情報処理学会論文誌　データベース　Vol.13　No.1　20–32 (Jan. 2020)

Secondary Indexを活用するNoSQLスキーマ推薦によるクエリ処理高速化

涌田 悠佑[1,a)]　善明 晃由[2,b)]　松本 拓海[1,c)]　佐々木 勇和[1,d)]　鬼塚 真[1,e)]

受付日 2019年6月10日, 採録日 2019年10月7日

概要：NoSQL データベースは，高い性能やスケーラビリティによってソフトウェアのバックエンドとして広く使用されており，そのスキーマ設計は性能を引き出すための重要な課題である．しかし，手動での設計で性能を十分に引き出すことは困難であるため，スキーマ推薦フレームワークによる自動最適化が求められている．本稿では Secondary Index の活用を考慮したスキーマ推薦によりクエリ処理および更新処理の高速化を図る．具体的にはクエリ処理に利用可能な Column Family，Secondary Index 候補群を列挙し，Binary Integer Programming によって最適なスキーマ設計およびクエリプランを導出する．評価実験により，更新処理が多い場合に相当する容量制限下において，頻度による重み付き平均応答時間を既存手法に対して 78.0%低減可能であることを確認した．

キーワード：NoSQL，スキーマ推薦，Secondary Index，Binary Integer Programming

NoSQL Schema Recommendation by Utilizing Secondary Index for Efficient Query Processing

Yusuke Wakuta[1,a)]　Teruyoshi Zenmyo[2,b)]　Takumi Matsumoto[1,c)]　Yuya Sasaki[1,d)]
Makoto Onizuka[1,e)]

Received: June 10, 2019, Accepted: October 7, 2019

Abstract: NoSQL engines provide high performance and scalability for large-scale databases, so they are widely used at the backend of various applications. Designing good NoSQL schema is a critical problem so as to maximize the performance of databases, however it is difficult for database administrators to manually design good schema. In this paper, we present a schema design method that speeds up processing queries and update operations by utilizing Secondary Index. In detail, this method first enumerates candidates of Column Families and Secondary Indexes for processing queries and update operations and, then, obtains optimized schema and query plans by applying Binary Integer Programming. The experiments show that the method successfully reduces the frequency-weighted average latency by 78.0% for modified RUBiS benchmark compared to the state-of-the-art method under certain storage constraint, which simulates a write-intensive workload.

Keywords: NoSQL, schema recommendation, Secondary Index, Binary Integer Programming

1. はじめに

NoSQL データベースはスケールアウトに適したデータ構造やトランザクションの緩和により分散環境において高い性能を達成し [11], [15]，ソフトウェアのバックエンドとして広く使用されている．本稿では NoSQL データベースの中でも Extensible Record Store [2] に分類される Cassandra [14] に着目する．

[1] 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University, Suita, Osaka 565–0871, Japan
[2] 株式会社サイバーエージェント
CyberAgent Inc., Chiyoda, Tokyo 101–0021, Japan
[a)] wakuta.yusuke@ist.osaka-u.ac.jp
[b)] zenmyo_teruyoshi@cyberagent.co.jp
[c)] matsumoto.takumi@ist.osaka-u.ac.jp
[d)] sasaki@ist.osaka-u.ac.jp
[e)] onizuka@ist.osaka-u.ac.jp

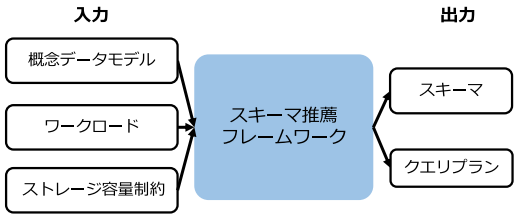


図 1　スキーマ推薦フレームワーク概念図 [21]
Fig. 1　Schema recommendation method [21].

Cassandra では Column Family [*1] に任意の時点でカラムを追加できる．しかし，カラムを追加する対象となる Column Family はあらかじめ定義しなければならない．また，カラムは自由に追加できるが，カラムも含めて設計したスキーマを与えることでより効率的なデータの格納と検索の実現が可能である．したがって，Get や Put の性能を最大化するスキーマの設計が必要となる．また，Cassandra ではクエリ言語として CQL [*2] が導入されており，Get や Put を SQL に近い書式のクエリ処理や更新処理として記述できる．本項で提案するフレームワークは，クエリ処理だけでなく更新処理も多く含むアプリケーションにおいて NoSQL データベースのスキーマ設計を行う．スキーマ設計の課題として，クエリ処理と更新処理の性能のトレードオフがあげられる．たとえば，各クエリごとに実体化結果を格納する Column Family を定義すると，各 Column Family により該当のクエリに応答できるため応答時間を低減できる．この方法の欠点は，クエリごとに単独で応答可能な Column Family を定義すると，一般に複数の Column Family 間でデータの重複が発生するため更新処理のコストが増大することである．一方，データの重複を削減するために，複数のクエリ間で Column Family を共有する方法が考えられる．この場合は，1 クエリを回答する際に複数の Column Family を結合して回答することが可能になる．これにより更新処理の負荷は軽減できるが，複数の Column Family をジョインする必要が生じる．Cassandra を含む，多くの NoSQL データベースではデータベース側でのジョイン処理機能を提供しないため，クライアント側でジョイン処理を行う必要がありクエリの応答時間が大幅に増大する．スキーマ設計は一般的に技術者の経験則により行われているが，クエリやスキーマが複雑になるとクエリ処理と更新処理の性能のトレードオフをバランスする最適なスキーマを設計することは困難になる．また，Cassandra は RDBMS とは異なるデータモデルを採用し，Column Family をジョインする機能を提供していない等，多くの RDBMS との相違点がある．そのため，RDBMS を想定するスキーマ推薦技術 [5], [16], [17] を Cassandra に適用することは困難である．

