一般発表 | Track 2: ビッグデータ基盤技術・データセキュリティ・プライバシ

2025年2月27日(木) 10:00 ~ 12:10 C会場

[1C]データベースコア技術

座長:金政 泰彦(富士通株式会社)　コメンテータ:油井 誠(トレジャーデータ株式会社)

10:00 ~ 10:20

[1C-01]

ログ削減による決定論的並行性制御の高速化

*宮島 蒼一郎[1]、川島 英之[1] (1. 慶應義塾大学)

10:20 ~ 10:45

[1C-02]

統一スキーマモデルを活用した柔軟なデータ配置を可能とするマルチモデルデータ管理

*山下 史紘[1]、常 穹[1]、宮崎 純[1] (1. 東京科学大学 情報理工学院 情報工学系 宮崎研究室)

10:45 ~ 11:10

[1C-03]

結合を含むクエリに対するシノプシス埋込みよる近似問合せ処理の評価

*髙田 実佳[1]、合田 和生[1] (1. 東京大学)

Accelerating Deterministic Concurrency Control with Log Reduction

Soichiro MIYAJIMA† and Hideyuki KAWASHIMA†

† Keio University, 5322 Endo, Fujisawa-shi, Kanagawa 252-0882, Japan
E-mail: †{s20811sm,river}@sfc.keio.ac.jp

Abstract Deterministic concurrency control is recognized as an effective approach for distributed transaction processing, eliminating the need for distributed commit protocols. However, in existing systems like Calvin, logging overhead has become a significant performance bottleneck. This paper proposes a novel log reduction technique for deterministic concurrency control that improves system performance while maintaining correctness. The key feature of our method is limiting recovery-required information to only the final write values for each data item, enabling the omission of intermediate state records. We introduce a selective logging mechanism that identifies and omits unnecessary log records, particularly for read-only transactions. To evaluate the proposed method, we implemented it in a deterministic database system based on Calvin's architecture and conducted experiments with various workloads. The results show significant performance improvements, especially in low-contention workloads where traditional logging mechanisms tend to impose unnecessary overhead.

Key words Database, Transaction Processing, Deterministic Concurrency Control, Log Management, Performance Optimization

1 Introduction

1.1 Motivation

In distributed database systems, concurrency control protocols play a crucial role in maintaining data consistency while enabling parallel transaction processing. Deterministic concurrency control protocols, exemplified by systems like Calvin [1], have gained attention for their ability to provide strong consistency without the overhead of distributed commit protocols. These protocols pre-determine transaction execution orders, eliminating the need for complex coordination mechanisms during execution.

However, a significant challenge in these systems is the overhead associated with logging mechanisms. Traditional approaches require logging all transaction operations and their intermediate states to ensure recoverability. This comprehensive logging strategy, while ensuring system reliability, introduces substantial I/O overhead and storage costs. The impact is particularly significant in modern high-throughput environments where databases process millions of transactions per second.

1.2 Problem

The current logging mechanisms in deterministic concurrency control systems face two primary challenges:

First, these systems suffer from excessive logging overhead due to recording all intermediate states of transactions. For example, in a transaction that performs multiple updates to the same data item, traditional approaches log each update operation, resulting in unnecessary I/O operations. The high frequency of these log writes creates an I/O bottleneck, particularly in high-throughput scenarios, as the system must ensure these records are durably written to disk before proceeding. This comprehensive logging strategy significantly impacts system throughput through both storage consumption and synchronous I/O requirements.

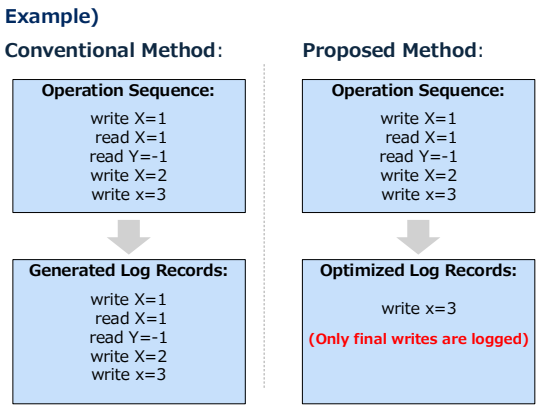


Figure 1 Overview: Conventional vs Proposed Methods

Second, current systems unnecessarily log read-only transactions, which do not modify the database state. In many real-world workloads, read-only transactions constitute a significant portion of the workload, making this overhead particularly problematic.

1.3 Contribution

This paper presents a novel log reduction technique for

deterministic concurrency control systems. Our key contributions include:

1. We introduce a selective logging mechanism that maintains only the final write values for each data item, eliminating the need to log intermediate states while ensuring recoverability.
2. We propose an efficient approach for handling read-only transactions that minimizes logging overhead without compromising system correctness or recovery capabilities.
3. We implement our approach as a modular component and demonstrate through comprehensive experimental evaluation that it achieves significant performance improvements, particularly in low-contention workloads where logging overhead is a dominant factor.

Our evaluation shows that the proposed approach reduces logging overhead by up to 54% in high-contention workloads (skew factor 0.9) while maintaining the deterministic properties of Calvin. The performance benefits are observed across all workload patterns, with 9-10% reduction in low-contention scenarios and up to 22% improvement in moderate-contention workloads.

1.4 Organization

The rest of this paper is organized as follows. Section 2 provides background on deterministic concurrency control and discusses existing logging mechanisms. Section 3 presents our proposed log reduction technique in detail. Section 4 describes our implementation and presents experimental results. Section 5 discusses related work, and Section 6 concludes the paper.

2 Preliminaries

2.1 Deterministic Protocol

Concurrency control protocols in distributed database systems can be classified into two categories: deterministic [1–4] and non-deterministic [5–9] approaches. In deterministic protocols, transaction ordering is predetermined, ensuring consistent execution results across multiple runs. This property is particularly valuable for distributed systems as it simplifies fault tolerance and recovery [1].

A deterministic protocol collects a set of transactions and then executes them in a predefined order. The result of transactions is always the same even if they are executed multiple times. For recovery purposes, these protocols typically maintain extensive logs including:

- **Write-ahead Logs**: Recording transaction operations before execution
- **Recovery Logs**: Maintaining system recovery information
- **State Logs**: Tracking database state changes

2.2 Calvin Architecture

Calvin [1] represents a significant advancement in deterministic protocols, implementing a three-phase transaction processing approach:

1. **Sequencing Phase**: Orders incoming transactions globally
2. **Scheduling Phase**: Manages transaction execution timing
3. **Execution Phase**: Processes transactions according to the schedule

In Calvin's distributed architecture, each transaction's read set and write set must be copied to all participating nodes before execution to ensure deterministic behavior. This design differs from traditional database systems where only write operations need to be logged for recovery. Calvin requires logging of read operations because the read results from one node may affect the execution at other nodes in the distributed setting.

2.3 Problem of Calvin

The current logging mechanism in Calvin faces several challenges: First, the system maintains extensive operation logs that consume significant I/O resources. While traditional databases only log write operations for recovery, Calvin must log both read and write sets for distributed transaction processing. These logs include read operations and intermediate states that may not be necessary for recovery in non-distributed settings.

Second, even local read-only transactions must be logged due to Calvin's distributed architecture. This creates unnecessary overhead in both storage and processing resources, especially when these transactions do not require distributed coordination.

Third, the synchronous nature of log writes creates a performance bottleneck. Each distributed transaction must wait for its logs to be durably written at all participating nodes before proceeding, leading to increased latency and reduced throughput.

3 Proposed Method

3.1 Design Principle

Our fundamental observation is that for recovery purposes, we only need to maintain the final state of data items modified by committed transactions. Consider a simple UPDATE query:

```
UPDATE table
SET device_name = 'device-X'
WHERE device_name = 'device-Y'
```

In traditional logging systems, such updates generate multiple log records tracking intermediate states. However, since deterministic systems guarantee identical execution se

quences, we can optimize logging by only recording the final committed values.

3.2 Log Reduction Algorithm

Our log reduction technique consists of two essential phases:

1. **Analysis Phase**: During transaction execution, we track write operations and identify opportunities for log reduction by detecting multiple writes to the same data item and identifying read-only transactions.
2. **Optimization Phase**: Before committing logs, we eliminate intermediate state records and maintain only the final committed values necessary for recovery. For read-only transactions, we skip logging entirely as they do not modify database state.

3.3 Implementation Details

We implement this optimization by extending Calvin's architecture with a write-set analyzer and a log optimizer. The write-set analyzer tracks modified data items during execution, while the log optimizer filters unnecessary records before disk writes. This approach maintains system correctness by preserving all final committed states while eliminating redundant intermediate logging.

4 Evaluation

4.1 Environment

We implemented our log reduction algorithm on Calvin based on CCBench [10]. The implementation was done in C++, and consists of approximately 600 lines of code. For experiments, we used a seaver equipped with an 8-core CPU and 16GB of memory.

4.2 Hypothesis

When setting up the experimental conditions, we considered two key factors that could affect log reduction effectiveness. First, skew in data access patterns may affect opportunities for log reduction since frequently accessed items are more likely to have redundant write operations. Second, the read/write ratio of transactions may impact reduction opportunities, as write-heavy workloads potentially offer more chances for optimization.

Based on these factors, we hypothesize that:

1. Higher skew will lead to better log reduction rates due to more redundant writes to hot items
2. Write-heavy workloads will show higher reduction rates compared to read-heavy workloads

4.3 Experimental Setup

To validate our hypotheses, we conducted experiments under various conditions. We tested skew values ranging from 0.0 (uniform) to 0.99 (highly skewed), while varying read/write ratios from 0

We measured the log reduction rate, defined as:

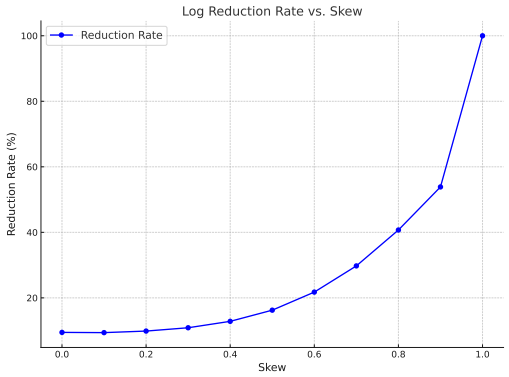


Figure 2 Log reduction rate under various skew values

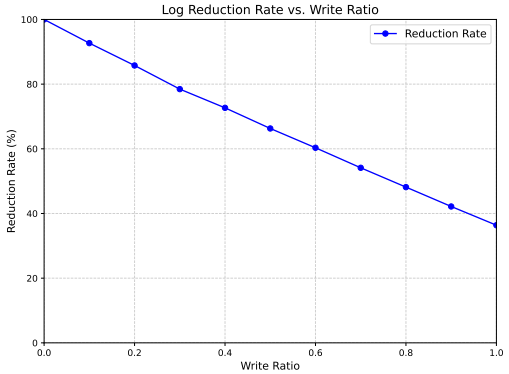


Figure 3 Log reduction rate under various skew values

$$ReductionRate = (1 - \frac{Size_{optimized}}{Size_{original}}) \times 100\%$$

4.4 Results

4.4.1 Basic Performance

Figure 2 shows the log reduction rate achieved under different skew values (ranging from 0.0 to 1.0). Our results demonstrate an exponential relationship between skew and reduction rate. With minimal skew (0.0), we observe a baseline reduction rate of 9.46%. The improvement is modest for low skew values (9.39% at 0.1, 9.87% at 0.2), but then shows exponential growth as skew increases, with particularly dramatic improvements in high-skew scenarios (40.71% at 0.8, 53.86% at 0.9).

