筑波大学 情報学群 情報メディア創成学類

卒業研究論文

NoSQL型データベースシステムでの実体化ビュー選択に関する研究

高木 颯汰

指導教員　古瀬 一隆 陳 漢雄

2019年1月

概要

本論文では NoSQL の一種であるドキュメント指向データベースに実体化ビューを導入する事によって問い合わせ処理を高速化する手法を提案する．ドキュメント指向データベースでは従来のリレーショナルデータベースにあったような参照型のデータ構造に加えて埋込型のデータ構造を選択できる．参照先の内容を埋め込む事によって結合処理をしなくて済むが，ファイルサイズが大きくなる傾向にあり，フラグメンテーションが発生し，逆にパフォーマンスが落ちる可能性がある．そこで本手法ではリレーショナルデータベースで実現されている実体化ビューの概念を NoSQL にも応用する事で，問い合わせ処理を自動的に高速化する．具体的には，頻繁に問い合わせのある結合処理や集計処理を自動的に検知してその部分のみ予め実体化することでデータベースアクセスの高速化を実現している．実体化する箇所の選択を自動化することにより，データベースシステム管理者が行なっていた作業を簡略化し，客観的で正確な実体化ビュー選択が可能となる．

目 次

図 目 次

表 目 次

第1章　はじめに

数年前までは主要なデータストアとして，リレーショナルデータベースがあげられることがほとんどであった．それは多くの開発者が SQL に慣れ親しんでおり，正規化されたデータモデル，トランザクションの必要性，耐久性のあるストレージエンジンが提供する保証を受けられるからである [1]．しかし近年高いスケーラビリティや大量なデータ処理が得意であることなどから NoSQL に対する需要が急激に増えている．

例えば NoSQL の一種のドキュメント指向データベースはデータベースの構造を表すスキーマを定義する必要がなく，大量なデータを事前準備なしで格納することができる．従来のリレーショナルデータベースにあったような参照型のデータ構造に加えて埋込型のデータ構造を選択できる．型宣言の必要のないスクリプト言語と相性が良いことなども合間って，プロトタイプを高速に開発することが求められるビジネスの現場で採用されることが増えている [2].

一方でドキュメント指向データベースの特徴とも言える階層的なデータモデルが更新処理速度の低下やデータ参照の柔軟性を低下を招くことがある．これを防ぐためにはドキュメントに階層的に埋め込むフィールドを適切に選択する必要がある．本論文ではこの選択の自動化し，ドキュメント指向データベースのデータモデルのチューニングを行い，データアクセスを高速化する．

本論文の構成は以下の通りである．まず，第 2 章において関連研究について紹介する．次に，第 3 章において本研究の提案手法について説明をし，第 4 章にて提案手法に関する実験を行い，提案手法の有用性を確める．第 5 章において実験の結果と考察を述べ，最後に第 6 章において本論文のまとめと今後の課題を示す．

第2章　関連技術

2.1　Materialized View

リレーショナルデータベースにおけるビュー (view) はリレーショナルデータモデルの発案者であるコッドにより導入された概念であり [3]，1 つ以上の表（または他のビュー）から任意のデータを選択し，それらを表したものである．ビューの実体はデータを持たない SQL 文であり，実行された際にはバックグラウンドで SELECT 処理が毎回実行される．それに対して実体化ビュー（Materialized View）はビューと同じく複数の表の結合処理や集計処理を行うが，その結果を実際のテーブルに保持する．保持された実体化ビューは元のテーブルが更新されるたびに更新される．そのため，最新でない状態を取得する可能性はあるが，結合処理が必要ないため効率的なアクセスが可能になる．その一方，更新処理が増加するので実体化ビュー化する部分の選択は慎重に行う必要があり，この作業を自動化する研究が行われている [4]．図 2.1 は 1 対 1 のデータモデルの実体化ビューを図示したものである．

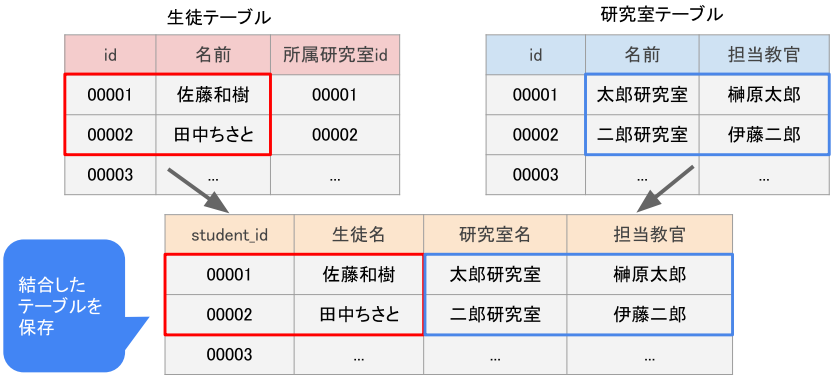

図 2.1: 実体化ビュー

多対多の結合を表す際には中間テーブルを用意してそれぞれのテーブルからレコードを結合する．その際の流れを図 2.2 に示す．履修中間テーブルに生徒 id と授業 id を格納している．id が 00001 の生徒が履修している授業を取得する際には履修中間テーブルの student_id が 00001 のレコードを取得し，付随する class_id を用いて授業の情報を取得する．結合元のテーブルのレコード数が無数にある場合には結合元テーブルでの検索時間が増加し，結合元のテーブル

のレコード数の増加と共に中間テーブルのレコード数も増える傾向にあるので，中間テーブルでの検索時間も増加する．

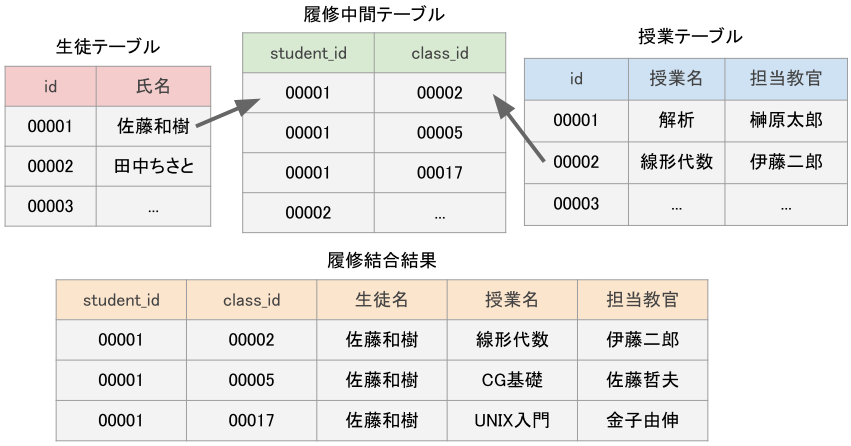

図 2.2: 多対多の結合

Materialized View を用いた際のメリットとして，結合処理や集計処理が不要になり高速化することに加え，問い合わせに用いる SQL 文が簡素化することが挙げられる．表 2.1 は図 2.2 の生徒テーブルを student，授業テーブルを class，履修中間テーブルを course_selection，生徒の履修結合結果を得るために作成した Materialized View を mv_course_selection とした際に，id が 00001 の生徒が履修している授業を取得する SQL 文を Materialized View を使用する場合としない場合を比較したものである．

表 2.1: 生徒テーブルと授業テーブルを結合する SQL

Materialized View 不使用	SELECT student.id AS student_id, class.id AS class_id, student.name AS student_name, class.name AS class_name, class.teacher AS teacher FROM student, class, course_selection WHERE course_selection.student_id = 00001 AND course_selection.student_id = student.id AND course_selection.class_id = class.id;
Materialized View 使用	SELECT student_id, class_id, student_name, class_name, teacher FROM mv_course_selection WHERE student_id = 00001;

2.2 NoSQL

NoSQL とは，“Not only SQL”の略称であり，SQL を用いないデータベースの総称を表す [5]．情報の大規模化が進み，ビッグデータと呼ばれる概念が登場すると共に，構造が複雑な様々なデータが登場するようになった．NoSQL は，そのような複雑な構造のデータに柔軟に対応し処理を行うことができる．Google や Amazon，Twitter など，世界的規模を誇る企業が NoSQL データベースを利用しており，今後ますますデータの大規模化が進む現代社会において，重要な役割を果たすデータベースである [5]．NoSQL データベースはキー・バリュー型，カラム指向型，ドキュメント指向型，グラフ型の 4 種類の型に大別することができる．キー・バリュー型は，インデックスであるキーと値であるバリューのペアでデータが構成され，キーを指定することでデータを呼び出すことができる．カラム指向型は行に対してキーが付され，それが複数の列 (カラム) に対応する形のデータモデルである．ドキュメント指向型は，JSON や XML などの形式で記述されたドキュメントの形でデータを扱うデータモデルである．グラフ型は，データ間の関係性をグラフの構造で表すデータモデルである [5]．

2.3 MongoDB

MongoDB とは，JSON や XML などの形式で記述されたドキュメント指向型のデータを扱う NoSQL データベースの代表的なものの一つである．RDB とは違い，スキーマの定義を必要としない [5][6]．また，JSON 形式のデータを扱うため，Web システムなどに利用しやすい．MongoDB においては，RDB のテーブルにあたるものとしてコレクション，RDB の行にあたるものとしてドキュメント，RDB の列にあたるものとしてフィールドというデータ構想が使われる．

MongoDB ではデータの格納に JSON をバイナリエンコーディングした BSON 形式を用いる [7]．RDBMS と MongoDB における一般的な検索のクエリを表 2.2 に，更新のクエリを表 2.3 に示す．また，MongoDB はクエリの結果を JSON 形式で返す．返却されるクエリセットの例を図 2.3 に示す．

表 2.2: RDBMS と MongoDB における検索クエリ

RDBMS	SELECT * FROM comments WHERE story = ”Next Generations”;
MongoDB	db.comments.find({ story: ”Next Generations” });

表 2.3: RDBMS と MongoDB における更新クエリ

RDBMS	UPDATE comments SET story = ”Next Generations 2” WHERE story = ”Next Generations”;

MongoDB	db.comments.update({
	story: ”Next Generations”
	}, {
	$set: {story: ”Next Generations 2”}
	});

```
{
  "_id": ObjectId("507f1f77bcf86cd799439011"),
  "speak": {
    "speaker": "名無しさん",
    "comment": "この話はとても面白い."
  },
  "story": "Next Generations",
  "created_at": "2015-04-23 03:07:27",
  "updated_at": "2015-04-23 03:07:27"
}
```

図 2.3: MongoDB から返されるクエリセットの例

2.4 Restful API

REST とは Roy Fielding が提唱した概念であり [8],“REpresentational State Transfer”の略である．分散システムにおける複数のソフトウェアを連携させるのに適した考え方であり，やりとりされる情報はそれ自体で完結して解釈できるステートレス性，全てのリソースが一意的なアドレスを持つアドレス可能性，他の基盤的な機能を用いずに別の情報や状態を含むことで他のリソースを参照できる持続性，HTTP メソッド（“GET”や“POST”など）の統一インターフェースを提供していることなどの原則から成る．REST の原則に則り構築された HTTP の呼び出しインターフェースを RESTful API と呼ぶ．本論文では RESTful API をミドルウェアに実装し実験を行う．