[*1] 本稿で使用した Cassandra では Column Family は RDBMS のテーブルに相当するデータ構造である．
[*2] http://cassandra.apache.org/doc/latest/cql/

このような背景から，NoSQL データベース向けのスキーマ推薦技術が提案されている [18], [21]．その中の最先端の技術として NoSQL Schema Evaluator（**NoSE**）[21] があげられるが，Secondary Index [24] は考慮していない．Secondary Index を使用することで Column Family のキー以外のカラムを条件として使用するクエリを高速に実行できる．しかし NoSE では，Secondary Index は性能に課題があるとして活用していない．これにより更新処理が多い場合ではスキーマが正規化され，複数の Column Family をクライアント側でジョインするクエリプランが推薦されてしまう場合が生じる．

本稿では，NoSE を拡張することで Secondary Index を活用したスキーマ推薦を行うフレームワークを提案する．Secondary Index の性能の課題としては，まず Secondary Index を用いたクエリ処理は Cassandra ではすべてのデータベースノードに問合せを行うことがあげられる．これにより，クエリ結果の件数がノード数に比べて少ない場合や，該当するデータが 1 つのノードに集中していた場合にオーバヘッドが大きくなる．また，Secondary Index を使用する際は Column Family と同等のデータ構造を経由して対象とする Column Family にクエリ処理を行うためオーバヘッドが生じる．しかし，Cassandra 3.4 以降では従来の Secondary Index の性能を改善した SSTable Attached Secondary Index（**SASI**）[*3] が提供されている．SASI は，Column Family と同等のデータ構造を必要とせずオーバヘッドが低減されている [*4]．したがって，近年ではクエリ処理の高速化のために Secondary Index を用いることが有用となっている．本フレームワークの特徴は Secondary Index を含むスキーマ・クエリプラン候補を列挙し，Secondary Index の特徴を考慮したコストモデルにより実行コストを評価し，最終的に Binary Integer Programming（**BIP**）としてスキーマ推薦問題を解決することにある．このフレームワークを実現するための課題として，クエリプランでの使用方法によって変化する Column Family や Secondary Index のコストを評価することが困難であることがあげられる．本稿では，Column Family を用いるクエリプランとの処理手順の差異を用いることで，Secondary Index を使用するクエリプランのコストを評価する手法を提案する．NoSE ではクライアント側で複数の Column Family をジョインするクエリプランが推薦される場合があったが，提案フレームワークでは Secondary Index を用いるクエリプランを活用できる．これにより，クライアントとデータベース間でのデータ転送回数が減少するため，大幅にクエリの応答時間を低減できる．

[*3] https://docs.datastax.com/en/dse/5.1/cql/cql/cql_using/useSASIIndexConcept.html
[*4] 以降では，明示的に示さない場合 SSTable Attached Secondary Index を Secondary Index と呼ぶ．

評価実験においては，提案フレームワークによって推薦されたスキーマを Cassandra 上に作成し，クエリプランを実行する際の応答時間・データベースノードを増加させた際の応答時間のスケーラビリティの評価を行った．その結果，更新処理の負荷が高い状況に相当するストレージ容量制約を持つワークロードにおいて，応答時間の平均値を既存手法に対して 73.5%低減することを確認した．また，応答時間のスケーラビリティに関する実験により，既存手法よりも優れたスケーラビリティを持つことを確認した．

本稿の構成は以下のとおりである．2 章にて事前知識について説明し，3 章にて提案手法について示す．4 章にて評価実験について説明し，5 章において関連研究について述べ，6 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概念を説明する．2.1 節では，Extensible Record Store について，2.2 節では，Column Family について説明する．また，2.3 節では Secondary Index について説明する．そして，2.4 節では Extensible Record Store におけるスキーマ推薦フレームワークである NoSQL Schema Evaluator について説明する．

2.1 Extensible Record Store

Extensible Record Store [2] のデータモデルは複数のカラムを持つレコードの集合により構成されている．また，Extensible Record Store を使用する際は，より高い性能を得るためにクライアントと複数のデータベースノード群を作成する．データベースノード群にデータを分散する際は，カラム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，カラムは 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．Extensible Record Store の具体例としては Cassandra や HBase [*5]，BigTable [4] が存在する．

2.2 Column Family

Column Family は本稿で対象とする Cassandra では，RDBMS のテーブルに対応するデータ構造である．本稿では式 (1) の書式で Column Family を表現する．

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

partition key は各レコードを識別するためのキーである．Column Family 内の各レコードはその *partition key* の値によって Cassandra のノードに分散して保持される．*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 [*6]．

2.3 Secondary Index

Secondary Index は Cassandra において提供されているデータ構造であり，Column Family の *partition key* 以外のカラムを等号条件として用いるクエリ処理を高速化できる．具体例として，以下の Column Family に対して Secondary Index を使用する場合を考える．

$$CF_1 = CF([partition\ key_1][clustering\ key_1] \rightarrow [value_1, value_2]) \quad (2)$$

$value_1$ に Secondary Index を定義する場合の Secondary Index を式 (3) の書式で記述する．

$$SI_1 = SI([value_1] \rightarrow [partition\ key_1], CF_1) \quad (3)$$

$value_1$ が SI_1 の key, $partition\ key_1$ が value となる．この Secondary Index を使用することで，$value_1$ を等号条件に使用するクエリを高速に処理できる．Secondary Index の各レコードは Secondary Index を定義した Column Family のレコードと同じノードに分散して保持される．更新処理においては，SI_1 と CF_1 はアトミックに更新されるため，レコードの整合性の維持が容易である．

