DEIM Forum 2021 B14-1

時刻変化するワークロードのための NoSQL スキーマの最適化

涌田 悠佑† MichaelMior†† 善明 晃由††† 佐々木勇和† 鬼塚 真†

† 大阪大学大学院情報科学研究科 〒 565–0871 大阪府吹田市山田丘 1-5
†† Rochester Institute of Technology 102 Lomb Memorial Drive Rochester, NY 14623-5608
††† 株式会社サイバーエージェント 〒 101–0021 東京都千代田区外神田 1-18-13 秋葉原ダイビル 13 階 1302 号室
E-mail: †{wakuta.yusuke,sasaki,onizuka}@ist.osaka-u.ac.jp, ††mmior@cs.rit.edu,
†††zenmyo_teruyoshi@cyberagent.co.jp

あらまし NoSQL データベースにおけるスキーマ設計は性能を引き出すために重要な技術分野である．複雑なワークロードに対して最適なスキーマを手動で設計することは困難なため静的なスキーマの最適化技術が提案されている．しかし，時刻変化するワークロードではスキーマも動的に変更しなければその性能を十分に発揮できない．本稿では時刻変化するワークロードに対応した NoSQL データベースのスキーマ最適化手法を提案する．提案手法は，ワークロードの実行コストとマイグレーションコストの総和を整数線形計画問題を用いて最小化することで，ワークロードの実行コストとマイグレーションコストのトレードオフを踏まえた最適なスキーマを推薦する．さらに，最適化モデルの一部の時刻を抜き出すことで抽象化したモデルの構築と最適化の反復により最適化候補を削減する候補削減手法を提案する．評価実験の結果，時刻変化を加えた TPC-H において静的な最適化を上回る性能を達成した．

キーワード NoSQL，スキーマ推薦，整数線形計画問題

1 は じ め に

NoSQL データベースは RDB と比べてシンプルなデータ構造やトランザクションを提供しており，大規模なアプリケーションにおいて高い分散処理性能を達成している．本稿で NoSQL データベースの代表として使用する Extensible Record Store [2] はレコードを水平・垂直に分割することで，複数のデータベースサーバに効率良くレコードを分散する．また，プロダクト毎に詳細な定義は異なるが，RDB のテーブルに相当するデータ構造として Column Family を提供している．

ワークロードに適したスキーマを設計することは，NoSQL データベースの性能を引き出すために非常に重要である．NoSQL データベースのスキーマ設計には主に 2 つの問題がある．1 点目の問題は，最適な Column Family の選択と，Column Family を利用したクエリプランの最適化を同時に解く必要がある点である．2 点目の問題は，時刻変化するワークロードではクエリプランの最適化に加え，マイグレーションとのトレードオフを考慮しなければならない点である．1 点目の問題について，クエリに高速に応答するためには，クエリ全体を実体化した Column Family を用意してそれを利用するクエリプランが有用である．しかし，このようなクエリプランを全てのクエリに使用すると， クエリ数の増加に伴いストレージ容量が大きくなる傾向がある．したがって，クエリ数の多い場合ではストレージ容量の制約等から，実行頻度の大きいクエリに優先的に高速なクエリプランを割当てることが重要となる．2 点目の問題について，ワークロードの変化に応じて，各時刻ごとに最適化なスキーマ設計を行う方法は応答時間の低減に有用であるが，大量のマイグレーションはクエリプランの処理を圧迫し応答時間が増加する．これらの問題から，複雑で時刻変化するワークロードに対してデータベース管理者が手動でスキーマを最適な状態に保つことは困難である．

NoSQL データベースでは，静的なワークロードに対するスキーマ設計手法として NoSQL Schema Evaluator (NoSE) [7] が提案されている．NoSE は整数線形計画問題 (ILP) を用いてスキーマを最適化する．ただし，静的なワークロードのみを対象としており，マイグレーションを含めた最適化は行えない．Hillenbrand ら [4] は，時刻変化するワークロードに対してデータをマイグレーションする手法を複数示し，閾値に基づいた選択方法を提案している．しかし，本稿で対象としているスキーマのマイグレーションは入力として想定しており，スキーマのマイグレーションの実行方法やその時刻については考慮しない．時刻変化するワークロードでは，クエリプランとマイグレーションプランの選択において次のような課題がある．まず，初めに各時刻のクエリプランを最適化すると，マイグレーションプランの実行時間が増加する場合や実行可能なマイグレーションプランを得られない場合がある．一方で，初めにマイグレーションプランを最適化すると，各時刻のクエリプランの応答時間が長くなる場合がある．したがって，時刻変化するワークロードでは，このクエリプランとマイグレーションプランの依存関係を踏まえたスキーマ設計が必要となる．

本稿では，時刻変化するワークロードに対応した NoSQL スキーマ最適化手法を提案する．提案手法ではワークロードの実行コストとマイグレーションの実行コストの総和の最小化を 1 つの整数線形計画問題として定式化する．これにより，ワークロードの実行コストとマイグレーションコストのトレードオフや，クエリプランとマイグレーションプランの依存関係を踏まえた最適化が可能になる．ここで，提案する目的関数では各時

刻の候補に決定変数を割り当てるため，時刻数 n の増加とともに決定変数が $O(n)$ で増加し最適化時間も増加する．そこで，最適化問題の抽象化によってその候補を削減する．この抽象化による候補削減では，最適化モデルの一部の時刻を持つ抽象化したモデルの構築と最適化を反復する．この手法では得られた解を次の反復において制約式として追加することで，各モデルに大域的な頻度変化を反映する．そして，各モデルの解から，全時刻最適化において推薦される見込みの小さい候補を削除する．

本稿の貢献点を以下に示す．

（1） クエリプランとマイグレーションプランの依存関係を整数線形計画問題として定式化し，総実行コストを最小化する目的関数と制約条件の提案.