第3章　提案手法

3.1　ドキュメント指向型データベースにおける実体化について

ドキュメント指向型データベースの特徴として埋め込み（embed）がある．従来の RDB では複数の表による 1 対 1 や 1 対多，多対多の関係を表す際に，参照先のプライマリーキーのみを保存して SELECT される際に結合処理を行う．それに対してドキュメント指向型データベースでは参照先の実データを参照元に埋め込むことができ，これによって結合処理を省くことができる．埋め込み先が複数の場合には更新処理が増加し，従来の参照型に比べてデータアクセスの柔軟性が損なわれるというデメリットがある [1]. 図 3.1 はドキュメント指向型データベースの参照型を，図 3.2 は図 3.1 のデータを埋込型で表した図である．

また．図 3.1 と図 3.2 のドキュメントを得るためのクエリを表 3.1 に示す．参照型のクエリでは MongoDB の aggregate（集計）メソッドを用いて参照先のドキュメントを結合している．まず，”$match”演算子を用いて id が 12345 である people ドキュメントをフィルタリングし，”$lookup”演算子で参照先のドキュメントと結合している．”$lookup”演算子では参照先のコレクションを”from”で，参照元の id のフィールド名を”localField”で，参照先コレクション内での id のフィールド名を”foreignField”で定義して結合処理を行なっている．埋込型のクエリでは families ドキュメントが埋め込まれているので，find メソッドを用いて id が 12345 である people ドキュメントをフィルタリングし取得するだけで埋め込まれている families ドキュメントも取得することができる．

id が 123456 の families ドキュメントを更新するクエリを表 3.2 に示す．両者とも MongoDB の update メソッドを用いている．第一引数には検索内容，第二引数には置き換えるドキュメントを指定している．”$set”演算子を用いることで第二引数で指定しているフィールド以外のデータを保持しつつ，更新処理を行うことができる．参照型は families コレクションに対しての 1 度の更新処理だけで終了するが，埋込型に関しては埋め込まれているコレクション全てに対してクエリを実行する必要がある．

表 3.1: MongoDB における参照型データモデルと埋込型データモデルでの検索クエリ

参照型	db.people.aggregate([{$match: {ID: 12345}}, {$lookup: { from: ”families”, localField: ”familyID”, foreignField: ”ID”}}]);
埋込型	db.people.find({ID: 12345});

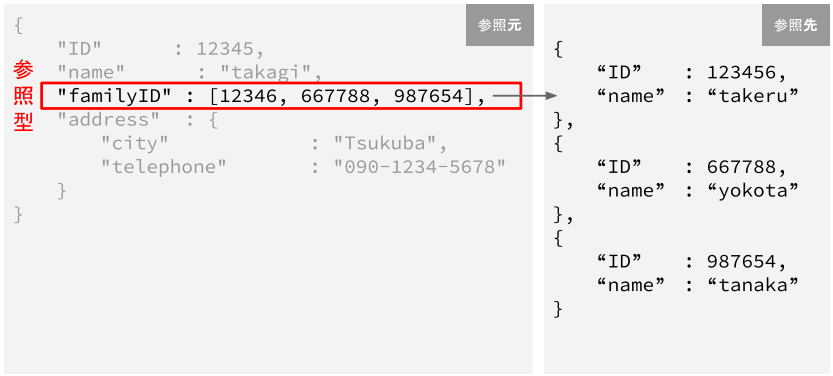

図 3.1: 参照型

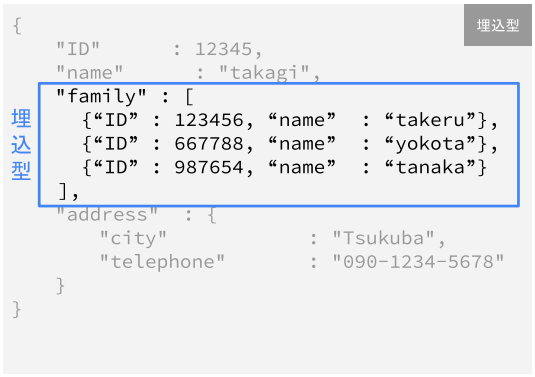

図 3.2: 埋込型

表 3.2: MongoDB における参照型データモデルと埋込型データモデルでの更新クエリ

参照型	db.families.update({ ID: 123456},{ $set: { name: ”takebayashi”}});
埋込型	db.people.update({”family.ID”: 12345}, { $set: {family.$.name: ”takebayashi”}});

全てのドキュメントを埋め込み型として保存すると埋め込み先のドキュメントの更新処理が増え，著しく更新時間が増加する為，データモデルとして最適とは言えない．埋め込み型のデータモデルとして保存するコレクションを最適に選択し，データモデルを最適化することがドキュメント指向型データベースを高速に使用することに繋がる．本論文ではこの実体化するコレクションの選択を自動化する．

3.2 提案手法の構成

本論文ではドキュメント指向型データベースの埋込型データモデルをリレーショナルデータベースの実体化ビューと置き換えて考える．ドキュメント指向型データベースのよくアクセスされる部分や処理速度がネックとなっている部分を埋込型として別コレクションに保持することで“実体化ビュー作成”，どの部分を埋込型にするかの判断を“実体化ビュー選択”とする．図 3.3 は本論文での“実体化ビュー作成”を示したものである．

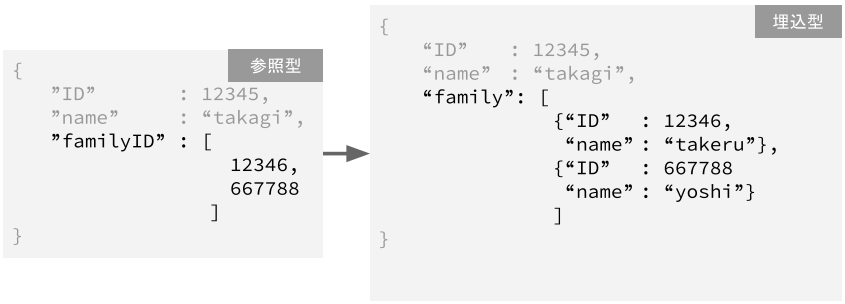

図 3.3: 参照型から埋込型への書き換え

実装システムについて図 3.4 に示す．ユーザーからのデータアクセスから実体化ビュー作成までの流れを図 3.4 を用いて説明する．図 3.4-①まずユーザーがアプリケーションからミドルウェアに対してデータアクセスの要求する．図 3.4-②ミドルウェアでは，頻繁にアクセスされるドキュメントを分析するために，クエリに関するログを残す．図 3.4-③次に MongoDB に対してクエリを発行する．図 3.4-④ MongoDB から返ってきたクエリセットをアプリケーションに返却する．図 3.4-⑤クエリログを解析し，ボトルネックとなっているところや呼び出し回数の多い条件の実体化ビューを作成する．図 3.4-⑥実体化したドキュメントに関してログに記録する．

実体化した後のデータアクセスの流れを図 3.5 に示す．図 3.5-①’アプリケーションからデータベースにアクセスがあった場合，まずログからアクセスされたデータが実体化されて

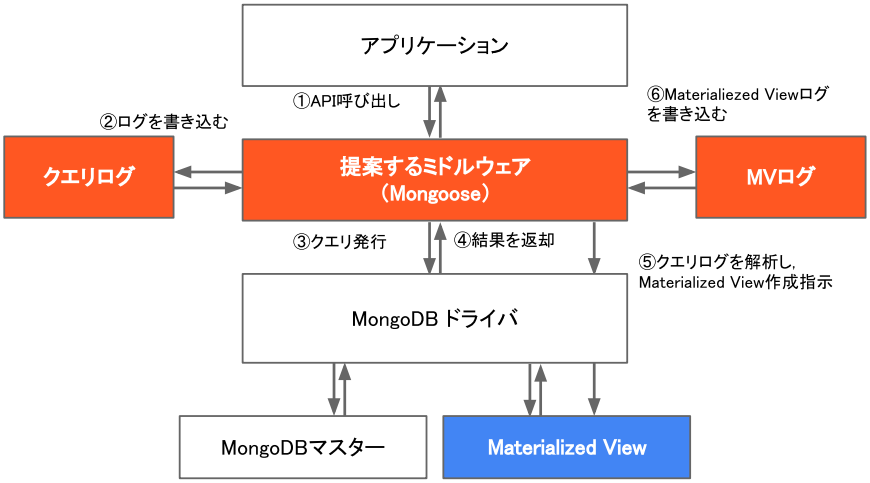

図 3.4: 提案ミドルウェア（実体化前）

いるか判定する．図 3.5-②’実体化されている場合はクエリを書き換えて実体化ビューから結果を取得する．図 3.5-③’アプリケーションに結果を返す際には元のクエリに合うように適宜変換する．

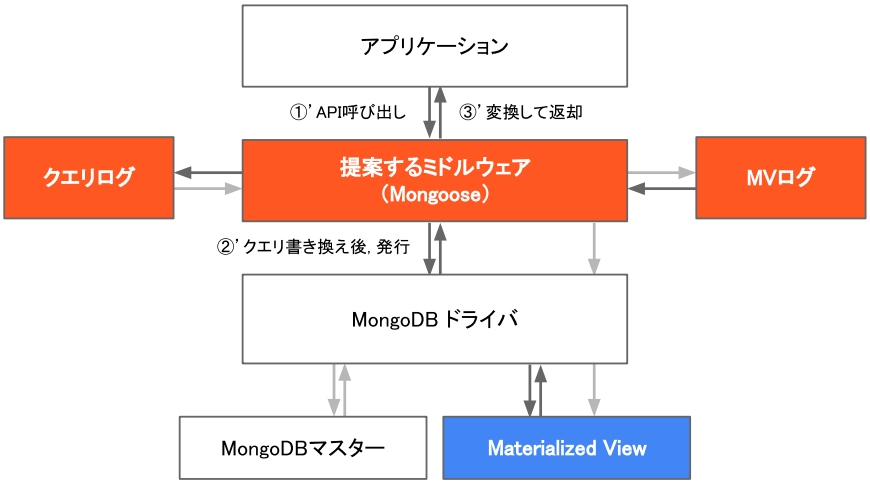

図 3.5: 提案ミドルウェア（実体化後）

次に具体的なクエリを用いて実装システムの流れを説明する．まだ実体化ビューが作成されていない，図 3.6 に示すような people コレクションと families コレクションに対して表 3.3 のクエリが順に実行されたとする．クエリ 1.1 が実行された際にはまず MV ログを確認し，people コレクションが実体化されているか確認する．実体化されていないので，クエリをそのまま MongoDB のドライバに発行し，アプリケーションに結果を返す．その際にクエリロ

グを記録する．各クエリに対するクエリログを表 3.4 に示す．クエリ 1.2，1.3，1.4 も同様に people コレクションが実体化されていないので，そのままクエリが実行される．

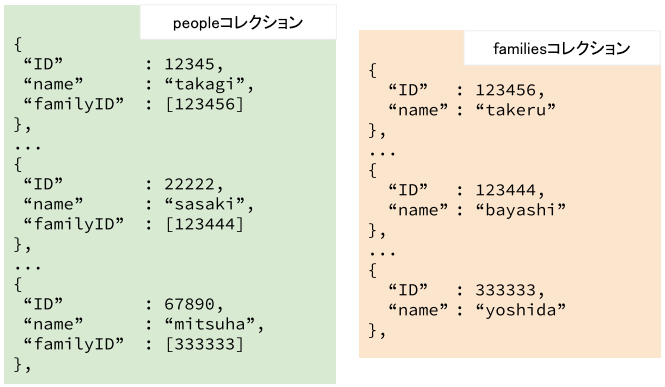

図 3.6: people コレクションと familes コレクションの構成

表 3.3: ユーザークエリ 1

クエリ 1.1	db.people.aggregate([{$match: {ID: 12345}}, {$lookup: { from: ”families”, localField: ”familyID”, foreignField: ”ID”}}]);
クエリ 1.2	db.people.aggregate([{$match: {ID: 22222}}, {$lookup: { from: ”families”, localField: ”familyID”, foreignField: ”ID”}}]);
クエリ 1.3	db.people.aggregate([{$match: {ID: 67890}}, {$lookup: { from: ”families”, localField: ”familyID”, foreignField: ”ID”}}]);
クエリ 1.4	db.families.update({ ID: 123456}, { $set: { name: ”takebayashi”} });

