グラフモデルを基盤とする異種データベース統合システムの開発と評価

若林 拓† 常 穹† 宮崎 純†

† 東京科学大学情報理工学院情報工学系 〒 152–8550 東京都目黒区大岡山 2 丁目 12 − 1
E-mail: †wakabayashi.t.d410@m.isct.ac.jp, ††{q.chang,miyazaki}@comp.isct.ac.jp

あらまし データの多様化に伴い、特性の異なる複数のデータベース（DB）を併用する機会が増えている一方で、それぞれが扱うクエリ言語やデータ型が異なるため、管理・操作の負荷増大が大きな課題となっている。本研究ではこの問題に対し、グラフデータベース、RDB、および主要な 3 種の NoSQL データベースを統合し、これらの一元的な操作を可能にするクエリ実行基盤を提案する。提案手法では、入力されたクエリを特定の言語に依存しない抽象化した演算子へと変換し、各 DB のネイティブクエリと結びつけるアプローチを採用した。これにより、個別の DB 実装に縛られることなく、異種 DB を跨ぐ複雑な連携を柔軟に組み立てることが可能となる。また、実行エンジン内に中間結果を保持・利用する仕組みを構築したことで、システム間の冗長なデータ移動を抑制し、複数の DB が関与する動的なクエリに対しても一貫した制御手順での実行を実現している。さらに、本システムは DB 間でデータを移行する機能を備えており、蓄積されたデータの特性や要件の変化に応じた再配置が可能である。本研究では、これら一連の機能を備えた実行基盤の実装を通じて、異種データベースが混在する分散環境においてもクエリが正しく動作する実行環境を構築し、クエリ実行順序の制御やデータ配置の変更を柔軟に行うことを可能とした。

キーワード polystore system,NoSQL データベース, プロパティグラフ, データ統合技術

1 はじめに

情報社会の進展に伴い、テキストや時系列、グラフデータといった多種多様なデータが混在し、その規模も増大し続けている。取り扱うデータの性質や要件に応じて、構造化データを管理する RDB、製品カタログ等の文書データに適したドキュメント DB、友人関係等のつながりを表現するグラフ DB など、最適なデータモデルは異なる。

特にグラフデータベースはエンティティ間の複雑な関係性をノードとエッジで柔軟に表現できる一方、特有の課題も存在する。例えば、膨大な属性データがメモリを占有することで、グラフ探索に必要なメモリ展開領域が阻害されるという問題がある。また、グラフデータベースは構造中心のインデックス設計を採用しているため、大規模な属性フィルタリング処理には適していない。こうした単一データベースによる対応の限界を克服するため、複数のデータベースを併用し、データの性質に応じた適材適所の配置を実現する Polystore System が必要とされている。

しかし、Polystore System の運用においても問い合わせ最適化の困難さという課題が残されている。各データベースごとに性能特性が異なるため、単一 DB 向けの最適化手法が通用しない。特にグラフデータにおけるパス検索は計算の前後関係が構造的に固定されるため、他の DB とは異なる最適化アプローチが求められる。

これらの課題に対し、本研究ではグラフデータベースを主軸とし、肥大化した属性情報を他の異種データベースへ分散配置するという先行研究のアプローチを継承する。具体的には、関係探索に特化したグラフ構造のみをグラフデータベースで管理し、それに付随する膨大な属性情報を KVS や RDB 等へ切り出して保持させることで、グラフ探索におけるメモリ効率の向上を図るものである。

本研究ではこの構成をさらに発展させ、複雑な構造条件を含むユーザークエリを特定の言語に依存しない共通形式へと変換する実行基盤を導入した。この共通形式を介することで、異種データベース間を跨ぐ一連の処理順序を論理レベルで統合的に制御することが可能となる。これにより、将来的な実行計画の最適化を容易にする土台を築くとともに、異種分散環境における効率的なクエリ実行を可能とする管理基盤を実現する。