本稿では，Secondary Index の実装として Cassandra 3.4 以降で提供されている SASI を使用する．

2.4 NoSQL Schema Evaluator

NoSQL Schema Evaluator（**NoSE**）[21] は Extensible Record Store を対象とするスキーマ推薦フレームワークである．NoSE の入出力は**図 2** のとおりである．まず，ここでの概念データモデルは制限された Entity-Relationship モデルである．また，ワークロードはクエリ処理や更新処

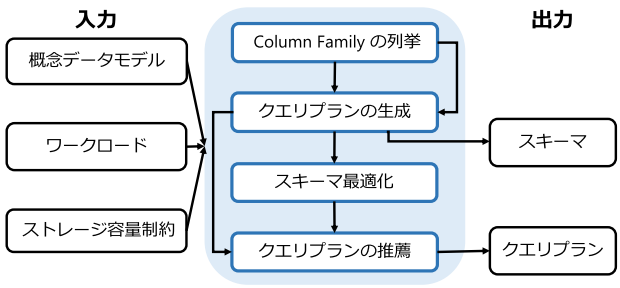


図 2　NoSE 概念図 [21]
Fig. 2　Overview of NoSE [21].

[*5] Apache HBase, http://hadoop.apache.org/hbase

[*6] Cassandra では *partition key* を等号条件として持たないクエリを実行する際，分散する全データベースノードから全レコードを取得してから条件によるフィルタリングをクライアント側で行うため，クエリの応答時間が非常に大きくなる．

理の集合であり，ストレージ容量制約は推薦する Column Family の合計のストレージサイズの上限値である．NoSE では，入力を元に図 2 に示される 4 つの段階に分けて処理を行う．

まず，入力されたクエリ，更新処理集合からそれらに応答する際に使用できる Column Family の列挙を行う．初めに，各クエリに対して Materialized View として使用できる Column Family をそれぞれ列挙する．次に，ワークロード内のクエリに加え，概念データモデルで使用している entity に基づいてそれぞれのクエリを分解したクエリ[*7]や，各クエリの WHERE 句のカラムを SELECT 句に移動して条件を緩和したクエリを生成する．そして，各クエリについて Materialized View としての Column Family を作成する．さらに，key に基づいて複数の Column Family を結合した Column Family も追加で列挙する．

次に，列挙した Column Family を使用して，各クエリ，更新処理集合に応答する際のクエリプランを生成する．クエリプランは Column Family に対する Index Lookup や更新処理等の手順を表すステップで構成される．NoSE は Column Family のみを使用するため，単独の Column Family を用いるプランや，複数の Column Family をジョインするプランを生成する．

次に，生成したクエリプランのコスト計算を行う．コスト計算では，クエリプランを構成する各 IndexLookup ステップごとに式 (4) を用いて計算を行う．

$$cf\_cost = base\_cost + query\_num \times query\_cost + width \times width\_cost \quad (4)$$

式 (4) において，1 項目の $base\_cost$ は Index Lookup を行う際につねに発生するコスト，2 項目はクエリ処理の要求をデータベースに対して行う際のコスト，3 項目が結果のデータ転送コストを表している．$query\_num$ は，実行されるクエリ回数，$query\_cost$ は各クエリを行うコスト，$width$ は結果として取得できるカラムの数，$width\_cost$ はカラム数に対応したコストを表す．2 つ以上の Column Family をジョインするクエリプランでは，1 度目の Index Lookup の結果のサイズを表す $width$ を 2 度目の Index Lookup のコスト関数の $query\_num$ に代入する．これにより，2 つ目の Index Lookup 回数が 1 つ目の Index Lookup の結果件数に依存することを表現する．また，更新処理のコストの計算を式 (5) によって行う．

$$cost = width \times write\_cost \quad (5)$$

$write\_cost$ は更新を行うコストを表す．

次に，BIP を用いた最適化により，ワークロード内の各クエリ処理・更新処理について最適なスキーマとクエリプランを選択する．具体的な目的関数を式 (6) に示す．

[*7] クエリの分割は一度のみ行う．

$$\text{minimize} \sum_i \sum_j f_i C_{ij} \delta_{ij} + \sum_m \sum_n f_m C'_{mn} \delta_n \quad (6)$$

1 項目はクエリ処理の合計コストを表しており，2 項目は更新処理の合計コストを表している．まず，第 1 項目に関して，変数 f_i は入力されるクエリ内の i 番目のクエリの頻度，C_{ij} は i 番目のクエリに対して j 番目の Column Family を使用する際のコストである．また，δ_{ij} は i 番目のクエリが j 番目の Column Family を使用するかどうかを表す 0 もしくは 1 の値を取る変数である．第 2 項目に関して，変数 f_m は入力される更新処理内の m 番目の更新処理の頻度，C_{mn} は m 番目の更新処理によって n 番目の Column Family を更新する際のコストである．また，δ_n は n 番目の Column Family が推薦するスキーマに含まれているかどうかを表す 0 もしくは 1 の値を取る変数である．