クエリ 1.4 が実行された後に自動実体化のプロセス 1 が実行されたとする．実際の運用では集計やコレクションの作成などを行う可能性があるので，定期的にアクセスの少ない深夜帯などに実行されることが望ましい．このプロセス 1 では前述で記録したクエリログを元に実体化するコレクションを判定する．このプロセス 1 の詳細は 3.3 節で説明する．ここでは 3.3 節において people コレクションが実体化されたとする．その際に作成された people の実体化

表 3.4: ユーザークエリ 1 で記録されるクエリログ

クエリログ 1.1	{”method” : ”aggregate”, ”lookup” : [”families”], ”collection_name” : ”people”, ”query” : ”{$match: {ID: 12345}}”, ”elapsed_time” : 55.66362400003709, ”is_rewrited” : false}
クエリログ 1.2	{”method” : ”aggregate”, ”lookup” : [”families”], ”collection_name” : ”people”, ”query” : ”{$match: {ID: 22222}}”, ”elapsed_time” : 42.66362400003732, ”is_rewrited” : false}
クエリログ 1.3	{”method” : ”aggregate”, ”lookup” : [”families”], ”collection_name” : ”people”, ”query” : ”{$match: {ID: 67890}}”, ”elapsed_time” : 57.26362400003767, ”is_rewrited” : false}
クエリログ 1.4	{”method” : ”update”, ”collection_name” : ”families”, ”query” : ”{ $set: { name: ”takebayashi”}}}”, ”elapsed_time” : 70.36362400003744, ”is_rewrited” : false}

ビューのコレクション名を”mvpeople”とし，ドキュメントの内容を図 3.7 に，同時に記録される MV ログを表 3.5 に示す．実体化したコレクション名を”original_coll”で，参照先のコレクション名を”lookup”で，実体化ビューの削除フラグを”is_deleted”で表している．

次に表 3.6 のクエリが順に実行されたとする．クエリ 2.1 が実行された際にはまず MV ログを取得し，people コレクションが実体化されているか確認する．表 3.5 から，people コレクションに families コレクションが埋め込まれた実体化ビューが存在するので，クエリを書き換えて MongoDB ドライバにクエリを発行する．書き換え前と書き換え後のクエリの比較を表 3.7 に示す．クエリ 2.2 では families コレクションを更新しているが，表 3.5 から families ドキュメントが people コレクションに実体化ビューとして埋め込まれていることが分かるので，ID が 123444 の families ドキュメントを保持している people ドキュメントに関しても更新処理を行なっている．クエリ 2.3 では people コレクションの更新処理を行なっている．people コレクションは実体化されているので実体化ビューとオリジナルのコレクションの更新を同

図 3.7: people コレクションの実体化ビュー

表 3.5: MV ログ 1

MV ログ 1	{”original_coll” : ”people”, ”lookup” : [”families”], ”is_deleted” : false, }

時に行う．

クエリ 2.3 が実行された後に自動実体化のプロセス 2 が実行されたとする．実体化されている people コレクションに対して逆実体化するべきか，実体化されていない families コレクションに対して実体化するべきかを評価する．逆実体化に関しては 3.4 節で詳しく説明する．ここでは 3.4 節で people コレクションを逆実体化するべきだと判断されたとする．逆実体化時には該当の MV ログに対して削除フラグを立てる．その際の MV ログを表 3.9 に示す．実体化時もオリジナルのコレクションに対して更新を行なっているので，逆実体化時にコレクションの操作は発生しない．

3.3 実体化アルゴリズムについて

クエリログから実体化するコレクションを決定する条件を適切に設定することで，実体化ビュー選択を自動化することができる．この実体化条件については以下の条件が考えられる．

1. クエリログの統計
2. ドキュメント内の参照数

1 の条件は実際のクエリの検索回数・更新回数やその処理時間を元に実体化するコレクションを選定する．2 の条件ではドキュメント内の他コレクションへの参照数を元に実体化後の更

表 3.6: ユーザークエリ 2

クエリ 2.1	db.people.aggregate([{$match: {ID: 67890}}, {$lookup: { from: ”families”, localField: ”familyID”, foreignField: ”ID”}}]);
クエリ 2.2	db.families.update({ ID: 123444}, { $set: { name: ”zenbayashi”} });
クエリ 2.3	db.people.update({ ID: 12345}, { $set: { name: ”souta”} });

新処理の増加を予想し，実体化するコレクションを選定する．本論文では更新処理が用意であり，クエリに柔軟に対応できるクエリログの統計を元に実体化する条件を作成する．

3.2 節のプロセス 1 ではクエリログ 1.1 から 1.4 を集計して実体化するコレクションを決定する．提案手法では検索クエリ数と更新クエリ数の比があらかじめ決めた比を超えていた際に実体化を行う．クエリログ 1.1 から 1.4 を集計した結果を表 3.10 に示す．”is_rewrited”でクエリが実体化ビューに対してか否かを，”total_time”でクエリの合計処理時間を，”average_time”でクエリの平均処理時間を，”count_query”でクエリ数を表している．ここでは例として検索クエリ数/更新クエリ数比が 20 を超えていた際に実体化すると定義すると，プロセス 1 時点の検索クエリ数/更新クエリ数比は people コレクションが 3/0，families コレクションが 0/1 なので，people コレクションを実体化する．

3.4 逆実体化アルゴリズムについて

クエリのログを元に実体化した場合，クエリの傾向が変わることで実体化していない状態の方が望ましくなる可能性や，コレクションの特性によっては実体化によるメリットがデメリットより少ない可能性がある．そのような事を防ぐために，定期的に実体化したコレクションに対してもメンテナンスを行い，場合によっては実体化したコレクションをオリジナルのデータモデルに戻すことが必要である．本論文では実体化したコレクションのデータモデルを元に戻す事を逆実体化と定義する．この逆実体化に対しても実体化同様，適切な逆実体化条件を設定する必要がある．

表 3.7: 書き換え後のユーザークエリ 2

	書き換え前のクエリ	書き換え後のクエリ
クエリ 2.1	db.people.aggregate([{$match: {ID: 67890}}, {$lookup: { from: ”families”, localField: ”familyID”, foreignField: ”ID”}}]);	db.people.find([{ID: 67890}]);
クエリ 2.2	db.families.update({ ID: 123444}, { $set: { name: ”zenbayashi”} });	db.families.update({ ID: 123444}, { $set: { name: ”zenbayashi”} }); db.mvpeople.update({ family.ID: 123444}, { $set: { family.$.name: ”zenbayashi”} });
クエリ 2.3	db.people.update({ ID: 12345}, { $set: { name: ”souta”} });	db.mvpeople.update({ ID: 12345}, { $set: { name: ”souta”} }); db.people.update({ ID: 12345}, { $set: { name: ”souta”} });

本論文ではクエリのログの統計から実体化後の検索処理時間と更新処理時間を比較し，各コレクションに対して逆実体化の必要性を確認する．

3.2 節のプロセス 2 では実体化されているコレクションのうち，クエリログ 1.1 から 1.4 とクエリログ 2.1 から 2.3 を集計して逆実体化するコレクションを決定する．クエリログ 1.1 から 2.3 を集計した結果を表 3.11 に示す．”is_rewrited”が”true”であるオブジェクトが実体化後の集計結果である．people コレクションの実体化後の検索処理累計時間が 37ms，更新処理累計時間が 320ms で，更新処理累計時間の方が長いので逆実体化を行う．

表 3.8: ユーザークエリ 2 で記録されるクエリログ

クエリログ 2.1	{”method” : ”find”, ”collection_name” : ”people”, ”query” : ”{$match: {ID: 67890}}”, ”elapsed_time” : 37.26362400003767, ”is_rewrited” : true}
クエリログ 2.2	{”method” : ”update”, ”collection_name” : ”families”, ”query” : ”{ $set: { name: ”zenbayashi”}}}”, ”elapsed_time” : 155.36362400003744, ”is_rewrited” : true} {”method” : ”update”, ”collection_name” : ”people”, ”query” : ”{ $set: { family.name: ”zenbayashi”}}}”, ”elapsed_time” : 89.56362400003744, ”is_rewrited” : true}
クエリログ 2.3	{”method” : ”update”, ”collection_name” : ”people”, ”query” : ”{ $set: { name: ”souta”}}}”, ”elapsed_time” : 230.54762400003744, ”is_rewrited” : true}

表 3.9: 逆実体化後の MV ログ

MV ログ	{”original_coll” : ”people”, ”lookup” : [”families”], ”is_deleted” : true, }

表 3.10: プロセス 1 の際に取得されるクエリログ集計結果

コレクション	集計結果
people	{ ”_id” : { ”collection_name” : ”people”, ”method” : ”findOne”, ”is_rewrited” : false }, ”total_time” : 155.590872, ”average_time” : 51.863624, ”count_query” : 3 }
families	{ ”_id” : { ”collection_name” : ”families”, ”method” : ”update”, ”is_rewrited” : false }, ”total_time” : 70.36362400003744, ”average_time” : 70.36362400003744, ”count_query” : 1 }

表 3.11: プロセス 2 の際に取得されるクエリログ集計結果

コレクション	集計結果
people	{ ”_id” : { ”collection_name” : ”people”, ”method” : ”findOne”, ”is_rewrited” : false }, ”total_time” : 155.590872, ”average_time” : 51.863624, ”count_query” : 3 } { ”_id” : { ”collection_name” : ”people”, ”method” : ”findOne”, ”is_rewrited” : true }, ”total_time” : 37.263624, ”average_time” : 37.263624, ”count_query” : 1 } { ”_id” : { ”collection_name” : ”people”, ”method” : ”update”, ”is_rewrited” : true }, ”total_time” : 320.111248, ”average_time” : 160.055624, ”count_query” : 2 }
families	{ ”_id” : { ”collection_name” : ”families”, ”method” : ”update”, ”is_rewrited” : false }, ”total_time” : 225.727248, ”average_time” : 112.863624, ”count_query” : 2 }

第4章　実験

4.1　Mongooseについて

Mongooseとは MongoDB 用モデリングツールで，Node.js の非同期環境でうまく動作することを目的として設計されている．Mongoose を使用すれば，モデルを定義して操作することで，MongoDB のコレクション/ドキュメントを操作できる [9]．本論文では Mongoose を用いて MongoDB を操作するミドルウェアを実装する．

4.2　実験環境

実験環境に関する情報を表 4.1 に示す．

表 4.1: 実験環境

マシン	MacBook Pro (Retina, 13-inch, Early 2015)
プロセッサ	2.9 GHz Intel Core i5
メモリ	16 GB 1867 MHz DDR3
データベースシステム	Mongodb version 3.1.10