At extreme skew (1.0), we achieve a near-complete reduction of 99.99%, though this represents a theoretical maximum rather than a practical scenario. This exponential pattern can be attributed to the increasing concentration of write operations on a smaller set of hot items as skew increases, creating more opportunities for log reduction through elimination of intermediate states.

4.4.2 Impact of Read/Write Ratio

Figure 3 illustrates how the write ratio affects reduction effectiveness. The results show a clear linear relationship between write ratio and reduction rate. At 100% reads (write ratio = 0.0), we achieve the maximum reduction rate of 100%, as read operations can be completely eliminated from

the log.

As the proportion of writes increases, the reduction rate decreases linearly, reaching 36.38% at 100% writes. This pattern aligns with our expectation that workloads with more read operations offer more opportunities for log reduction, as read operations do not need to be logged for recovery purposes.

5 Conclusion

Traditional deterministic concurrency control systems suffer from significant logging overhead by maintaining complete transaction histories. This paper proposed a novel log reduction technique that selectively maintains only the final write values necessary for recovery while ensuring correctness. Our experimental evaluation demonstrated that the proposed method can substantially reduce logging overhead in low-contention workloads while maintaining deterministic execution guarantees. Particularly in read-heavy workloads with low skew, our approach achieved a baseline reduction rate of 9.46% under minimal skew (0.0), which increased exponentially to 53.86% at a skew of 0.9 and reached up to 99.99% under extreme skew (1.0). For workloads dominated by read operations (write ratio = 0.0), the reduction rate reached a maximum of 100% as read operations do not require logging. These results highlight the potential of intelligent log management to significantly optimize deterministic database systems without compromising their fundamental properties of consistency and recoverability.

6 Acknowledgments

This paper is based on results obtained from the project ” Research and Development Project of the Enhanced Infrastructures for Post-5G Information and Communication Systems (JPNP20017) “ and JPNP16007 commissioned by the New Energy and Industrial Technology Development Organization (NEDO), and from JSPS KAKENHI Grant Number 22H03596, and from JST CREST Grant Number JPMJCR24R4 and from SECOM Science and Technology Foundation.

References

[1] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 1–12, New York, NY, USA, 2012. Association for Computing Machinery.

[2] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. High performance transactions via early write visibility. *Proc. VLDB Endow.*, 10(5):613–624, jan 2017.

[3] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Aria: a fast and practical deterministic oltp database. *Proc. VLDB Endow.*, 13(12):2047–2060, jul 2020.

[4] Jun Nemoto, Takashi Kambayashi, Takashi Hoshino, and Hideyuki Kawashima. Oze: Decentralized graph-based concurrency control for real-world long transactions on bom benchmark. *arXiv preprint arXiv:2210.04179*, 2022.

[5] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. ERMIA: fast memory-optimized database system for heterogeneous workloads. In *SIGMOD Conference*, pages 1675–1687. ACM, 2016.

[6] Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 21–35, 2017.

[7] Tatsuhiro Nakamori, Jun Nemoto, Takashi Hoshino, and Hideyuki Kawashima. Decentralization of two phase locking based protocols. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, pages 281–282, 2022.

[8] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[9] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1629–1642, 2016.

[10] Takayuki Tanabe, Takashi Hoshino, Hideyuki Kawashima, and Osamu Tatebe. An analysis of concurrency control protocols for in-memory databases with ccbench. *Proc. VLDB Endow.*, 13(13):3531–3544, sep 2020.

[11] Alexander Thomson and Daniel J Abadi. The case for determinism in database systems. *Proceedings of the VLDB Endowment*, 3(1-2):70–80, 2010.

[12] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. An evaluation of distributed concurrency control. *Proceedings of the VLDB Endowment*, 10(5):553–564, 2017.

統一スキーマモデルを活用した柔軟なデータ配置を可能とするマルチモデルデータ管理

山下 史紘† 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: †yamashita@lsc.c.titech.ac.jp, ††{q.chang,miyazaki}@c.titech.ac.jp

あらまし ビッグデータ活用やソーシャルネットワークの発達に伴い, NoSQL データモデルを含む複数のデータモデルを併用することが一般的になってきている. そのような状況においてデータベースの管理は複雑になりやすい. その原因の一端は NoSQL データモデルがスキーマレスなモデルであり, データ構造が不明瞭であることにある. 特にグラフデータモデルは他のデータモデルに比べてデータ構造やクエリ方法の柔軟性が高く特異的である. 本研究では unified schema という統一スキーマモデルでデータベースのスキーマを管理するマルチモデル DBMS の提案を行う. 提案手法ではグラフの柔軟性を維持した上で, 従来手法と比較して透過性の高いインターフェースとデータマイグレーションを提供することを目的とする. 評価においてはフレームワークを用いた定性的な評価と, データマイグレーションを通じたパフォーマンスの比較による定量的な評価を行った. 評価の結果, 従来手法と比較して透過的なクエリ処理が実現されていることが示された. また, データベース間のデータマイグレーションによってクエリの処理速度が大きく変化することが確認され, マイグレーションによる最適化の可能性を示した.

キーワード polystore system, データマイグレーション, クエリ書き換え, プロパティグラフ

1 はじめに

2000 年以降インターネットが大きく普及したことに伴いアプリケーションの要件も変化し, リレーショナルデータベース (RDB) のみでは要件を満たすことが出来ず, 今日に至るまでに様々なデータベースが登場している. これらのデータベースは総称して NoSQL データベースと呼ばれ, グラフデータベース, ドキュメントデータベース, カラム指向型データベース, キーバリューストア等が含まれる. NoSQL データベースは RDB と比較して格納できるデータの自由度が高く, 複雑なデータ構造をシンプルに表現することが可能である. 例えば, 複雑なリレーションシップや階層構造を表現する場合, RDB では多数のテーブル間の複雑な結合操作が必要となるが, グラフデータベースやドキュメントデータベースなどの NoSQL データベースを用いれば, これらの関係性や構造をより自然かつ直感的に表現可能である. アプリケーション開発においては用途に応じて適切なデータベースを選択することが推奨されており, 複数のデータベースを併用する場面も少なくない. しかし, 各データベースはそれぞれ独立したシステムであり, インターフェースやクエリ言語はそれぞれ異なる. そのため, 複数のデータベースを管理することは複雑になりやすくユーザビリティが低下しがちである. このような課題に対する取り組みとして polystore system というアプローチが提案されている.

polystore system は複数の異種データモデルに対して共通のインターフェースを提供するシステムの総称であり, 複数のデータベースを管理する場面において有効なアプローチとして様々なシステムが提案されている. しかし, 実際に提案されている polystore system には透過性をはじめとする複数の観点において改善の余地がある. 透過性には位置に対する透過性と移動に関する透過性の観点がある. 位置に対する透過性とはデータベースが複数サーバーに分散していることをユーザーが意識することなしに使えるという性質のことであり, 移動に関する透過性とはデータマイグレーションの前後でユーザーの操作に影響がないという性質のことである. 多くの polystore system のクエリ言語は取得するデータの保存先を明示する必要があり, ユーザーはデータ配置を認識しておくことが求められる. このような透過性の低さをもたらしている原因の一端は NoSQL データベースがスキーマレスなデータモデルであることにある. スキーマレスであることによりデータの構造が不明確になりやすく, システムがデータ構造を把握することが困難であり, ユーザーがクエリ内で明示する必要性が生まれている. また, NoSQL データベースの中でグラフデータベースは特異な性質を持っており, データ間の関係性や経路を表現することに長けている. このような特徴は他のデータベースで代替することは困難であり, データ統合の難しさから polystore system の従来手法においても扱いが十分であるとは言えない.

本研究では U-Schema [1] という統一スキーマモデルを活用した透過性の高い polystore system を提案する. U-Schema は RDB と 4 つの NoSQL データベースを統一表現する抽象メタモデルであり, 複数の異種データベースを一つのデータモデルとしてスキーマ表現することが可能である. これを活用することで各データベースに分散されたデータ配置を解釈可能にし, 入力クエリを適切に書き換える. また, スキーマの統一表現により複数のデータベース全体を一つのデータモデルとして解釈可能にし, データ全体のスキーマの一貫性を保ったまま, 異種

データベース間のマイグレーションが可能となっている．提案手法がデータ配置を最適化し，透過性に優れた手法であることを示す．

2 関連研究

本節では統一スキーマモデル U-Schema と Polystore system の従来手法を紹介する．

2.1 U-Schema

U-Schema は RDB と 4 つの NoSQL データベース，グラフデータベース，ドキュメントデータベース，キーバリューストア，カラム指向型データベースに対する統一スキーマモデルである．NoSQL データベースはスキーマレスであると表現されるが，実際には暗黙的なスキーマが存在する．しかしそのスキーマは格納されているデータに依存するため，スキーマオンリードとも表現される．そのような暗黙的なスキーマを統一モデルの形式で抽出することを目的とした研究である．

NoSQL データベースがスキーマレスであるということは，同じデータベースでも異なる構造のデータオブジェクトを持つことが可能である．この特徴を Structural variation として捉え，スキーマに反映させている．Structural variation とは同じラベルやエンティティのデータオブジェクトを，オブジェクトの構成要素の違いによって識別するものである．例えばグラフデータベースにおいて，同じ Person エンティティのノードでも，プロパティのスキーマが異なる場合にそれらを異なるバリエーションとして識別する．U-Schema はこの Structural variation によってスキーマレスである NoSQL データベースの柔軟性を表現可能にしている．