（2） 大規模な時刻変化するワークロードに対応するための抽象化による候補削減手法の提案.

（3） Column Family とクエリプランから，推薦される見込みの大きいマイグレーションプランの列挙手法の提案.

（4） Extensible Record Store におけるマイグレーションの実行方法と，そのコストモデルの提案.

本稿の構成は以下の通りである．2 章にて事前知識について説明し，3 章にて提案手法の概要と最適化候補の列挙方法について説明する．そして，4 章にて最適化について説明する．5 章にて評価実験について説明し，6 章にて関連研究について述べる．そして，7 章にて本稿をまとめ，今後の課題について論ずる．

2 事 前 知 識

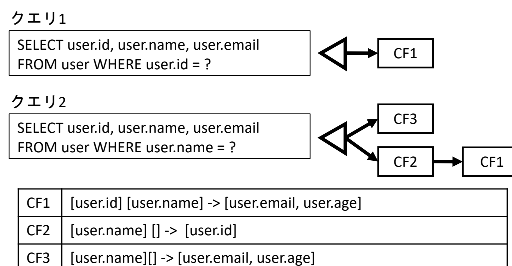
本章では，NoSQL データベースに関する技術について説明する．2.1 節では，NoSQL データベースの一種である Extensible Record Store について説明する．2.2 節では，Extensible Record Store のデータ構造である Column Family について説明する．そして，2.3 節ではスキーマ設計について具体例を用いて詳細に説明する．

2.1 Extensible Record Store

Extensible Record Store [2] はデータを分割することで，高いスケーラビリティを達成している NoSQL データベースである．プロダクトの具体例としては，Cassandra, HBase, Bigtable 等がある．Extensible Record Store は属性に基づく垂直分割とレコードに基づく水平分割を組み合わせ，効率的なデータ分散を行う．垂直分割では，属性は Column Family 等の集合へ分割され，各データベースノードに割り当てられる．水平分割では各データベースノードに各レコードのキーの値を用いたハッシュ分散や範囲分散によって各データベースノードに割り当てられる．

2.2 Column Family

Column Family (以下，CF と表記する) は RDB のテーブルに対応するデータ構造である．ただし，本稿では Extensible Record Store の具体的なプロダクトとして Cassandra を対象



CF1	[user.id] [user.name] -> [user.email, user.age]
CF2	[user.name] [] -> [user.id]
CF3	[user.name][] -> [user.email, user.age]

図 1 スキーマ設計の具体例．各クエリに対して使用可能なクエリプラン候補から 1 つのクエリプランを選択する．

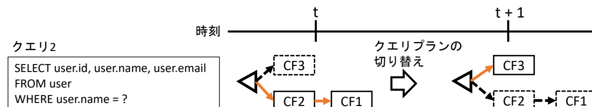


図 2 クエリプランの変更の具体例．各時刻のワークロードに応じてクエリプランを変更することで，ワークロード全体の実行コストを低減できる．

とする．CF は任意の時点で新たなカラムを追加できるため，ワークロードの変化に対して柔軟なデータ構造である．本稿では式 (1) の書式で CF を表現する．

$$[partition\ key][clustering\ key] \rightarrow [value] \qquad (1)$$

CF のレコードは *partition key* の値に基づいてノードに分散される．*clustering key* は各ノード内でレコードをソートし格納する際にソートキーとして利用される．*value* は各カラムの値である．クエリ処理を行う際に *partition key* を等号条件として利用することで，その値によりレコードの存在するデータベースノードを高速に特定できる [1]．

2.3 スキーマ設計

本章では，Extensible Record Store である Cassandra におけるスキーマ設計の具体例を示す．Cassandra のクエリ言語である CQL の記法で Put, Get を表記する．まず，ワークロードが図 1 のクエリ 1, 2 を持つとする．各クエリが使用できるクエリプランツリーを各クエリの右側に表記する．クエリプランツリーの根から葉までの 1 つの経路がクエリプラン候補である．本稿では 1 つの CF のみを持つクエリプランを **MV プラン**，2 つ以上の CF を持つクエリプランを**ジョインプラン**と呼ぶ．MV プランでは，クエリに対して 1 つの CF のみを用いるため高速な応答ができる．一方，ジョインプランでは 2 つ以上の CF をジョインしてクエリに応答する．しかし，Extensible Record Store はジョイン機能を提供しておらず，クライアント側でのジョインが必要となるため応答時間が増加する．ただし，ジョインプランを用いることで，CF が正規化されるため CF 間でのカラムの重複が減少する．これにより，更新処理において一貫性を保ちながら更新しなければならない CF の数が減少し，更新処理の性能が向上する場合がある．具体例として，

1：*partition key* を等号条件を持たないクエリを実行する場合，全データベースノードから全レコードを取得してからクライアントでフィルタリングするため応答時間が非常に大きくなる．

ワークロードが以下の更新処理を持つ場合を考える．

```
U1: UPDATE user SET user.email = ? WHERE user.id = ?
```

ここで，クエリ 1,2 がそれぞれ CF1, CF3 の MV プランを使用する場合では，CF1 と CF3 は $user.email$ を共有してるため，$U1$ は 2 つの CF を更新しなければならない．次に，クエリ 2 が CF1, CF2 を使用するジョインプランを使用する場合では応答時間は増加するが，$user.email$ は CF1 にのみ存在するため $U1$ の処理速度は向上する．このようにクエリ処理と更新処理の性能のトレードオフを踏まえて，適切なスキーマ設計を用いることが重要となる．