4.3　実験方法

本論文の実験で用いたコレクションは本のレビューサイトのデータベースを想定し作成した．person コレクション，story コレクション，comment コレクション，publisher コレクションがある．コレクションの構造，コレクション同士の参照に関しては図 4.1，図 4.2 に示す．story コレクションには筆者として 1 つの person ドキュメントの id を格納する．この story ドキュメントが検索された際には筆者の id を perosn コレクションから検索し，結合して結果を返す．同じようにファンとして person ドキュメントの id を配列で格納することで複数の person ドキュメントを story ドキュメントに埋め込む．実際の実験データではファンとして 1 から 100 の person ドキュメントの id を埋め込む．出版社として publisher ドキュメントの id も格納する．

コレクションの特性として，story ドキュメントに数十のドキュメントが参照されているので，参照型のデータモデルでは結合処理が多く発生する．逆に埋込型のデータモデルでは person ドキュメントの更新の際に埋込先の story ドキュメントの更新処理が増加する．

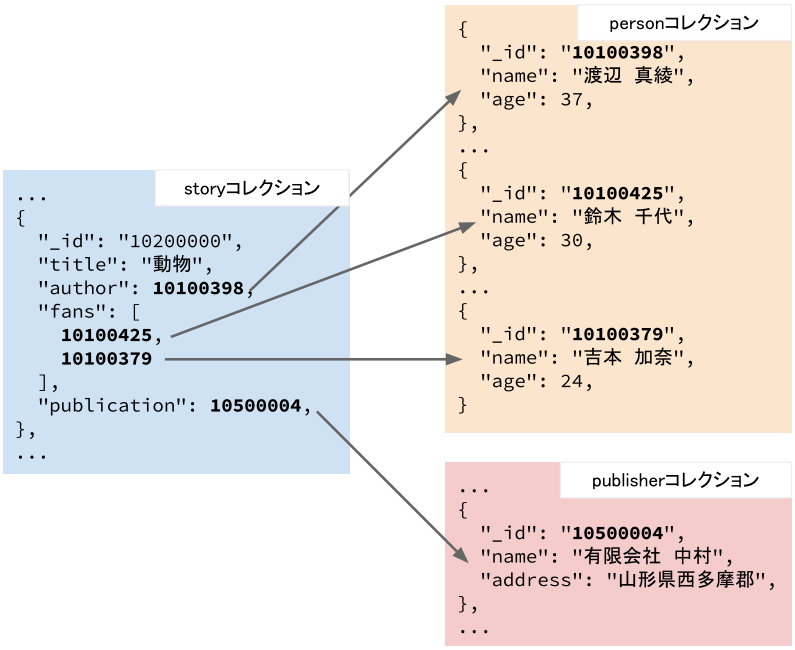

図 4.1: story コレクションから各コレクションへの参照

実験において使用するデータは全て参照型のデータモデルで挿入する．実体化していない状態で検索された場合には結合処理を行い結果を返す．実験に用いるドキュメントは Python のライブラリである faker[10] を用いて作成した．各コレクションに対して 10 万ドキュメントを作成した．各コレクションの構成を表 4.2 に示す．

表 4.2: 各コレクションの構成

コレクション名	ドキュメント数	埋め込みコレクション	埋め込みドキュメント数
person	100,000	なし	0
story	100,000	person,publisher	2〜101
publisher	100,000	なし	0
comment	100,000	person,story	2

実験では全てのコレクションが従来の参照型のデータモデルを用いたシステムと実体化条件を用いずに全てのコレクションを実体化したシステム，実体化条件を用いて適宜コレクションを実体化するシステムの 3 つのデータベースシステムを比較する．その際，実体化を用いたシステムでは検索や更新を提案ミドルウェアを用いて処理する．

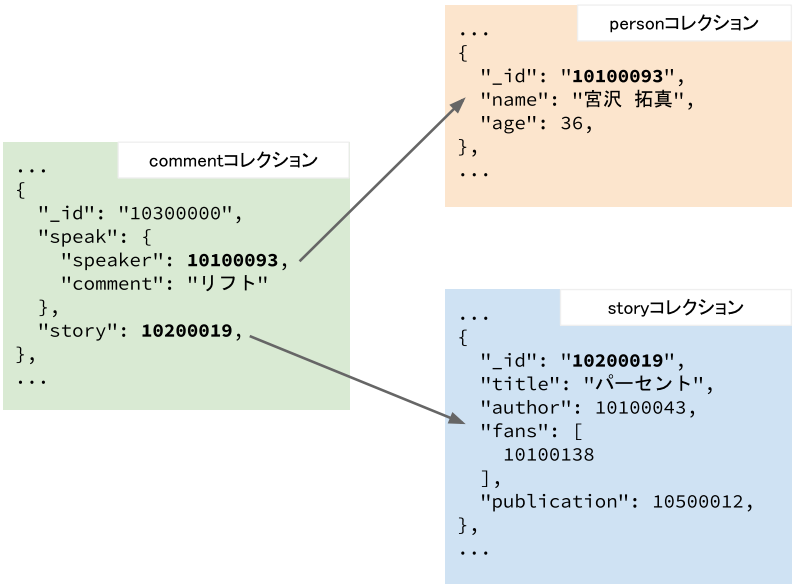

図 4.2: comment コレクションから各コレクションへの参照

この 3 つのシステムを比較する際，検索と更新の比率を変えた 5 つのクエリパターンを用いて行う．このクエリパターンを表に示す．このパターンを用いて実験 A，B，C，D，E，F，G，H を行う．それぞれのパターンに対して，使用したクエリを表 4.4 と表 4.5 で示す．MongoDB のドライバのみを用いる場合には 3.1 節で説明した様に集計機能を用いて結合処理を実現したが，Mongoose では populate メソッドでアプリケーションレベルでの結合処理を実現しており，内部的には find メソッドを複数に分けて実行し，アプリケーション側で結合処理を行う．表 4.4 のクエリは Mongoose の populate メソッドを用いて結合処理をおこなっており，comment コレクションのクエリを例とすると，id が TestID の comment ドキュメントを”speak.speaker”と”story”のフィールドを結合した状態で find する処理である．内部的には comment，person，story コレクションに対してクエリを発行している．

提案手法を用いる場合には検索・更新に関わらずクエリが 1,000 回処理された際に実体化条件を検討し，適宜実体化ビューを作成，破棄を行う．

表 4.3: 各コレクションへのクエリパターン

	検索回数比 (%)	更新回数比 (%)	検索クエリ数	更新クエリ数	総クエリ数
パターン A	99.99	0.01	19,998	2	20,000
パターン B	99.98	0.02	19,998	4	20,000
パターン C	99.96	0.03	19,994	6	20,000
パターン D	99.95	0.05	19,990	10	20,000
パターン E	99.93	0.07	19,986	14	20,000
パターン F	99.90	0.10	19,980	20	20,000
パターン G	99.80	0.20	19,960	40	20,000
パターン H	99.70	0.30	19,940	60	20,000

表 4.4: 実験で使用した検索クエリ

コレクション	クエリ
person	db.person.find({_id: testID}) .populate([]);
story	db.story.find({_id: testID}) .populate([”author”, ”fans”, ”publication”, ”comments”]);
comment	db.comment.find({_id: testID}) .populate([”speak.speaker”, ”story”]);
publisher	db.publisher.find({_id: testID}) .populate([]);

表 4.5: 実験で使用した更新クエリ

コレクション	クエリ
person	db.person.update({_id: testID}, {$set: {name: ”太郎”} });
story	db.story.update({_id: testID}, {$set: {title: ”研修資料”} });
comment	db.comment.update({_id: testID}, {$set: {”speak.comment”: ”いい天気”} });
publisher	db.publisher.update({_id: testID}, {$set: {address: ”つくば市天王台”} });

第5章　結果・考察

5.1　実験Aの結果

実験結果を図5.1，5.2，5.3に示す．図5.1，5.2は各コレクションにおけるクエリの平均処理時間を示している．図5.3は検索処理と更新処理の累計処理時間を示している．検索処理速度に関しては全実体化と提案手法が実体化なしのデータモデルより速い結果となった．一方，更新処理速度は実体化なしのデータモデルが最速であり，personコレクションに関しては全実体化と提案手法でのデータモデルが実体化なしと比べて100倍前後遅い結果となった．累計処理時間に関しては実体化なしに比べて全実体化と提案手法が3倍速い結果となった．

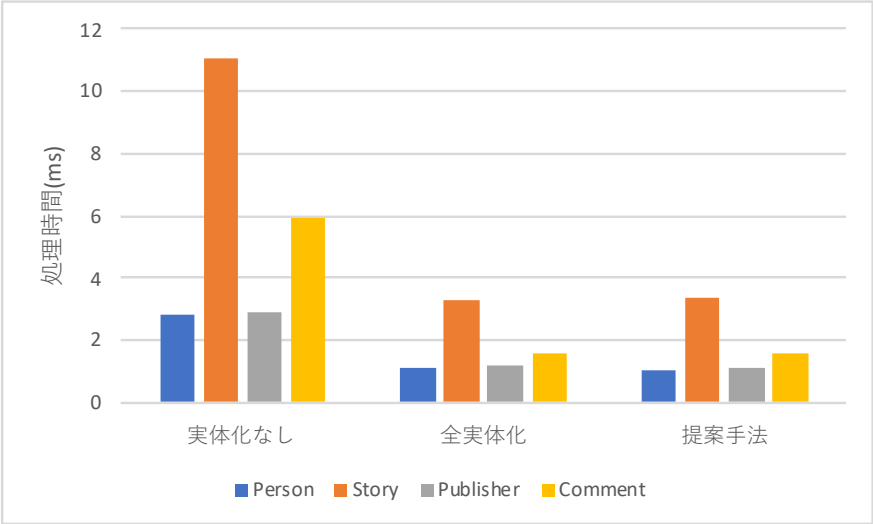

図5.1: 実験Aの各コレクションの平均検索時間

5.2　実験Bの結果

実験結果を図5.4，5.5，5.6に示す．図5.4，5.5は各コレクションにおけるクエリの平均処理時間を示している．図5.6は検索処理と更新処理の累計処理時間を示している．全実体化と比べて提案手法は検索時間はさほど変わらないが，更新時間が半分程度となったため，累計処理時間で提案手がもっとも速い結果となった．

