要視されていることがわかる。今後は、汎用性の非常に高い RDBMS とポリグロット永続化が CPU とアクセラレータの関係と同様に、併用されていくと考えられる。

また、このような構造型ストレージやポリグロット永続化は、SNS 等のアプリケーションや検索エンジンに用いられており、それらのシステムは多数の計算機の集合であるデータセンターで運用されている。世界で生成されるデータ量の増加に伴って構造型ストレージで扱うべきデータ量も増加し、それに伴ってクエリの計算量も増加するため、クエリの処理のために必要な計算機の数も増大すると予測される。近年、データセンターの需要の高まりによって世界全体でのデータセンターにおける消費電力も増加しており、2014 年の時点で世界全体の電力消費量の 1.6%を占めると推計されている [11]。この電力消費量は、データセンターの運用費用のみならず、火力発電による大気汚染等の問題の一因ともなっており、消費量の抑制が求められている。データセンターの電力消費量の削減手法として、排熱効率の向上による冷却の電力消費を削減する手法が考えられるが、既に最新のデータセンターではデータセンター全体の電力消費量を IT 機器の電力消費利用で割った電力効率 (PUE:Power Usage Effectiveness) は 1.1 と理論値の 1.0 に近づいており、排熱効率の更なる効率化によって削減される電力消費量は限られている [11]。また、プロセスの微細化に伴うプロセッサの電力効率の向上という観点においても、分子の大きさという物理的な限界やリーク電力の問題によって電力効率の向上率が鈍化傾向にある。本論文では、GPU を用いて構造型ストレージのクエリを効率的に処理する手法を提案し、それにより構造型ストレージで要求される高いスループットを達成することで、データセンターの効率化にも貢献する。

1.2 研究目的

背景で述べたとおり、構造型ストレージとアクセラレータの利用において特定用途特化という同様の傾向にあると言える。今後はそれらを組み合わせ、特定用途に特化したソフトウェアのうち、アクセラレータで高速化できる箇所はアクセラレータを用いるという方向へ発展すると考え

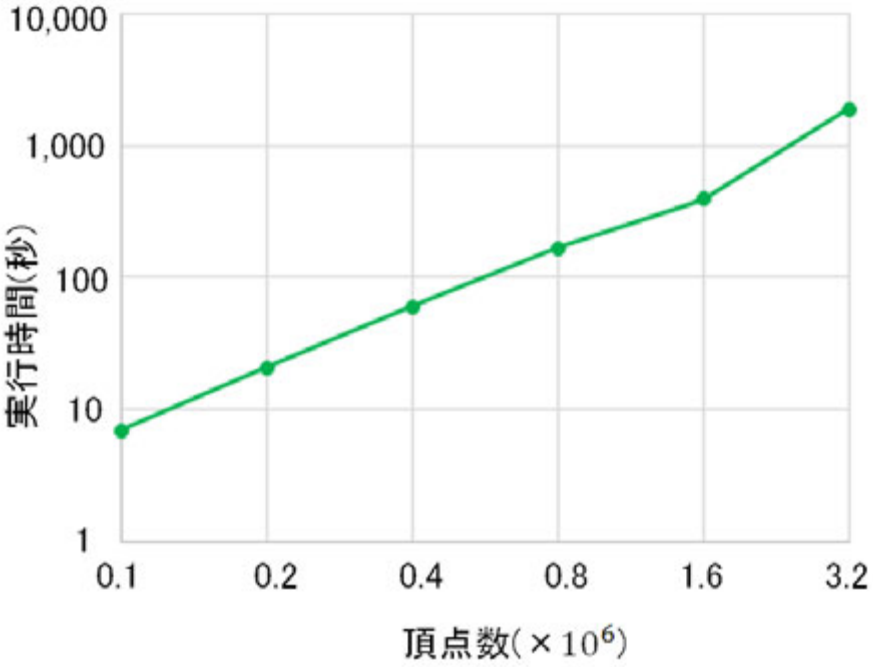


図 1.1: グラフ型ストアにおける単一始点最短経路問題の実行時間

られる。また、需要の高まりに対処するためには、その方向への発展が不可欠であるとも言える。構造型ストレージにおいては、計算量の大きい正規表現探索やグラフ探索がボトルネックとなり、それらの処理は GPU を用いた並列処理によって高速化可能である。しかし、現状の多くの構造型ストレージはアクセラレータを用いることを考慮されておらず、GPU 処理をそのまま適用するのは困難である。そこで、本論文では、構造型ストレージから GPU 処理に用いるデータのみを抽出して GPU に適した構造のキャッシュを作成し、キャッシュに対して GPU 処理を行うことで構造型ストレージを高速化することを提案する。その際には、特定用途に特化するという考え方に則り、GPU 処理を考慮した上で、キャッシュの構造はそれぞれの構造型ストレージ毎に、それぞれの探索に特化したデータ構造とすることで、探索の効率化を図る。また、キャッシュを GPU に保持することを考えた場合には、GPU の特性として、処理に適したデータ構造のみではなく、GPU のデバイスメモリの容量の制約を考慮する必要がある。一般的に GPU のデバイスメモリの容量はホストメモリの容量よりも小さく、デバイスメモリを越える大きさのデータを扱う場合、処理を行う度にデータの転送を繰り返す必要があり、オーバーヘッドが大きい。そこで、本論文では、単に GPU を用いるだけでなく、10Gb Ethernet を経由したネットワーク経由で複数の GPU を用いることで全体としてメモリ容量を確保し、この問題に対処する。

構造型ストレージのクエリの一連の処理のうち、GPU 処理に適しており、かつ計算量の大きい処理が含まれるのは、各種探索処理である。特に、複数種の構造型ストレージを組み合わせて用いるポリグロット永続化では、構造型ストレージの中で計算量の大きい探索を行う構造型ストレージがボトルネックとなる。KVS 型、カラム指向型の処理は、単純な Put、Get 操作が主な処理であり、計算量は小さい。一方で、ドキュメント指向型における正規表現探索、グラフ型におけるグラフ探索は計算量が大きく、かつ、データ量が大きくなるにつれさらに計算量が大きくなる。特に計算量の大きいグラフ型におけるグラフ探索の例として、図 1.1 は、次数 100 のランダムグラフにおける単一始点最短経路問題を代表的なグラフ型ストアである Neo4j で実行した際の実行時間を示す両対数グラフである。図から、頂点数が大きくなるにつれて実行時間が増し、頂点数 320 万の時は 1,914 秒と非常に長い時間がかかることが分かる。このような結果からも、構

造型ストレージの性能はデータ量が大きい場合における処理性能が十分とはいえず、構造型ストレージの拡張性を高めるために、構造型ストレージにおける探索を高速化する必要がある。

そこで、本論文では、構造型ストレージのうち、ドキュメント指向型とグラフ型を主な対象とし、それぞれの構造型ストレージの枠組みを維持しつつ GPU を用いて高速化することを主目的とする。さらに、高速化に用いた手法の実問題に対する応用やポリグロット永続化への応用方法を示すことで、本論文で提案する手法が構造型ストレージ全体および構造型ストレージを用いる様々な実問題に適用できることを示す。

1.3 本論文の貢献

構造型ストレージは、データの管理に適したデータ構造をしており、GPU で処理することを考慮されたデータ構造ではないため、データをそのまま GPU へ転送して処理することは困難である。そこで、本論文では、上述した目的の達成のために、メモリ利用効率が良く、かつ GPU 処理に適したキャッシュを構造型ストレージに設けることを提案する。キャッシュに対して、GPU で文字列探索やグラフ探索処理等の探索処理を行うことにより、それぞれの枠組みを維持しつつ、ドキュメント指向型、グラフ型ストアを含む構造型ストレージを高速化できる。このキャッシュの実現に際して、本論文の貢献は以下の通りである。

1) ドキュメント指向型、グラフ型それぞれにおいて各構造型ストレージのデータ構造を GPU 処理に適した配列構造のキャッシュにデータを抽出する手法を提案した。

2)GPU 処理では、GPU のメモリ容量がホストメモリよりも小さく、GPU のメモリ容量を越える大きさのデータを扱う場合には複数回に分けて GPU への転送、計算を繰り返すため、転送オーバーヘッドが大きくなる。本論文では、提案するキャッシュ機構を 10GbE で遠隔接続する複数の GPU に拡張し、複数の GPU へ分散してデータを保持させることを提案し、GPU のデバイスメモリを越える大きさのデータもキャッシュにより効率的に処理可能にした。

3)10GbE で遠隔接続された GPU では、複数 GPU 間の同期や結果の転送等のオーバーヘッドが大きくなる。本論文では、ドキュメント指向型ではハッシュ機構を用いたキャッシュの分散手法、グラフ型では非同期グラフ処理を用いた同期手法をそれぞれ提案することで、このオーバーヘッドを削減しつつ、複数の遠隔 GPU への拡張を可能にした。

4) 本論文で提案するキャッシュ手法を応用することで、KVS を用いたブロックチェーン検索の高速化を実現し、提案手法が構造型ストレージを用いる様々な実問題に応用可能であることを示した。また、提案したキャッシュの組み合わせによるポリグロット永続化への応用方法を示した。

また、構造型ストレージのアクセラレータによる高速化という大きな観点からみると、本論文の貢献はアクセラレータの一種である GPU を構造型ストレージに適用可能にしたといえる。将来的には、本論文で対象とする GPU だけではなく、FPGA NIC などその他のアクセラレータを用いた手法と本論文の手法が組み合わせて利用され、さらに用途に応じて多種のアクセラレータやデータ構造を併用されるようになると考えている。GPU は並列処理に適したアクセラレータであるため、計算ネックかつ並列化可能な処理に適している。一方で、FPGA NIC は I/O ネックの処理に適しており、これらを組み合わせる事で I/O ネック、計算ネックの双方の処理を高速化できる。

図 1.2 に大きな観点から見た本論文の対象を示す。ここまでに述べたように、複数の構造型ストレージは、組み合わされてポリグロット永続化として様々なアプリケーションが実行しており、各構造型ストレージの処理に特化したアクセラレータを組み合わせることで高速化がなされるようになると考えており、本研究ではそのうちの GPU による構造型ストレージの高速化を対象とする。

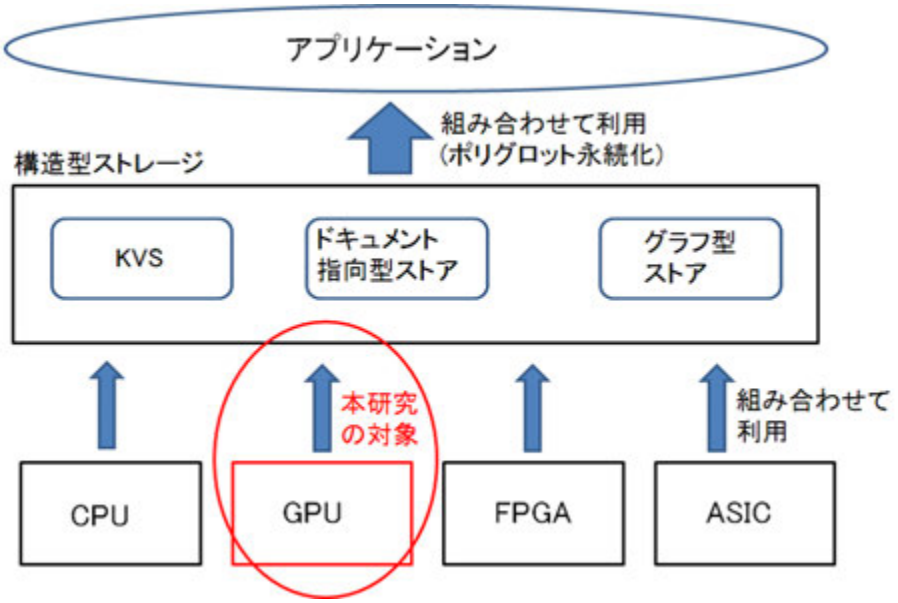


図 1.2: 本研究の位置付け

1.4 本論文の構成

本論文の構成は以下の通りである。2 章では、本研究の関連研究を述べる。3 章、4 章では、本研究の主な対象とするドキュメント指向型ストアとグラフ型ストアについてそれぞれ述べる。5 章、6 章では、ドキュメント指向型ストア、グラフ型ストアそれぞれに対する分散キャッシュ手法について述べる。7 章では、本提案手法の実応用として、キーバリューストアを用いたブロックチェーン検索への応用を述べる。8 章では、本研究のポリグロット永続化への応用について延べ、9 章で本論文をまとめる。

第 2 章

関連研究

近年の情報量の増加に対処するために、既存の RDBMS に加えて、特定用途に特化しているが、スループットの高い構造型ストレージの利用が進んでいる。また、ハードウェアの観点からも、汎用的な CPU に加えて、特定用途に特化した GPU、FPGA、ASIC などの各種ハードウェアを用いることでシステムの性能を向上させる手法が広がりをみせている。今後はこれらのソフトウェア、ハードウェア双方の手法が併用されていくと見られ、本論文はその先駆けとして各種構造型ストレージに GPU に特化したデータ構造のキャッシュを設けることで、GPU を用いて各種構造型ストレージの高速化を行う。本章では、本論文の前提となる各々の要素である構造型ストレージ、GPU の構造および GPU を用いた各種アルゴリズムの高速化、遠隔 GPU 接続等の関連研究を紹介する。

2.1 構造型ストレージ

2.1.1 構造型ストレージの分類

構造型ストレージはデータ構造によって KVS 型、ドキュメント指向型、グラフ型に分類され、用途に応じて適切な構造型ストレージを選択して用いることで、高い性能を発揮する事ができる [12]。この節では、それぞれの構造型ストレージのデータ構造と特徴を述べる。

図 2.1 は、KVS 型のデータ構造を示している。KVS 型のデータ構造は図 2.1 のように、キーとバリューが 1 対 1 で対応する組となる単純な構造である。キーはバリューを見分けるための識別子であり、バリューは個々のキーによって識別されるデータそのものである。新しくデータを加える場合、キーとバリューの組でデータを追加する。図 2.1 では、下にデータが追加されて行き、縦方向にデータが増えることになる。

データ構造だけでなく、実装されている操作も単純であり、データをバリューに書き込む SET 操作とデータをバリューから取り出す GET 操作という 2 つの操作が主である。そのデータ構造と操作の単純さ故に汎用性は乏しいが、高速であり分散化が容易であるため拡張性に優れている。

KVS 型の代表的な実装には Memcached [13] があげられる。Memcached はメモリ上で動作する揮発性の KVS であり、システムのキャッシュとして広く用いられている [14][15]。メモリ上で動作し、かつ単純な操作のみの KVS 型である Memcached は高速であるため、一度 Memcached にキャッシュされたデータは高速にアクセスできるようになり、ヒット率の高いワークロード [16] に適用すれば、システム全体の性能を向上することができる。また、不揮発性の KVS としては、Dynamo [17] があげられる。

また、KVS 型をキーとバリューの関係を 1 対 1 から 1 対多に拡張することで、より汎用性を高めることができる。そのように拡張されたシステムは、カラム指向型ストアとも呼ばれる。図 2.2 はカラム指向型のデータ構造を示している。カラム指向型は KVS 型を拡張したデータ構造をもっ

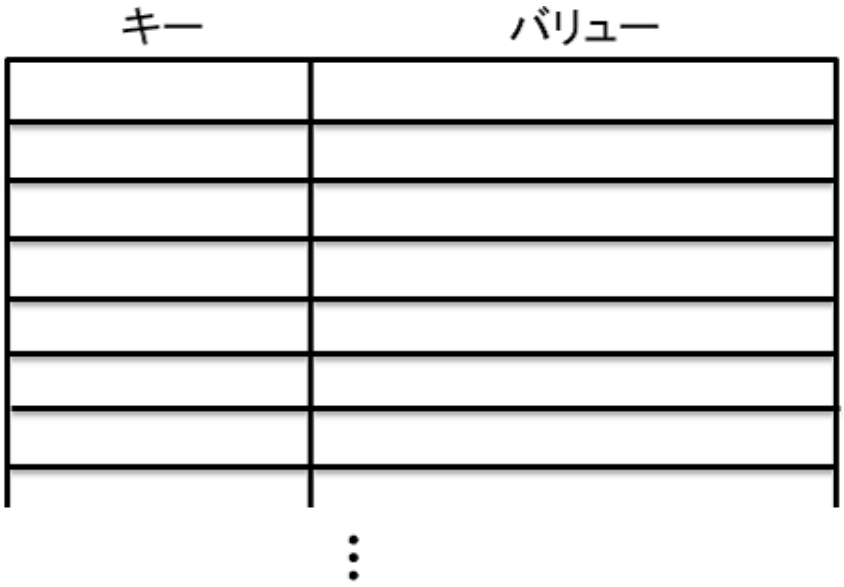


図 2.1: KVS 型のデータ構造

ており、キーはKVS 型と同じであるが、それに対応するものがバリューからカラムに変わっている。カラムとは、列という意味で、一つのカラムに複数のバリューを含めることができる。これにより、キーとバリューの関係が 1 対 1 から 1 対多に拡張されている。また、図 2.2 からもわかるとおり、カラムに含まれるバリューの数は一定ではなく、一行毎に異なる。さらに、カラムが持つバリューの数はカラム生成時に設定するものではなく、構造型ストレージ運用中に増やす事ができる。すなわち、データの追加は、KVS 型のようにキーとカラムを追加する方法と、カラムの持つバリューを増やすという 2 つの方法が行える。図 2.2 では、キーとカラムの追加で縦、カラムのバリュー追加で横と 2 方向にデータが増えることになる。

データ構造は KVS 型よりも複雑になっているが、1 つのキーとカラムの組を複数のキーとバリューの組に変形することにより、キーとバリューで表す事ができる。

図 2.3 は、カラム指向型のデータ構造を変形してキーとバリューの 1 対 1 の組で表現する例を示している。カラムにつき一つだけだったキーをバリュー毎に変えることでデータ構造を変形する。変形により、内部的には KVS 型と同様のデータ構造になるため、KVS 型と同様の実装で複雑なデータ構造を扱いつつ、KVS 型の利点も享受することができる。

実行可能なデータ操作も KVS 型から拡張されており、KVS 型の SET、GET 操作に加え、1 度にカラム全体に対して操作を行う、カラムの一部に対して操作を行う等、操作を複数のバリューにまたがって範囲を指定して行う事ができる。

代表的なカラム指向型ストアには、HBase [18] や Cassandra [19] や Bigtable [20] があり、大量のデータを扱う検索エンジン等で用いられている。

ドキュメント指向型ストアは、あらかじめ決められた枠組みに合わせて利用者がデータを追加するのではなく、ドキュメント毎に異なるフィールドを持つドキュメントを保存し、それに対する探索クエリを実行できるシステムであり、スキーマレス構造とも呼ばれている。ドキュメント指向型ストアのクエリは、ドキュメントの内容を指定して探索を行う等、KVS 型やカラム指向型よりも複雑なデータ操作を行う事が出来る。

代表的なドキュメント指向型ストアには、MongoDB [21] や CouchDB [22] がある。Web サイトの一つの記事をドキュメントとして使用するなど、ドキュメント指向のデータ構造を自然に利用できるため、Web アプリケーション等によく用いられる。ドキュメント指向型は、キーでドキュ

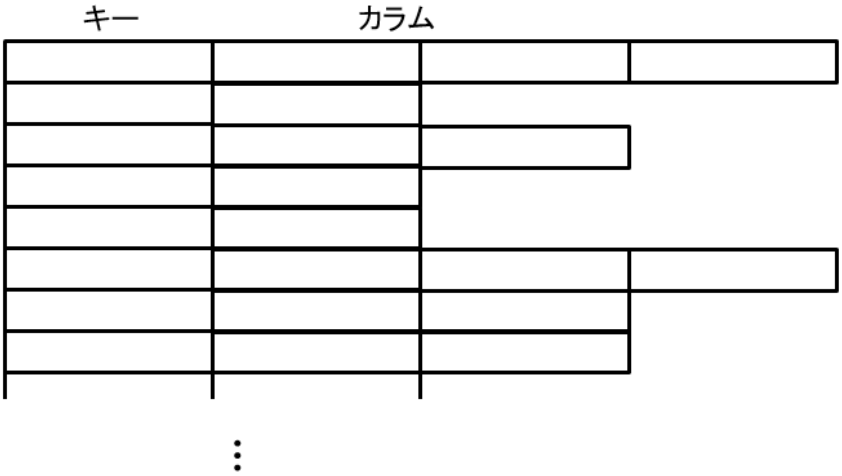


図 2.2: カラム指向型のデータ構造

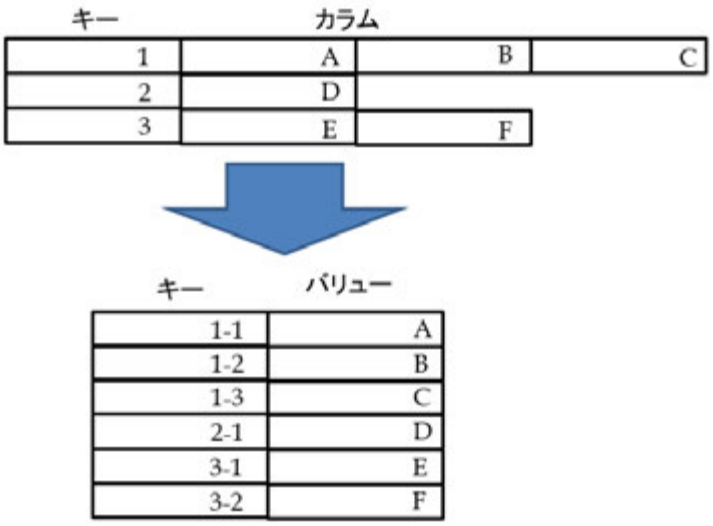


図 2.3: カラム指向型のデータ構造の変形

メントを参照するデータ構造になっている。ドキュメントは、それぞれの構造型ストレージの独自のバイナリ形式で記述される。ドキュメントの内容の条件を指定して検索を行う等、KVS 型やカラム指向型よりも複雑なデータ操作を行う事ができる。

ドキュメント指向型については、2.2 節で詳しく述べる。

グラフ型ストアは、頂点、辺、属性の 3 つの基本構成要素により、オブジェクトである頂点間の関係性を表現する。RDBMS や他の構造型ストレージでも 1 対 1 の関係性を表現することは容易であるが、多対多の複雑な関係を表現するのは非常に難しい。しかし、グラフ型ストアならば、頂点間の関係性がいかに複雑になっても、辺を増やすだけで容易に対応することができる。

グラフ型ストアでは、データの読み書きの他に、グラフ探索を行ってノード間の関係を探索できる。グラフ型ストアについては 2.3 節で詳しく述べる。

2.1.2 構造型ストレージのクエリの計算量

複数種の構造型ストレージを組み合わせて用いるポリグロット永続化においては、組み合わせたシステムのうち、最も計算量の大きいシステムがボトルネックとなる。そのため、それぞれの構造型ストレージのクエリの計算量を把握することで、どのシステムのどのクエリがボトルネックとなり得るかを把握できる。

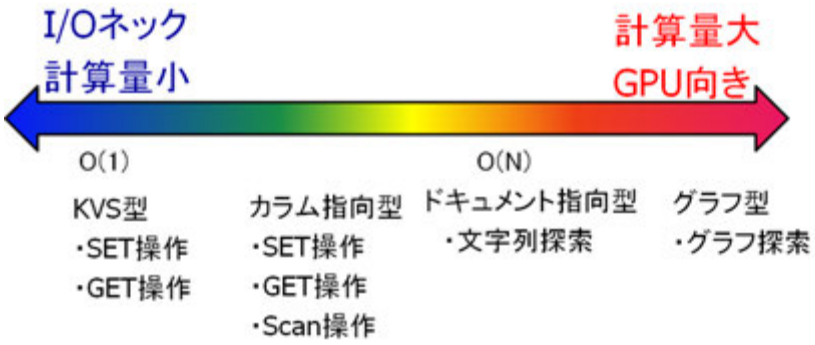


図 2.4: 構造型ストレージのクエリの計算量

図 2.4 に、構造型ストレージのクエリと、その計算量を示す。図中、及び以下の記述のオーダ表記に用いる N はデータ件数とする。この図は、計算量の大小を線表で表したものであり、左側のクエリほど計算量が小さく、I/O ネックのクエリであり、右側のクエリほど計算量が大きく計算ネックのクエリである事を表す。本研究で扱う GPU は、並列性を活かした計算量の大きい処理に適しており、右側のクエリほど GPU 処理向きのクエリであると言える。それぞれの構造型ストレージの主なクエリとして、KVS 型は Put、Get 操作およびカラム指向型への拡張も考慮すると Scan 操作があげられる。また、ドキュメント指向型は文字列探索、グラフ型はグラフ探索があげられる。

KVS 型の Put、Get 操作の計算は、キーを基にバリューがどこにあるかを探索する処理である。KVS 型のキーはハッシュを用いて管理されているため、計算量はハッシュの探索の計算量である $O(1)$ となる。カラム指向型へ拡張する際の範囲検索である Scan 操作に関しても、開始位置と終了位置のキーをそれぞれ探索する処理である。この拡張を行う際には、範囲検索を開始位置と終了位置の計算のみで行えるよう、キーは並び替えが行われ、保存されているキーは全て並び替え済みである。KVS 型と同様にキーのハッシュを用いて探索を行う場合は、カラム指向型のクエリの計算量は KVS 型と同様に $O(1)$ である。また、ハッシュを使わずに、並び替え済みのデータを二分探索で探索する場合には、$O(\log N)$ となる。

ドキュメント指向型の文字列探索は、KVS 型とは異なり、キーではなくバリューに条件を与え、その条件を満たすかどうかで探索を行う。そのため、ドキュメント毎にバリューが条件を満たしているかどうかの比較演算を行う必要がある。よって、文字列探索の計算量は $O(N)$ となる。2.2.2 節で後述するが、インデックスを使うことで、文字列探索の一部のクエリの計算量を削減し、計算量のオーダーを $O(\log N)$ にすることができる。しかし、削減できるクエリは一部であるため、全体の計算量のオーダーは $O(N)$ である。

グラフ型のグラフ探索の計算量は、実行するグラフ探索アルゴリズムの計算量に一致する。各アルゴリズム毎に計算量のオーダーが異なるため、一般的にグラフ探索のオーダーを表記することはできないが、グラフ全体を探索するアルゴリズムの内、計算量の比較的小さい幅優先探索や深さ優先探索であっても計算量は辺の数を N とした時 $O(N)$ である。そのため、図 2.4 では $O(N)$ 以上とし、ドキュメント指向型の右側に配置した。

$O(1)$ や $O(\log N)$ の KVS 型では、データ量が大きくなっても計算量の増加は少ないが、$O(N)$ のドキュメント指向型およびグラフ型は、データ量が大きくなるにつれ計算量が大きくなるため、1 つの GPU のデバイスメモリに収まらないような大量のデータを扱う場合には、計算量が非常に大きくなり、ボトルネックとなる。よって本論文では、これらのボトルネックの解消のため、ドキュメント指向型の文字列探索、グラフ型のグラフ探索を対象に GPU による高速化を行い、さらにそれを複数の遠隔 GPU へ分散させることを提案する。また、KVS 型においても、要求されるクエリのスループットが大きい場合、クエリのキーの検索がボトルネックとなりうる。本論文では、そのような場合の一つである応用例として、ブロックチェーン検索を対象に、本論文のキャッシュによる高速化を適用する。

2.2 ドキュメント指向型ストア

ドキュメント指向型ストアには、MongoDB[21] や CouchDB[22] などがあげられるが、本論文では、最も代表的なドキュメント指向型ストアである MongoDB を対象とする。

2.2.1 MongoDB のデータ構造

MongoDB では、データ記述言語 JSON(JavaScript Object Notation) を拡張した BSON(Binary JSON) を用いてドキュメントを保存する。図 2.5 に BSON を用いたドキュメントのデータ構造を示す。この例では、ID と Name と Age という 3 つのフィールドを持つドキュメントを示している。MongoDB は大量のドキュメントを管理しており、各ドキュメントは以下に示す 3 つの要素からなっている。

1. ドキュメントの長さをバイト数で表すヘッダ。
2. 各フィールドのキーバリューのペア。キーはバリューの型識別値とフィールド名の複合キーとなっており、バリューはキーで指定された型の情報である。
3. ドキュメントの終端を表す終端記号。

図 2.5 の例では、2 番目の要素のキーで使われている型識別値はそれぞれ 7、2、16 であり、ID、UTF-8 の文字列、32bit 整数の型を表す。この他にも、MongoDB は、この例の他に 64bit 整数、倍精度浮動小数、論理型、タイムスタンプ、UTC の日付型などの型を扱うことができ、現在 19 種類の型に対応している。

2.2.2 ドキュメント指向型ストアのクエリ

MongoDB のクエリは主に、ドキュメントの各フィールドのバリューに対して条件を与え、それを満たすドキュメントを返すクエリである。例えば、図 2.5 のドキュメントを保持している MongoDB に、「name = Tom かつ age = 25」という条件を与えると、図 2.5 のドキュメントが返される。この例では _id に対する条件がないように、全てのフィールドに対して条件を指定する必要はない。また、各フィールドに与える条件は例のような一致だけではなく、数値型であれば大小比較や範囲指定、文字列型であれば文字列の正規表現探索など様々な条件をサポートしている。

このようなクエリの計算量は、ドキュメント件数を N とすると、全てのドキュメントに対して条件を満たすがどうかの比較を行うため、$O(N)$ である。よって、膨大な情報を扱う場合は、ドキュメント件数が多くなるため、計算量が大きくなる。ドキュメント指向型ストアでは、インデッ

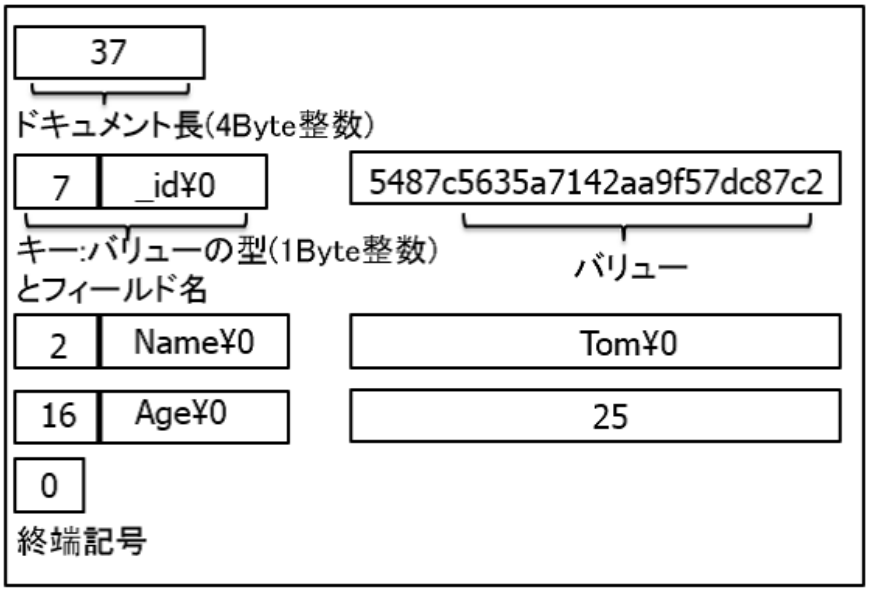


図 2.5: BSON を用いたドキュメントのデータ構造

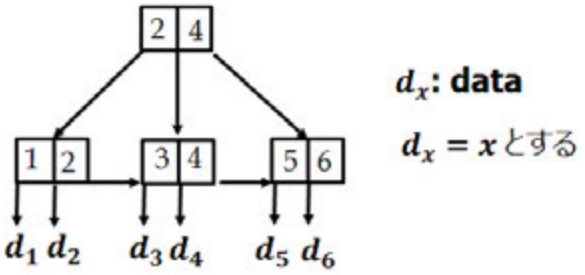


図 2.6: B+tree インデックスの構造

クスを用いることでこの問題に対処している。今回対象とする MongoDB でもインデックスが利用可能であり、主に B+tree インデックスが利用される。

B+tree は木構造の一種であり、データへのポインタが葉ノードに格納され、根ノードから葉ノードまでの途中のノードは全てキーのみを格納するという特徴を持つ。木の次数は自由に設定することができる。図 2.6 に B+tree インデックスの構造の例を示す。この例は、B+tree の子ポインタ数が 3 の場合を示している。葉ノードに格納されている d_x はそれぞれデータを表し、それぞれの添字と同じ数値がデータとして格納されている。また、各ノードの中に記されている数字はキーを表す。データは左から昇順に並べられており、キーを調べることで、そのノードよりも左側のノードはキー以下の値、右側はキーよりも大きい値が格納されているとわかる。根ノードから順にキーを探索し、探索対象とキーの大小の比較を繰り返し、葉ノードに到達すると、データの格納先がわかる。

例えば、図 2.6 で 3 を探索するとする。まず、根ノードを探索し、3 は 2 より大きく 4 以下であるため、次の探索対象は中央の子ノードとなる。次に、子ノードを探索し、3 は 3 以下であり、かつこのノードは葉ノードであるため、データ d_3 に到達し、探索が終了する。

また、B+tree では、葉ノードは、右の葉ノードを指すポインタを持ち、最も右の葉では NULL を指す。このポインタは、単純な探索の場合は用いられる事はないが、範囲検索を行う場合に、隣のノードに効率的にアクセスするために設けられている。

B+tree インデックスを用いた場合の探索の計算量は、次数を d、データ件数を N としたとき、葉

ノードに到達するために必要なノードアクセス回数が $log_d N$ であるため、$O(\log N)$ となる。よって、インデックスを用いない場合の $O(N)$ に比べて大幅に計算量を削減できる。しかし、文字列の正規表現探索など、探索に整列された情報を利用できないクエリの場合は、計算量が $O(N)$ のままであり、これらのクエリがボトルネックとなる。

また、MongoDB は、B+tree インデックスだけでなく、文字列探索のためにテキストインデックスをサポートしている。テキストインデックスを用いることで、文字列の単語検索といった B+tree インデックスでは対応できないクエリを一部対応可能である。しかし、MongoDB のマニュアルにもあるとおり [23]、テキストインデックスの作成には、巨大な複合キーインデックスを作る必要があり、B+tree インデックスに比べてメモリ使用量、インデックス作成時間共に長くなる。

ドキュメント指向型ストアを KVS 型やカラム指向型と比較すると、データ構造は 2.2.1 節で述べた様に、データ構造は KVS 型やカラム指向型と同じく、キーバリューの組を用いて管理しているのに対して、クエリは KVS 型やカラム指向型ではキーバリューの組のキーに対して条件を与えて探索を行っていたのに対し、ドキュメント指向型では、バリューに対して条件を与えて探索を行う。バリューに対する探索は、キーに対する探索に比べて計算量は大きいが、実際の情報であるバリューに対して条件を指定できるため、より柔軟な探索が可能である。

このように、MongoDB で対応する型およびクエリは様々な種類があげられるが、それらのうちで計算量の大きく、ボトルネックとなるのは、文字列型に対する正規表現探索である。そのため、本論文では文字列の正規表現探索を主な対象とし、正規表現探索クエリを高速化することで、システム全体の性能を向上させることを目的とする。

2.3 グラフ型ストア

グラフ型ストアでは、グラフ構造を用いてデータを表現できる構造型ストレージであり、多対多の関係性を表現するのに適している [24]。その特徴を活かして、人と人の関係を表現するソーシャルネットワーク [25][26]、物と人の関係を表現する商品推薦システム [27]、人々の要求と事業者の関係を表現する健康管理システム [28]、様々な遺伝子等の関係を表現する生命情報学 [29]、など、幅広い用途に用いられている [30]。

代表的なグラフ型ストアとして、Neo4j [31]、InfiniteGraph [32]、AllegroGraph [33] などがある [24]。このうち、高速化の対象とするグラフ型ストアして本論文では Neo4j を対象とした。Neo4j は Java 言語で実装されたオープンソースかつ代表的なグラフ型ストアの一つであり、本章では、グラフ型ストアのデータ構造および クエリは Neo4j を例として述べる。

2.3.1 Neo4j のデータ構造

図 2.7 に Neo4j のデータ構造を示す。Neo4j のデータは、グラフの構成要素である頂点、辺と頂点、辺の情報を付加する属性の三つの要素からなる。

Neo4j は Java 言語で実装されており、頂点、辺をそれぞれクラスとして実装し、インスタンスを生成することで、頂点や辺を追加する。属性は、頂点、辺にメンバとしてとして付加される。図 2.7 の例では、頂点に ID と名前と性別と年齢が属性として与えられている。この属性は、予め与える属性を決めておく必要や全ての頂点に一律に与える必要はなく、ユーザーが自由に属性を追加できる (例えば、図 2.7 の一つの頂点にのみ国籍の属性を与える等。)。このように、自由度の高い属性と、頂点と辺を用いたグラフ構造を用いることで、頂点間に複雑な関係性をもつデータを表現できる。

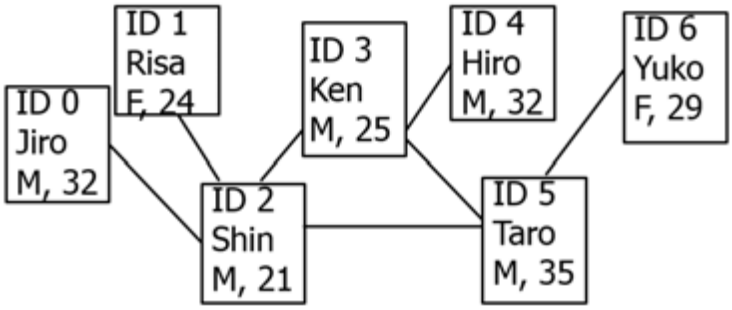


図 2.7: Neo4j のデータ構造

2.3.2 グラフ型ストアのクエリ

グラフ型ストアのクエリは、以下に示す 4 種類に分類される [34]。

1. 書き込みクエリ:グラフ型ストアの各要素 (頂点、辺、属性) の追加、削除、変更を行うクエリ。
2. 属性取得クエリ:ある頂点や辺から属性を取得するクエリ。
3. 横断クエリ:ある頂点から辺を跨いで他の頂点を参照する横断処理を伴ってグラフの一部を横断するクエリ。
4. グラフ探索クエリ:グラフ全体に跨って探索を行うクエリ。

これらのクエリの内、本論文の対象とするのは、探索等を伴い、計算がボトルネックとなる書き込みクエリ以外の 3 つの読み込みクエリである。3 種類の読み込みクエリのうち、横断クエリとグラフ探索クエリはグラフ型ストア特有のクエリである。横断によってグラフの辺を跨いで探索することにより、頂点間の関係性を調べる事が出来る。実際の運用の際は、これらの 3 種類の読み込みクエリは組み合わせて利用することで、グラフ型ストアは様々な用途に用いられている。利用例としては、ソーシャルグラフに基づく顧客への商品推薦システムや配送事業における最短経路検索システム等があげられる [24]。

3 種類の読み込みクエリ間でクエリの計算量を比較すると、属性取得クエリは、頂点や辺を指定して属性を取得するのみであるため、計算量が最も小さい。横断クエリとグラフ探索クエリは、クエリの規模がグラフの一部であるか全体であるかという違いがあり、規模が大きいグラフ探索クエリの方が計算量が大きい。また、グラフ探索クエリでは、扱うグラフが大きくなるにつれ、計算量が大きくなる。そのため、巨大なグラフを扱う場合は、この様なグラフ探索クエリの計算量が非常に大きく、グラフ型ストアのボトルネックとなる。本論文では、3 種類の全ての読み込みクエリを対象に高速化を行うが、計算量の観点からグラフ探索クエリが最も GPU 向けであり、本論文による提案手法による高速化が期待できる。グラフ探索アルゴリズムには様々な種類が存在するが、Neo4j で実装されているアルゴリズムおよび本論文で対象とするアルゴリズムについては次節で述べる。

2.3.3 グラフ探索アルゴリズム

Neo4j には、グラフ探索アルゴリズムが 5 つ実装されている。この節では、それらのアルゴリズムについて説明し、どのアルゴリズムを対象とするかを述べる [35]。

2.3.3.1 Dijkstra 法

Dijkstra 法は、重み付きグラフにおいてある頂点からの最短経路を求めるアルゴリズムである。一般的に、Dijkstra 法はある頂点から全ての頂点に対して最短経路を求めるが、Neo4j では、ある 2 頂点間の最短経路を求める仕様となっている。ここでは、Neo4j の仕様に合わせて、2 頂点間の最短経路を求めるものとする。この変更でアルゴリズムを変更する部分は、終了条件である。全ての頂点の探索が終了した時から 2 頂点間の最短経路が求まったときに変更する。

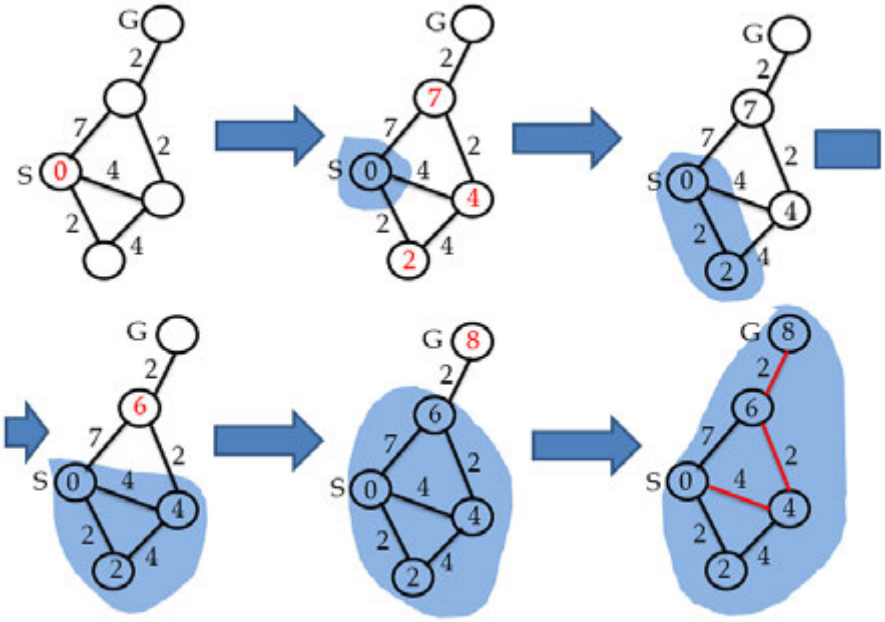


図 2.8: Dijkstra 法の実行例

Dijkstra 法の詳細は以下の通りである。

- 最短経路を求める 2 頂点を S と G とし、S から G への最短経路を求めるとする。
- 頂点 A から B への距離を A(B) とする。A と B が隣接している場合 A(B) は A と B を結ぶ辺の重みに一致する。
- 初期状態として、各頂点の S からの距離を設定する。S(S) は 0、S 以外の頂点は探索を行っていないため、無限大とする。
- 以下の操作を操作 2 によってアルゴリズムが終了するまで繰り返す。

  1. まだ探索を行っていない頂点のうち、S からの距離が最も短い頂点を選択する。この頂点を V とする。また、距離が無限大の頂点しかない場合はどの頂点も選択しない。
  2. V が G と一致していれば、アルゴリズムを終了する。また、V としてどの頂点も選択されていない場合は、探索は失敗とし、終了する。
  3. V が G と一致していない場合、V に隣接する全ての頂点について以下の操作を行う。
     - V の隣接頂点を X とする。
     - $S(X) > S(V) + V(X)$ ならば、$S(X) = S(V) + V(X)$ とし、X の親として V を記憶する。
  4. V を探索済みにする。

- 探索が失敗していなければ、アルゴリズム終了時点で、$S(V) = S(G)$ であり、これが S から G までの最短の距離を表す。また、G から順に S にたどり着くまで親をたどることで、最短経路を求める事ができる。

- 計算量は、頂点数を V とすると $O(V^2)$ である。Dijkstra 法のボトルネックとなる部分は操作 1 である。操作 1 は V 個の頂点の距離の最小値を求めるもので、1 回あたり $O(V)$ であり、これが V に比例する回数行われるため、$O(V^2)$ となる。

図 2.8 は Dijkstra 法を 5 頂点のグラフに対して実行したときの例である。S から G への最短経路を求めている。頂点の中の数字は S からの距離、辺の横の数字はその辺の重みを表している。頂点の中が空欄の場合、S からの距離が無限大であることを示す。色が塗られている領域は、既に探索が終了しており、最短経路が分かっている領域である。

6 つの図はアルゴリズムのループ 1 回が終わる毎の状態を表している。1 回のループで探索頂点を選び、その隣接頂点の距離を更新している。

左上の図が初期状態を表している。頂点 S の S からの距離は 0 であり、その他の頂点はまだ探索を行っていないため、距離は無限大を表す空欄である。

2 番目の図では、まだ訪れていない頂点のうち最も距離が短い、距離 0 の頂点を探索する頂点として選び、その頂点と隣接する頂点の距離を更新している。隣接していれば、距離は必ず無限大よりも小さいため、隣接する全ての頂点が更新されている。

3 番目の図では、距離 2 の頂点が最も距離が短いため、探索する頂点として選ばれる。2 つの頂点が隣接しているが、この頂点を通ると逆に距離が長くなってしまうため、距離は更新されない。

4 番目の図では、距離 4 の頂点が選ばれ、1 つの頂点がこの頂点を通った場合の方が距離が短くなるため、更新されている。このように、まだ探索していない頂点のうち距離の最も短いものを選び、その隣接頂点の距離の更新を繰り返す。右下の 6 番目の図では、S から G までの最短経路が求まっている。その距離は 8 で経路は親頂点を S までたどることで求められる。

2.3.3.2 A*法

A*法はグラフのそれぞれの頂点から目的頂点までの最短距離の推定値が分かっている場合に、ある 2 頂点間の最短経路を求めるアルゴリズムである。これは、Dijkstra 法を推定値付きの場合に改良したもので、計算時間は Dijkstra 法以下になる。

ある頂点から目的頂点までの推定値を a、真の最短距離を b としたとき、$0 \leq a < b$ の条件を満たす必要がある。a の値が b に近いほど計算時間が短く、$a = 0$ のときは Dijkstra 法と同じ計算時間になる。

A*法の詳細は以下の通りである。

- 最短経路を求める 2 頂点を S と G とし、S から G への最短経路を求めるとする。

- 頂点 A から B への距離を A(B) とする。

- 頂点 A から目的頂点 G への距離の推定距離を g(A) とする。

- 頂点 S から頂点 A を通り、頂点 G へたどり着くまでの推定距離を f(A) とする。f(A) は $f(A) = S(A) + g(A)$ で求められる。

- 計算中の頂点を格納するリストを Open リスト、計算済みの頂点を格納するリストを Close リストとする。

- 初期状態として、Open リストに S を追加する。Close リストは空にする。

- 以下の操作を操作 2 によってアルゴリズムが終了するまで繰り返す。
  1. Open リストに含まれる頂点のうち、G までの推定距離が最も小さい頂点を選択する。この頂点を V とする。
  2. V と G が一致していれば、アルゴリズムを終了する。また、Open リストが空の場合、探索は失敗とし、終了する。
  3. V と G が一致していない場合、V に隣接する全ての頂点について以下の操作を行う。
     - V の隣接頂点を X とする。
     - X が Open リストにも Close リストにも含まれない場合、$f(X) = S(V) + V(X) + g(X)$ とし、X を Open リストに加え、X の親として V を記憶する。
     - X が Open リストに含まれる場合、$f(X) > S(V) + V(X) + g(X)$ ならば $f(X) = S(V) + V(X) + g(X)$ とし、X の親を V に書き換える。
     - X が Close リストに含まれる場合、$f(X) > S(V) + V(X) + g(X)$ ならば $f(X) = S(V) + V(X) + g(X)$ とし、X を Close リストから Open リストに移し、X の親を V に書き換える。
  4. V を Close リストに加える。
- 探索が失敗していなければ、$f(V) = f(G)$ であり、$f(G) = S(G) + 0$ であるため、$f(V)$ が最短距離を表す。また、G から順に S にたどり着くまで親をたどることで、最短経路を求めることができる。
- 計算時間は Dijkstra 法以下となるが、その時間は V^2 に比例するため、計算量は $O(V^2)$ である。

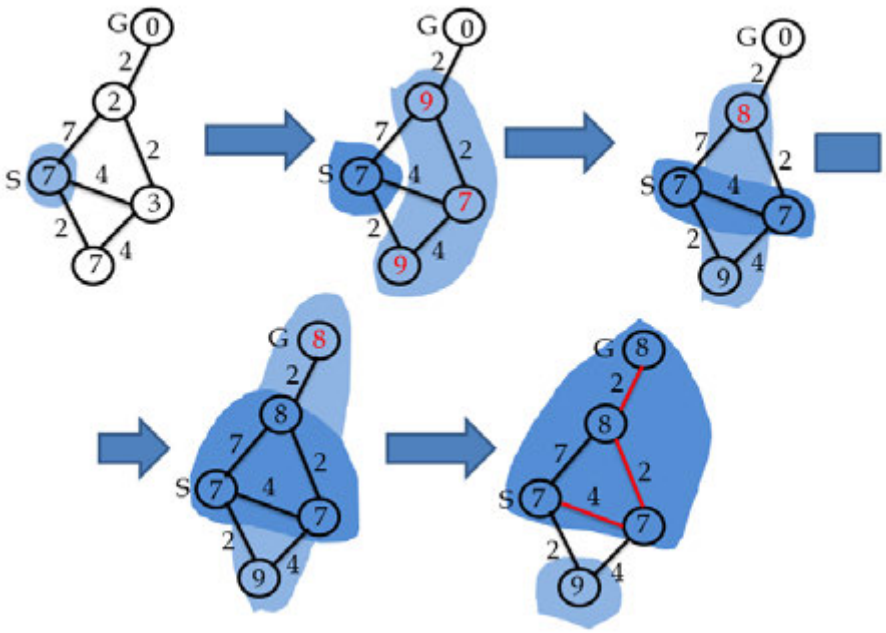


図 2.9: A*法の実行例

図 2.9 は A*法を 5 頂点のグラフに対して実行したときの例である。S から G への最短経路を求めている。濃い色で塗られている領域は、Close リストに含まれていることを示し、薄い色で塗られている領域は Open リストに含まれていることを示す。頂点の中の数字は、その頂点が Open

リストにも Close リストにも含まれない場合、その頂点から目的頂点までの推定値を表す。頂点が Open リストか Close リストのいずれかに含まれる場合は、頂点 S からその頂点を通って頂点 G へたどり着くまでの推定値を表す。

5 つの図はのループ 1 回が終わる毎のを示している。1 回のループで Open リストから探索する頂点を選び、その隣接頂点を通る道の推定距離を更新する。

左上の図は初期状態を表している。Open リストは頂点 S のみが含まれ、Close リストは空である。各頂点の中の数字はそれぞれの頂点から頂点 G までの推定距離である。

2 番目の図では、初期状態では Open リストには頂点 S しか含まれないため、頂点 S を探索頂点として選び、隣接頂点の探索を行っている。隣接頂点は全て、Open リストにも Close リストにも含まれていないため、Open リストに加えてその頂点を通る道の推定距離として頂点の中の数値を更新する。

3 番目の図では、Open リストに含まれる 3 つの頂点の内、最も推定距離が小さい距離 7 の頂点を探索頂点として選ぶ。隣接頂点の内、一つの頂点がこの頂点を通った場合の方が推定距離が短くなるため、更新されている。このように、Open リストのうち最も推定距離が短いものを一つずつ探索し、その隣接頂点の推定距離を更新することを繰り返す。5 番目の図では、頂点 G が探索頂点として選ばれたため、アルゴリズムが終了している。頂点 S から頂点 G までの最短距離は 8 で頂点 G から親をたどることで最短経路が求められる。

図 2.8 と図 2.9 を比較すると、図 2.9 で用いたグラフは図 2.8 で用いた例と同じものであり、最短距離と最短経路は Dijkstra 法で求めたものと一致していることがわかる。Dijkstra 法では、探索を 4 頂点に対して行っているのに対し、A*法では 3 頂点に対してしか行っていない。このことから、A*法は Dijkstra 法よりも計算時間が短いことがわかる。

2.3.3.3 Shortest Path

Shortest Path は、ある 2 頂点間のホップ数が最小となる経路を全て求めるアルゴリズムである。Dijkstra 法や A*法は重みを考慮した最短経路を求めるが、このアルゴリズムは重みは考慮せず、ホップ数が少ない経路を求める。また、最小ホップ数の経路が複数存在する場合その全てを求めるため、経路が 1 つ見つかっても探索を終了せず、そのホップ数の経路を全て探索しなければならない。

Shortest Path の詳細は以下の通りである。

- 最小ホップの経路を求める 2 頂点を S と G とし、S から G への経路を求めるとする。
- 探索が終了した経路を格納するリストを CLose リスト、S から G までの最小ホップの経路を格納するリストを Answer リストとする。
- 初期状態として、S 一つのみの経路を Close リストに追加する。
- 探索中のホップ数を n とし、n の初期値は 1 とする。また、初期状態の S のみの経路はホップ数 0 である。
- 操作 3 でアルゴリズムが終了するまで、以下の操作を繰り返す。
  1. Close リストのうち、ホップ数が n-1 の経路全てを選択し、その最後の頂点を V とする。
  2. V の隣接頂点を X とし、X の状態に応じて以下の操作を行う。
     - $X \neq V$ の場合、X を経路に加えた新しい経路を Close リストに追加する。
     - $X = V$ の場合、X を経路に加えた新しい経路を Answer リストに追加する。