2 関連研究

本節では提案手法においてデータモデルとして活用する U-Schema と先行研究において提案された polystore system をいくつか紹介する。

2.1 U-Schema

U-Schema [1] は RDB と 4 つの主要な NoSQL データベースであるグラフ DB、ドキュメント DB、カラム指向 DB、キーバリューストアに対する統一的なスキーマモデルである。これは Entity-Relation モデルに基づいて構成されており、データオブジェクトをエンティティタイプ、エンティティの相互関係をリレーションタイプとして定義している。またエンティティの種類を示すデータラベルが同じだが、異なるプロパティを持つような状態 structural variation として記述する。これにより NoSQL データベースが持つスキーマレスな性質が反映されるようになっている。

2.2 CloudMdsQL

CloudMdsQL [2] は SQL をベースとした拡張クエリ言語を提供するクエリエンジンであり、RDB、ドキュメント DB、グラフ DB に対応している。SQL クエリの中にアクセス対象の DB のネイティブ関数をサブクエリとして直接埋め込むことで、統合的なアクセスを実現している。

しかしデータの保存先をクエリ内で明示する必要があり、データ配置の透過性は低い。またデータマイグレーションも導入されていない。

2.3 Léa らの手法

Léa ら [3] は RDB、ドキュメント DB、グラフ DB に対応している polystore system を提案している。このシステムでは、Entity-Relationship モデルに基づいて全体のデータモデルが構築されており、これを用いてシステム側が各データベースにおけるデータ配置を統一的に把握できるようになっている。これによりユーザがデータ配置を意識することのない透過的なクエリを実現している。また、対応する 3 種類のデータベースのネイティブクエリを受け付ける設計となっており、受け取ったクエリは問い合わせ木に変換される。この問い合わせ木を用いることで、実行順序の入れ替えによるクエリ処理最適化ができるようになっている。

一方で、このシステムにはデータマイグレーション機能が組み込まれておらず、異種データベース間でのデータ配置の最適化は考慮されていないという課題が残されている。

2.4 山下らの手法

山下らが提案したシステム [4] はグラフ DB とキーバリューストアに対応しているシステムであり、データモデル全体をグラフ構造として捉えるようにしている。

本手法では U-Schema [1] を利用してデータ配置を把握し、これに基づいてマッピング辞書というファイルを作成している。マッピング辞書は JSON 形式で記述され、各エンティティが持つプロパティのデータ配置についての情報を保持している。システムはクエリ実行時にこのマッピング辞書を参照することでアクセス対象のデータの位置を特定できる。これにより、分散されたデータベース間においても透過的なクエリを実現している。

また本手法では、キーバリューストアのデータモデルを統合するために転置キーを作成している。キーバリューストアでは、キーに対応するバリューを取得する方法でデータを抽出する。ここで用いるキーは、エンティティまたはリレーションシップのラベル、ID、およびプロパティのラベルから構成される複合キーである。一方で、分散されたプロパティを統合的に管理するためには、バリューを基点として対応する ID を取得する機能が求められる。そこでエンティティまたはリレーションシップのラベル、プロパティのラベル、およびバリューから構成される転置キーを作成し、これを対応する ID の集合と関連付ける。この転置キーを用いることでキーバリューストア上で特定のプロパティ値を持つエンティティやリレーションシップを効率的に抽出することが可能となる。

さらにデータマイグレーションも実装しており、この機能を用いてキーバリューストアにプロパティデータを分散配置させることでクエリ処理性能が向上することを実験的に示している。

本手法は U-Schema の設計上さらに RDB、ドキュメント DB、カラム指向 DB に対応を広げられると考えられる。またグラフ DB を軸とする設計上の観点からグラフクエリしか受け付けないが、分散配置されたプロパティにアクセスする際に生じるクエリの変換の機能は一部にとどまっており、より複雑な構文に対応するよう拡張が求められる。加えて現状のクエリ変換方式では実行順序をシステム側が制御することは困難であり、またグラフ探索途中に生成される中間結果の活用することができない。このため異種 DB をまたがる問い合わせ最適化を設計しづらいという課題が残されている。