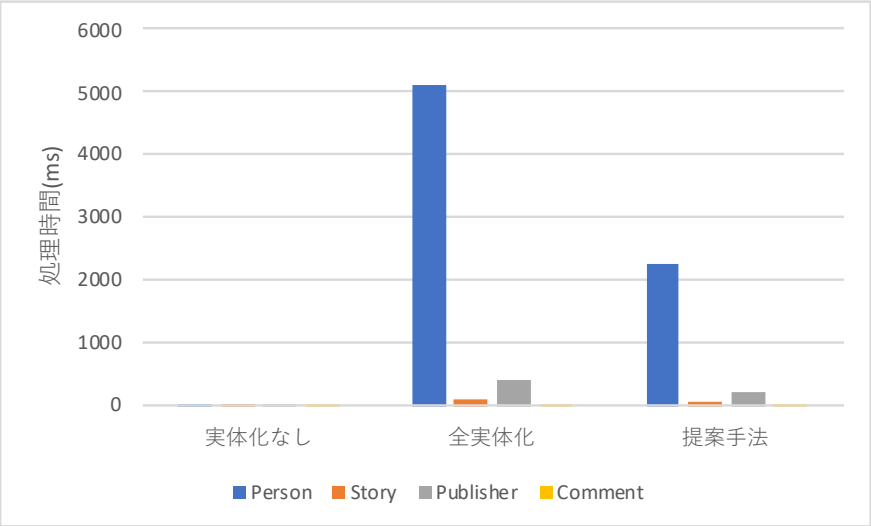

図 5.2: 実験 A の各コレクションの平均更新時間

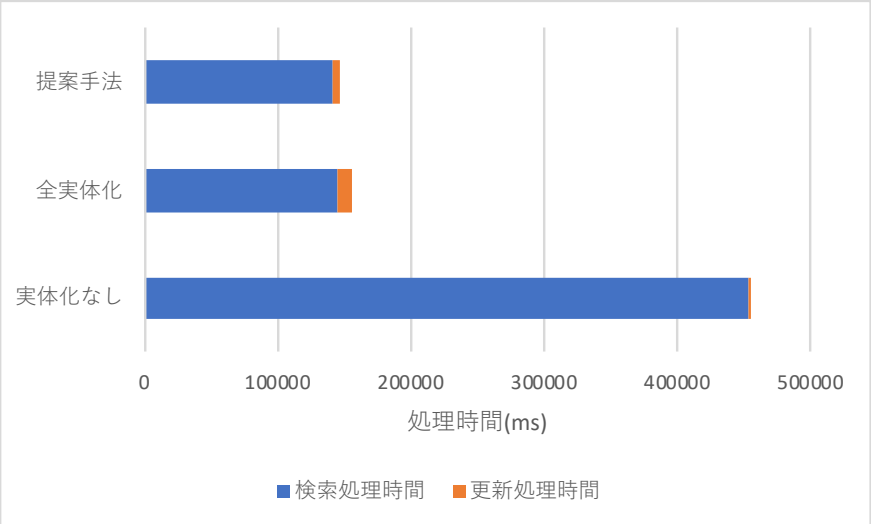

図 5.3: 実験 A の累計処理時間

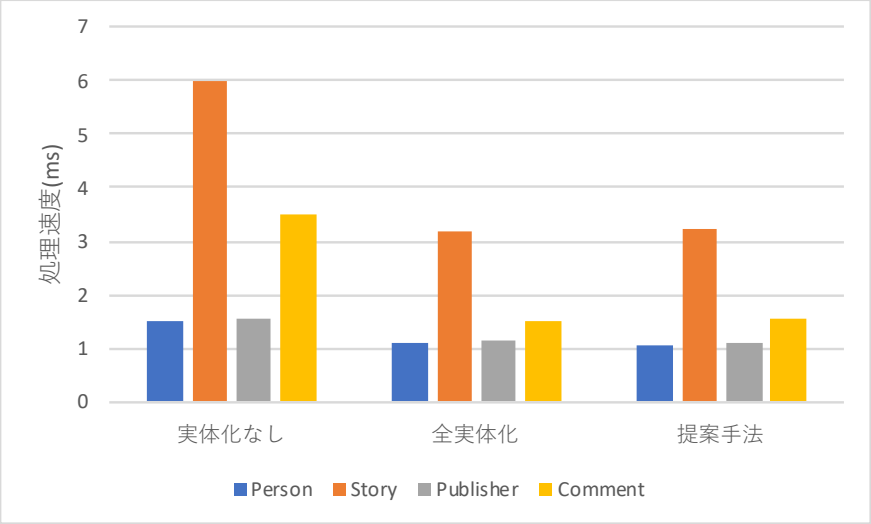

図 5.4: 実験 B の各コレクションの平均検索時間

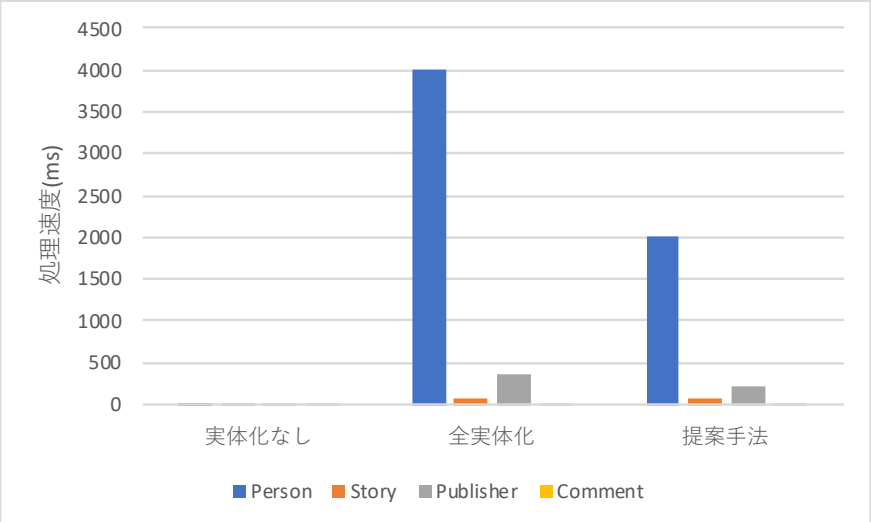

図 5.5: 実験 B の各コレクションの平均更新時間

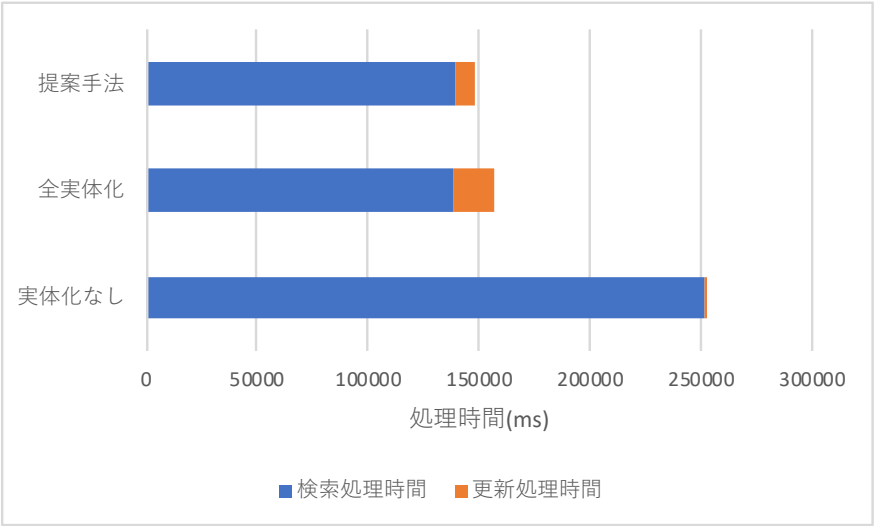

図 5.6: 実験 B の累計処理時間

5.3 実験 C の結果

実験結果を図 5.7，5.8，5.9 に示す．更新時間では提案手法が全実体化の半分程度だが，検索時間は全実体化の方が速く，累計処理時間は全実体化が一番速い結果となった．

5.4 実験 D の結果

実験結果を図 5.10，5.11，5.12 に示す．概ね実験 C と同じで，更新時間では提案手法が一番速いが，検索時間は全実体化の方が速く，累計処理時間では全実体化が一番速い結果となった．

5.5 実験 E の結果

実験結果を図 5.13，5.14，5.15 に示す．更新時間が全実体化と比べて提案手法 2 倍程度速い結果だが，検索時間は全実体化と比べて提案手法が 1.2 倍程遅く，累計処理時間では全実体化が最速という結果となった．

5.6 実験 F の結果

実験結果を図 5.16，5.17，5.18 に示す．検索時間で全実体化が提案手法よりも 1.5 倍程度速く，更新時間で提案手法が全実体化と比べて 2 倍程度速い結果となった．累計処理時間では全実体化が最速となった．