多くのアプリケーションにおいて，データベースで実行する処理は時刻に応じて変化する．図 2 に時刻変化するワークロードに対するスキーマ設計の具体例を示す．クエリ 1 は CF 1 の MV プランを使用するとする．時刻 t から $t+1$ にかけて U1 の実行頻度が減少するとする．時刻 t では，クエリ 2 は CF1, CF2 をジョインするジョインプランを使用している．ここで，$user.email$ は CF1 にのみ存在するため，頻度の大きい $U1$ は CF1 のみ更新すれば良い．時刻 $t+1$ においてはクエリ 2 は CF3 を使用する．これにより，クエリ 2 は MV プランで高速に応答できるが，頻度の小さい $U1$ は CF1, CF3 を一貫性を保ちながら更新しなければならない．このようにマイグレーションによってクエリプラン変更することで重要度の高い処理を高速に処理することが有用である．しかし，頻繁にマイグレーションすると他の処理を圧迫するため，マイグレーションを行うか否かはそのメリットとコストを踏まえて判断しなければならない．

NoSQL データベースのスキーマ設計手法として NoSE [7] が提案されている．NoSE は静的なワークロードのクエリ処理と更新処理の実行コストを見積もり，スキーマ設計を行う．しかし，時刻変化するワークロードに対応していないため，ワークロードの変化に応じてスキーマを変更することはできない．したがって，一度スキーマを最適化してもワークロードの変化によって性能が低下する可能性がある．

3 提案手法

本フレームワークは，時刻変化するワークロードにおけるスキーマを ILP を用いて最適化することで，最適なワークロードの実行プランとマイグレーションプランを推薦する．本稿で対象とする時刻変化するワークロードは各時刻のワークロード $w(t)$ の系列 $(w(0), w(1), ..., w(T))$ と表せる．このとき，各 $w(t)$ は時刻 t におけるクエリ処理・更新処理の実行頻度の情報を含む静的なワークロードである．本手法はワークロードの時刻変化は既知であるとして最適化を実行する [2]．最適化ではワークロードの実行コストとマイグレーションの実行コストの総和の最小化を 1 つの整数線形計画問題として定式化する．これにより，ワークロードの実行コストとマイグレーションコストのトレードオフや，クエリプランとマイグレーションプランの依存関係を踏まえた最適化を行う．また，提案する目的関数では各時刻の候補に決定変数を割り当てるため，時刻数 n の増加とともに決定変数が $O(n)$ で増加し最適化時間も増加するため，最適化問題の抽象化によってその候補を削減する．この抽象化による候補削減では，最適化モデルの一部の時刻を持つ抽象化したモデルの構築と最適化を反復する．そして，各モデルの解から，全時刻最適化において推薦される見込みの小さい候補を削除する．

本稿で提案するフレームワークは以下の処理手順で時刻変化するワークロードに対応したスキーマ設計を行う．

（1） Column Family の列挙: ワークロードのクエリへの応答に使用できる CF を列挙する．

（2） クエリプランの列挙: 列挙した CF を活用した各クエリ処理と更新処理のクエリプラン候補を列挙する．

（3） マイグレーションプランの列挙: 列挙した CF，クエリプランを用いてマイグレーションプラン候補を列挙する．

（4） コスト推定: クエリ処理と更新処理，マイグレーションプランの実行コストを推定する．

（5） 抽象化による候補削減: 抽象化した全体最適化の ILP の構築と最適化を反復することで，全体最適化の候補を削減する．

（6） 全時刻最適化: ワークロード実行コストとマイグレーションコストの総和を最小化する ILP を最適化する．

3.1 Column Family とクエリプランの列挙

本フレームワークでは NoSE の手法を改良した手法で CF の候補を列挙する．NoSE は各クエリと，各クエリを元に列挙した部分クエリや条件緩和クエリに対して，その Materialized View に相当する CF を列挙する．まず，各クエリをクエリグラフの辺で分解することで，部分クエリグラフに対応した部分クエリを列挙する．この部分クエリ全体を実体化した Column Family を作成することで，ジョインプランで使用できる Column Family が列挙される．さらに，各部分クエリに対して WHERE 句のカラムの一部を SELECT 句に移動した条件緩和クエリを列挙する．この緩和方法のパターンによって，キーの属性の全ての順列毎の Materialized View を得る．しかし，クエリグラフが多くの辺を持つ場合や，WHERE 句に多くの条件を持つ場合には急激に CF の候補が増加する．まず，クエリグラフが k 個の辺を持つ場合，部分クエリの数は $N_k = 1 + k + \sum_{j=1}^{k-1} N_j = 2^{k+1} - 1$ 個となる．さらに，部分クエリが WHERE 句に l 個の条件を持つ場合は，条件緩和クエリは $l!$ 個生成される．

本手法ではクエリの WHERE 句で使用される条件の頻度に基づいて条件緩和クエリを削減する．この手法では，クエリや部分クエリの WHERE 句の条件集合から，頻出な等号条件の属性のグループを FP-growth [3] を用いて取得する．これらの同時に使用される頻度の高い条件属性の集合ごとに SELECT 句へ移動させるかどうかを確定する．ここで，Cassandra では，partition key を指定しなければ Get を実行できない．そのた