3 提案手法

本研究では山下らの提案システム [4] を拡張し、U-Schema [1] を用いた統一的なスキーマ管理とデータ配置の透過性を実現する。既存手法 [2] [3] では困難であった透過的クエリ処理や異種データベース間のデータ移行機能を継承しつつ、本研究では対応データベースをドキュメント DB、カラム指向 DB、リレーショナル DB に拡充した。さらに、特定のクエリ言語に依存しない「抽象化オペレーター」と「中間結果保持機構」を新たに導入することで、論理レベルでの処理の共通化と冗長な計算コストの削減を図り、クエリ実行基盤を高度化させている。これにより、データの性質に応じた適材適所の分散配置を可能にし、大規模グラフデータに対する柔軟で拡張性の高い管理基盤を構築した。図 1 に提案システム全体の構成を示す。

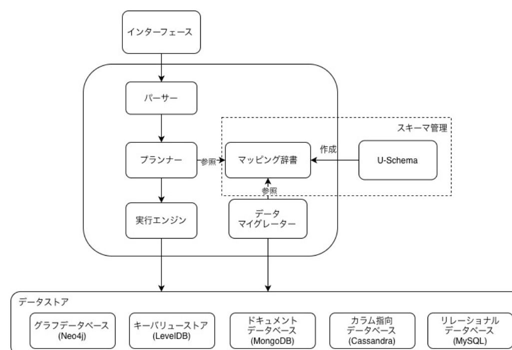

図 1: システムの構成

3.1 データ構造

このシステムにおいて、データ全体をプロパティグラフとして捉える。プロパティグラフとはグラフ構造を基盤としたデータモデルであり、特徴としてエンティティだけでなくリレーションシップ自体もプロパティを保持できる点が挙げられる。

提案手法では、エンティティおよびリレーションシップから構成されるグラフ構造についての情報をグラフ DB に格納し、それらが保持するプロパティについてはデータの性質や利用目

的に応じて異種データベース間に分散配置する。グラフ構造はRDB 等でも管理可能だが、探索時には再帰的な結合に伴う計算コストの増大が避けられない。加えて、密に結合したノード群を集約させるような最適分割は困難であり、結果として DB 間通信を頻発させる要因となる。こうした背景を踏まえ、本研究ではグラフ構造をグラフ DB だけで管理することでグラフ探索の効率を確保する。

また分散されたプロパティデータを統合的に扱うため、各エンティティおよびリレーションシップに対して ID を付与する。異種データベースに格納されたプロパティは、この ID を共通の参照キーとして管理される。これにより、ID を基点として同一のエンティティまたはリレーションシップに属するデータを特定でき、分散環境下においてもデータ全体を論理的に一貫したプロパティグラフとして把握することが可能となる。

3.2 マッピング辞書

山下らの手法[4] においてマッピング辞書はエンティティを軸として構成されており、リレーションシップについてはどのエンティティ同士を接続しているかという情報しか保持されておらず、エンティティに付随するものとして管理されている。本システムにおいてはリレーションシップが持つプロパティも分散されることを想定しており、またグラフデータベースに対してクエリを発行する際ににエンティティとリレーションシップは異なる方法で記述する必要がある。以上の理由から本システムではリレーションシップを独立した要素としてマッピング辞書に記述する。

3.3 クエリ処理

提案するシステムはグラフデータベースである Neo4j のクエリ言語である Cypher によって書かれたクエリをユーザクエリとしている。クエリ処理は、プラン作成段階と実行段階の二つのフェーズから構成される。