式 (6) に対して式 (7)，(8) の制約条件を追加する．

$$\delta_{ij} \leq \delta_j, \quad \forall i, j \quad (7)$$

$$\sum_j s_j \delta_j \leq S \quad (8)$$

制約条件 (7) は使用される Column Family は推薦されることを保証する．制約条件 (8) は各 Column Family のサイズ s_j の総和はストレージ容量制約 S 以下であることを保証する．この制約を用いることでストレージ容量に限りがある場合に対処できるだけでなく，スキーマの正規化の度合いを変更できる．したがってこの制約を厳しくすることで，更新処理が多い場合に対応できるスキーマを推薦できる．また，推薦するクエリプランを構成する Column Family をすべて推薦するための制約も加える．

3. 提案手法

本稿では，Secondary Index を活用するスキーマ推薦フレームワークを提案する．本フレームワークにより，更新処理の多いワークロードにおいても Secondary Index を活用することでクエリの応答時間を低減できる．

3.1 スキーマ推薦におけるクエリプラン分類

本稿では，スキーマ推薦におけるクエリプランを 4 種類に分類する分類法を提案する．スキーマ推薦を行う際は，**図 1** に示した入力を受け取り，ワークロード内の各クエリ処理・更新処理のコストを最小化するスキーマを選択することが課題となる．この際，クエリと更新処理の性能のトレードオフを考慮し，どの程度スキーマを正規化するかが非常に重要な問題となる [25]．スキーマ推薦で使用するクエリプランを以下の 4 種類に分類する．

- 単独の Column Family のみを用いて応答するクエリプラン（**実体化プラン**）
- 正規化された複数の Column Family を用いて応答するクエリプラン（**ジョインプラン**）

ソースコード 1　クエリプラン例

```
1  --クエリ 1
2  SELECT id, firstname, lastname, password, email
3  FROM user
4  WHERE firstname = ?
5
6  --実体化プラン
7  Index Lookup Column Family A:
8      CF([user.firstname][user.id] →
9          [user.lastname, user.password, user.email])
10
11 --ジョインプラン
12 Index Lookup Column Family B:
13     CF([user.firstname] [user.id] → [user.email])
14 Index Lookup Column Family C:
15     CF([user.id] [ ] → [user.lastname, user.password])
16
17 -- Secondary Index プラン
18 Index Lookup Secondary Index D:
19     SI([user.firstname] → [user.id],CF_E)
20 Index Lookup Column Family E:
21     CF([user.id] [ ] →
22         [user.firstname, user.lastname, user.password, user
               .email])
23
24 --クエリ 2
25 SELECT lastname FROM user WHERE firstname = ?
26
27 --単独 Secondary Index プラン
28 Index Lookup Secondary Index F:
29     SI([user.firstname] → [user.lastname],CF_X)
```

- Column Family と Secondary Index を組み合わせて応答するクエリプラン（**Secondary Index プラン**）
- 単独の Secondary Index を使用するクエリプラン（**単独 Secondary Index プラン**）

ソースコード 1 にそれぞれのクエリプランの例を示す．ただし，Index Lookup ステップはキーの値を等号条件としたクエリ操作を表す．

実体化プラン（6–9 行目）は，クエリ結果を保持する Column Family をあらかじめ用意する方法であり，1 つの Column Family にアクセスすることでクエリ結果を取得できるため応答時間が短い．ただし，各クエリに対して実体化プランを生成すると，データが非正規化される．これにより，同一のデータが複数の Column Family に重複して存在する可能性が生じる．そのため，更新処理を行う際に重複データ間の整合性を保つ負荷が増加する．

ジョインプラン（11–15 行目）は各 Column Family が正規化されるため重複データを削減できる．そのため，更新処理が容易である．また，Column Family が正規化されているためストレージ容量の削減も可能となる．しかし，このクエリプランでは Column Family B，Column Family C に関して，入れ子ループ結合 [22] に相当する処理を行う必要がある．RDBMS における入れ子ループ結合では，2 つのテーブルをジョインする際，データベース内において 1 つ目のテーブルのそれぞれのレコードに対して 2 つ目のテーブルの全レコードを比較し，条件に一致するものを取得する．一般的な NoSQL データベースでは，この処理をクライアント側で行う必要がある．そのため，入れ子ループ結合におけるレコードの比較回数分の通信処理が必要となり，実行コストが増加する．

Secondary Index プラン（17–22 行目）は，Column Family E の $user.firstname$ に Secondary Index D を定義することを表している．Secondary Index D に対する Column Family E を**後続の Column Family** と定義する．このクエリプランは，Secondary Index を用いることで，ジョインプランと同様にデータを正規化できる．Secondary Index プランを実行する際は，データベースノード内において Secondary Index のキーのカラム（**Secondary Index カラム**）を用いて後続の Column Family の *partition key* が取得される．よって，データベースノードに 1 度のみクエリを実行することで最終的な結果を得る．したがって，ジョインに相当する処理を行う必要がなく，クエリ処理を高速化できる[*8]．

単独 Secondary Index プラン（27–29 行目）はクエリ 2 に対して単独で応答可能な Secondary Index F を用いる．このプランを用いることで Column Family 上のレコードの重複を低減できる[*9]．

NoSE [21] は Secondary Index は性能に問題があるとして Column Family のみの手法を提案しているが，Secondary Index プランはジョインプランのようにクライアント側でのジョイン処理を行わないため応答時間の低減が見込める．また，Cassandra 3.4 以上で提供されている SASI を活用することで，実体化プランに比べても Secondary Index プランのオーバヘッドを低く抑えることができるため Secondary Index の活用は有用である．

3.2　フレームワークの概要

NoSE [19], [21] に対して Secondary Index の列挙ステップを追加し，クエリプランの生成ステップやコスト計算ステップ，最適化の際の制約条件にも Secondary Index を活用するための変更を加えた．入力は NoSE と同様であるが，出力は Secondary Index の活用する点が NoSE とは異なる．フレームワークの概念図を**図 3** に示す．また，処理の概要は以下のとおりである．