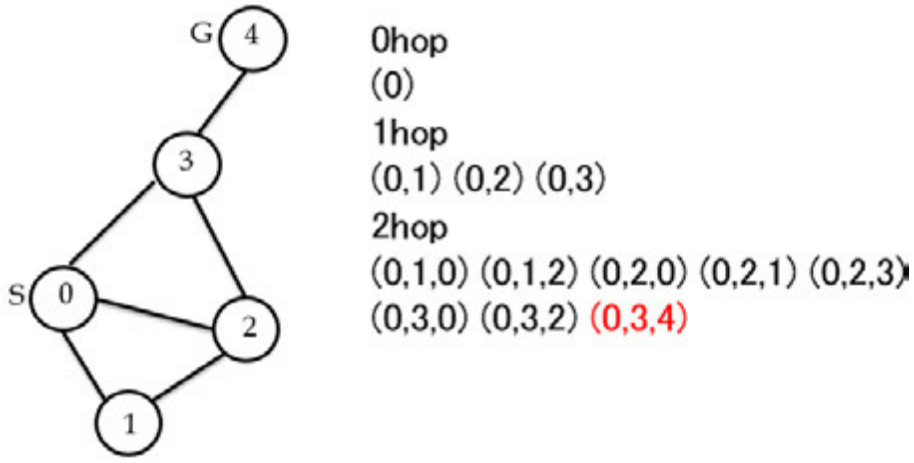


図 2.10: Shortest Path の実行例

3. Answer リストに経路が存在する場合、アルゴリズムを終了する。
4. n に 1 を加える。

- アルゴリズム終了後、Answer リストに存在する経路が S から G までの最小ホップ数の経路を表す。
- 計算量は、最小ホップ数を n、頂点の次数を d とするとき、d^n である。上記の操作のループ 1 回につき、探索する頂点の数が d 倍されるためである。

図 2.10 は、Shortest Path を 5 頂点のグラフに対して実行したときの例である。S から G への最小ホップの経路を求める。頂点の中にかかれた数字は頂点 ID である。

図の右側には、Close リストあるいは Answer リストに加えられたホップ数毎の経路が示されている。初期状態として、ホップ数 0 の S すなわち頂点 0 のみの経路が始めにリストに追加される。その後、探索のループを一回行う毎に 1 ホップずつ新たな経路がリストに加えられる。

1 ホップ目は、0 ホップ目の経路の最後の頂点、すなわち頂点 0 の隣接頂点を探索する。頂点 0 の隣接頂点は頂点 1、頂点 2、頂点 3 であるため、この 3 つの頂点が経路に加えられ、リストに追加されている。同様に、2 ホップ目は、1 ホップ目の経路の最後の頂点、すなわち頂点 1、頂点 2、頂点 3 の 3 つの頂点の隣接頂点を探索する。それぞれの頂点の隣接頂点が経路に加えられ、リストに追加される。この時、経路 (0,3,4) は頂点 G である頂点 4 への経路となっているため、Answer リストに加えられる。Answer リストに経路が加えられたため、ここでアルゴリズムのループが終了する。この時、最短ホップ数は 2 であり、その経路は (0,3,4) である。

2.3.3.4 All Path

All Path は、ある 2 頂点間の経路の設定したホップ数以下の経路を全て求めるアルゴリズムである。グラフ探索の方法は、Shortest Path と同じであり、異なるのは終了条件のみである。Shortest Path では、経路が見つかった場合、そのホップ数で探索を終了していたが、All Path では、経路が見つかっても探索を終了せず、設定したホップ数まで探索を続ける。逆に、経路がみつからなくても、設定したホップ数まで探索が行われれば、探索を終了する。 図 2.11 は、All Path を 5 頂点のグラフに対して実行したときの例である。設定するホップ数は 3 である。また、3 ホップ目は経路の数が多いため、リストの一部のみを記載している。

図 2.10 と図 2.11 を比較すると、探索方法が同じであるため、Shortest Path の探索が終了する 2 ホップ目までは全く同じである。図 2.11 では、ホップ数を 3 に設定しているため、3 ホップ目ま

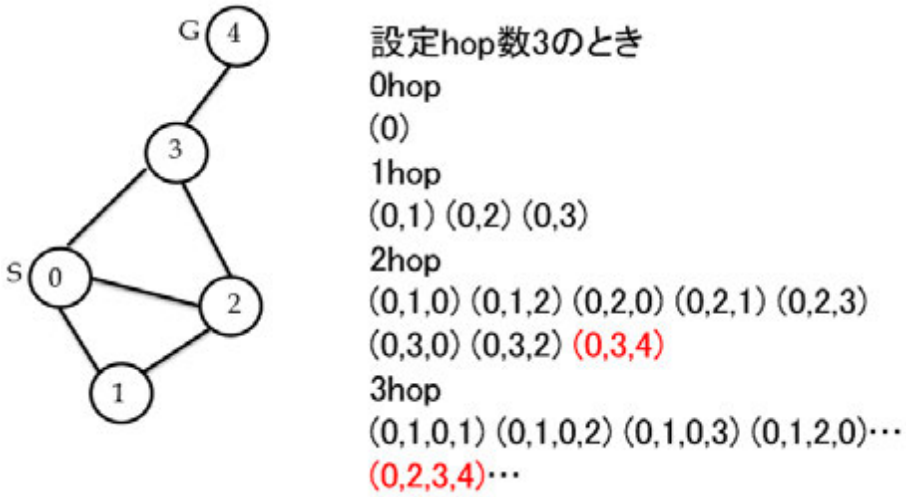


図 2.11: All Path の実行例

図 2.12: All Simple Path の実行例

で探索を行う。3 ホップ目で新たに (0,2,3,4) という経路を見つけたため、3 ホップ以下の経路は 2 ホップの (0,3,4) を加えて、(0,3,4) と (0,2,3,4) の 2 つである。

2.3.4 All Simple Path

All Simple Path は All Path に変更を加えたもので、All Path で探索した経路のうち、2 回同じ頂点を通る経路を除いたものである。探索の方法は、Shortest Path や All Path と同じで、終了条件は、All Path と同じく設定したホップ数まで探索を終えたときである。

図 2.12 は、All Simple Path を 5 頂点のグラフに対して実行したときの例である。設定するホップ数は 3 である。3 ホップ目は経路の数が多いため、リストの一部のみを記載している。図の右側のリストのうち、薄い色で塗られている経路は、同じ頂点を 2 回通ったため除外されたものを示している。

図 2.12 は図 2.11 から 2 回同じ頂点を通る経路を除外しただけで、他は一致している。今回の例は頂点数が少ないため、同じ頂点を 2 回通る経路の割合が多いが、大規模なグラフの場合、そのような経路の割合は少なく、All Path と All Simple Path の差は小さくなる。

Answer リストに含まれる S から G への経路の結果を見ると、図 2.11 の結果に 2 回同じ頂点を通る経路が含まれていないため、図 2.12 の結果も同じである。

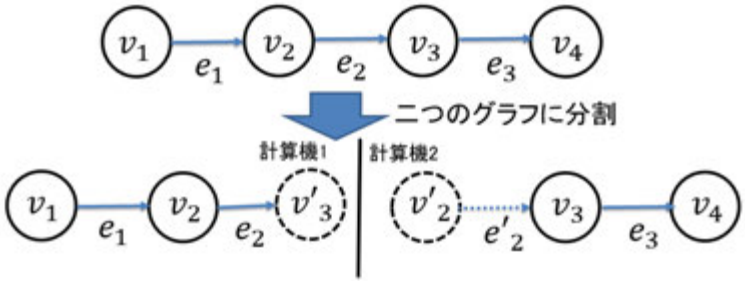


図 2.13: グラフ型ストアの分割

2.3.4.1 対象アルゴリズムのまとめ

Neo4j に実装されているアルゴリズムは 5 種類あるが、Shortest Path、All Path、All Simple Path の 3 種類は、条件が異なるだけで行う探索は同じである。そのため、探索の種類としては、Dijkstra 法と A*法と合わせて実質 3 種類のみである。そのうち、Dijkstra 法と A*法は単一始点最短経路問題であり、Shortest Path、All Path、All simple Path は幅優先探索の処理を行っている。本論文では、このうち、主に単一始点最短経路問題を対象に、実装、評価を行う。

2.3.5 グラフ型ストアの分割

グラフ型ストアをより多くのデータが扱えるよう拡張するための手法として、グラフ型ストアを複数台の計算機にグラフ型ストアを分割する手法が挙げられる。Averbuch らがグラフ型ストアの一つである Neo4j を対象に、グラフ型ストア分割をいくつかのグラフ分割アルゴリズムで評価している [36]。

また、複数の計算機への分散をサポートしているグラフ型ストアとして、Apache Titan [37] があげられる。本節においては、Neo4j の分割について述べる。

グラフ型ストアを分割する際、分割後の各グラフに跨る辺をどのように扱うかが問題となる。彼らは各グラフの境界に属する辺と頂点に対して仮想の辺と仮想の頂点を作成することで境界辺を扱っている。彼らが提案したグラフ分割手法を図 2.13 に示す。図 2.13 では、4 頂点からなるグラフを 2 頂点の 2 つのグラフに分割している。図中に実線で描かれた e_1、e_2、e_3 の 3 つの辺と v_1、v_2、v_3、v_4 の 4 つの頂点は分割前から存在する実際の辺と頂点を表し、点線で描かれた e'_2 は仮想の辺、v'_2、v'_3 は仮想の頂点を表す。仮想の辺と頂点は属性を持たない。属性を参照したり、仮想の頂点からさらに探索を続ける場合はグラフ間を跨る参照を行って実際の辺や頂点を参照する必要がある。例えば、図 2.13 において、計算機 1 に属する左側のグラフに対して探索を行い、v'_3 に到達し、その属性が知りたい場合は、計算機 2 と通信を行って v_3 の参照を行う。

この手法を用いれば、種々のグラフ分割アルゴリズムをグラフ型ストアに応用できる。グラフ分割については既にいくつものアルゴリズムが提案されているが、グラフ型ストアの分割に適したアルゴリズムは限られている。実際のグラフ型ストア適用するには、分割の際にグラフ全体を参照することなく、かつアルゴリズム実行中にグラフの修正が発生する環境下で動作する必要がある。彼らは、これらの条件を満たすアルゴリズムである EvoPartition [38]、DCCA (The Dynamic Cut-Cluster Algorithm) [39]、DiDiC (The Distributed Diffusive Clustering algorithm) [40] の 3 つのアルゴリズムを比較した結果、DiDiC が最もグラフ型ストアに適すると判断した。

DiDiC は、粒子がランダムに散らばることで密度の高い領域から密度の低い領域に伝搬し、やがて均一の密度になる物理現象である拡散現象をグラフ上で擬似的に再現し利用することでグラフ分割を行うアルゴリズムである。

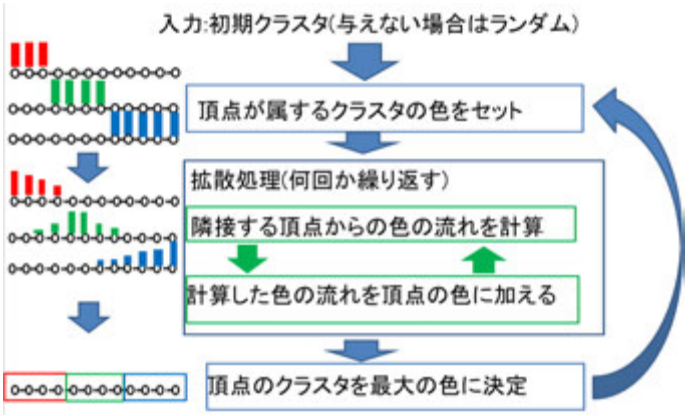


図 2.14: DiDiC に用いる拡散の概念図と DiDiC アルゴリズムのフローチャート

図 2.14 に DiDiC に用いる拡散の概念図と DiDiC アルゴリズムのフローチャートを示す。左側の概念図と右側のフローチャートの各段階が対応関係にある。まず、アルゴリズムの入力として、初期クラスタを与える。初期クラスタを設定しない場合は、各頂点がランダムにいずれかのクラスタに割り当てられる。次に、クラスタ数を k とすると、それぞれのクラスタに対応する k 種類の色を用意し、各頂点にクラスタの色を割り当てる。これが一番上の概念図に相当する。次に、これらそれぞれの色に対して別々に拡散処理を実行する。拡散処理とは、隣接する頂点からの色の流れを加える処理である。色の流れは実際の拡散の現象と同じように、色が濃い部分から薄い部分へと流れていく。概念図では色の濃さを高さで表している。二番目の概念図ではある程度拡散が進んだ様子を示しており、固まっていた色が周囲に広がっている。拡散処理を続けていると最終的には全て均一になるが、密グラフになっている部分はその外に色が流れにくいため、ある程度で止めた場合は密グラフの部分が同じ色を共有している可能性が高い。そのため、拡散処理を何回か繰り返したところで終了し、その時に最も濃い色のクラスタに頂点を割り当てる。これを一番下の概念図とフローチャートが表している。ここでクラスタが決定するため、アルゴリズムを終了できるが、決定したクラスタを入力として再びアルゴリズムを実行することでより良い結果が得られる。

Averbuch らは、グラフ型ストアの分割に DiDiC を用いることによって、各グラフ間のトラフィックをランダム分割に比べて最大 90%削減に成功した [36]。

このように、グラフ型ストアは複数台の計算機に分割して運用することが可能であるが、分割を行うとグラフ間を跨る処理の性能が大幅に低下する。彼らが Neo4j を 2 つに分割し、グラフ間の辺を跨ぐ処理の性能を元の Neo4j と比較した結果、処理速度が約 15 倍悪化した。本論文では、このように複数に分割されたグラフ型ストアから情報を抽出し、複数の遠隔 GPU への分散キャッシュを用いることで、グラフ探索を高速化する。その際、GPU 間の同期手法として非同期更新を行うことで、GPU 毎に分割されたグラフ間を跨る処理の性能の低下を削減する。

2.4 GPU による汎用計算

GPU は、本来グラフィックス処理に特化したハードウェアであり、非常に高い並列性を持っており、その並列性を生かせば、高い性能を発揮する。近年、その高い性能が注目され、科学技術計算等、汎用的に使われている。その使用用途の一つに本研究で扱うキーや文字列の検索処理やグラフ探索処理が含まれる。

2.4.1 GPU アーキテクチャ

GPU は数百から数千という多数のコアで構成されている。GPU は多数のコアの並列性を活かすのに適したアーキテクチャを採用している [10]。

GPU を提供する主なベンダーには Nvidia 社と AMD 社があり、ベンダーによってアーキテクチャは異なるが、本節では、本研究の評価で用いた Nvidia 社の GPU アーキテクチャについて述べる。

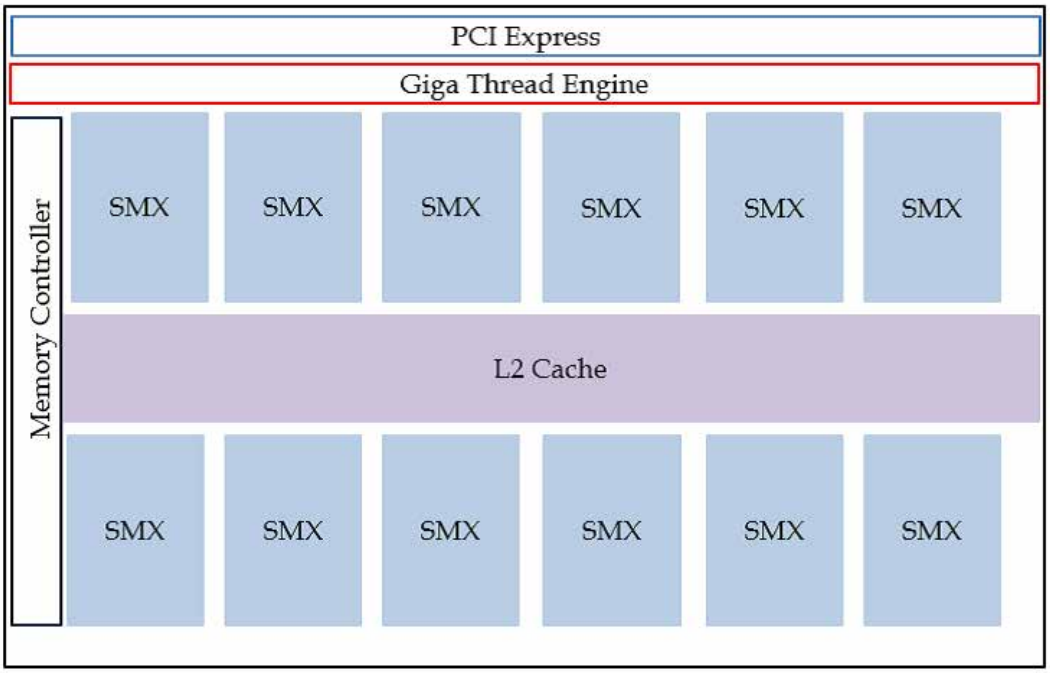


図 2.15: Nvidia 社 GPU の Kepler アーキテクチャ

図 2.15 は、Nvidia 社の GPU アーキテクチャを表している。Nvidia 社の GPU アーキテクチャにはその世代毎にいくつか種類があるが、このアーキテクチャは GPU のアーキテクチャの一種である Kepler アーキテクチャである。図 2.15 からも分かる通り、Kepler アーキテクチャの主な構成要素として、PCI Express、Giga Thread Engine、Memory Controller、SMX、L2 Cache があるが、そのうちこのアーキテクチャに特有のものは、Giga Thread Engine と SMX である。

GPU はコア数が非常に多いため、一つ一つのコアを独立に扱うと管理が難しい。そこで、このアーキテクチャでは、192 個のコアを一つのまとまりとして管理している。そのまとまりが SMX である。そのため、このアーキテクチャの GPU のコア数は、192 の倍数となる。SMX の内部のアーキテクチャについては後述する。

GPU では、その並列性を活かすために、コアの数よりもさらに 多数のスレッドを発行し、処理を行う。スレッドをそれぞれのコアで処理するが、スレッド一つ一つ独立にどのコアで処理するかを割り振ろうとすると、スレッドとコアの数が共に多いため、スレッドの割り振りのオーバーヘッドが大きくなってしまう。そのため、複数のスレッドを一つにまとめてスレッドブロックとし、そのスレッドブロックをコアのまとまりである SMX に割り振る。スレッドブロックにいくつのスレッドが含まれるかはユーザが決めることができるが、最大値は 1,024 である。このスレッドブロックをどの SMX に割り振るかを決めるのが Giga Thread Engine である。

図 2.16 は GPU のコアをまとめた SMX のアーキテクチャを表している。SMX は Kepler アーキテクチャの GPU の一部分であり、この図は 2.15 の SMX の一つを詳細に表したものである。図 2.16 からも分かる通り、SMX の主な構成要素として、Warp Scheduler、Register File、Core、Shared

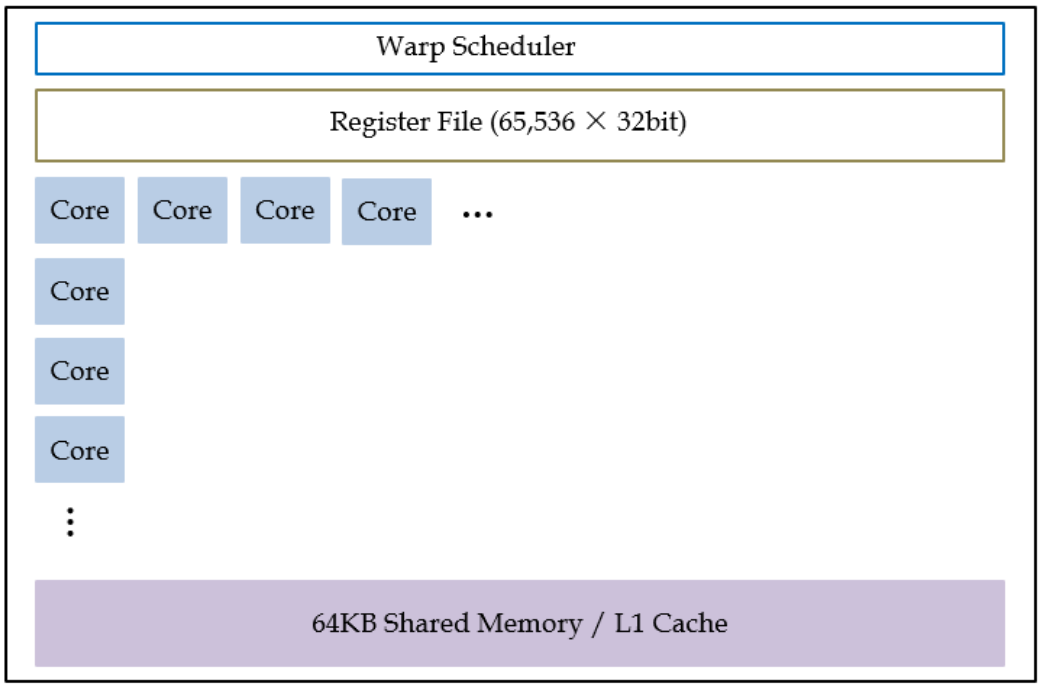


図 2.16: SMX のアーキテクチャ

Memory、L1 Cache がある。このうち、このアーキテクチャに特有のものは、Warp Scheduler と Shared Memory である。

SMX にスレッドブロックとして、スレッドのまとまりが与えられるが、スレッドブロックもスレッドの最大数 1,024 と多く、それぞれを独立にコアに割り降るのはオーバーヘッドが大きい。そこで、スレッドを 32 個まとめてワープという単位で扱う。スレッドブロックがいくつかのワープに分かれることになるが、このワープのうち、どのワープを実行するかを決めるのが Warp Scheduler である。Kepler アーキテクチャでは、一度に 4 つのワープが実行でき、ワープ一つに対して同時に 2 つの命令を実行する事ができる。つまり、合計 8 個の命令が同時に実行される。SMX 内のコアの数が 192 個であるため、各ワープの一つの命令あたり 24 個 (192/8) のコアが使われる。

Shared Memory は、SMX 内で共有されたメモリである。これにより、スレッドブロック内でデータの共有が発生しても、高速にアクセスができるようになる。Shared Memory の大きさは 64KB 以内でユーザが決めることができる。スレッドブロック内でのデータ共有が多い場合はこの大きさを大きくすることで、メモリアクセス時間を削減する事ができる。この 64KB の領域は、Shared Memory と L1 Cache で共有されており、ユーザが Shared Memory の大きさを決め、先に Shared Memory の領域を確保し、残りの領域が L1 Cache に割り当てられる。この 2 つを共有することにより、Shared Memory をあまり使わない場合でもこの領域の無駄は発生しない。

このように、GPU のアーキテクチャは階層構造になっており、GPU に与える命令である多数のスレッドも階層的に処理される。GPU 全体でスレッドの処理の流れをみると、まず、スレッドブロックで大きなまとまりとなり、Giga Thread Engine で SMX に割り振られる。スレッドブロックが 32 スレッド毎にワープに分けられ、それが SMX 内の Warp Scheduler によって管理されて実行されるという流れになる。

2.4.2 GPU による単一視点最短経路問題の高速化

Ortega-Arranz らが GPU を用いた単一視点最短経路問題の高速化を行った [41]。単一視点最短経路問題は、一つの頂点からその頂点以外の全ての頂点への最短経路を求める問題である。Ortega-Arranz らは、単一視点最短経路問題を解く代表的なアルゴリズムである Dijkstra 法を GPU 向けに並列化することで、GPU によるこの問題の高速化を行った。Dijkstra 法は大きく分けると次の二つの操作に分けられ、この二つの操作を繰り返すことで、最短経路を求める。また、この問題は本論文の対象とする問題の 1 つである。

1. まだ探索していない頂点のうち、始点からの距離が最短であるものを選択する。
2. 操作 1 で選択した頂点と隣接する頂点の始点からの距離が選択した頂点を通った方が短くなる場合、隣接する頂点の距離を更新する。

操作 1 で始点からの距離が最短である頂点を選ぶ理由は、その頂点は後に操作 2 で距離を更新されることがないからである。よって、操作 1 で選ぶ頂点が後に操作 2 で距離が更新されないという条件を満たしていれば、距離が最短である頂点以外を選んでもアルゴリズムの結果を損ねることはない。そこで、Ortega-Arranz らは条件を満たす複数の頂点を並列に操作 1 で選び、それらに対して操作 2 を並列に実行することで並列化を行った。

頂点 x の始点からの距離を $d(x)$、頂点 x から出る全ての辺の重みの最小値を $E(x)$ とする。まだ探索を行っていない頂点の集合を U とする。このとき、操作 2 で今後更新されないための条件は次に示す式 2.1 の Δ よりも始点からの距離が短いことである。

$$\Delta = min(d(u) + E(u)) : u \in U \tag{2.1}$$

すなわち、ある頂点 v が条件を満たす条件は、$d(v) < \Delta$ である。

GPU で並列化した Dijkstra 法を実行するために、1 回のループにつき minimum カーネル、update カーネル、relax カーネルという 3 つのカーネルに分けて実行している。minimum カーネルは、式 2.1 を各頂点に対して並列に実行することで、Δ を求める。update カーネルは $d(v) < \Delta$ の計算を各頂点に対して並列に行い、次に探索する頂点を決定する。relax カーネルは update カーネルで決めた頂点に対してそれぞれ操作 2 を実行し、距離を更新する。このように 3 つのカーネルに分けることにより、頂点数を越える多数のスレッドを作成することができ、GPU の高い並列性を活かすことができる。

Ortega-Arranz らはこの GPU による並列化により、CPU に対して 13 倍から 220 倍の性能向上に成功した。

本論文では、グラフ型ストアを対象に作成したキャッシュに対して Ortega-Arranz らの高速化手法を適用し、本論文の手法を用いることで既存のグラフ探索高速化手法をグラフ型ストアに応用可能になることを示す。

2.4.3 GPU による幅優先探索の高速化

Merrill らが GPU を用いた幅優先探索の高速化を行った [42]。幅優先探索は、グラフの全ての頂点を探索するアルゴリズムで、始点からのホップ数が少ない順に探索を行う。グラフ探索のためにキューを用意し、始めに始点をキューに加える。その後次の操作を繰り返し、全ての頂点の探索が終了するまで探索を行う。

1. キューの先頭の頂点を取り出す。

2. 選んだ頂点の隣接頂点のうち未探索の頂点を全てキューに追加する。

このアルゴリズムは、全ての頂点を 1 回以上探索すればよいため、並列に探索して一つの頂点を同時に探索を行ったとしても問題はない。そこで、Merrill らは始点からのホップ数が同じ頂点を同時に探索する手法をとった。逐次実行のアルゴリズムとの変更点として、キューの代わりに次に探索する頂点の集合を用いている。その集合を N とするとき、Merril らが行った並列の幅優先探索のアルゴリズムは以下のようになる。

- N の初期値は始点頂点一つのみとする。
- N の頂点全てに対して並列に以下の操作を行う。操作を行う前に、N を空集合にする。
  1. 頂点の隣接頂点を全て選択する。隣接頂点を v とする。
  2. v が未訪問であれば、v を N に含め、v を探索済みにする。

このように、ホップ数毎に並列に実行している。この時、並列処理する頂点数は毎回異なる。この並列化をマルチコア CPU で行った場合、一度に並列処理できる頂点数は少ないが、並列処理に伴うオーバーヘッドは小さい。一方、並列化を GPU で行った場合、一度に並列処理できる頂点数は多いが、CPU から GPU へのデータ転送等が発生するため、並列処理に伴うオーバーヘッドが大きい。そのため、Merrill らは、並列に探索する頂点数が少ない場合はマルチコア CPU、頂点数が多い場合は GPU というように、CPU と GPU の実行を動的に切り替えるハイブリッドな実装により、高速化を行っている。

この実装により、Merrill らは CPU の逐次実行に対して最大 29 倍の性能向上に成功した。

2.4.4 GPU による最小スパニングツリー構築の高速化

Nobart らが GPU による重み付きグラフの最小スパニングツリーを構築するアルゴリズムの高速化を行った [43]。最小スパニングツリーを構築するアルゴリズムはいくつかあるが、そのうち Prim 法を対象としている。

スパニングツリーとは、グラフの全ての頂点を含み、かつ閉路を持たないツリーのことで、最小スパニングツリーはスパニングツリーのうち、辺の重みの合計が最小となるスパニングツリーのことである。

Prim 法は、既にツリーに含まれる頂点の集合を Q とし、初期値として Q に頂点内の任意の頂点を一つ Q に含めた時、以下の操作で表せる。

1. Q に隣接する頂点のうち、Q からの頂点が最も短い頂点とそれを結ぶ辺を Q に加える。
2. Q がグラフの全ての頂点を含む場合、アルゴリズムを終了する。そうでない場合、操作 1 に戻る。

Nobart らは、Prim 法を初期頂点を複数選び、それぞれに対して並列に実行している。以下の操作を並列に実行している。

1. 空集合 P を用意し、探索頂点としてまだ選ばれていない頂点を一つ選択し、P に加える。
2. P に隣接する頂点のうち、P からの頂点が最も短い頂点を v とする。
3. v が探索済みでない場合、P に頂点 v およひ、v への辺を加え、操作 2 に戻る。

4. v が探索済みである場合、P には v への辺のみを P に加え、操作を終了する。

この操作により、それぞれ並列に作成された P がいくつかできる。操作 4 で探索済みであるというのは、並列に探索している他のスレッドで探索されたということである。よって、操作終了時点で P は他のスレッドで作成した集合と隣接している。この各スレッドで作成された集合全ての和集合を作ることにより、全ての頂点を含む最小スパニングツリーが構築される。

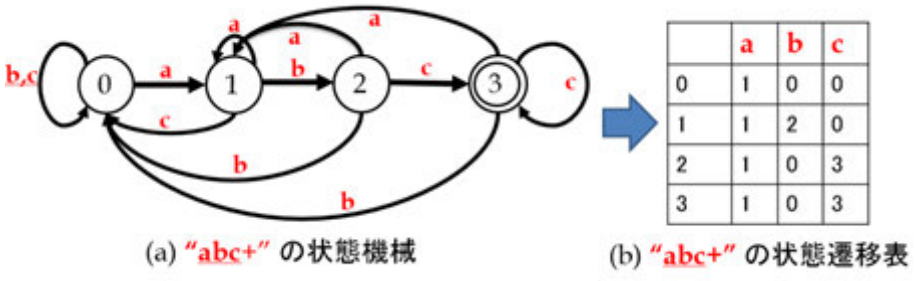
この手法は、任意の数の初期頂点に対して並列に操作を行えるため、GPU で多数のスレッドを作成し、それぞれのスレッドで別の初期頂点から操作を行うことで、GPU の並列性を活かすことができる。

この手法により、Nobart らは CPU の逐次実行に対して最大 14 倍の性能向上に成功した。

このように、グラフ処理アルゴリズムを GPU で高速化した例は多数報告されている。しかし、本論文のように、これらをグラフ型ストアに統合した例は存在しない。

2.4.5 GPU による文字列の正規表現探索の高速化

GPU を用いた正規表現探索の手法として、決定性有限オートマトン (DFA: Deterministic Finite Automaton) を用いた手法が Vasiliadis らによって提案されている [44]。DFA とは、状態と入力によって次に遷移すべき状態が一意に定まる状態機械である。GPU で DFA を扱うために、状態機械を表形式で表す、状態遷移表を作成する必要がある。



	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	1	0	3

図 2.17: “abc+” の状態機械と状態遷移表

図 2.17 に状態機械から状態遷移表を作成する例を示す。図 2.17(a) の状態機械は、入力文字列に “abc+” が含まれるかどうかを調べるものである。初期状態 0 から始まり、文字が入力される度に状態が遷移し、状態 3 になれば、条件を満たす。これを基に作成した状態遷移表が図 2.17(b) である。状態遷移表は、状態毎に各入力が与えられたときの次の状態を示す表である。例えば、状態 0 の時に’a’ が与えられれば状態 1、’b’、’c’ ならば状態 0 が次の状態であることを示す。

この状態遷移表は GPU で扱いやすい配列構造に格納できるため、配列構造に格納後、GPU に転送する。GPU の各スレッドでは、現在の状態を保持し、スレッド毎に文字を 1 文字読み込み、現在の状態と読み込んだ文字から状態遷移表を参照し、次の状態に移行するという処理を行う。複数の文字列に対してこの処理をする場合、文字列毎に独立した処理となる。そのため、文字列毎にスレッドを発行し、並列に行うことで、GPU の並列性を活かし、高いスループットを得られる。

Vasiliadis らは、ネットワークを流れる文字列のストリームに対する正規表現探索を対象アプリケーションとしているため、パケット毎にスレッドを生成し、評価を行った。その結果、GPU で正規表現探索を行った場合、CPU で処理した場合に比べて最大 30 倍の高速化に成功した。

また、GPU での正規表現探索に非決定性有限オートマトン (NFA: Non-deterministic Finite Automaton) を用いる手法も提案されている [45]。NFA は DFA とは異なり、ある状態と入力に対

して次の状態が一意に決定しない状態機械である。DFA を用いる手法の場合、条件として与える文字列の組み合わせが多くになると、状態の数が非常に多くなってしまうという問題がある。対して、NFA では、状態が一意に決まらず、処理が複雑になる代わりに、状態の数の増加を抑えることが出来る。そのため、DFA と NFA を用いる 2 つの手法は、DFA を用いる手法の方が高速ではあるが、NFA を用いる手法の方がメモリ利用効率が良いという関係にある。

さらに、一般的な正規表現探索ではなく、いくつかの規則に限定し、より単純化することで、DFA を用いる手法よりも高速に処理する手法も提案されている [46]。また、GPU を用いた手法を想定はしていないが、正規表現のキーを複数の細かいキーの組み合わせに分け、組み合わせを木構造によって管理する手法 [47] も提案されており、この手法は GPU を用いた手法と併用できると考えられる。また、予めデータセットが与えられ、長時間の前処理とメモリ使用量の多さを許容されている場合にはインデックスを用いて正規表現探索を行う手法も利用可能である [48] が、本論文で対象とするドキュメント指向型ストアの想定とは異なる。

本論文では、対象とするドキュメント指向型ストアにおいて、一度にメモリ効率が問題になるほど複雑なクエリは扱わないため、一般的な正規表現探索を高速に処理できる DFA を用いる手法を採用し、我々が提案するキャッシュに実装する。

2.5 グラフ処理システム

2.4 節において、複数の GPU によるグラフ処理アルゴリズムについて述べたが、これらの手法は各アルゴリズムに特化しており、様々な処理を行うことを想定する場合にもこれらの手法を用いようとすると、それぞれのアルゴリズム毎に実装を行う必要があり、開発負荷が大きくなる。そこで、様々なアルゴリズムを一つのシステムで実行できる、グラフ処理システムが広く用いられている。グラフ処理は主に、ある頂点から他の頂点へと辺を辿って横断し、2 つの頂点、および辺のもつ情報を基に行う計算と、その計算結果の更新の繰り返しによって行われる。この際の横断処理の内容を変更することで、様々なアルゴリズムを実行できる。

表 2.1: グラフ処理システム

システム名	同期手法	処理種別	ノード数
Pregel [49]	バルク同期並列 (BSP)	CPU 処理	複数ノード
GraphLab [50]	非同期	CPU 処理	複数ノード
PowerGraph [51]	BSP と非同期共に対応	CPU 処理	複数ノード
Medusa [52]	BSP	GPU 処理	単一ノード
Gunlock [53]	BSP	GPU 処理	単一ノード

グラフ処理システムとして既に様々なシステムが報告されており、表 2.1 に代表的なグラフ処理システムの特徴をまとめて示す。各システムの違いとして、計算結果の更新の際の各頂点間の同期、および複数ノードに跨る場合は複数の計算機間での同期の手法の違いがあげられる。その同期手法は、計算結果の更新方法によってバルク同期並列 (Bulk Synchronous Parallel: BSP) と非同期の二つに分類される。

BSP は、全ての頂点からの横断と計算処理を一回行う毎にそれらの計算で得られた結果をまとめて更新する手法である。対して、非同期は、各頂点での計算結果をその都度更新する手法である。図 2.18 に BSP を用いたグラフ処理システムにおけるグラフ処理の様子を示す。図に示す通り、BSP においては、グラフ処理がいくつかのステップに分けられる。この例では、各頂点の持つ重みを連接頂点の重みに足す、という処理をしている。赤丸で示した部分が各ステップで横断を行う頂点であり、グラフ全体においてあるステップで処理を行う頂点全ての処理を同一のステッ

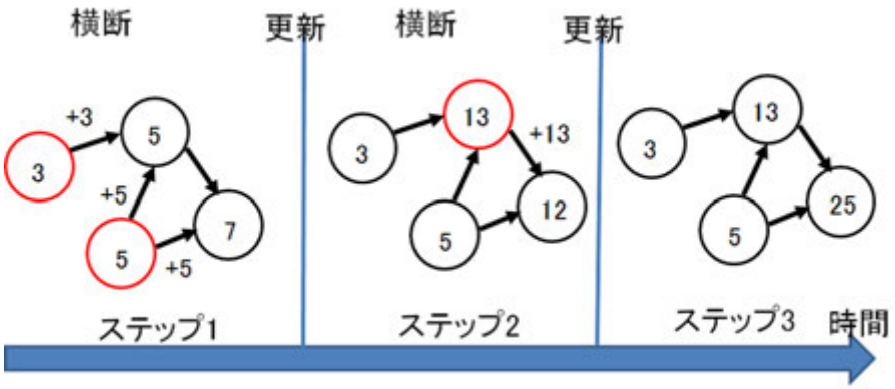


図 2.18: バルク同期並列を用いたグラフ処理システムの処理

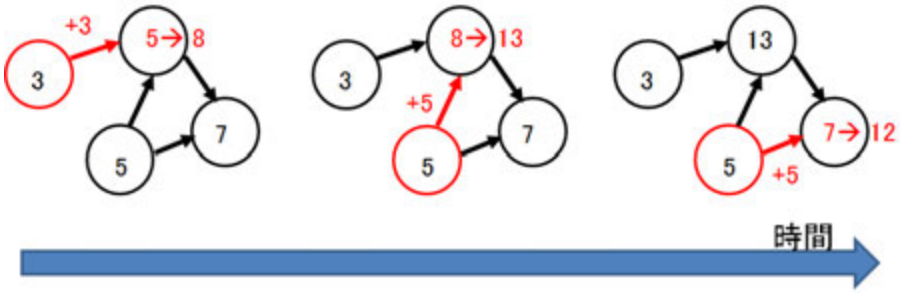


図 2.19: 非同期更新を用いたグラフ処理システムの処理

プで行う。その後、ステップ間の更新において、全ての処理結果を更新する。各頂点は複数の頂点と隣接し得るため、全ての頂点からの横断処理で、複数回の計算が行われる可能性がある。BSP では、この複数回の更新を一度に行えるため、更新回数を抑えることができるが、計算後にまとめて更新するため、ステップと次のステップの間に更新のオーバーヘッドが発生する。

図 2.19 に非同期更新によるグラフ処理システムにおけるグラフ処理の様子を示す。この手法では、BSP のように処理がステップ毎に分けられることなく、横断が行われた直後に更新も行われる。この手法においては、依存関係が無い場合には、次の横断処理の内容を実行している間に更新を行うといったように、計算と更新を重複して実行できる。そのため、非同期では、更新回数は多いが計算と更新の重複によって、更新のオーバーヘッドを隠蔽できる。しかし、更新回数の増加により、BSP の場合よりも更新時間が増加し、一部を計算と重複させたとしても、全体として処理時間が長くなってしまう可能性もあるため、どちらの手法が高速であるかは、対象グラフ、アルゴリズム、ノード数などの環境によって異なる。

本論文が対象とする GPU 処理を行うシステムである Medusa と Gunlock は共に同期手法として BSP を採用している。これは、単一計算機上の GPU 間通信は、高速な PCIe で通信できるため、更新オーバーヘッドが小さいためである。また、GPU 処理では、計算は GPU 上で行い、GPU 間の転送命令は CPU 上で行うため、非同期処理を行うと、CPU から GPU の状態を確認して大量の転送命令を行う必要があり、それに伴うオーバーヘッドが非常に大きくなる。

しかし、本論文では、高速な PCIe で接続された GPU ではなく、PCIe と比較して低速な 10GbE で遠隔接続された GPU の利用も考慮している。この場合、BSP の更新の際に発生する GPU 間の通信オーバーヘッドが大きくなる。そこで、本論文では、GPU 間の通信を部分的に非同期にする手法の提案を行い、CPU からの転送命令の数を抑えつつ、転送オーバーヘッドを抑える手法を提案する。

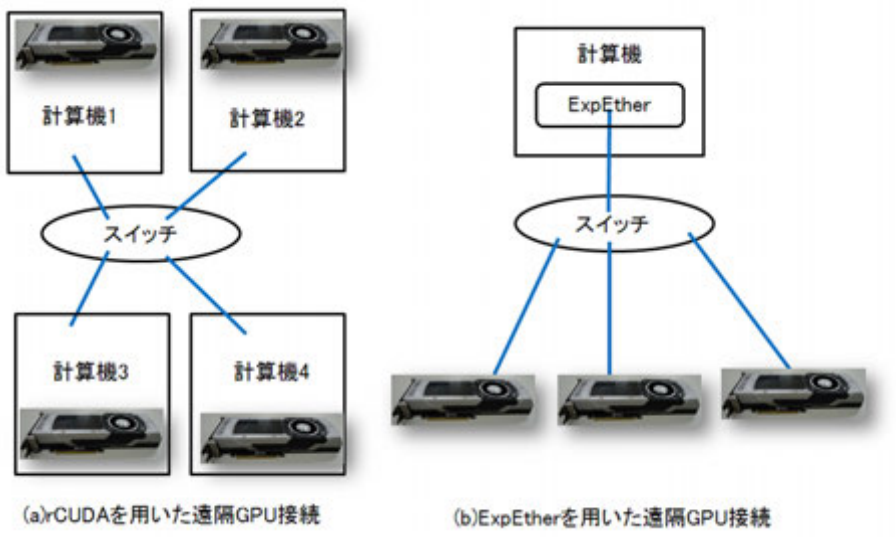


図 2.20: 2 つの遠隔 GPU 接続手法の比較

2.6 遠隔 GPU 接続

2.6.1 遠隔 GPU 接続を実現する二つの手法

本論文では、構造型ストレージのキャッシュの大きさが GPU のデバイスメモリを大きさを越えた場合の性能劣化を防ぐため、複数の GPU へ拡張し、キャッシュを分散することで対処する。しかし、1 台のホストに搭載できる GPU の数はマザーボードの PCIe ポートの数などの要因によって制限される。構造型ストレージのデータ量が大きく、キャッシュの大きさがホストに搭載している複数の GPU のデバイスメモリの合計を越えた場合、単一 GPU において発生していたデバイスメモリを越えた場合の問題と同様の問題が生じる。この問題に対処するために、本論文では、10GbE ネットワークを介して遠隔にある GPU を用いることを提案する。遠隔 GPU を用いることで、マザーボード等の制約なく、多数の GPU を利用可能になり、更なる拡張が可能になる。遠隔 GPU を用いる手法は 2 つ提案されており、本節ではこれらを比較し、どちらが本論文の手法に適しているか検討する。

一つ目の手法は、クライアントサーバーモデルに基づいたソフトウェアサービスを利用する手法である。PCIe で直接接続された GPU をもつ計算機を用意し、rCUDA[54] と呼ばれるサービスを用いて、処理内容を TCP/IP 通信で Ethernet を介して GPU の持つ計算機に送り、その計算機において GPU 処理をすることで、別の計算機に存在する GPU を利用できる。本研究で用いる GPU 向けの開発環境は CUDA であるため、ここでは rCUDA について述べるが、同種のサービスが OpenCL 向けにも提供されている [55]。

二つ目の手法は、どの計算機とも直接接続されていない GPU が多数用意し、それらの GPU を直接 Ethernet を介して利用する手法である。この手法は、NEC ExpEther[56][57][58][59] によって実現されており、ExpEther を用いることで、GPU 等に対する PCIe のパケットを、Ethernet フレームにカプセル化して 10GbE 上を転送できる。

図 2.20 に、2 つの遠隔 GPU 接続手法を用いたシステムの様子を示す。(a) が rCUDA を用いた手法を示しており、4 台の計算機それぞれの PCIe スロットに GPU が接続されており、それらの計算機が Ethernet で接続されている。この図では全ての計算機が GPU を搭載しているが、GPU を持たない計算機がネットワークに参加し、他の計算機の GPU を用いることもできる。この例では、それぞれの計算機が GPU を搭載しているため、基本的に各計算機に搭載されている GPU

を用い、GPU 資源が不足した場合に他の計算機に要求し、他の計算機の GPU を用いることになる。その場合、他の計算機の要求を受けて、各計算機が搭載している GPU へ命令を出すため、あくまで GPU を直接利用しているのは各計算機である。そのため、例えば、計算機 1 が計算機 2、3、4 全てから要求を受けて、それぞれの要求を GPU へ命令を出す、といったように、複数のホストで 1 台の GPU を共有することができる。また、用いる GPU の変更等も容易に可能であり、クエリ毎に異なる GPU へ要求を出す事ができる。よって、rCUDA を用いた手法は、複数のホストで 1 台の GPU を用いたり、GPU の利用が一時的であるような用途に適していると考えられる。実際に、文献 [54] においてもデータセンターにおいて、多数のホストで 1 台の計算機を共有することで各 GPU の利用率を上げ、データセンター全体の GPU 数を削減し、消費電力を下げるといった利用方法が想定されている。

図 2.20(b) には、ExpEther を用いた手法が示されている。ホスト CPU の PCIe スロットに、GPU の代わりに ExpEther が接続されている。その ExpEther を介して、Ethernet ネットワーク上の多数の GPU を利用できる。この時、ホストの計算機上では、各 GPU が PCIe に直接接続されているときと同様に認識されており、直接接続された GPU であるかのように GPU を利用することができる。ExpEther を用いた場合の利点は、計算機と GPU の間に他の計算機が介在しないことである。そのため、rCUDA を用いた手法では生じる TCP/IP の処理などの計算機間での通信によるオーバーヘッドを削減できる。一方で、基本的にある計算機が GPU を占有して用いるため、多数のホストで GPU を共有したり、頻繁に GPU を用いるホストを変更したりといった用途においては、rCUDA の利用が適している。よって、ExpEther を用いた手法は、ある計算機が GPU を占有し、ある程度の期間使い続けるような用途において有効であると考えられる。また、rCUDA の場合は、多数の GPU を用いる場合、多数の計算機が必要であるが、ExpEther を用いる場合には、計算機を用意せずに、多数の GPU を利用する事ができる。

その上で、本論文で提案するキャッシュ手法は、構造型ストレージのキャッシュを多数の GPU に分散して保持する手法である。その場合、一度キャッシュを保持した GPU を使いつづけることになるため、ExpEther を用いる手法が適切であり、本論文における遠隔 GPU 接続は ExpEther を用いた手法で行う。また、rCUDA において、計算と更新をパイプライン化することで、転送オーバーヘッドを削減する手法が提案されている [60]。本論文においても、ExpEther を用いているが、ハッシュ機構によるキャッシュの分散やグラフの非同期更新によって計算と更新を重複させることで転送オーバーヘッドを削減する。

2.6.2 本研究以外の遠隔 GPU 接続の応用例

2.6.2.1 ラックスケールアーキテクチャ

近年、データセンターのような多数の計算機を用いる環境において、従来よりも効率的なアーキテクチャとして、ラックスケールアーキテクチャが注目されている [61][62]。

図 2.21 にラックスケールアーキテクチャの概念図を示す。図 2.21(a) のような従来のアーキテクチャは、ラックに一通りのハードウェア資源がそろった計算機が何台もあり、それら互いに接続されている。対して、図 2.21(b) のラックスケールアーキテクチャでは、それぞれの資源を大量に 1 つのラックに入れ、それらを接続することでラック全体を 1 つの計算機として考える。従来のアーキテクチャでは、計算機単位で保有するハードウェア資源の量が決まっているため、あるハードウェアの必要量が多くなると、そのハードウェアに合わせて計算機を用意しなければならない。本論文が対象とする構造型ストレージにおいても、GPU のメモリ容量の関係で多く必要であり、GPU の数に合わせて計算機を用意するのは非効率である。対して、ラックスケールアーキテクチャでは、各アプリケーションにラック内のハードウェア資源を自由に割り当てられる。

そのため、各アプリケーションに必要な分の資源だけを割り当てる事が出来るため、最も多く使うハードウェアに合わせて計算機を用意する必要がなく、従来のアーキテクチャよりも効率的にハードウェア資源を活用できる。

このようなラックスケールアーキテクチャは、GPU だけでなく、全てのハードウェア資源を遠隔接続したものであり、遠隔 GPU 接続の拡張であると言える。よって、本論文で提案する多数の遠隔 GPU を用いた手法は、ラックスケールアーキテクチャの考え方にも適合している。

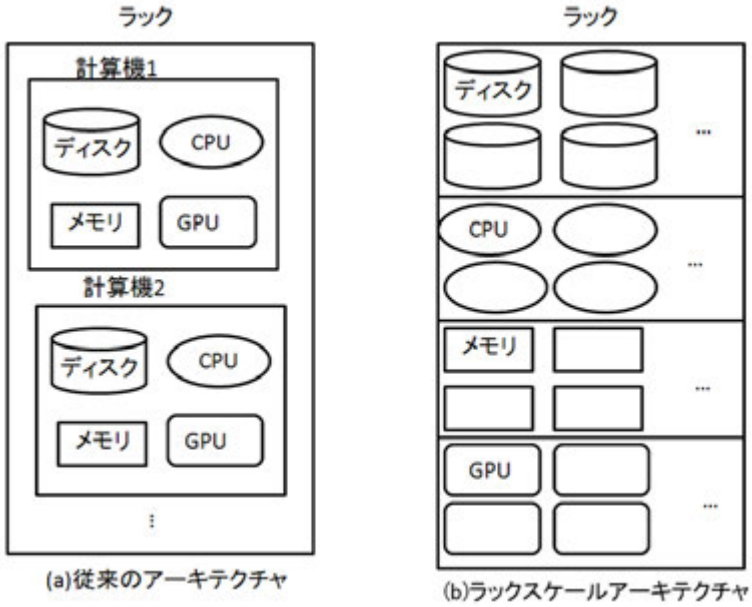


図 2.21: ラックスケールアーキテクチャの概念図

2.6.2.2 遠隔 GPU を用いた VR アプリケーション

近年普及が拡大している VR アプリケーションにおいても、遠隔 GPU 接続を応用することができる [63]。VR アプリケーションは、計算量が大きく、高性能な GPU が要求される。しかし、各利用者毎に、高性能な GPU を用意するのは利用者の負担が大きい。そこで、GPU プールある多数の GPU を利用時にのみ遠隔で利用することにより、全体の GPU 数を削減している。この様な用途の場合、上述した二つの手法の比較では、rCUDA を用いる手法が適していると考えられるが、rCUDA を用いる場合には rCUDA を利用できるようにアプリケーションに追加実装する必要があり、既に VR アプリケーションが普及していることを考えると現実的ではないため、ホストから直接接続しているように認識可能であり、追加実装の必要ない ExpEther を用いる手法が採用されている。

第 3 章

GPUを用いたドキュメント指向型ストアの高速化手法

3.1 システム全体像

ドキュメント指向型ストアのクエリで計算量の大きい処理は、条件に文字列を与えてドキュメントを検索する文字列探索処理である。本章では、本論文の基本的な考え方である特定用途特化の考え方に則り、ドキュメント指向型のボトルネックとなるクエリである文字列探索処理に特化し、かつ GPU 処理に適した構造のキャッシュであるドキュメントキャッシュを提案する。特に、2.2.2 節で述べたように、文字列探索処理のうち B+tree を用いたインデックスが利用できない正規表現探索クエリがボトルネックとなる。そのため、ドキュメントキャッシュのデータ構造として、文字列探索のうち、特に正規表現探索に特化した構造を提案する。ドキュメントキャッシュに対して GPU で正規表現探索処理をすることにより、ドキュメント指向型ストアの枠組みを維持しつつ、正規表現探索クエリを高速化できる。ドキュメントキャッシュは正規表現探索を主な対象としているが、文字列の完全一致探索などのその他のクエリに対しても利用可能である。そのため、完全一致探索などにも流用する場合を考え、正規表現探索における性能を最大化しつつ、完全一致探索クエリのスループットも正規表現探索の性能を損ねない範囲で高めることを目的としている。

図 3.1 に提案するドキュメント指向型ストアの全体像とクエリの流れを示す。本システムは、ドキュメント指向型ストア (図中では、対象とするシステムである MongoDB)、ホストメモリ上のドキュメントキャッシュ、10GbE 経由で遠隔接続された GPU 上のドキュメントキャッシュの 3 つの構成要素からなる。ドキュメントキャッシュでは、文字列探索に必要な情報のみを抽出して構築するが、文字列探索に必要な情報はドキュメントのバリューに相当する部分である。ホストの主記憶や二次記憶よりも GPU のデバイスメモリは小さいため、バリューのみを抽出してもドキュメントキャッシュの大きさが GPU のデバイスメモリの大きさを越えてしまうことが考えられる。そのため、本論文では、GPU のキャッシュを 10GbE で遠隔接続された複数の GPU に拡張することで、1 台の GPU のデバイスメモリを越える大きさのキャッシュも扱えるようにする。また、遠隔接続していることによって、ホストの PCIe の数によって GPU の数が制限されることもなく、より大規模なキャッシュを扱うことができる。本システムでは、MongoDB とドキュメントキャッシュは同一の計算機上で動作し、そのホストが複数の遠隔 GPU を確保してその GPU 上にキャッシュを転送、GPU で検索処理を行うことを想定している。ホストがドキュメントキャッシュ用に用いる GPU の数はデータ量や負荷に応じて変更する事もでき、ドキュメントキャッシュは GPU 処理に適したデータ構造であり、それをハッシュ機構を用いて分割し、複数の GPU に分散して転送される。各クエリに対しての動作は以下のようになっている。