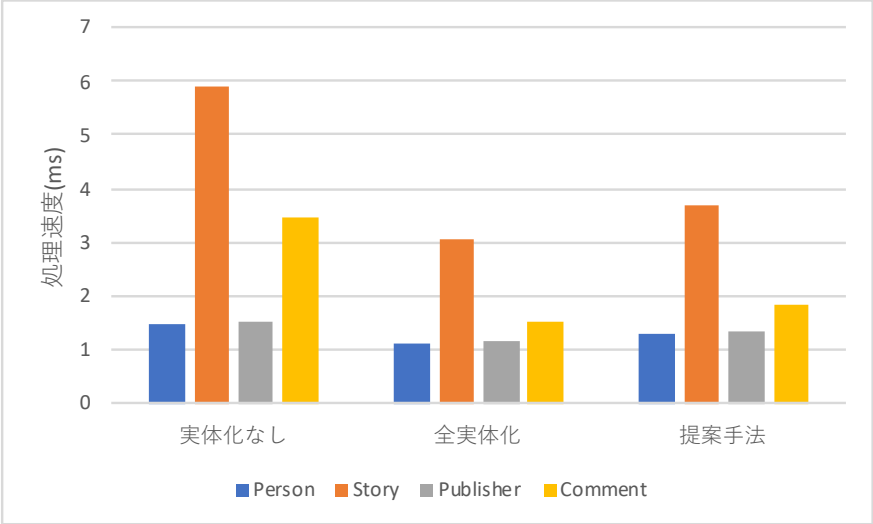

図 5.7: 実験 C の各コレクションの平均検索時間

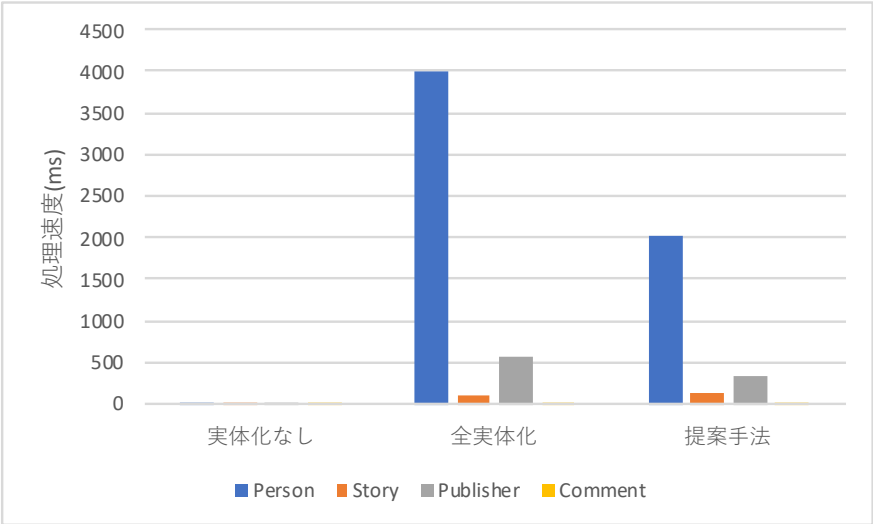

図 5.8: 実験 C の各コレクションの平均更新時間

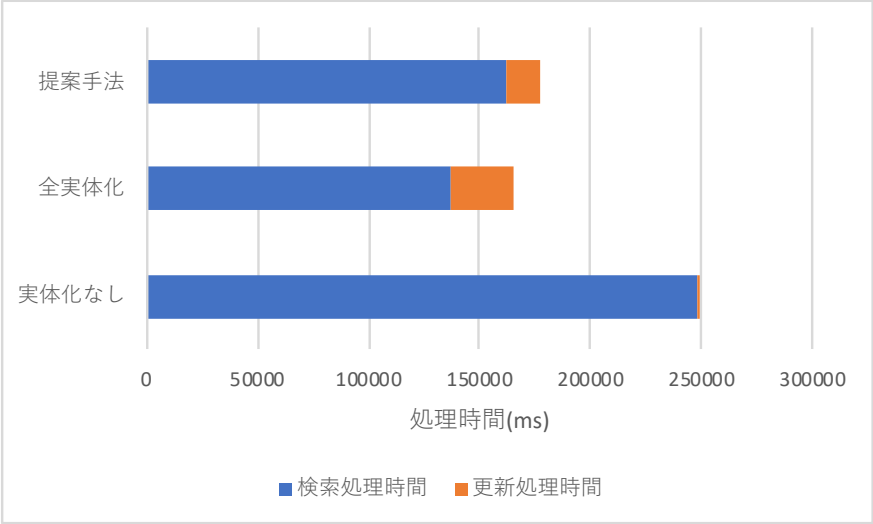

図 5.9: 実験 C の累計処理時間

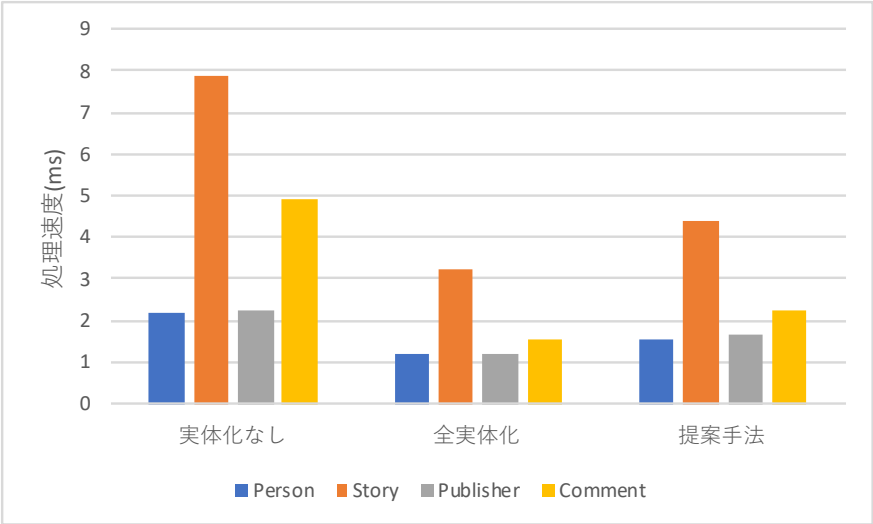

図 5.10: 実験 D の各コレクションの平均検索時間

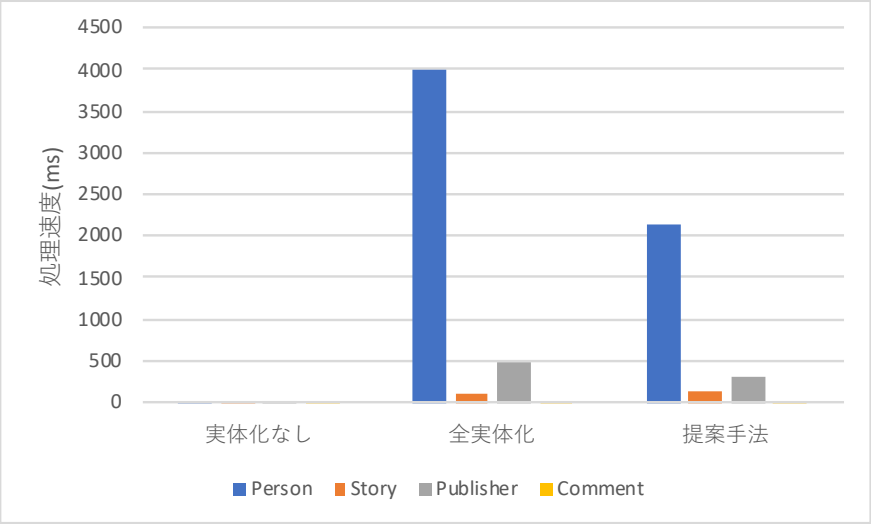

図 5.11: 実験 D の各コレクションの平均更新時間

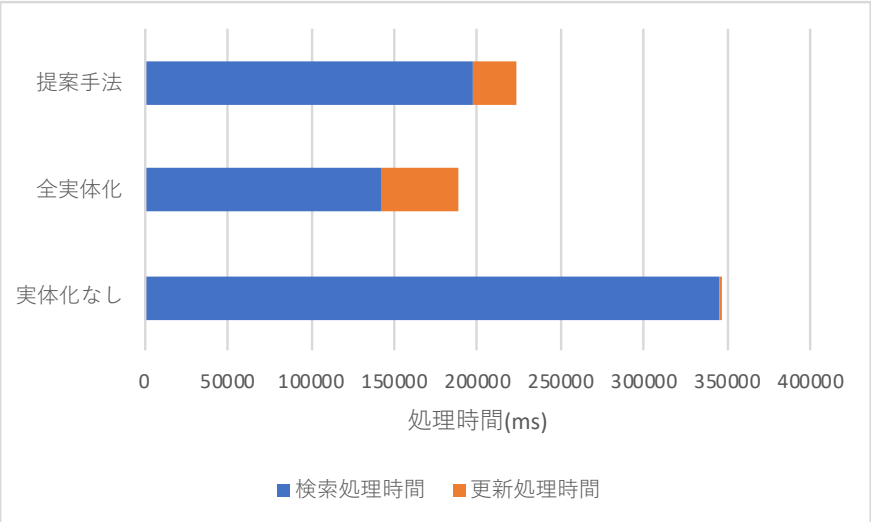

図 5.12: 実験 D の累計処理時間

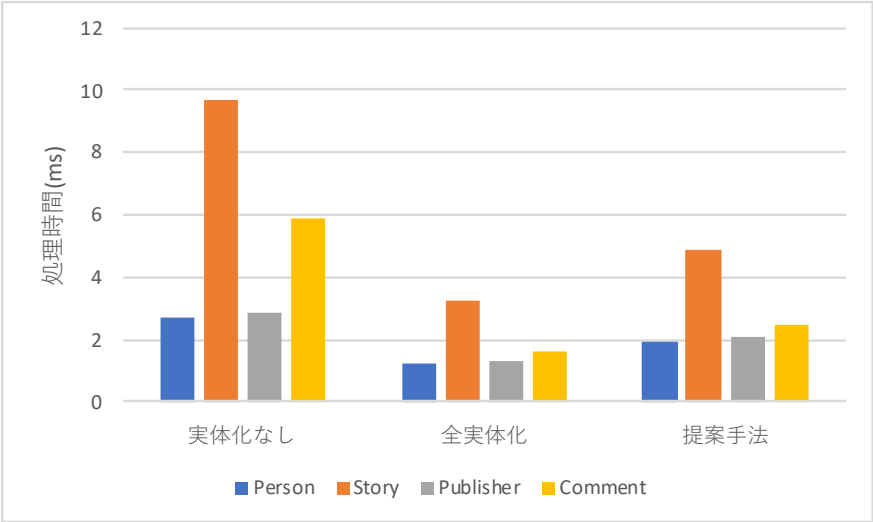

図 5.13: 実験 E の各コレクションの平均検索時間

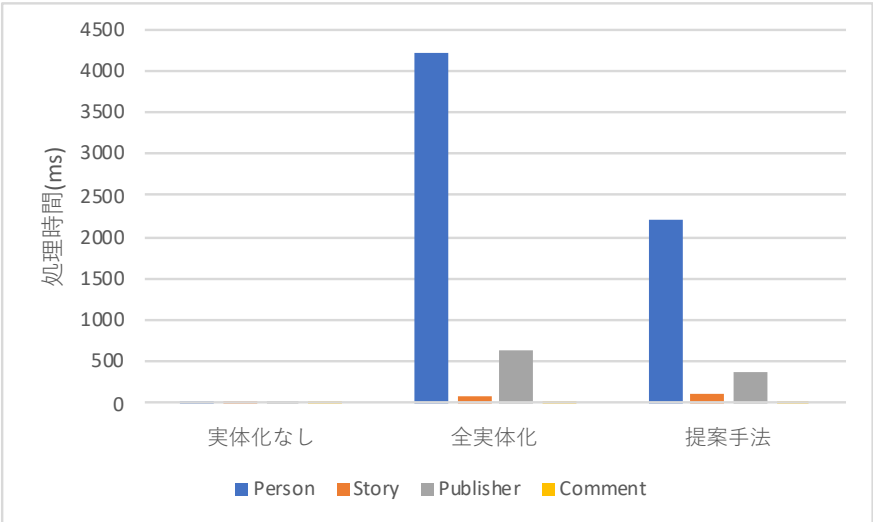

図 5.14: 実験 E の各コレクションの平均更新時間

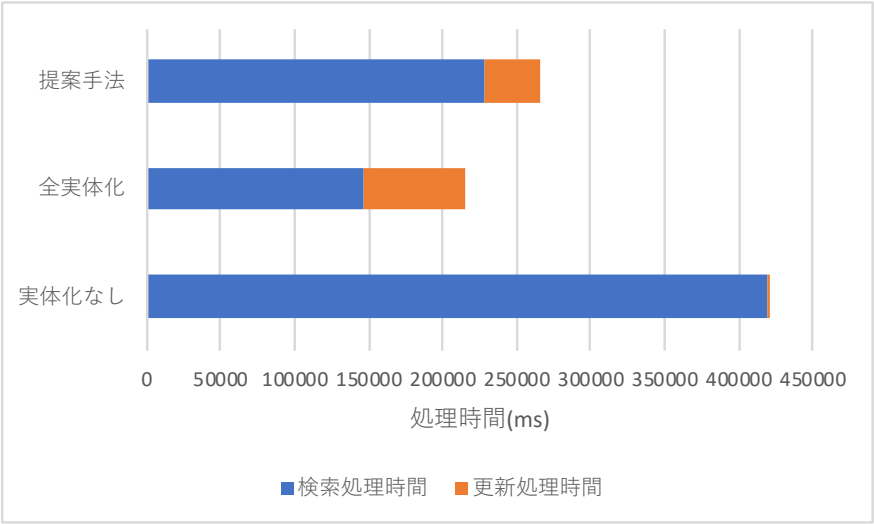

図 5.15: 実験 E の累計処理時間

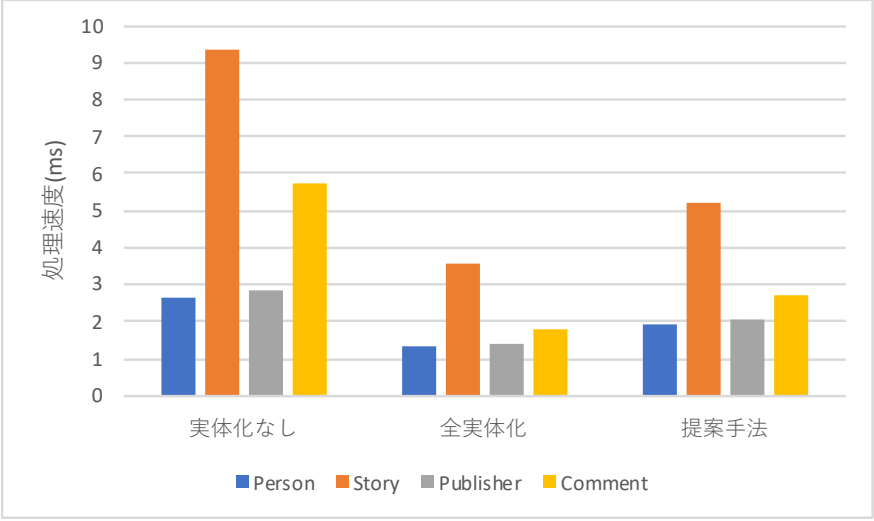

図 5.16: 実験 F の各コレクションの平均検索時間

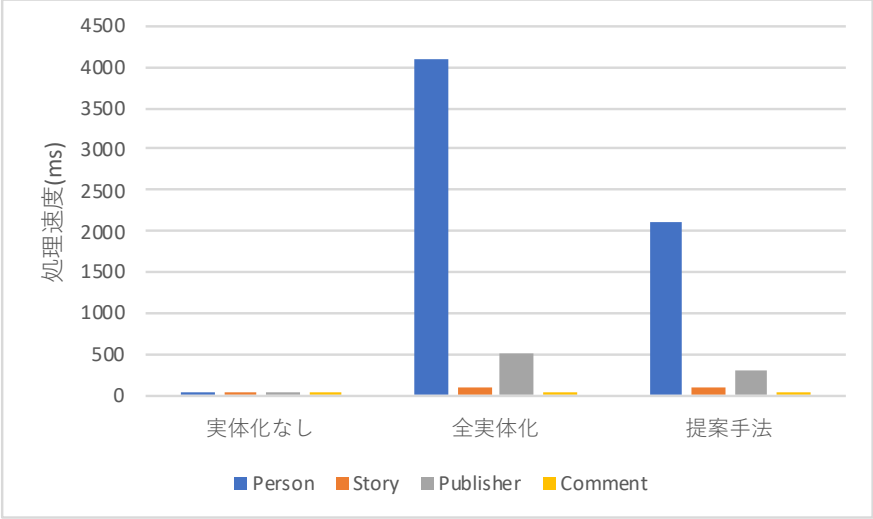

図 5.17: 実験 F の各コレクションの平均更新時間