3.3.1 プラン作成段階

プラン作成段階では、まずユーザークエリとして与えられた Cypher クエリをパースし、クエリ構造を抽象構文木として取得する。次にこのパース結果を基に実行プランを生成する。実行プランは Neo4j のクエリ処理方式を参考に設計しており、EntityScan、Expand、VarLengthExpand、Filter、Projection のオペレーターからなる線形なオペレーション列として表現される。またこの実行プラン生成の過程において、マッピング辞書を参照し各オペレーターがアクセスすべきデータベースを決定する。図 2 はクエリをオペレーターに変換した例を示している。

以下に、各オペレーターの挙動について説明する。

1.EntityScan

クエリ処理における探索の起点となるエンティティを取得するオペレーターである。システムに登録されたエンティティ全体の中から、クエリで指定されたエンティティのラベルやプロパティ条件に基づいて候補を選択する。本オペレーターは、探索対象となるエンティティを初期段階で限定することで、後続処理における探索範囲を削減する。出力は条件を満たすエンティティの ID 集合であり、以降のオペレーターはこの ID を基準として処理を行う。

```
MATCH (n:Person {age: 25})-[r:KNOWS]->(m:Person)
WHERE m.name = "Andy"
RETURN n.name, n.age
```

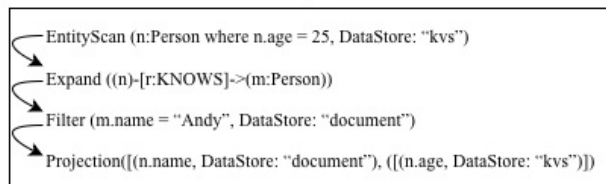

図 2: プラン作成の例

2.Expand

直前のオペレーションによって得られたエンティティを起点として、リレーションシップを 1 ホップ分探索するオペレーションである。エンティティ間の接続関係を辿ることで、新たなエンティティおよび対応するリレーションシップを取得する。これはグラフ構造に基づく探索を担うオペレーターであり、グラフ構造の情報を参照する必要がある。したがって本手法においては、グラフ構造を管理しているグラフデータベース上でのみ実行される。

3.VarLengthExpand

直前のオペレーターによって与えられたソースエンティティを起点として、可変長のマルチホップ探索を行うオペレーターである。単一ホップに限定されない探索を可能とすることで、複雑なグラフ構造に対するパス検索を可能とする。本オペレーターも Expand と同様、グラフデータベース上でのみ実行される。

挙動の詳細として、指定されたホップ数の探索を一括で実行する。その際、最終的な出力として始点と終点の重複のないペアを取得する仕様としており、この重複排除によって中間結果の削減を図っている。

一方で、本オペレーターは内部的には探索の過程で通過するエンティティおよびリレーションシップを取得しているため、これらの中間データに基づいた枝刈りの実装も可能である。ただし枝刈りの拡張を行う場合には、道中の情報を条件評価を用いるために経路情報を保持する必要があり、上述した始点と終点のみに基づいた重複排除という仕様が必ずしも適用できない場合がある。

4.Filter

それ以前のオペレーターによって生成された中間結果に対して、クエリで指定された条件を適用し、処理対象をさらに限定するオペレーターである。構造探索などによって得られた候補から、条件を満たすエンティティおよびリレーションシップのみを選択することで、後続のオペレーターの効率化を図る。

直前のオペレーターから受け取った中間結果を入力とし、そ

こに含まれる ID に対応するエンティティおよびリレーションシップにプロパティ条件を適用する。条件を満たさない ID は中間結果から除外され、評価後に残った ID 集合が次のオペレーターへ引き渡される。

5.Projection
クエリ処理の最終段階として実行され、先行するオペレーターによって選択されたエンティティおよびリレーションシップを対象に、処理結果をユーザーに提示する形式へと変換するオペレーターである。

挙動としては、先行するオペレーターから受け取った中間結果を入力とし、そこに含まれる ID に基づいて対応するエンティティおよびリレーションシップが保持する必要なプロパティ値を取得する。