(1) 概念データモデルおよびワークロードから，Column Family を列挙する．

[*8] Column Family E のデータの更新を行った際は Secondary Index D の該当するレコードの更新処理もアトミックに行われるため，データの整合性を保つユーザの負荷を低減できる．

[*9] 本稿の評価において使用した Cassandra では，単独で Secondary Index を使用できないため，この Secondary Index を定義できる Column Family（CF_X）を決定しなければならない．

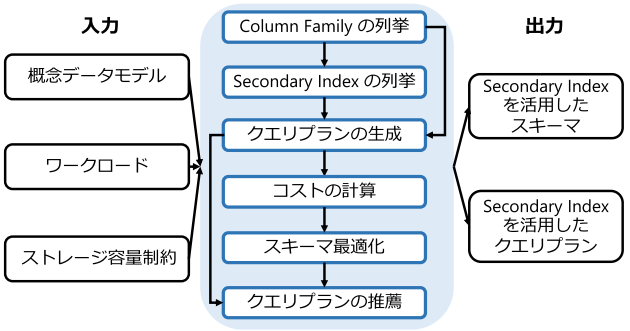


図 3　提案手法概念図．図 2 に対して Secondary Index の列挙ステップを追加し，クエリプランの生成ステップやコスト計算ステップの処理内容にも Secondary Index を活用するための変更を加えた

Fig. 3　Overview of the proposed method. Secondary Index enumeration step is added to Fig. 2. Both query plan enumeration step and cost estimation step are extended to utilize Secondary Index.

(2) 各 Column Family を元に，Secondary Index と後続の Column Family の組を列挙する．
(3) 列挙した Column Family，Secondary Index を用いてクエリプランを列挙する．
(4) クエリプランの各ステップについて，実行コストを見積もる．
(5) BIP を用いることで各クエリについて最適なクエリプランを選択する．

本節では，Secondary Index を活用するために NoSE の操作を変更した箇所について詳しく説明する．

3.3　Secondary Index の列挙

NoSE が列挙した Column Family を用いて，Secondary Index プランおよび単独 Secondary Index プランで使用する Secondary Index を列挙する．具体的な Secondary Index の列挙の手順を Algorithm 1 に示す．Algorithm 1 では，Column Family を入力として受け取り，Secondary Index と後続の Column Family を出力する．*generate_si*() メソッドは第 1 引数を *key*，第 2 引数を *value* として持つ Secondary Index を生成する．各 Column Family に対して，以下の 2 種類の Secondary Index の列挙を行う．

- primary key を value として持つ Secondary Index（4–13 行目）
- 非 primary key を value として持つ Secondary Index（14 行目）

1 つ目の Secondary Index は，Secondary Index プランにおいて使用される．ここで，クエリプランの数を削減するため，Secondary Index プランでは Secondary Index の value は概念データモデルの entity の primary key でなければならないという制約を設けている．そのため，Secondary Index の後続として使用する Column Family を

Algorithm 1: Secondary Index 列挙アルゴリズム

Input : Column Family（**CF**），extra.entity.primary は extra が概念データモデルにおいて属する entity の primary key，key.entity.primary も同じく key が概念データモデルにおいて属する entity の primary key を表す
Output: Secondary Index と後続の Column Family

```
   /* list of column families                          */
1  cfs ← []
   /* list of secondary indexes                        */
2  sis ← []
   /* iterate for non-partition key column of CF       */
3  foreach (extra) ∈ (CF.clustering_key + CF.value) do
       /* iterate for partition key of CF              */
4      foreach (key) ∈ (ColumnFamily.partition_key) do
5          if key != extra.entity.primary then
6              si1 = generate_si(key,extra.entity.primary)
7              cfs += generate_additional_cf(si1)
8              sis += si1
9          end
10         if key != key.entity.primary then
11             si2 = generate_si(key,key.entity.primary)
12             cfs += generate_additional_cf(si2)
13             sis += si2
14         end
15         sis += generate_si(key,extra)
16     end
17 end
18 return distinct(sis), distinct(cfs)
```

generate_additional_cf メソッドにより追加で生成する（7，12 行目）．具体的には，非 *partition key* カラムに Secondary Index のキーを追加し，*partition key* に Secondary Index の *value* を持つ Column Family を生成する．これにより，Secondary Index の後続の Column Family が存在することを保証する．また，2 つ目の Secondary Index は単独 Secondary Index プランに使用する．

具体例として，以下の Column Family G に対して Secondary Index と後続の Column Family を列挙する場合を考える．

$$CF_G = CF([user.firstname][user.id] \rightarrow [user.lastname])$$

まず，Secondary Index プランのための Secondary index を列挙する．変数 *key*，*extra* はそれぞれ以下の値を取る．

$$key \in \{user.id, user.lastname\}$$
$$extra \in \{user.firstname\}$$

まず，5–9 行目の条件分岐において，それぞれの extra の属する user entity の primary key である *user.id* について Sec-

ondary Index を作成する[*10]．そして，*user.id* を partition key として持つ後続の Column Family を作成する．

$$SI([user.firstname] \rightarrow [user.id], CF_X)$$
$$CF([user.id][] \rightarrow [user.firstname, user.lastname])$$