2：したがって，未知の頻度変化を対象外であるが，全ての時刻を横断して 1 つの ILP モデルを作成し最適化することで全時刻に対して最適なスキーマ設計を行う．

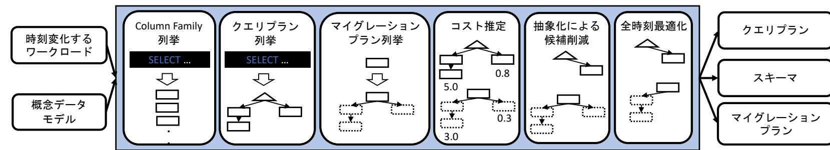


図 3　提案手法の概要図

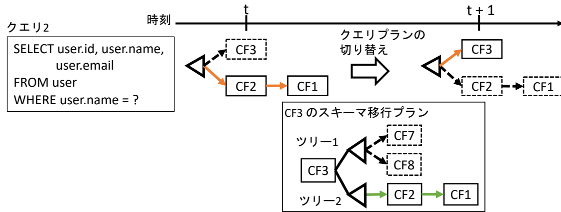


図 4　マイグレーションの具体例．各 CF にマイグレーションプランの候補を列挙する．

め，頻繁に条件として参照される属性であっても一部の属性が partition key や clustering key の先頭属性に存在しない場合は，この Column Family は使用できない．そのため，頻出な属性をグループ化し，グループ内の属性が同時に partition key に含まれるようにする．また，1 つのクエリにおいてのみ使用される条件属性については，SELECT 句に移動する順序をそのカーディナリティ順に固定することで，緩和クエリの数を削減する．

次に列挙した CF を用いて，クエリの実行プランであるクエリプランを列挙する．クエリプランの列挙においても NoSE の手法を効率化した手法を使用している．ただし，NoSE の手法では，各部分クエリから多くの条件緩和クエリが生成されている場合，ジョインプランの候補数が急激に増加する．

NoSE の手法ではクエリが多くの条件やエンティティをジョインしている場合にクエリプランの候補数が急激に増加する．そこで本手法では，MV プランの用途に着目してプラン数を削減する手法を提案する．Column Family の列挙において，各クエリの MV プランで使用可能な Column Family が多く列挙される．ここで，MV プランは高速にクエリに応答できることに加え，複数のクエリに対して応答できる場合にはデータの冗長性も低減できる．また，各クエリに対して最も高速に応答できる MV プランはクエリで使用されている条件や属性と，4.4 節で述べるコストモデルから推定される．そこで，列挙した MV プランにおいて他のクエリと共有されていない MV プランを削除し，最も性能の高い MV プランと複数のクエリで共有されている MV プランのみを用いる．さらに，NoSQL データベースではクライアントでジョインを行わなければならないため，低い性能が想定される 3 つ以上の CF を用いるジョインプランを枝刈りする．これらの枝刈り手法により，推薦される可能性の高いクエリプランのみを列挙する．

3.2　マイグレーションプランの列挙

スキーマ変更の実行計画であるマイグレーションプランを列挙する．マイグレーションプランでは，時刻 t に存在する CF を使用して時刻 $t+1$ で使用する CF のデータを取得する．本手法では 2 種類のマイグレーションプランを列挙する．1 つ目の列挙手法では，CF の属性を取得するクエリを新たに生成しクエリプランを列挙する．ただし，最適化候補となる CF とクエリプランの削減のため，このクエリのための CF の列挙はせず既に列挙済みの CF を使ってクエリプランを列挙する．具体例を図 4 に示す．この例では，時刻 $t+1$ に新たに CF 3 を使用するため，時刻 t のクエリプランから CF3 を作成する．1 つ目の列挙手法によるクエリプランツリーは図 4 のツリー 1 に対応している．しかし，この手法では作成する CF とキー属性が一致する CF が存在しない場合等にマイグレーションプランが推薦できない可能性がある．

そこで，2 つ目のマイグレーションプランの列挙方法ではワークロードのクエリのクエリプランを再利用する．同一のクエリプランツリー内のクエリプランは同一のクエリに応答するため，似通った属性集合のレコードを取得すると考えられる．そこで，マイグレーションで作成する CF を持つクエリプランツリーをマイグレーションプランのクエリツリーとして再利用する．ただし，ワークロード内のクエリは集約処理等の属性を取得する以外の処理が含まれる場合がある．したがって，ワークロード内のクエリから集約関数等を省いたクエリを作成し再度クエリプランツリーを作成し，マイグレーションプランツリーの候補に追加する．図 4 においてクエリ 2 は集約処理等は持たないため，時刻 t のクエリプランツリーを再利用できる．よって，時刻 t に使用したクエリプランをマイグレーションプランのツリー 2 として使用する．

ただし，各 CF は同じクエリプランツリー内の他のクエリプランでは取得できない属性を持つ場合がある．この場合，この手法ではマイグレーションプランを作成できない．この時，1 つ目の列挙手法では，属性を全て取得するクエリを生成するため，同一のクエリプランツリーに含まれない CF を用いたマイグレーションプランを列挙できる．したがって，これらの 2 つの手法を組み合わせて使用することが重要となる．

マイグレーションプランで作成する CF は各属性に関して全てのレコードを持つことを保証する．しかし，マイグレーションプランは CF の属性集合に基づいて構築されるため，CF がジョインしているエンティティを考慮しない．したがって，Cassandra へのデータのロードにおけるジョインでのレコードの欠落を防ぐために，外部結合によってデータをロードする．さらに，Cassandra は partition key, clustering key が一致するレコードは同一レコードと見なす．したがって，value のみが異なる CF のレコードは，同じ key に対する INSERT で上書きされる．そこで，value レコードのハッシュ値を clustering key の末尾に属性として追加することで，上書きによるレコードの欠損を防止する．

4 最 適 化

3.1, 3.2 節で列挙したクエリプランとマイグレーションプランの候補を用いて，時刻変化するワークロードに対応したスキーマ最適化を行う．4.1 節では，全時刻最適化で用いる ILP の目的関数について説明する．4.2 節では，ILP の制約条件について説明する．次に，4.3 節では，全時刻最適化の最適化候補を削減するための抽象化による候補削減について説明する．そして，4.4 節では，各実行コストの推定方法について説明する．