2.2 Polystore system

本節ではグラフデータベースをサポートしている代表的な Polystore system として CloudMdsQL [7] と Wong らの提案 [11] を紹介する．

2.2.1 CloudMdsQL

CloudMdsQL は異種クラウドデータストアから SQL ライクな共通クエリ言語でクエリ可能である Polystore system であり，RDB とドキュメントデータベース，グラフデータベースをサポートしている．各データベースに対するクエリをサブクエリとし，その実行結果をリレーショナルテーブルに変換し，メインクエリでそれらのテーブルに対する SQL を記述して統合処理を行う．1 にクエリ例を示す．クエリを記述するにはユーザーはどのデータベースにどのデータが入っているかを認識し，それを明示する必要があるためクエリの透過性は低い．また，データベース間のデータマイグレーションはサポートしていない．データ配置を最適化する戦略として，頻繁に使用されるクエリとその結果を別のテーブルにキャッシュしておき，クエリの実行時間を削減している．

2.2.2 Wong らの提案

Wong らによって提案された Polystore system はグラフデータベースとキーバリューストアをサポートしており，

```
1  // RDB に対するクエリ
2  T1(x int, y int)@rdb =
3      (SELECT x, y FROM A)
4  // ドキュメント DB に対するクエリ
5  T2(x int, z array)@mongo = {*
6      db.B.find({$lt: {x, 10}}, {x:1, z:1})
7  *}
8  SELECT T1.x, T2.z
9  FROM T1, T2
10 WHERE T1.x = T2.x AND T1.y <= 3
```

図 1 CloudMdsQL のクエリ例

```
1  SELECT gt.id, kvt.value
2  FROM KVS AS kvt
3  GRAPH_TABLE(
4      MATCH (m)-[]->(n)
5      WHERE (m.id="D1")
6      COLUMMNS(m.id AS id)
7  ) AS gt
8  WHERE kvt.key = gt.id;
```

図 2 Wong らのクエリ例

SQL/PGQ [6] をベースとしたクエリ言語を提供している．キーバリューストアは一つのテーブルとして捉え，グラフデータベースは COLUMN 句を使用してクエリ結果をテーブル形式にして扱う．各データベースに対するクエリは FROM 句にサブクエリとして記述可能でありテーブルを構成する．メインクエリにおいて各テーブルを統合する処理を記述する仕様になっている．クエリ例を 2 に示す．CloudMdsQL と同様にユーザーはデータ配置の認識とグラフデータベースとキーバリューストアからテーブルの作成を求められるため，透過性は低いシステムとなっている．また，データベース間のデータマイグレーションはサポートしておらず，CloudMdsQL のようなキャッシュテーブルも提供していない．

2.3 プロパティグラフモデル

プロパティグラフはノードとエッジそれぞれがラベルとプロパティの集合を持つグラフである．ラベルとはノードやエッジの種類を示すメタ情報であり，ラベルによってノードやエッジを特徴付け，それらをグループ化することができる．プロパティとは各ノードやエッジが持つ属性情報であり，ノードやエッジが表現するオブジェクトをプロパティとしてノードに付与することができる．プロパティグラフの代表的なデータベースには Neo4j[1]が挙げられ，プロパティグラフデータベースの可能性を最大限に引き出すことが出来るクエリ言語として Cypher [4] を提供している．[2] Cypher の構文は基本的に SQL に類似しているが，パターンマッチと呼ばれる特有の構文を持っており，ノードやエッジのパターンを指定することでネットワーク構造を柔軟にクエリすることが可能である．

1：https://neo4j.com/
2：https://neo4j.com/docs/cypher-manual/current/introduction/

3 提案手法

本研究は 2.2 節で紹介した手法とは異なり, 統合処理のための中間テーブルを持たない. U-Schema を活用してマルチモデルデータのデータ配置を把握し, 入力クエリを各データベースのネイティブクエリに書き換えてクエリを実行する.

3.1 データ構造

提案する Polystore system は, データモデル全体をグラフ構造として捉えることで異種データ間の関係性を自然に表現する. 各データエンティティをノード, データ間のリレーションシップをエッジとしてモデル化することで一貫性のある操作が可能となる. このデータの管理方法は安田らの提案 [12] の実装を参考にしている. グラフデータベースは大規模なテーブル型データを扱うことに適していないことから, 安田らはデータの依存関係のみをグラフデータベースで管理し, 対応するデータをキーバリューストアで管理することを提案している. 我々はグラフデータの各ノードを一意に識別するためにエンティティ名, ノード ID, フィールド名を用いて, グラフデータベースのノードと対応するデータを紐づけている. ここでエンティティとはラベル集合を意味し, Person というラベルが付与されたノードのエンティティは Person となる. 例えば Person ラベルが付与されたノードの Age というフィールドをキーバリューストアで持つ場合は Person:1:Age といったキーで, グラフデータベースの Person ノードと紐づけられる.

3.1.1 キーバリューストアの転置インデックス

キーバリューストアはキーで問い合わせて値を取得するという非常にシンプルな問い合わせの仕組みを持つ. そのため, RDB のように格納されている値でフィルタリングするといったような逆引きの問い合わせが出来ない. キーバリューストアにはエンコーディングパターンが設定されており, キーを一意に特徴づけているのは ID である. しかし逆引きは不可能であるため, 値から ID を取得することは出来ない. そこでエンティティ, プロパティ, 値の組み合わせから ID を取得する逆方向のキーを作成する. <エンティティ名>:<ID>:<プロパティ名>というキーに対し, index:<エンティティ名>:<プロパティ名>:<値>というキーを作成し, このキーの値に ID の集合を格納する.

3.2 スキーマ管理

異種データベースにまたがるデータを統合的に扱うため, U-Schema に基づいて各データベースのスキーマを俯瞰し統合スキーマを定義する. 図 3 のように, 統合スキーマと各種データベースのスキーマをそれぞれ用意し, それらを比較またはマージすることによって, 各エンティティのどのプロパティがどのデータベースに存在するのかを把握する. さらに, これらのスキーマからマッピング辞書 (図 4 参照) を作成する. マッピング辞書は, 各エンティティのプロパティやリレーションシップが, どのデータベースに配置されているかを示すメタデータを保持する. 以下のようにモデル化することができる.

$$\mathcal{M} = \{\ell_1 \mapsto E_1,\ \ell_2 \mapsto E_2,\ \ldots\}, \tag{1}$$

ここで, ℓ_i はラベル, E_i はそのラベルに対応するエンティティ集合とプロパティ配列やリレーションシップをまとめた情報を表す. 具体的には, E_i の各要素として

$$E_i = \{(ent_1, P_1, R_1),\ (ent_2, P_2, R_2), \ldots\} \tag{2}$$

のように, エンティティ名 ent, プロパティ集合 P, リレーションシップ集合 R を含むタプルが並ぶ構造を想定する. プロパティ集合 P は

$$P = \{\ prop \mapsto (\text{type},\ \text{database}),\ \ldots\} \tag{3}$$

のように, `prop` はプロパティ名 (例: `uri`, `Age` など) を表す. すなわち, プロパティグラフモデルで言及される「キー (プロパティ名)」に相当する文字列である. ここで, `type` はプロパティのデータ型や構造を示し, `database` は `graph`, `kvs` といった複数のデータベース名が格納される集合であり, そのプロパティが格納されているデータベースを一覧として示す. また, リレーションシップ集合 R には, ターゲットエンティティを格納しておき, エッジの探索や書き換え時に参照できるようにしている.

3.3 クエリ書き換え

本システムにはクエリを実行する際に, ノードやエッジなどの構造を保持しつつ, そのプロパティがどのデータベースに配置されているかに応じてクエリを書き換える. 先述の通りデータモデル全体はグラフ構造となっているため, 入力されるクエリはグラフクエリとなる. この書き換えに当たっては, 予め生成されたマッピング辞書を参照して各プロパティがどのデータベースに格納されているかを照合する. 以下では, その具体的な流れを述べる.

1. **WHERE 句の書き換え:**
   - **プロパティごとの格納先の判定:**
     WHERE 句で参照される各プロパティについて, マッピング辞書を用いてどのデータベースに存在するかを照合する. もし `"graph"` に含まれるなら, グラフ上で直接評価することが可能なため, クエリ内の評価式をそのまま残す. 一方, `"kvs"` に格納されている場合は, グラフには該当プロパティが存在しないため, キーバリューストアとの連携が必要である.
   - **キーバリューストアのインデックス検索:**
     `kvs` にあるプロパティの場合, まず キーバリューストアのインデックスキーを作成し該当するノード ID の一覧を取得する. 例えば, `(p.age = 30)` という評価式の場合は, `index:People:age:30` で ID 集合を問い合わせ, その結果 (例: [1,2,5]) をもとに Cypher クエリの `WHERE` 句を
     `WHERE n.ID IN [1, 2, 5]`
     の形に書き換える. これにより, Neo4j には存在しない `age` プロパティの代わりに, `ID` を用いたグラフトラバーサルが可能になる.

```
<USchema:USchema xmi:version="2.0" ...>
<entities name="People" root="true">
    <variations variationId="1">
        <features xsi:type="Attribute" name="ID">
            <type name="integer"/>
        </features>
        <features xsi:type="Attribute" name="Age">
            <type name="integer"/>
        </features>
        ...
```

統合スキーマ

```
<USchema:USchema xmi:version="2.0" ...>
<entities name="People" root="true">
    <variations variationId="1">
        <features xsi:type="Attribute" name="ID">
            <type name="integer"/>
        </features>
    </variations>
    ...
```

グラフスキーマ

```
<USchema:USchema xmi:version="2.0" ...>
<entities name="People" root="true">
    <variations variationId="1">
        <features xsi:type="Attribute" name="Age">
            <type name="integer"/>
        </features>
    </variations>
    ...
```

キーバリューストアスキーマ

図 3　各種スキーマの例

```
"entities": {
    "People": {
      "properties": {
        "ID": {
          "type": "integer",
          "database": "graph"
        }
        "Age": {
          "type": "integer",
          "database": "kvs"
        }
      },
      ...
},
```

図 4　マッピング辞書の例

2. **RETURN 句の書き換え:**
   - **プロパティごとの格納先の判定:**
     RETURN 句で指定されるプロパティ(例: `m.Name`) についても, マッピング辞書を参照して格納先の判定を行う.
   - **グラフデータベースで取得する項目とキーバリューストアで取得する項目:**
     `"kvs"` にあるプロパティの場合グラフではノード ID(`m.ID`) だけを返すように書き換える. エンティティとプロパティは明らかであるため, グラフのパターンにマッチする ID が判明すればキーバリューストアから該当する値を取得することが可能である. 一方, `"graph"` にある場合は書き換える必要はない.
3. **クエリ実行と結果統合:**
   - **グラフクエリ実行:**
     書き換えの結果得られた Cypher クエリを, グラフデータベースに対して発行する. RETURN 句の書き換えにより, 返ってくる結果はグラフ上の期待するプロパティの値か ID である.
   - **キーバリューストア問い合わせと値取得:**
     RETURN 句において ID を返すように書き換えたプロパティについて, キーバリューストアに問い合わせて対応する値を取得する. ID はリストで返され, 複数ある場合は ID の数だけキーバリューストアのキーを作成し問い合わせる. 例えば, グラフデータベースから返ってきた `m.ID` が `[1,2]` である場合, `Movie:1:name`, `Movie:2:name` といったキーを作成し, それぞれの値を取得する.