- 更新クエリでは、まず MongoDB を更新し、続いてドキュメントキャッシュを更新する。

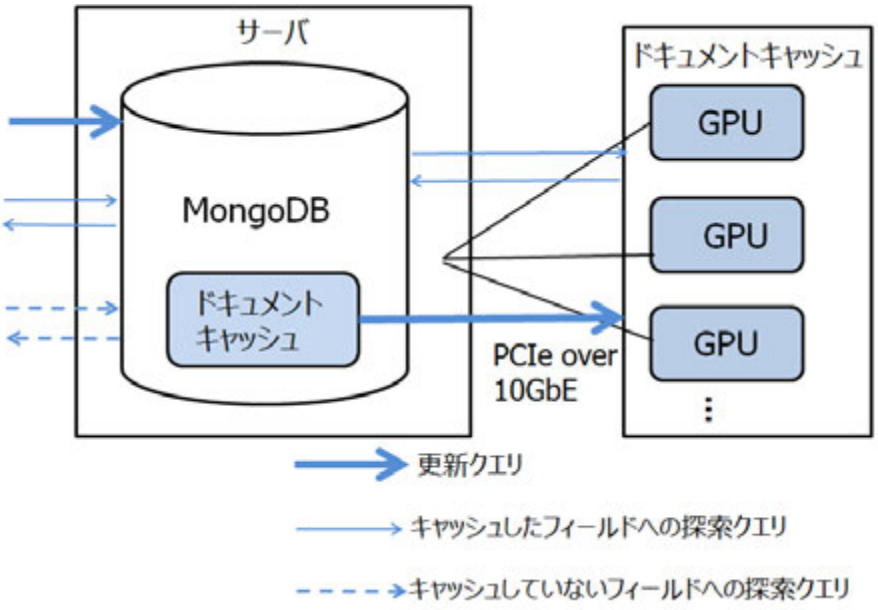


図 3.1: 提案するドキュメント指向型ストアの全体像とクエリの流れ

- ドキュメントキャッシュを作成していないフィールドに対する探索クエリでは、ドキュメントキャッシュや GPU を使わずに MongoDB で処理し、結果を返す。
- ドキュメントキャッシュ作成済みのフィールドに対する探索クエリでは、GPU を用いたドキュメントキャッシュを探索する。この際の探索結果は MongoDB を介してユーザーに返される。

本章では、まず 3.2 節と 3.3 節で複数 GPU への分割を考慮しない、分割前のドキュメントキャッシュのデータ構造および、キャッシュに対する処理方法を述べる。3.4 節で分割前のドキュメントキャッシュを用いた性能を評価し、その評価結果を踏まえて分割前のドキュメントキャッシュの問題点を述べる。3.5 節でその問題点を解消するためのハッシュ機構を用いた分割および複数 GPU への分散手法を述べ、複数 GPU への拡張を行い、3.7 節で分散ドキュメントキャッシュの性能を評価する。

3.2 ドキュメントキャッシュのデータ構造

ドキュメントキャッシュは、MongoDB の各フィールド毎のキャッシュであり、各フィールドのバリューのみを抽出して生成する。ドキュメントキャッシュのデータ構造は、1 つもしくは 2 つの一次元配列からなる構造である。配列が 1 つであるか 2 つであるかは、フィールドの型によって異なる。整数型等、バリュー毎にデータサイズが変わらないフィールドでは 1 つ、文字列型等、バリュー毎にデータサイズが異なるフィールドでは 2 つの配列である。配列が 1 つの場合は、その配列はフィールドの各バリューを抽出したバリュー配列であり、2 つの場合には、バリュー配列において各バリューが何番目から始まるかを示す配列 PTR を追加する。

ドキュメントキャッシュが抽出するデータ量は、各フィールドのバリューのデータ量の合計であるが、ドキュメントの更新によって、この大きさは動的に変化する。そのため、ドキュメントキャッシュの大きさも生成時に固定するのではなく、更新に備えて空き領域を確保しておき、空き領域がなくなった際により大きな配列として再構成することで、データ量の動的変化に対応する。空き領域を用いた更新への対応方法は、次章で述べるグラフ型ストアの更新への対応にも適

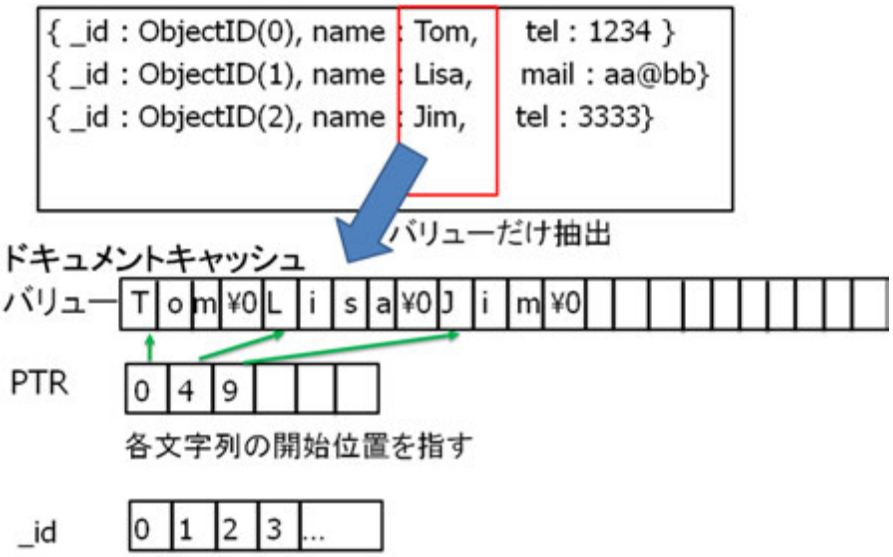


図 3.2: ドキュメントキャッシュの構成例

用できる。グラフ型ストアにおいては、グラフの各頂点に隣接辺としてデータが追加されるため、頂点毎に空き領域を確保するが、ドキュメント指向型では、ドキュメントキャッシュの末尾にのみ空き領域を確保すればよい。また、この空き領域を用いた更新の処理方法については、3.3.2 節で述べる。

図 3.2 にドキュメントキャッシュの構成例を示す。図 3.2 の上半分が複数のドキュメントを保存した MongoDB のデータ構造を簡略化したものを表しており、下半分がドキュメントキャッシュを表している。MongoDB は複数のキーバリューの組で一つのドキュメントを表し、さらにそのドキュメントが複数あるという構造である。ドキュメントキャッシュは、複数のドキュメントの中からあるフィールドのバリューを抽出し、一次元の配列に格納した構造である。キーとなっているフィールド名および型は同じフィールドならば共通であるため、ドキュメント毎に保存する必要はなく、フィールド毎にドキュメントキャッシュの型と名前として保存すればよい。キーをドキュメントキャッシュ全体で 1 つで管理できるため、キーのメモリ使用量は、c をフィールド名の長さとするとき、ドキュメント数にかかわらず $(c+1)$ バイトとなる。図 3.2 の例では、Name フィールドのバリューである「Tom」、「Lisa」、「Jim」がそれぞれ抽出され、ドキュメントキャッシュに格納される。

もし、このように文字列を一次元配列に格納した際、バリュー配列のみでキャッシュを構成する場合、各ドキュメントの区切りを知るためには配列を探索して文字列の終端記号を調べる必要があり、並列探索には適さない。そのため、ドキュメントキャッシュでは、バリュー配列に加えて各文字列の先頭位置を示すポインタの役割をする配列 PTR を設け、2 つの配列で構成している。この例では、配列 PTR は「Tom」、「Lisa」、「Jim」の先頭アドレスである 0、4、9 をそれぞれ表している。また、図 3.2 の状態では、配列 PTR の隣り合う要素を用いて各文字列の長さを取得できるが、3.3.2 節に示すドキュメントキャッシュの更新によって、配列 PTR の値が変化するため、常に配列 PTR から文字列の長さを取得できるわけではない。そのため、配列 PTR が存在するドキュメントキャッシュにおいても、各文字列の終端記号は必要である。前述したとおり、数値等の固定長の型の場合は、ポインタ用の配列は必要なく、バリュー配列のみで構成される。

この例では、一つのフィールドに対してのみドキュメントキャッシュを作成したが、同様にして他のフィールドに対するドキュメントキャッシュも作成できる。その場合、複数のドキュメントキャッシュがメモリ上に存在する。探索を行う際は、この複数のドキュメントキャッシュからクエリで条件を指定しているフィールドの配列を選択して探索を行う。例えば、図 3.2 で name、tel、

mail のフィールドに対してそれぞれドキュメントキャッシュを作成し、name と tel のフィールドを探索するクエリが与えられた場合は、作成した 3 つのドキュメントキャッシュのうち、name と tel のドキュメントキャッシュのみを選択し、探索を行う。

3.3 ドキュメントキャッシュに対する処理

ドキュメントキャッシュでは、以下の 3 種類の処理を利用可能である。

1. 作成 (再構成):ドキュメントキャッシュを MongoDB からデータを抽出して作成する、あるいは更新に伴い空き領域が不足した既存のドキュメントキャッシュを新たに再構成する処理。
2. 更新:作成したドキュメントキャッシュに対するバリューの追加、削除、更新処理。
3. 検索:ドキュメントキャッシュに含まれる全バリューから条件に合うものを GPU を用いて検索する処理。

本章の各節では、これらの 3 つの処理についてをそれぞれ述べる。また、ドキュメントキャッシュは、バリューの型によって、構成要素となる配列の数が異なるが、配列が 1 つの場合については、配列が 2 つの場合の配列 PTR に対する処理と同様に処理できるため、本章における各処理は、2 つの配列からなる場合のみ述べる。

3.3.1 ドキュメントキャッシュの作成および再構成

Algorithm 1 にドキュメントキャッシュの作成および再構成の処理を示す。ドキュメントキャッシュの再構成は、他の処理から呼び出して実行する処理であるため、この処理を行う関数を DDBmake とし、他の処理から呼び出す際にはこの関数を用いる。1 行目と 2 行目の定数 N および S はユーザーが事前に定める定数である。N は確保する空き領域の要素数であり、N を大きくすると、空き領域を大きく確保するため、メモリ使用量が大きくなるが、ドキュメントキャッシュの再構成の頻度が下がる。S はバリュー配列を配列 PTR の 1 要素に付きどの程度確保するかを決める定数であり、N と同様に、大きいほどメモリ使用量が大きく、再構成の頻度が下がる。N は更新の頻度が高ければ大きく設定し、S は各バリューの平均的な大きさを基に定める。3 行目の L_V と L_{PTR} は再構成の時のみ入力として与えられ、古いドキュメントキャッシュの大きさを表す。再構成は、3.3.2 節で述べる更新処理の一部で呼び出され、L_V と L_{PTR} の値も更新処理と同時に計算される。また、ドキュメントキャッシュの作成の場合は古いドキュメントキャッシュが存在しないため、それぞれ 0 となる。

Algorithm 1 の 6 行目以降が実際に作成および再構成をしている処理部分であり、まず、6 行目と 7 行目に示すように、古いドキュメントキャッシュと空き領域の合計の大きさ分の領域を確保し、配列を作成する。それ以降は処理が作成か再構成かによって分岐し、その条件判定を 8 行目の IF 文にて行っている。L_V が 0 であれば、古いドキュメントキャッシュが存在せず、作成の処理である。作成処理では、10 行目と 11 行目に示すように、MongoDB から対象のフィールドのバリューを 1 件ずつ取得し、ドキュメントキャッシュに追加している。11 行目の DDBadd は 3.3.2 節で述べるバリュー追加の処理を表す関数である。L_V が 0 でない場合は処理が再構成であることを示しており、14 行目に示すように古いドキュメントキャッシュをコピーする。16 行目と 17 行目では L_V、L_{PTR} を、作成もしくは更新後のドキュメントキャッシュに合わせて更新している。

Algorithm 1 で定義した L_V、L_{PTR}、$V[]$、$PTR[]$ は、以降で述べる処理でも共有して保持する。

Algorithm 1 DDBmake:ドキュメントキャッシュの作成および再構成

1: $N \leftarrow$ ドキュメントキャッシュの空き領域の大きさを表す定数
2: $S \leftarrow$ バリュー配列を配列 PTR の何倍の大きさに確保するかを表す定数
3: L_V、$L_{PTR} \leftarrow$ それぞれバリュー配列と配列 PTR の大きさ、ドキュメントキャッシュ作成しの初期値は 0、再構成の場合は古いドキュメントキャッシュの大きさを表す
4: $V[] \leftarrow$ ドキュメントキャッシュのバリュー配列
5: $PTR[] \leftarrow$ ドキュメントキャッシュの配列 PTR
6: 配列 $V[]$ を配列長 $N \times S + L_V$の配列として作成
7: 配列 $PTR[]$ を配列長 $N + L_{PTR}$の配列として作成
8: **if** $L_V = 0$ **then**
9: **for** $i = 1$ to キャッシュ対象のドキュメント数 **do**
10: $MongoDB$ のクエリを用いて対象フィールドのバリューを取得、取得したバリューを v とする
11: $DDBadd$ 関数でドキュメントキャッシュにバリューを追加詳細は $Algorithm2$ にて記述
12: **end for**
13: **else**
14: 古いドキュメントキャッシュのバリューのうち、削除されていないバリューを $V[]$、$PTR[]$ にコピー
15: **end if**
16: $L_V \leftarrow N \times S + L_V$
17: $L_{PTR} \leftarrow N + L_{PTR}$

3.3.2 ドキュメントキャッシュの更新

ドキュメントキャッシュにおける更新処理は、バリューの追加、削除、更新の 3 種類からなる。追加と削除は MongoDB においてドキュメントが追加、削除された時に行われ、更新は既存のドキュメント中でキャッシュしたフィールドが更新された時に行われる。3 種類の処理のうち、削除と更新処理では、MongoDB が更新された時にドキュメントキャッシュの対応する箇所を探し、更新する必要がある。ドキュメントキャッシュを探索するコストを削減するため、MongoDB の各ドキュメントがドキュメントキャッシュの何番目に保存されているかを示すルックアップテーブルを作成する。ルックアップテーブルは、MongoDB の各ドキュメントの_id をキー、そのドキュメントに対応するドキュメントキャッシュの配列の要素番号をバリューとするキーバリューストアと同様の単純な構造である。

Algorithm 2 にドキュメントキャッシュの追加処理を示す。Algorithm 1 でこの処理を用いるため、この処理を関数 DDBadd とする。まず、関数の入力として、追加するバリュー v が与えられ、その長さを始めに取得する。その後の 3 行目の条件はドキュメントキャッシュの空き領域が十分であるかを判定しており、空き領域が不足している場合には、Algorithm 1 で示した DDBmake 関数を用いてドキュメントキャッシュを再構成する。

7 行目以降がドキュメントキャッシュにバリューを追加する処理である。7 行目と 8 行目がドキュメントキャッシュの 2 つの配列への追加であり、バリュー配列に v を保存し、v の先頭位置は p_V であるため、対応する配列 PTR に p_V を記録する。この追加でバリューの末尾の位置が変化するため、9 行目と 10 行目で p_V と p_{PTR} の更新を行っている。Algorithm 2 で定義した p_V、p_{PTR} は、以降で述べる処理でも共有して保持する。

Algorithm 3 にドキュメントキャッシュの削除処理を示す。入力として、ルックアップテーブルから取得したバリューの位置を i として与える。ドキュメントキャッシュは、GPU で探索するた

Algorithm 2 DDBadd:ドキュメントキャッシュの追加処理

1: $v \leftarrow$ 本処理で追加するバリュー
2: v の長さを取得、その長さを l とする
3: p_V、$p_{PTR} \leftarrow$ それぞれバリュー配列と配列 PTR の空き領域の先頭を表すポインタ、ドキュメントキャッシュ作成時の初期値は 0
4: **if** $p_{PTR} = L_{PTR}$ または $p_V + l \geq L_V$ **then**
5: $DDBmake$ 関数でドキュメントキャッシュを再構成
6: **end if**
7: 対応するルックアップデーブルに p_{PTR}を保存
8: $V[p_V]$ から l 要素に v を保存
9: $PTR[p_{PTR}] \leftarrow p_V$
10: $p_V \leftarrow p_V + l$
11: p_{PTR}をインクリメント

Algorithm 3 ドキュメントキャッシュの削除処理

1: $i \leftarrow$ 削除するバリューの配列 PTR の位置
2: $V[PTR[i]] \leftarrow null$

めのキャッシュであるため、探索の際に得られる結果がバリューを削除した時と一致すれば、実際にバリューを削除せずに削除したとみなすことができる。そのため、探索の際に条件を満たさなくなる値を null 値とし、バリューの先頭をこの値に書き換えることで削除処理とした (本論文で評価した文字列処理では、終端文字を null 値として用いた。)。これにより、GPU での探索の際に各バリューが削除済みかどうかに関わらず配列 PTR を基に全てのバリューを探索すればよく、各バリューが削除済みかを判定する分岐処理を削減できる。よって、削除処理に伴いドキュメントキャッシュの変更は、2 行目に示す null 値の代入のみとなる。実際の削除は、ドキュメントキャッシュの再構成の際に、削除済みの値を新しいドキュメントキャッシュにコピーしないことでまとめて行う。

Algorithm 4 にドキュメントキャッシュの更新処理を示す。ドキュメントキャッシュの更新は、新しいバリューを追加し、更新対象のバリューの配列 PTR の値を書き換えることで行う。7 行目がバリューの追加を表しており、ドキュメントキャッシュの追加処理と同様にバリュー配列にバリューを追加している。7 行目の更新により、バリューの先頭位置が p_v に変化したため、8 行目では、配列 PTR の値を p_v に書き換えている。この書き換えにより、更新前のバリューは配列 PTR から参照されなくなり、探索結果に影響を与えないため、削除する必要はなくなる。古いバリューの削除は、ドキュメントキャッシュの再構成の際にコピーしないことでまとめて実行される。また、この処理ではバリューの追加を行うため、4 行目から 6 行目に示すように、空き領域が足りない場合にはドキュメントキャッシュの再構成を行う。

3.3.3 ドキュメントキャッシュにおける検索処理

本論文では、MongoDB における主な検索処理である文字列探索処理を実装、評価した。具体的には完全一致探索、正規表現探索を実装、評価したが、完全一致探索については、単純に与えられたクエリの文字列と各バリューを比較するのみの単純な CUDA カーネルであるため詳細は省き、本節では正規表現探索のカーネルについて主に述べる。

本論文では、GPU 処理の実装に NVIDIA 社が提供する GPU 向けの開発環境である CUDA を用いる [64]。

Algorithm 4 ドキュメントキャッシュの更新処理

1: $i \leftarrow$ 更新するバリューの配列 PTR の位置
2: $v \leftarrow$ 更新後のバリュー
3: v の長さを取得、その長さを l とする
4: **if** $p_V + l \geq L_V$ **then**
5: 　$DDBmake$ 関数でドキュメントキャッシュを再構成
6: **end if**
7: $V[p_V]$ から l 要素に v を保存
8: $PTR[i] \leftarrow p_V$
9: $p_V \leftarrow p_V + l$

ドキュメントキャッシュに対する探索を行う CUDA カーネルは、文献 [44] に基づいて DFA を用いた正規表現探索を行う。DFA における状態遷移を GPU 上で処理するため、まず、二次元配列で構成される状態遷移表を作成する。状態遷移表は、入力を文字と現在の状態とし、その入力に対する次の状態を表す表である。本論文では、入力文字は ASCII コードで表すものとし、表の大きさは 128 × 状態数 で実装した。正規表現探索では、条件とする正規表現に対して状態遷移表を作成し、正規表現を満たす文字列を探索した時に遷移する状態を受理状態とすることで、探索後の状態が受理状態か否かで文字列の条件判定を行う。

GPU を用いた正規表現探索では、ホスト側で状態遷移表を作成し、GPU に転送を行った後、CUDA カーネルを実行し、ドキュメントキャッシュ内の各文字列が正規表現を満たすかどうかの判定を行う。

ドキュメントキャッシュの作成時にはドキュメントキャッシュ全体、更新時には更新部分を GPU に転送し、それを GPU のグローバルメモリに保持する。そのため、探索クエリ毎にドキュメントキャッシュの転送を行う必要はない。また、状態遷移表については、カーネル内で複数回の参照が行われるため、GPU のシェアードメモリに格納する。Algorithm5 にドキュメントキャッシュに対する正規表現探索の CUDA カーネルを示す。各 CUDA スレッドがドキュメントキャッシュの各バリューを探索対象とするため、ドキュメントキャッシュの配列 PTR と CUDA スレッドの数を一致させて実行する。7 行目で配列 PTR から処理すべきバリュー配列の位置を特定し、8 行目から 10 行目でバリュー配列の文字列を入力として、状態遷移を文字列が終端記号に達するまで繰り返し探索している。12 行目から 18 行目に示す様に、探索後の状態が受理状態であれば、その文字列は条件を満たすとして、結果用配列に ID が格納される。カーネル実行後に、結果用配列の索引番号である n 要素分だけ結果用配列をホスト側に転送することで、条件を満たす文字列の ID からなる配列が得られる。このスレッド ID は、ドキュメントキャッシュの配列 PTR の位置を表している。

本提案手法による処理は、正規表現探索の探索内容であるキーは利用者が決めるため、クエリ毎に与えられたクエリを基に DFA を作成し、正規表現探索を行っている。一方で、既存の GPU を用いた手法 [44] では、予め多数のキーからなる DFA を作成して GPU にキャッシュし、そこに探索対象であるデータを転送して正規表現探索を行っている。これは本提案手法における想定と探索対象とキーの関係が逆であり、ドキュメントキャッシュでは、探索対象をキャッシュされており、キーがクエリとして与えられるのに対し、既存手法の想定では、キーがキャッシュされており、探索対象が後に与えられる。既存手法のような想定の場合には、多数のキーをキャッシュする必要があるため、それらから作成される DFA の大きさが問題となるが、ドキュメントキャッシュにおいては 1 つのキーに対する DFA であるため、DFA によるメモリ消費量は極めて小さい。しかし、ドキュメントキャッシュにおいても、バッチ処理を行う場合には多数のキーから DFA を作ることも可能であり、その場合のメモリ使用量は、平均キー長 19.53、キー数 50,000 において

Algorithm 5 正規表現探索の CUDA カーネル

1: $V[] \leftarrow$ ドキュメントキャッシュのバリュー配列
2: $PTR[] \leftarrow$ ドキュメントキャッシュの配列 PTR
3: $tid \leftarrow$ スレッド ID、スレッド数が配列 PTR の要素数と一致
4: $DFA[][] \leftarrow DFA$ を元に作成した状態遷移表、$DFA[i][j]$ で入力が文字 i、状態 j の時の次の状態を表す
5: $s \leftarrow$ 現在の状態、初期値は 0
6: $R[] \leftarrow$ 結果格納用配列、n を索引番号とし、n の初期値は 0、n は全スレッドで共有する
7: $i \leftarrow$ スレッドで現在処理している $V[]$ の位置、初期値は $PTR[tid]$
8: **while** $V[i] \neq' \backslash 0'$ **do**
9: $s \leftarrow DFA[V[i]][s]$
10: i をインクリメント
11: **end while**
12: **if** s が受理状態 **then**
13: 不可分操作開始
14: $R[n] \leftarrow tid$
15: $n \leftarrow n + 1$
16: 不可分操作終了
17: **end if**

799.76MB となり [44]、ドキュメントもキャッシュすることを考慮すると、これを越えるような場合には複数回に分けて処理を行うか、よりメモリ容量の大きいホストメモリ上で CPU を用いてバッチ処理を行うべきである。

3.4 ドキュメントキャッシュの性能評価

分割前のドキュメントキャッシュに関する評価では、以下の 3 種類のクエリの評価を行った。

1. 単一フィールドに対する文字列完全一致クエリ
2. 単一フィールドに対する文字列の正規表現探索クエリ
3. 新しいドキュメントを追加するクエリ

ドキュメントキャッシュは文字列以外の型 (例:整数型やタイムスタンプ等) も扱えるが、これらの型に対するクエリの実行時間は文字列の完全一致クエリの実行時間と傾向が一致しているため、本論文ではこれらの結果は省く。正規表現探索クエリでは、正規表現探索の一種のランダムな文字列を含む探索である部分一致探索によって評価を行った。それぞれのクエリにおいて、オリジナルの MongoDB と提案手法の比較を行った。オリジナルの MongoDB の評価では、公平に比較するため、MongoDB を tmpfs を用いてメモリファイルシステム上で動作させた。

また、評価にもちいた CPU は Intel Xeon E5-2637v3 で動作周波数は 3.5GHz、メモリ容量は 128GB である。GPU は GeForce GTX 980 を用いた。用いた GPU の諸元を表 3.1 に示す。対象とするドキュメントストアとして、MongoDB 2.6.6 を用い、GPU 処理の実装には CUDA version 6.0 を用いた

表 3.1: NVIDIA GeForce GTX 980 の主な諸元

性能項目	GeForce GTX 980
コア数	2,048
コアクロック	1,126MHz
メモリバス幅	256bit
メモリバンド幅	224GB/s
メモリ容量	4GB

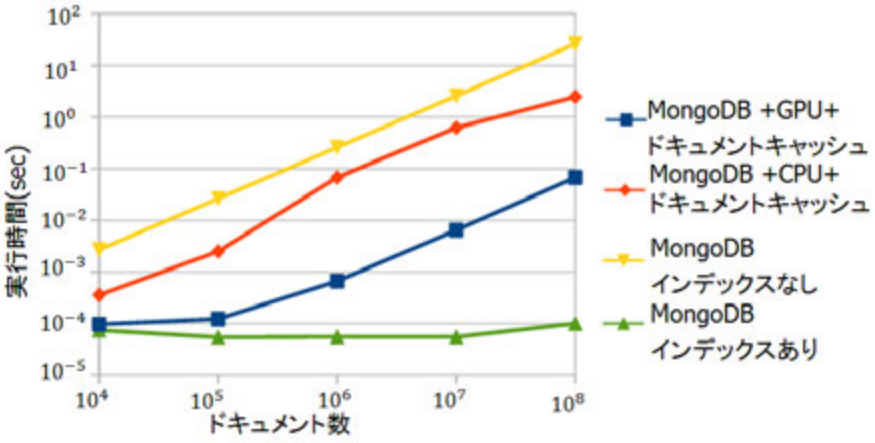


図 3.3: 単一フィールドに対する文字列の完全一致の実行時間

3.4.1 単一フィールドに対する文字列完全一致クエリ

対象ドキュメントとして、_id と 8 文字の文字列の 2 つのフィールドのみをもつドキュメントを作成した。単一フィールドに対する文字列完全一致をこのドキュメントに対して実行した。以下に文字列完全一致クエリの例として、“field1” というフィールドのバリューが文字列 “abc” と一致しているかどうかを探索するクエリを示す。

```
find({field1:"abc"})
```

本評価では、この時の条件を “abc” ではなく、ランダムに作成した 8 文字の文字列として探索を行った。

図 3.3 は、ドキュメント件数と単一フィールドに対する文字列完全一致探索の実行時間の関係を表す両対数グラフである。GPU+ドキュメントキャッシュ(ドキュメントキャッシュに対して GPU で探索を行った場合) は、インデックスを用いない MongoDB に対してドキュメント件数に関わらず高速化に成功しており、1 億件の時は 458 倍の高速である。対して、インデックスによる探索と比較すると、ドキュメント件数が少ない場合はほぼ同等の性能であるが、ドキュメント件数が増えるにつれてインデックスによる高速化のほうが高速になる。これは、ドキュメント件数を N としたとき、インデックスを用いない場合の計算量が $O(N)$ なのに対して、インデックスを用いた場合の計算量が $O(\log N)$ であり、ドキュメント件数の増加に対してインデックスによる探索の計算量の増加が小さいためである。そのため、ドキュメントキャッシュを用いることで、完全一致探索の性能が低下し、完全一致探索がボトルネックとなることでシステム全体の性能が低下してしまう可能性がある。

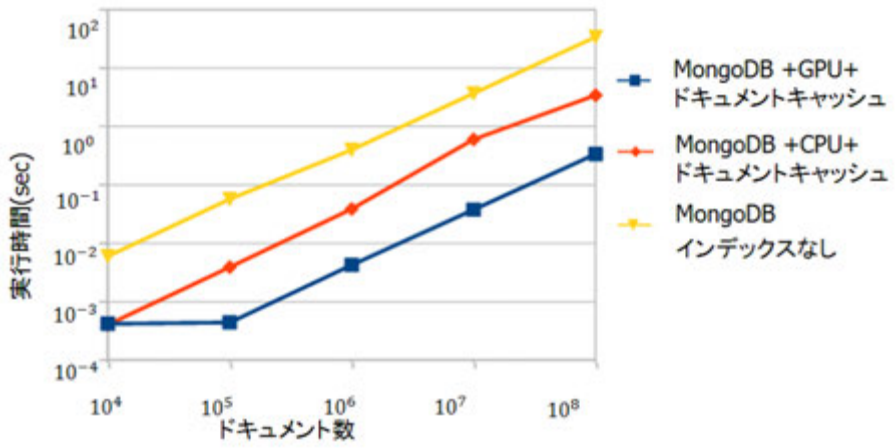


図 3.4: 単一フィールドに対する正規表現探索の実行時間

3.4.2 文字列の正規表現探索クエリ

対象ドキュメントとして、_id と 16 文字のフィールドからなるドキュメントを作成した。このドキュメントの文字列フィールドに、正規表現探索として、文字列の部分一致を実行した。以下に、このようなクエリの例として、“field1” フィールドのバリューに “abc” を含むかどうかを探索するクエリを示す。

```
find({field1:{$regex:/abc/}})
```

本評価では、“abc” の代わりに、ランダムに作成した 4 文字の文字列を用いた。

図 3.4 は、ドキュメント件数と正規表現探索クエリの実行時間の関係を表す両対数グラフである。正規表現探索では、インデックスが使用不可能であるため、インデックスありの評価はない。GPU+ドキュメントキャッシュは、インデックスを用いない場合やドキュメントキャッシュの CPU+ドキュメントキャッシュ(ドキュメントキャッシュを CPU で探索した場合) より高速であり、1 億件の場合に MongoDB に対して 101 倍、ドキュメントキャッシュを用いた CPU 探索よりも 10 倍高速である。ドキュメントキャッシュを用いた CPU 探索では、GPU 処理と同様に DFA を用いた正規表現探索を行っている。ドキュメント件数が 1 万件の場合のみ、GPU+ドキュメントキャッシュと CPU+ドキュメントキャッシュがほぼ同じ時間となっているが、これはドキュメント件数が少ないために GPU で実行する際のスレッド数が少なく、GPU の並列度を活かせていないためである。

3.4.3 書き込み性能

MongoDB にインデックスを用いた場合とドキュメントキャッシュを用いた場合の書き込み性能を比較した。書き込みクエリとして、6 フィールドのドキュメント (_id フィールドと 8 文字の文字列からなるフィールド 5 つ) が 1 億件保存されている状態に新しいドキュメントを追加するクエリを実行し、書き込みスループットを評価した。

図 3.5 に MongoDB にインデックスを用いた場合とドキュメントキャッシュを用いた場合の書き込み性能を示す。この書き込み性能は、単にキャッシュに書き込みを行った場合の性能を示したものではなく、MongoDB に書き込みを行い、そのドキュメントを抽出してキャッシュを更新するという一連の流れを行った場合の性能を示している。横軸はインデックスあるいはドキュメントキャッシュを作成した_id フィールド以外のフィールド数 (_id フィールドは必ずインデックスが作

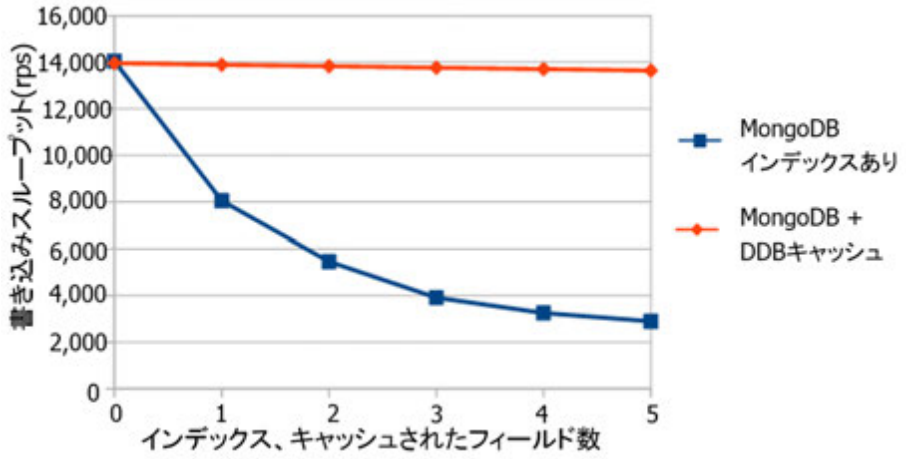


図 3.5: インデックスとドキュメントキャッシュの書き込みスループットの比較

成される。) を示し、縦軸は 1 秒あたりに書き込めるドキュメント数を rps (requests per second) で示す。図 3.5 から、ドキュメントキャッシュを用いた場合は、キャッシュしたフィールド数が増えても書き込み性能の低下は少なく、ドキュメントキャッシュを設けることによる書き込みオーバーヘッドが小さいことがわかる。このことから、ドキュメントキャッシュは書き込みのオーバーヘッドを抑えつつ、ボトルネックとなる正規表現探索を高速化できると言える。また、MongoDB を用いてインデックスを利用した場合も示しており、インデックス数が増加するにつれて性能が低下しているが、この要因には MongoDB の実装上の問題が多分に含まれるため、B+tree に比べて本手法の書き込み性能が高いことを示しているわけではない。

3.4.4 CPU-GPU 間のデータ転送時間

3.4.2 節で用いたドキュメントを、CUDA の cudaMemcpy 関数を用いて 1 億件転送したときの CPU-GPU 間の転送時間は 0.21 秒であり、これは正規表現探索クエリの 64%に相当する。しかし、1 億件を全て転送するのはドキュメントキャッシュを作成する時のみであり、通常は更新部分のみを転送する。よって、更新時の CPU-GPU 間のデータ転送時間は 0.21 秒よりも大幅に小さくなる。もし、クエリ実行の度に 1 億件全てを転送したとしても、GPU+ドキュメントキャッシュでの正規表現探索はオリジナルの MongoDB よりも 62 倍高速である。しかし、正規表現探索以外のクエリに関しては、一つのクエリの実行時間が小さいため、この転送が発生した場合には、実行時間のを転送時間が占める割合がさらに大きく、スループットの低下率も大きくなる。ドキュメントキャッシュの大きさが GPU のデバイスメモリの容量を越える場合、クエリの実行毎に上述した CPU-GPU 間の転送時間がクエリ毎のオーバーヘッドとして発生する。

3.4.5 分割前のドキュメントキャッシュの問題点

分割前のドキュメントキャッシュには、インデックスを用いた高速化手法と比較して、以下の 2 種類の拡張性の問題があることが分かる

1. インデックスが利用可能なクエリを含む全てのクエリで、n をドキュメント数としたとき、計算量が $O(n)$ である。
2. キャッシュの大きさが GPU のデバイスメモリよりも大きい場合、CPU-GPU 間の転送によって大幅にスループットが低下する。

1 つ目の問題は、インデックスを用いた手法と比較して計算量が大きくなるという問題である。完全一致探索クエリの評価で示した通り、B+tree 等のインデックスを用いた場合に比べて計算量が大きく、ドキュメント数が多くなると、インデックスを用いる手法に比べて大幅に性能が低下してしまっている。その結果、ドキュメント数が多い場合にドキュメントキャッシュを設けることで完全一致探索に関しては性能が低下してしまう。インデックスを用いずにドキュメントキャッシュを用いた場合には完全一致探索がボトルネックとなり、インデックスを用いた場合と比較してシステム全体の性能を低下させてしまう可能性がある。

2 つ目の問題は、一般的に、GPU のデバイスメモリがホストメモリよりも小容量であることによって発生する。MongoDB のインデックス作成は、ホストメモリの大きさを上限として作成することができるが、ドキュメントキャッシュでは、1 つの GPU にドキュメントキャッシュを格納して処理する場合、GPU のデバイスメモリの大きさにキャッシュ出来る大きさが限られてしまう。GPU のデバイスメモリの大きさを越える場合は、クエリの実行毎にドキュメントキャッシュの一部を転送、処理することを繰り返すことで処理できるが、この際の転送オーバーヘッドは転送時間の評価で示した通り非常に大きく、高速化率が大きく低下する。

このように、分割を行わないドキュメントキャッシュはドキュメント数が増え、キャッシュするデータサイズが大きくなった場合の拡張性に乏しい。本論文では、これらの問題に対処するため、ハッシュ機構を用いてドキュメントキャッシュを分割し、複数 GPU に分散させることを提案する。複数 GPU に分散することで、GPU のデバイスメモリを越える大きさのキャッシュも扱えるようになる。さらに、分割に用いるハッシュ値をインデックスとして利用可能にすることで、完全一致探索や前方一致探索のクエリの探索範囲を限定可能にし、計算量の問題にも対処できるようにする。このハッシュ機構を用いた分割と複数 GPU への割り当ては、次節で述べる。

3.5 ドキュメントキャッシュの複数 GPU への分散

3.5.1 ハッシュ機構を用いたドキュメントキャッシュの分割

本節では、3.4.5 節で述べたドキュメントキャッシュの拡張性に関する問題点を解決するため、以下の 2 つの目的を達成するハッシュ機構を提案する。

1. ハッシュ機構で用いるハッシュ値を用いて完全一致探索や前方一致探索のクエリの探索範囲を限定可能にし、それらのクエリの計算量を削減する。

2. ドキュメントキャッシュを複数 GPU に分散させて保持可能にし、探索時のそれぞれの GPU 負荷も分散するようにする。

1 つ目の目的を達成するためには、同じ探索範囲に属するデータ群が全て同じハッシュ値 を持つ必要がある。本論文では、文字列の先頭数文字をそのままハッシュ値とし、先頭の数文字が同じデータ群は全て同じハッシュ値をとるようにすることで、この条件を満たした。例えば、頭文字 1 文字のみがハッシュ値であるとすると、‘a’ から始まる文字列は全てハッシュ値も ‘a’ となり、‘a’ 以外から始まる文字列のハッシュ値が ‘a’ となることはない。そのため、完全一致探索の際の条件が ‘a’ から始まる文字列との一致であった場合、ハッシュ値が ‘a’ となる文字列のみを探索範囲とすることができる。

このようにハッシュ値で探索範囲を限定できるようにした時、1 つのハッシュ値に対応するデータの集合を**ブロック**と定義する。ブロックに含まれるデータの数は、ハッシュ値に対応するデータ数であり、それぞれのブロックに含まれるデータ数は異なる。そのため、ブロックを処理単位として GPU で処理する場合、各ブロックの処理負荷が異なってしまい、2 つ目の目的である負荷分散に適さない。

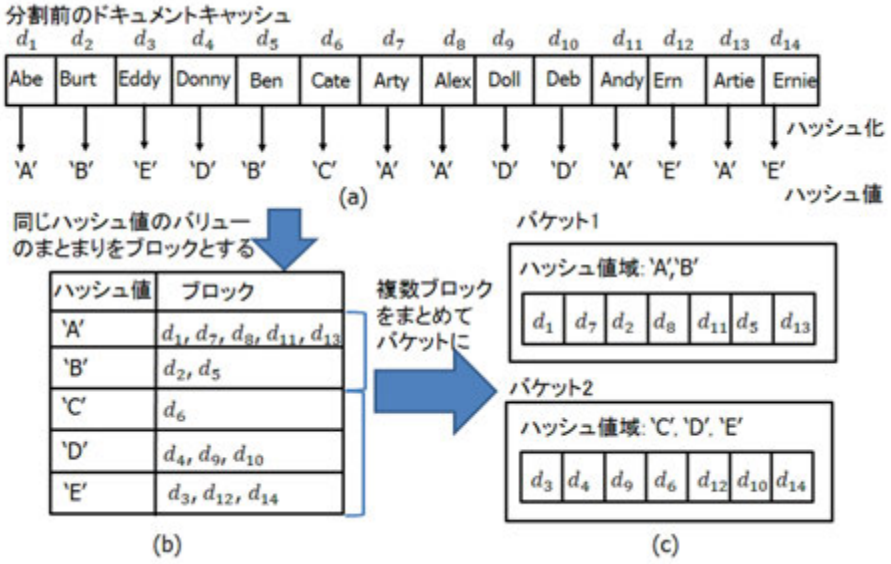


図 3.6: ハッシュ機構におけるブロックとバケットの関係

そのため、本論文では、不均一な大きさのブロックを複数まとめて均一に近いまとまりを作成し、これを GPU で処理する際の処理単位とする。このまとまりの事を**バケット**と定義する。また、各バケットを構成する複数のブロックに対応するハッシュ値の集合を**ハッシュ値域**と定義する。このバケットを 1 つの単位として、複数のバケットを各 GPU に割り当てる (割り当ての詳細は 3.5.2 節で述べる。)。

図 3.6 に本ハッシュ機構におけるブロックとバケットの関係をまとめた図を示す。図では、単純化のため、データは d_1 から d_{14} とし、ハッシュ化によって得られるハッシュ値を A、B、C、D、E の 5 つとする。図の上側の (a) に示すのが単純化した分割前のドキュメントキャッシュ、および各データをハッシュ化した時のハッシュ値である。図 3.2 では、1 マスは 1 文字を表していたが、ここでは、単純化のために 1 マスが文字列からなるデータを表しており、図中の各マス中の文字列がそれぞれデータ d_1 から d_{14} に相当する。分割前のドキュメントキャッシュは 3.2 章のデータ構造を単純化して示したものであり、これをハッシュ関数を用いてハッシュ化して得られるハッシュ値を各データの下側に示している。このように、複数のデータが同じハッシュ値を取り、同じハッシュ値を持つデータの集合がブロックである。各ハッシュ値とそれに対応するブロックを図 3.6(b) に示す。各ブロックに含まれるデータの数は不均一であり、これらを複数まとめてバケットを作成することで、バケットに含まれるデータ数を均一に近づける。図 3.6(c) が複数のブロックをバケットにまとめたものである。この例では、ハッシュ値 A、B のブロックをまとめてバケット 1 とし、C、D、E のブロックをまとめてバケット 2 を作成している。それぞれのハッシュ値域に対応するデータがバケットに格納されており、各バケットの大きさは均一になっている。各バケットのデータ構造は 3.2 章で述べたドキュメントキャッシュと同じ 2 つの配列からなる構造とする。すなわち、ドキュメントキャッシュと同様の構造の配列がバケットの数と同数存在し、ハッシュ値域に応じて各バケットにデータを追加することで、ドキュメントキャッシュをバケットに分割できる。また、バケット内のデータが満たすべき条件はハッシュ値域だけであり、ブロック毎にまとまっている必要はない。全体のデータ構造の変化の流れをまとめると、全てのデータが格納された巨大なドキュメントキャッシュを、ハッシュ値域の条件を満たすデータのみを格納する小さなドキュメントキャッシュである各バケットに分散させたという流れである。このデータ構造において、書き込みを行う場合は、書き込むデータのハッシュ値を求め、そのハッシュ値を含むバケットへ書き込みを行う。

Algorithm 6 バケットの分割

1: $A \leftarrow$ 分割対象のバケット
2: $B \leftarrow$ 新しく追加するバケット
3: H_A、$H_B \leftarrow$ それぞれ A、B のハッシュ値域
4: $h_A \leftarrow h_A \in H_A$を満たすハッシュ値
5: x_A、$x_B \leftarrow$ それぞれ H_A、H_Bの要素数
6: **if** $x_A = 1$ **then**
7: 　分割不可、全体を再構成
8: **end if**
9: **for** $i = 1to\lfloor x_A/2 \rfloor$ **do**
10: 　最大の h_Aを H_Bに追加し、H_Aから削除
11: **end for**
12: $x_A \leftarrow \lceil x_A/2 \rceil$//分割後の H_Aの要素数
13: $x_B \leftarrow \lfloor x_A/2 \rfloor$//分割後の H_Bの要素数
14: A に含まれていた全てのデータを取り出し、改めて追加

図 3.6 では、各バケットでデータ数が均等になっているが、書き込みクエリによってデータ数は動的に変化するため、各バケットのデータ数も動的に変化し、時間の経過と共に不均一になってしまう。そのため、バケットのハッシュ値域の設定も動的に変更できなければならない。これには、各バケットに含まれるデータ数の最大値を定数で予め定め、それを越えた場合にバケットをさらに 2 つに分割することで対処する。その時の最大値には、キャッシュしたドキュメント数、あるいは、データのバイト数の合計のいずれかを用いる。

バケット分割の擬似コードを Algorithm 6 に示す。ここでは、あるバケット A をバケット A と B に 2 分割している。バケットのハッシュ値域が 1 つのハッシュ値の時、それ以上分割することは出来ないため、ハッシュ値の設定の際に満たすべき条件は、1 つのハッシュ値に対応するデータが設定した最大値よりも多くならないことである。ドキュメント数の増加により、ハッシュ値域が 1 つのハッシュ値のバケットを分割する必要がある場合は、6 行目から 8 行目に示すとおり、ハッシュ値をより細かい粒度のものに変更し (文字列の場合はハッシュ値とする文字数を増やす)、全体の再構成を行う。9 行目から 11 行目では、ハッシュ値域の半分をバケット A からバケット B に移すことで、ハッシュ値域を 2 分割している。12 行目と 13 行目で新しく分割後のハッシュ値域の要素数を代入し、この値は再びバケットの分割を行う際に用いる。最後に、バケット A に含まれるデータを削除し、改めて追加しなおすことで、追加の際に新しいハッシュ値域を基にバケット A、B に分けて格納され、データの分割が行える。

ドキュメントキャッシュ全体を新しく作成する、または再構成する操作は、ドキュメントが 0 件の状態に大量にドキュメントを追加する操作とみなせるため、初期状態として、バケット A 一つ、H_A はハッシュ値全てとし、大量のドキュメント追加に伴い Algorithm 6 を何度も実行し、多数のバケットへの分割を行うことで、ドキュメントキャッシュ全体の作成、または再構成とする。

3.5.2 分割後のドキュメントキャッシュの複数 GPU への割り当て

3.5.1 節で述べたように、分割後のドキュメントキャッシュの処理単位であるバケットの大きさの最大値を定めて分割を行うことで、各バケット毎の処理負荷を均一に近づけている。そのため、各 GPU に割り当てるバケットの数が均等に近ければ、各 GPU に処理負荷を分散できる。複数 GPU に分散が必要な規模のデータを扱う場合、GPU 数よりもバケット数が多くなるため、各バケットをランダムに GPU に割り当てるだけで負荷分散が行える。具体的には、割り当てのため

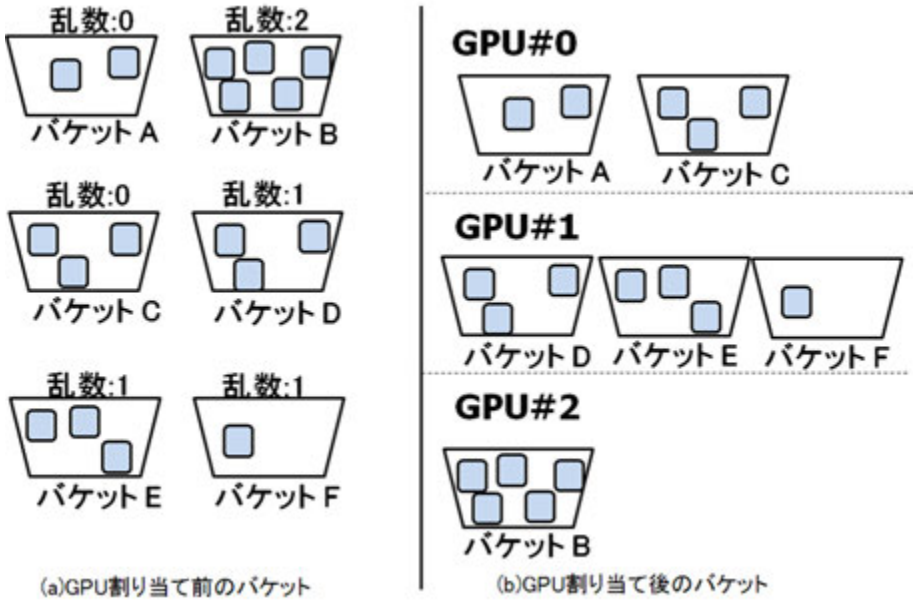


図 3.7: バケットの GPU 割り当て

に、各バケットに GPU 数に合わせた乱数 (例:GPU 数が 3 であれば 0〜2 の整数の内のどれか) に基づいた整数値を保持させ、保持する乱数と同じ ID の GPU へバケットを割り当てることでランダム割り当てを行う。また、この手法であれば、バケットの分割によるバケットの追加の時に既存のバケットの GPU 割り当ての変更の必要がなく、新しく追加するバケットを新たに GPU に割り当てるだけでよく、バケット追加の際のオーバヘッドが小さい。

図 3.7 に GPU 割り当て前後のバケットの例を示す。この例では、図 3.7(a) に示す 7 つの割り当て前のバケットが図 3.7(b) の様に 3 つの GPU へ割り当てられている。割り当て後、バケットはドキュメントキャッシュの再構成が起こるまでは同じ GPU に割り当てられる。

また、遠隔 GPU 環境においては、動的にシステムが確保する GPU の数を変更することが想定される。GPU を追加する場合には、全てのバケットから、全てのバケット数/*GPU* 数 をランダムに選び、新しい GPU へ割り当てるバケットとする。GPU 数を削減する場合には、削減するバケットに割り当てられていた全てのバケットをランダムに他の GPU へ割り当てる。

3.5.3 分割後のドキュメントキャッシュの更新

分割後のドキュメントキャッシュにおいても、分割前のドキュメントキャッシュと同様に、作成、更新、検索の 3 つの処理ができる。そのうち、作成についてはドキュメントキャッシュの分割して多数のバケットにすることを意味しており、3.5.1 節で述べた分割によって行える。

更新処理については、複数のバケットに分割されたドキュメントキャッシュも、各バケットのデータ構造がドキュメントキャッシュと同じ配列構造であるため、3 種類の更新処理 (追加、削除、更新) に対応するバケットを指定し、3.3 章で述べた更新処理と同様の処理を行うことで、各更新処理を実行できる。

Algorithm 7 に分割後のドキュメントキャッシュの更新処理を示す。この処理は、バリューのハッシュ値に対応するバケットに対して、3.3 章で述べた 3 つの更新処理を実行する処理である。また、更新処理と同時に、3 行目と 5 行目に示すように、バケットに含まれるバリューの数の管理を行う。このバリュー数が 3.5.1 節で述べたバケットの大きさの最大値を超えた場合には、Algorithm 6 を用いてバケット A を 2 つに分割する。

Algorithm 7 分割後のドキュメントキャッシュの更新処理

1: 更新するバリューのハッシュ値 h を取得する
2: $A \leftarrow$ をハッシュ値域に含むバケット
3: $n \leftarrow A$ に含まれるバリュー数
4: バケット A に対して更新処理 (追加、削除、更新) を行う
5: 処理が追加であれば n をインクリメント、削除ならば n をデクリメントする
6: n がバケットの大きさの最大値を超えた場合、バケット A を分割する

Algorithm 8 分割後のドキュメントキャッシュに対する検索処理

1: $v \leftarrow$ 検索クエリ
2: **if** v が探索範囲限定可能 **then**
3: 　v のハッシュ値を求め、ハッシュ値をハッシュ値域に含むバケットに対して $CUDA$ カーネルを実行
4: **else**
5: 　全てのバケットに対して正規表現探索の $CUDA$ カーネルを実行
6: **end if**

3.5.4 分割後のドキュメントキャッシュに対する検索処理

分割後のドキュメントキャッシュに対する検索処理も、更新処理と同様に、対象とするバケットに対して 3.3.3 節で述べた探索処理を実行することで、各バケットの探索が行える。検索処理と更新処理で異なる点は、対象となるバケットが単一のバケットだけでなく、すべてのバケットとなり得る点である。具体的には、文字列の完全一致や前方一致などのハッシュ値の特徴を用いて探索範囲を限定できるクエリでは、探索対象となるバケットは単一のバケットであるが、それ以外のクエリでは、全てのバケットを探索する必要がある。

Algorithm 8 に分割後のドキュメントキャッシュに対する検索処理を示す。2 行目に示す通り、まず始めにクエリの種類に応じて、対象とするバケットを選択する。クエリが完全一致探索など、探索範囲がハッシュ値によって限定可能である場合には、3 行目に示すようにハッシュ値からバケットを特定し、文字列比較などの CUDA カーネルを実行する。探索範囲が限定不可能な正規表現探索の場合には Algorithm 5 で示した正規表現探索の CUDA カーネルを全てのバケットに対して実行する。

3.5.5 スキーマレス構造への対応

ドキュメント指向型ストアのデータ構造は、ドキュメント毎にフィールドの有無が変わるスキーマレス構造、例えば、あるドキュメントが A と B というフィールドを持っているが、他のドキュメントはフィールド C しかないといった構造であるため、ドキュメントキャッシュもこれに対応できなければならない。