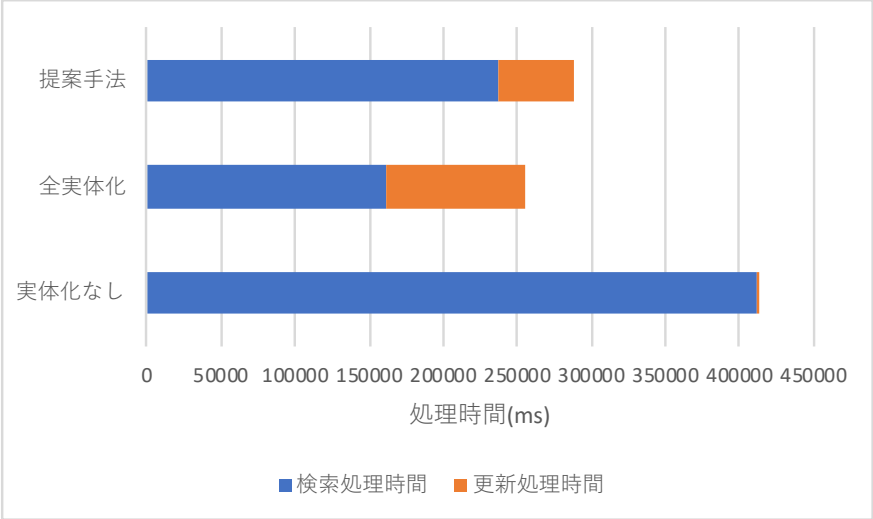

図 5.18: 実験 F の累計処理時間

5.7　実験 G の結果

実験結果を図 5.19，5.20，5.21 に示す．検索時間で全実体化が提案手法よりも 1.2 倍程度速く，更新時間で提案手法が全実体化と比べて 2 倍程度速い結果となった．累計処理時間では提案手法が最速となった．

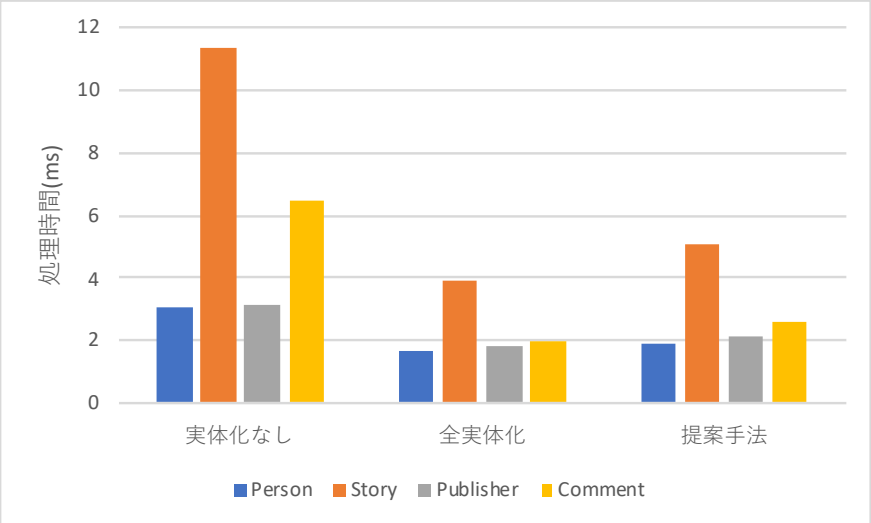

図 5.19: 実験 G の各コレクションの平均検索時間

5.8　実験 H の結果

実験結果を図 5.22，5.23，5.24 に示す．検索時間で全実体化が提案手法よりも 1.2 倍程度速く，更新時間で提案手法が全実体化と比べて 2 倍程度速い結果となった．累計処理時間では提案手法が最速となった．

5.9　実験の考察

全実験において全実体化と提案手法は実体化なしと比べて累計検索処理時間が約半減した．通常であれば元のクエリに加えて結合先のクエリを MongoDB 発行しアプリケーション層でドキュメントを結合するが，Materialized View を用いた場合には追加のクエリ発行と結合処理を省くことができる．その為，結合処理が発生する story，comment コレクションの検索時間が高速化した．また，実体化した場合には更新時間が増大した．これはオリジナルのドキュメントに加えて，そのドキュメント自体の Materialized View と，ドキュメントが埋め込まれている Materialized View を更新する必要があり．MongoDB 側に発行するクエリが増えるからである．今回の実験では story コレクションに最大 100 の person ドキュメントが埋め込ま