3.3.2 実行段階

実行段階では、生成された実行プランに従って各オペレーターを順番に実行し、必要なデータを取得して最終的なクエリ結果を出力する。この際、expand 以外の各オペレーターについては、プラン作成段階で決定されたデータベースに対応するクエリが動的に生成され、実行される。一方で、Expand オペレーターと VarLengthExpand オペレーターはグラフ DB への直接アクセスとして実行される。各オペレーター間では、エンティティおよびリレーションシップの ID で構成されたテーブルを受け渡すことで中間結果を管理している。

3.4 データマイグレーション

さらに柔軟なデータ配置を実現するための機能として、山下らの提案システム [4] と同様にデータマイグレーションを導入する。本手法におけるデータマイグレーションとは、エンティティおよびリレーションシップが保持するプロパティを異なるデータベース間で移行可能とする仕組みであり、データ配置の変更による最適化を行うことを目的に導入している。提案手法では、グラフ構造自体はグラフデータベースに保持されたまま、プロパティのみが分散配置される。そのため、プロパティの配置を変更することで、クエリ特性や取り扱うデータの性質に応じた柔軟なデータ管理が可能となる。この仕組みをグラフデータベースとキーバリューストアの間だけでなく、RDB、ドキュメント DB、カラム指向 DB とも双方向にマイグレーションできるよう拡張している。データマイグレーションの実装は、U-Schema [1] における各データモデルとの対応づけをもとに行った。

4 評価実験

4.1 実験の目的

本実験では、提案手法に基づくシステム全体の基本的な動作を検証するとともに、現状における性能上のボトルネックを明らかにすることを目的とする。具体的には、分散配置されたデータが設計通りに正しく取得・統合されるかを確認した上で、処理段階ごとの実行時間を計測し、処理時間の集中箇所を把握する。さらに、将来的なデータ配置の最適化に向けた知見を得るため、データ配置ごとの性能特性についても分析する。これにより、現段階におけるシステムの性能特性を把握し、大規模データへの対応に向けた具体的な課題を抽出する。

4.2 使用したデータセットおよびクエリ

データセットとして、SNS を模した標準ベンチマークである LDBC Social Network Benchmark [5] を使用する。本研究が想定するデータ構成は大規模かつ多様な属性を有するものであるが、現時点における提案手法には、処理能力および対応可能なデータ規模に制約がある。本データセットは、パラメータ調整によって中間結果の生成量を容易に制御できる特性を有しており、手法の完成度が十分でない現状においても、負荷を適切に管理した状態で基本動作の検証が可能である。このような特性から、本研究では本データセットを実験用データとして選定した。

データセットの規模、およびラベルごとのデータ件数を表 1 に示す。

表 1: データセットの構成

要素	データ数
全ノード数	2,997,352
全エッジ数	34,393,552
ノード: Person	10,295
ノード: Organisation	7,955
ノード: Place	1,460
エッジ: KNOWS	346,028
エッジ: WORK_AT	44,088

また実験に用いるクエリについては、LDBC が定義する読み取り専用のワークロードに基づき、以下の条件を持つものを抽出して実行する。

- **探索範囲**: 特定の `personId` を持つノードを起点とする 1〜2 ホップ以内の知人 (`KNOWS`) 関係
- **構造条件**: 知人が `WORK_AT` 関係を持つ組織（`Company`）に所属し、その組織が特定の国（`countryName`）に対して `IS_LOCATED_IN` 関係を持つこと。
- **抽出条件**: 勤務開始年が指定された値（`workFrom`）未満である
- **出力**: 知人の ID、名、姓、会社名、入社年を射影し、入社年（昇順）、知人の ID（昇順）、会社名（降順）の優先順位でソートした上位 10 件を出力する