MongoDB では、各ドキュメント毎に存在するフィールドの種類は異なるが、全てのドキュメントが主キーとして_id フィールドを用いるため、_id フィールドは必ず全てのドキュメントに存在する。ドキュメントキャッシュにおいても、_id フィールドから作成した配列を他のフィールドから作成した配列を参照する際の主キーとして用いる。_id から他のフィールドを参照するためには、1 つのドキュメントにつき、各フィールドのバケット ID とバケット内で何番目にあるかを記録する必要がある。そのため、_id フィールドでは、バリューを保存するドキュメントキャッシュ以外に (フィールド数 $\times$ 2) 個の配列を用意し、フィールド毎に_id に対応するバケット ID とバケット

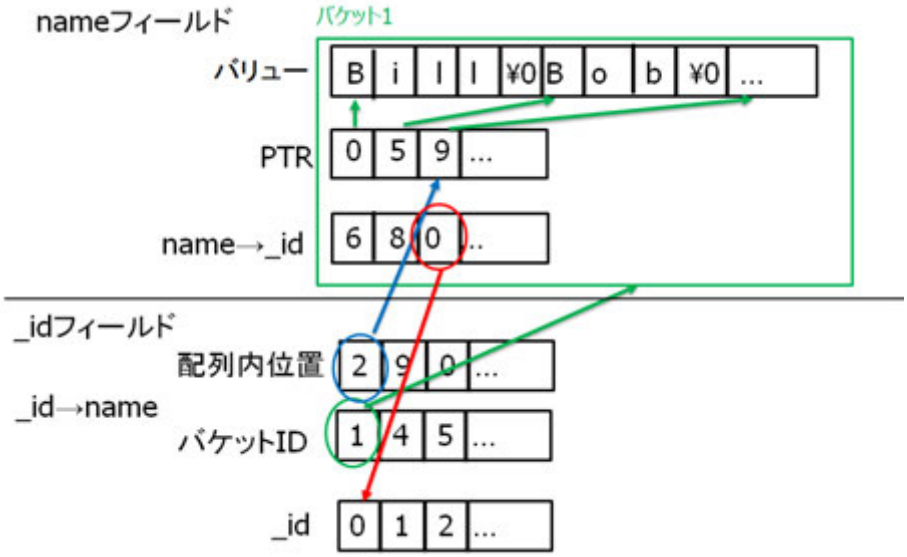


図 3.8: 異なるフィールドのドキュメントキャッシュ間の関係

内の場所を記録する。また、逆に_id以外のフィールドでは、各ドキュメント毎に対応する_idが保存されている場所を記録する配列を作成する。すなわち、これらの追加した配列が同じドキュメント内の_idフィールドと他のフィールドの関係を保存している。この追加の配列を用いることで、ドキュメント内のあるフィールドから、他のフィールドを参照できる。

この配列を追加し、異なるフィールドのドキュメントキャッシュ間の関係を含むデータ構造を図3.8に示す。図3.8では、_idフィールドとnameフィールドがキャッシュされている例を表している。図の上半分がnameフィールドの構造を表しており、図では、分割された後の1つのバケットのみ図示しているが、フィールド全体としてはこれと同様の構造のバケットが多数存在している。nameフィールドにおいて、追加されているのは、nameフィールドから_idフィールドへのポインタの役割をする配列である。この配列はホストメモリ上に保存され、対応する_idの位置を保存している。また、図の下半分は_idフィールドの構造を表している。_idフィールドにおいては、それぞれのドキュメントに対して、バケットIDとバケット内の配列PTRの何番目に位置するかという2つのポインタを表す配列を持つ。

また、ドキュメントキャッシュとして全てのフィールドがキャッシュされていない場合もあり、キャッシュされていないフィールドを参照する場合は、_idをMongoDBに返し、MongoDBを介してキャッシュされていないフィールドを参照する。

これらの構造をまとめたドキュメントキャッシュ全体の構造を図3.9に示す。ドキュメントキャッシュは、図3.9の下に示すように、配列構造となっており、この配列は複数のドキュメントから、特定のフィールドのバリューのみを抽出して作成したものである。さらに、ハッシュ機構を用いて細かく分割され、複数台のGPUに分散される。_idフィールドの分散は行わず、同じ_idのドキュメントは、_idからのポインタをたどることで各フィールドのバリューがどこに分散されているか分かる。

3.6 ドキュメントキャッシュにおけるクエリの流れ

本章では、正規表現探索に特化したドキュメントキャッシュの構造、および探索や更新等の処理内容、複数のGPUへの分散方法などのドキュメントキャッシュの各要素を述べてきた。本節で

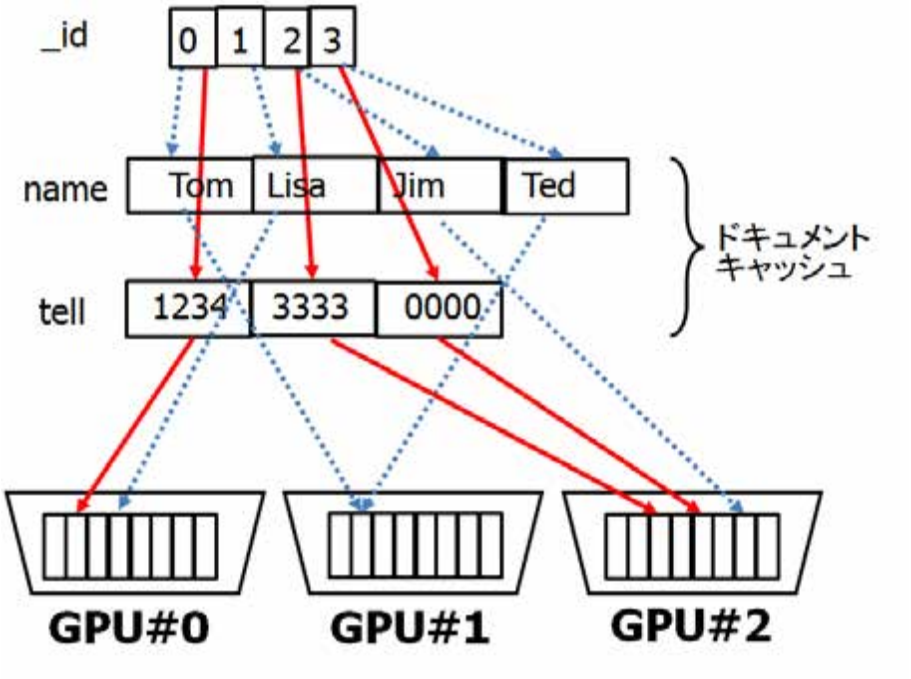


図 3.9: ドキュメントキャッシュの複数 GPU への分散

は、これらの内容を踏まえて、クエリを実行する際に全体としてドキュメントキャッシュがどのような流れで処理を行うかを整理する。

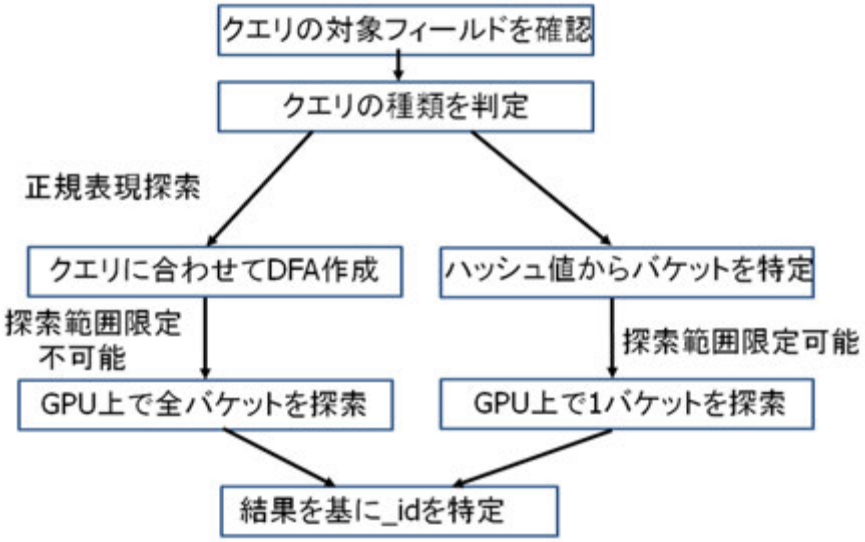


図 3.10: ドキュメントキャッシュにおけるクエリ処理の流れ

図 3.10 に、ドキュメントキャッシュを用いたクエリ処理の流れをフローチャートで示す。まず、与えられたクエリの内容から、検索対象となるフィールドを読み取り、そのフィールドがドキュメントキャッシュとしてキャッシュされているかどうかを確認する。キャッシュされていなければ MongoDB を用いて探索を行い、キャッシュされていれば、対象となるドキュメントキャッシュに対してそれ以降の処理を行う。キャッシュを用いるかどうかの判断を最初に行うことで、キャッシュされていない場合にキャッシュに対する処理を最小限にでき、オーバーヘッドを削減できる。

次に、クエリの種類判定を行う。クエリの種類がハッシュ値を用いて探索範囲限定不可能な正

規表現探索である場合は、クエリの内容をあわせた DFA の作成を行い、作成した DFA を全ての GPU に転送し、全ての GPU を用いて全てのバケットを探索する。クエリがハッシュ値を用いて探索範囲を限定できる完全一致探索などのクエリの場合は、ハッシュ値を用いて探索対象のバケットを特定し、対象のバケットを保存している GPU に対して DFA を転送し、その GPU において 1 バケットを探索する。この際のバケットの探索以外の処理は、全てホスト上の処理となる。最後に、クエリの種類に関わらず、GPU より返答された結果を基に、各フィールドのドキュメントキャッシュから_id へのポインタを表す配列を用いて、条件を満たすドキュメントの_id を特定する。全てのクエリにおいて実行される一連の流れが以上の通りであり、もしクエリによって_id 以外のフィールドが要求されていた場合はこれに加えて_id から他のフィールドの内容を取得する。

3.7 ドキュメント指向型ストアにおける分散キャッシュの性能評価

本評価では、ハッシュ機構によって分散したドキュメントキャッシュを遠隔接続した 3 台の GeForce GTX 980 を用いてオリジナルの MongoDB と比較した。遠隔 GPU への接続では、ExpEther10G を用いた。その際の CPU-GPU 間帯域は 2 本の 10Gbps の Ethernet ケーブルを用いて接続しているため、20Gbps である。また、実際のシステムでは、遠隔 GPU とホストの間にスイッチが入ることが想定されるが、ここでは単純化のため、スイッチは用いずに評価を行った。比較対象として、直接 PCIe と接続した 3 台の GeForce GTX 980 との比較との比較も行った。

評価するクエリは、3.4 節と同じく、完全一致、正規表現、書き込みの 3 種類のクエリの評価を行った。また、正規表現探索においては、本提案手法の実用性を示すために、ランダムに生成した 8 文字のドキュメントを用いた評価に加えて、文字列の長さを変えた時の評価、および 2 種類の実データを用いた場合の評価を行った。2 種類の実データの評価の比較、およびランダムに生成した文字列の評価と比較するために、2 種類の実データから 1000 万件のドキュメントを抽出し、ランダムな文字列の評価と合わせて評価を行った。用いたデータセットは Wikipedia [65] のデータセットであり、データセット名および平均文字数を表 3.2 に示す。各データセットの呼称は、以降では”commonswiki”と”wikidatawiki”にそれぞれ省略する。また、ドキュメントキャッシュを用いた評価における処理は、特に断りのない限り 3.6 節で述べたクエリの一連の流れを全て実行した場合を示している。この一連の流れは、MongoDB において、クエリから_id を取得する流れと同様の流れであるため、MongoDB の性能をドキュメントキャッシュの主な比較対象としている。また、全ての評価において、ハッシュ値としては先頭の 3 文字を用いた。

表 3.2: 実データを基にしたドキュメントの諸元

データセット名	平均文字数
commonswiki-20171120-pages-articles-multistream-index.txt	71
wikidatawiki-20171120-pages-articles-multistream-index.txt	31

3.7.1 バケットサイズとスループットの関係

ここでは、キャッシュの分散の際に用いるバケットの大きさを変化させた時のスループットの変化を評価する。3.4 節では、実行時間を評価対象としていたが、ハッシュによる分散を行ったことで、完全一致クエリが並列に実行できるようになったことから、実行時間ではなく、スループットを用いて評価した。クエリの性能評価に加えて、遠隔 GPU を用いた場合に実行可能である GPU の追加及び削減を行った場合のバケットの大きさ毎の実行時間も評価する。ここでのド

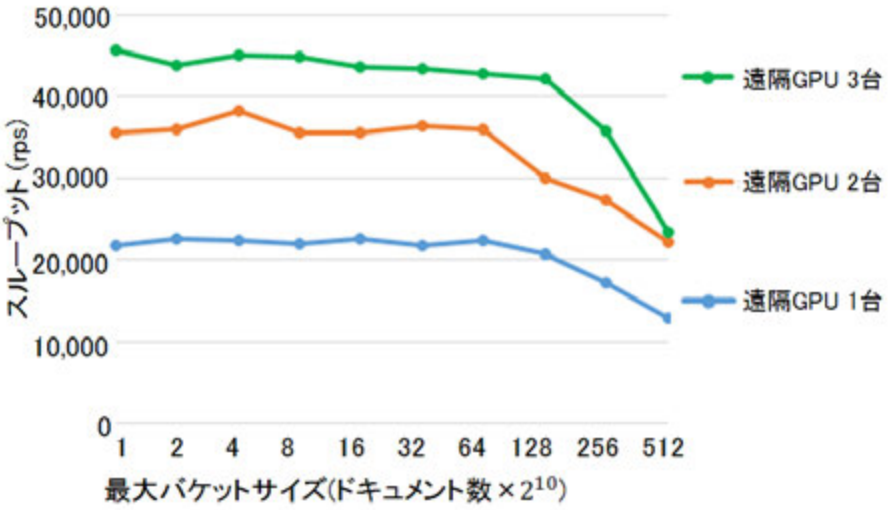


図 3.11: バケットの大きさを変化させた時の文字列完全一致クエリのスループット

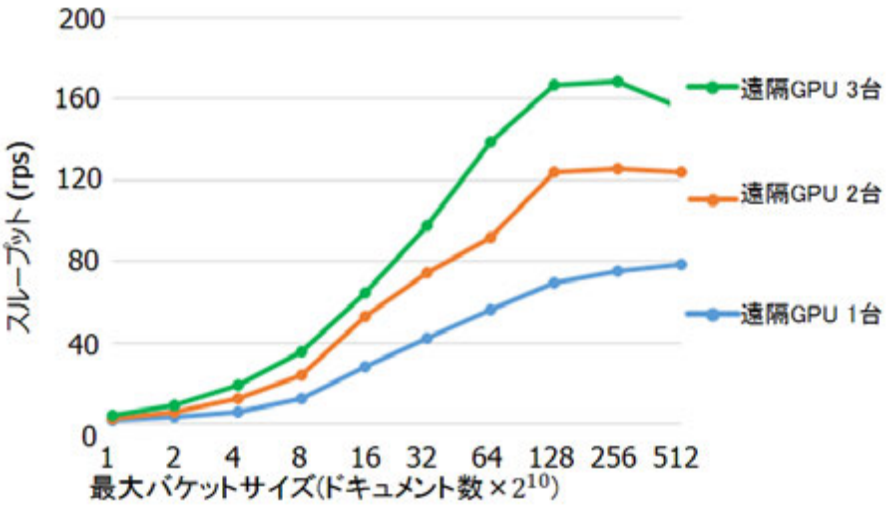


図 3.12: バケットの大きさを変化させた時の正規表現探索クエリのスループット

キュメント数は 1,000 万とし、ドキュメントの中身は 3.4 節での単一フィールドの探索と同じ_id と 8 文字の文字列の 2 つのフィールドのみをもつドキュメントとする。

完全一致クエリおよび正規表現探索のクエリの内容も 3.4 節でのクエリと同じクエリを実行した。

図 3.11 にバケットの大きさを 1×2^{10} から 512×2^{10} まで変化させた時のスループットを示す。スループットは rps で示しているが、これは request per second の略である。完全一致探索クエリは、単一のバケットを探索するため、バケットの大きさに比例して探索空間が大きくなる。しかし、バケットの大きさが GPU の並列度を活かすのに十分な大きさがないため、一定の閾値以下のバケットの大きさではスループットがほぼ一定となっている。バケットの大きさが 128×2^{10} もしくは 256×2^{10} からは、GPU の並列度を越えるバケットの大きさとなるため、スループットが低下している。

図 3.12 に正規表現探索クエリにおけるバケットの大きさとスループットの関係を示す。正規表現探索クエリでは、全てのバケットを検索するため、バケットの大きさに関わらず探索空間は一定である。しかし、バケットの大きさが小さい場合、多くの CUDA スレッドが小さいバケットに対して実行され、CUDA カーネルの立ち上げや CPU-GPU 間のデータ転送の回数が多くなり、

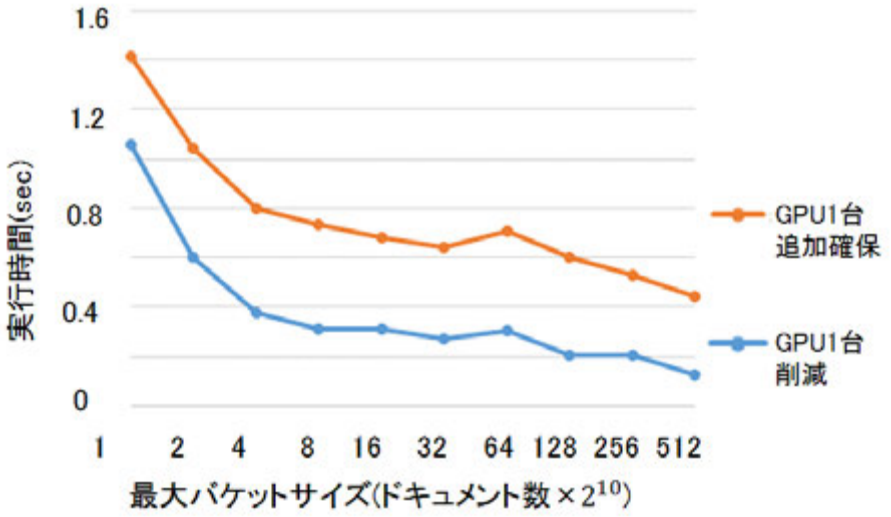


図 3.13: GPU デバイスをキャッシュ用に確保する数を動的に増減させた時の実行時間

オーバーヘッドが発生するため、スループットが小さくなる。そのため、図 3.12 ではバケットの大きさが 128×2^{10} まではバケットサイズが大きくなるに連れてスループットが向上している。特に GPU が 3 台の時において、バケットの大きさが 512×2^{10} の時には大幅にスループットが低下している。これは、バケットの大きさが大きく、バケットの数が少なくなるため、ワークロードが 3 台の GPU に均等に割り当てることができなくなるためである。

図 3.13 に GPU デバイスを確保する数を動的に増減させた時の実行時間を示す。GPU 数増加の時は、GPU 数 1 から 2、削減の時は 2 から 1 へと GPU 数を増減させている。増減双方の場合で、バケットの大きさが大きくなるに連れて実行時間が減少している。これは、バケットの大きさが大きくなり、バケット数が減ることによって、メモリの割り当てや CPU-GPU 間のデータ転送の回数が削減できるためである。増加の場合と削減の場合を比較すると、実行時間は削減の場合の方が短い。これは、GPU 数の増加の際には、新たな GPU のデバイスメモリの確保とデータ転送が行われるためである。一方で GPU 数の削減の際には、データの転送のみが行われる。

まとめると、バケットの大きさが大きすぎると、完全一致クエリの性能が低下し、バケットの大きさが小さすぎると、正規表現探索クエリの性能が低下する。GPU 数の増減を考えると、バケットの大きさは大きい方が性能が良い。これらの結果から、正規表現探索の性能を最大化しつつ、完全一致探索等のその他のクエリの性能をできるだけ高めることを考えると、128×2^{10} が今回評価したバケットの大きさの中では最も適していると考えられる。次節以降の評価では、バケットの大きさは全て 128×2^{10} とする。

3.7.2 GPU の直接接続と遠隔接続の性能比較

本章では、GPU デバイスはホストの計算機と 10GbE で接続することを想定し、分散キャッシュを提案した。当然、この分散キャッシュはホストと GPU 間を PCIe で直接接続した場合にも利用可能である。しかし、その場合には、GPU の数はマザーボード等の制約によって制限される。ここでは、GPU デバイスを直接接続した場合と、遠隔接続した場合での性能を比較する。比較に用いるクエリは、ここまでの評価と同じく、文字列の完全一致探索クエリと正規表現探索クエリであり、クエリの内容も同一のものとする。

図 3.14 に、GPU の数が 1 と 3 の時の GPU の直接接続と遠隔接続の文字列の完全一致探索クエリのスループットを示す。前述した通り、完全一致探索は 1 つのバケットのみを探索するため、バ

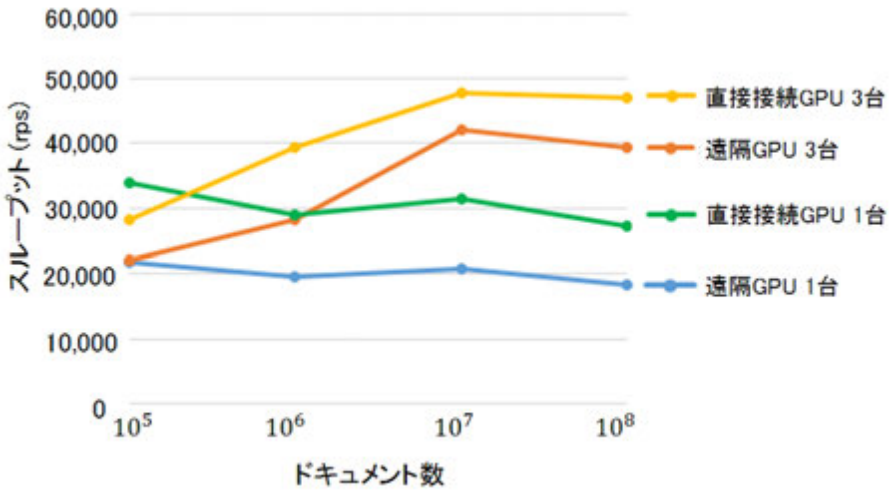


図 3.14: GPU の直接接続と遠隔接続の場合の文字列の完全一致探索クエリのスループット

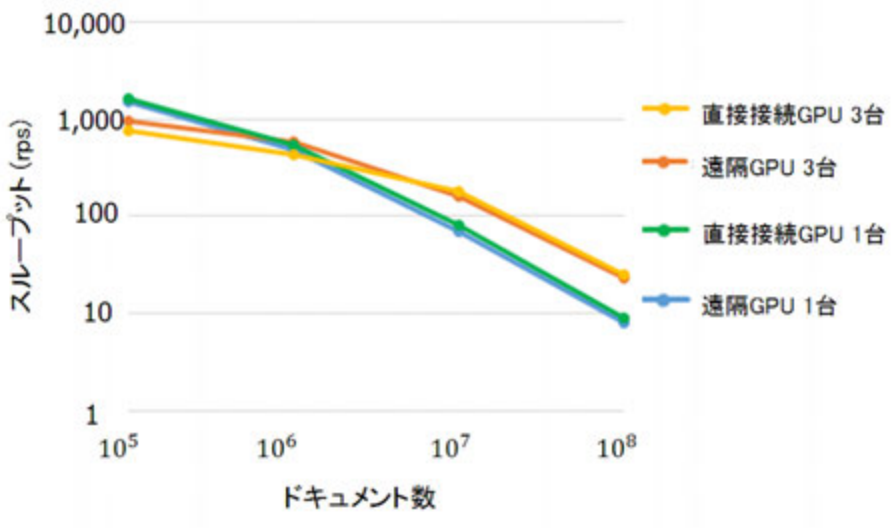


図 3.15: GPU の直接接続と遠隔接続の場合の文字列の正規表現探索クエリのスループット

ケットの大きさによって計算量が決定され、全体のドキュメント数が増加しても計算量は一定である。しかし、ドキュメント数が極めて小さい場合、バケットの数も同様に小さくなるため、複数の GPU にバケットを分散させることができなくなる。そのため、ドキュメント数が小さい場合の 3GPU のスループットは低下しており、この傾向は直接接続の場合も遠隔接続の場合も同じである。直接接続と遠隔接続を比較すると、常に直接接続の方が性能が良い。しかし、GPU の台数が増えた時の性能向上率は、遠隔接続の方が大きい。実際に、ドキュメント数 1 億件の時では、遠隔 GPU において、1GPU から 3GPU へ GPU 数を増やした場合にスループットは 2.14 倍となり、直接接続の場合の 1.73 倍に比べて上回っている。3GPU でドキュメント数 1 億件の時には、直接接続は遠隔接続の場合に比べて 1.20 倍のスループットである。この遠隔 GPU 接続の性能低下は、CPU-GPU 間の帯域が PCIe を用いた直接接続は片方向で 256Gbps であるのに対し、遠隔 GPU は 20Gbps のみであることを考慮すると、ハッシュの分散によって複数のバケット間で通信と計算を重複させて通信オーバーヘッドを隠し、性能低下率を抑えられていると言える。

図 3.15 に GPU の遠隔接続と直接接続の場合の正規表現探索クエリのスループットを示す。この図は、スループットとドキュメント数が対数の両対数グラフである。ドキュメント数が増加す

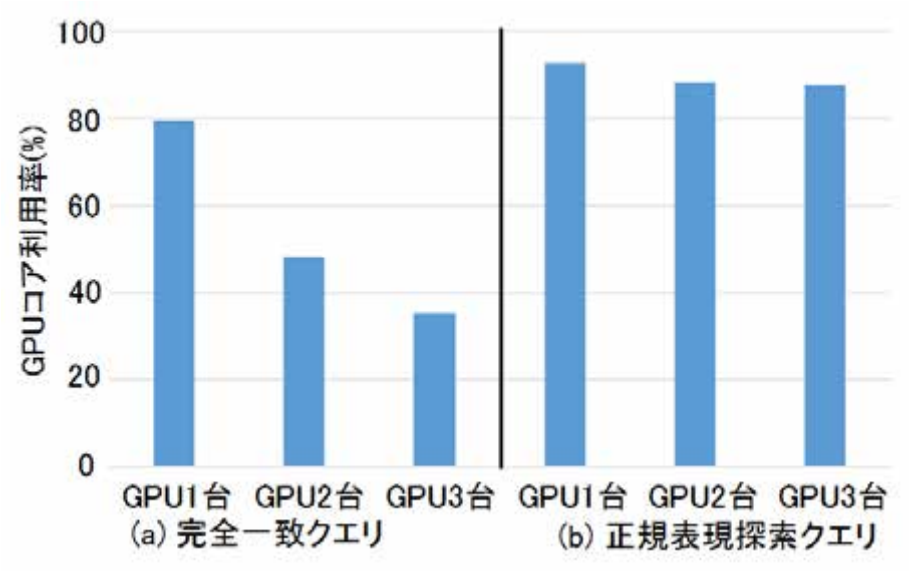


図 3.16: DDB キャッシュにおける完全一致クエリと正規表現探索クエリの実行時の GPU コア使用率

ると、全てのバケットを探索する正規表現探索クエリでは計算量が増加し、スループットは低下する。しかし、その性能低下率は、複数のバケットを並列に探索しているため、ドキュメントの増加率に比べると小さい。ドキュメント数が少ない時には、スループットは 1GPU から 3GPU に GPU が増えることによる性能の増加は無いが、ドキュメント数が多い時には、1GPU から 3GPU に GPU が増えることによる性能向上率は図 3.14 の場合と同様に遠隔接続の場合の方が大きい。実際に、ドキュメント数 1 億件の場合には、直接接続の場合のスループットは 3GPU の時が 1GPU の時の 2.71 倍であるのに対し、遠隔接続の場合は 2.89 倍である。この時、直接接続と遠隔接続のスループットは遠隔接続の場合の 1.08 倍と、遠隔接続の性能低下率が完全一致探索クエリよりも小さくなる。これは、正規表現探索の方がクエリの計算量が大きいため、より多くの転送オーバーヘッドを隠すことができるためである。

レイテンシに関しては、ドキュメント数 1 億件、GPU 数 3 台の時、完全一致探索クエリの実行時間は直接接続の場合 0.22msec、遠隔接続の場合 0.3msec であった。完全一致探索クエリは、単一のクエリに関しては 1 つの GPU で実行し、GPU 数を増やしても同時に実行するクエリが増加するだけなため、GPU 数によってレイテンシの変化は無い。また、レイテンシの方がスループットよりも関節接続と直接接続の差が大きく、ハッシュ機構による分散による通信と計算の重複でスループットが向上していることを示している。正規表現探索クエリのレイテンシは、同様の条件で、直接接続が 40.9msec、遠隔接続が 44.0msec であった。このレイテンシに関しては、正規表現探索クエリは複数の GPU で並列に実行されているため、GPU 数が増加するとレイテンシは低下する。

正規表現探索クエリと完全一致探索クエリを比較すると、正規表現探索クエリの方が GPU の台数を増やした時の性能向上率が大きい。これは、完全一致クエリは正規表現探索に比べてクエリ 1 つあたりの計算量が小さく、時間あたりに実行されるクエリの数が多いため、GPU へのクエリの転送やクエリの結果の返答等のクエリ毎に行わなければならない処理の実行回数も多く、これらの計算以外の処理がボトルネックとなるためである。それを確かめるために、それぞれのクエリの GPU コア利用率を評価した。図 3.16 にドキュメント 1 億件に対して、GPU 数が 1 台から 3 台それぞれで文字列完全一致と正規表現探索を実行した時の GPU コア利用率の平均を示す。文字列完全一致クエリでは、GPU 数が増加すると共に GPU コア利用率が大幅に低下しているた

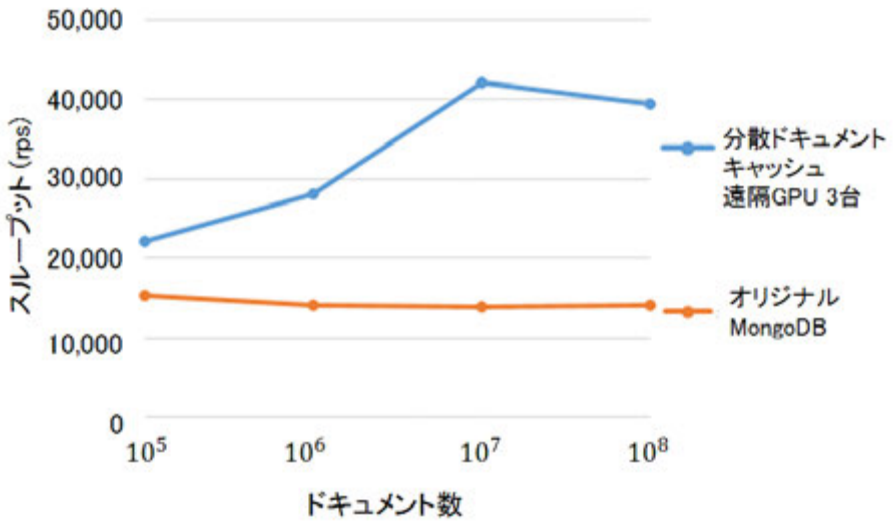


図 3.17: 分散キャッシュを用いた提案手法と MongoDB の文字列完全一致クエリのスループットの比較

め、クエリの計算以外の処理がボトルネックとなり、GPU を増やしてもスループットの向上はあまり見込めない。一方、正規表現探索クエリでは、GPU 数の増加に対する GPU コア利用率の低下は小さいため、計算がボトルネックとなり、GPU 数を増やすことでスループットの向上が見込める。

3.7.3 MongoDB との性能比較

ここでは、遠隔 GPU3 台を用いた提案手法とオリジナルの MongoDB のスループットを前節までの評価と同様の完全一致探索クエリと正規表現探索クエリで比較評価する。MongoDB においては、インデックスが利用できる完全一致探索クエリにおいては、B+tree インデックスを適用し、インデックスが利用できない正規表現探索クエリにおいては、インデックスは利用しない。また、3.4 節と同様に、書き込みクエリの評価も行い、キャッシュの分散によって書き込み性能の優位性が失われていない事を示す。MongoDB は 3.4 節と同様にメモリ上で動作させた。

図 3.17 に、分散キャッシュを用いた提案手法と MongoDB における完全一致探索クエリのスループットを示す。提案手法と MongoDB を比較すると、ドキュメント数に関わらず提案手法が MongoDB の性能を上回っており、ドキュメント数が 1 億件の時には、提案手法は MongoDB の 2.79 倍のスループットを達成した。

図 3.18 に分散キャッシュを用いた提案手法と MongoDB における正規表現探索クエリのスループットを示す。複数 GPU を用いた分散キャッシュにおいても、分割前のキャッシュの場合と同じく、正規表現探索クエリでは MongoDB がインデックスを用いていないため、大幅な性能向上を達成している。特に、ドキュメント数の増加に伴うスループットの低下が MongoDB よりも提案手法の方が小さいため、ドキュメント数が増加するにつれて性能向上率が高くなっている。その結果、ドキュメント数が 1 億件の時、分散キャッシュは MongoDB の 640.8 倍のスループットを達成した。CPU を用いた探索との比較においても、分割前のキャッシュと同様に GPU によって並列化したことに加えて、GPU 数の増加と多数のバケットへの分割による計算と転送の重複したことよって、1 億件の時 69.8 倍のスループットを達成している。

レイテンシに関しては、完全一致探索における分散キャッシュのレイテンシは 0.3msec であるのに対し、MongoDB のレイテンシは 0.071msec であり、MongoDB の方がレイテンシが短い。一方

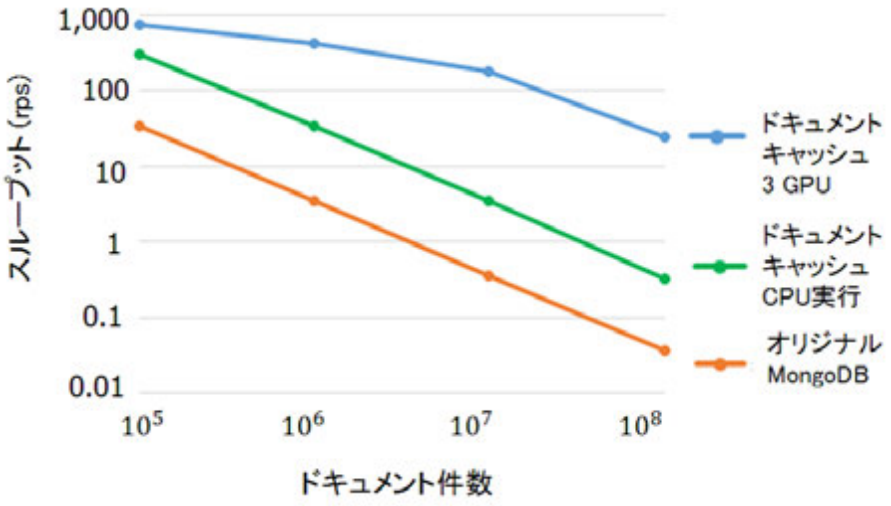


図 3.18: 分散キャッシュを用いた提案手法と MongoDB の正規表現探索クエリのスループットの比較

で、正規表現探索においては、分散キャッシュのレイテンシは 44.0msec であるのに対し、MongoDB のレイテンシは 28198.8msec であり、レイテンシに関しても大幅に分散キャッシュによって改善している。

これらの結果から、本提案手法を用いることで、完全一致探索の性能が低下することはなく、ボトルネックである正規表現探索を高速化できるといえる。しかし、本提案手法はあくまで正規表現探索に特化した手法であり、完全一致探索のみに特化した各手法を用いることで、完全一致探索に関しては更なる高速化が可能である。それらの手法と本提案手法の比較は 3.7.5 節に示す。

図 3.19 に分散キャッシュと B+tree インデックスを用いた MongoDB の書き込み性能の比較を示す。この評価では、3.4 節と同様に、分散キャッシュもしくはインデックスを作成するフィールド数を変化させて、既に 1,000 万件のドキュメントが保存されている状態への書き込みクエリのスループットを評価した。図 3.19 から、分散キャッシュの書き込みスループットは分割前の書き込みスループットと同様に、キャッシュするフィールド数の増加に伴うスループットの低下が抑えられている。この結果は、このハッシュ機構を用いた分散手法によっても書き込み性能のオーバーヘッドが小さく、キャッシュを複数の遠隔 GPU へ拡張できていることを示している。

3.7.4 ドキュメントサイズと正規表現探索クエリのスループットの関係

ここまでの評価では、8 文字のフィールドに対する探索を行っていたが、正規表現探索においては、より長い文字列に対して探索を行うことが想定される。本節では、そのような場合における本提案手法の実用性を示すために、探索対象のフィールドの文字数を変化させた場合および実データのデータセットを用いた場合における正規表現探索のスループットを示す。

図 3.20 は、_id と 1 つの文字列型のフィールドを持つドキュメントにおける、ドキュメント件数を 1 千万件とし、探索対象フィールドの文字列の長さを変化させた時の正規表現探索のスループットを示す両対数グラフである。ここでの GPU 処理は遠隔 GPU を 3 台用いている。正規表現探索の計算量は、文字数に比例するため、ドキュメントキャッシュを用いた場合と MongoDB を用いた場合のいずれも文字数が増加するにつれてスループットが低下している。MongoDB においては、ドキュメントキャッシュと異なり、フィールドのみを抽出した構造ではないため、ドキュメントからフィールドのバリューを特定するオーバーヘッドが発生する。そのオーバーヘッドは、

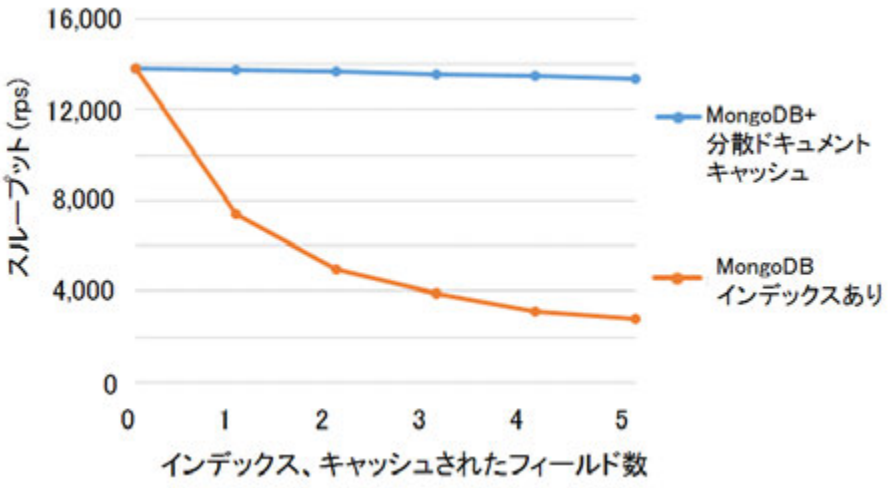


図 3.19: 分散キャッシュと B+tree インデックスを用いた MongoDB の書き込みのスループットの比較

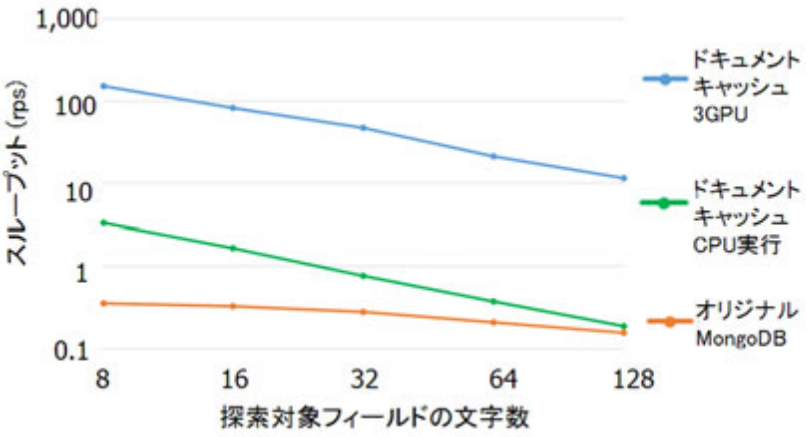


図 3.20: 探索対象の文字数と正規表現探索クエリのスループットの関係

フィールドの文字数に関わらず発生するため、文字数が増加したときのスループットの低下率は小さくなり、クエリに占める文字列探索の処理の割合が大きくなるにつれてスループットがドキュメントキャッシュに対する CPU 処理に近くなる。正規表現探索の CPU 処理と GPU 処理を比較すると、文字数によらず GPU 処理によって正規表現探索が高速化されており、文字数が 128 文字の時は、GPU 処理は CPU 処理の 61.7 倍のスループットとなった。また、MongoDB と GPU 処理を比較すると、文字数が大きい場合には MongoDB のオーバーヘッドの割合が低下するため高速化率は低下するが、128 文字の時においても GPU 処理が 75.0 倍のスループットと大幅な性能向上を達成している。

表 3.3: 実データを用いた正規表現探索クエリのスループット

データセット名	キャッシュ+3GPU(rps)	キャッシュ+CPU(rps)	MongoDB(rps)
commonswiki	18.54	0.36	0.20
wikidatawiki	43.96	0.80	0.28

同様に 1000 万件のドキュメントの実データのデータセットから抽出して正規表現探索を行った

場合のスループットを表 3.3 に示す。紙面の都合上、ドキュメントキャッシュはキャッシュと略称している。commonswiki の平均文字数は 71、wikidatawiki の平均文字数は 31 である。実データにおいてもランダムデータを用いた場合と同様の傾向を示し、平均文字数が小さい wikidatawiki の方が全ての手法でスループットが高く、MongoDB のスループットの変化率が小さい。また、この評価においても GPU を用いた手法によって CPU 処理および MongoDB を用いた場合に比べて大幅に高いスループットを達成しており、本手法が実データへも応用可能であることを示している。

3.7.5 完全一致探索における各手法の比較

本章で述べたドキュメントキャッシュの構造は、正規表現探索に特化した構造であり、完全一致探索には特化した構造ではないが、ハッシュ値をインデックス代わりに用いる分散によって探索範囲を限定することで、MongoDB のインデックスを用いた場合を上回る性能を達成し、完全一致探索クエリの性能を下げずに正規表現探索の高速化に成功している。しかし、MongoDB は探索に特化したデータ構造や探索手法ではないため、この結果は完全一致探索に特化した構造や探索手法と比較してドキュメントキャッシュの構造によって完全一致探索を高性能であることを示しているわけではない。ここでは、ドキュメント指向型の一連のクエリではなく、単純な完全一致探索の探索のみを行った時の性能を各手法で比較する。まず、完全一致探索の探索処理は、KVS 型におけるキーの検索と同等の処理であるため、KVS 型の GPU を用いた高速化手法 [1] によるハッシュテーブルを用いた手法が利用可能である。ここでは、この手法 (GPU-hash と呼称) を実装し、直接接続の GPU、遠隔接続の GPU でそれぞれ実行した場合のスループット、参考として CPU-GPU 間転送を含まない計算のみのスループットの 3 種類を示す。また、CPU 上での手法として、MongoDB のインデックスで用いられている B+tree で探索のみを実行した場合 (CPU-B+tree)、KVS で用いられるハッシュテーブルを用いた場合 (CPU-hash) を用いた場合の 2 種類を示す。GPU、CPU 共にハッシュの連想度は 1 とし、ハッシュミス時の処理は考慮しないものとする。それに加えて、本提案手法 (GPU-array) における完全一致探索を直接接続の GPU、遠隔接続の GPU で実行した場合と MongoDB の B+tree を用いた場合の処理を含め、合計 8 種類の手法を比較する。また、探索対象のキーは 32bit、1 億件とし、GPU 数 1 台とし、GPU 処理においては、バッチ処理で同時に複数の探索を行う。

表 3.4: 各手法における完全一致探索の探索処理スループット

探索手法およびデータ構造	スループット (rps)
GPU-hash(計算のみ)	1,326,430,588
GPU(直接接続)-hash	542,148,223
CPU-hash	438,827,453
CPU-B+tree	166,710,845
GPU(遠隔接続)-hash	154,849,075
GPU(直接接続)-array	676,225
GPU(遠隔接続)-array	648,482
MongoDB B+tree(参考)	26,816

表 3.4 に各手法における完全一致探索の探索処理のスループットを示す。それぞれ、1 億件の探索対象からキーを探索する処理を行っているが、MongoDB のみはクエリとしてドキュメントに対する検索を行っているため、公平な比較ではない。スループットに影響を与える各手法の計算量は、それぞれのデータ構造における比較回数によって決まる。まず、ハッシュを用いている場合

は、比較回数は連想度となるため、連想度 1 の場合は比較回数も 1 である。また、B+tree は、各頂点のキー数によって比較回数が異なるが、今回はキー数 3 としており、その場合の平均比較回数は $\log_3(10^8) \times 1.5$ となり、およそ 25.2 回となる。ドキュメントキャッシュの比較回数は、最大バケットサイズによって決まり、今回の最大バケットサイズは 128×2^{10} であり、131,072 である。各バケットに含まれるキーの数はバケット毎に異なるが、バケット生成時に最大バケットサイズに達したバケットの半分が移動されて新しいバケットが作られることから、64×2^{10} と 128×2^{10} の平均である 98,304 回が平均として見積もられる。この見積りは正確性は欠けるが、完全一致探索に特化した他の手法に比べて比較回数が多いことがわかる。その上で各手法のスループットを見ると、比較回数が少なく、GPU 処理を行っている GPU-hash が最もスループットが高い。しかし、GPU 処理では CPU-GPU 間転送のオーバーヘッドが発生するため、計算のみの場合は CPU-hash の 3.0 倍のスループットであるが、転送を含めると 1.2 倍のスループットとなっている。この GPU 処理のスループットは、文献 [1] におけるスループットと比較して大きくなっているが、これは文献 [1] においては、ネットワーク処理等を含めたクエリ全体のスループットを示しているいること、キー長が 16Byte から 128Byte と本評価に比べて長いこと、連想度が 16 と本評価に比べて大きいことが主な要因である。また、遠隔接続の場合は CPU-GPU 間転送のオーバーヘッドがより大きくなるため、CPU 処理の方がスループットが高くなる。B+tree を用いた手法は、ハッシュテーブルより比較回数が多いため、ハッシュテーブルに次ぐスループットとなっている。これらの手法と比較して、ドキュメントキャッシュを用いた手法は、バッチ処理を行い、探索のみとなっているため、3.7.3 節の評価と比較するとスループットは大きいが、他の手法と比較すると大幅にスループットが小さい。これは、上述した比較回数の多さが要因であり、最大バケットサイズを小さくすることでスループットを高めることができるが、その場合正規表現探索の性能が低下してしまうため、本提案手法の趣旨に反する。

以上の結果から、本提案手法よりも、完全一致探索に特化した各手法の方が完全一致探索の性能は高く、これは本論文の基本的な考え方である特定用途特化の考え方にも合致している。よって、特定用途特化の考え方から、正規表現探索は本提案手法、完全一致探索はハッシュテーブルを用いた手法とポリグロット永続化のように併用して用いた場合が、システム全体の性能が最も高くなる。また、ドキュメント指向型ストアにおける範囲検索クエリも考慮した場合には、ハッシュテーブルではなく、B+tree を用いるべきである。さらに、B+tree は本来ディスク上に構築し、ページアクセス回数を削減することを目的としたデータ構造であるため、メモリ上でキャッシュを作成する場合には二分探索木や Masstree [66] のようなデータ構造を採用することで、より高性能化することも可能である。

しかし、各フィールドにおいて、それぞれに特化した複数のキャッシュを併用する場合、メモリ容量などの制約に抵触する可能性がある。そのような場合は、ボトルネックとなる正規表現探索を高速化できる提案手法のドキュメントキャッシュを主として用い、正規表現探索の対象とならない数値型等のフィールドのみ完全一致探索に特化したキャッシュを用いることでシステム全体を高速化できる。

3.7.6 分割前のキャッシュに対する性能評価との比較

本節の評価結果を 3.4 節の評価と比較すると、最も大きく評価結果が異なるのは完全一致探索クエリの評価である。分割前のキャッシュでは、GPU による探索はキャッシュ全体に対して行う必要があり、ドキュメント件数が増加するにしたがい実行時間が伸び、インデックスを用いた MongoDB に比べて大幅に実行時間が劣っていた。しかし、分散キャッシュにおいては、バケットを用いた分割を行い、探索対象を一つのバケットに絞ることで、完全一致探索クエリにおいてもインデックスを用いた MongoDB を上回るスループットを達成した。正規表現探索クエリにおいては、GPU

数の増加による並列化と分散ハッシュを用いた計算と転送の重複によって、性能向上率が分割前のキャッシュよりも高い結果となった。また、書き込みクエリにおいてはMongoDBの書き込み性能の低下を抑えつつ、拡張に成功した。これらの結果から、分散ハッシュにおける拡張は、分割前のキャッシュの問題点を解決しつつ、複数の遠隔GPUへの拡張を可能にしたと言える。

第 4 章

GPUを用いたグラフ型ストアの高速化手法

2 章て述べたように、各種構造型ストレージのクエリにおいて、最も計算量の大きいクエリは、グラフ探索クエリである。そのため、複数の構造型ストレージを組み合わせた際にボトルネックとなる可能性が高く、高速化の必要性が最も高い。本章においても、本論文の基本的な考え方である特定用途特化に則り、GPU 処理に適しており、かつグラフ探索に特化したデータ構造のキャッシュを提案する。

グラフキャッシュに対して GPU でグラフ探索処理を行うことで、グラフ探索クエリを高速化できる。一方で、グラフ型ストアにおいては、グラフ探索に必要な情報は、グラフの構造に相当する辺の始点、終点および重みのみである。そのため、それらの情報のみを抽出してキャッシュを作成した場合、各頂点や辺の属性の情報量やグラフ型ストアの実装によってはグラフ型ストア全体のメモリ使用量に比べてキャッシュのメモリ使用量が大幅に小さくなりうる。そのような場合においては、1 台のホスト上のグラフ型ストアだけでなく、複数台のホストで動作するグラフ型ストアから情報を抽出し、単一の計算機におけるグラフキャッシュで処理できる。また、属性等の情報の割合が少なく、ドキュメントキャッシュと同様に 1 台のホストに対し複数の遠隔 GPU への拡張が必要になる場合も想定できる。よって、本章では、上記の二つの場合に応じて、以下の二つのシステムを想定し、それに対するグラフキャッシュを提案する。

- 複数台の計算機で動作するグラフ型ストアから、グラフ構造のみを抽出し、単一計算機上で動作するグラフキャッシュにキャッシュする。その上で、そのキャッシュを単一の GPU へ転送し、グラフ探索処理を行う。
- 単一計算機上で動作するグラフ型ストアの同一計算機上にグラフキャッシュを設け、そこにグラフ構造をキャッシュする。その際、グラフキャッシュは複数の遠隔 GPU へ分散して保持し、GPU は保持しているキャッシュに対してグラフ探索処理を行う。

4.1 複数台の計算機からのキャッシュシステムの全体像

本節で述べるグラフキャッシュは必要最低限の情報で構築することによって、オリジナルのグラフ型ストアに比べてメモリ使用量が大幅に削減できた場合を想定したキャッシュシステムである。メモリ使用量が大幅に削減できた場合、単一計算機上でより大規模なグラフを扱うことができ、複数台のグラフ型ストアに跨るグラフを扱うことで、拡張性が改善する。

図 4.1 に提案するグラフ型ストアシステムの全体像とクエリの流れを示す。本システムは、複数のグラフ型ストアサーバとグラフキャッシュサーバの 2 つで構成される。図 4.1 における複数台のグラフ型ストアは 2.3.5 節で述べた分割グラフ型ストアではなく、それぞれ独立している。2.3.5

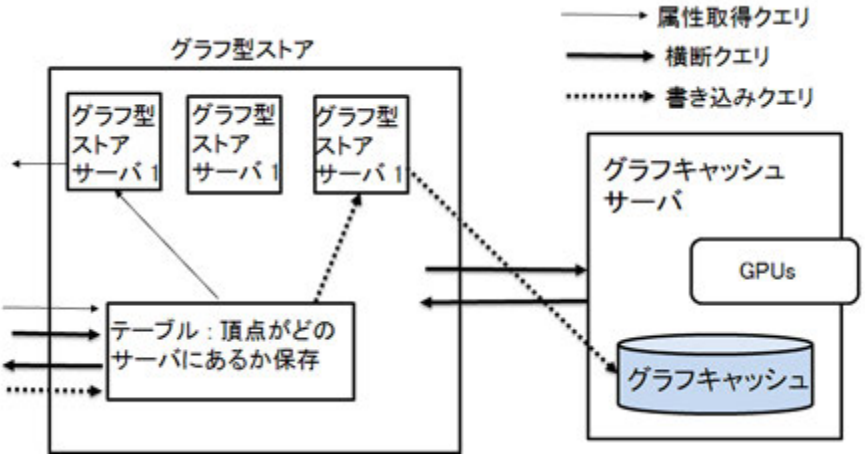


図 4.1: 複数のグラフ型ストアのキャッシュシステムの全体像とクエリの流れ

節で述べたように、グラフ型ストアを分割する際、問題となるのは各計算機の境界部分の横断処理である。非横断クエリと書き込みクエリのみであれば、分割されたグラフを複数台の独立したグラフ型ストアサーバに保存し、各頂点、辺がどの計算機に保存されているかを記録するテーブルを追加するだけで、独立した複数台のグラフ型ストアでも実現できる。ただし、境界部分の辺に関しては、境界外にある終点となる頂点を追加する。この場合同じ頂点が複数の計算機に存在するため、メモリ利用効率は低下するが、グラフ型ストアに変更を加える必要はない。また、複数台のグラフ型ストアからグラフキャッシュを作成することで、複数台のグラフ型ストアに保存されていたグラフに対する横断処理ができるようになる。よって、複数台のグラフ型ストアに対して、横断処理はグラフキャッシュを用い、他の処理はグラフ型ストアを用いるように分担することで、グラフ型ストアの機能を維持しつつ、複数台の計算機にグラフ型ストアを拡張できる。