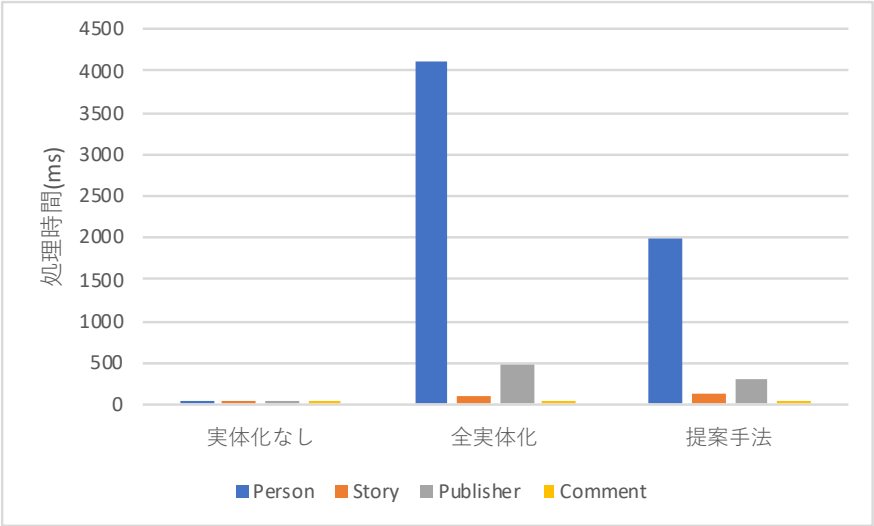

図 5.20: 実験 G の各コレクションの平均更新時間

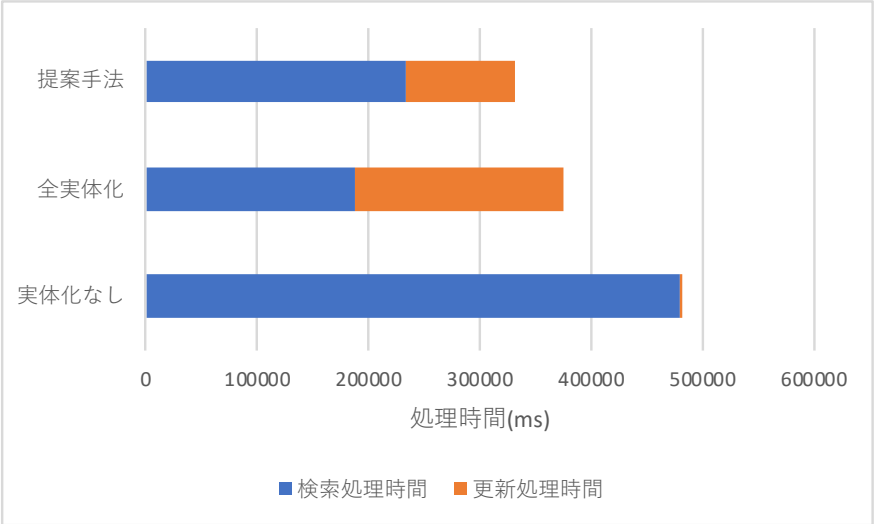

図 5.21: 実験 G の累計処理時間

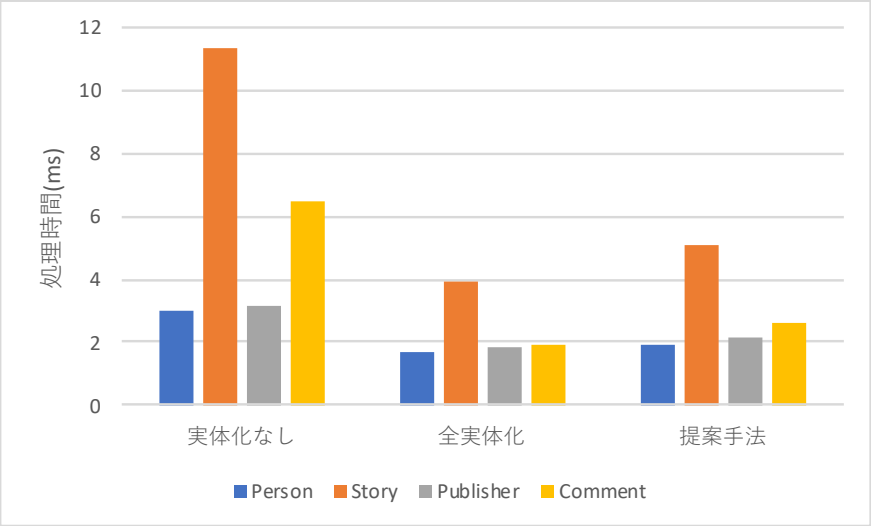

図 5.22: 実験 H の各コレクションの平均検索時間

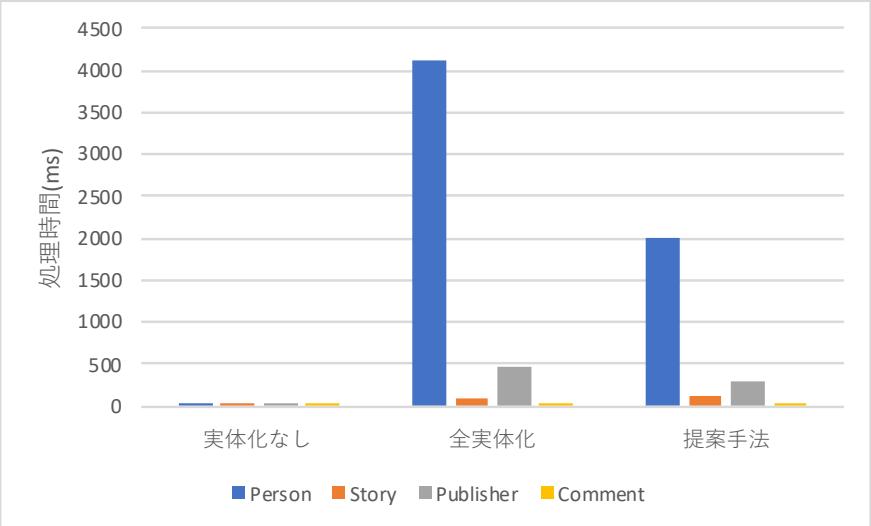

図 5.23: 実験 H の各コレクションの平均更新時間

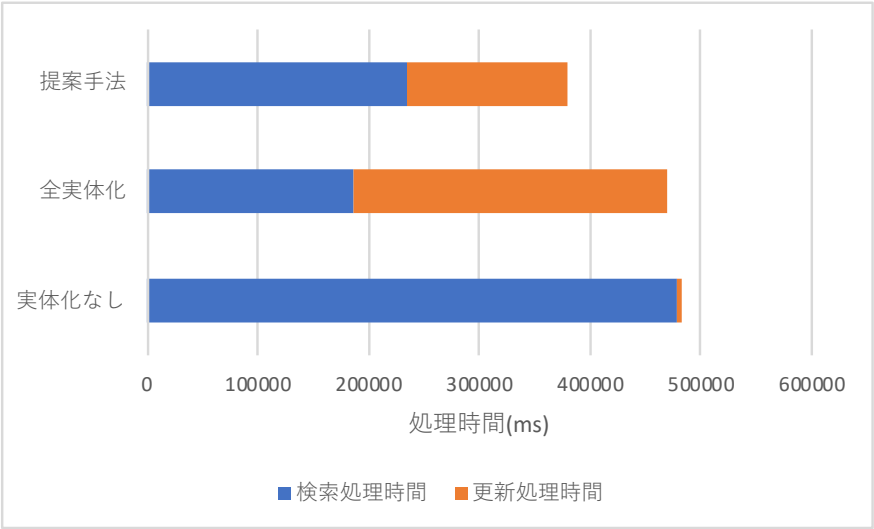

図 5.24: 実験 H の累計処理時間

れており，person ドキュメントを更新する際には実体化した person ドキュメントと，そのドキュメントが埋め込まれている story ドキュメントを更新する必要がある．実際に，実験 E ではユーザーからの person コレクションへの 9 つのクエリに対してミドルウェアで 521 のクエリを発行していた．

実験 A から F を通しての累計検索処理時間の推移を図 5.25 に，累計更新処理時間の推移を図 5.26 に，検索と更新を合わせた累計処理時間の推移を図 5.27 に示す．

今回の実験では update クエリ比が 3%以下の場合には実体化なしと比べて全実体化と提案手法が累計処理時間が速い結果となった．提案手法は update クエリ比が約 1.4%以上の際に全実体化と比べて累計処理時間が速いという結果となった．提案手法は想定通り全実体化と比べて更新処理時間を削減することができた．

図 5.25 から分かるように，検索速度に関しては全実体化が最速である．全実体化と提案手法の検索処理時間の差は update クエリ比が 1.0%以降，ほぼ一定である．一方図 5.26 から分かるように，update クエリ比が増加するに比例して全実体化と提案手法の更新時間が増加するが，同様に全実体化と提案手法の差も増加する．この全実体化と提案手法の更新処理時間の差が検索処理時間の差を超える update クエリ比が約 1.4%であり，これ以降累計処理時間で提案手法が最速となった．

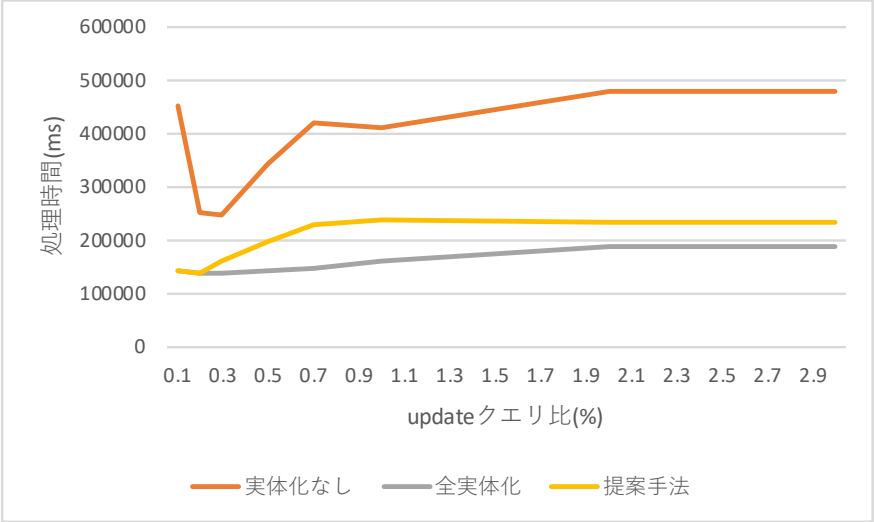

図 5.25: 累計検索処理時間の推移

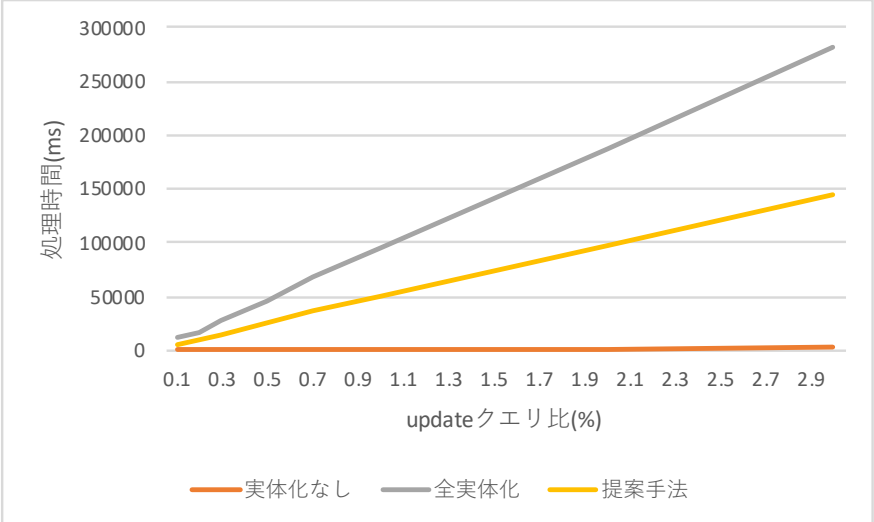

図 5.26: 累計更新処理時間の推移

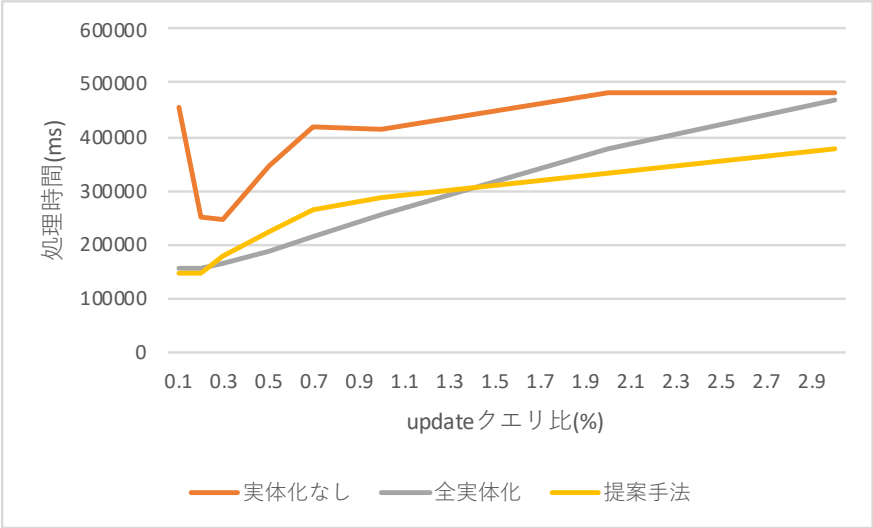

図 5.27: 累計処理時間の推移

第6章　まとめ

リレーショナルデータベースシステムで用いられる技術の一つの実体化ビューは負荷が大きい処理を事前に実データとして保持しておくことで，システムを高速化することができる．しかし，実体化ビューを作成する箇所の選択やメンテナンスを適切に行わないと，返ってボトルネックになってしまう可能性がある．本研究では NoSQL の一種であるドキュメント指向型データベースにおける実体化ビュー選択の自動化を提案する．提案手法ではミドルウェアを実装し，集積したユーザーからのクエリのログを実体化する箇所の判定に用いた．また，クエリのログや処理時間などの情報を元にボトルネックとなっている実体化ビューを探して，元のデータモデルに戻す実装を行った．

実験では，提案手法と全体を実体化したコレクションと元のデータモデルを比較し，提案手法の有用性を確かめた．実験の結果，提案手法と全体を実体化したコレクションが元のデータモデルと比べて検索処理時間が高速であった．実体化ビューを用いると更新処理速度が低下する傾向にあったが，あるクエリ条件下では提案手法が検索処理速度を保ちつつ，全体を実体化したコレクションの約 2 倍の更新速度を実現し，モデルパターンの中で最速となった．

今後は，課題となった更新速度を克服するために実体化条件と逆実体化条件に処理時間削減率等の概念を追加し，さらなる機能向上を目指す．また，複雑なクエリパターンや時系列でクエリパターンが変化する状況でも提案手法の有用性を示せるように，ミドルウェアの性能を向上させる必要がある．

謝辞

本研究を進めるにあたり，指導教員の古瀬一隆先生と陳漢雄先生から，丁寧かつ熱心なご指導を賜りました．ここに感謝の意を表します．また，研究室での議論を通じ，多くの知識をいただいた DSE 研究室の皆様に感謝いたします．

参考文献

[1] Kyle Banker. MongoDB イン・アクション. オライリージャパン, 第 1 版, 2012.

[2] 渡部徹太郎, 河村康爾, 北沢匠, 佐伯嘉康, 佐藤直生, 原沢滋, 平山毅, 李昌桓. RDB 技術者のための NoSQL ガイド. 秀和システム, 第 1 版, 2016.

[3] E. F. Codd. Recent investigations in relational data base systems. In *IFIP Congress*, 1974.

[4] Hoshi Mistry, Prasan Roy, S Sudarshan, and Krithi Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *ACM SIGMOD Record*, Vol. 30, pp. 307–318. ACM, 2001.

[5] 本橋信也, 河野達也, 鶴見利章. NOSQL の基礎知識 (ビッグデータを活かすデータベース技術). リックテレコム, 第 1 版, 2012.

[6] MongoDB. What is mongodb? https://www.mongodb.com/what-is-mongodb. (参照 2019-01-09).

[7] Guy Harrison. Next generation databases: Nosqland big data (english edition), 12 2015.

[8] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*, Vol. 7. University of California, Irvine Irvine, USA, 2000.

[9] Valeri Karpov. Mongoose odm v5.4.4. https://mongoosejs.com/. (参照 2019-01-09).

[10] Flavio Curella. Welcome to faker's documentation! ¶ . https://faker.readthedocs.io/en/master/. (参照 2019-01-09).