4.1 全時刻最適化の目的関数

ワークロード実行コストとマイグレーション実行コストの総和を最小化するための ILP の目的関数を式 (2) に示す．

$$M = argmin(\sum_{t=0}^{T} workload(S_t) + migrate(S_{t-1}, S_t)) \quad (2)$$

M は各時刻 $t = 0...n$ のスキーマの系列 $(S_0, S_1, ...S_n)$ である．$workload(S_t)$ は時刻 t にスキーマ S_t を用いてワークロードのクエリ処理と更新処理を実行する際のコストである．また，$migrate(S_{t-1}, S_t)$ は時刻 $t-1$ のスキーマ S_{t-1} から，時刻 t に使用するスキーマ S_t にマイグレーションする際のコストである．この 2 項の全時刻における総和を最小化することでスキーマ系列 M を得る．本章では，目的関数や制約条件の定式化について詳細に述べ，各処理の具体的なコスト推定方法は 4.4 章で説明する．

4.1.1 ワークロード実行コストの定式化

時刻 t のワークロードの実行コストに対応する項を式 (3) に示す．

$$workload(S_t) = \Sigma_i \Sigma_j f_i(t) C_{ij} \delta_{ijt} + \Sigma_m \Sigma_n f_m(t) C'_{mn} \delta_{nt} \quad (3)$$

この式は NoSE を時間軸方向に拡張した式で，第 1 項がクエリ処理のコストを，第 2 項が更新処理のコストを表している．$f_i(t)$ はクエリ処理 i の時刻 t における実行頻度，C_{ij} はクエリ処理 i において CF j を使用するコストを表している．$f_m(t)$ は更新処理 m の時刻 t における実行頻度，C'_{mn} は更新処理 m において CF n を更新するコストを表している．δ_{ijt} はクエリ処理 i が時刻 t に CF j を使用するか否かを示す 0 もしくは 1 の値をとる 2 値変数である．また，δ_{nt} は CF n が時刻 t で存在するか否かを示す 2 値変数である．ここで，S_t は，式 (4) で表現できる．

$$S_t = \{\forall j.CF_j | \delta_{jt} \geqq 1\} \quad (4)$$

本手法では各処理の重要度はその実行頻度に基づいて評価する．そして，重要度の高い処理はコストの小さいクエリプランが割り当てられることが望ましい．そのため，クエリ処理・更新処理共に頻度と実行コストの積を決定変数の係数とすることで，頻度の高い処理ほどコストの小さいクエリプランが割り当てられるように最適化する．

4.1.2 マイグレーション実行コストの定式化

マイグレーション処理では，既にデータベースに存在する CF から，新規の CF を作成する．ただし，マイグレーション処理はデータベースとの通信の応答時間が短いサーバに配置されたマイグレータで実行するものとする．マイグレーションのコストを推定する項を式 (5) に示す．

$$\begin{aligned} migrate(S_{t-1}, S_t) = & \sum_{gh} C''_{gh} \delta'_{ght} + \sum_{j} C'''_{j} \delta''_{jt} \\ & + \sum_{m} \sum_{n} f_{m(t-1)} C'_{mn} \delta''_{nt} \end{aligned} \quad (5)$$

第 1 項は新規の CF のためにマイグレーションプランによってデータを収集する処理 (**収集ステップ**) の実行コストを評価する．C''_{gh} は CF g のマイグレーションプランにおいて CF h を使用するコストである．δ'_{ght} は決定変数であり，時刻 t に CF g を作成するマイグレーションプランにおいて CF h が使用されるかを表す 2 値変数である．また，時刻 t において，マイグレーションによって新たに CF g を作成しない場合でも 0 をとなる．

マイグレーションプランは，新たに作成する CF の全てのレコードを取得するため，実行方法がクエリプランとは異なる．クエリの各実行では等号条件に一致したレコードのみが取得される．そして，式 (3) に示した通り，実行頻度と各実行時のコストの積を決定変数の係数として使用する．一方で，マイグレーションプランではプラン内の CF のレコードを全て取得するが実行回数は一度のみである．したがって，CF のレコードを全件取得する際のコスト C''_{gh} のみを決定変数の係数として使用する．

第 2 項は新規の CF に対してマイグレータに収集したデータをロードする処理 (**ロードステップ**) の実行コストを評価する．C'''_j は CF j にデータをロードする際のコストである．決定変数 δ''_{jt} は時刻 t に向けて時刻 $t-1$ から CF j を作成するマイグレーションを行うか否かを表す 2 値変数である．

第三項は，マイグレーション中に実行される更新処理を新規作成中の CF に対しても実行するコストを評価する．本稿で提案するマイグレーション処理では，マイグレーション処理途中でもワークロードのクエリ処理と更新処理は変わらず実行する．この時，時刻 $t-1$ の時点で存在している CF に対して更新処理を行うだけでは，マイグレータに収集されたデータに対して更新処理が実行されない．その結果，既存の CF と新たに作成された CF で同じ属性を持っていても，レコードの値に差異が生じる．この問題に対応するために，マイグレータは時刻 $t-1$ になった際に時刻 t のために新規作成する全ての CF を定義する．そして，時刻 $t-1$ までに存在していた CF に加えて，これらの CF に対しても同様に更新処理を実行する．これによって，マイグレーション処理による更新処理の欠損を防ぐ．$f_{m(t-1)}$ は時刻 $t-1$ の更新処理 m の実行頻度である．そして，C'_{mn} は，作成中の CF を更新するコストである．また，決定変数 δ''_{nt} は δ''_{jt} と同様の決定変数である．