上記の手順により, プロパティの格納先に応じて WHERE 句や RETURN 句を分割・書き換えし, 必要ならキーバリューストアのインデックス検索と ID 連携を行う仕組みが実現される. 結果として, ユーザはどのデータベースにどのデータがあるかといったデータ配置を意識することなく, あたかも一つのグラフを扱っているかのように透過的に複数データベースに散在するデータを取得することができる.

3.4　データマイグレーション

任意のエンティティやその特定プロパティ単位で, データを異なるデータベース間で動的にマイグレーションすることが可能である. これは, あるエンティティのプロパティをキーバリューストアからグラフデータベースに移すことでアクセス特性を最適化したり, あるいは逆にグラフデータベースからキーバリューストアへ移すことで全体的なクエリ性能とストレージ効率を両立させるためである. 以下では, その一連のマイグレーション動作の流れを示す.

1. **マッピング辞書の参照:**
   マイグレーション対象となるプロパティが現在格納されているデータベースをマッピング辞書から取得し, マイグレーション先のデータベースを確認する.
2. **データ転送:**
   マイグレーション元のデータベースにプロパティが存在する場合, ID ごとのデータを転送する. たとえば, graph→KVS のマイグレーションでは, `(People:IDValue:name)` のキー形式でキーバリューストアに書き込み, 併せて `index:People:name:value` という形のインデックスキーに ID を登録する. 逆方向（KVS→graph）では, キーバ

リューストアから ID と値を読み取り, エンティティ名と ID で検索したグラフノードにデータを書き込む. 以下のような Cypher クエリを用いることでグラフデータベースへの書き込みが行われる.

```
MATCH (n:People)
WHERE n.ID = IDValue
SET n.name = value
```

3. **マイグレーション元のデータの削除:**
   マイグレーション元のデータベースに残っている古いプロパティは削除し, データの重複や不整合を防ぐ. 例えば, graph→KVS の場合, キーバリューストアへ書き込みが完了したらグラフデータベース上のプロパティを削除し, 逆方向のマイグレーションの場合にはキーバリューストアから以降が完了したデータのキーとインデックスキーを削除する. これによりデータの重複を防ぎ, データモデル全体としてデータの一貫性を保つ.
4. **マッピング辞書の更新:**
   移行後はプロパティの格納先の変化に併せて, マッピング辞書における `Databases` フィールドを更新する. たとえば, graph→KVS への移行が成功した場合, そのプロパティの `Databases` フィールドから `graph` を削除し `kvs` を追加する. これにより常に最新のデータ配置を反映したマッピング辞書を保持する.

上記のステップを通じて, データとマッピング辞書を常に同期させたまま移行できるため, 統合モデルに対する実際のデータ配置を一貫して把握し続けられる. 高頻度で問い合わせられるプロパティはグラフに配置し, ほとんど参照されないプロパティはキーバリューストアへ退避するといった自動最適化も考えられる.

4 評　　価

提案手法の検証のために Go 1.23 [3] を用いて実装を行った. グラフクエリの構文解析を行うパーサーは antlr 4.13 [9] を用いてビルドした. また, グラフデータベースは Neo4j 5.12 を採用し, キーバリューストアは BadgerDB 1.6 [4] を用いた. システムのアーキテクチャを図 5 に示す.

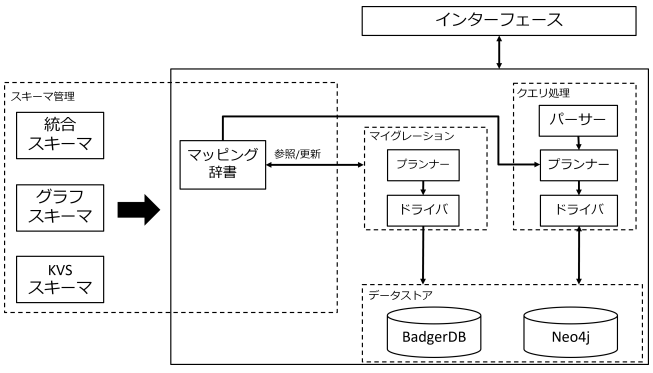


図 5　システムアーキテクチャ

3：https://go.dev/doc/go1.23
4：https://dgraph.io/docs/badger/

評価においては Ran らによって提案された Polystore system の評価フレームワーク [10] による定性的な評価を行った.

4.0.1　フレームワークを用いた評価

このフレームワークは異質性, 自律性, 柔軟性, 透過性, 最適性の 5 つの大項目からなり, 各項目は 2 つ又は 3 つの小項目を持つ. 各項目の詳細は Tan らの論文を参照されたい.

表 1 に評価結果をまとめた. 各小項目を 3 段階で評価しており, 比較手法の評価は [10], [11] にてされている評価を引用している. 提案手法は 2 つの比較手法の評価に対して相対的な評価を行った.

A. 異質性

1) データストアの異質性

現時点で扱えるデータベースはグラフデータベースとキーバリューストアにとどまっている. ただ, U-Schema はその他に RDB とドキュメントデータベース, カラム指向型データベースをサポートしており, それらをサポートするように拡張する予定である.

2) 処理エンジンの異質性

各データベースの処理エンジンを直接使用しており, それらの処理能力を制限することはない. それぞれのエンジンは対応するデータモデルに対して最適化されている.

3) クエリインターフェースの異質性

データモデル全体がグラフ構造となっているためグラフクエリをメインとして, その背後でキーバリューストアへのアクセスを併用している. そのためグラフデータベース以外のデータベースだけを操作する際も Cypher の文法で記述する必要がある.

B. 自律性

1) 連携の自律性

複数のデータベースが動的に連携するような連邦的なシナリオには対応しておらず, データベースの追加や削除を容易に行うことは出来ない. クエリの処理を行う際にはマッピング辞書を参照して対象のデータベースを特定し, 必要なデータベースにのみアクセスする. また, マッピング辞書はマイグレーションが発生するたびに動的にアップデートされる.

2) 実行の自律性

各データストアはネイティブクエリをローカルに実行する設計であるため, アプリケーションからの干渉を受けることなく実行することが出来る.

3) 進化の自律性

各データベースに対して共通のインターフェースを与えているのみで, それらのバージョンには依存しない. そのため, 各データベース製品がアップデートされても問題なく動作させることが可能である.

C. 柔軟性

1) スキーマの柔軟性

ユーザー定義の統合スキーマを入力することが可能であり, 各種データベーススキーマは 2.1 で説明したようにデータから抽出することが可能である. U-Schema は Athena [2] というスキーマ抽象宣言言語が提供されており, これを用いることでユーザー定義のスキーマを作成することが可能である.

2) インターフェースの柔軟性

構成するデータベースの機能を制限することはないため, データベースが受け入れられるクエリや関数を処理することが可能である. ユーザー定義関数の処理は行うことが出来ないため, 今後の課題とする.

3) アーキテクチャの柔軟性

インターフェースは特定のシステムである必要は無く, 互換性のある比較的モジュール化されたアーキテクチャである. U-Schema の抽出プロセスが適用可能なのは特定のデータベース製品に限られており, 同じデータモデルで異なるデータベース製品を使用するには初期状態のスキーマを Athena で定義する必要がある.

D. 透過性

1) 位置の透過性

ユーザーはクエリ内にデータの所在を明示する必要はなく, ユーザは Cypher クエリで一貫した問い合わせを行い, 背後でどのデータベース (キーバリューストア/グラフ DB) から値を取得するかをシステムが自動的に判断する.

2) 移動の透過性

データベース間のマイグレーションはプロパティ単位で行うことが可能であり, マッピング辞書でそれぞれの型も管理しているため, ユーザーはデータの方や構造を意識することなくマイグレーションを行うことが可能である. 今後統合スキーマをもとに統計情報を取得し, 動的なマイグレーションが行われるように拡張予定である.

E. 最適性

1) 連合プランの最適性

詳細な単位でのデータマイグレーションをサポートしているが, クエリを処理する上で動的なデータ変換やマイグレーションを活用出来ていない. しかし, 先述の統計情報に基づいた動的なマイグレーションを実現し, クエリ実行の際にマイグレーションせずともあらかじめ最適なデータ配置となることを目指す.

2) データ配置の最適性

データベース間でのマイグレーションがプロパティ単位で可能である. 頻繁にクエリされるプロパティは Neo4j に配置し, そうでないプロパティは他のデータベースに配置しておくといったような最適化が可能である.