本システムにおけるグラフ型ストアの各クエリの流れは以下の通りである。

- 属性取得クエリは、テーブルでどのグラフ型ストアサーバに探索対象があるかを確認し、そのグラフ型ストア内で処理し、ユーザに結果を返す。
- 横断クエリ (グラフ探索クエリも含む) は、グラフ型ストアからグラフキャッシュサーバにクエリを渡し、グラフキャッシュサーバにおいてグラフキャッシュを GPU を用いて探索し、結果をグラフ型ストアサーバに返し、グラフ型ストアサーバからユーザーに結果を返す。
- 書き込みクエリは、テーブルでどのグラフ型ストアサーバに探索対象があるかを確認し、グラフ型ストアの情報を更新した後、グラフキャッシュを更新する。

4.2 グラフキャッシュのデータ構造

Neo4j は Java 言語で実装されており、頂点や辺に関する情報を集約し、それらを様々な用途に使用できるよう頂点や辺はクラスとして定義されている。各頂点や辺はそのクラスのインスタンスを生成することで実装される。Neo4j のグラフ探索アルゴリズムは、これらのインスタンスをそのまま用いて実装されているが、このようなデータ構造をそのまま並列ライブラリを用いて並列化したり、GPU 上に実装することは困難である。また、このようなデータ構造は、グラフ型ストアとして運用には適しているが、グラフ探索のみを考慮すると、冗長な情報が多い。そのため、グラフ探索に必要最低限の情報のみを抽出し、キャッシュを作成する。

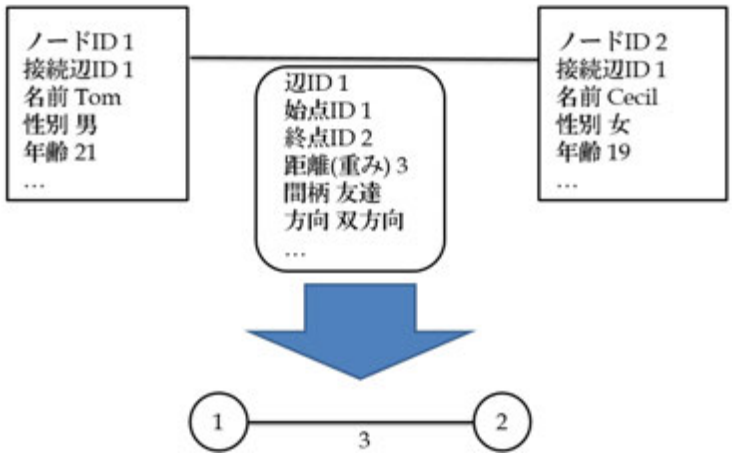


図 4.2: Neo4j のデータ抽出

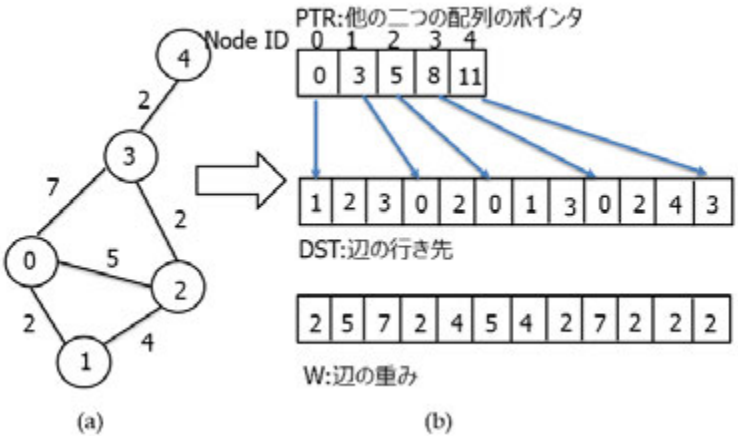


図 4.3: グラフキャッシュの作成

図 4.2 に、Neo4j のデータ構造で表されるグラフと、それを基に抽出したグラフ探索に必要最小限の情報からなるグラフを示す。図の上側が Neo4j のデータ構造を示している。この例は、人間関係を表すグラフの例で、2 人の人の情報を表す頂点とその関係を表す辺で構成されている。頂点、辺にはそれぞれ様々な情報が格納されているが、グラフ探索に用いる情報は、頂点の頂点 ID、辺の始点 ID、終点 ID、重みのみである。それらを抽出してグラフ化したものが、図の下側のグラフである。頂点の中の数字が頂点 ID、辺の横の数字が辺の重みを表している。

このように抽出した情報を、GPU の処理に適したデータ構造に変換を行う。GPU に適したデータ構造として、CSR(Compressed Sparse Row) を用いる。CSR に似た構造でグラフの表現によく用いられる構造に頂点リストがあげられる。しかし、頂点リストの実装にはポインタを多数用いており、ホストメモリ上のポインタには GPU が直接アクセスできない。よって、GPU にポインタを確保しなおす必要があるため、オーバーヘッドが発生してしまい、CSR の方がより GPU 実装に適している。

図 4.3 は図 4.2 のように情報の抽出された 5 頂点のグラフを CSR で変換して作成したグラフキャッシュの例である。図 4.3(a) において、頂点内の数字が頂点 ID、辺の横の数字が辺の重みを表す。辺の方向は全て双方向とする。図 4.3(b) は、図 4.3(a) のグラフに対応する隣接リストを三つの配列を用いて表現している。配列は、他の二つの配列のポインタの役割をする配列 PTR、各辺の行き先を表す配列 DST、各辺の重みを表す配列 W の三つである。各配列の要素の内容を以

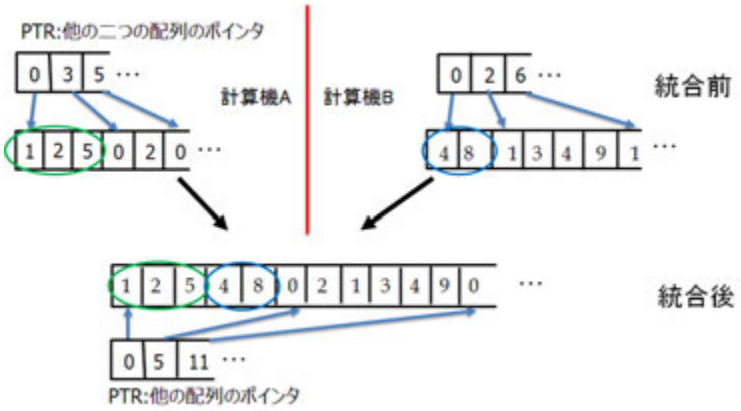


図 4.4: グラフキャッシュの併合

下に示す。

- PTR の n 番目の要素 $PTR[n]$ は n 番目の頂点までの累積次数を表す。すなわち、D_i を頂点 i の次数としたとき、$PTR[n] = \Sigma_{i=0}^{n} D_i$ である。
- p の範囲が $PTR[n] \leq p < PTR[n+1]$ のとき、$DST[p]$ は頂点 n が始点の辺の終点を表す。
- $W[p]$ は $DST[p]$ に対応する辺の重みを表す。

例えば、図 4.3(b) において、頂点 0 は頂点 1、頂点 2、頂点 3 への辺と隣接しており、それぞれ重みは 2、5、7 である。

このデータ構造を用いて GPU でグラフ探索を行ったのち、探索結果を Neo4j に戻し、Neo4j で通常の後処理を行うことで、Neo4j で直接実行した場合と同じ出力形式の結果が得られる。

4.3 グラフキャッシュの併合

4.2 節で述べたグラフキャッシュの作成は、Neo4j からデータを抽出してグラフキャッシュを作成する操作である。独立した複数台のグラフ型ストアにグラフが保存されているため、グラフキャッシュ作成の操作は各計算機で別々に行う操作となる。具体的には、各計算機で作成したグラフキャッシュをグラフキャッシュ格納用の計算機に送信し、各グラフのグラフキャッシュを併合して 1 つのグラフキャッシュにする。

図 4.4 にグラフキャッシュの併合の様子を示す。図の上半分が併合前のグラフキャッシュ、下半分が併合後のグラフキャッシュを示す。グラフの抽出の際に、各グラフのグラフキャッシュは頂点 ID に基づいてソートされる。すなわち、各グラフのグラフキャッシュの配列 DST と配列 W は頂点 ID が小さい順に各頂点の隣接辺に関する情報が格納されており、配列 PTR で管理されている。

各グラフのグラフキャッシュがソートされていることを利用して、グラフキャッシュの併合を行う。併合の擬似コードを Algorithm 9 に示す。この擬似コードでは、1 行目から 3 行目までに示す併合後の各配列に 4 行目から 6 行目までに示す各配列を併合する。10 行目と 11 行目が配列 DST と配列 W の併合を表しており各配列の併合は、各頂点の隣接辺毎に併合後の配列にコピーすることで実現できる。12 行目は、各配列の起点となるポインタの役割をする k を求めるため、各グラフの配列 PTR から次数を求めてそれを各グラフ型ストア毎に足している。14 行目で k を配列 PTR に代入することで、併合後の PTR の値となる全てのグラフ型ストアの累積次数の総和が配列 PTR に代入できる。

Algorithm 9 グラフキャッシュの併合

1: $PTR \leftarrow$ 併合後のグラフキャッシュの配列 PTR
2: $DST \leftarrow$ 併合後のグラフキャッシュの配列 DST
3: $W \leftarrow$ 併合後のグラフキャッシュの配列 W
4: $PTR_j \leftarrow j$ 番目のグラフ型ストアから抽出したグラフキャッシュの配列 PTR
5: $DST_j \leftarrow j$ 番目のグラフ型ストアから抽出したグラフキャッシュの配列 DST
6: $W_j \leftarrow j$ 番目のグラフ型ストアから抽出したグラフキャッシュの配列 W
7: $k \leftarrow 0$ に初期化
8: **for** $i = 0$ *to* (頂点数 $- 1$) **do**
9: **for** $j = 1$ *to* (グラフ型ストアの数) **do**
10: $DST_j[PTR_j[i]]$ から $DST_j[PTR_j[i+1]-1]$ までを $DST[k]$ を起点として DST にコピー
11: $W_j[PTR_j[i]]$ から $W_j[PTR_j[i+1]-1]$ までを $W[k]$ を起点として W にコピー
12: $k \leftarrow k + PTR_j[i+1] - PTR_j[i]$
13: **end for**
14: $PTR[i+1] \leftarrow k$
15: **end for**

4.4 グラフキャッシュの更新

作成したグラフキャッシュはグラフ型ストアの更新を反映するように動的に更新を行う必要がある。動的な更新を実現するために、オリジナルのグラフ型ストア、グラフキャッシュの双方に修正を加える。

4.4.1 オリジナルのグラフ型ストアの修正

オリジナルのグラフ型ストアである Neo4j で、グラフの更新が発生した時、グラフキャッシュも動的に更新を行うため、グラフキャッシュの更新情報をグラフキャッシュに送信する必要がある。グラフキャッシュでは、各辺の始点、終点、重みのみを保持しているため、辺の更新情報がグラフキャッシュの更新に必要な情報である。Neo4j の持つ情報は辺の更新情報に比べて多いため、Neo4j から直接辺の更新情報を読み込むのはオーバーヘッドが大きい。そこで、辺の更新情報のみを保存する更新用リストを新たに作成する。更新用リストは、更新種類 (追加もしくは削除)、辺の始点、辺の終点、辺の重みの 4 つの要素を持つ構造体の線形リストとし、一つの構造体が一つの辺の更新情報を表す。グラフの辺に変更があった場合、Neo4j の更新と同時に、このリストに更新情報を追加する。グラフキャッシュに送信する際は、更新用リストの中身を配列に変換し、それを TCP/IP などの通信プロトコルを用いて送信する。グラフキャッシュが受信した更新情報は既に更新済みとみなし、更新用リストから削除する。後述するコンシステンシを保証する実装の場合は、グラフキャッシュが探索クエリを実行する前に全ての Neo4j に対して更新用リストを確認する。更新に失敗している場合は更新用リストが残っているため、探索は行われず、Neo4j が停止していた場合も応答がないため更新に失敗したと判定され、探索は行われない。よって、更新の失敗等によってコンシステンシが損なわれることない。

4.4.2 更新を考慮したグラフキャッシュのデータ構造

グラフキャッシュのデータ構造は配列で実装されるため、要素を挿入する場合、挿入箇所の後ろの要素を全て 1 つ後ろにずらさなければならない。削除の場合は逆に削除箇所の後ろの要素を

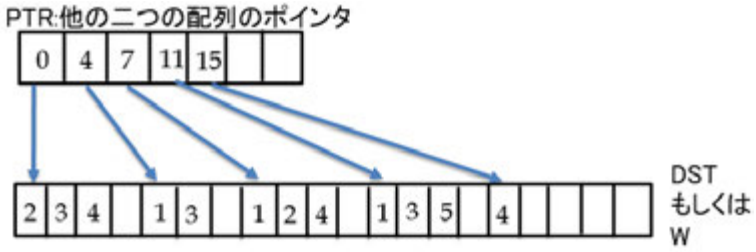


図 4.5: 更新を考慮したグラフキャッシュ

全て 1 つ前にずらさなければならない。挿入、削除箇所の後ろの要素数は、要素数を N とするとき平均 $N/2$ であるため、この操作には平均 $N/2$ 回の配列の書き換えが発生する。よって、挿入、削除は、$O(N)$ の計算量となりコストが大きい。そこで、グラフキャッシュでは、CSR 構造の各配列に一定の割合でダミー要素を追加する。探索の際にはダミー要素をスキップすることで探索に影響を与えずに将来の要素追加に備えて空き領域を確保できる。ダミー要素を用いると、データの挿入ではダミー要素を通常の要素に、削除では通常の要素をダミー要素に変えるだけで更新ができ、計算量が大幅に削減できる。配列 DST や配列 W には通常負数は格納されないため、ダミー要素には -1 などの負数を入れておく。頂点や辺の追加があった場合はダミー要素に値を書き込み、逆に頂点や辺の削除では値を負数に書き換えることでダミー要素（空き領域）に戻すことができる。

図 4.5 に更新を考慮したグラフキャッシュを示す。配列の空白部分は空き領域を示している。図に示すように、どの頂点に辺が追加されても対応できるよう、各頂点の隣接辺の領域に空き領域を確保している。頂点や辺が追加された場合、この空き領域を書き換えることで対応し、空き領域が足りなくなった際にはグラフキャッシュの再構成を行う。頂点や辺の追加、削除およびグラフキャッシュの再構成については、次節以降で詳しく述べる。空き領域をどの程度確保するかはユーザーが決め、空き領域を多く確保するとメモリ使用量が増える代わりに再構成の回数を減らすことができる。グラフキャッシュのメモリ使用量は、4.6.3 節の評価で示す通り元のグラフ型ストアに比べて非常に小さく、空き領域を確保することによりメモリ使用量が増加してもその影響は限定的である。

更新処理にかかる時間は Neo4j の更新とグラフキャッシュの更新間の遅延となる。よって、Neo4j とグラフキャッシュ間のコンシステンシを保つ場合、この時間がオーバーヘッドとなり、性能が低下する。本システムでは、コンシステンシを保証するか性能を優先しコンシステンシを保証しないかを選択することができる。コンシステンシを保証する場合、グラフキャッシュの探索の際に、Neo4j を実行する各計算機に更新用リストが空であるかどうか問い合わせる。もし空であれば、Neo4j のグラフとグラフキャッシュのグラフは一致しているのでそのまま探索を実行し、空でなければ更新用リストをグラフキャッシュに送信し、グラフキャッシュの更新を行う。コンシステンシを保証しない場合、探索の際に Neo4j に問い合わせることなく探索を行う。この場合は更新に伴う性能の低下は発生しない。コンシステンシを保証することによる性能低下は 4.6.6 節で評価する。コンシステンシと性能の優先度は利用するユーザー毎に異なるため、どちらを選択するかの判断はユーザが行う。

4.4.3 グラフキャッシュの頂点及び辺の削除

図 4.5 に示したグラフキャッシュにおいて、頂点、辺の削除を行った時について説明する。図 4.6 は、図 4.5 から頂点 2 とそれに隣接する辺を削除した例である。辺の削除は、元々値が入っていた要素をダミー要素にするだけで行える。各頂点の最後だけでなく、始めまたは途中が空欄と

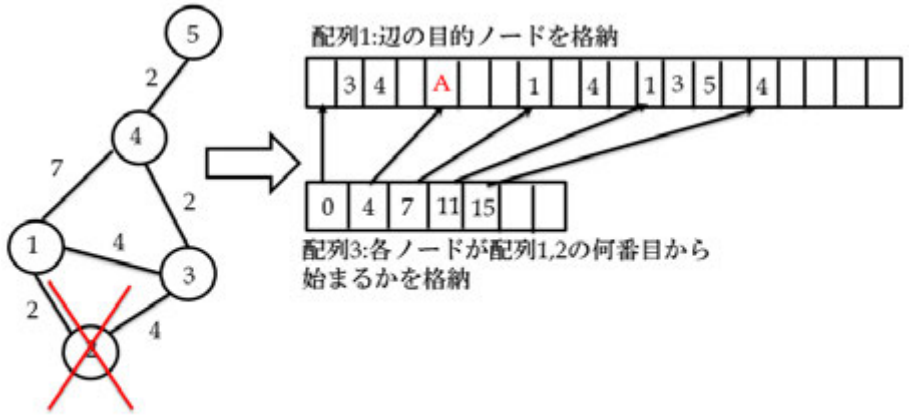


図 4.6: グラフに頂点および辺を削除したときのグラフキャッシュの更新

なってもその部分を無視して次の要素を探索するので、問題なく探索ができる。また、辺を追加する際は、このダミー要素となった部分に辺を追加できる。

グラフの探索処理において、ある頂点に隣接する辺が全て削除されれば、その頂点は探索されることはなくなり、削除されたとみなせる。そのため、頂点の削除という操作を新たに追加する必要はなく、頂点に隣接する辺を全て削除することで、頂点の削除を行える。配列 PTR は各頂点から出る辺が配列 DST および W のどこまでかを知るために次の要素を参照することがあるため、図 4.6 にあるように、PTR の値は変更しない。この時、頂点が削除されていない場合と区別するために、配列 DST の ID2 の頂点から出る辺の値を頂点が削除されていることを示す値 (図では A) に変更する。

4.4.4 グラフキャッシュの頂点及び辺の追加

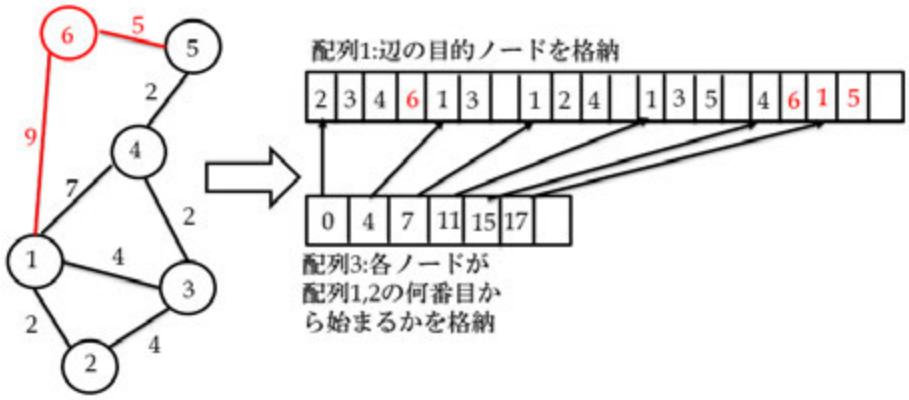


図 4.7: グラフに頂点および辺を追加したときのグラフキャッシュの更新

図 4.7 は図 4.5 に頂点と辺を追加した例を表している。図 4.5 に新たに頂点 6 を追加し、頂点 1 と頂点 5 から頂点 6 への辺を追加している。

グラフキャッシュへの頂点の追加は、現状のグラフキャッシュに存在しない頂点に対して辺を作ることで、頂点の追加とみなす。この例では、頂点 1 から頂点 6 への辺を作ることで頂点 6 の追加が行われる。頂点 6 を追加するためには、配列 PTR に頂点 6 配列 DST の位置を示す値を追加しなければならない。配列 DST の最後の空き領域を頂点 6 の領域とし、その場所を配列 PTR の値に設定する。図 4.7 では、図 4.5 に示した配列 1 の最後の要素は 15 番目であるため、そこから辺の追加に備えて 1 要素分の領域をあけて、17 番目からを頂点 6 の領域としている。この例では、辺の追加に備えての空き領域は 1 要素分だけしか用意していないが、実際には更新頻度等を考慮して予め決めた数だけ空き領域をあけて、次の頂点の領域とする。

図 4.7 では、さらにその後、頂点 5 と頂点 6 への辺を追加している。頂点の追加の際に頂点 5 の領域をあけたことにより、グラフキャッシュの再構成を行うことなく、辺の追加を行うことができる。

また、頂点及び、辺の変更については、削除を行った後追加をすることで対応する。

4.4.5　グラフキャッシュの再構築

図 4.7 にさらに頂点 1 や頂点 5 に隣接する辺を追加したり、新しい頂点を追加したりしようとすると、グラフキャッシュの空き領域が足りず、追加不可能である。そのため、グラフキャッシュを再構築する必要がある。

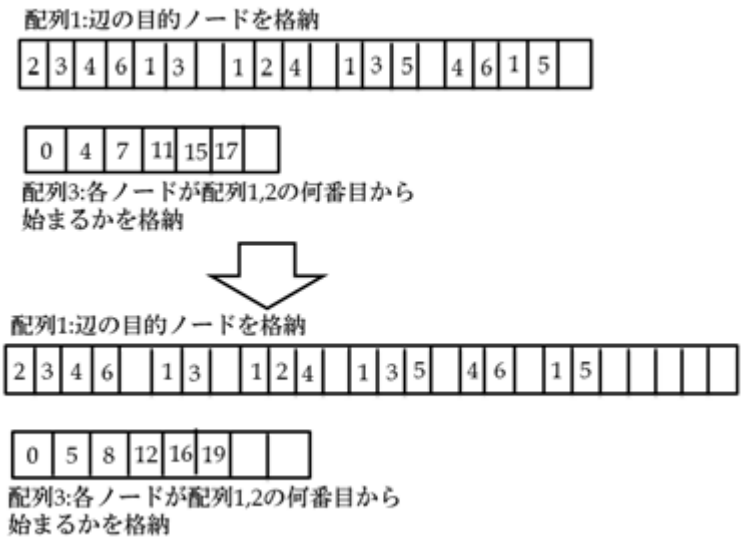


図 4.8: グラフキャッシュの再構築

図 4.8 は配列 DST と配列 PTR をで再構築する例である。配列の大きさは、どの程度余分な領域を確保するかによって決まる。これは、図 4.5 で始めにグラフキャッシュを生成したときと同じである。この例では、配列 DST には各頂点毎に 1 要素、最後に 4 要素、配列 PTR には 2 要素分の領域が余分に確保されている。

この再構築は、配列を全て置き換える必要があるため、オーバーヘッドが発生する。しかし、この配列の再生成の場合、始めの配列生成の場合と異なり、元の Neo4j のクラスへのアクセスは行わず、古い配列と新しい配列の値の操作のみであるため、始めの配列生成に比べてオーバーヘッドは小さい。また、この例では空き領域の確保量が少ないため、同じ頂点に複数辺を追加するだけで配列の再生成が必要となるが、実際はより大きな空き領域を設定するため、再構成の頻度は低い。

Algorithm 10 転送単位の決定

```
T ← 転送単位の大きさ制約、この大きさよりも転送単位が小さくなるようにする
n ← グラフキャッシュの頂点数
s ← 転送単位の始まりの頂点 ID、初期値 0
for i = 1 to n do
    if PTR[i] − PTR[s] > T then
        配列 DST、W の PTR[s] から PTR[i − 1] − 1 までを転送単位とする
        s ← i − 1//s の更新
    end if
end for
PTR[s] から PTR[n] までを転送単位とする
```

4.5 グラフキャッシュに対する GPU を用いたグラフ探索

4.5.1 GPU への転送単位

グラフキャッシュは、グラフ型ストアの情報を抽出し、一台の計算機に保存されるが、GPU のメモリ容量は計算機のメモリ容量やディスクの容量に比べて小さい。そのため、グラフキャッシュが GPU のメモリ容量よりも大きい場合、グラフキャッシュを GPU のメモリ容量に合わせて複数回に分けて転送し、GPU でその都度処理を行うか、複数 GPU への拡張が必要である。複数 GPU への拡張した場合においても、ここで述べる分割手法と同様に分割し、各 GPU に割り当てることで複数 GPU へ拡張できるが、ここでは、単一の GPU で複数回に分けて転送、処理を行う場合について述べる。

グラフを探索する際、ある頂点に隣接する辺を全て参照する操作が多用されるため、ある頂点に隣接する辺は同じ転送単位に含まれるように転送することが望ましい。4.2 節で述べた CSR を用いたデータ構造では、各頂点に隣接する辺がまとまって格納されており、配列 PTR によって管理されているので、配列 PTR を基に他の二つの配列を転送することで、「ある頂点に隣接する辺を全て含む転送単位」の決定は容易である。また、配列 PTR の要素の値は、グラフの累積次数を表しているため、配列 PTR の値を調べることで各転送単位の大きさを求めることができる。そのため、GPU のメモリ容量に収まる大きさに設定してグラフキャッシュを転送できる。

この転送単位の決定方法の擬似コードを Algorithm 10 に示す。転送単位の大きさ制約 T は GPU のメモリ容量に合わせて予め設定しておき、それを基に転送単位を決定する。

また、この方法ならば、転送用の配列を別に作成して保存する必要はなく、GPU に転送する際に、転送単位の範囲毎に転送すればよい。そのため、分割にかかる計算時間は分割箇所を決める計算時間のみであり、グラフ探索や GPU へのデータ転送の計算時間に比べて非常に短い。

この手法によって転送単位を決定した場合の転送単位とグラフの対応例を図 4.9 に示す。実際には配列 DST、W の二つの配列を転送するが、ここでは図が煩雑になることを避けるために配列 DST のみ示している。図の例は Algorithm 10 に T=5 を与えた場合である。Algorithm 10 の $i = 3$ の時、$PTR[3] - PTR[0] = 8$ であり、T よりも大きくなるため、$PTR[0]$ から $PTR[2] - 1$ までを転送 1 の転送単位とする。同様にして、Algorithm 10 にしたがって転送 2、転送 3 の転送単位も決定できる。図 4.9 のグラフは、グラフの各辺がどの転送単位で転送されるかを表している。図から、転送単位 1 から 3 までで全ての辺を網羅していることがわかる。仮に始点を 0、終点を 4 として最短経路を求める場合、頂点 0,1 から出る辺は転送単位 1 が GPU に転送されたとき、頂点 2 から出る辺は転送単位 2 が GPU に転送されたとき、頂点 3,4 から出る辺は転送単位 3 が GPU に転送されたときに次節で述べるように探索を行うことで最短経路を求められる。

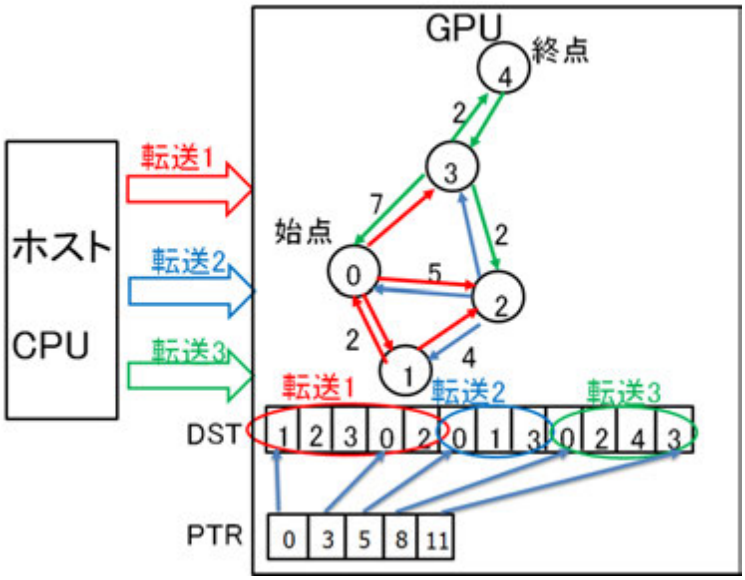


図 4.9: 転送単位とグラフとの対応例

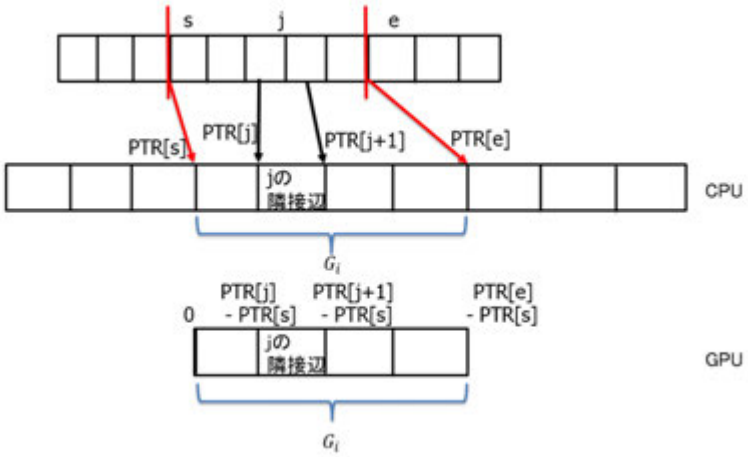


図 4.10: 各転送単位のグラフの索引番号

また、配列 PTR に関しては、配列 DST、W に比べて小さいため、分割して転送する必要はない。

4.5.2 グラフキャッシュに対する GPU を用いた単一始点最短経路問題

本節では、グラフ探索クエリの例として、単一始点最短経路問題のグラフキャッシュへの適用方法について述べる。2.4.2 節で述べた Ortega-Arranz らの実装方法を、グラフキャッシュに対して適用する。本節で述べるアルゴリズムはグラフキャッシュが GPU のメモリサイズを上回った場合に対応するように Ortega-Arranz らの実装方法を修正したものである。GPU 向けの最適化実装については本論ではないため、実装の最適化等は行っていない。また、その他のアルゴリズムにおいても、同様の拡張を行うことで実装可能である。実装環境は NVIDIA 社が提供する開発環境である CUDA を用いた。Algorithm 11 にこの実装の擬似コードを示す。

始めに、1 行目から 4 行目までで表される初期化を行う。次に、前節で述べたように配列 PTR を調べて転送単位を決める。この操作は計算時間が GPU への転送時間よりも短いため、CPU で

行う。次に、2.4.2 節で述べた並列に探索する対象を決定する式 2.1 で表される Δ を求める前計算として、カーネル delta_prepare を各転送単位で実行する。カーネル delta_prepare の擬似コードを Algorithm 12 に示す。カーネル delta_prepare では、各頂点の隣接辺に対して重みの最小値を計算する。この処理では、多数の頂点において隣接辺の重みの最小値である $d[j]$ を並列に計算している。各スレッドに 1 つずつ頂点が割り当てられ、その頂点についての $d[j]$ を計算するため、複数のスレッドが同時に同じ要素に書き込む事はなく、競合は発生しない。GPU には配列 DST と配列 W の一部しか送らないため、この 2 つの配列の索引番号が配列 PTR の値と異なることに注意が必要である。その関係を図 4.10 に示す。この図では、一つ目の配列が配列 PTR、二つ目の配列が CPU における配列 DST もしくは配列 W、三つ目の配列が GPU における配列 DST もしくは配列 W を表す。配列 PTR の s と e の間の部分が転送単位 G_i である。CPU においては、配列 DST と配列 W の全て記憶されているが、GPU の場合、G_i に相当する部分のみしか記憶していない。そのため、配列の索引番号が分割前のグラフとずれることになる。図に示す通り、そのズレ幅は $PTR[s]$ の値に等しく、分割後のグラフで配列 DST と配列 W を参照する時は元の索引番号から $PTR[s]$ を引いた値を用いる。配列 PTR と各頂点の始点からの距離を表す配列 c に関しては、配列の大きさが配列 DST や配列 W に比べて小さいため、分割を行わずに全体を転送し、GPU で全体を保持する。ここまでは対象グラフに対して一度のみ行う処理である。

以下の処理はループとして探索終了まで繰り返す。ループの始めに、Δ を求めるためにカーネル delta を実行し、未探索の各頂点に対して、頂点の重みとその隣接辺の重みの最小値の和を求める。隣接辺の重みの最小値はカーネル delta_prepare で求めたものを用いる。カーネル delta の実装の擬似コードを Algorithm 13 に示す。カーネル delta では、i 番目の分割グラフ G_i に対して探索を行っている。

カーネル delta で各頂点に対して探索を行った後は、その結果の最小値をカーネル min で求める。カーネル min は、単純に最小値を求める問題であり、最小値を求めるのに一般的なリダクション演算を用いているため、カーネルの詳細説明は省略する。カーネル min の結果を CPU に転送し、その結果が ∞ の場合、ループから抜けて、アルゴリズムは終了する。

カーネル min の結果が ∞ でない場合、Δ の値を用いて、カーネル update で各頂点の重みを更新する。カーネル update の擬似コードを Algorithm 14 に示す。カーネル update はカーネル delta と同様に、分割グラフ G_i に対して実行する。ある頂点の重みが Δ の値よりも小さい場合、その頂点と隣接する頂点の重みを更新する。並列に更新を行うため、一つの頂点の重みを同時に複数スレッドから変更する可能性がある。この競合を避けるために、この操作は各頂点毎の不可分操作とする。不可分操作は、CUDA の atomicCAS 関数を用いて ATOMIC REGION 内に一度に入るスレッドを一つにすることで実装する。異なる頂点に対する更新は同時に行うことができるため、不可分操作による計算時間の増加は少ない。

カーネル update で更新を終えた後は、while 文の始まりに戻り、再び Δ の計算を行う。全ての頂点に対して探索を終える、もしくは始点からのパスが存在しない頂点のみが未探索頂点の場合 Δ の値は ∞ になり、Algorithm 11 の break 文によってループから抜けて、アルゴリズムが終了する。

4.5.3 GPU を用いた単一始点最短経路問題の並列度

4.5.2 節中のカーネルはカーネル delta_prepare、カーネル delta、カーネル min、カーネル update の四つである。カーネル delta_prepare、カーネル delta の各頂点に対する計算はそれぞれ依存関係が無いため、全て並列に実行できる。カーネル delta_prepare は転送単位毎に複数回に分けて実行され、カーネル delta はグラフ全体に対して実行するため、各転送単位の頂点数を n_i、グラフ全体の頂点数を n とすると、カーネル delta_prepare の並列度は n_i、カーネル delta の並列度は n

Algorithm 11 グラフキャッシュに対する単一始点最短経路問題の実装

1: $c[j] \leftarrow$ ある頂点 j の始点からの距離、j が始点なら初期値 0、始点でなければ初期値 ∞。要素数はグラフ全体の頂点数
2: $v[j] \leftarrow$ ある頂点 j が探索済みかどうかを表す。0 なら未探索、1 なら探索済みとする。初期値は 0。要素数はグラフ全体の頂点数
3: $d[j] \leftarrow$ ある頂点 j の隣接辺の最小値、初期値は ∞、要素数はグラフ全体の頂点数
4: $\Delta[j] \leftarrow$ ある頂点 j におけるΔの候補、初期値は ∞、要素数はグラフ全体の頂点数
5: $Algorithm10$ を用いて転送単位を決定する。各転送単位をチャンク G_iとし、チャンク数は n とする
6: **for** $i = 1$ to n **do**
7: チャンク G_iを転送 $(CPU \rightarrow GPU)$
8: チャンク G_iに対して $delta\_prepare$ カーネルを実行
9: **end for**
10: **while** $TRUE$ **do**
11: 対象グラフ全体に対して、$delta$ カーネルを実行し$\Delta[j]$ を求める
12: min カーネルを実行し、$\Delta[j]$ の最小値であるΔを求める
13: **if** $\Delta \neq \infty$ **then**
14: $break$
15: **end if**
16: **for** $i = 1$ to n **do**
17: チャンク G_iを転送 $(CPU \rightarrow GPU)$
18: チャンク G_iに対して $update$ カーネルを実行
19: **end for**
20: **end while**

Algorithm 12 delta_prepare カーネル

1: $s \leftarrow PTR$ におけるチャンク G_iの最初の要素番号
2: $e \leftarrow PTR$ におけるチャンク G_iの最後の要素番号
3: $W_i \leftarrow$ チャンク G_iに対応する配列 W
4: $j \leftarrow$ スレッド ID + ブロック ID × ブロックの大きさ + s
5: **while** $j < e$ **do**
6: **for** $k = PTR[j]$ to $PTR[j+1]$ **do**
7: **if** $d[j] > W_i[k - PTR[s]]$ **then**
8: $d[j] \leftarrow W_i[k - PTR[s]]$
9: **end if**
10: **end for**
11: $j \leftarrow j +$ ブロックの大きさ × ブロック数
12: **end while**

Algorithm 13 delta カーネル

1: $c[j] \leftarrow$ ある頂点 j の始点からの距離、j が始点なら初期値 0、始点でなければ初期値 ∞。要素数はグラフ全体の頂点数
2: $v[j] \leftarrow$ ある頂点 j が探索済みかどうかを表す。0 なら未探索、1 なら探索済みとする。初期値は 0。要素数はグラフ全体の頂点数
3: $d[j] \leftarrow$ ある頂点 j の隣接辺の最小値、初期値は ∞、要素数はグラフ全体の頂点数
4: $\Delta[j] \leftarrow$ ある頂点 j におけるΔの候補、初期値は ∞、要素数はグラフ全体の頂点数
5: $s \leftarrow PTR$ におけるチャンク G_iの最初の要素番号
6: $j \leftarrow$ スレッド ID + ブロック ID × ブロックの大きさ
7: **while** $j < e$ **do**
8: $\quad \Delta[j] \leftarrow \infty$
9: $\quad$ **if** $c[j] \neq \infty$ かつ $v[j] = 0$ **then**
10: $\quad\quad \Delta[j] \leftarrow c[j] + d_j$
11: $\quad$ **end if**
12: $\quad j \leftarrow j +$ ブロックの大きさ × ブロック数
13: **end while**

Algorithm 14 update カーネル

1: $s \leftarrow PTR$ におけるチャンク G_iの最初の要素番号
2: $e \leftarrow PTR$ におけるチャンク G_iの最後の要素番号
3: $DST_i \leftarrow$ チャンク G_iに対応する配列 DST
4: $W_i \leftarrow$ チャンク G_iに対応する配列 W
5: $j \leftarrow$ スレッド ID + ブロック ID × ブロックの大きさ + s
6: **while** $j < e$ **do**
7: $\quad$ **if** $c[j] \leq \Delta$かつ $v[j] = 0$ **then**
8: $\quad\quad v[j] \leftarrow 1$
9: $\quad\quad$ **for** $k = PTR[j]$ to $PTR[j+1]$ **do**
10: $\quad\quad\quad BEGIN\ ATOMIC\ REGION$
11: $\quad\quad\quad$ **if** $c[DST_i[k - PTR[s]]] > c[j] + W[DST_i[k - PTR[s]]]$ **then**
12: $\quad\quad\quad\quad c[DST_i[k - PTR[s]]] \leftarrow c[j] + W_i[DST_i[k - PTR[s]]]$
13: $\quad\quad\quad$ **end if**
14: $\quad\quad\quad END\ ATOMIC\ REGION$
15: $\quad\quad$ **end for**
16: $\quad$ **end if**
17: $\quad j \leftarrow j +$ ブロックの大きさ × ブロック数
18: **end while**

となる。これらの並列度では、グラフの大きさが GPU 処理に適さないほど小さい場合を除けば GPU 処理が有効である。

カーネル min では、リダクション演算を用いているため、ループの k 周目の並列度は $\frac{n}{2^k}$ である。よって、演算の後半は並列度が低くなり、CPU 処理の方が有効になる。しかし、アルゴリズム全体の実行時間に対するカーネル min の実行時間の割合は小さく、後半の一部分を CPU で行うことによる高速化率は非常に小さいため、本節では考慮しない。

カーネル update では、未探索かつ始点からの距離が Δ 以下であるという条件を満たす頂点のみが探索対象となるため、探索対象となった頂点数が並列度となる。この数が少ない場合には、CPU 処理の方が GPU 処理よりも有効と考えられるが、どちらが有利かの境界は対象グラフによって異なるため、定式化することは困難である。そのため、4.6.7 節で並列度と実行時間の関係を評価し、それに基づいて CPU と GPU 処理を組み合わせた場合の実行時間を示す。

4.6 グラフキャッシュの性能評価

4.6.1 評価環境

評価に用いた CPU は Intel Xeon E5-1620 v2 であり、動作周波数は 3.7GHz、CPU コア数 4、メモリ容量は 128GB である。GPU 処理部分の開発には CUDA6.0 を用いた。対象とする構造型ストレージはグラフ型は Neo4j である。グラフ型ストアの高速化の評価は、GPU は NVIDIA GeForce GTX 780 Ti を用いた。GPU の主な諸元は表 4.1 の通りである。

表 4.1: NVIDIA GeForce GTX 780 Ti の主な諸元

性能項目	GeForce GTX 780 Ti
コア数	2,880
コアクロック	875MHz
メモリバス幅	384bit
メモリバンド幅	336GB/s
メモリ容量	3GB

4.6.2 対象グラフ

評価には、ランダムグラフを用いた。代表的な SNS である Facebook の平均次数がおよそ 200 [67] であるため、ランダムグラフの平均次数は 200 とした。辺の数の合計が 頂点数 $\times$ 200 となるまで辺を作成することでグラフを作成した。グラフの頂点数は、オリジナルの Neo4j とグラフキャッシュを用いた場合の比較の評価では、100,000 頂点から 400,000 頂点まで 100,000 頂点ずつ変化させて評価を行った (計算時間の評価と同じグラフサイズでは、Neo4j のメモリ使用量が評価に使用した計算機のメモリ容量 (128GB) を超えてしまい測定できないためこのグラフサイズとした)。グラフキャッシュに対する CPU 処理と GPU 処理の比較と CPU と GPU の混成実行では $2^{20} \times 1$ から $2^{20} \times 10$ まで 2^{20} ずつ頂点数を変化させて評価を行った。コンシステンシを保証する場合の性能低下の評価では、$2^{20} \times 10$ に固定し、データ更新量を変化させた。

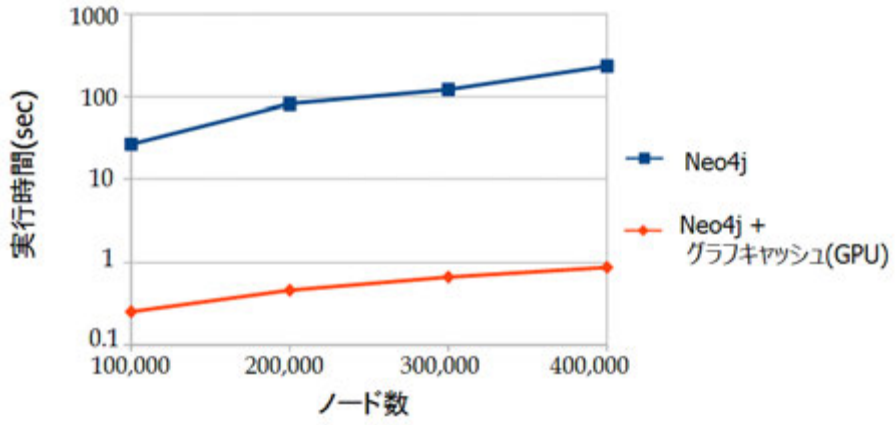


図 4.11: オリジナルの Neo4j とグラフキャッシュを用いた場合の実行時間の比較

4.6.3 オリジナルの Neo4j とグラフキャッシュを用いた場合の比較

4.6.3.1 メモリ使用量の比較

Neo4j でグラフを作成したときのメモリ使用量とデータを抽出して作成したグラフキャッシュのメモリ使用量を比較した。

表 4.2: Neo4j とグラフキャッシュのメモリ使用量

頂点数	Neo4j	キャッシュ	キャッシュ(空き領域 10%)
100,000	20815.3MB	152.6MB	167.8MB
200,000	43115.5MB	305.2MB	335.7MB
300,000	64373.6MB	457.8MB	503.5MB
400,000	79343.1MB	611.4MB	672.5MB

表 4.2 に平均次数 200、頂点数 100,000 から 400,000 のグラフの Neo4j とグラフキャッシュのメモリ使用量を示す。図中の「キャッシュ」はグラフキャッシュを表す。Neo4j に対しては、グラフ作成に必要最低限の情報しか与えずに評価した。そのため、実際の運用で様々な属性が頂点や辺に対して与えられると、Neo4j のメモリ使用量はさらに大きくなる。

表 4.2 に示した通り、グラフキャッシュのメモリ使用量は Neo4j に比べて大幅に削減され、0.71%から 0.77%になった。この大幅なメモリ使用量の削減により、複数台の計算機に保存されている Neo4j のグラフを一台の計算機に収まる大きさにすることが可能である。また、10%の空き領域を持たせても、グラフキャッシュのメモリ使用量は Neo4j に対して 0.78%から 0.85%と極めて小さく、空き領域を持たせてもグラフキャッシュの有効性は保たれる。

4.6.3.2 単一始点最短経路問題の実行時間の比較

図 4.11 に平均次数 200、頂点数 100,000 から 400,000 のグラフに対して、オリジナルの Neo4j とグラフキャッシュを用いて GPU で探索を行った場合の単一始点最短経路問題の実行時間を示す。グラフキャッシュを用いた探索はオリジナルの Neo4j よりも大幅に高速で、頂点数 400,000 の時は 272 倍高速である。

オリジナルの Neo4j の評価では、Java プログラムを用いて Neo4j にアクセスして単一始点最短経路問題を実行した。単一始点最短経路問題は、Neo4j のマニュアルの 35.10 節 [68] を基に実行

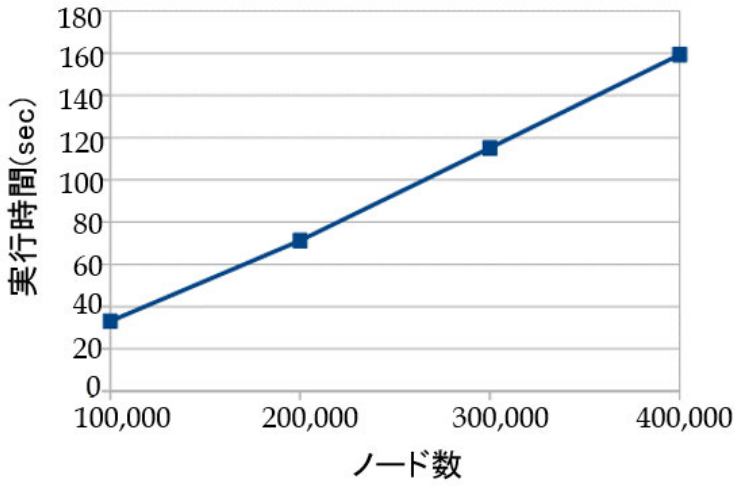


図 4.12: グラフキャッシュの作成時間

した。具体的なクエリを以下に示す。

```
PathFinder<WeightedPath> finder = GraphAlgoFactory.dijkstra(
PathExpanders.forTypeAndDirection( ExampleTypes.MY_TYPE, Direction.BOTH ), "cost" );
WeightedPath path = finder.findSinglePath( nodeA, nodeB );
```

この始点と終点に当たる nodeA、nodeB をランダムに選択してこのクエリを 100 回実行し、実行時間の平均を求めて評価に用いた。

4.6.3.3　グラフキャッシュの作成時間

図 4.12 に、平均次数 200、頂点数 100,000 から 400,000 のグラフに対して、Neo4j からグラフキャッシュを作成する際の実行時間を示す。複数台の計算機を用いた評価では、計算機の台数や通信環境により評価結果が異なり一般性を保つことが困難であるため、この評価は 1 台の計算機を用いた場合のみとした。図 4.12 から、作成時間は頂点数に比例することがわかる。

ここで示す作成時間は、グラフキャッシュを作成するときに一度のみかかる時間であり、探索の性能には殆ど影響しない。性能に影響を与えるのは、4.6.6 節で述べるグラフキャッシュの更新時間である。また、図 4.12 は 4.6.6 節で述べる更新時間よりもグラフキャッシュ作成時間の方が頂点数あたりの実行時間が大きいことを示すが、これは作成の際には更新用リストが存在しないため、直接 Neo4j から情報を読み取る必要があるためである。

4.6.4　拡張性の評価

メモリ使用量と書き込み性能を基に、グラフキャッシュを用いることで拡張性が最大でどの程度改善し得るかを評価した。

図 4.13 は、64GB のメモリを有する計算機を想定した場合に計算機 1 台分のグラフキャッシュでオリジナルの Neo4j の何台分の大きさのグラフ構造を格納できるかを示す。オリジナルの Neo4j は複数台の計算機での実行サポートはしていないが、2.3.5 節で示した文献 [36] の手法によって複数台に分割することができる。1 頂点当たりの平均次数は 200 とし、オリジナルの Neo4j は 1 台の計算機でグラフの作成を繰り返すことで独立した複数台の計算機の代わりとした。図 4.13 か

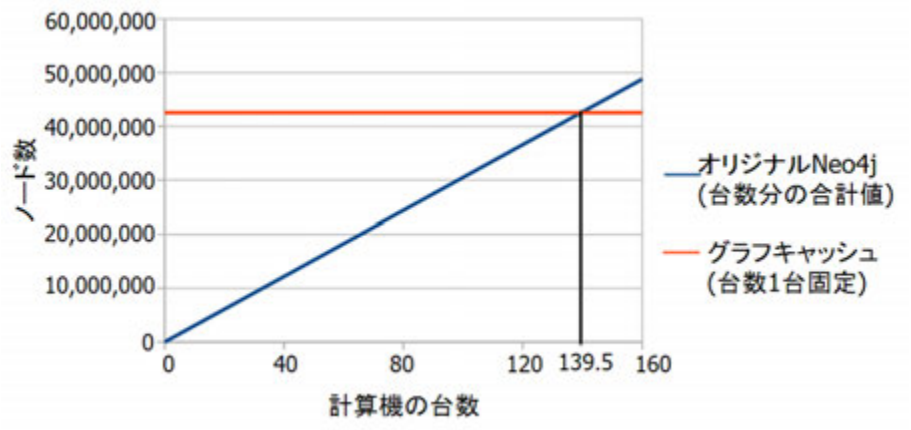


図 4.13: 複数台のオリジナル Neo4j とグラフキャッシュの頂点保存可能数の比較

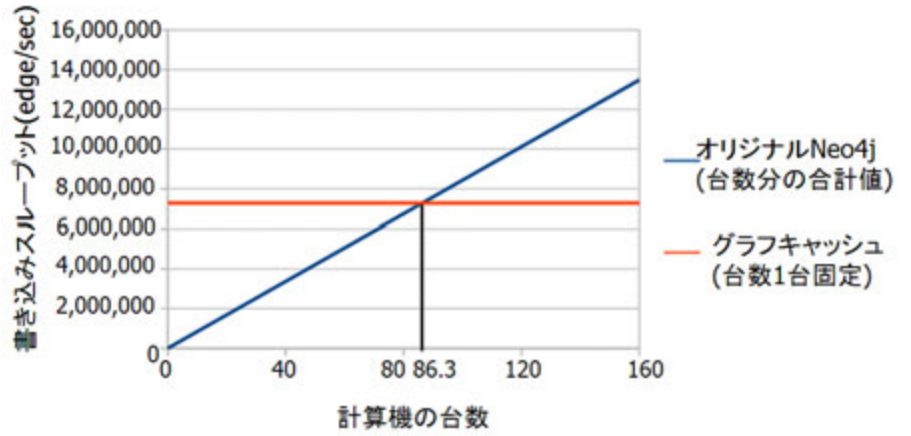


図 4.14: 複数台のオリジナル Neo4j とグラフキャッシュの書き込み性能の比較

ら、オリジナル Neo4j のメモリ効率の場合、139.5 台分のメモリ容量で格納できるグラフサイズを GDB ならば 1 台分のメモリ容量で格納できている。

図 4.14 は、グラフキャッシュとオリジナルの Neo4j の書き込み性能から、オリジナルの Neo4j を何台まで書き込み性能の低下を抑えて拡張できるかを示す。まず、C 言語で実装した TCP/IP サーバプログラムでグラフの更新情報を受信し、グラフキャッシュを更新するようにした。単一の 1GbE スイッチを介して接続された計算機から 64 個の辺の書き込み情報を持つ 1KB データを大量に送信することでグラフキャッシュへの書き込みスループットを測定した。図 4.14 は、1 秒間あたりの頂点または辺の書き込みスループットを表しており、グラフキャッシュの書き込みスループットはオリジナルの Neo4j の 86.3 台分の書き込みスループットの合計値と等しい。なお、今回は安価な 1GbE 通信環境を想定したが、この値はネットワークの混雑度合いや通信規格によって増減する。このようにグラフキャッシュはメモリ利用効率および書き込みスループットの点において高い拡張性を有していると言える。