ただし，更新処理によって挿入されるレコードの partition key, clustering key の組み合わせが既に存在する場合は，データ収集後に実行された更新処理がデータのロードによって上書きされ，更新処理が欠損する場合がある．この解決方法として，

実行された更新処理をタイムスタンプ付きでキューに保存し，データのロードの後にタイムスタンプの時刻を比較しながら更新処理を適用する方法等が考えられる．しかし，本研究は最適化手法に重点を置くためこれらの手法は範囲外とする．

4.2 制約条件

クエリプランとマイグレーションプランを選択するための制約条件について説明する．クエリプランに割当てる制約式によって，各クエリの各時刻において 1 つのクエリプランが推薦されることを保証する．図 2 に示した具体例を用いて説明する．各時刻においてクエリプランを 1 つ選択する制約式は NoSE で提案されている制約式を時刻方向に拡張したものである．クエリ 2 の時刻 t における制約式は式 (6) に示す通りである．

$$\begin{aligned} \delta_{21t} &\geqq \delta_{22t} \\ \delta_{2jt} &\geqq \delta_{jt} \quad \forall j \in \{1,2,3,4\} \\ \delta_{23t} + \delta_{22t} = 1, &\quad \delta_{23t} + \delta_{23t} = 1 \end{aligned} \tag{6}$$

また，容量制約を定義することで，各時刻で使用される CF のストレージサイズを制限する．容量制約を制約式 (7) に示す．

$$\sum_j s_j \delta_{jt} \leqq B, \quad \forall t \in [0, T] \tag{7}$$

s_j は CF_j のサイズである．また，B は制約として与えるストレージ容量の上限値である．ここで，多くの更新処理が実行される場合では，時間の経過とともに CF のサイズが変化する場合があるが，簡単のため CF のサイズは一定であるとする．

各時刻の決定変数からマイグレーションを実行するか否かを判定するための決定変数 δ''_{jt} を求める．時刻 $t-1$ から時刻 t に向けてクエリプランを変更する際，時刻 $t-1$ に存在しない CF はマイグレーションプランによって新たに作成しなければならない．各時刻の CF の集合からマイグレーションプランを作成するか否かを判定する制約条件を式 (8) に示す．

$$\delta_{jt} - \delta_{j(t-1)} \leqq \delta''_{jt}, \quad 0 \leqq \delta''_{jt} \tag{8}$$

この制約式により δ''_{jt} は時刻 t に CF j を新たにマイグレーションで作成する場合に 1 となる．また，時刻 t 以降に使用されない CF は随時削除する．

列挙した複数のマイグレーションプランのクエリプランツリーから制約式を用いて 1 つのクエリプランを選択する．図 4 の CF 3 のためのマイグレーションプランについて制約式を示す．まず，ツリー 1，ツリー 2 それぞれ 1 つのクエリプランを選択するための制約式を式 (9) に示す．

$$\begin{aligned} \delta_{21t} &\geqq \delta_{22t}, \\ \delta'_{32t} = \delta'''_{31t}, &\quad \delta'_{31t} = \delta'''_{31t}, \\ \delta'_{37t} + \delta'_{38t} &= \delta'''_{32t} \end{aligned} \tag{9}$$

この制約式は制約式 (6) を拡張したものであり，ツリーが選択される場合にのみクエリプランが 1 つに定まる制約が適用される．

Algorithm 1: 抽象化による候補削減のアルゴリズム

Input : model (最適化モデル), start_ts (開始時刻), end_ts (終了時刻)
Output: 有効性の低い CF を除いた最適化候補

```
1  Function pre_solve(model, pruned_cfs, start_ts, end_ts) is
2      if start_ts + 1 is end_ts then
3          return model.candidates
4      end
5      solution = solve_ilp(model)
6      middle_ts = (end_ts - start_ts) / 2 + start_ts
7      new_const = prepare_const(solution, start_ts, middle_ts,
        end_ts)
8      left_model = setup_submodel(model, new_const,
        start_ts, middle_ts)
9      pruned_cfs += pre_solve(left_model, start_ts, middle_ts)
10     right_model = setup_submodel(model, new_const,
        middle_ts, end_ts)
11     pruned_cfs += pre_solve(right_model, middle_ts,
        end_ts)
12     return pruned_cfs
13 end
```

次に複数のマイグレーションプランツリーから制約式によって 1 つのツリーを選択する．ここで，決定変数 δ'''_{jkt} を導入する．δ'''_{jkt} は時刻 t にむけて実行する CF j を作成するマイグレーションで，マイグレーションプランツリー k が選択されるか否かを表す 2 値変数である．この変数を用いた制約式を式 (10) に示す．

$$\delta'''_{31t} + \delta'''_{32t} = \delta''_{3t} \tag{10}$$

これらの制約式によって，ある CF を新たにマイグレーションで作成する場合にのみ，そのマイグレーションプランを選択する制約が適用される．

4.3 抽象化による候補削減

4.1 節に示した ILP は時刻方向に対して線形に決定変数が増加する．したがって，時刻数の多いワークロードに関しては最適化に必要な時間が増加してしまう．

そこで，抽象化による候補削減により最適化候補を削減する．抽象化による候補削減では，ワークロードを多段階に 3 時刻ずつのワークロードに要約する．そして各要約されたワークロードに対して，最適化モデルの構築と最適化を反復して行う．