大項目	小項目	CloudMdsQL	Wong らの提案	提案手法
異質性	データストア	優	普通	普通
	処理エンジン	優	優	優
	インタフェース	優	普通	普通
自律性	連携	普通	優	劣
	実行	優	優	優
	進化	普通	優	優
柔軟性	スキーマ	劣	普通	普通
	インターフェース	普通	劣	普通
	アーキテクチャ	普通	普通	普通
透過性	位置	普通	劣	優
	移動	普通	劣	優
最適性	連合プラン	普通	劣	劣
	データ配置	普通	劣	優

表 1　評価フレームワークに基づく各手法の比較

	$prop_{where}$	$prop_{return}$	$prop_{otherwise}$
状態 (i)	BadgerDB	BadgerDB	BadgerDB
状態 (ii)	BadgerDB	Neo4j	BadgerDB
状態 (iii)	Neo4j	BadgerDB	BadgerDB
状態 (iv)	Neo4j	Neo4j	BadgerDB
ベースライン	Neo4j	Neo4j	Neo4j

表 2　各マイグレーション状態におけるデータ配置

4.1　マイグレーションによるクエリ処理時間の変化

要素	数
ノード数	1252
エッジ数	3231
ノードのラベル数	15
エッジのラベル数	12

表 3　LUBM データセットの構成要素

LUBM [5] のデータセットを用いて, データベース間でのマイグレーションを行うことでクエリの処理にかかる時間がどのように変化するかを測定しマイグレーションの効果を評価した. LUBM は大学の組織構造や活動を模擬したグラフデータセットであるため, Neosemantics [8] というプラグインを活用し, RDF 形式のデータをプロパティグラフに変換して使用した. プロパティグラフとした際のデータセットの構成要素を表 3 に示す. クエリは 9 種類の LUBM のテストクエリを用いた. プロパティのアクセスにおいてキャッシュの影響を抑えるために WHERE 句において使用される評価式の値をランダムに設定するようにした. マイグレーションの状態は (i) 全てのプロパティのデータが BadgerDB にある状態, (ii) WHERE 句でアクセスされるデータは BadgerDB にあり RETURN 句でアクセスされるデータは Neo4j にある状態, (iii) WHERE 句でアクセスされるデータは Neo4j にあり RETURN 句でアクセスされるデータは BadgerDB にある状態, (iv) WHERE 句と RETURN 句でアクセスされるデータ両方が Neo4j にある状態 の 4 種類である. WHERE 句でアクセスされるプロパティを $prop_{where}$, RETURN でアクセスされるプロパティを $prop_{return}$, それ以

外のプロパティを $prop_{otherwise}$ とし，各マイグレーション状態のデータ配置を表 2 に示す．クエリを各マイグレーションの状態でランダムな順に実施し，それぞれのクエリの処理時間とその内訳を計測し，マイグレーションの状態ごとに平均値を求めた．実行時間の内訳は，Neo4j での処理時間，WHERE 句でアクセスされる BadgerDB の処理時間，RETURN 句でアクセスされる BadgerDB の処理時間，その他のオーバーヘッドである．ベースラインとして，アクセスされないデータも含めて全て Neo4j に配置した状態でクエリを実行した場合の実行時間は平均 46.44ms であった．測定の結果を図 6 に示す．マイグレーションの状態によって処理時間が変化しており，(i) の状態の時が最も処理時間が短く，(iv) の状態の時が最も処理時間が長いことが分かる．また，いずれの状態においてもベースラインを上回り，高速化されている．

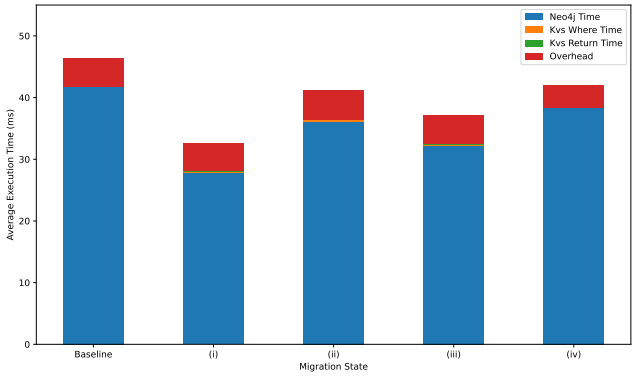


図 6　マイグレーションの状態によるクエリ処理時間の変化

5　考　　察

マイグレーションによるクエリ処理時間の変化について考察する．本実験では，クエリを実行する前に 4 種類のマイグレーション状態のいずれかを適用し，プロパティ単位で Neo4j と BadgerDB のどちらにデータを置くかを変更したうえで実際のクエリを実行しその実行時間を計測した．このとき，すべてのプロパティが BadgerDB 上に配置した状態が最良のパフォーマンスを示した．逆に，クエリでアクセスされる全てのプロパティを Neo4j に配置した状態では最も遅くなるという結果を得た．図 6 に示した実行時間の内訳を見ると，Neo4j が実行時間に大きく影響を与えており，状態 (iv) では Neo4j の実行時間が大きく増大している．一方，BadgerDB での実行時間は非常に小さく，ほぼ 0 ms に近い値となっている．本来，クエリでアクセスされる全てのプロパティを Neo4j に置く状態は，Cypher クエリの書き換えやデータ統合の負担が軽減されるため，最も高速に動作すると想定していた．しかし，実験結果ではむしろ最悪の性能を示した．提案手法が行う “`WHERE p.ns0__name = '...'`” を “`WHERE p.ID IN [...]`” に書き換える手法が，BadgerDB 側のインデックスを非常に高速に活用できているため，結果的に Neo4j における同種のフィルタ処理を上回る性能を示したと考えられる．

しかし，今回のように全てのプロパティデータを BadgerDB に配置する方法は一般化できるとは限らず，Neo4j にアクセスされるプロパティを配置しておく方が良い場合もあるだろう．データベースリソースの状況やネットワーク状況によっても各データベースのパフォーマンスは変化するため，最適な配置は変わる．これらの要素を総合的に考慮して最適な配置を選択することが重要である．この点については，今後の更なる検討を行いたい．いずれにせよ，データベース間でデータの配置を変更することで，クエリの処理時間に大きな影響を与えることが確認できた．提案手法はこのようなデータマイグレーションをサポートしているため，クエリ処理時間を短縮するという観点でデータ配置を最適化出来る可能性がある．

6　お わ り に

U-Schema を活用したスキーマ管理により，統一的なデータモデルのクエリによる polystore system の提案を行った．統合スキーマと各種データベーススキーマからマッピングを管理する辞書を作成し，統合モデルであるグラフ構造のどの部分がどのデータベースに格納されているかを解釈可能にした．それにより，クエリ処理時にはマッピング辞書を参照することで入力されるグラフクエリを透過的に書き換えることが出来るため，ユーザーは分散されたデータ配置を意識することなくクエリすることが可能である．従来手法では SQL ライクな言語を提供し，サブクエリで各データベースのネイティブクエリを記述し，その結果をテーブルに格納してメインクエリで統合処理していたが，提案手法ではグラフクエリを入力としてシステムが各データベースのネイティブクエリに書き換える．そのため，従来手法と比較して透過性の高いクエリ処理が実現されている．また，数多のデータモデルの中でも特異的な性質を持つグラフデータベースをベースとしており，テーブル構造に限らないより柔軟なクエリが可能である．

データマイグレーションはプロパティ単位で行うことが可能であり，マッピング辞書にてデータ構造や型を管理することでマイグレーションの詳細はユーザーから隠されており，透過的に行われる．従来手法ではデータベース間のマイグレーションはサポートされていなかったが，仮にデータベース間のマイグレーションが行われる場合は，新しいデータ配置に併せてクエリを書き換える必要がある．提案手法ではマッピング辞書にてデータ配置を管理しているため，ユーザーはデータの移行に関して意識することなく一貫したクエリで実行することが可能である．

また，データマイグレーションによりデータ配置を最適化する可能性を示した．データベース間でデータ配置を変更することで，クエリの処理時間に大きな影響を与えることが確認できた．2.2 節で紹介したグラフデータベースをサポートしている polystore system の従来手法は，データベース間のマイグレーションをサポートしていないため，このような最適化の可能性がない．この点において，提案手法の方が拡張性が高くより優れた手法であると言えるだろう．

今後の課題としては以下が挙げられる．初めに動的なマイグレーションの実装である．現時点では詳細な単位でのマイグレーションがサポートされているが, 実際にクエリの頻度などを集計して最適なデータ配置にマイグレーションするようにはなっていない．本稿では紹介しなかったが BigDAWG [3] という polystore system においては, データマイグレーションを行ってより最適なクエリプランを選択することが可能となっている．提案手法ではクエリの過程では無く, 予めデータ配置を最適にしておくことでより良いクエリプランが作成されるようにアップデートすることを目指している．

次にマイグレーションが正常に終了しなかった場合のトランザクション管理である．マイグレーションの途中で何らかの理由（ネットワーク障害やノード障害, ストレージ残容量不足など）によってマイグレーションが正常に終了しなかった場合, 途中状態でデータが両方のデータベースに部分的に書き込まれていたり, あるいはインデックスやマッピング辞書が不整合を起こしたりする可能性がある．このような不測の事態に対しては, 以下のようなトランザクション管理を考える必要がある．

次に対応データモデルの拡充である．比較手法である CloudMdsQL は 3 種類のデータモデルをサポートしており, この点において提案手法は劣っている．U-Schema が対応するデータモデルには今回の実装で用いたモデル以外に 3 種類のモデルをサポートしているため, これらを順次追加していきたい．

謝　　辞

本研究は, JST CREST JPMJCR22M2 の支援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] Alberto Hernández Chillón, Diego Sevilla Ruiz, and Jesús García Molina. Athena: A database-independent schema definition language. In Iris Reinhartz-Berger and Shazia Sadiq, editors, *Advances in Conceptual Modeling*, pp. 33–42, Cham, 2021. Springer International Publishing.

[3] Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, Magda Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stan Zdonik. The bigdawg polystore system. *SIGMOD Rec.*, Vol. 44, No. 2, p. 11–16, aug 2015.

[4] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, p. 1433–1445, New York, NY, USA, 2018. Association for Computing Machinery.

[5] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. Lubm: A benchmark for owl knowledge base systems. *Journal of Web Semantics*, Vol. 3, No. 2-3, pp. 158–182, 2005.

[6] CH. International Organization for Standardization, Geneva. *ISO/IEC 9075–16. 2022. Information technology - Database languages SQL - Part 16: Property Graph Queries (SQL/PGQ). Standard.* 2023.

[7] 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, pp. 463–503, 2016.

[8] Ezequiel José Veloso Ferreira Moreira and José Carlos Ramalho. SPARQLing Neo4J. In Alberto Simões, Pedro Rangel Henriques, and Ricardo Queirós, editors, *9th Symposium on Languages, Applications and Technologies (SLATE 2020)*, Vol. 83 of *Open Access Series in Informatics (OASIcs)*, pp. 17:1–17:10, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[9] Terence Parr, Peter Wells, Ric Klaren, Loring Craymer, Jim Coker, Scott Stanchfield, John Mitchell, and Chapman Flack. What's antlr, 2004.

[10] Ran Tan, Rada Chirkova, Vijay Gadepally, and Timothy G. Mattson. Enabling query processing across heterogeneous data models: A survey. In *2017 IEEE International Conference on Big Data (Big Data)*, pp. 3211–3220, 2017.

[11] Jun Miyazaki Weijun WONG, Qiong CHANG. A polystore system for graph and key-value databases based on sql/pgq. *16th Forum on Data Engineering and Information Management*, 2024.