またこの条件に基づいて記述された Cypher クエリを図 3 に、このクエリを提案手法に基づいてオペレーターに分解した結果を表 2 に示す。

図 3: 実際に使用する Cypher クエリ

```
MATCH (p:Person {id:$personId})
-[:KNOWS*1..2]-(friend:Person)
-[work:WORK_AT]->(comp:Organisation {type: "Company"})
-[:IS_LOCATED_IN]->(:Place {type: "Country",
name: $countryName}) WHERE work.workFrom < $work-
FromYear RETURN friend.id, friend.firstName, friend.lastName,
comp.name, work.workFrom
ORDER BY work.workFrom ASC, friend.id ASC, comp.name
DESC LIMIT 10
```

4.3 評価手法

提案手法の評価にあたり、グラフ探索以外の演算で参照されるプロパティ（personId、workFrom 等）をすべて特定のデータベースへ配置し、それぞれの構成における実行性能を計測した。また比較用ベースラインとして、同一クエリを直接 Neo4j [7] へ送信した場合の処理時間を計測した。計測に際しては、OS および各データベースのキャッシュ機構に起因する計測値の変動を抑制するために同一条件下で 10 回の試行を実施した。本実験ではその算術平均を実行時間として採用する。

4.4 実験環境

提案システムは Go1.24.4 を用いて実装した。ユーザーがシステムに対して発行するクエリの記述言語には、主要なグラフデータベースサービスの一つである Neo4j [7] で用いられているクエリ言語 Cypher を採用している。システム内のパーサーには言語認識ツールである ANTLR4 [6] を用いている。

また表 3 に実験に使用したマシンの環境、表 4 に本システムの実装に伴い各データモデルごとに利用したデータベースを示す。

表 3: マシン環境

項目	仕様
OS	macOS 15.6 (Sequoia)
CPU	Apple M1 Pro (8-core CPU)
Memory	16 GB (Unified Memory)

表 4: 使用データベース環境

データモデル	データベース (バージョン)
グラフ	Neo4j 5.24.0 [7]
ドキュメント	MongoDB v8.0.10 [8]
キーバリューストア	LevelDB v1.0.0 [9]
カラム指向	Cassandra 6.2.0 [10]
リレーショナル	MySQL 9.3.0 [11]

4.5 実験結果

図 4 に、ベースラインおよび提案手法の各データ配置構成における全体の実行時間を示す。まず、提案手法に基づく全ての配置構成において、ベースラインと同等の出力結果が得られることを確認した。一方で実行時間については、全ての配置構成においてベースラインである Neo4j に直接クエリを送信する場合よりも大幅に増大していることが確認できる。

次に提案手法の各データ配置構成ごとの詳細を見ていく。表 5 は、実行プラン上の各ステップにおける処理時間をデータ配置ごとに示したものである。

計測結果を確認すると、グラフ構造の走査を担う Step2 (VarLengthExpand) および Step3・6 (Expand) においては、すべての構成で実行時間が概ね同等であった。

また、配置変更の影響を受ける Step1 (EntityScan)、Step4・5・7 (Filter)、および Step8 (Projection) においては、選択したデータモデルによって顕著な性能差が現れている。

例えば約 15000 件の中間結果に対してプロパティ評価を行う Step4 において最良値を示したキーバリューストア配置の 47.65ms に対し、最悪値となったカラム指向 DB 配置では 1357.54ms と約 28 倍の時間を要しており、このステップにおける処理時間が全体の大部分を占める結果となった。

また最終的な出力を生成する Step8 (Projection) においては、ドキュメント DB の 18.97 ms やキーバリューストアの 2.37 ms に対しカラム指向 DB は 0.29 ms と極めて低い値を記録した。