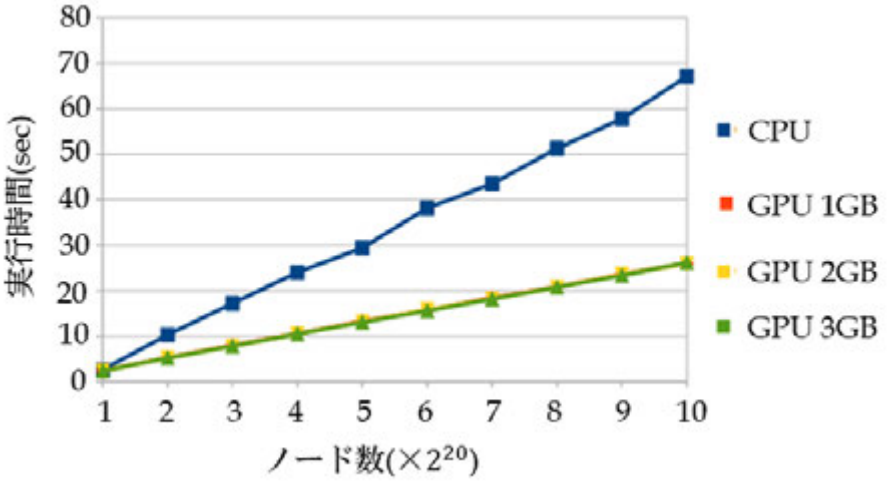


図 4.15: グラフキャッシュに対する CPU と GPU の実行時間

4.6.5　グラフ探索の実行時間

4.6.5.1　グラフキャッシュに対する CPU と GPU の実行時間の比較

グラフキャッシュに対するグラフ探索として、単一始点最短経路問題を GPU と CPU で実行し、その実行時間を比較した。また GPU においては、GPU のメモリ容量は GPU の種類によって異なることを考慮して転送単位を変えて評価を行った。CPU における実装も GPU 実装と同様に Ortega-Arranz らの手法を実装し、評価した。

図 4.15 に、次数 200、頂点数が $2^{20} \times 1$ から $2^{20} \times 10$ までのグラフのグラフキャッシュに対する単一始点最短経路問題の実行時間を示す。CPU は CPU の実行時間を示す。GPU 1GB、GPU 2GB、GPU 3GB は GPU の実行時間で、GPU に確保するメモリの大きさをそれぞれ 1GB、2GB、3GB としたときのの実行時間である。このメモリサイズに合うように 4.5.1 節で述べた方法でグラフキャッシュを転送単位毎に GPU に転送する。ただし、頂点数 2^{20} のグラフキャッシュのサイズはおよそ 1.6GB であるため、2GB と 3GB の GPU メモリの評価では転送単位がグラフ全体となる。そのため、4.5.2 節で示したアルゴリズムのようにループ毎にグラフの転送を行う必要は本来は無いが、図 4.15 では比較のためループ毎の転送を行った。図 4.15 によると、GPU の 3 つの評価結果は重なっており、分割後のグラフサイズは実行時間に影響を与えていないことがわかる。転送するデータ量が多い場合、グラフの転送にかかる時間は転送の回数ではなく、合計転送量に比例するためである。CPU と GPU の実行時間を比較すると、すべての場合で GPU のほうが高速で、最大で 2.5 倍の性能向上を達成している。

また、対象とするグラフが GPU メモリを越えるかどうかでどの程度性能に影響を与えるかを評価するため、GPU に確保するメモリ容量 3GB の時、GPU メモリ内に収まる場合は 4.5.2 節のように繰り返し転送を行わずに実行時間を測定した。GPU メモリを越える場合は、4.5.2 節に述べたようにグラフ全体を繰り返し転送を行っている。その時の実行時間を図 4.16 に示す。図 4.16 から、グラフキャッシュのメモリ容量を越えるおよそ $2^{20} \times 1.75$ を境に、GPU を用いた場合の実行時間が大きく変わっていることが分かる。CPU と比較した場合には、GPU メモリを越えた場合は最大 2.5 倍の性能向上であったのに対し、GPU メモリ以下の場合には、最大 10.1 倍の性能向上となる。このように、単一 GPU を用いた場合には、キャッシュの大きさが GPU のデバイスをメモリを上回ると、性能が大幅に低下する。この問題を解決するために、本章の 4.7 節以降の想定では、複数の GPU へのグラフキャッシュ分散を行い、グラフ全体の繰り返し転送を不要にする。

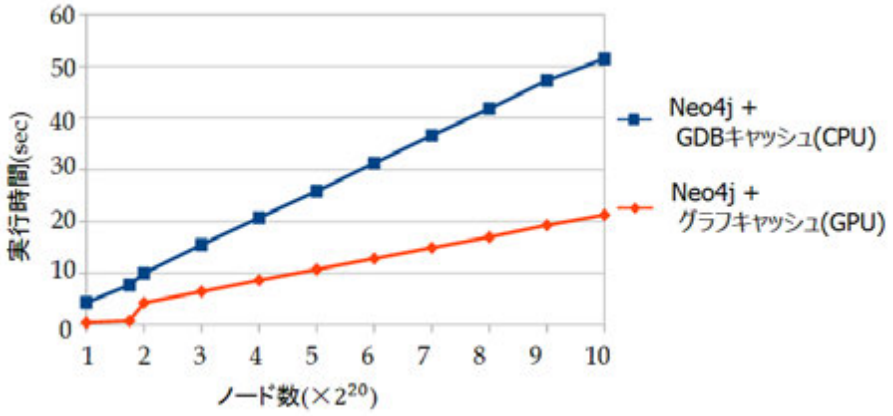


図 4.16: 対象が GPU メモリを越える事によるグラフキャッシュに対する実行時間の影響

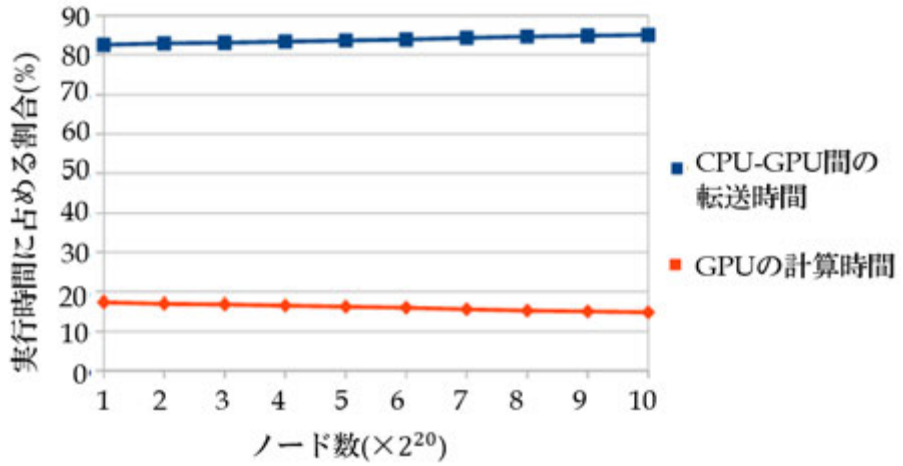


図 4.17: 実行時間に占める計算時間と転送時間の割合

4.6.5.2　グラフキャッシュに対する GPU 処理の実行時間の内訳

グラフキャッシュに対する GPU 処理の実行時間には、CPU-GPU 間のデータ転送と GPU での計算時間の二つの時間が含まれる。本節では、転送単位 3GB の時のこれらの割合を解析する。

図 4.17 に、実行時間に占める計算時間と転送時間の割合を示す。図 4.17 を見ると、実行時間に占める割合の大半はデータ転送時間である。これは、並列 Dijkstra 法のループ毎に転送を行っているためである。並列 Dijkstra 法は、条件に合う頂点を並列に探索するが、各ループ毎に条件に合う頂点の数が異なるため、計算時間がループ毎に変化する。しかし、グラフの転送時間はどのループでも同じである。そのため、計算量が小さいループでは転送時間が実行時間の大半を占め、これが全体の実行時間の割合に影響を与えている。例えば、1 回目のループは条件を満たす頂点は始点 1 つであり、計算量は非常に小さくなり、実行時間にほぼ全てをデータ転送時間が占める。頂点数が増えるにつれ転送時間の割合がわずかに増加しているが、それは 1 回目のループのように頂点数が増えても並列度が変わらないループがあるためである。

1 回目のループのように、転送時間の割合が極端に大きなループは GPU に転送せずに CPU で実行を行い、計算量が多い場合のみ GPU に転送して計算を行うことで、更なる高速化ができる。この高速化については、4.6.7 節で述べる。

また、GPU のメモリサイズがグラフキャッシュのサイズを上回っている場合、グラフキャッシュの転送はループ毎に行う必要がなく、探索開始時の一度のみでよいため、転送時間は短くなる。こ

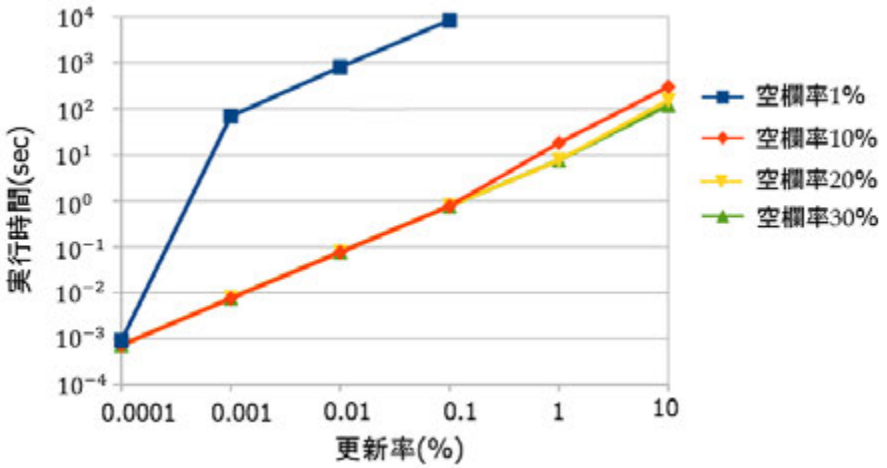


図 4.18: グラフキャッシュの更新率と更新時間の関係

れは、1 台の GPU の場合だけでなく、複数台の GPU を用いた場合にも当てはまる。すなわち、この手法を複数台の GPU を用いて拡張し、複数台の GPU メモリの総量がグラフキャッシュのサイズを上回れば、グラフキャッシュの転送は一度で済む。このような結果を受けて、本章の後半の想定では複数の遠隔 GPU へのグラフキャッシュの拡張を行う。

4.6.6　コンシステンシを保証する場合の性能評価

4.4 節で述べたように、グラフキャッシュと Neo4j との間のコンシステンシを保証する場合には、性能が低下する。性能低下の原因は、グラフキャッシュの更新オーバーヘッドであるため、グラフキャッシュの更新時間を評価する。更新時間を評価するために、空欄率と更新率という二つの用語を定義する。空欄率は、4.4 節で述べたグラフキャッシュに対して持たせる空欄が元のグラフの大きさの何%に相当するかを表す。更新率は、元のグラフの大きさの何%が更新されるか、すなわち、4.4 節で述べた更新用リストの大きさが元のグラフの大きさの何%であるかを表す。

図 4.18 は、次数 200、頂点数 $2^{20} \times 10$ のグラフにおける、更新率と更新時間の関係を空欄率毎に評価した結果を示す両対数グラフである。実際は更新処理は複数台の計算機に跨るため通信が発生するが、通信遅延を含めると評価の一般性を保つのが困難であるため、通信遅延は考慮せずに同一計算機上で更新時間を評価した。グラフキャッシュの更新では、辺や頂点の追加の際にグラフキャッシュに空欄があればそのまま辺や頂点を追加し、空欄がなければグラフキャッシュを再構成する。図 4.18 では、空欄率 1%の実行時間が非常に大きくなっているが、空欄が少ないためすぐに埋まってしまい、グラフキャッシュの再構成が頻繁に実行されるためである。その他の空欄率の場合は、更新率 1%までは再構成が実行されないため、ほぼ同じ実行時間である。更新率 10%の時はすべての空欄率で再構成が実行されるが、空欄率が大きいと再構成の頻度が少ないため実行時間が短い。空欄率が大きいとメモリ使用量が大きくなるため、メモリ使用量と更新時間のトレードオフの関係になる。どの程度の空欄率が良いかは更新頻度や計算機のメモリ量などによって異なるため、ユーザが判断する必要がある。

図 4.18 で示した更新時間が、コンシステンシを保証した時に生じる性能低下に一致する。図 4.15 の GPU 処理の実行時間にこの更新時間を加えると、更新率 10%の場合は空欄率 30%であっても実行時間が 6.8 倍となり、大幅な性能低下となってしまうが、更新率が 0.1%以下の場合は実行時間は 4%の増加にとどまり、性能低下はほとんどない。

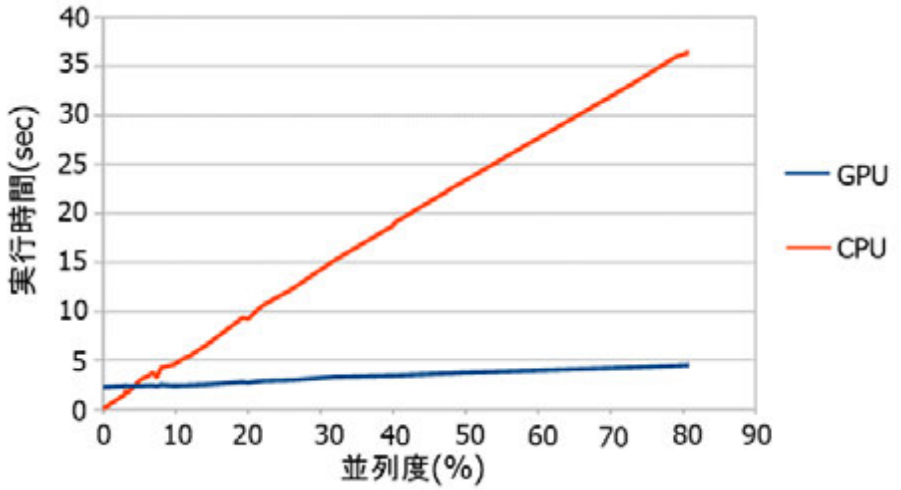


図 4.19: カーネル update の並列度と実行時間の関係

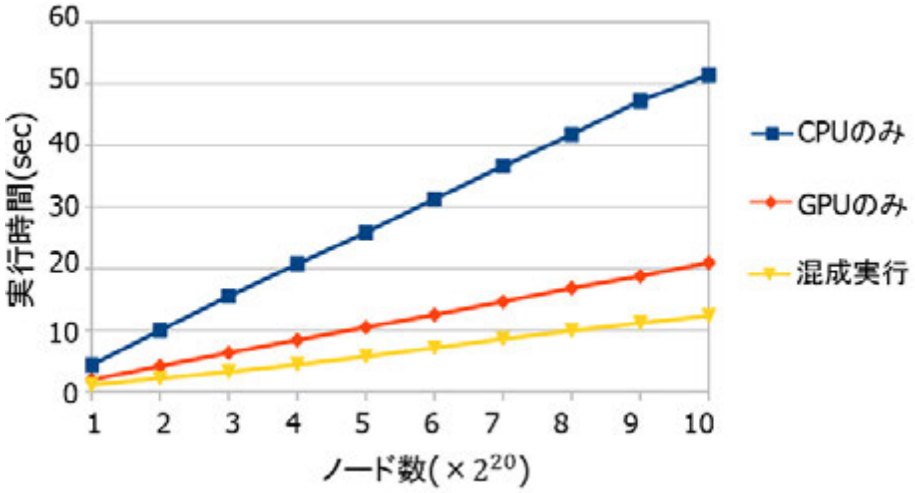


図 4.20: CPU、GPU、混成実行の実行時間

4.6.7 CPU と GPU の混成実行

4.5.3 節で述べたように、カーネル update の並列度は実行する度に異なり、並列度により GPU と CPU のどちらが有効かが異なる。本節では、並列度と実行時間の関係を評価し、それに基づいて CPU と GPU の混成実行することによりどの程度の性能向上が得られるかを評価する。

図 4.19 に次数 200、頂点数 $2^{20} \times 10$ のグラフにおけるカーネル update の並列度と実行時間の関係を示す。並列度はカーネル update で探索対象となる頂点の割合で示しており、100%の時の並列度はグラフの転送単位の頂点数である。図 4.19 から、並列度が小さい場合は CPU 処理の方が高速で、その分岐点は並列度 4%の時である。よって、並列度が 4%未満の時は CPU で実行し、4%以上の時は GPU で実行することにより、カーネル update の実行を高速化できる。

図 4.20 は、次数 200、頂点数が $2^{20} \times 1$ から $2^{20} \times 10$ までのグラフのグラフキャッシュに対して、CPU 実行、GPU 実行、CPU と GPU の混成実行のそれぞれで単一始点最短経路問題を実行した時の実行時間を表す。実行時間は、単一始点最短経路問題を始点をランダムに選択し、100 回実行した際の平均を用いた。GPU のメモリの大きさはすべて 3GB である。CPU と GPU の混成実行では、並列度 4%未満は CPU 実行、4%以上は GPU 実行とした。図 4.20 から、すべての場合で CPU と GPU の混成実行が最も高速であり、GPU のみに対して最大 2.0 倍、CPU のみに対

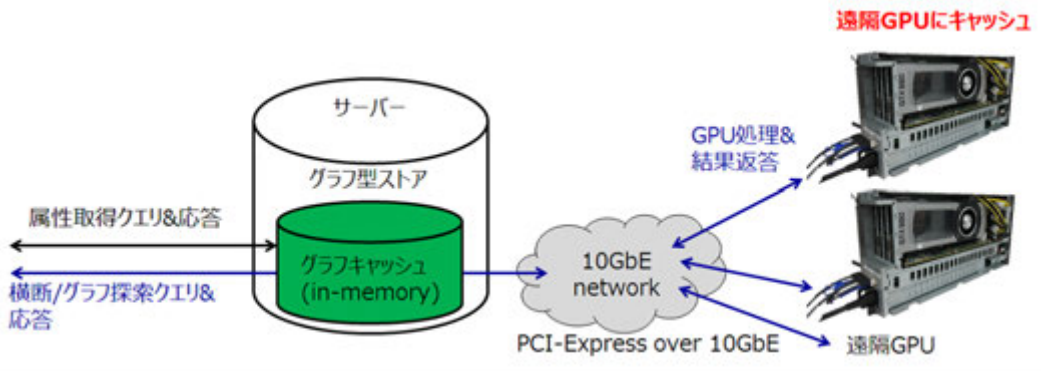


図 4.21: 分散グラフキャッシュシステムの全体像

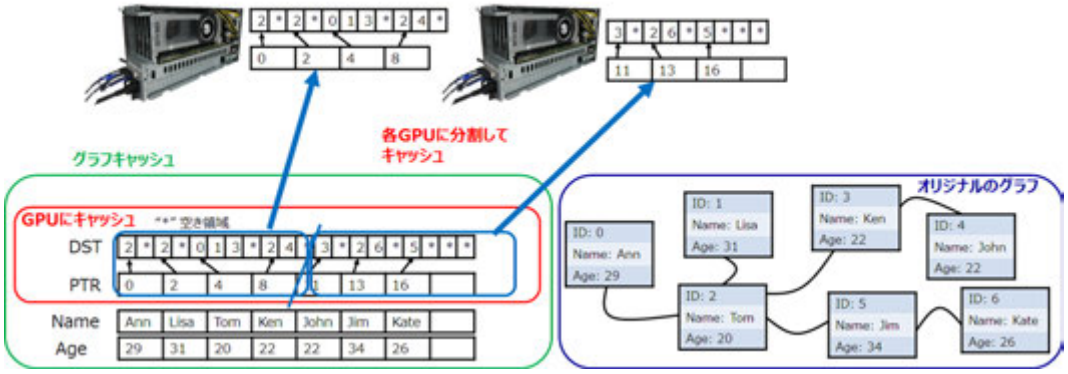


図 4.22: 分散グラフキャッシュシステムのデータ構造

して最大 4.8 倍高速である。

また、混成実行の粒度はクエリ単位ではなく、カーネル単位であるため、CPU 実行は update カーネルの並列度が 4%以下の場合のみ実行される。よって、混成実行の場合は update カーネルの並列度が毎回 4%以下であっても他の delta カーネル等は GPU で実行される。また、アルゴリズムの開始時は必ず並列度が 4%以下となるため、混成実行においては少なくとも 1 回は update カーネルが CPU で実行される。よって、今回の評価条件においては、いかなるクエリにおいても混成実行が CPU 実行または GPU 実行の性能と同一になることはなく、混成実行の方が高速である。

4.7 単一計算機上の分散グラフキャッシュシステムの全体像

図 4.21 に提案する分散グラフキャッシュシステムの全体像を示す。全体像を表しており、図に示す通りオリジナルのグラフ型ストアとグラフキャッシュが同一計算機上にあり、そのグラフキャッシュが 10GbE で接続された複数の GPU へ分散されている。図 4.22 にこの分散グラフキャッシュシステムにおけるデータ構造を示す。緑色の枠内がグラフキャッシュのデータ構造を表しており、赤枠の部分は 4.2 節で述べた構造と同じであるが、そこに Name や Age といった属性が追加されている。グラフを複数の GPU への分散は、4.5.1 節で述べた単一の GPU に複数回分けて送る時の分割手法と同様に分割を行い、4.5.1 節では一つの転送単位としていた単位を各 GPU に割り当てる単位とすることで実現できる。この時、ある頂点の隣接辺が全て同じ GPU に格納されるように分散されている。その様子が図の上半分にある各 GPU におけるキャッシュが表しており、赤枠で示したグラフキャッシュ全体を青枠のように 2 つに分けて各 GPU に分散している。4.2 節の

想定では、属性等の情報が多数あり、属性の情報が大半を占めるグラフ型ストアから、グラフ構造のみを抽出していたが、本節では、属性等の割合が小さい場合を想定しており、属性もグラフキャッシュにキャッシュできる大きさである場合には、グラフキャッシュに属性も一部キャッシュし、属性取得クエリを含む全ての読み込みクエリを高速化する。

すなわち、グラフキャッシュにキャッシュされる情報は以下の通りである。

- グラフの横断もしくはグラフ探索に必要な辺と頂点の情報
- 属性取得クエリにおいて頻繁に要求される辺や頂点の属性の一部

属性部分の構造は、ホストメモリ上で管理されているため、ドキュメントキャッシュのようにバリュー配列を用いる必要はなく、それぞれを文字列として表現する。その上で、配列 PTR の添字と Name や Age といった各属性の属性名を ID として管理して文字列への各文字列へのポインタを用意することで、頂点が与えられた時に属性の読み込みができる。辺の属性の場合は図 4.22 にはないが、頂点と同様に配列 DST の添字と属性名を ID として管理することで、辺の属性の読み込みができる。

また、2.3 節で述べた 3 種類の読み込みクエリである、属性取得クエリ、横断クエリ、グラフ探索クエリの 3 つのクエリは図 4.21 に示すように、それぞれ以下のような流れで処理する。

- 属性取得クエリ:グラフキャッシュにキャッシュされた属性が要求された場合、グラフキャッシュからの結果をグラフ型ストアを介して利用者に返し、そうでなければ、通常のグラフ型ストアと同様にグラフ型ストアで処理する。
- 横断クエリ、グラフ探索クエリ:10GbE で接続された複数の GPU で処理を行い、その結果を集約してグラフ型ストアを介して利用者に返す。

4.8 分散グラフキャッシュにおけるクエリ処理

4.8.1 属性取得クエリ

属性取得クエリで要求されたクエリがグラフキャッシュにおいてキャッシュされている場合にはグラフキャッシュを用いて処理が行われる。その処理は単純にグラフキャッシュの配列 PTR もしくは配列 DST の添字および属性名を用いて属性のアドレスを取得し、そのアドレスを基に属性を取得する。

属性取得クエリは、一般的に横断クエリやグラフ探索クエリと併用して実行される。クエリの例としては、「ある頂点から 2 ホップで、Age が 25 以下の頂点を探す」といったクエリが考えられる。このようなクエリの場合は、頂点からの横断クエリで 2 ホップ離れた頂点全ての属性を取得し、取得した Age 属性の値が 25 以下かどうかを判定する、といった流れで処理できる。

4.8.2 横断クエリ

横断クエリでは、10GbE で遠隔接続された複数の GPU が用いられるため、グラフキャッシュは分割してそれぞれの GPU へ割り当てる必要がある。分割の方法は、4.5.1 節で述べた、GPU のデバイスメモリを越える大きさのキャッシュを複数回に分ける手法と同じ手法で分割できる。すなわち、GPU のデバイスメモリの大きさに合うように配列を区切り、転送単位としていたものを各 GPU へ割り当てる範囲とすることで実現できる。グラフキャッシュが複数の GPU に跨っている場合、横断クエリでも GPU 間を跨る辺を横断する。GPU 間を跨る横断が発生すると、GPU

間で情報の同期が必要であり、そのような横断の度に同期を行うと、GPU 間の同期の頻度が極めて多くなってしまう。そのような頻繁な同期は、オーバーヘッドが大きく、分散グラフキャッシュによる利得を大幅に損ねてしまう。そこで、横断クエリにおいては、複数の横断クエリをまとめてバッチ処理を行い、多数のクエリの同期を同時に行う手法を用いる。その際には、2 章で述べたグラフ全体を一度に同期する同期手法であるバルク同期並列を用いる。バッチ処理によってまとまった量の計算を行った後バルク同期並列を用いることで、計算に対する同期のオーバーヘッドの割合を削減できる。しかし、バッチ処理を行う場合、クエリのレイテンシは増加してしまうという欠点がある。計算量の小さいクエリで、かつスループットよりもレイテンシを重視する場合には、GPU を用いずにキャッシュに対する CPU 処理によって対応することもできる。また、4.6.7 節において示したように、クエリ中の並列度の変化に応じて CPU と GPU の混成実行することによって性能を向上させることもできる。

グラフキャッシュは GPU 処理で頻繁に用いられる配列表現であるため、横断クエリの実装も既存のグラフ処理手法を利用することができる。評価では、横断クエリとして、与えられた頂点から 3 ホップまでの頂点を探索するクエリを実装し、評価した。横断クエリにおいては、文献 [42] を基に CUDA カーネルを実装し、1 ホップと 2 ホップ目は計算量が小さいため CPU で実行する CPU と GPU の混成実行を用いた。

4.8.3 グラフ探索クエリ

横断クエリと異なり、グラフ探索クエリは常にグラフ全体を探索する。そのため、一般的に、グラフ探索クエリの計算量は横断クエリよりも大きく、バッチ処理を行わずに 1 つのクエリを並列に複数 GPU で処理し、クエリ間は逐次的に実行する。グラフ探索クエリにおける同期もバルク同期並列で行うこともできるが、計算量の大きいグラフ探索処理においては、GPU 間の同期を部分的に非同期で行うことによって計算と同期を重複させ、さらに性能を向上することができる。しかし、この同期手法ではデータ構造を一部変更する必要があり、グラフ探索クエリその他のクエリを同一の環境で比較するため、評価においてはバルク同期並列を用いてグラフ探索クエリを評価した。

分散キャッシュを用いた各クエリの評価結果を示した後に、この同期手法および同期手法を用いた際の性能向上率を評価する。

4.9 分散グラフキャッシュの評価

本節の評価に用いた CPU は Intel Xeon E5-2637v3 で動作周波数 3.5GHz、メモリ容量は 512GB である。オリジナルのグラフ型ストアとして Neo4j バージョン 3.1.3 を用いた。分散グラフキャッシュでは、遠隔 GPU として 3 台の GeForce GTX 980 Ti を用い、遠隔 GPU 接続には ExpEther 10G を用いた。GeForce GTX 980 の動作周波数は 1,038MHz でコア数は 2,816 メモリ容量は 6GB である。評価対象としては、次数 100 のランダムグラフおよび、3 種類の実グラフを用いた。実グラフは表 4.3 の 3 種類である。これらの実グラフは、ランダムグラフよりも次数の偏りが大きく、一部の頂点が多くの辺を持つ構造であるという特徴を持つ。

ランダムグラフにおいては、頂点数を 10 万から 320 万まで変化させて評価を行った。このグラフの大きさは Neo4j にメモリ使用量が頂点数 320 万次数 100 のグラフで 295GB となり、Neo4j の制約によって決められたものである。グラフキャッシュにおいては、1 台の GeForce GTX 980 Ti で 8 億辺 (次数 100 のグラフの場合頂点数約 800 万) まで格納でき、この数は GPU 数を増やすことによってさらに増加する。よって、この大きさのグラフは 1 台の GPU で扱うことができるが、ここでは分散キャッシュとの比較のために 3 台に分散させて格納した。

表 4.3: 対象の実グラフの諸元

グラフ名	頂点数	辺数	平均次数
soc-LiveJournal1	4,847,571	68,993,773	14.2
soc-Pokec	1,632,803	30,622,564	18.8
com-Orkut	3,072,441	117,185,083	38.1

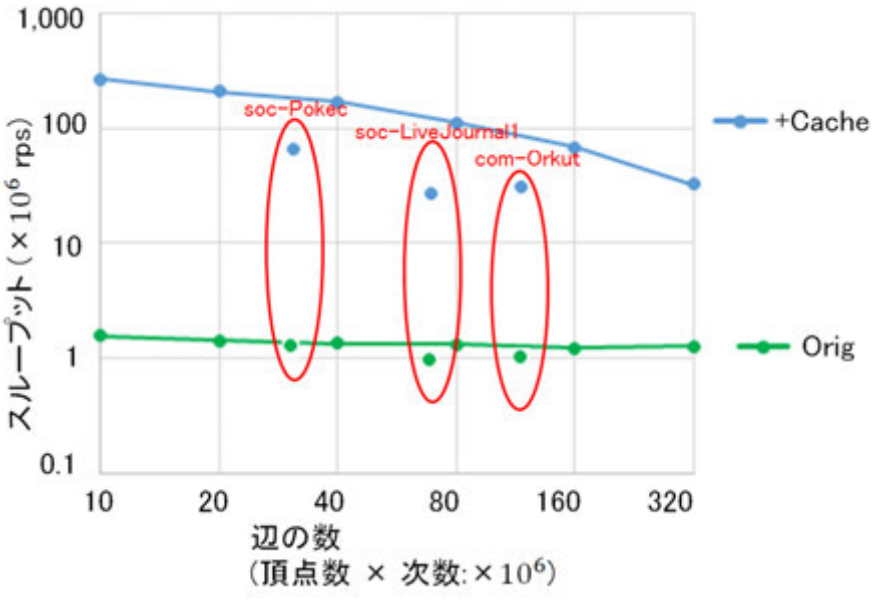


図 4.23: 属性取得クエリのスループット (両対数グラフ)

4.9.1 属性取得クエリ

図 4.23 にオリジナルの Neo4j と提案する分散グラフキャッシュを用いた場合のスループットを示す。グラフキャッシュにおいて、属性取得は GPU 処理は行わないため、オリジナルの Neo4j(Orig) とグラフキャッシュを用いた CPU 処理手法 (+Cache) の 2 つの比較を行った。それぞれのクエリは、ランダムに頂点を選び、その属性を取得するクエリで、スループットは rps(request per sec) で表される。

図から、グラフキャッシュを用いた手法が常にオリジナルの Neo4j の性能を上回っていることがわかる。グラフサイズが小さい場合、CPU キャッシュを使うことができるため、グラフの大きさが大きくなると、グラフキャッシュを用いた手法の性能向上率は低下する。しかし、属性を単純な構造のグラフキャッシュにキャッシュすることにより、頂点数 320 万のグラフにおいても、オリジナルの Neo4j の 25.9 倍の性能を達成した。3 種類の実グラフにおいても、同様の結果が得られた。

4.9.2 横断クエリ

図 4.24 に Orig、+Cache に加えて、分散グラフキャッシュに対して GPU 処理を行った場合 (+Cache+GPU) の 3 つの手法を用いたランダムグラフと 3 つの実グラフにおける横断クエリのスループットを示す。それぞれのクエリは、ランダムに与えられた頂点から 3 ホップの頂点を検索するクエリとする。+Cache+GPU においては、最初の 2 ホップを CPU、最後の 3 ホップ目を GPU で行う混成実行を行った。また、32 クエリをバッチとしてバッチ処理を行った。

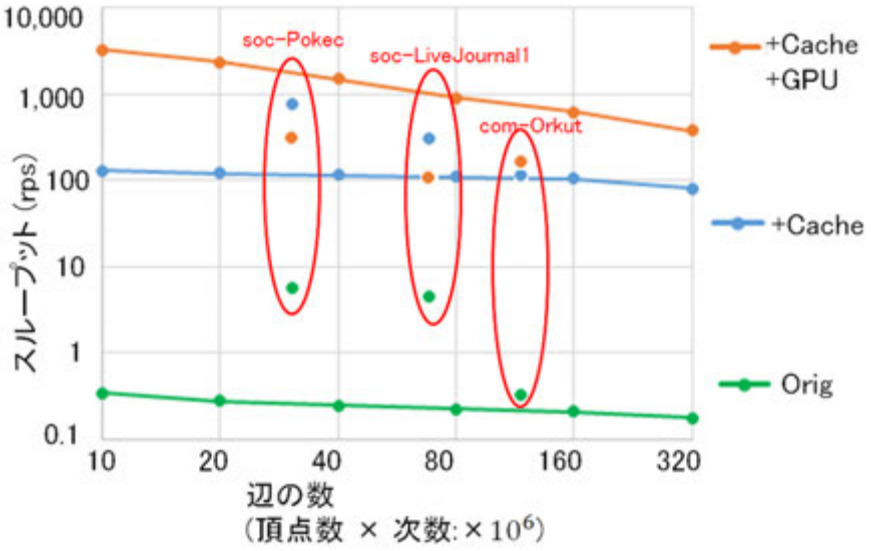


図 4.24: 横断クエリのスループット (両対数グラフ)

+Cache は常に Orig の性能を上回っており、さらに+Cache+GPU がそれを上回る結果となった。しかし、+Cache+GPU は頂点数 10 万の時は+Cache の 25.5 倍のスループットであったのに対し、頂点数 320 万の時は 4.7 倍とグラフが大きくなるにつれて性能向上率が低下している。これは、グラフの一部を探索する横断クエリにおいては、グラフの大きさが大きくなってもクエリの計算量は変わらないのにも関わらず、同期のオーバーヘッドが大きくなるためである。そのため、より大きいグラフを扱う場合には、同期のオーバーヘッドを削減するためにより大きいバッチでバッチ処理を行う必要がある。オリジナルの Neo4j との比較では、頂点数 320 万の時、+Cache が 53.7 倍、+Cache+GPU が 252.4 倍と共に大幅なスループットの向上を達成した。

3 つの実グラフにおいては、+Cache と+Cache+GPU は共に Orig の性能を上回ったが、次数が小さい実グラフにおいては、分散 GPU ストアの並列度を活かせず、+Cache の方が良い性能となった。このような場合には、バッチをより大きくする必要がある。また、実グラフにおいては頂点の次数の偏りが大きいため、全体の探索を行わない横断クエリにおいては各クエリ毎に計算量の偏りが発生するため、バッチ毎の計算量を均一化するという観点からもより大きなバッチサイズが望ましい。

4.9.3　グラフ探索クエリ

グラフ探索クエリとして、SSSP の実行時間を評価した。図 4.25 に Orig、+Cache、+Cache+GPU の 3 つの手法をランダムグラフと 3 つの実グラフにおける SSSP の実行時間を示す。

属性取得クエリや横断クエリはグラフの大きさによって計算量が変化しないが、グラフ探索クエリにおいては、グラフの大きさが大きくなると計算量も大きくなる。そのため、実行時間は全ての場合で頂点数が増えるにつれて増加している。横断クエリとは異なり、+Cache と+Cache+GPU の間の性能の差はグラフが大きくなるにつれて拡大している。Orig と+Cache+GPU の間の性能差も同様で、頂点数が 10 万、80 万、320 万の時でそれぞれ 55.8 倍、156.4 倍、383.2 倍の性能向上となった。

3 つの実グラフにおいてもグラフ探索の計算量が大きいため、+Cache+GPU が他の 2 つの手法の性能を上回る結果となった。また、グラフ全体の探索を行うため、横断クエリのように頂点の次数の偏りによるクエリ毎の計算量の違いは生じない。

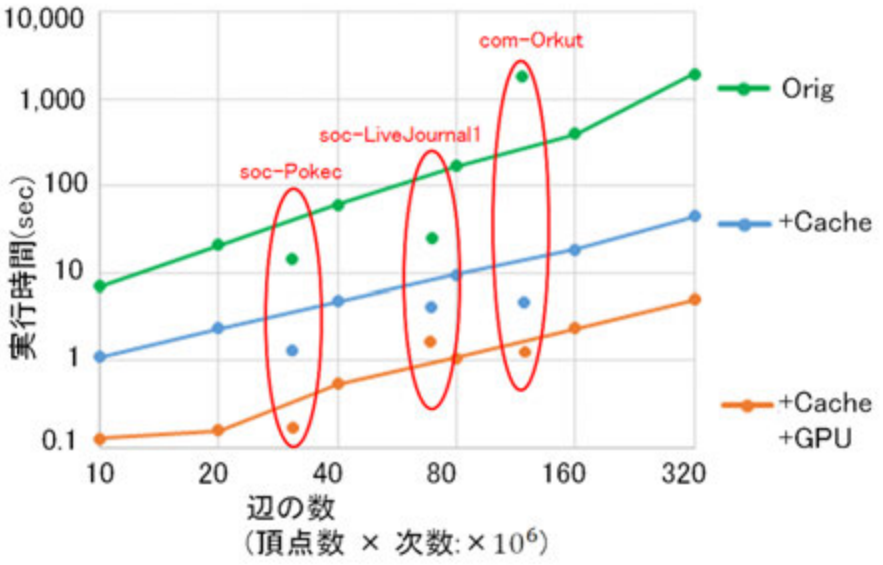


図 4.25: グラフ探索クエリの実行時間 (両対数グラフ)

4.10　遠隔 GPU 間の同期手法

遠隔接続を行う場合、直接接続よりも多くの GPU を利用可能であるが、ホストマシンと GPU 間の転送速度は直接接続よりも低速である。グラフ処理を行う場合、対象とするグラフの転送自体は一度転送を行えば、対象グラフを変更しない限り再度の転送は必要ないため、転送速度の影響は少ない。しかし、グラフ処理に伴う頂点の更新の際には、他の GPU との同期を行う必要があり、同期のオーバーヘッドは転送速度の低下に伴って増加する。さらに、GPU 間の同期では、各 GPU が他の全ての GPU に対して転送を行うため、転送量は、GPU 数を N とすると、$N \times (N-1)$ に比例して大きくなるため、多数の GPU を扱うことが多いリモート GPU 環境では、より同期オーバーヘッドが問題となる。本節では、グラフ型ストアのグラフ探索のような、グラフ探索処理において、グラフの同期を部分的に非同期で行うことで、処理の計算と同期を重複させることで同期オーバーヘッドを削減する手法を提案する。

4.10.1　部分的非同期更新におけるデータ構造

まず、同期手法の前提として、GPU に格納されたグラフは、4.7 節で述べたように、ある頂点の隣接辺が全て同じ GPU に格納されるように、分割されて複数の GPU へ保存されているものとする。その様子を改めて図 4.26 に示す。図の上半分に示すように、グラフ全体は CSR を用いた配列構造で表されており、ポインタを表す配列 PTR と辺の行き先を表す配列 DST の二つの配列からなっている。そして、各頂点の隣接辺がまとまって保存されている。それを赤線のように区切り、各 GPU へ保存されている。各 GPU では、その GPU に割り当てられた頂点の全ての隣接辺を持っている。また、隣接辺の行き先はグラフ全体に及ぶため、結果保存用の配列として、頂点数分の要素数を持った配列を各 GPU が持っている。

4.10.2 節で後述するが、本論文の同期手法では、辺の行き先がどの GPU に属しているかを基に複数回に分けて同期を行うため、データ構造も同期手法に合わせて辺の行き先毎にさらに分割を行う。図 4.27 に、図 4.26 のグラフをさらに辺の行き先に合わせて分割した様子を示す。この図では、図 4.26 における GPU0 のグラフを分割した様子である。辺の行き先が、どの GPU に属しているかによって、グラフを分割し、それぞれを CSR で表現する。この時、各グラフの配列 DST

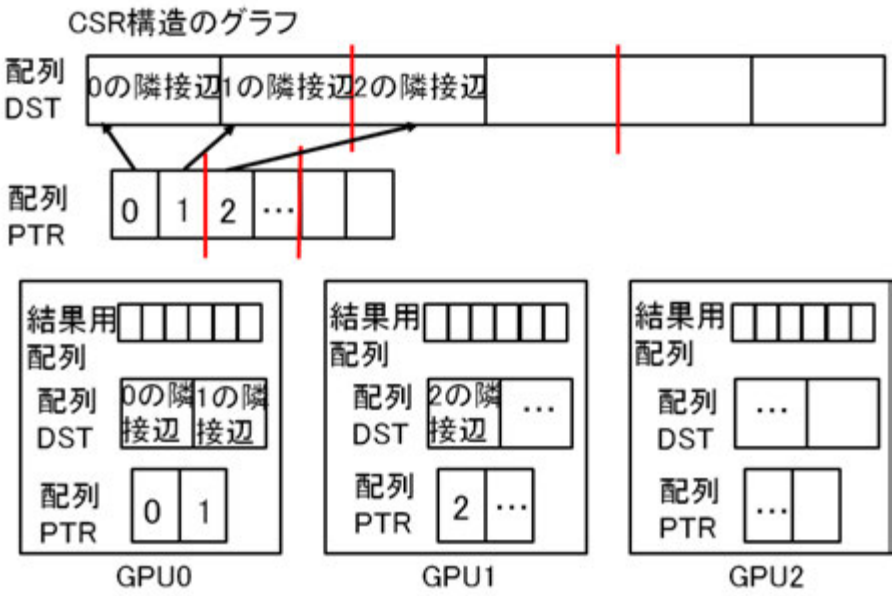


図 4.26: 複数 GPU 向けの CSR

の要素数の合計が基のグラフの配列 DST と一致し、各グラフの計算結果は、辺の行き先に限定され、各グラフの結果格納配列も行き先に合わせて用意すれば良いため、その要素数の合計は基のグラフの結果格納配列の要素数と同じである。配列 PTR は分割後の各グラフに必要なため、この分割によるメモリ使用量の増加は配列 PTR によるものだけである。

また、この分割は計算結果の転送を送る GPU 毎に分けて行うために行うものであり、転送単位をさらに細かくするために、さらに細かく分割を行うこともできる。その場合、各転送単位の大きさを均等になるように、各行き先 GPU 毎に同じ数だけ分割する。ただし、その場合には辺の行き先が同じ GPU 内 (図 4.27 では GPU0) に属するグラフは結果を他の GPU に転送せず、分割の意味がないため、分割を行わない。

4.10.2 リモート GPU 環境での GPU 間同期手法

主に GPU でのグラフ処理に採用されている同期手法である BSP は、本論文で対象とするリモート GPU 環境では、CPU-GPU 間および複数 GPU 間の通信速度が低速であるため、同期の際の通信オーバーヘッドが大きくなる。一方、計算の度に更新を行う非同期更新では、同期の際の通信は計算と重複して行えるが、更新の度に CPU 側から通信命令を行わなければならず、大量の通信命令の発行によるオーバーヘッドが大きくなる。本論文での提案手法では、辺の行き先毎に分割されたグラフを利用して、ある程度の大きさ処理単位毎に計算、転送を行う。それにより、転送回数を抑えつつ、計算と通信を同時に行い、通信オーバーヘッドを隠蔽できる。

Algorithm 15 に提案する手法における各 CPU スレッドの擬似コードを示す。このスレッドを CPU 側で GPU の数と同じ数並列に実行し、各 GPU の制御を行う。4.10.1 節で述べたように、各 GPU に格納されたグラフは、辺の行き先が属している GPU 毎に分割され、さらにそれぞれを定数に基づいて等分と、二段階の分割が行われる。ここでは、ある GPU で扱うグラフ全体を G、GPU 数を n、グラフの分割数を p とする。Algorithm 15 の 8 行目から 10 行目で示す通り、二段階の分割後の各区分を処理単位として、計算、行き先の GPU への結果転送を行う。この時の GPU への結果転送を計算と非同期に実行することで、次以降の区分の計算処理と転送を同時に実行でき、転送のオーバーヘッドを削減出来る。また、Algorithm 15 の 14 行目で示す通り、他の GPU

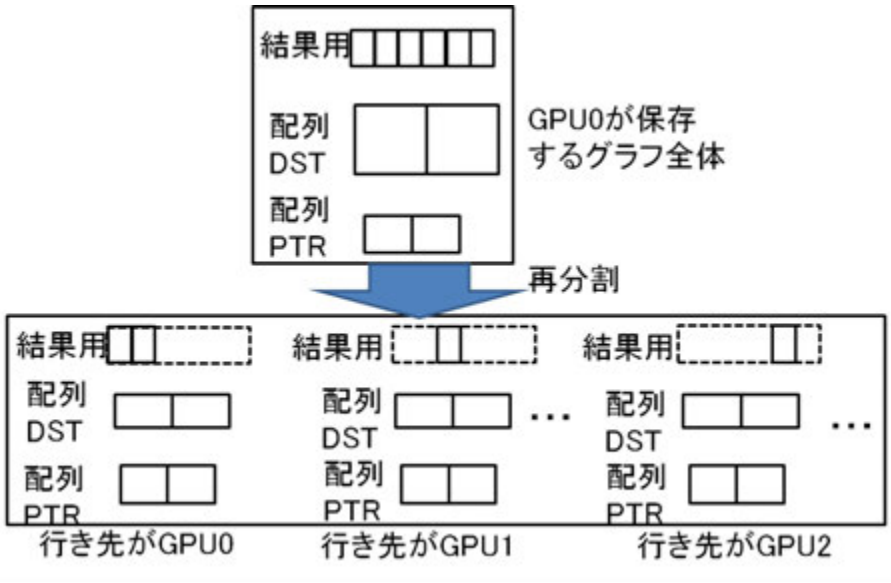


図 4.27: 行き先毎に分割したグラフ

への転送が必要な処理が全て終わった後に、転送の必要のない辺の両端が同じ GPU 内のグラフの計算を行う。終了していない転送が残っている場合には、この計算処理と同時に行うことができる。15 行目までの処理が、BSP における全頂点での計算処理に相当する。

ここまでの処理を全ての GPU に対して完了すれば、グラフ全体での計算および転送処理が完了している。よって、それらの結果を基に更新を行えば、BSP を用いて更新した場合と同じ結果が得られる。このスレッドの計算処理の部分で各アルゴリズムの計算処理を行うことで、既存の様々なアルゴリズムを実行できる。また、スレッドの終了後に、アルゴリズムの終了条件を満たすかどうかを判定し、条件を満たすまで繰り返しスレッドを実行することで、アルゴリズム全体の処理とする。

4.10.3 同期手法を適用したグラフ処理

本章では、代表的なグラフ探索処理である幅優先探索と単一始点最短経路問題をそれぞれ Merrill らの手法 [42] と Ortega-Arranz の手法 [41] を基に本論文の同期手法を適用し、実装を行い、分散グラフキャッシュに適用した。また、GPU 処理の実装環境として、NVIDIA 社が提供する開発環境である CUDA [64] を用いた。

本節の同期手法は、15 の 8 行目と 14 行目にあたる GPU カーネルにそれぞれにアルゴリズム用のカーネルに変えることで各アルゴリズムに適用出来る。

Algorithm 16 に幅優先探索の GPU カーネルを示す。このカーネルを GPU スレッド数を各 GPU に属する頂点数と一致させて実行する。このカーネルの実装において、本節の同期手法を適用するために、通常の単一 GPU での実装からの変更は必要なく、本節が既存のアルゴリズムに容易に適用できることを示している。同期手法における転送では、結果格納用配列の $R[]$ および配列 $v[]$ を転送する。このカーネルを Algorithm 15 に適用し、配列 $v[]$ の値が全て 1 になり、全ての頂点が探索済みとなった時点でアルゴリズムを終了する。

Algorithm 17 に単一始点最短経路問題の GPU カーネルを示す。幅優先探索のカーネルと同様に、GPU スレッド数を各 GPU に属する頂点数と一致させて実行する。このアルゴリズムでは、この計算以外に、前計算として、各頂点の隣接辺の重みの最小値を求める処理と、Algorithm 15 の一度の実行毎に Δ の更新処理が必要である。Δ は、式 2.1 で示した様に、未探索の全ての頂点

Algorithm 15 提案手法における CPU スレッドの動作

```
1: n ← GPU の数
2: p ← 転送先の各 GPU 毎の分割数
3: G ← 処理対象のグラフ、G_npで分割後の各区分を表す。この場合 G_np は n 番目の GPU を行き先とする p 番目の区分を意味する
4: x ← このスレッドが担当する GPU の ID
5: for i = 1 to p do
6:   for j = 1 to n do
7:     if j ≠ n then
8:       G_jiに対するグラフの横断、及び計算処理用の GPU カーネルの実行
9:       カーネルの終了まで待機
10:      G_jiにおける計算結果を j 番目の GPU への転送を非同期実行
11:    end if
12:  end for
13: end for
14: G_xに対するグラフの横断、及び計算処理用の GPU カーネルの実行
15: カーネルの終了まで待機
16: 全 GPU でここまでの処理が終わるまで待機
17: 他の GPU からの計算結果を基に G_xの結果配列を更新
```

において、各頂点の隣接辺の重みの最小値と $R[]$ に保存される始点からの距離の和を求め、それらの最小値を新しい Δ とすることで更新を行う。この更新処理は、本節の同期手法を用いるかどうかに関わらず発生するものであり、同期手法による追加オーバーヘッドとはならない。GPU カーネルは、幅優先探索と同様に単一 GPU での実装からの変更は必要ない。転送も、幅優先探索と同様に $R[]$ および $v[]$ を転送し、全ての頂点が探索済みとなった時点でアルゴリズムを終了する。

Algorithm 16 幅優先探索の GPU カーネル

```
1: V ← 対象グラフの頂点数
2: R[] ← 結果格納用配列、要素数はグラフ全体の頂点数。アルゴリズムの始点のみ 0、その他は ∞ で初期化。このアルゴリズムでは、始点からのホップ数が保存される
3: tid ← GPU スレッドの ID、スレッド数が V と一致
4: x ← 各スレッドが担当する頂点、グラフ内の tid 番目の頂点
5: v[] ← ある頂点が探索済みかどうかを表す配列。0 なら未探索、1 なら探索済みとする。初期値は 0、要素数はグラフ全体の頂点数
6: d ← 現在探索対象とする頂点の始点からの距離、初期値 0、カーネル呼び出し毎にインクリメントする
7: if R[x] = d then
8:   for i = PTR[tid] to PTR[tid + 1] do
9:     if v[DST[i]] = 0 then
10:      R[DST[i]] ← d + 1
11:      v[DST[i]] ← 1
12:    end if
13:  end for
14: end if
```

Algorithm 17 単一始点最短経路問題の GPU カーネル

```
1: V ← 対象グラフの頂点数
2: R[] ← 結果格納用配列、要素数はグラフ全体の頂点数。アルゴリズムの始点のみ 0、その他は ∞ で初期化。このアルゴリズムでは、始点からの距離が保存される
3: tid ← GPU スレッドの ID、スレッド数が V と一致
4: x ← 各スレッドが担当する頂点、グラフ内の tid 番目の頂点
5: v[] ← ある頂点が探索済みかどうかを表す配列。0 なら未探索、1 なら探索済みとする。初期値は 0、要素数はグラフ全体の頂点数
6: W[] ← 重み格納用配列、要素数は配列 DST と一致
7: Δ ←　式 2.1 によって求められる値
8: if R[x] < Δ かつ v[x] = 0 then
9:   v[x] ← 1
10:  for i = PTR[tid] to PTR[tid + 1] do
11:    if R[x] + W[DST[i]] < R[DST[i]] then
12:      R[DST[i]] ← R[x] + W[DST[i]]
13:    end if
14:  end for
15: end if
```

4.11 遠隔 GPU 間の同期手法の評価

4.11.1 評価環境

評価で用いた CPU は Intel Xeon E5-2637 v3 で、動作周波数は 3.5GHz であり、メモリ容量は 128GB である。GPU の評価では、NVIDIA Geforce 980、CUDA バージョン 6.5 を用いた。GPU のコア数は 2,048、コアクロックは 1,126MHz、メモリ容量は 4GB である。その他の諸元の詳細は、ドキュメントキャッシュにおける評価である 3.4 節の表 3.1 に示しているため、ここでは省く。本評価で用いる GPU の数は 3 台とし、ホストと各 GPU 間は NEC 10Gb ExpEther を用いて 10GbE ケーブル 2 本を用いて接続する [56]。

4.11.2 BSP と提案手法との比較

BSP は、図 4.26 で示した提案手法用に各 GPU 内での再分割を行っていない状態のデータ構造において、提案手法と同様の GPU カーネルを用いて実装した。また、提案手法は、GPU 内での再分割での分割数 p が 1、3、5 のそれぞれで評価を行った。

対象のグラフとして、平均次数 10、100 で頂点数を変えて生成したランダムグラフと、実グラフとして SNAP のデータセット [69] 中のソーシャルグラフ 2 種類を用いた。対象のソーシャルグラフの諸元は、表 4.4 の通りである。