[12] 安田香子, 常穹, 宮崎純．リネージュデータ管理システムによるデータの流通管理・予測．第 16 回データ工学と情報マネジメントに関するフォーラム, 2024.

結合を含むクエリに対するシノプシス埋込みよる近似問合せ処理の評価

高田　実佳†　　合田　和生†

† 東京大学情報理工学系研究科　〒 153–8503 東京都目黒区駒場 4-6-1
E-mail: †{mtakata,kgoda}@tkl.iis.u-tokyo.ac.jp

あらまし　業務データを収集し, 蓄積したビッグデータを分析することによる業務の改善・新たな知識発見への期待が高まっている．そのような分析では集計が頻繁に実施される．集計には，必ずしも従来データベースが求める正確な値は必要ではなく，それよりもレスポンスの早さが求められることがある．本論文では，索引構造に付加情報を埋め込むことによって結合を含む問合せに対する近似解を高速検索する方式を既存 RDBMS で実装し，検証する．

キーワード　データベース，索引，近似問合せ

1　はじめに

多くの企業や組織では，日々様々な業務データが生成され，そして収集・蓄積された大量データを分析することによって業務の改善や新たな知識発見に期待が高まっている．業務データの蓄積・管理には構造データベースが広く利用されており，様々な合計値や最大値，最小値など集計値の問合せ処理が分析には必要とされる．例えば，小売業では，日々の売り上げが期待できる製品，製品数や人員を適切に調整する為，製品の購買履歴を用いて，一日の売り上げ総数，在庫数，来客数等の定期的な集計が必要とされている [1]．この様な分析はインタラクティブに実施されることが多く，その為にはデータベースに蓄積されたデータに対し，高速に様々な集計値計算が実行されることが求められる．

高速な集計値計算に向けて，正確な値を高速に求めるデータベースの検索技術はこれまで多数提案されてきた [2]．代表的なものの一つには索引があり，中でも B^+-tree [3] は多くの関係データベースで利用されている． B^+-tree は範囲検索を効率的に行えるメリットがあり集計計算に必要な範囲のデータを高速に検索する為に有効である．しかし，日々業務データが生成・蓄積されることで，そうして増大した業務データを対象とした集計計算の場合，索引のサイズが大きくなり，最下層のリーフページまでノードを辿ると検索時間が増大するという問題が生じている．

インタラクティブな分析における高速な集計値計算を考慮した時に，正確性は必ずしも必要ではない．こうした観点に着目し，[4] らは，既存の B^+-tree など索引にシノプシスを埋込むことで似問合せの高速化手法 (Synopsis-aware search; SAS, Synopsis-aware search plus inter-attribute correlation; SAS+) を提案している．SAS は，B^+-tree の中間ノードに下位ノードの最大値，最小値，合計値，総数といった統計情報をシノプシスとして持たせておく事で B^+-tree のリーフページまで辿らずとも，近似値を求めることができるという近似問い合せ手法であり，SAS+は SAS を発展させ，複数カラム間の相関を考慮してシノプシスの利用可否を決定する近似問合せ手法である．本論文では，[4] の研究を発展させ，業務データの蓄積・管理に広く利用されている関係データベースシステムである PostgreSQL [5] にシノプシスを埋込んだ索引による近似問合せ処理 (SAS, SAS+) を組込み，試作を実装し，単一表に対する問合せに加え，KD-tree を用いて結合を含む問い合わせに対して検証した．

本論文の構成は，以下の通りである．第 2 章では，既存研究とその課題について言及し，第 3 章では，結合を含むクエリに対するシノプシス埋込み索引による近似問合せ処理について説明する．第 4 章では，実装方式について述べ，第 5 章では，評価を示す．第 6 章では，今後に向けた課題と将来展望について纏める．

2　関連文献

オンライン分析処理（OLAP）[6] をはじめ分析クエリ処理は，トランザクションデータが増加する中で分析性能を向上させるために広く研究されてきた分野である．その検索性能を向上させる従来の手法の 1 つとして，データ構造の設計が挙げられる．広く使われるデータ構造としては，検索空間を限定することで高速なデータ取得を可能にする索引が存在する．例えば，B+木 [2,3] のような索引は，指定されたキーを保持し，与えられたクエリ制約に基づいてデータの探索範囲を絞り込むことで効率的にクエリ結果を取得することが可能である．また，また別の手法として，事前計算されたクエリ結果を格納するマテリアライズドビュー [7] が存在する．マテリアライズドビューを利用することで，クエリ処理時には事前計算されたデータを活用し，データ取得・計算コストを削減してクエリ処理を高速化することが可能である．特に，複数の類似したクエリを含むワークロードにおいて，マテリアライズドビューはクエリ性能を向上させる可能性を持つ．しかし，これらのデータ構造はデータ取得性能を効率化する一方で，ストレージのオーバーヘッドやメンテナンスコストを伴うという課題が存在する．そのため，データ取得性能とオーバーヘッドのバランスを最適化すること

が重要である．

一方，厳密な値ではなく近似値を求めることを目的とした近似クエリ処理（Approximate Query Processing, AQP）が研究されてきている．AQP は，主にオンライン処理とオフライン処理の 2 つのカテゴリに分類される．オフライン処理は，保存されたデータからサンプリングされたデータを用いて，クエリ時に集計結果を計算・推定するものである [8–10]．問合せ時にサンプリングデータを用いて近似解を推定し，インタラクティブなクエリの繰り返しを通じて近似解の精度を向上させる技術も提案されている [8]．オンライン処理は，事前処理や事前計算値を保持するためのストレージオーバーヘッドを必要としないという利点を持つが，クエリ時に追加のサンプリング処理が必要となるため，クエリ処理時間が増加する可能性がある．一方，オフライン処理では，統計情報や要約情報などを事前に計算・格納し，クエリ時にその事前計算された情報を利用して近似解を返す方法である [11, 12]．このアプローチでは，シノプシス（要約情報や追加情報）を事前に保持し，それを利用して高速に近似解を返すことが可能である．例えば，ウェーブレットアプローチとして，圧縮された要約情報を保持して検索空間を絞り込む手法が研究されている [13, 14]．また，ヒストグラムをシノプシスとして保持し，データセットの分布を示す方法も存在する [15]．このような事前処理アプローチは，クエリ時の計算コストを削減し，オンライン処理よりも高い近似精度を提供する可能性を持つが，事前計算された追加情報を保持するためのストレージ容量などオーバーヘッドを伴うという課題がある．

近年では，オンライン処理と事前計算アプローチのそれぞれの利点と欠点を考慮し，両者を組み合わせたハイブリッドアプローチが提案されている．BlinkDB [16] は，大規模データにおける他の既存 AQP アプローチと比較して，高精度な近似解を提供することを実証した．BlinkDB は，オンラインのサンプル選択戦略と，ワークロードに基づく多次元層化サンプリングを採用することで，高精度な近似解を取得することが可能である．また，[17] では，シノプシスとサンプリングを組み合わせ，インタラクティブな応答時間で集計値の近似解を返す AQP++が提案されている．さらに，[18] による提案手法では，クエリに応じてオンライン処理またはサンプリングデータを使用するかを判断することで，高精度な近似クエリ処理を実現している．これらのアプローチは高い精度を示しているが，既存のデータベースシステムへの統合が十分に考慮されていないという課題が存在する．

既存データベースシステムへの統合を考慮するため，Hive, Spark，Impala，Redshift などの既存データベースシステムにおいて，より広範なクエリをカバーするクエリリライトを提供するミドルウェアアーキテクチャである Aqua [19]，IDEA [20]，および VerdictDB [21] が提案されている．特に VerdictDB は，既存の SQL ライクなデータベースシステムに対して，システムの変更を加えることなく導入可能であることを実証している．これらのアプローチは本研究の手法と類似しているが，既存のシステムに統合するオーバーヘッドについては十分に議論されていない．

以上を踏まえ，本論文では既存のデータベースシステムにおける近似問合せの適用した際のストレージオーバーヘッドと問合せ実行性能について検証する．

3 結合を含むクエリに対するシノプシス埋込みによる近似問合せ

本章では，シノプシス埋込み索引を用いた近似問合せ処理と，それを単一表のみならず結合を含む問合せに適用する方式について提案する．

これまで，単一表に対するシノプシス埋込み索引を用いた近似問合せ処理については，[4] らは，既存の索引に付加情報であるシノプシスを埋込むことによって近似問合せの高速検索する SAS(Synopsis-aware search) と，属性間の相関を考慮した SAS+(Synopsis-aware search plus inter-attribute correlation) を提案してきた．SAS は，事前にシノプシスを埋め込んだ索引を生成しておき，問合せクエリが入力されると，その索引に埋め込まれたシノプシスを利用する事で探索範囲を狭め，高速に近似解を返す検索方式であり，B^+-tree を用いた評価では，ストレージ負荷が約 2%程度に抑えながら最大 25% 以上近似解の応答時間を高速化を達成できる場合があることを示している．

本研究では，シノプシス埋込み索引を結合を含むクエリに対しても適用する方式を提案する．図 1 を用いて SAS の適用可能性と有効性を説明する．図 1(a) は関係データベースで管理されたテーブルの例 (CUSTOMER 表，ORDERS 表) であり，それらのテーブルの ORDERKEY, CUSTKEY の組み合わせを索引キーとし，付加情報を埋め込んだ KD-tree の例を図 1(b) に示す．図 1(b) には中間ノードとリーフノードを有しており，中間ノードには索引キーとして 1 段目には ORDERKEY，2 段目には CUSTKEY と（水色箇所），下位ノードへのポインタ（灰色箇所），さらに下位ノードの集計値をシノプシスと呼ばれる付加情報として埋め込まれている（橙色箇所）．図 1(b) の例では，付加情報として CUSTKEY 毎の SUM(PRICE) を保有している．リーフノードには，あらかじめ問合せに含まれる結合処理をした結合表のデータあるいはデータへのポインタを保有している．単一表に対する SAS による問合せ処理同様．ルートから深さ毎に対応する索引キーを用いて深さ優先探索を実施し，問合せクエリに対してマッチする集計値を中間ノードで発見した時点で，それより下位のノード検索をスキップする．これにより，データページへのアクセス量を削減し，実行性能が向上できると予想できる．また，オーバーヘッドに関しても，KD-tree は，B^+-tree 同様に，中間ノードは軽量であり，その特性を利用することで，中間ノードに下位ノードの集計値をシノプシスとして埋め込んだとしても，シノプシスによるストレージオーバーヘッドは軽量になると推定できる．以上により，問合せを含む検索においてもオーバーヘッドを抑えて，索引のリーフノードの探索時間を削減し，問合せ処理時間を大幅に短縮できると考える．