抽象化による候補削減のアルゴリズムをアルゴリズム 1 に示す．*solve_ilp*() 関数は与えられた ILP を最適化する．*prepare_const*() 関数は，解から *start_ts*, *middle_ts*, *end_ts* の CF 集合を固定する制約条件を作成する．*setup_submodel*() 関数は，新たな開始時刻と終了時刻を持ち，*new_const* を適用した最適化モデルを作成する．そして，この関数で得られた部分問題に対応したモデルを再帰的に *pre_solve*() 関数に与えて再帰的に最適化する．1 反復目として，ワークロードの開始時刻，中間時刻，終了時刻を抜き出して要約したワークロードに対して ILP を作成し，最適化する．次の反復においては開始時刻

と中間時刻，中間時刻と終了時刻をそれぞれ開始時刻と終了時刻として持ち，さらに新たな中間時刻を持つ要約されたワークロードをそれぞれ最適化モデルを作成し最適化する．ここで，最適化により得られた解を次の反復で構築するモデルに制約式として追加することで，モデルに大域的な頻度変化を反映する．この処理を全ての時刻において解が得られるまで反復する．そして，どのモデルにおいても推薦されない候補を全時刻最適化から除外する．

4.4 コスト推定

ILP の目的関数で用いるワークロードとマイグレーションの実行コストを見積もる．ワークロード内のクエリ処理と更新処理の実行コストを計算するコストモデルは NoSE のコストモデルを使用する．NoSE のコストモデルでは，Get の回数や返却される属性数に基づいてコストを推定する．

次に，マイグレーション処理の収集ステップ，ロードステップ，作成中の CF に対する更新処理のそれぞれの実行コストを評価する．まず，収集ステップの実行コスト ($collect_cost$) を見積もる．収集ステップでは，CF のデータを全て取得しなければならない．ただし，実行は一度のみのため，ワークロード内のクエリ処理とは異なり頻度は考慮しない．CF のサイズに基づいてコストを式 (11) によって推定する．

$$C''_{ght} = migrate_plan_coeff \times CF_g.entries \times C_{ght} \quad (11)$$

マイグレーション処理によるワークロード内のクエリ処理と更新処理への影響を評価するために，定数 $migrate_plan_coeff$ を係数として用いる．また，Get の回数を評価するために，対象の CF のエントリ数 $cf.entries$ をクエリと同様のコストモデルで計算する cf_cost に掛け合わしている．

ロードステップのコストはその CF のサイズに依存する．したがって，データベースの挿入性能に依存する係数を CF のサイズに掛け合わせてコストを見積もる．

$$C'''_j = creation_coeff \times s_j \quad (12)$$

次に，マイグレーションのため作成中の CF に対しても更新処理を実行する際のコスト C'_{mn} を式 (13) で見積もる．

$$C'_{mn} = (T_n/interval) \times C_{mn} \quad (13)$$

T_n は CF n にデータをロードする際の処理時間を表しており，$T_n/interval$ で，タイムステップ間のインターバル中に CF を作成している時間を表している．CF n を作成する際に必要な時間 T_n を CF n のサイズ s_n と定数 α を用いて式 (14) で計算する．

$$T_n = \alpha s_n \quad (14)$$

α はデータベースの挿入性能に基づいて決定する定数であり，$f_{m(t-1)}$ は更新処理 m の時刻 $t-1$ における実行頻度，C'_{mn} は更新処理 m が CF n を更新する際のコストである．

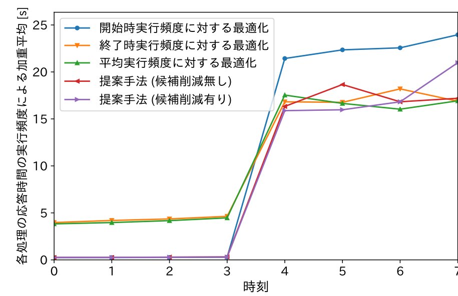


図 5　各処理の実行頻度による応答時間の加重平均．提案手法が前半の時刻では開始時実行頻度に対する最適化，後半の時刻では終了時実行頻度に対する最適化と同等程度の応答時間を達成している．

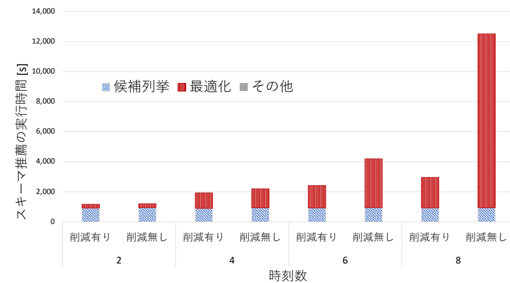


図 6　時刻数を変化させた際の抽象化による候補削減の実行時間の推移．抽象化による候補削減を用いない提案手法では，時刻数の増加とともに実行時間が大幅に増加するが，抽象化による候補削減を用いることで実行時間の増加を低減している．

5 評価実験

評価実験として時刻変化を加えた TPC-H における応答時間を評価した．時刻変化を加えるため TPC-H を 2 つのグループに分割し，それぞれ実行頻度変化を与えた．比較手法として開始時の実行頻度，終了時の実行頻度，平均実行頻度をそれぞれ用いて静的な最適化を行う 3 手法を用いた．また，各クエリに実体化ビュー相当のテーブルを利用するクエリプランを推薦できない場合を想定し，静的な最適化の解から 20% 低減した容量制約を設けた．実験結果から，提案手法は応答時間の実行頻度による加重平均の総和を，開始時実行頻度，終了時実行頻度，平均実行頻度に対する最適化に比べてそれぞれ 22.5%，17.5%，15.3% 低減していることを確認した．提案手法は 4 つのクエリでマイグレーションによってクエリプランを変更し，実行頻度に応じてクエリプランを変更した．また，図 5 から提案手法がマイグレーションによって各時刻における応答時間の加重平均を低減していることを確認した．次に，抽象化による候補削減が提案手法の最適化実行時間に与える影響について確認する．ワークロードの時刻数を変化させて各時刻数における最適化時間を測定した結果を図 6 に示す．図示した実行時間は各手法の総実行時間を表している．したがって，どの実験設定における実行時間においても，候補列挙と最適化の段階が実行時間の大部分を占めていることが分かる．候補列挙は，提案手法の Column Family の列挙，クエリプランの列挙，マイグレーショ