また，10–14 行目の条件分岐において，*key* の属する entity についても同様の処理を行う．この例では，5–9 行目の条件分岐と同じ Secondary Index と Column Family を作成する．次に，15 行目において *key* と *extra* のすべての組合せについて Secondary Index を列挙する．この Secondary Index は単独 Secondary Index プランのために使用する．

$$SI([user.firstname] \rightarrow [user.id], CF_X)$$
$$SI([user.firstname] \rightarrow [user.lastname], CF_X)$$

そして，18 行目において重複を排除して出力する．

3.4 Secondary Index プランの生成

NoSE の列挙するクエリプランに加えて，Secondary Index プランを列挙する．Secondary Index に対する Index Lookup ステップは，結果をクライアントに転送することなく，後続の Column Family に対する Index Lookup を行う．さらに，2 つ以上の Column Family を使用する Secondary Index プランにおいては 3.1 節で述べたエンコーディング方法を再帰的に使用する．

Secondary Index プランは，Secondary Index のキーが後続の Column Family の非 *partition key* であり，*value* が後続の Column Family の *partition key* でなければならないことを考慮し列挙する．ただし，Secondary Index プランを実行する際は Secondary Index と後続の Column Family のそれぞれに対する Index Lookup を後続の Column Family に対する 1 つの Index Lookup として扱う．また，単独 Secondary Index プランは他のクエリプランに後続の Column Family が存在するとして列挙する．

3.5 コストの計算

Column Family と Secondary Index に対する Index Lookup についてそれぞれ異なるコストモデルを用いることで，Secondary Index の特性を考慮したコストの評価を行う．ここでの課題として，クエリプランでの使用方法によって変化する Column Family や Secondary Index のコストを評価することが困難であることがあげられる．本稿では，Column Family を用いるクエリプランとの処理手順の差異を用いることで，Secondary Index を使用するクエリプランのコストを評価する手法を提案する．Secondary Index プランはジョインプランと比較し，単独 Secondary Index プランは実体化プランと比較することでコスト計算を行う．ただし，実体化プランとジョインプランのコストの評価では，NoSE と同じ手法を使用する．

[*10] この時点では，Secondary Index の後続の Column Family は確定していないため，CF_X と記述する．

Algorithm 2: クエリプランのコスト計算

```
Input  : base_cost, query_plan, query_num, query_cost,
         width, width_cost, cf_query_cost, si_query_cost
Output: コスト計算済みの query_plan
1  foreach (step) ∈ query_plan do
2      if step is alone and step is secondary_index then
3          step.cost = (base_cost +
             query_num × query_cost + width × width_cost)
             * (si_query_cost/cf_query_cost)
4      end
5      if step is secondary_index then
6          width_cost = width_cost × ((si_query_cost −
             cf_query_cost)/(cf_query_cost × width))
7      end
8      else if step.prev is secondary_index then
9          query_cost = query_cost × ((si_query_cost −
             cf_query_cost)/(cf_query_cost × width))
10     end
11     step.cost = base_cost +
         query_num × query_cost + width × width_cost
12 end
13 return query_plan;
```

Algorithm 2 の手順で単独 Secondary Index プラン，Secondary Index プランのコスト評価を行う．入力として式 (4) で定義した変数に加え，*cf_query_cost*，*si_query_cost* を与える．ここで，式 (4) と同様の変数は NoSE で用いられていた値をそのまま使用した．*cf_query_cost* は *partition key* を等号条件として用いて Column Family に Index Lookup を行う応答時間の実測値である．また，*si_query_cost* は Secondary Index を定義した Column Family に対して，Secondary Index カラムを等号条件として用いる Index Lookup を行う応答時間の実測値である．

まず，式 (9) により単独 Secondary Index プランのコスト計算を行う（2–4 行目）．

$$cost = cf\_cost \times (si\_query\_cost/cf\_query\_cost) \quad (9)$$

単独の Column Family を使用する場合に比べ，Secondary Index を使用する場合にはコストが増加する．そのため，単独の Column Family を使用する際のコスト（*cf_cost*）に Secondary Index を使用する際のコスト増加分をかけ合わせることでコストの評価を行う（3 行目）．

次に，Secondary Index プランのコスト計算を行う（6–10 行目）．2 つの Column Family を使用するジョインプランとの処理手順を比較することでコストの評価を行う．ジョインプランにおいては Column Family へクエリを行う度にクライアントとの通信を行う．また，2 つ目の Index Lookup は 1 つ目の Index Lookup の結果件数と同じ回数行う．一

方，Secondary Index プラン内の Secondary Index から後続の Column Family へのクエリ処理ではクライアントとの通信を行わない．また，Index Lookup 回数は 1 度のみである．したがって，Secondary Index プランにおいて，1 つ目のステップからの結果の転送と，2 つ目のステップへのクエリ処理におけるコストがジョインプランに比べ小さい．このコストの低減率を用いることで，Secondary Index プランの処理コストを見積もる．具体的には，式 (4) の定数 *query_cost*，*width_cost* の値を Secondary Index を活用することによるコストの減少をふまえたコストに再設定する．該当するステップが Secondary Index に対する Index Lookup である場合は結果をクライアントに転送しないため，*width_cost* を低く見積もる（6 行目）．また，1 つ前のステップが Secondary Index に対する Index Lookup である場合はクライアントからクエリ処理を取得しないため，*query_cost* を低く見積もる（9 行目）．これらの再設定したコストを使用して式 (4) の計算を行うことで各ステップのコストを取得する（11 行目）．