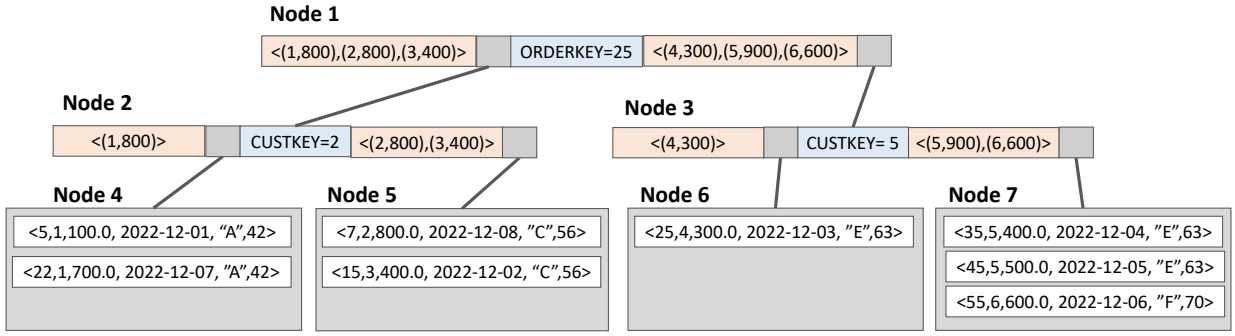
CUSTOMER

CUSTKEY	NAME	AGE
1	“A”	42
2	“B”	32
3	“C”	56
4	“D”	88
5	“E”	63
6	“F”	70
7	“D”	45
8	“E”	36

ORDERS

ORDERKEY	CUSTKEY	PRICE	ORDERDATE
5	1	100.0	2022-12-01
7	2	800.0	2022-12-08
15	3	400.0	2022-12-02
22	1	700.0	2022-12-07
25	4	300.0	2022-12-03
35	5	400.0	2022-12-04
45	5	500.0	2022-12-05
55	6	600.0	2022-12-06

(a) テーブル例



(b) シノプシス埋込み KD-tree の例

図 1: シノプシス埋込み索引の例

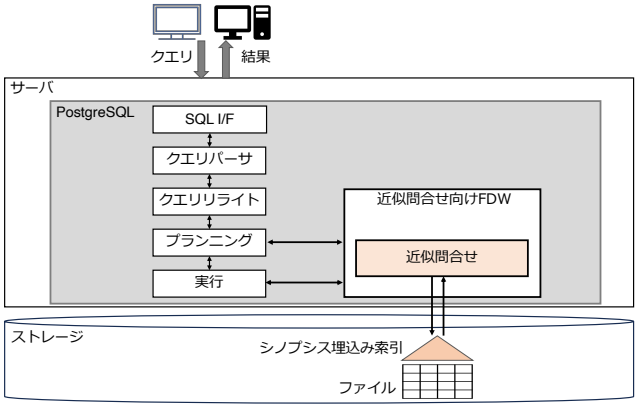


図 2: PostgreSQL 外部データラッパを用いたアーキテクチャ

4 既存関係データベースシステムへの近似問合せ手法の組込み

本章では，既存のデータベースシステムとの併用に向けて，シノプシス埋込み索引を用いた近似問合せ（SAS, SAS+）の実装について述べる．この実装により，従来の索引を用いた検索を超えるクエリ処理性能を実現することを目的としている．

既存のデータベースシステムとして，本研究ではビジネスデータを管理するための最も広く知られたデータベースシステムの 1 つである PostgreSQL [5] を用いた．PostgreSQL は，外部データラッパー（Foreign Data Wrapper, FDW）をサポートしており，外部データを外部テーブルとして定義することで，データベースドメイン外に保存されたデータにアクセスすることを可能にする API を提供している．クエリが与えられると，PostgreSQL エンジンはクエリの解析，書き換え，アクセスプランを生成し，実行して必要な値を取得する．その際に，提供された FDW の API を独自に実装することで，指定の外部データソースにアクセスするためのアクセスプランを生成，および実行プロセスを可能とする．その為，主に 2 種類の API が必要であり，FDW は 1 つはクエリ処理の計画を立てるためのものであり，もう 1 つはクエリ処理を実行するためのものである．本研究では，PostgreSQL は，外部の PostgreSQL にアクセスする為の postgres_fdw [22] を外部データラッパーの 1 つとして公開しており，これを参照し，シノプシス埋込み索引を用いた近似問合せ手法（SAS, SAS+）を実行する，クエリ処理計画用の *GetForeignRelSize, GetForeignPaths, GetForeignPlan* およびクエリ処理実行用の *BeginForeignScan, IterateForeignScan, ReSacnForeignScan, EndForeignScan* を実装した．図 2 は,FDW（Approximate_fdw）を用いて SAI にアクセスする PostgreSQL サーバーのクエリ処理過程を示している，標準 SQL インターフェースを通じてクエリが受け取られると，PostgreSQL エンジンがその解析と書き換えを行う，クエリの対象から，外部テーブルへのアクセスが必要な場合，PostgreSQL サーバーは以下の API を呼び出す．外部テーブルのサイズを推定するための *GetForeignRelSize*，外部テーブルへのアクセスパスを構築するための *GetForeignPaths*，実行プランを作成するための *GetForeignPlan*，そして FDW の下で定義された外部テーブルにアクセスするためのクエリ処理を準備する *BeginForeignScan* ，クエリ処理を実行する *IterateForeignScan, ReSacnForeignScan*，およびクエリ処理を終了する *EndFor-*

eignScan である．本研究では, シノプシス埋込み索引にアクセスするために，シノプシス埋込み索引をストレージ内のファイルシステム上に配置し，それらを用いた近似問合せ処理（SAS, SAS+）を *IterateForeignScan* にて呼び出し，クエリに対して正確な値または近似値を算出し，その値が SQL インターフェースに返される．本研究では，FDW を用いて PostgreSQL にシノプシス埋込み索引を用いた近似問合せ手法を適用し，様々なクエリに対して正確な解または近似解を返す実装アプローチを提案している．設計の詳細は異なる可能性があるものの，類似の設計を他のデータベースシステムにも適用可能であると考えられる．

5 評　　価

本章では，[4] らの提案したシノプシス埋込み索引を用いた近似問合せ手法 (SAS．SAS+) を既存データベースシステムに組込み，その有効性を単一表に対するクエリに対しては B^+-tree, 複数の表に対して結合を含むクエリに対しては KD-tree を用いて検証する．

5.1 実 験 環 境

本実験では，ベンチマークセットである TPC-H [23] の dbgen を用いて生成したデータセットを利用する．データセットのスケールファクタ (SF) は，1,10 を用意した．

次に，問合せ処理に用いる索引として，単一表に対するクエリに対しては，多くのデータベースシステムで適用されている B^+-tree [2, 3] を用い，以下の索引キーをもつ B^+-tree 索引を用意した．全てノードサイズは 8192Byte とした．

- ORDERS (ORDERKEY): ORDERS 表の O_ORDERKEY を索引キーとしてもつ索引
- LINEITEM (ORDERKEY, LINENUMBER): LINEITEM 表の (L_ORDERKEY, L_LINENUMBER) を索引キーとする索引
- LINEITEM (SHIPDATE): LINEITEM 表の L_SHIPDATE を索引キーとする索引
- LINEITEM (PARTKEY): LINEITEM 表の L_PARTKEY を索引キーとする索引.

索引に埋込む付加情報であるシノプシスとしては，以下を用意した．

- **pg-orig**: シノプシス埋込みなし（ネイティブな PostgreSQL）
- **Nosyn**: シノプシス埋込みなし (FDW による実装)
- **Syn1**: 索引キーの集計値 (SUM, MAX, MIN, COUNT) (e.g., ORDERS (ORDERKEY) の場合 O_ORDERKEY の集計値)
- **Syn2**: Syn1 に加え，索引キーとは異なる属性の集計値 (SUM, MAX, MIN, COUNT)
- **Syn3**: Syn2 に加え，索引キーと索引キーとは異なる属性との相関係数および相関係数計算に必要な集計値

本実験では評価クエリとして，単一表の問合せに対してクエリは (Q1-Q4) を用いた．

表 1: 単一表に対するクエリ

Q1	SELECT SUM(L_EXTENDEDPRICE) FROM LINEITEM WHERE L_ORDERKEY BETWEEN 36000000 and 42000000 AND L_SHIPDATE BETWEEN 1992-01-01 and 1992-12-31;
Q2	SELECT MAX(L_EXTENDEDPRICE) FROM LINEITEM WHERE L_ORDERKEY BETWEEN 36000000 and 42000000 AND L_SHIPDATE BETWEEN 1992-01-01 and 1992-12-31;
Q3	SELECT SUM(L_EXTENDEDPRICE) FROM LINEITEM WHERE L_ORDERKEY BETWEEN 36000000 and 42000000 AND L_PARTKEY BETWEEN 1800000 and 2000000;
Q4	SELECT MAX(L_EXTENDEDPRICE) from LINEITEM WHERE L_ORDERKEY BETWEEN 36000000 and 42000000 AND L_PARTKEY BETWEEN 1800000 and 2000000;

さらに，結合を含む問合せに対しては，TPC-H の Q3 を対象として，KD-tree [24] を用いて，索引キーとして (C_MKTSEGMENT, O_ORDERSDATE, L_SHIPDATE) とした．索引に埋込むシノプシスとしては，

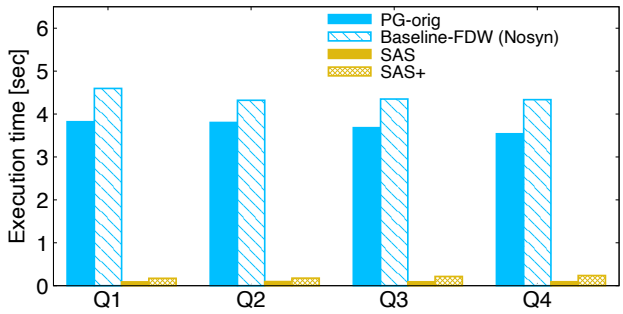
- **Nosyn**: シノプシス埋込みなし
- **Syn**: (L_ORDERKEY, O_ODERDATE, O_SHIPRIORITY) のグループ数と SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) を用いた．