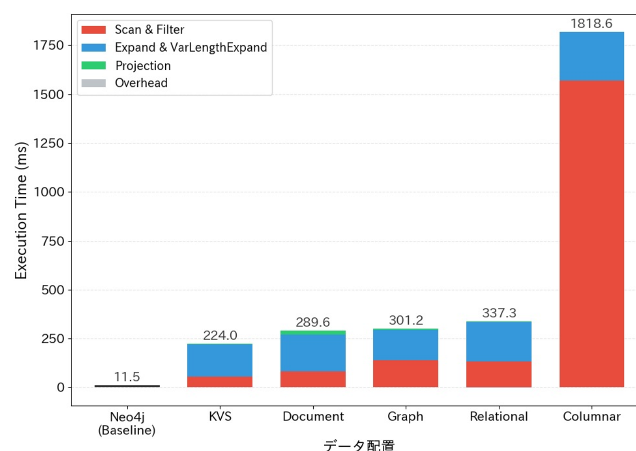

図 4: 全体実行時間比較

4.6 考察

まずベースラインから大幅に実行時間が増大した要因として、外部データベースとの通信コストに加え、各オペレーターの実行ごとに発生する中間結果保持のためのメモリ確保およびデータコピーのオーバーヘッドがあげられる。単一バッファ内で処理が完結するベースラインに対し、提案手法ではオペレーターごとに新しいメモリ領域を割り当てる構造であるため、中間結果が膨大になるほど管理負荷が実行時間を押し上げる結果となった。

各データ配置間の比較では、キーバリューストア配置が比較的安定した性能を示した。これは、外部キーに基づく参照がキーバリューストアの単一キーアクセスの特性と合致し、かつ複合キーの設計により取得範囲を物理レベルで最小化できたためと考えられる。

表 2: 各実行ステップにおける処理内容とアクセス先の対応

Step	オペレーター	役割	アクセス先
1	EntityScan	personId に基づく起点ノードの特定	各データベース
2	VarLengthExpand	KNOWS 関係に基づく可変長パス探索 (1-2 ホップ)	グラフ構造 (Neo4j)
3	Expand	WORK_AT 関係に基づく単一ホップの構造参照	グラフ構造 (Neo4j)
4	Filter	プロパティ値（workFrom）に基づく比較演算と削減	各データベース
5	Filter	プロパティ値 (type) に基づく比較演算と削減	各データベース
6	Expand	IS_LOCATED_IN 関係に基づく単一ホップの構造参照	グラフ構造 (Neo4j)
7	Filter	プロパティ値 (name) に基づく比較演算と削減	各データベース
8	Projection	最終的な出力の構成と出力プロパティの射影 (firstName, etc.)	各データベース

表 5: 提案手法における各データ配置構成のステップ別実行詳細

Step / Location	Graph (ms)	KVS (ms)	Document (ms)	Columnar (ms)	Relational (ms)	Rows
Total Latency	301.22	223.96	289.56	1818.63	337.32	-
1. EntityScan	2.18	0.03	4.05	73.45	4.45	1
2. VarLengthExpand	33.87	35.70	45.31	62.42	45.73	7,354
3. Expand	102.45	109.51	111.22	133.48	125.00	15,741
4. Filter	124.93	47.65	72.90	1357.54	115.27	10,459
5. Filter	11.98	5.94	5.02	124.03	8.81	10,459
6. Expand	16.86	21.38	30.31	53.09	30.93	10,459
7. Filter	1.39	0.91	1.31	13.73	1.38	381
8. Projection	6.94	2.37	18.97	0.29	5.26	10

対照的に、RDB 配置やドキュメント DB 配置、グラフ DB 配置では、特定のプロパティ抽出時においてもエンティティ単位のデータ管理構造がボトルネックとなった可能性がある。具体的には、目的外のプロパティを含むデータブロック全体の読み出しや、レコード単位の排他制御といった付随的な管理コストが発生する。こうした特性が、純粋なキー参照に特化した KVS と比較して実行時間を増大させた要因と考えられる。

一方、カラム指向 DB 配置においては、外部キーに基づくインデックス作成が Projection の効率化に大きく寄与した。インデックスを通じて必要なカラムの格納場所を直接特定し、不要なプロパティの I/O をスキップして目的の列データのみを選択的に読み出せたことが、高速化の主因と推測される。