次に，Secondary Index を考慮して更新処理のコストを評価する．ここで，Secondary Index が定義されている Column Family を更新する場合，Column Family 上のレコードと該当する Secondary Index のためのデータをアトミックに更新する．したがって，Secondary Index を定義することにより後続の Column Family の更新処理は応答時間が増加する．この応答時間の増加を評価するために，Secondary Index は *key* と *value* の 2 つのカラムを持つ Column Family と同様であると見なして更新コストの評価を行う．これにより，Secondary Index による更新処理負荷の増加を考慮する．

3.6　スキーマ最適化

スキーマ最適化では，Secondary Index を Column Family として扱うことで NoSE と同様の BIP 目的関数や制約式を使用する．したがって，最適化処理においてはジョインプランと Secondary Index プランは区別しない．さらに，単独 Secondary Index プランを推薦する際は，後続の Column Family も推薦するスキーマ内に含まれるように，スキーマ最適化の制約条件を各 Secondary Index に対して追加する．具体的な制約の例としてソースコード 1 における Secondary Index プランについて式 (10) の制約条件を作成する．また，ここでは Column Family F も Secondary Index D の後続の Column Family として使用できる場合を考える．

$$\delta_D \leq \delta_E + \delta_F \tag{10}$$

式 (10) は Secondary Index D を推薦するならば，Column Family E，Column Family F のいずれかは推薦しなければならないという制約を表している．これにより，推薦される Secondary Index は必ず後続の Column Family を持つ．また，更新処理において Secondary Index は，後続の Column Family を更新することでアトミックに更新処理が行われるため，Secondary Index に対して更新処理を行うクエリプランの推薦は行わない．このスキーマ最適化の結果，各クエリに対して使用するスキーマとクエリプランが確定する．

ソースコード **2**　入力

```
1  -- 概念データモデル: user entity
2  CREATE TABLE user(
3          id integer PRIMARY KEY,
4          firstname text,
5          lastname text,
6          password text );
7  --クエリ
8  SELECT id,firstname,lastname,password FROM user
       WHERE user.id = ?
9  SELECT id,firstname,lastname,password FROM user
       WHERE user.firstname = ?
```

ソースコード **3**　NoSE の出力

```
1  --スキーマ
2   Column Family H:
3      CF([user.id] [] → [user.firstname, user.lastname, user.
           password])
4   Column Family I:
5      CF([user.firstname] [user.id] → [])
6
7  --クエリプラン
8  SELECT id,firstname,lastname,password FROM user
       WHERE user.id = ?
9      -- Index Lookup Column Family H
10 SELECT id,firstname,lastname,password FROM user
       WHERE user.firstname = ?
11     -- Index Lookup Column Family I
12     -- Index Lookup Column Family H
```

ソースコード **4**　提案手法の出力

```
1  --スキーマ
2   Column Family J:
3      CF([user.id] [ ] → [user.firstname, user.lastname, user.
           password])
4   Secondary Index K:
5      SI([user.firstname] [ ] → [user.id],CF_I)
6
7  --クエリプラン
8  SELECT id,firstname,lastname,password FROM user
       WHERE user.id = ?
9      -- Index Lookup Column Family J
10 SELECT id,firstname,lastname,password FROM user
       WHERE user.firstname = ?
11     -- Index Lookup Secondary Index K
12     -- Index Lookup Column Family J
```

3.7　入出力例

ソースコード 2 で定義される user entity のみで構成される概念データモデルと 2 つのクエリが入力される場合を

考える．更新処理の多い場合を想定するため，容量制約を用いてスキーマ推薦を行った．NoSE はソースコード 4 で表されるスキーマとクエリプランを推薦する．このクエリプランでは 1 つのクエリに対してジョインプランを推薦している．一方，提案手法はソースコード 4 で表されるスキーマとクエリプランを推薦する．この結果から NoSE ではジョインプランを推薦するクエリに対して Secondary Index プランを推薦していることが確認できる．

4. 評価実験

本章では提案した Secondary Index を活用する NoSQL スキーマ推薦フレームワークの有用性を評価する．

4.1 実験環境

評価実験として，スキーマを手動で正規化する手法（**ベースライン**），NoSE，提案手法の 3 種類の手法について性能評価を行った．ベースラインは NoSE の論文 [21] において性能評価で使用されているものと同様の手順で作成した．具体的には，まず，入力された概念データモデルの各 entity に対応する Column Family を作成する．さらに，クエリの等号条件を *partition key* として持ち，各 entity に対応する Column Family の *partition key* を取得できる Column Family も作成する．そして，これらの Column Family を複数組み合わせてクエリに応答する．NoSE は GitHub [20] に公開されている NoSE の著者による実装を使用した．目的関数についてそれぞれ最適化した NoSE と提案手法のスキーマとベースラインのスキーマを Cassandra 上に作成し，クエリプランを実行する際の応答時間とデータベースのノード数における応答時間のスケーラビリティに関して評価を行った．

本稿ではベンチマークとしてオークションサイトを想定した Rice University Bidding System（**RUBiS**）[3] を使用する．RUBiS は 7 つの entity で構成される概念データモデル，頻度の与えられているクエリ，更新処理で構成されている．本稿で提案するフレームワークは，クエリ処理に加え更新処理を多く含む環境を対象としている．そのため，RUBiS のワークロードの中でも更新処理を含む bidding ワークロードを使用する．さらに，更新処理が多く行われる場合について評価を行うために，容量制限を設けて評価を行った．ただし，RUBiS のクエリは Secondary Index の有用性を確認するために最適ではない．そのため，WHERE 句に 1 つのみ等号条件を持つクエリに関して，SELECT 句，FROM 句はそのままで WHERE 句の等号条件で使用するカラムのみ変更したクエリを追加する．追加で生成したクエリの実行頻度は複製元のクエリと同様であるとする．user entity のレコード数が 200,000 件で作成した RUBiS のレコードを各手法により推薦されたスキーマに変換し，Cassandra 上に作成した．

表 1　AWS の実験環境
Table 1　Evaluation environment in AWS.