本実験では，索引は事前に生成しファイルとして保存し，問合せ時に索引を呼び出し，索引を用いた検索は，IterateForeignScan API 内で実施した．評価クエリでは ORDERS 表および LINEITEM 表を対象としていることから，ORDERS 表および LINEITEM 表を外部表として事前に create する．これにより，外部表に対する問合せクエリを PostgreSQL インターフェース (I/F) から入力すると，近似問合せ処理を実行する FDW(Approximate_FDW) が呼び出される様にした．

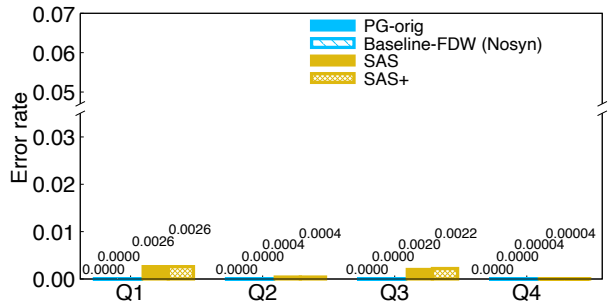
評価軸には，単一表に対する問合せに対して，Native PostgreSQL による B^+-tree 検索 (PG-orig)，FDW を用いた B^+-tree による検索 (Baseline-FDW) と SAS，および相関係数を考慮した SAS+による検索を比較し，エラー率 [18] および問合せに要する実行時間を用いて検証する．また，単一表ならびに結合を含む表に対して各種シノプシスを埋込み索引のリーフノード数，中間ノード数を計測し，シノプシス埋込みによるストレージオーバーヘッドを検証する．なお，全ての実験は，OS は Amazon Linux release 2 (Karoo)，PosgreSQL はバージョン 14.4 を用い，2CPU コア，14GB メモリ，640GB ストレージの環境下で行なった．

5.2 実 験 結 果

初めに，PostgreSQL がもつ既存の B^+-tree を用いた正解値を求める問合せ処理 (PG-orig) と，外部表 (FDW) に対して従来の B^+-tree を用いた正解値を検索する方法 (Baseline-FDW (Nosyn)) と，これらに対して，B^+-tree 索引に **Syn2** のシノプシスを保持し，SAS を FDW(Approximate_FDW) を用いて実行する方式 (SAS) と相関関係を考慮した方式 (SAS+) の実



(a) 実行時間



(b) エラー率

図 3: 単一表に対する近似問合せ処理

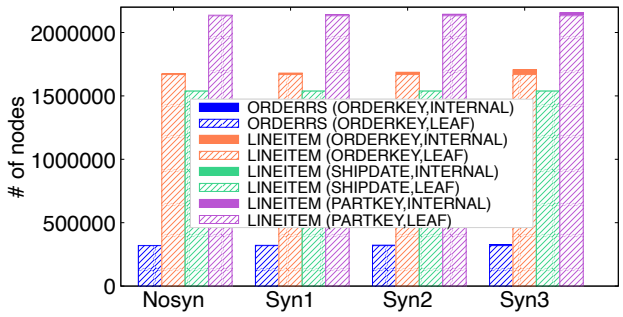


図 4: 単一表に対する索引へのシノプシス埋込みによるストレージオーバーヘッド

行性能を比較検証する．

図 3(a) に単一表に対するクエリ (Q1-Q4) における問合せ実行時間を比較し示す．シノプシスを利用することによって，SAS では約 97.4%，属性間の相関を考慮した SAS+で 94.6% 高速であることが確認できる．また，図 3(b) が示す通り，PostgreSQL に比べ，エラー率は無視できる程度であることが確認できる．本結果は，本実験で用いたデータセットが属性間にほぼ相関を持たず一様に分布している為であると考えられる．

次に，単一表に対する索引へのシノプシスを埋込む事による，索引容量オーバーヘッドを検証する．図 4 に，各索引およびシノプシスをもつデータ構造を保有することによるストレージオーバーヘッドを示す．[4] らの示した結果と同様であり，リーフノードが中間ノードに比べ圧倒的にサイズが大きいため，中間ノードはわずかに増加はしているものの，全体のノード数は，シノプシスによるストレージオーバーヘッドは **Syn3** でも約 1.83% 程度であることを示している．

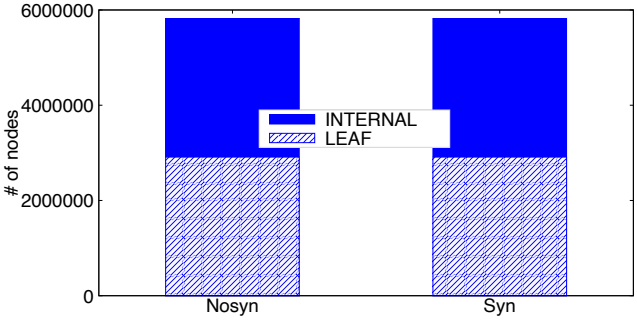


図 5: 結合表に対する索引へのシノプシス埋込みによるストレージオーバーヘッド

さらに，結合を含む問合せに対する結合表に対して索引を生成し，シノプシスを埋込みによる容量オーバーヘッドを検証する．図 5 に示すように，単一表と異なり，複数の検索キーに対応させるため，KD-tree を用いており，リーフページにデータページへのアクセスポインタのみを保有し，問合せに対する結合表をデータページに保有するよう実装したことで，中間ノードとリーフノードのノード数の差分は少ないが，全体のノード数を比較すると，単一表に対する索引同様，シノプシスによるオーバーヘッドは誤差の範囲に留まっている．これより，結合を含む問い合わせに対して，あらかじめ結合表を用意し，それに対しシノプシス索引を埋込んでもオーバーヘッドが少ないことが分かる．

6 おわりに

本論文では，単一表を対象とした問合せに加え，結合を含む問合せに対すして，シノプシスと呼ばれる付加情報を埋込んだ索引を利用し，集合値の問合せに対する近似解を高速に検索する方式を提案し検証した．実験では，単一表を対象とした問合せに対しては，従来の B^+-tree を要する PostgreSQL の検索に比べ，シノプシスを埋め込んだ索引を用いた集合値の計算処理が最大 97.4% 高速である事を確認した．また，単一表に対しても，結合を含む問合せに対してあらかじめ結合表を用意しそれに対してシノプシス埋込んだ場合においても，シノプシスを埋込んだ索引によるストレージオーバーヘッドが無視できる程度であることと，今後は，結合を含む多様な問合せやリアルデータを用いた評価を進める．

謝　辞

本研究の一部は，内閣府戦略的イノベーション創造プログラム（SIP）「統合型ヘルスケアシステムの構築」の助成を受けたものである．

文　献

[1] 川村晃一．売上分析．計測と制御, Vol. 28, No. 1, pp. 17–22, 1989.
[2] Ramez Elmasri. Fundamentals of database systems seventh edition. 2021.
[3] David J Abel. A b+-tree structure for large quadtrees.

Computer Vision, Graphics, and Image Processing, Vol. 27, No. 1, pp. 19–31, July 1984.

[4] Hiroki Yuasa, Kazuo Goda, and Masaru Kitsuregawa. Exploiting embedded synopsis for exact and approximate query processing. In *Proc. DEXA*, pp. 235–240, 2022.

[5] Postgressql. `https://www.postgresql.org/`. 2024-01-10 参照.

[6] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Rec.*, Vol. 26, No. 1, pp. 65–74, 1997.

[7] Imene Mami and Zohra Bellahsene. A survey of view selection methods. *SIGMOD Rec.*, Vol. 41, No. 1, pp. 20–29, 2012.

[8] Joseph M Hellerstein, Ron Avnur, Andy Chou, Christian Hidber, Chris Olston, Vijayshankar Raman, Tali Roth, and Peter J Haas. Interactive data analysis: the control project. *Computer*, Vol. 32, No. 8, pp. 51–59, 1999.

[9] Chris Jermaine, Subramanian Arumugam, Abhijit Pol, and Alin Dobra. Scalable approximate query processing with the dbo engine. *TODS*, Vol. 33, No. 4, pp. 1–54, 2008.

[10] Kai Zeng, Sameer Agarwal, Ankur Dave, Michael Armbrust, and Ion Stoica. G-ola: Generalized on-line aggregation for interactive analysis on big data. In *Proc. SIGMOD*, pp. 913–918, 2015.

[11] Niranjan Kamat, Prasanth Jayachandran, Karthik Tunga, and Arnab Nandi. Distributed and interactive cube exploration. In *Proc. ICDE*, pp. 472–483, 2014.

[12] Swarup Acharya, Phillip B Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. The aqua approximate query answering system. In *Proc. SIGMOD*, pp. 574–576, 1999.

[13] Ioannis Mytilinis, Dimitrios Tsoumakos, and Nectarios Koziris. Distributed wavelet thresholding for maximum error metrics. In *Proc. SIGMOD*, pp. 663–677. Association for Computing Machinery, 2016.

[14] Abdul Naser Sazish and Abbes Amira. An efficient architecture for HWT using sparse matrix factorisation and DA principles. In *Proc. APCCAS*, pp. 1308–1311, 2008.

[15] Graham Cormode, Antonios Deligiannakis, Minos Garofalakis, and Andrew McGregor. Probabilistic histograms for probabilistic data. *Proc. VLDB Endow.*, Vol. 2, No. 1, pp. 526–537, August 2009.

[16] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. BlinkDB: queries with bounded errors and bounded response times on very large data. In *Proc. EuroSys*, pp. 29–42, 2013.

[17] Jinglin Peng, Dongxiang Zhang, Jiannan Wang, and Jian Pei. AQP++: Connecting approximate query processing with aggregate precomputation for interactive analytics. In *Proc. SIGMOD*, pp. 1477–1492, 2018.

[18] Xi Liang, Stavros Sintos, Zechao Shang, and Sanjay Krishnan. Combining aggregation and sampling (nearly) optimally for approximate query processing. In *Proc. SIGMOD*, pp. 1129–1141, 2021.

[19] Swarup Acharya, Phillip B Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. Join synopses for approximate query answering. In *Proc. SIGMOD*, pp. 275–286, 1999.

[20] Alex Galakatos, Andrew Crotty, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. Revisiting reuse for approximate query processing. *Proc. VLDB Endow.*, Vol. 10, No. 10, pp. 1142–1153, 2017.

[21] Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. Verdictdb: Universalizing approximate query processing. In *Proc. SIGMOD*, pp. 1461–1476, 2018.

[22] postgres_fdw. `https://www.postgresql.jp/document/14/html/postgres-fdw.html`. Accessed on 02-26-2024.

[23] Tpc-h. `https://www.tpc.org/tpch/`. 2024-01-09 参照.

[24] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, Vol. 18, No. 9, pp. 509–517, 1975.