表 4.4: 対象のソーシャルグラフの諸元

グラフ名	頂点数	辺数	平均次数
soc-LiveJournal1	4,847,571	68,993,773	14.2
soc-Pokec	1,632,803	30,622,564	18.8

4.11.2.1 幅優先探索の実行時間

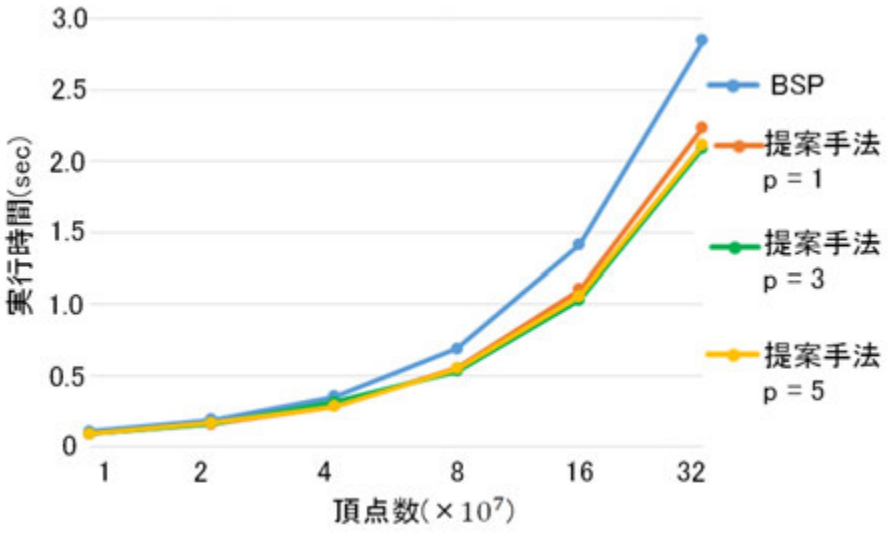


図 4.28: 平均次数 10 のグラフにおける幅優先探索の各同期手法における実行時間

図 4.28 に平均次数 10 のグラフにおける幅優先探索の実行時間を示す。グラフの頂点数は 100 万から 3,200 万まで変えて評価を行った。全てのグラフにおいて、提案手法は BSP の実行時間より短く、グラフの規模が大きくなるにつれて提案手法の高速化率が大きくなった。また、提案手法の各分割数を比較すると、小さなグラフでは、$p = 1$ の時が高速で、大きなグラフでは $p = 3$ の時が高速であった。また、最も BSP よりも提案手法が高速であるのは、頂点数 3,200 万のグラフにおける提案手法の $p = 3$ の時であり、BSP の 1.36 倍高速であった。これは、小さなグラフよりも大きなグラフの方が、転送量と計算量が共に増加するため、より転送オーバーヘッドが削減できることと、小さなグラフでグラフの分割数が増加すると、各カーネルの計算量が小さく、GPU の並列度を活かせなくなるためと考えられる。

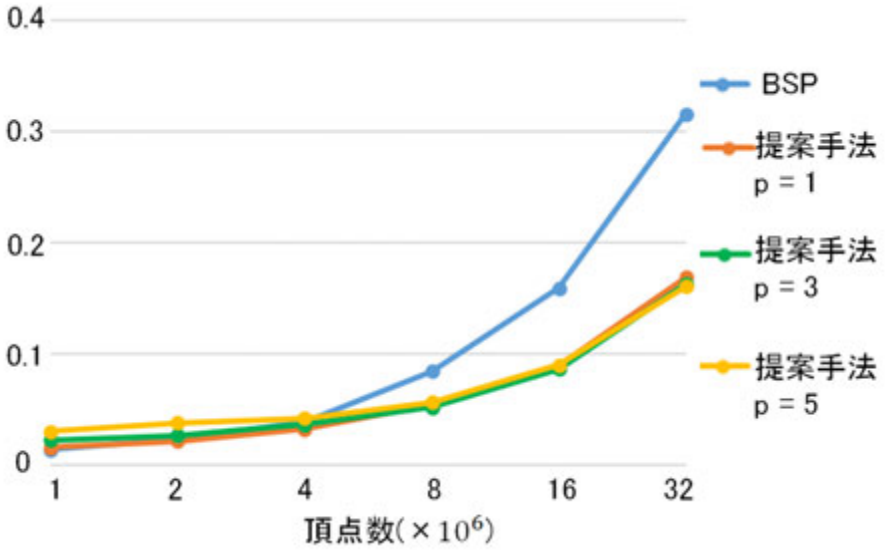


図 4.29: 平均次数 100 のグラフにおける幅優先探索の各同期手法における実行時間

図 4.29 に平均次数 100 のグラフにおける幅優先探索の実行時間を示す。グラフの頂点数は 10 万から 320 万まで変えて評価を行った。頂点数が 10 万のグラフにおいては、全ての提案手法が BSP よりも低速であった。これは、頂点数が少なく、同期の際の転送量も小さいため、GPU カーネルの実行回数や転送回数の増加によるオーバーヘッドが転送オーバーヘッドの削減効果を上回ったためである。そのため、提案手法の中でも $p = 5$ の実行時間が最も長かった。一方、大きなグラフにおいては、提案手法の方が BSP よりも高速であり、頂点数 320 万のグラフでは、$p = 5$ の提案手法が最も高速であった。平均次数 10 のグラフより、転送量に対する計算量の割合が大きいため、計算によって隠蔽できる転送オーバーヘッドの割合が大きく、頂点数 320 万の $p = 5$ の時に BSP の 1.97 倍と、高速化率も大きくなった。

表 4.5: ソーシャルグラフにおける幅優先探索の各同期手法における実行時間

グラフ名	BSP	提案 (p=1)	提案 (p=3)	提案 (p=5)
soc-LiveJournal1	0.517	0.458	0.476	0.543
soc-Pokec	0.172	0.161	0.162	0.171

表 4.5 にソーシャルグラフにおける幅優先探索の実行時間を示す。紙面の都合上、提案手法はそれぞれ提案と省略して示す。表 4.4 に示した評価対象のグラフの諸元は、本評価における平均次数 10 のグラフの規模の小さいグラフと近い。そのため、評価結果も平均次数 10 のグラフと同様の傾向を示しており、$p = 1$ の提案手法の実行時間が 2 つのグラフ共に最も短い。これらのグラ

フにおいても、提案手法は BFS よりも高速であるが、より規模が大きい実グラフでは、高速化率がさらに向上すると考えられる。

4.11.2.2　単一始点最短経路問題の実行時間

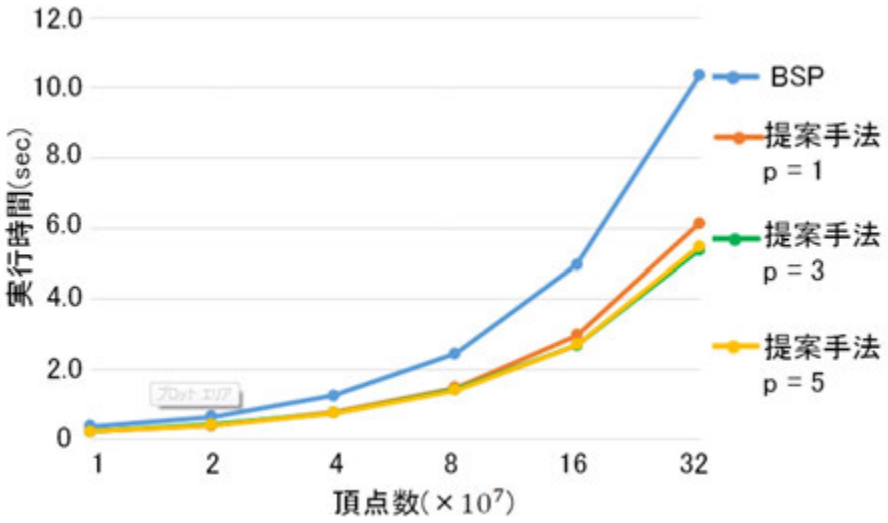


図 4.30: 平均次数 10 のグラフにおける単一始点最短経路問題の各同期手法における実行時間

図 4.30 に平均次数 10 のグラフにおける単一始点最短経路問題の実行時間を示す。頂点数は幅優先探索の評価と同じく、100 万から 3,200 万である。実行時間の傾向は、幅優先探索とほぼ同じであるが、幅優先探索よりもアルゴリズムの計算量が大きく、より多くの転送を計算で隠蔽できるため、高速化率も大きくなり、頂点数 3,200 万の $p = 3$ の提案手法は BSP の 1.89 倍高速であった。

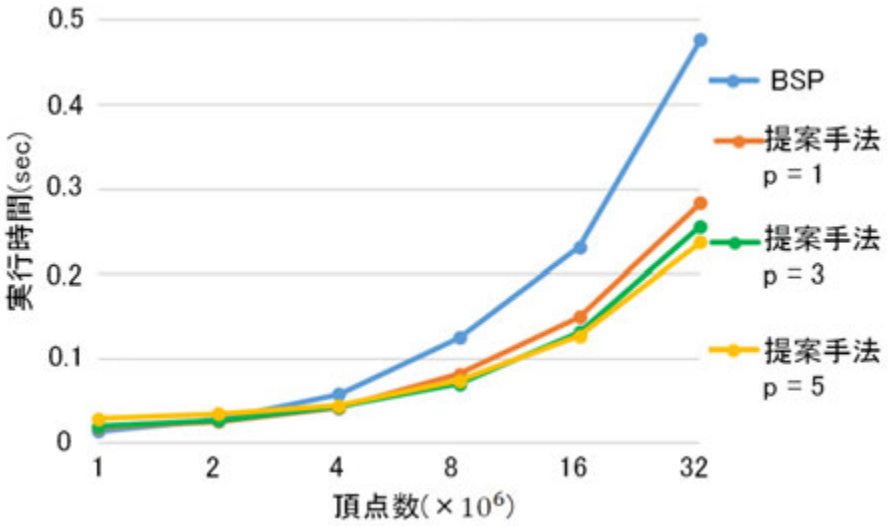


図 4.31: 平均次数 100 のグラフにおける単一始点最短経路問題の各同期手法における実行時間

図 4.31 に平均次数 100 のグラフにおける単一始点最短経路問題の実行時間を示す。頂点数は幅優先探索の評価と同じく、10 万から 320 万である。この結果も平均次数 10 の場合と同様に、幅

優先探索の結果と同じ傾向となった。しかし、平均次数 100 の場合は、幅優先探索であっても転送に対する計算量が大きいため、アルゴリズムの違いによる高速化率の増加は平均次数 10 の場合に比べて小さく、頂点数 320 万の $p = 5$ の提案手法の速度は、BSP の 2.01 倍となった。

表 4.6: ソーシャルグラフにおける単一始点最短経路問題の各同期手法における実行時間

グラフ名	BSP	提案 (p=1)	提案 (p=3)	提案 (p=5)
soc-LiveJournal1	1.71	1.20	1.25	1.30
soc-Pokec	0.412	0.385	0.385	0.398

表 4.6 にソーシャルグラフにおける単一始点最短経路問題の実行時間を示す。幅優先探索と同様に、$p = 1$ の提案手法が高速であるが、「soc-LiveJournal1」のグラフの高速化率は、幅優先探索よりも大きい。これは、平均次数 10 のグラフにおける高速化率の向上と同じく、アルゴリズムに違いによって隠蔽できる転送オーバーヘッドが増加したためである。「soc-Pokec」のグラフにおいても、高速化率が向上しているが、グラフの大きさが小さいため、その割合は小さい。

これらの評価結果から、提案手法は、頂点数が非常に小さいグラフを除いて BSP よりも高速であり、高速化率は、グラフが大きくなるにつれて大きくなることがわかる。本論文で対象とするリモート GPU 環境は、多数の GPU を用いて大きなグラフを処理することを想定した環境であるため、本論文の提案手法は、リモート GPU 環境に適しているといえる。

第 5 章

キーバリューストアへの応用

本章までにおいて、本論文では、構造型ストレージにおいて計算量の大きく、ポリグロット永続化の際のボトルネックとなる正規表現探索とグラフ探索を対象にそれぞれに特化したキャッシュを用いることで高速化する手法を提案した。それぞれのキャッシュの構造は特定用途に特化するという基本的な考え方に基づき、GPU 処理に適した配列構造かつ、各構造型ストレージのボトルネックとなる処理に特化した構造である。この考え方に基づくキャッシュの構造は、KVS 型の構造型ストレージにも適用可能である。また、特定用途に特化するという考え方をさらに発展させれば、ある構造型ストレージを用いる特定のアプリケーションに特化して、さらなる高速化を図ることも可能である。

本章では、本論文のキャッシュの応用として、KVS 型の構造型ストレージをデータの管理に用いており、かつ近年普及が進んでいるブロックチェーンを対象のアプリケーションとして、それに特化した構造のキャッシュを提案することで、本論文の応用を示す。

5.1 ブロックチェーンの概要

ブロックチェーンは、暗号通貨ビットコイン [70] で提案された P2P ネットワークで構成される分散型台帳システムである。近年、ビットコインをはじめとする暗号通貨の普及のみならず、通貨以外の資産の取引 [71][72] や契約の自動化 [73][74] 等の単純な取引以外への応用への期待からブロックチェーンへの関心が高まっている。このように、ブロックチェーンには様々なアプリケーションが考えられるが、本章では、現在最も普及しているブロックチェーンのアプリケーションであるビットコイン [70] で用いられているブロックチェーンを対象とし、以降で述べる内容はビットコインのブロックチェーンを基に述べる。

ブロックチェーンでは、P2P ネットワークに参加しているノードに取引情報が伝搬され、各ノードでその取引情報を検証している。この時、ネットワーク上の全ての取引を受け取り、検証するノードはフルノード、自分に関係がある一部の取引のみを検証するノードは SPV(Simplified Payment Verification) ノードと呼ばれる。また、フルノードは単に取引を検証するだけでなく、他のフルノードや SPV ノードの要求に応じて自分が持つ取引情報をネットワークに伝搬させている。利用者にとって、フルノードを立てるのは負荷が大きいが、SPV ノードは検証する取引が大幅に削減されるため、負荷が小さい。さらに、特定のフルノードを信用するのであれば、そのフルノードを立てている事業者に資産を預けることで、ノードを立てずに取引を行える。実際に、ビットコインを用いた取引所、決済、財布機能等のサービスの利用者の多くはノードを立てずにビットコインを利用している。例えば、平成 29 年 9 月現在、ビットコインネットワークのフルノードの数は約 9 千である [75] のに対し、財布機能等を提供する大手事業者の coinbase [76] の利用者は約 1040 万人であり、フルノードの数に比べて非常に多くの利用者がフルノードを介してビットコインを利用していることが分かる。よって、多数の利用者からの残高、取引内容、取引履歴等の

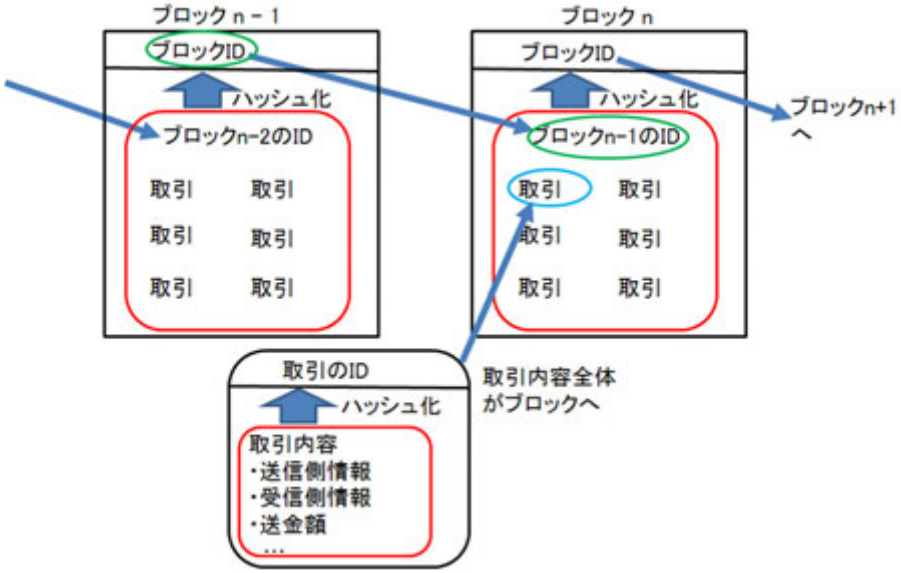


図 5.1: ブロックチェーンのデータ構造の概要

確認といったブロックチェーン検索クエリがフルノードに集中するため、フルノードの応答性能がシステムのボトルネックとなり得る。

ブロックチェーンでは、基本的に古いブロックの更新、削除は行われず、新しいブロックの追加のみ行われる。この特徴を活かすことで、更新や削除を考慮しなければならない一般的な構造型ストレージの検索クエリ以上に高速化できる。本章では、ブロックチェーンの特徴と GPU の特性双方に適した基数木の配列表現を用いることを提案し、ブロックチェーンの検索を高速化する。

5.2 ブロックチェーンのデータ構造

図 5.1 にブロックチェーンのデータ構造の概要を示す。図の上部に示す通り、ブロックチェーンは複数のブロックから構成されており、ブロックはブロック ID、一つ前のブロックの ID、取引内容の集合からなる。ブロックに一つ前のブロックの ID が含まれ、それが最初のブロックまで連なっていることから、その構造を鎖に例えてブロックチェーンと呼ばれている。また、各ブロックの ID は、ブロックに含まれる情報のハッシュ値であり、その ID は以降の全てのブロックに影響を与える。その特徴から、あるブロックの内容を変更すると、最新のブロックまで全てのブロックの ID が変わるため、取引内容の改竄に対する耐性が高い。図の下部は各取引のデータ構造の概要を示しており、各取引は、ID と取引内容を含む。取引内容には、送受信者や送金額等の取引に必要な内容が含まれ、これらのハッシュ値が ID となる。

ブロックチェーンへの書き込みは、最新のチェーンにつながる新しいブロックの追加のみが行われ、過去の取引内容、取引 ID、ブロック ID 等の更新や削除は行われない (厳密には、チェーンの分岐や再構成等で過去の全てのブロックが無効になり得る。しかし、古いブロックほど無効になる確率は低く、ビットコインのシステムでは最新のブロックから 6 ブロック前のブロックは無効になることは無いものとして運用されている。)。新しいブロックの生成の際には、悪意のあるネットワーク参加者による不正なブロック生成を防ぐために、ネットワークが正しいブロックと判定するための合意形成手法が用いられ、その手法に基づく条件を満たした場合のみブロックが生成される。合意形成手法には様々な手法が提案されており、代表的なものとして、PoW(Proof of Work)、PoS(Proof of Stake)、PBFT(Practical Byzantine Fault Tolelance) などがあげられる [77]。PoW は、ビットコインで用いられている合意形成手法であり [70]、ブロックの内容を僅か

に変更してハッシュ化を行ってブロック ID を求めるという処理を繰り返し、ブロック ID が一定の条件を満たす bit 列 (一定数の 0 が連続する bit 列) となった場合に正しいブロックとする手法である。PoS は、Peercoin [78] で導入された合意形成手法であり、保有している暗号通貨の量に応じてブロックを生成できる確率を定める手法である。PBFT は、HyperLedger [79] において提案された手法で、ブロック承認権を持つノードを定め、そのうち一定数のノードが承認したブロックを正しいブロックとする手法である。PoW と PoS は、特定のノードを信頼せずに合意形成を行う手法であり、PBFT は特定のノードを信頼する必要がある。そのため、PoW と PoS はパブリックチェーンと呼ばれる不特定多数のノードが自由に参加できるネットワークで用いられ、PBFT はプライベートチェーンと呼ばれる参加者が限定されるネットワークで主に用いられる。このように、合意形成手法には様々な手法があげられるが、それぞれの手法によって、ブロック作成の条件が異なるだけで、各取引のデータ構造などのブロックチェーンの基本的な構造は合意形成の手法によって変化しない。よって、本章では、ビットコインのブロックチェーンを対象としているが、その他の合意形成手法によって作成されたシステムにも本章の提案は適用可能であり、仮にビットコインの仕様が変更され、合意形成手法が PoS などに変化したとしてもそのまま適用可能である。

このような合意形成に基づいて追加されたブロックの情報は、各ノードへ伝搬されるが、伝搬に生じる遅延を考慮して、多くのノードが取引内容を共有できるようにブロックの追加間隔やブロックの大きさに制限が設けられている。また、過去のブロックの削除が行われないことから、システムが動作し続ける限りチェーンのデータ量は増加するため、ブロックの追加間隔や大きさの制限はデータ量の増加速度を抑える意味もある。この制限を緩和し、書き込みスループットを向上させるために、複数のチェーンを利用する手法が提案されている [80]。この手法では、複数のチェーンを取引が跨ぐことを可能にし、それぞれのチェーンのノード数を制限している。書き込みスループットの制限は、多数のノードへの伝搬のために設けられているため、ノード数を制限することで、この制限を緩和することができる。

一方で、ブロックチェーンの検索等の読み込みクエリは、このような制限はなく、フルノードへアクセスする利用者が増加するのに伴って増加し、フルノードの処理のボトルネックとなり得る。

本章では、更新や削除が行われないというブロックチェーンの特徴を活かしたデータ構造を提案し、ブロックチェーン検索を GPU を用いて高速化する。

5.3 既存の KVS 型の構造型ストレージの高速化

ブロックチェーンに用いられている KVS 型の構造型ストレージは、既に GPU や FPGA を用いた高速化が示されている。本節で、それらの既存手法を紹介し、次節以降でブロックチェーンに特化したデータ構造を提案し、既存の GPU を用いた手法と比較することで、ブロックチェーンに特化することで更なる高速化が実現できることを示す。

また、KVS 型の構造型ストレージの高速化は、本論文で対象とする特定の処理に適したアクセラレータと同様に特定用途に特化した構造型ストレージの組み合わせによる高速化の端緒である。本論文では、本節で述べるこれらの手法をさらに発展させ、構造型ストレージ全体および複数の構造型ストレージを組み合わせたポリグロット永続化へとその流れを拡大させている。

5.3.1 GPU を用いた KVS の高速化

構造型ストレージの一種である KVS の GPU による高速化は Hertherington らが行っている [81][1]。対象とする KVS は Memcached である。

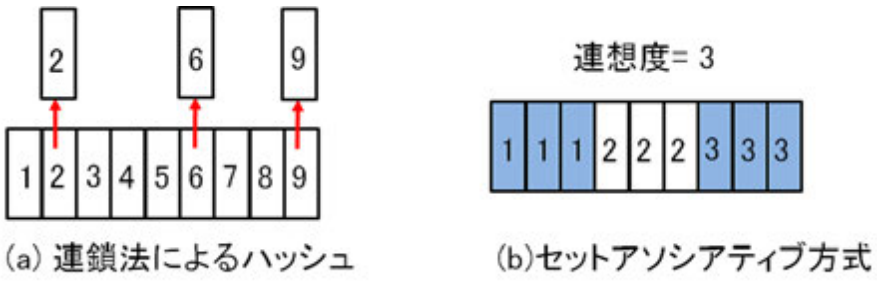


図 5.2: 文献 [1] で提案された GPU 上のハッシュテーブルの構造

GPU の並列性を活かすためには、多数のスレッドを用意して GPU に与える必要がある。しかし、KVS は SET や GET といった単純な操作しか行わないため、1 回の操作のリクエストでは多数のスレッドに分ける事はできない。そこで、数万リクエストといった大量のリクエストを蓄え、それらをまとめて GPU に与えて、GPU で一括処理することにより GPU の並列性を活かしている。

この手法では、GPU でのキーの探索処理にハッシュテーブルを用いている。GPU 上のハッシュテーブルの構造を図 5.2 に示す。対比のため、CPU で処理する際に一般的に用いられる構造も示している。図 5.2(a) がハッシュ値が衝突した際に用いられる一般的な手法の一つである連鎖法における構造を示している。連鎖法に基づく CPU 上の処理では、衝突が発生するとメモリを動的確保し、それぞれのキーをポインタを用いて表現する。しかし、GPU では、動的なメモリ確保は性能を悪化させる [82] ことと、ハッシュ値によって鎖の長さが異なり、分岐が発生によっても性能が悪化することから、連鎖法によるハッシュテーブルの実装は GPU には適していない。また、衝突の対処法として、衝突時に再ハッシュを行うオープンアドレス法もあげられるが、この手法も再ハッシュの回数がハッシュ値毎に異なるため GPU 処理に適さず、また、KVS のような削除を要する用途においてはメモリ利用効率が非効率的になる点もメモリ容量が限られた GPU には適さない。よって、オープンアドレス法によるハッシュテーブルの実装も GPU を用いた KVS の高速化には適していない。このような問題に対処するため、彼らは図 5.2(b) に示すような固定サイズのセットアソシアティブハッシュテーブルを提案した。この手法は、予めハッシュテーブル全体の大きさと連想度が決め、ハッシュテーブル全体のメモリを静的に確保する手法である。図では、連想度が 3 の場合を例示している。このように連想度を事前に定めることで、ハッシュ値による処理内容の違いを抑え、分岐の数を減らすことができる。しかし、連想度を越える数の衝突が起こった場合、ハッシュテーブルからあふれてしまうため、この手法はキャッシュとして用いられ、連想度を越えてしまった場合はキャッシュミスを起こして CPU 処理を行うこととなる。連想度を増やせば、キャッシュミスの起こる確率が低下するが、一つのハッシュ値に保存されるキーの数が増え、探索対象を特定するために行わなければならない比較の数が増加するため、計算量が増加してしまう。文献 [1] 中の評価では、連想度が 16 の時が計算量とミス率を考慮して適切であるとしている。

さらに、この手法では、NIC(Network Interface Controller) と GPU を GPUDirect [83] を用いてホストを介さずに GPU へ転送することで、転送のオーバーヘッドを削減している。このハッシュテーブルと NIC との直接転送を併用することで、評価では最大 1,300 万 rps(requests per second) の性能を達成した。

カラム指向型への拡張については、Hertherington らは対象としていないが、2.1.1 節で述べたように、カラム指向型は内部的には KVS 型に変換でき、かつ範囲検索以外のクエリは、KVS 型と類似したクエリであるため、この手法と同様の手法はカラム指向型の範囲検索以外のクエリにも応用できると考えられる。

本章においても、KVS型を利用するブロックチェーン検索を実用例としてGPUによる高速化を行う。ブロックチェーン検索においては、ブロックチェーンでは更新や削除を考慮しなくても良いという特徴を活かすことで、更なる高速化を実現できる。

5.3.2 FPGA NICを用いたKVSの高速化

2.1.2節で述べた通り、KVS型のクエリは計算量が小さい。そのため、計算ではなく、I/Oインテンシブな処理である[84]。その特徴から、FPGA(Field-Programmable Gate Array)を搭載したNICをキャッシュとして用いて、キャッシュヒット時にはホストを介さずに応答することで、KVSを高速化する手法が提案されている[85]。この手法では、FPGA上のDRAMに簡易版のKVSを実装しており、そのキーの検索には、GPUによる手法と同様の大きさ固定のセットアソシアティブハッシュテーブルが用いられている。GPUと同様に、FPGAのDRAMの容量もホストメモリより小さく、さらに、GPU処理の場合は検索に用いるキーのみをデバイスメモリに保持すればよいが、FPGAの場合はホストを介さずに処理するためにバリューも保持しなければならない。そのため、GPUによる手法よりもキャッシュできるエントリ数は少ないが、ヒット時には高速に応答できるため、局所性が高いワークロードにおいては特に有効である。評価では、NICのラインレートである10Gbpsでの応答に成功している。

この手法を用いた場合においても、キャッシュミス時には、ホスト上でのキー探索を行う必要があり、ミス時にはGPUを用いた手法を用いることが出来るため、この手法と文献[1]で述べた手法や本研究のGPU向けキャッシュ手法との併用によって、更なる高速化ができると考えられる。

また、本論文での実問題の応用としてキャッシュを適用するブロックチェーンの検索においても、FPGA NICを用いた高速化がなされており、数Gbpsでの応答に成功している[86]。ブロックチェーンにおいても同様に、本論文での手法とFPGAを用いた手法は併用可能である。

5.4 キャッシュを用いたGPUによるブロックチェーン検索

5.4.1 ブロックチェーン検索システムの全体像

KVS型は2章で述べた通り、キーとバリューの組からなる単純な構造の構造型ストレージである。KVSのクエリも単純な読み書きのGETとSETといったクエリのみからなる。ブロックチェーンのフルノードにおいて、KVSがデータの管理に用いられており、実際に最も普及しているビットコインのフルノードの実装であるBitcoin Core[87]においてもKVSの一種であるLevelDB[88]が用いられている。

ブロックチェーンの検索クエリは、SPVノードの検証用の情報取得、フルノードが運営する事業の利用者からの取引内容確認、取引履歴確認等があげられる。これらのクエリは、各取引のIDもしくは各利用者のアドレスをキー、各クエリで要求される情報をバリューとしてKVSに保存し、キーを検索することで実行できる。

ブロックチェーンには、5.2節で述べた通り、ブロックの更新や削除が行われないという特徴がある。この特徴を用いることで、更新や削除を考慮しなければならない一般的なKVSよりも効率的にブロックチェーンの検索が行える。図5.3に本章で提案するGPUを用いたブロックチェーン検索手法の概要を示す。まず、キーバリューの組をGPU上にキー、CPU上にバリューに分割して保持する。その際に、GPU上では、バリューの代わりに仮のバリューとして整数値を用意し、CPU上では、バリューの位置を示すポインタの配列を作成する。この際、仮バリューとポインタ配列の添字が一致するように仮バリューとポインタ配列を作成する。ブロックチェーンの書き込み処理は、各キーバリューの組の追加しか存在しないため、追加された順に仮バリューと添字共

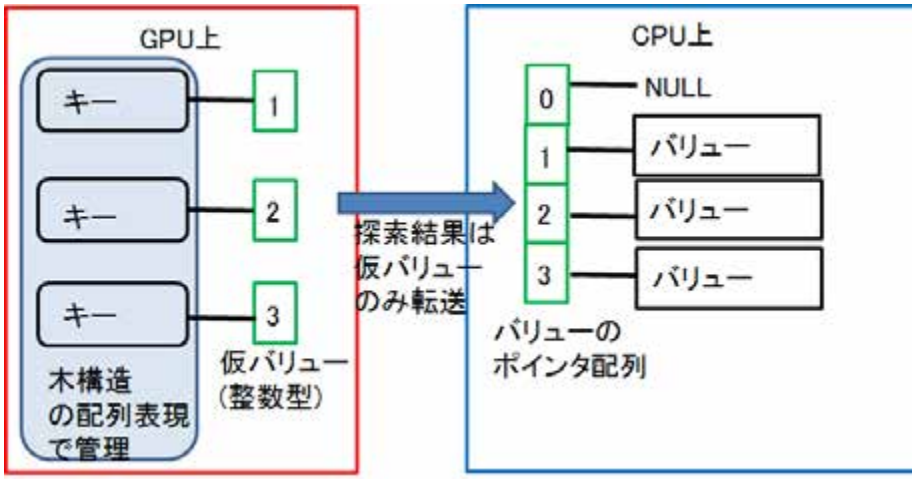


図 5.3: ブロックチェーン検索手法の全体像

に 1 から順にインクリメントすることで作成できる。また、仮バリュー 0 と添字 0 は探索の際に一致するキーが存在しなかった事を示す NULL に用いる。

このように仮のバリューを設けることで、GPU 上では、キーの検索結果が整数値で得られる。CPU 上では、GPU でのキーの検索で得られた仮バリューがバリューのポインタ配列の添字に一致しているため、仮バリューを用いた配列へのアクセス 1 回のみでバリューを取得できる。さらに、この手法では結果の転送は整数値で表される仮バリューのみと極めて小さくでき、GPU 上にバリューを保存する必要がないため、GPU のデバイスメモリの使用量も削減できる。また、GPU 上のキーの検索は、GPU での検索に適した配列表現の木構造を用いる。このように、本論文で提案する配列構造のキャッシュをキーと仮バリューのに応用し、ポインタ配列でドキュメント等のポインタを指す手法を仮バリューとバリューの関係に応用することで、ブロックチェーン検索に本論文の提案手法を応用できる。ブロックチェーンにおけるキー検索に用いる配列構造の詳細および GPU 処理の詳細は次節以降で述べる。

5.4.2　キー検索に用いるデータ構造

前述した通り、ブロックチェーンの検索には取引の ID や利用者のアドレスが用いられる。これらはそれぞれ、取引内容のハッシュ値と利用者の公開鍵のハッシュ値であり、ビットコインの場合、取引 ID は 256bit、アドレスは 160bit(ヘッダとチェックサムを含める場合 272bit) である。256bit の場合も 160bit の場合も処理内容は同様に表せるため、以降ではキーの長さは 256bit とする。2 章で述べた既存の KVS 高速化手法 [81] では、固定長のセットアソシアティブのハッシュテーブル [81] を用いて高速化を行っていた。しかし、この手法には二つの問題点がある。一つ目は、セットアソシアティブの連想度を越える数の衝突が発生した場合、キャッシュミスが発生することである。二つ目は、衝突によって、一つのハッシュ値に二つ以上のキーが対応する場合に、どのキーが検索されたキーと一致するかを確認するためにクエリ毎に複数回のキーの比較が発生し、GPU の計算量が大きくなることである。本章では、本論文で提案した配列構造のキャッシュをブロックチェーンの特徴に合わせることで、これらの問題を解決できることを示す。

具体的には、本章では、キーの各 bit を基数とする基数木を用いることを提案する。図 5.4 に基数木を用いた例を示す。図の左側に示す 4 つのキーを基数木で表したものが右側の木構造である。図中の丸が頂点を表し、頂点無いの数字が何 bit 目を探索するかを表している。各頂点は、2 つの子を持ち、それぞれ探索対象が 0 の時と 1 の時の行き先である。基数木ではキーの集合のう

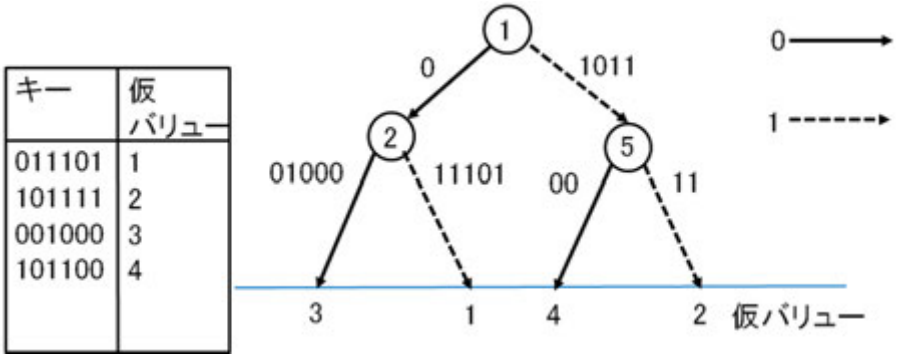


図 5.4: 基数木を用いたキーの表現例

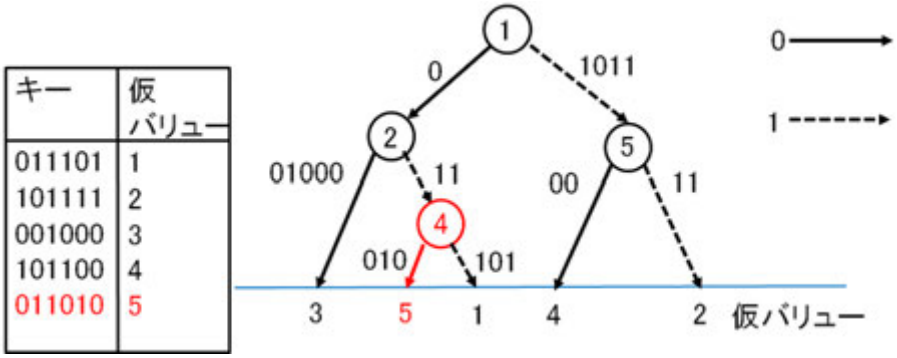


図 5.5: 基数木への新たなキーの追加の例

ち、共通する部分に関しては共通の辺をたどることで探索を行わずに、異なる箇所のみ探索を行う。そのため、木の頂点は、探索が必要なキーが共通しない箇所に作られる。図の例では、最初の bit が 1 のキーは 101111 と 101100 であるが、前半の 1011 は共通しており、後半の 11 と 00 が異なるため、4bit 目で探索が必要となり頂点が作られる。その際、共通する 1011 の bit 数である 4 を保存しておくことで、探索の際に次が何 bit 目を探索すれば良いかを特定できる。よって、各頂点毎に、探索箇所が 0 の時と 1 の時それぞれの行き先と共通する bit 数の合計 4 要素を持つ。また、各キーは一意であるため、共通しない箇所毎に作られる頂点の数は、キーの数を n とすると、$n-1$ となる。

また、木に新しいキーを追加する場合、途中まで木の探索をし、不一致となる箇所に新たな頂点を追加する。図 5.5 に新たなキーの追加の例を示す。新たに 011010 というキーを追加する場合、011 までは既存の仮バリュー 1 を示す 011101 と共通であり、共通しなくなる箇所は 3bit 目であるため、3bit 目に新たな頂点を追加する。

一般的に、木構造の辺の行き先はポインタで表し、ポインタが多用されるため、GPU での利用は適さない。そこで、本論文では、基数木を GPU 処理に適した配列で表現して GPU 上に保持する。ブロックチェーンでは、キーの更新や削除は行われず、追加処理のみであるため、頂点が追加された順に配列に格納するだけで、容易に配列構造で表現できる。具体的には、各頂点の 4 つの要素に合わせて 4 つの配列を用意し、先頭から順に頂点を追加する。

図 5.6 にキーの木構造を配列で表現した例を示す。左側の木構造を右側の 4 つの配列で表現している。4 つの配列は探索する bit が 0 の時と 1 の時それぞれの行き先と共通する bit 数である移動数を保存する配列からなる。左側の木構造の頂点の横の数字が頂点が追加された順番であり、配列の添字を表している。この配列表現では、各頂点の位置が配列の添字で表せるため、辺の行き先は配列の添字をポインタとして用いることで表現できる。また、子ノードがなく、仮バリュー

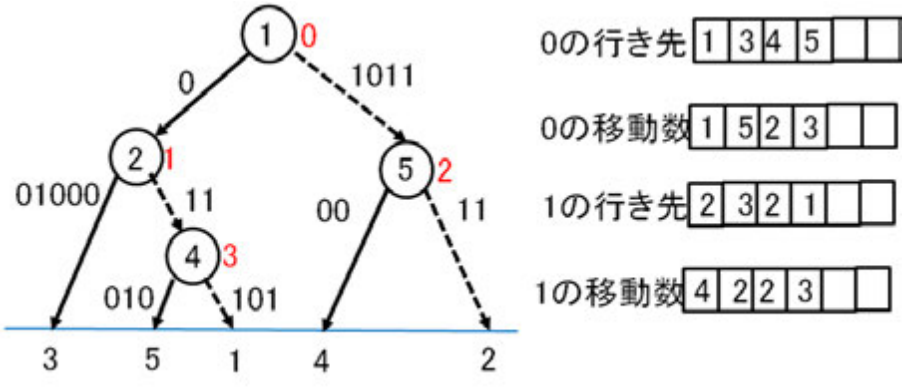


図 5.6: キーの配列表現

が子となる頂点 (図の例では頂点 1、2、3) では、行き先を配列の添字の代わりに仮バリューとする。これにより、探索終了時の行き先を取得することで仮バリューを取得できる。この配列を用いた探索処理の詳細は次節で述べる。

5.4.3　GPU を用いたキーの探索処理

本論文では、GPU 処理の実装に NVIDIA 社が提供する GPU 向けの開発環境の CUDA を用いる [64]。各キーの検索毎に CUDA スレッドを発行し、多数のスレッドを並列実行することで、多数のキーの検索を並列に実行する。この際、GPU の並列度を活かすためには、並列度に合わせてクエリをまとめてバッチ処理を行う必要がある。バッチサイズがどの程度が適しているかは 5.5 で評価する。

検索処理を Algorithm 18 に示す。1 行目の ptr が探索中の頂点、2 行目の $searchptr$ がキーの探索箇所を表す。3 行目と 4 行目の 4 つの配列が基数木を表す配列である。5 行目の $length$ はキーの長さであり、定数として与えられる。ビットコインの取引 ID の例では 256 である。6 行目の $key[]$ が与えられるキーで長さ $length$ の bit 列からなる。7 行目の条件は、探索が終了するかどうかを判定する。現在の探索位置がキー長よりも小さければ、探索途中であることを示す。8 行目から 10 行目が各 bit の探索処理であるが、8 行目は単なる一時保存であるため、実質的な探索処理は 9 行目と 10 行目である。$key[buffer]$ は現在の探索位置の bit を表し、0 か 1 で表される。よって、9 行目と 10 行目は、探索位置の bit に対応する頂点の行き先と次の探索位置への更新である。この 2 行が単なる代入と和の計算のみで表せるため、この探索は途中に条件分岐が発生せず、GPU で効率的に処理できる。各頂点の葉では、行き先が仮バリューを表すため、探索終了後は ptr を返すことで、仮バリューが得られる。このように、この手法では一つのキーに対して一つの仮バリューが得られ、既存手法のように一つのハッシュ値と複数のキーが対応しているという関係にない。そのため、クエリ毎に複数回のキーの比較を行ったり、キャッシュミスが発生し得るという既存手法の問題点を解決できる。

この検索処理によって仮バリューを得られるが、この探索では与えられたキーが木を構築するキーの集合に含まれない場合は仮バリューに対応するキーと一致しない。例えば、図 5.4 に対して 010000 というキーを与えると、仮バリュー 1 が得られるが、これは仮バリューに対応する 011101 と一致しない。そのため、探索対象としてに与えられたキーと仮バリューに対応するキーを比較して検証を行う必要がある。この検証の処理は、探索対象と結果として得られたキーを比較し、一致していれば仮バリューをそのまま返し、一致しなければ NULL を表す仮バリュー 0 を返す処理である。この時のキーの比較はキー長の bit 列同士の単純な比較である。この検証を GPU で行

Algorithm 18 キーの検索処理

1: $ptr = 0$//現在探索中の頂点、初期値は根にあたる 0
2: $searchptr = 0$//現在何 bit 目を探索しているかを表す変数
3: $P_0[]$、$P_1[]$//木の頂点でそれぞれ bit が 0 と 1 の時の行き先
4: $C_0[]$、$C_1[]$//木の頂点でそれぞれ bit が 0 と 1 の時に進める探索位置の数
5: $length$//探索するキー長
6: $key[]$//探索するキー
7: **while** $searchptr < length$ **do**
8: $buffer = searchptr$
9: $searchptr = searchptr + C_{key[buffer]}[ptr]$
10: $ptr = P_{key[buffer]}[ptr]$
11: **end while**
12: ptr を仮バリューとして返す

う場合、GPU に木構造だけでなくキーの集合も保存する必要があるため、GPU のメモリ容量に余裕がない場合は、キーを GPU に保存せずに CPU 上で検証を行うことで、GPU のメモリ容量を節約できる。

5.5 GPU を用いたブロックチェーン検索の性能評価

5.5.1 評価環境

本評価では、本論文の提案手法を用いたブロックチェーン探索を、以下の 4 つの手法で比較した。

- GPU:提案手法で結果の検証を含む全ての処理を GPU で行う手法
- GPU+CPU:提案手法で結果の検証は CPU で行い、キーの検索のみを GPU で行う手法
- HASH:固定サイズのセットアソシアティブハッシュテーブルを用いた GPU を用いた既存手法。セットアソシアティブにおける連想度は 16 とする。
- CPU:GPU を用いず、連鎖法に基づくハッシュテーブルを用いて、CPU で検索処理を行う手法

評価で用いた計算機の CPU は Intel Xeon E5-2637v3 であり、メモリ容量は 256GB であり、CUDA のバージョンは 6.5 である。GPU の評価では、NVIDIA GeForce GTX 980 Ti を用いた。GPU の主な諸元は表 5.1 に示す。

表 5.1: NVIDIA GeForce GTX 980 Ti の主な諸元

性能項目	GeForce GTX 980 Ti
コア数	2,816
コアクロック	1,000MHz
メモリクロック	7,010MHz
メモリバンド幅	336.5GB/s
メモリ容量	6GB

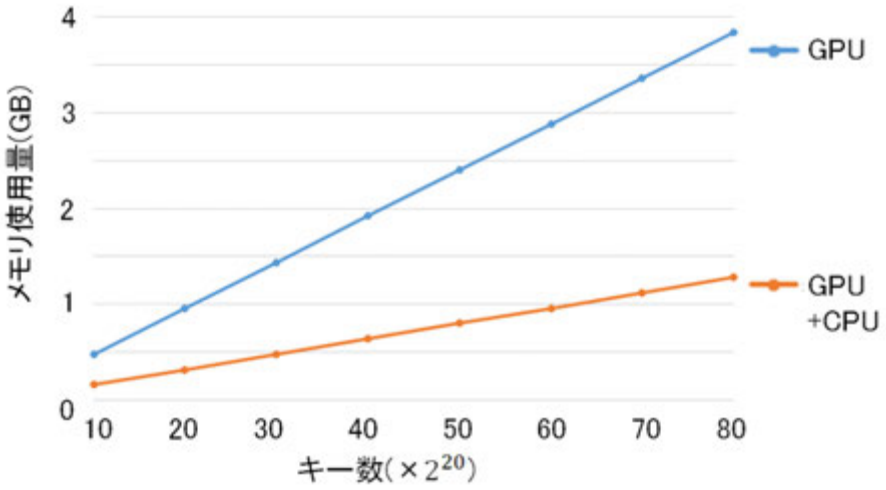


図 5.7: キーの数とデバイスメモリの使用量の関係

5.5.2　GPU 処理のデバイスメモリの使用量

5.4.3 節で述べたように、GPU で検索処理を行った後の結果の検証は CPU で行うか GPU で行うかによってメモリ使用量が異なる。図 5.7 にキー長 256bit でキー数を 10×2^{20} から 80×2^{20} まで変化させた時の GPU のデバイスメモリの使用量を表す。GPU が検証も含めて全て GPU 処理で行った場合で GPU+CPU が探索のみ GPU で行った場合のメモリ使用量を表している。図から、共にキーの数に比例してデバイスメモリの使用量が増加しているが、全ての場合で GPU+CPU の方がメモリ使用量が抑えられている。キーの数 1000 万の時には、GPU のメモリ使用量は、GPU+CPU の 3.0 倍である。実際のビットコインのブロックチェーンにおいては、全てのトランザクションの数は約 2 億 6 千万である。この時、全てのキーを一つの GeForce GTX 980 Ti に保存することはできないため、CPU+GPU の手法が有効である。

5.5.3　バッチサイズとスループットの関係

本提案手法では、GPU の各スレッド毎に 1 つの探索クエリを実行するため、GPU の並列度を活かすためには、多数のクエリをバッチ処理する必要がある。本節では、バッチサイズとスループットの関係を評価することで、必要なバッチサイズを明らかにする。

図 5.8 に探索するキー長は 256bit、キーの数は 80×2^{20} の時の探索のバッチサイズとスループットの関係を表す。スループットの単位 qps は query per sec を表す。HASH の手法においては、固定サイズのセットアソシアティブのハッシュテーブルを用いているため、キャッシュミスが発生し得り、キャッシュミス時の処理を CPU 処理として行う必要がある。しかし、ここでは HASH 手法の最も良い場合のスループットを見積り、本提案手法と比較するために、キャッシュミス時の処理時間はないものとした。図から、バッチサイズが大きくなるに従い、GPU の並列度が活かせるために 16×2^{10} まではスループットが向上することがわかる。GPU と GPU+CPU を比較すると、CPU 処理部分はバッチサイズの影響を受けないため、GPU のみの方がバッチサイズの増加による性能向上が大きい。HASH 手法においては、バッチサイズが大きい時にはスループットが低下している。これは、バッチサイズが大きく、バッチの数が少なくなるため、バッチ間の並列性を活かせなくなるためである。また、HASH 手法と CPU+GPU は、CPU+GPU において検証処理を CPU で行っているにも関わらず、ほぼ同じ性能となっている。

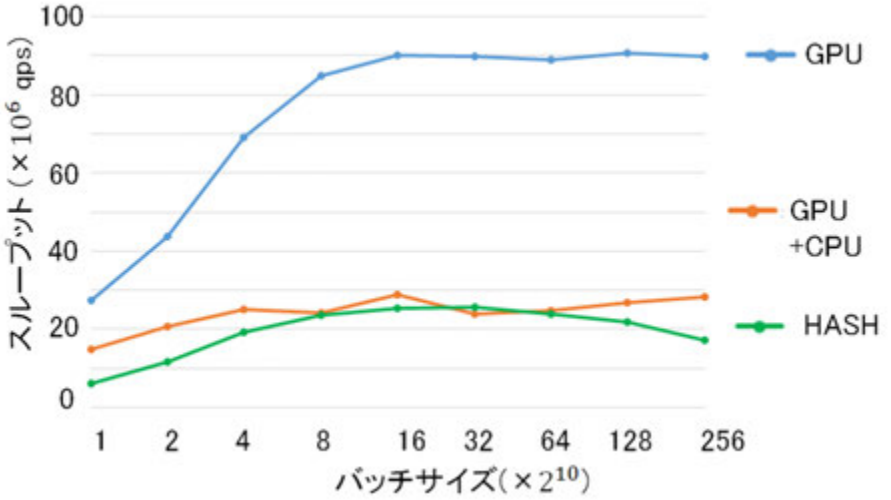


図 5.8: バッチサイズとスループットの関係

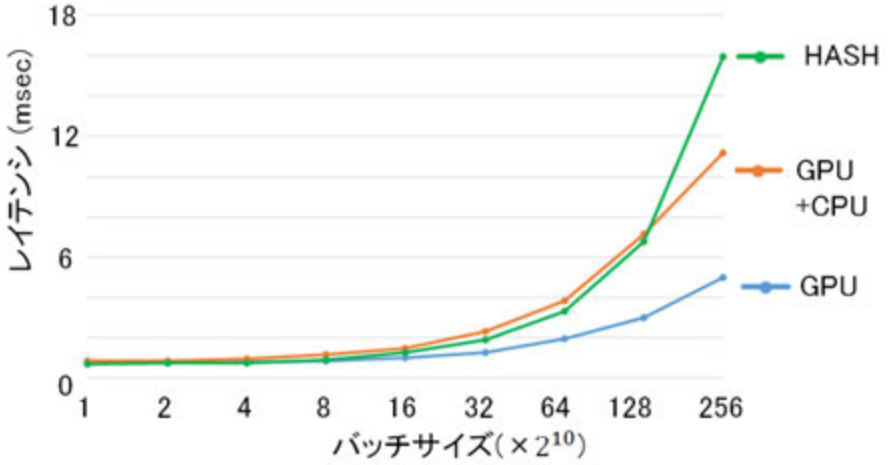


図 5.9: バッチサイズとレイテンシの関係

一般的に、スループットはバッチ数が大きくなると向上するが、一方でレイテンシが増加してしまう。図 5.9 にスループットの評価と同じ条件におけるバッチサイズとレイテンシの関係を示す。図から、GPU 手法では 16×2^{10} まではほぼレイテンシに変化がなく、それ以降ではレイテンシが増加している事が分かる。16×2^{10} と 256×2^{10} を比較すると 256×2^{10} のレイテンシは 16×2^{10} の 3.9 倍となる。しかし、両者のスループットは複数のバッチ間で計算とデータ転送の重複実行を行ったり、複数のバッチを並列実行することができるため、ほぼ一定となっている。GPU+CPU と HASH の手法においても、同様の傾向がみられる。

レイテンシとスループットの両方を考慮すると、今回の評価結果の中では、16×2^{10} が並列度を活かしつつ、レイテンシの増加が抑えられる適したバッチサイズであると言える。以降の評価では、これらの 3 つの手法の GPU 処理ではバッチサイズは 16×2^{10} とする。

5.5.4　キーの数とスループットの関係

図 5.10 に探索するキーの数を 10×2^{20} から 80×2^{20} に変化させた時のスループットを縦軸が対数の片対数図で示す。また、探索するキー長は 256bit である。提案手法では、キーの数に比例し

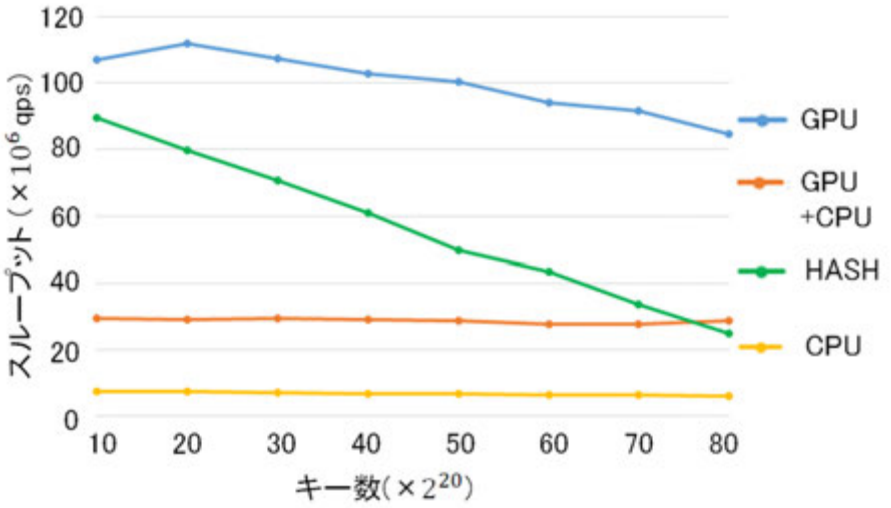


図 5.10: キーの数とスループットの関係 ($\leq 80 \times 2^{20}$)