しかし、Filter においては、インデックス未定義のプロパティに対する全行走査（ALLOW FILTERING）が致命的な遅延を招いた。これは、独立して格納された各列を突き合わせる際のデシリアライズ負荷が、処理件数に比例して累積したためと分析する。

5 おわりに

本研究では、U-Schema [1] に基づき分散配置されたプロパティグラフを統合管理し、異種データベース環境においてクエリを段階的に処理するシステムを構築した。先行研究 [4] のデータ移行や透過的クエリ機能を継承しつつ、新たに抽象化オペレーターを導入したことで、特定言語に依存しない実行基盤の高度化を実現している。本システムは、ユーザークエリをオペレーターへ変換し、中間結果を内部で再利用することで、データベース間を跨ぐ柔軟なクエリ実行を可能にした。また、対応データベースをドキュメント DB、カラム指向 DB、および RDB へと拡張し、より多くの性質を活用できるシステムを実現している。

一方、実験では中間結果の受け渡しを伴う逐次実行や、各データベースの特性を十分に活かせていない実装に起因する性能低下が確認された。そのため、中間処理の効率化や各データモデルに応じた実装の最適化を図る必要がある。

また、本手法はグラフ構造自体の分割管理を実装するまでには至っておらず、処理可能なクエリ表現も限定的な範囲に留まっている。今後は、データ規模の拡大や分析要求の高度化に適応するためのさらなる機能拡充に取り組む必要がある。

以上の課題に取り組んでいくことで、大規模グラフを異種分散環境において効率的に扱う Polystore system へと発展させていく予定である。

謝　辞

本研究は JST CREST JPMJCR22M2 ならびに JSPS 科研費 JP23K28091,JP23K28383,JP25H01167 の助成を受けたものである。

文　献

[1] Carlos J. Fernández Candel, Diego Sevilla Ruiz, and Jesús J. García-Molina. A unified metamodel for nosql and relational databases. *Information Systems*, Vol.104, p.101898, 2022.

[2] Boyan Kolev, Patrick Valduriez, Carlyna Bondiombouy, Ricardo Jiménez-Peris, Raquel Pau, and José Pereira. CloudMdsQL: querying heterogeneous cloud data stores with a common language. *Distributed and parallel databases*, Vol. 34, p.463–503, 2016.

[3] Léa El Ahdab, Imen Megdiche, André. Peninou, and Olivier Teste. Unified Models and Framework for Querying Distributed Data Across Polystores. *International Conference on Research Challenges in Information Science*, p.3–18. Springer, 2024.

[4] Fumihiro Yamashita, Qiong Chang, Jun Miyazaki. Unified Schema-Driven Graph Polystore: Achieving Transparency

in Multi-model Integration and Migration. Database and Expert Systems Applications: 36th International Conference, DEXA 2025, Proceedings, Part II. 2025, p. 136-143.

[5] Renzo Angles,János Benjamin Antal,Alex Averbuch,Altan Birler,Peter Boncz,Márton Búr,et al. The LDBC social network benchmark. arXivpreprint arXiv:2001.02299, 2020

[6] antlr4-go. ANTLR v4 Go runtime library. https://github.com/antlr4-go/antlr/v4 (accessed Jan 30, 2026)

[7] Neo4j, Inc. "Neo4j Graph Database." https://neo4j.com/ (accessed Jan 30, 2026).

[8] MongoDB, Inc. "MongoDB: The Developer Data Platform." https://www.mongodb.com/ (accessed Jan 30, 2026).

[9] Google. "LevelDB: A fast key-value storage library written at Google." https://github.com/google/leveldb (accessed Jan 30, 2026)

[10] The Apache Software Foundation. "Apache Cassandra." https://cassandra.apache.org/ (accessed Jan 30, 2026).

[11] Oracle Corporation. "MySQL: The world's most popular open source database." https://www.mysql.com/ (accessed Jan 30, 2026)