ンプランの列挙に対応している．図 6 から，候補列挙に必要な実行時間はどの時刻数のワークロードに対しても同程度であることが確認できた．最適化に関して，抽象化による候補削減を用いない提案手法では，時刻数の増加とともに最適化時間が大幅に増加していることが分かる．一方，抽象化による候補削減を用いる提案手法では，時刻数の増加に伴う最適化の実行時間の増加を低減している．ただし，抽象化による候補削減は 1 段階目の部分問題の最適化において 3 時刻を使用するので，3 時刻以下のワークロードに対しては抽象化による候補削減は適用していない．したがって，時刻数 2 の結果については，抽象化による候補削減有りと記載した結果についても，抽象化による候補削減は適用されていない．時刻数が 8 の場合では，抽象化による候補削減によって最適化の実行時時間を 76.3% 低減していることを確認した．また，図 5 に示した結果において，抽象化による候補削減を用いる手法の応答時間の加重平均の総和は抽象化による候補削減を用いない手法に比べて 0.88% のみ増加していた．この結果から，抽象化による候補削減が応答時間の増加を低減しつつ，最適化の実行時間を大幅に低減することを確認した．

6 関 連 研 究

インメモリデータベースを初めとする RDB を対象とした時刻変化するワークロードのためのスキーマ設計手法が提案されている [5,8]．Pavlo ら [8] はインメモリデータベース向けの self-driving database フレームワークである Peloton を提案している．Peloton は recending-horizon control model (RHCM) を用いて，ワークロードに変化に合わせたスキーマの操作を行う．Kossmann ら [5] はインメモリデータベースを対象とした動的な最適化手法を提案している．Kossmann らの手法では，Linear Programming を用いて，複数の最適化項目をどの順序で最適化するかを決定する．

また，スキーマだけでなくデータベースの様々なパラメータを動的に変更する手法も提案されている [9,10]．Wiese ら [10] は自動でチューニングを行うデータベースを提案している．提案されているデータベースでは，データベース管理者がチューニングの手順を指定し，指定された手順はデッドロックが一定数以上発生する等の条件が満たされた場合に自動で実行される．Aken ら [9] は保持しているデータを元に，メモリサイズ，キャッシュサイズ等のチューニングを自動で行う OtterTune を提案している．OtterTune は教師あり学習手法と教師なし学習手法を組み合わせて最も重要なチューニング項目を選択する．

NoSQL データベースにおいてのマイグレーションを対象とした研究も行われている [1,4]．Hillenbrand [4] らは，スキーママイグレーションを入力として，データマイグレーションの手法を複数提案し，ルールベースで適切なデータマイグレーション手法を選択する手法を提案している．本研究の提案手法では，時刻変化するワークロードのためのスキーママイグレーションを対象として ILP で最適化することで，全ての時刻においてクエリプランとマイグレーションプランを推薦する．

7 ま と め

本稿では，時刻変化するワークロードに対して NoSQL データベースのスキーマを設計する推薦フレームワークを提案した．このフレームワークでは，ワークロードの Put, Get の実行コストとマイグレーション処理の実行コストの総和を ILP として定式化する．そして，この ILP を最適化することで，全ての時刻に関して最適なスキーマとクエリプランとマイグレーションプランを推薦する．既存手法の静的なスキーマ最適化手法では，時刻変化するワークロードの変化に追従できず，性能が低下する場合でも，提案手法により安定して高い性能を達成できることを評価実験を通して確認した．今後の課題として，時刻間の幅が固定であるため自由な時刻幅に対応した最適化やクエリ頻度の予測手法 [6] と組み合わせて頻度が未知の場合での最適化が考えられる．

8 謝　　辞

この成果は, 国立研究開発法人新エネルギー・産業技術総合開発機構（ＮＥＤＯ）の委託業務の結果得られたものです．

文　　献

[1] p. boncz, s. manegold, a. ailamaki, a. deshpande, t. kraska, a. hillenbrand, m. levchenko, u. störl, s. scherzinger, and m. klettke. migcast: putting a price tag on data model evolution in nosql data stores. In *SIGMOD*, pages 1925–1928, 2019.

[2] R. Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Record*, 39:12–27, 2011.

[3] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. *SIGMOD Record*, 29(2):1–12, 2000.

[4] A. Hillenbrand, U. Störl, M. Levchenko, S. Nabiyev, and M. Klettke. Towards Self-Adapting Data Migration in the Context of Schema Evolution in NoSQL Databases. In *ICDEW*, volume 00, pages 133–138, 2020.

[5] J. Kossmann and R. Schlosser. A framework for self-managing database systems. In *ICDEW*, pages 100–106, 2019.

[6] L. Ma, D. V. Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *SIGMOD*, page 15, 2018.

[7] M. J. Mior, K. Salem, A. Aboulnaga, and R. Liu. NoSE: Schema design for NoSQL applications. *TKDE*, 29(10):2275–2289, 2017.

[8] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang. Self-driving database management systems. In *CIDR*, 2017.

[9] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *SIGMOD*, pages 1009–1024, 2017.

[10] D. Wiese, G. Rabinovitch, M. Reichert, and S. Arenswald. Autonomic tuning expert: a framework for best-practice oriented autonomic database tuning. In *CASCON, 27-30*, page 3, 2008.