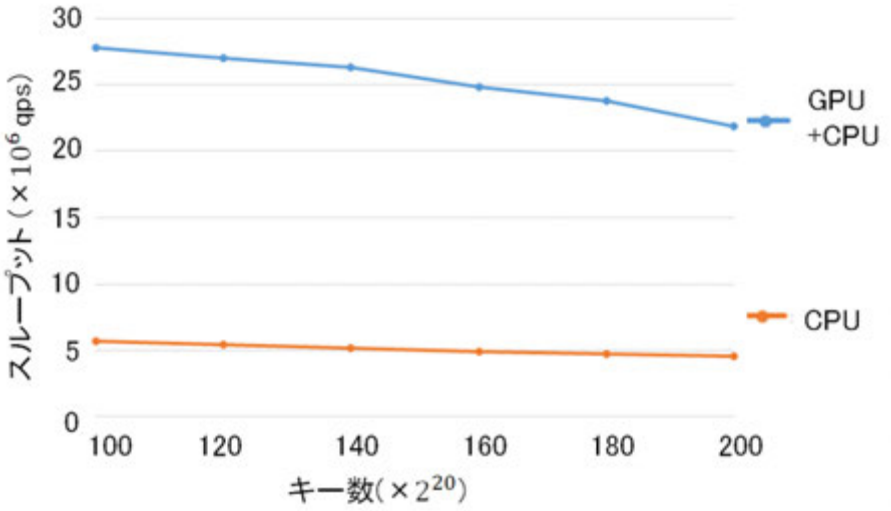


図 5.11: キー数とスループットの関係 ($\geq 100 \times 2^{20}$)

て頂点数が増え、頂点は 2 分探索で検索するためキーの数を n とした時、探索の計算量は $O(\log n)$ であり、キーの数が増加すると計算量が増加する。図 5.10 では、GPU 手法ではキー数が増えるにつれてスループットが低下している。しかし、GPU+CPU 手法においては、全ての場合で性能がほぼ一定である。これは、CPU+GPU 手法における CPU 処理である検証の時間がキーの数に関わらず一定であり、検証が全体にしめる割合が大きいためである。ハッシュを用いたキーの検索においては、キーの検索の計算量のオーダーは $O(1)$ である。しかし、図 5.10 では、キーの数が増えるにつれて HASH 手法のスループットが低下している。これは、ハッシュの衝突が原因となっている。固定サイズのセットアソシアティブハッシュテーブルにおいては、キーの数が増加するにつれて、衝突のためにキーの検証に必要な比較の数が最大で連想度と同じ 16 まで増加する。一方で、CPU 手法においては、全ての場合でほぼ同じスループットである。これは、CPU のメモリ容量が GPU よりも大きく、多くのメモリを用いることで衝突の確率が小さくなっているためである。それぞれの比較をまとめると、キーの数が 80×2^{20} の時、GPU 手法のスループットは HASH 手法の 3.4 倍、CPU 手法の 14.1 倍のスループットを達成している。

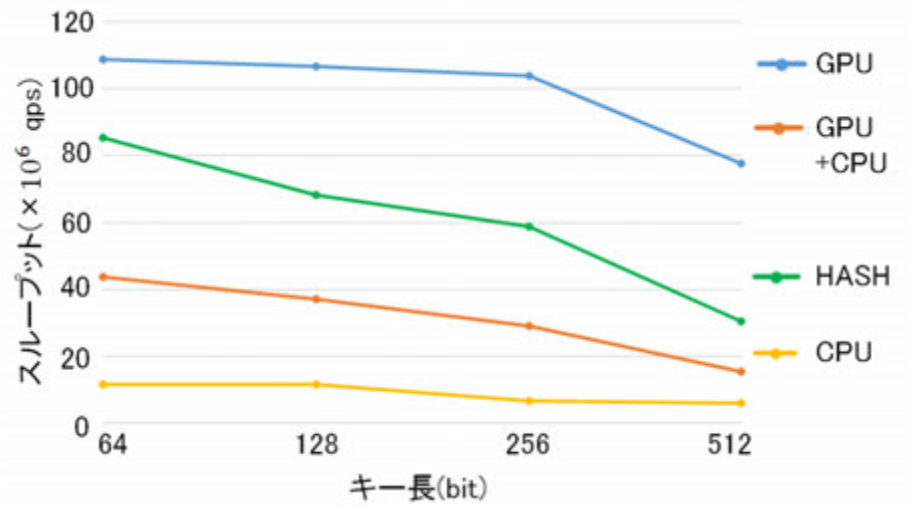


図 5.12: キー長とスループットの関係

GPU デバイスメモリの制限により、大量のキーを検索する場合、GPU+CPU 手法が GPU 手法における現実的な選択肢となる。この場合における評価として、ここでは、キー数が 100×2^{20} を越える場合の評価を行った。図 5.11 にキー数が 100×2^{20} 以上の場合のキー数とスループットの関係を示す。キー数が増加すると、GPU+CPU 手法のキー検索部分の計算量の増加に伴ってスループットが低下しているが、200×2^{20} においても CPU 手法の 4.8 倍のスループットを達成している。この結果は、GPU+CPU 手法が全てのキーを GPU に保存できない場合において良い拡張性を持つことを示している。

5.5.5　キー長とスループットの関係

図 5.12 に探索するキーの長さを 32bit から 512bit まで変化させた時のスループットを縦軸が対数の片対数図で示す。キーの数は 40×2^{20} である。

探索処理の計算量はキーの数のみによって決まるが、探索以外の処理である結果の検証とキーの転送の実行時間はキー長に比例する。そのため、全ての手法において、キーの長さに増加するにしたがってスループットが低下している。CPU 手法における性能の低下率は他の手法に比べて小さいが、これはキーの転送の必要がなく、探索以外の処理の割合が小さいためである。しかし、CPU 手法におけるスループットの低下の小ささを含めてもね GPU 手法はキー長がビットコインの ID の 2 倍となる 512bit の時でも CPU 手法の 13.1 倍のスループットを達成している。また、HASH 手法と比較すると、512bit の時に 2.6 倍のスループットを達成している。

第 6 章

分散キャッシュ手法のポリグロット永続化への応用

ここまで、本論文では、構造型ストレージを高速化する手法として、GPU 構造に適したキャッシュを作成し、それを複数の遠隔 GPU に拡張する手法を提案してきた。特に、計算量の大きく、GPU に適した処理である文字列探索やグラフ探索クエリが存在するドキュメント指向型とグラフ型についてそれぞれのキャッシュ手法を提案し、KVS 型においても実問題であるブロックチェーン検索への応用を示した。しかし、これらの手法はすべて単一の構造型ストレージに関する提案である。構造型ストレージは特定の用途に特化したシステムであり、汎用性を高めるために、複数の構造型ストレージを組み合わせたポリグロット永続化への応用が進んでいる。そのため、本論文で提案する構造型ストレージ向けのキャッシュにおいても、今後利用が拡大されると考えられるポリグロット永続化への拡張は不可欠である。ポリグロット永続化は複数の構造型ストレージの組み合わせであり、その組み合わせ方等に応じて非常に広い用途が想定される。そのため、ポリグロット永続化への応用を一般化して示すことは困難であり、本章では、構造型ストレージのポリグロット永続化への応用例を示し、そのシステムにおける各種構造型ストレージの利用方法および本論文で提案するキャッシュの適用方法を述べることで、本論文で提案するキャッシュがポリグロット永続化へ応用可能であることを示す。

6.1 応用例 1:ブロックチェーンを用いた文書管理

ブロックチェーンは 5 章で述べた通り、P2P ネットワークで管理された分散型の台帳であり、暗号通貨に主に用いられている。ブロックチェーンの特徴として、一度チェーンに取り込まれたブロックは、削除や更新は行われず、また、全てのブロックがハッシュ値を介して連結されているため、1 つの情報を変更ためにはそれ以降の全てのブロックを変更する必要があるという特徴がある。その特徴から、もしブロックの内容を改竄しようとした場合、それ以降の全てのブロックを改竄する必要があるため、改竄への耐性が高く、とりわけ多くの人々によって分散管理された公開型のブロックチェーンは改竄耐性が非常に高い [89][90][91]。その改竄耐性を活かして、様々な文書を確実に保存する手法として、ブロックチェーンを用いるプロジェクトが注目を集めている [92]。このプロジェクトでは、文書のハッシュ値を公開型ブロックチェーンであるビットコインにのブロックチェーンに書き込むことで、改竄を防止している。

そのような文書管理システムでは、ブロックチェーンのフルノードの運用と文書の管理を行う必要がある。フルノードの運用は、5 章で述べた通り KVS 型の構造型ストレージが用いられており、文書の管理にはドキュメント指向型の構造型ストレージが用いられる。図 6.1 において、そのようなシステムの概要を示す。KVS 型では、文書のハッシュ値をキー、文書本体がバリューとして保存され、ドキュメント指向型にも文書のハッシュ値が ID として文書が保存される。このシ

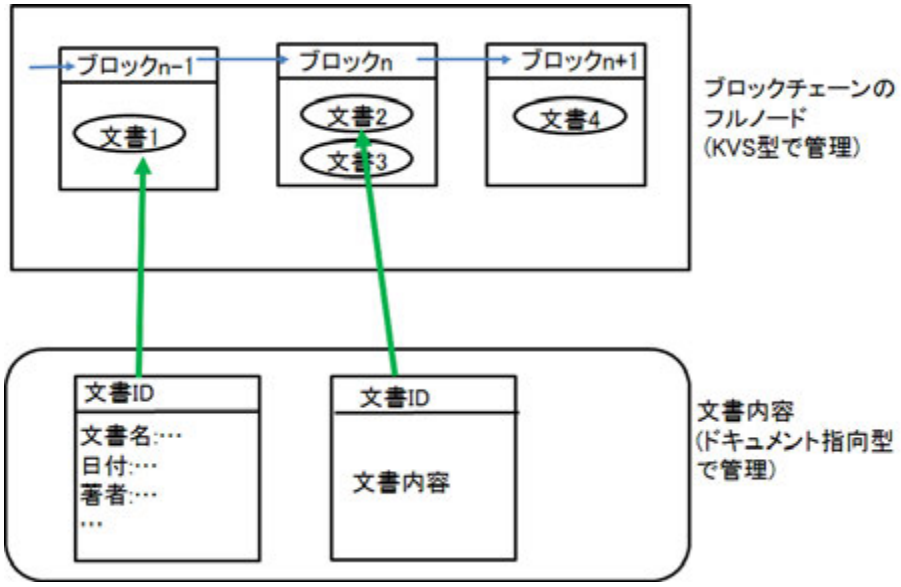


図 6.1: KVS 型とドキュメント指向型を用いた文書管理システム

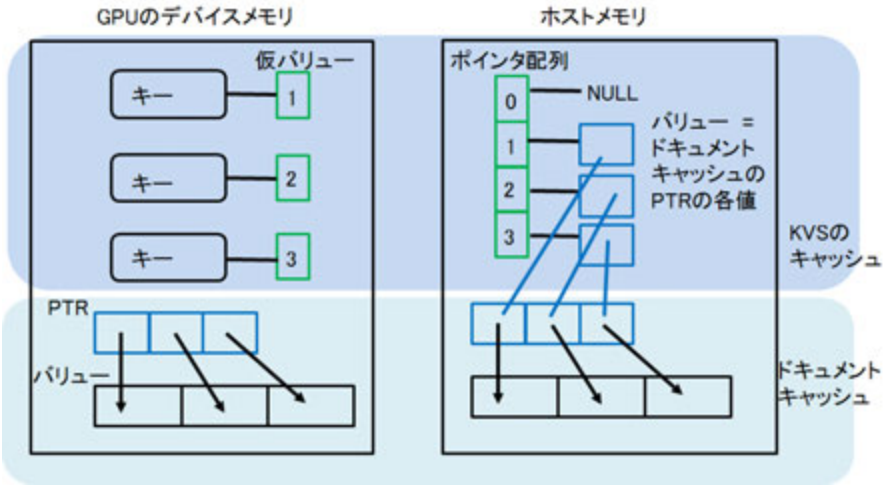


図 6.2: KVS 型とドキュメント指向型の混合キャッシュ

ステムにおいて、ID を基に文書を取得する場合は、KVS 型でキーとしてハッシュ値が保存されているため、KVS 型で高速に取得することができ、文書の内容を検索したい場合には、ドキュメント指向型を用いることで検索できる。

このようなシステムに本論文で提案するキャッシュを導入する場合、KVS 型とドキュメント指向型のキャッシュをシステムに持たせる必要がある。この際、KVS 型のキャッシュのバリューをドキュメントキャッシュのバリュー配列とし、仮バリューにバリュー配列へのポインタを持たせることで、KVS 型とドキュメント指向型の混合キャッシュを作る事ができる。図 6.2 に、KVS 型とドキュメント指向型の混合キャッシュの構造を示す。図の左側が GPU で保持されているキャッシュ、右側が CPU で保持されているキャッシュである。GPU 側では、KVS 型のキーの探索のためのキーと仮バリューおよびドキュメントキャッシュが保持されている。CPU 側では、KVS 型のバリューとドキュメントキャッシュが保持されている。この時の KVS 型のバリューはドキュメントキャッシュにおけるバリューと同じものである。そのため、KVS 型のバリューをドキュメントキャッシュの配列 PTR の各値とすることで、関節的にバリューを管理できる。これによって KVS

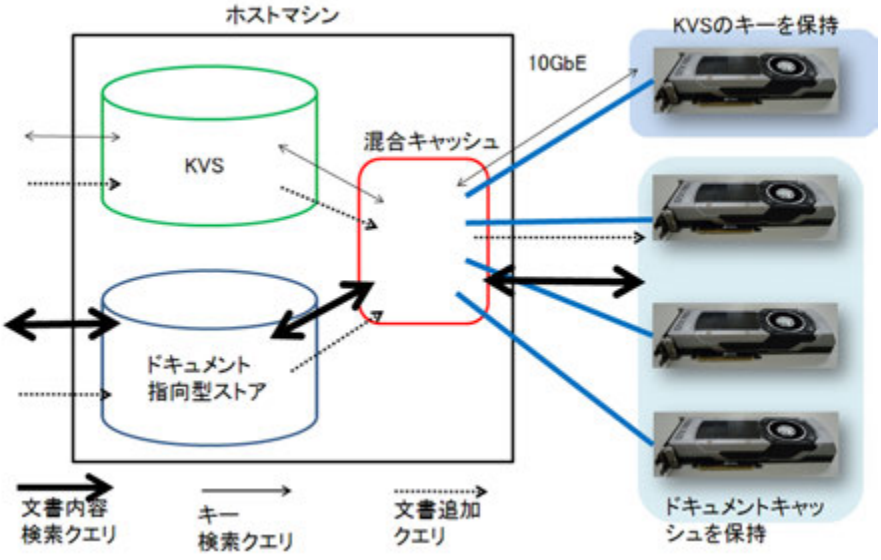


図 6.3: 文書管理システムに混合キャッシュを導入した場合の全体像とクエリの流れ

型とドキュメント指向型の双方を扱える混合キャッシュを作成できる。このキャッシュにおいて、キー (すなわち、文書の ID) を検索して文書を取得する場合には KVS 型のキャッシュを用いて検索し、バリュー (すなわち、文書本体) に対して条件を指定して一致する文書を取得する場合にはドキュメントキャッシュを用いて検索することでこの二種類の構造型ストレージのクエリを実行できる。

図 6.3 に、文書管理システムにキャッシュを導入した場合の全体像とクエリの流れを示す。構造型ストレージに保存されているデータのうち、キーはデータ量が大きく無いため、全てのキーをキャッシュする。バリューに相当する文書全体に関しては、全体をキャッシュし、GPU で処理するにはデータ量が大きいと考えられるため、頻繁に利用される一部のフィールドのみをキャッシュする。KVS とドキュメント指向型の計算量およびデータ量を考慮すると、共にドキュメント指向型の方が大きいため、図では、遠隔接続された GPU のうち、1 台を KVS 用、残りの多数の GPU をドキュメントキャッシュ用として利用すると仮定している。このシステムに対するクエリとそれに対する流れは以下の通りである。

- キー検索クエリは、KVS をキャッシュした GPU で検索され、仮バリューを用いてバリューを特定してバリューが利用者に返される。
- 文書内容検索クエリのうち、キャッシュされているものを対象とする場合は、複数の GPU で並列にバリュー内容の検索が行われ、検索結果が利用者に返される。
- 文書内容検索クエリのうち、キャッシュされていないものは、ドキュメント指向型ストアで検索を行い、結果が利用者に返される。
- 文書追加クエリでは、KVS 型とドキュメント型で書き込みクエリを行うと同時にそれぞれのキャッシュを更新する。

このように、KVS 型とドキュメント型を用いるポリグロット永続化の例であるブロックチェーンを用いた文書管理システムにおいて、本論文で提案するキャッシュを適用できる。

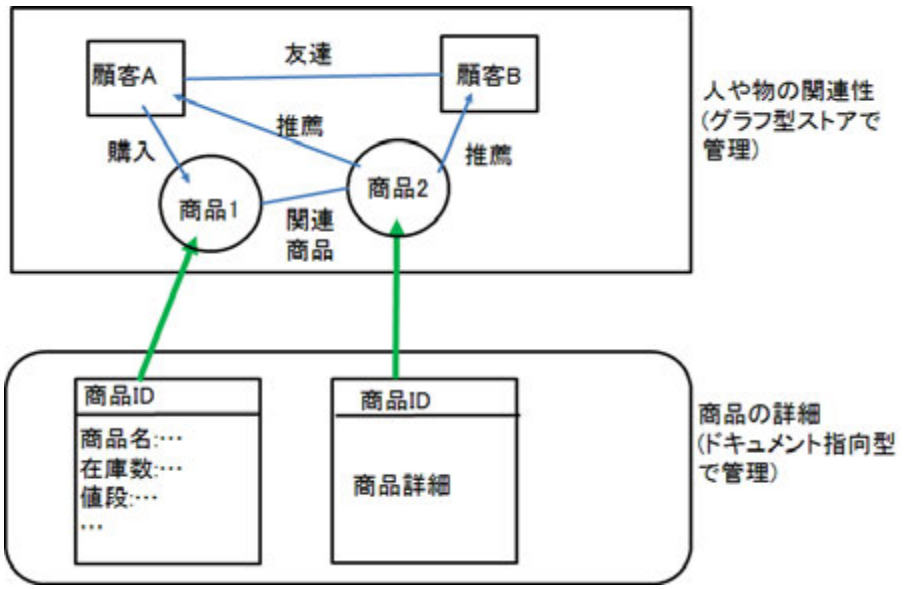


図 6.4: グラフ型とドキュメント指向型を用いた商品推薦システム

6.2 応用例 2:電子商取引における商品推薦システム

近年、電子商取引の規模は拡大しつづけており、我が国においても、平成 22 年には約 7 兆 8,000 億円だった市場規模が平成 28 年には約 15 兆 1,000 億円と大幅に拡大している [93]。電子商取引においては、実店舗のように実物の陳列によって顧客に訴えかけることができないため、いかに顧客が興味を持つ商品を推薦するかによって売上が左右される。その商品の推薦を行うシステムが商品推薦システムである。商品の推薦の手法として、顧客がこれまでに購入した履歴や人間関係などから、過去に購入した商品と関連性が高い商品や関連性が高い人が購入した商品を薦める手法が考えられる。そのような推薦に必要な人と人、人と物、物と物といった様々な関係性を管理するのには、グラフ型ストアが適している。また、顧客に商品の詳細を求められた場合にその内容を文書で提示する必要があるため、ドキュメント指向型ストアも同時に用いられる。よって、商品推薦システムにおいては、グラフ型ストアとドキュメント指向型ストアのポリグロット永続化が利用できると考えられる。図 6.4 において、そのようなシステムの概要を示す。グラフ型では、人や物等の関連性が頂点と辺として保存され、商品名等が属性として保存される。ドキュメント指向型では、商品の詳細や顧客情報等の文書が保存されている。このシステムにおいて、推薦のために関連性を調べたい場合にはグラフ型ストアへのクエリ、文書の内容を検索したり、提示したりする場合にはドキュメント指向型ストアへのクエリが実行される。

このようなシステムに本論文で提案するキャッシュを応用する場合、ドキュメント指向型とグラフ型のキャッシュであるドキュメントキャッシュとグラフキャッシュを持たせる必要がある。この際、商品名などのグラフ型の属性とドキュメント型のフィールドで共通してキャッシュされる部分に関しては、グラフキャッシュのうち、属性を指していたポインタをドキュメントキャッシュのバリュー配列を指すように変更することで、ドキュメント指向型とグラフ型の混合キャッシュを作ることができる。この作成方法は、ポインタの差し替えのみであり、前述した KVS 型とドキュメント指向型の混合キャッシュの作成方法とほぼ同じ手法であることから、本提案手法は様々な構造型ストレージを組み合わせるポリグロット永続化に簡単に応用でき、拡張性が高いと言える。図 6.5 にドキュメント指向型とグラフ型の混合キャッシュを示す。GPU 上には、グラフキャッシュのうちグラフ構造の部分とドキュメントキャッシュが保持されている。また、ホストメモリ上で、グラフキャッシュからドキュメントキャッシュへのポインタによってグラフキャッシュの属性が保

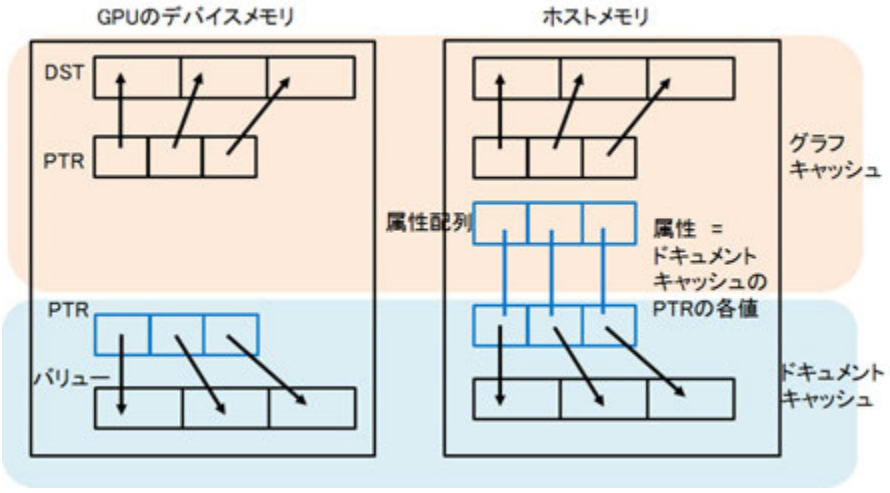


図 6.5: グラフ型とドキュメント指向型の混合キャッシュ

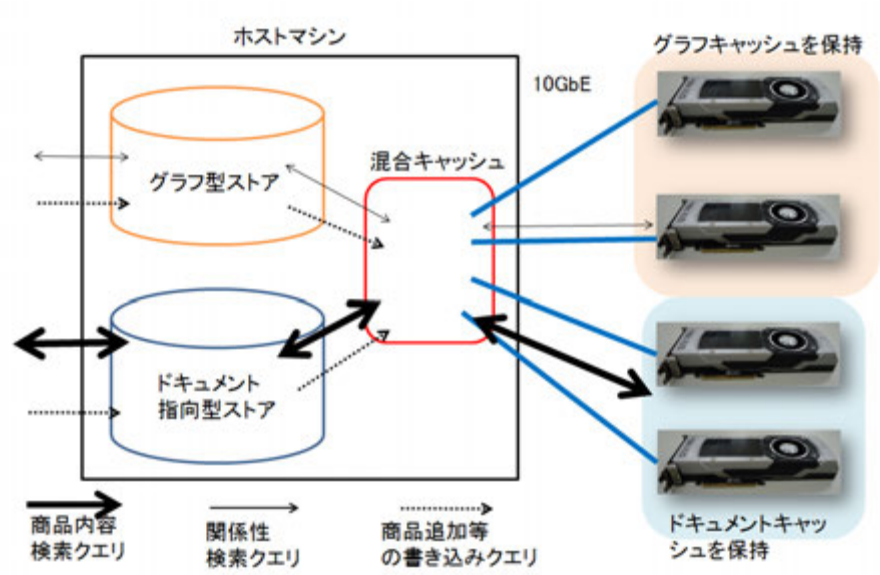


図 6.6: 商品推薦システムに混合キャッシュを導入した場合の全体像とクエリの流れ

持されている。この混合キャッシュにおいて、グラフ探索等のグラフ型ストアに対するクエリはグラフキャッシュ、文書の検索等のドキュメント指向型ストアに対するクエリはドキュメントキャッシュに対して検索を行うことで、それぞれのクエリをキャッシュを用いて実行できる。

図 6.6 に、混合キャッシュを含むシステムの全体像とクエリの流れを示す。構造型ストレージに保存されているデータのうち、グラフ構造のデータ量は小さいため、グラフ構造は全体をキャッシュする。文書管理システムの時と同じく、文書全体は GPU で処理するにはデータ量が大きいため、ドキュメントは頻繁に利用されるフィールドをキャッシュする。グラフ型とドキュメント指向型を比較すると、どちらの構造型ストレージも計算量の大きいクエリが存在するため、この例では、遠隔 GPU はそれぞれ 2 つずつ利用すると仮定している。この比率は、各構造型ストレージのクエリの比率などに応じて動的に変更可能である。このシステムに対するクエリの流れは以下の通りである。

- 関連性検索クエリは、横断を伴うクエリであるため、GPU 上のグラフキャッシュを用いて GPU で処理され、結果が返される。

- 商品内容クエリは、文書管理システムの文書検索と同様に、キャッシュされていればドキュメントキャッシュを介し、そうでなければドキュメント指向型ストアを介して処理される。
- 商品追加等の書き込みクエリは、それぞれの構造型ストレージと対応するキャッシュを更新する。

このように、グラフ型とドキュメント型を用いるポリグロット永続化の例である商品推薦システムにおいても、本提案手法を用いて GPU を用いた高速化が可能である。

本章では、ポリグロット永続化を用いる例として 2 つのシステム例を紹介したが、ポリグロット永続化の応用範囲はこれらのシステム以外にも多岐にわたり、様々なシステムのキャッシュとして本提案手法は応用可能である。

第 7 章

結論

近年、生成される情報量の増大や情報の多様化によって、汎用的な RDBMS に加えて、特定用途に特化しているが情報を高スループットで処理できる構造型ストレージおよび複数の構造型ストレージの組み合わせであるポリグロット永続化の利用が進んでいる。また、計算機においても、汎用的な CPU に加えて特定用途に特化した専用ハードウェアを利用することで、システム全体の性能を高める手法が広く用いられており、特定用途特化の流れはハードウェアとソフトウェアの両面で進んでいる。本論文では、この双方の流れを踏まえて、ハードウェアの一種である GPU を用いて、構造型ストレージに GPU 処理に特化したキャッシュを設けることで、構造型ストレージを高速化することを提案した。特に、構造型ストレージのクエリの中には、情報量に比例して計算量が大きくなる文字列の正規表現探索やグラフ探索などのクエリがあり、情報量が大きい場合のこれらのクエリに対する構造型ストレージのスループットは十分とは言えないため、キャッシュの構造はこれらのクエリに特化し、これらのクエリを高速化した。具体的には、ドキュメント指向型とグラフ型の構造型ストレージを中心に、そのままでは GPU 処理に向かない構造から探索に必要な情報のみを抽出し、配列構造のドキュメントキャッシュとグラフキャッシュを作成する手法をそれぞれ提案した。

また、GPU 処理においては、キャッシュのデータ量が GPU のデバイスメモリの容量を越えると性能が大幅に低下する問題がある。本論文では、10GbE で遠隔接続された複数の GPU へキャッシュを分散し、拡張する手法を提案し、デバイスメモリを越える大きさのキャッシュにも対応可能にした。遠隔接続を行う際には、CPU-GPU 間の転送帯域が制限されるが、ドキュメント指向型ではハッシュを用いた分散手法、グラフ型では部分的な非同期更新手法を提案し、この制限による転送オーバーヘッドを削減した。

評価では、本論文で提案した分散キャッシュを用いて遠隔 GPU3 台でクエリを処理する手法とオリジナルの構造型ストレージのスループットを比較した。その結果、ドキュメント指向型の評価では、文字列の正規表現探索クエリをドキュメント 1 件あたり 128 文字、1 千万件のドキュメントに対して実行し、提案手法はオリジナルのドキュメント指向型ストアに対して 75.0 倍のスループットを達成した。また、グラフ型の評価では、単一始点最短経路問題を頂点数 320 万、次数 200 のグラフに対して実行した際、提案手法はオリジナルのグラフ型ストアに対して 383.2 倍のスループットを達成し、本提案手法による GPU 処理に適したデータ構造と複数 GPU による並列処理で構造型ストレージを大幅に高速化できることを示した。

さらに、これらの構造型ストレージのキャッシュ手法が構造型ストレージを用いる様々な応用に利用できることを示すために、実問題への応用例を示すとともに、アプリケーションに特化させることで更なる高速化が図れることを示すために、KVS 型の構造型ストレージを用いるブロックチェーン探索への適用方法を示した。具体的には、ブロックチェーン探索では、ドキュメント指向型やグラフ型と同様な配列構造のキャッシュを基数木を用いて作成する手法を提案し、ブロックチェーン上の取引内容等を検索するのに必要な ID などのキーの検索を GPU で処理可能にした。

提案手法を用いることで、実際のブロックチェーンシステムであるビットコインの ID と同じキー長でキー数約 8,000 万件の時に CPU 処理の 14.1 倍のスループットを達成し、本論文のキャッシュ手法が構造型ストレージを用いる実問題へ応用できることを示した。

また、複数の構造型ストレージを組み合わせて利用するポリグロット永続化における応用例として、ブロックチェーンを用いた文書管理システムと電子商取引における商品推薦システムを例示し、それぞれのシステムにおける構造型ストレージの組み合わせ方を示した。その構造型ストレージの組み合わせに対して、本論文で提案した複数の構造型ストレージのキャッシュの混合キャッシュとしての利用方法を示し、本論文の手法がポリグロット永続化に対しても応用可能であることを示した。

本論文の貢献は、データ構造やデバイスメモリの大きさなど、GPU での高速化の際に生じる諸問題を上述した提案によって解決し、評価によって性能向上可能であることを示した点である。将来的に、ソフトウェアとハードウェア両面における特定用途特化の流れを踏まえて、構造型ストレージは、GPU や FPGA NIC など様々なハードウェアをボトルネックに合わせて高性能化がなされていくと考えられる。本論文はそのうちの GPU による高性能化の部分を担っており、その他のハードウェアにおける研究と組み合わせて利用されることで、将来的な構造型ストレージの発展に寄与する。

参考文献

[1] Tayler H. Hetherington, Mike O'Connor, and Tor M. Aamodt. MemcachedGPU: Scaling-up Scale-out Key-value Stores. In *Proceedings of the ACM Symposium on Cloud Computing*, pp. 43–57, August 2015.

[2] John A. Stankovic. Research Directions for the Internet of Things. *IEEE Internet of Things Journal*, Vol. 1, pp. 3–9, February 2014.

[3] The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things. `http://http://idcdocserv.com/1678`.

[4] Codd E. F. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, Vol. 13, pp. 377–387, June 1970.

[5] Danah m. Boyd and Nicole B. Ellison. Social Network Sites: Definition, History, and Scholarship. *Computer-Mediated Communication*, Vol. 13, pp. 210–230, October 2007.

[6] Tilmann Rabl, Mohammad Sadoghi, Hans-Arno Jacobsen, Sergio Gmez-Villamor, Victor Munts-Mulero, and Serge Mankowskii. Solving Big Data Challenges for Enterprise Application Performance Management. In *Proceedings of the VLDB Endowment*, pp. 1724–1735, August 2012.

[7] Chanankorn Jandaeng. Comparison of RDBMS and Document Oriented Database in Audit Log Analysis. In *Proceedings of the International Conference on Information Technology and Electrical Engineering*, pp. 332–336, October 2015.

[8] Vicknair Chad, Macias Michael, Zhao Zhendong, Nan Xiaofei, Chen Yixin, and Wilkins Dawn. A Comparison of a Graph Database and a Relational Database: A Data Provenance Perspective. In *Proceedings of the Southeast Regional Conference*, No. 42, April 2010.

[9] Oliveira Fábio Roberto and del Val Cura Luis. Performance Evaluation of NoSQL Multi-Model Data Stores in Polyglot Persistence Applications. In *Proceedings of the International Database Engineering & Applications Symposium*, pp. 230–235, July 2016.

[10] David A. Patterson and John L. Hennessy. *Computer Organization and Design.* Morgan Kaufmann, 2008.

[11] New technologies and architectures for efficient data center. `https://www.i-micronews.com/power-electronics-report/product/new-technologies-and-architectures-for-efficient-data-center.html`.

[12] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence.* Addison-Wesley, August 2013.

[13] Danga Interactive. *memcached - a distributed memory object caching system.* `http://memcached.org/`.

[14] Fitzpatrick Brad. Distributed caching with memcached. *Linux Journal*, Vol. 2004, No. 124, August 2004.

[15] Jure Petrovic. Using Memcached for Data Distribution in Industrial Environment. In *Proceedings of International Conference on Systems*, pp. 368–372, April 2008.

[16] Atikoglu Berk, Xu Yuehai, Frachtenberg Eitan, Jiang Song, and Paleczny Mike. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the joint international conference on Measurement and Modeling of Computer Systems*, pp. 53–64, June 2012.

[17] DeCandia Giuseppe, Hastorun Deniz, Jampani Madan, Kakulapati Gunavardhan, Lakshman Avinash, Pilchin Alex, Sivasubramanian Swaminathan, Vosshall Peter, and Vogels Werner. Dynamo: amazon's highly available key-value store. In *Proceedings of symposium on Operating systems principles*, pp. 205–220, October 2007.

[18] Apache Software Foundation. *The Apache HBase Project.* `http://hbase.apache.org/`.

[19] Apache Software Foundation. *The Apache Cassandra Project.* `http://cassandra.apache.org/`.

[20] Chang Fay, Dean Jeffrey, Ghemawat Sanjay, Hsieh Wilson C., Wallach Deborah A., Burrows Mike, Chandra Tushar, Fikes Andrew, and Gruber Robert E. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, Vol. 26, No. 4, June 2008.

[21] MongoDB. `http://www.mongodb.org`.

[22] Apache Couch DB. `http://couchdb.apache.org`.

[23] The MongoDB 3.0 Manual. `http://docs.mongodb.org/manual`.

[24] Ian Robinson, Jim Webbem, and Emil Eifrem. *Graph Databases.* Oreilly, 2013.

[25] Rania Soussi, Marie-Aude Aufaure, and Hajer Baazaoui. Towards Social Network Extraction Using a Graph Database. In *Proceedings of International Conference on Advances in Databases Knowledge and Data Applications*, pp. 11–16, April 2010.

[26] Cattuto Ciro, Quaggiotto Marco, Panisson André, and Averbuch Alex. Time-varying Social Networks in a Graph Database: A Neo4J Use Case. In *Proceedings of International Workshop on Graph Data Management Experiences and Systems*, No. 11, June 2013.

[27] Huang Zan, Chung Wingyan, Ong Thian-Huat, and Chen Hsinchun. A Graph-based Recommender System for Digital Library. In *Proceedings of the 2Nd ACM/IEEE-CS Joint Conference on Digital Libraries*, pp. 65–73, July 2002.

[28] Yubin Park, Mallikarjun Shankar, Byung-Hoon Park, and Joydeep Ghosh. Graph Databases for Large-Scale Healthcare Systems: A Framework for Efficient Data Management and Data Services. In *Proceedings of International Conference on Data Engineering Workshops*, pp. 12–19, March 2014.

[29] Mark Graves, Ellen R. Bergeman, and Charles B. Lawrence. Graph Database Systems for Genomics. *IEEE Engineering in Medicine and Biology special issue on Managing Data for the Human Genome Project*, No. 11, January 1995.

[30] Justin J. Miller. Graph Database Applications and Concepts with Neo4j. In *Proceedings of the Southern Association for Information Systems*, No. 24, March 2013.

[31] Neo4j. `https://neo4j.com/`.

[32] InfiniteGraph. `www.objectivity.com/products/infinitegraph/`.

[33] AllegroGraph. `https://franz.com/agraph/allegrograph/`.

[34] Vojtěch Kolomičenko, Martin Svoboda, and Irena Holubová Mlýnková. Experimental comparison of graph databases. In *Proc. of International Conference on Information Integration and Web-based Applications & Services*, pp. 115–124, 2013.

[35] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, July 2009.

[36] Alex Averbuch and Martin Neumann. Partitioning Graph Databases - A Quantitative Evaluation. *CoRR*, Vol. abs/1301.5121, pp. 1–92, 2013.

[37] Apache Titan. `http://titan.thinkaurelius.com/`.

[38] Reid Andersen and Yuval Peres. Finding Sparse Cuts Locally Using Evolving Sets. In *Proceedings of the Annual ACM Symposium on Theory of Computing*, pp. 235–244, 2009.

[39] Robert Görke, Tanja Hartmann, and Dorothea Wagner. Dynamic graph clustering using minimum-cut trees. In *Argorithms and Data Structures*, pp. 339–350, 2009.

[40] Joachim Gehweiler and Henning Meyerhenke. A Distributed Diffusive Heuristic for Clustering a Virtual P2P Supercomputer. In *Proceedings of International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pp. 1–8, April 2010.

[41] Hector Ortega-Arranz, Yuri Torres, Diego R. Llanos, and Arturo Gonzalez-Escribano. A New GPU-based Approach to the Shortest Path Problem. In *Proc. of International Conference on High Performance Computing and Simulation*, pp. 505–511, July 2013.

[42] Duane Merill, Michael Garland, and Andrew Grimshaw. Scalable GPU Graph Traversal. In *Proc. of International Symposium on Principles and Practice of Parallel Programming*, pp. 117–128, August 2012.

[43] Sadegh Nobari, Thanh-Tung Cao, Stéphane Bressan, and Panagiotis Karras. Scalable Parallel Minimum Spanning Forest Computation. In *Proc. of International Symposium on Principles and Practice of Parallel Programming*, pp. 205–214, August 2012.

[44] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. Parallelization and Characterization of Pattern Matching using GPUs. In *Proceedings of the International Symposium on Workload Characterization (IISWC'11)*, pp. 216–225, November 2011.

[45] Yuan Zu, Ming Yang, Zhonghu Xu, Lin Wang, Xin Tian, Kunyang Peng, and Qunfeng Dong. GPU-based NFA Implementation for Memory Efficient High Speed Regular Expression Matching. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*, pp. 129–140, February 2012.

[46] Jamin Naghmouchi, Daniele Paolo Scarpazza, and Miaden BereKovic. Small-ruleset Regular Expression Matching on GPGPUs: Quantitative Performance Analysis and Optimization. In *Proceedings of the International Conference on Supercomputing (ICS'10)*, pp. 337–348, June 2010.

[47] Chee-Yong Chan, Minos Garofalakis, and Rajeev Rastogi. RE-tree: an efficient index structure for regular expressions. *The International Journal on Very Large Data Bases*, pp. 102–119, August 2003.

[48] Junghoo Cho and Sridhar Rajagopalan. A Fast Regular Expression Indexing Engine. In *Proceedings of the International Conference on Data Engineering*, pp. 1–12, February 2002.

[49] Malewicz Grzegorz, Austern Matthew, Bik Aart, Dehnert James, Horn Ilan, Leiser Naty, and Czajkowski Grzegorz. Pregel: a system for large-scale graph processing. In *International Conference on Management of data*, pp. 135–146, June 2010.

[50] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. GraphLab: a new framework for parallel machine learning. In *Conference on Uncertainty in Artificial Intelligence*, pp. 340–349, July 2010.

[51] Joseph Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Power-Graph: Distributed Graph-Parallel Computation on Natural Graphs . In *USENIX Symposium on Operating Systems Design and Implementation*, pp. 17–30, October 2012.

[52] Jianlong Zhong and Bingsheng He. Medusa: Simplified Graph Processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 25, pp. 1543–1552, April 2013.

[53] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A High-Performance Graph Processing Library on the GPU. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 11:1 – 11:12, March 2016.

[54] Jose Duato, Antonio Pena, Federico Silla, Rafael Mayo, and Enrique Quintana-Orti. rCUDA: Reducing the Number of GPU-based Accelerators in High Performance Clusters. In *Proc. of the International Conference on High Performance Computing and Simulation (HPCS'10)*, pp. 224–231, June 2010.

[55] A. Barak and A. Shiloh. The virtualcl (vcl) cluster platform. `http://www.mosix.cs.huji.ac.il/vcl/VCL_wp.pdf`.

[56] Jun Suzuki and Yoichi Hidaka and Junichi Higuchi and Takashi Yoshikawa and Atsushi Iwata. ExpressEther — Ethernet-Based Virtualization Technology for Reconfigurable Hardware Platform . In *Proceedings of the IEEE Symposium on High-Performance Interconnects*, October 2006.

[57] Jun Suzuki and Teruyuki Baba and Yoichi Hidaka and Junichi Higuchi and Nobuharu Kami and Satoshi Uchida and Masahiko Takahashi and Tomoyoshi Sugawara and Takashi Yoshikawa. Adaptive Memory System over Ethernet. In *Proceedings of the Workshop on Hot Topics in Storage and File Systems*, June 2010.

[58] Jun Suzuki, Yuki Hayashi, Masaki Kan, Shinya Miyakawa, and Takashi Yoshikawa. End-to-End Adaptive Packet Aggregation for High-Throughput I/O Bus Network Using Ethernet. In *Proc. of International Symposium on High-Performance Interconnects*, pp. 17–24, August 2014.

[59] Jun Suzuki, Yoichi Hidaka, Junichi Higuchi, Yuki Hayashi, Masaki Kan, and Takashi Yoshikawa. Disaggregation and Sharing of I/O Devices in Cloud Data Centers. *IEEE Transactions on Computers*, Vol. 66, No. 10, pp. 3013–3026, October 2016.

[60] Carlos Reaño, Rafael Mayo, Enrique S. Quintana-Ortí, Federico Silla, José Duato, and Antonio J. Peña. Influence of InfiniBand FDR on the Performance of Remote GPU Virtualization. In *Proceedings of the International Conference on Cluster Computing (CLUSTER'13)*, pp. 1–8, September 2013.

[61] Intel Rack Scale Architecture: Faster Service Delivery and Lower TCO. `http://www.intel.com/content/www/us/en/architecture-and-technology/intel-rack-scale-architecture.html`.

[62] Sergey Legtchenko, Nicholas Chen, Daniel Cletheroe, Antony Rowstron, Hugh Williams, and Xiaohan Zhao. XFabric: A Reconfigurable In-Rack Network for Rack-Scale Computers. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)*, pp. 15–29, Mar 2016.

[63] Shin Morishima and Masahiro Okazaki and Hiroki Matsutani. A Case for Remote GPUs over 10GbE Network for VR Applications. In *Proceedings of the International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, June 2017.

[64] NVIDIA CUDA. `https://developer.nvidia.com/cuda-zone`.

[65] Wikipedia. `https://www.wikipedia.org/`.

[66] Mao Yandong, Kohler Eddie, and Morris Robert Tappan. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pp. 183–196, April 2012.

[67] Johan Ugander, Brian Karrer, Lars BackStrom, and Cameron Marlow. The Anatomy of the Facebook Social Graph. In *Arxiv preprint arXiv:1111.4503*, November 2011.

[68] The Neo4j Manual. `http://neo4j.com/docs/stable`.

[69] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[70] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. `https://www.bitcoin.com/bitcoin.pdf`.

[71] Counterparty. https://counterparty.io/.

[72] Amy Nordrum. Wall Street Occupies the Blockchain - Financial Firms Plan to Move Trillions in Assets to Blockchains in 2018. *IEEE Spectrum*, pp. 40–45, September 2017.

[73] Ethereum Project. https://www.ethereum.org/.

[74] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making Smart Contracts Smarter. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pp. 254–269, October 2016.

[75] Global Bitcoin Nodes Distribution. https://bitnodes.21.co/.

[76] Coinbase. https://www.coinbase.com.

[77] Dinh Tien Tuan Anh, Wang Ji, Chen Gang, Liu Rui, Ooi Beng Chin, and Tan Kian-Lee. BLOCKBENCH: A Framework for Analyzing Private Blockchains. In *Proceedings of the International Conference on Management of Data*, pp. 1085–1100, May 2017.

[78] Peercoin. https://peercoin.net/.

[79] Hyperledger. https://www.hyperledger.org/.

[80] Wenting Li, Alessandro Sforzin, Sergey Fedorov, and Ghassan O. Karame. Towards Scalable and Private Industrial Blockchains. In *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*, pp. 9–14, April 2017.

[81] Tayler H. Hetherington, Timothy G. Rogers, Lisa Hsu, Mike O'Connor, and Tor M. Aamodt. Characterizing and Evaluating a Key-value Store Application on Heterogegenenous CPU-GPU Systems. In *Proceedings of the International Symposium on Performance Analysis of System and Software*, pp. 88–98, April 2012.

[82] Xiaohuang Huang, Christopher I. Rodrigues, Stephen Jones, Ian Buck, and Wen mei Hwu. XMalloc: A Scalable Lock-free Dynamic Memory Allocator for Many-core Machines. In *Proceedings of the IEEE International Conference on Computer and Information Technology*, pp. 1134–1139, June 2010.

[83] Developing a Linux Kernel Module using GPUDirect RDMA. http://docs.nvidia.com/cuda/gpudirect-rdma/index.html.

[84] Chalamalasetti Sai Rahul, Lim Kevin, Wright Mitch, AuYoung Alvin, Ranganathan Parthasarathy, and Margala Martin. An FPGA Memcached Applianc. In *Proceedings of the International Symposium on Field Programmable Gate Arrays*, pp. 245–254, February 2013.

[85] Michaela Blott, Kimon Karras, Ling Liu, Kees Vissers, Jeremia Bär, and Zsolt István. Achieving 10Gbps Line-rate Key-value Stores with FPGAs. In *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing*, June 2013.

[86] Yuma Sakakibara, Kohei Nakamura, and Hiroki Matsutani. An FPGA NIC Based Hardware Caching for Blockchain. In *Proceedings of the International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, Jun 2017.

[87] Bitcoin Core. https://bitcoin.org.

[88] LevelDB. http://leveldb.org/.

[89] Yonatan Sompolinsky and Aviv Zohar. Secure High-Rate Transaction Processing in Bitcoin. In *Proceedings of Financial Cryptography and Data Security*, pp. 507–527, June 2015.

[90] Ittay Eyal and Emin Gün Sirer. Majority is not Enough: Bitcoin Mining is Vulnerable. In *Proceedings of Financial Cryptography and Data Security*, pp. 436–454, March 2014.

[91] Gervais Arthur and Karame Ghassan O. and Wüst Karl and Glykantzis, Vasileios and Ritzdorf Hubert and Capkun Srdjan. On the Security and Performance of Proof of Work Blockchains. In *Proceedings of the Conference on Computer and Communications Security*, pp. 3–16, October 2016.

[92] Factom. https://www.factom.com/.

[93] 経済産業省 商務情報政策局情報経済課. 平成 28 年度我が国におけるデータ駆動型社会に係る基盤整備（電子商取引に関する市場調査）. http://www.meti.go.jp/press/2017/04/20170424001/20170424001-2.pdf.

関連著作等

定期刊行誌掲載論文　(主論文に関連する原著論文)

[1] Shin Morishima, Hiroki Matsutani, "High-Performance with an In-GPU Graph Database Cache", IEEE IT Professional, Special Issue on Graph Databases and Their Applications, Vol.19, No.6, pp.58-64, Nov/Dec 2017.

[2] 森島 信, 松谷 宏紀, "GPU を用いたドキュメント指向型データベースの高速化", 電子情報通信学会論文誌 情報・システム, Vol.J100-D, No.12, pp.949-963, Dec 2017.

[3] 森島 信, 松谷 宏紀, "GPU を用いたグラフ型データベースの高速化および拡張性の改善", 電子情報通信学会論文誌 情報・システム, VolJ98-D, Vol.J98-D, No.12, pp.1436-1450, Dec 2015.

[4] Shin Morishima, Hiroki Matsutani, "Performance Evaluations of Graph Database using CUDA and OpenMP-Compatible Libraries", ACM SIGARCH Computer Architecture News (CAN), Vol.42, No.4, pp.75-80, Sep 2014.

国際会議論文 (関連論文)

[5] Shin Morishima, Hiroki Matsutani, "Accelerating Blockchain Search of Full Nodes Using GPUs", Proc. of the 26th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'18), Mar 2018.

[6] Shin Morishima, Hiroki Matsutani, "Distributed In-GPU Data Cache for Document-Oriented Data Store via PCIe over 10Gbit Ethernet", Proc. of the 22nd International European Conference on Parallel and Distributed Computing (Euro-Par'16) Workshops (HeteroPar'16), pp.41-55, Aug 2016.

[7] Shin Morishima, Hiroki Matsutani, "Performance Evaluations of Document-Oriented Databases using GPU and Cache Structure", Proc. of the 13th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA'15), pp.108-115, Aug 2015.

その他の国際会議発表

[8] Shin Morishima, Masahiro Okazaki, Hiroki Matsutani, "A Case for Remote GPUs over 10GbE Network for VR Applications", Proc. of the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART'17), Jun 2017.

[9] Yasuhiro Ohno, Shin Morishima, Hiroki Matsutani, "Accelerating Spark RDD Operations with Local and Remote GPU Devices", Proc. of the 22nd IEEE International Conference on Parallel and Distributed Systems (ICPADS'16), pp.791-799, Dec 2016.

[10] Shin Morishima, Hiroki Matsutani, "A GPU-Based Acceleration Method for Document-Oriented Databases", The 6th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART'15), Poster session, Jun 2015.

[11] Shin Morishima, Hiroki Matsutani, "Performance Evaluations of Graph Database using CUDA and OpenMP-Compatible Libraries", The 5th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART'14), Jun 2014.

国内研究会

[12] 森島 信, 松谷 宏紀, "GPU を用いたブロックチェーン検索の高速化", 電子情報通信学会技術研究報告 CPSY2017-47 (DesignGaia'17), Vol.117, No.278, pp.63-68, Nov 2017.

[13] 榊原 優真, 三塚 皐矢, 森島 信, 松谷 宏紀, "FPGA NIC を用いたブロックチェーン向けデータベースのキャッシング", 電子情報通信学会技術研究報告 CPSY2017-31 (SWoPP'17), Vol.117, No.153, pp.165-170, Jul 2017.

[14] 森島 信, 岡崎 真博, 松谷 宏紀, "VR アプリケーションのためのリモート GPU 割り当ての検討", 電子情報通信学会技術研究報告 CPSY2016-118, Vol.116, No.416, pp.85-90, Jan 2017.

[15] 竹本 一馬, 林 愛美, 森島 信, 松谷 宏紀, "GPU の計算結果を集約する 10GbE FPGA スイッチの検討", 電子情報通信学会技術研究報告 CPSY2016-113, Vol.116, No.416, pp.43-48, Jan 2017.

[16] 大野 泰弘, 森島 信, 松谷 宏紀, "ローカルおよびリモート GPU を用いた Spark RDD 操作の高速化", 情報処理学会研究報告 2017-DBS-164, No.6, Jan 2017.

[17] 森島 信, 松谷 宏紀, "リモート GPU を用いたグラフ処理における GPU 間同期手法の検討", 電子情報通信学会技術研究報告 CPSY2016-56 (DesignGaia'16), Vol.116, No.336, pp.53-58, Nov 2016.

[18] 原 弘明, 森島 信, 鯉渕 道紘, 天野 英晴, 松谷 宏紀, "ラック間をまたぐリモート GPU および SSD 間通信への光無線割り当ての評価", 電子情報通信学会技術研究報告 CPSY2016-18 (SWoPP'16), Vol.116, No.177, pp.89-94, Aug 2016.

[19] 大野 泰弘, 森島 信, 松谷 宏紀, "GPU を用いた Spark のリダクション及びトランスフォーメーションの性能評価", 電子情報通信学会技術研究報告 CPSY2015-113, Vol.115, No.399, pp.25-30, Jan 2016.

[20] 森島 信, 松谷 宏紀, "リモート GPU クラスタを用いたドキュメント指向型データベースの性能評価", 電子情報通信学会技術研究報告 CPSY2015-61 (DesignGaia'15), Vol.115, No.342, pp.1-6, Dec 2015.

[21] 森島 信, 松谷 宏紀, "GPU を用いたドキュメント指向型データベースの高速化", 電子情報通信学会技術研究報告 CPSY2014-122, Vol.114, No.427, pp.1-6, Jan 2015.

[22] 森島 信, 松谷 宏紀, “GPU を用いた分割グラフ型データベースの高速化”, 電子情報通信学会技術研究報告 CPSY2014-38 (SWoPP’14), Vol.114, No.155, pp.167-172, Jul 2014.

[23] 森島 信, 松谷 宏紀, “マルチコアおよび GPU を用いたグラフ型データベースの性能評価”, 電子情報通信学会技術研究報告 CPSY2013-92, Vol.113, No.417, pp.113-118, Jan 2014.

受賞

[24] 森島 信, “電子情報通信学会 コンピュータシステム研究会 優秀若手講演賞 (2014)”.