設定項目	設定値
インスタンスタイプ	c4.8xlarge
リージョン	米国西部（オレゴン）
cpu	Intel Xeon CPU E5-2666 v3 @ 2.90 GHz
cpu 論理コア数	36
マシン台数	1, 10, 20
メモリ（GiB）	60
OS	ubuntu 16.04
ネットワークパフォーマンス（Gbit）	10

表 2　最適化を行う手法により推薦されたクエリプラン数の内訳
Table 2　Statistics of the query plans recommended by each method.

	ベースライン	NoSE	提案手法
実体化プラン	15	42	37
ジョインプラン	38	11	1
Secondary Index プラン	-	-	15

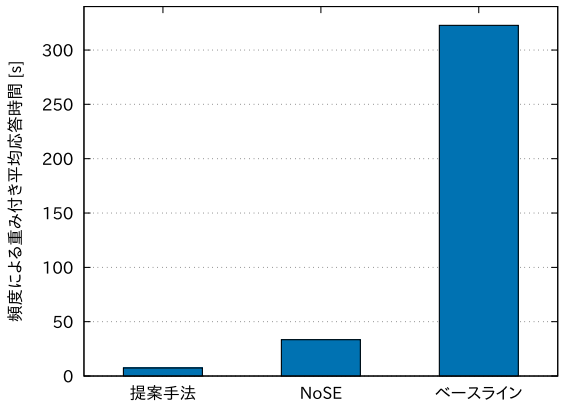


図 4　ワークロード内のすべてのクエリ処理・更新処理の頻度による重み付き平均応答時間．図 5 に示した各手法の応答時間の平均値に対応している
Fig. 4　Average frequency-weighted latency of all transactions (See in Fig. 5) for three approaches (the proposed method, NoSE, baseline).

本実験では，実験環境として Amazon Web Service（AWS）を使用する．具体的な環境を**表 1** に示す．また，Cassandra のバージョンは 3.11 であり，マシンごとにノードを作成した．Replication Factor はノード数が 3 より少ない場合は 1 とし，3 以上の場合は 3 とした．また，Replication Strategy は SimpleStrategy を使用し，Consistency Level は ONE を使用した．クエリを実行するクライアントは Cassandra と同様の AWS のリージョンに作成した．これにより，データベースノード間と，データベースノードとクライアント間のネットワーク環境の差異による性能への影響を低減した．

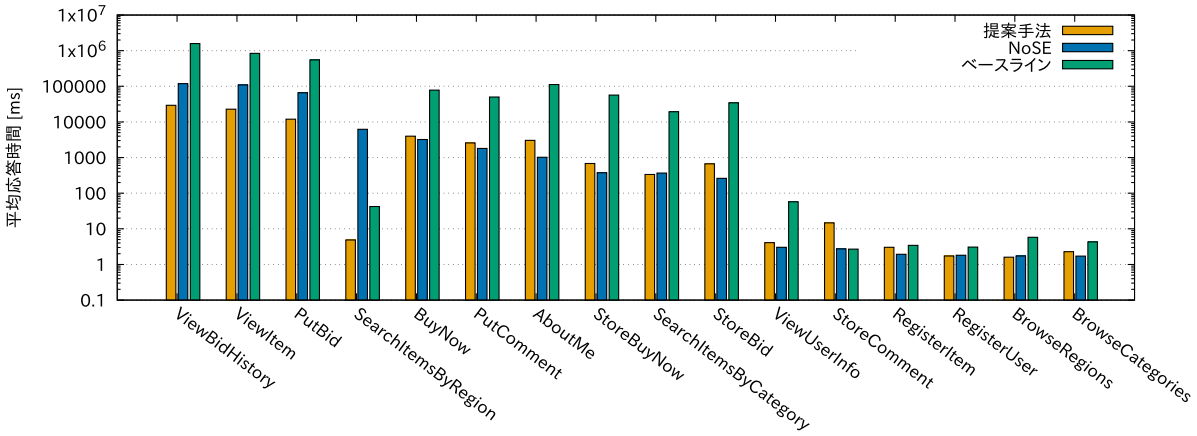


図 5　問合せ集合ごとの応答時間．NoSE やベースラインの応答時間の長いトランザクションにおいて応答時間を低減し，ベースラインや NoSE で応答時間の短いクエリには同様に短い応答時間を達成している

Fig. 5　Average latency of each transaction for three approaches (the proposed method, NoSE, baseline). The proposed method performs higher than NoSE and baseline in latency for long-running transactions and it is comparable for short-running transactions.

4.2　実験結果

4.2.1　応答時間

ワークロード内のクエリ処理に対する NoSE と提案手法が推薦したクエリプランの内訳を**表 2** に示す．また，**図 4** にそれぞれのスキーマに対してクエリ処理を行った際の頻度による重み付き平均応答時間を示す．Cassandra のデータベースノードは 1 つで測定した．評価結果として，ベースラインと NoSE に対して，提案手法はそれぞれ 97.7%，78.0%の重み付き平均応答時間の低減を達成した．表 2 より，NoSE は提案手法に比べてジョインプランを複数推薦したことで，クエリの応答時間が増加したと考えられる．一方，提案手法では Secondary Index プランを活用することで応答時間を低減している．

図 5 に RUBiS の bidding ワークロード内のそれぞれのトランザクションの応答時間を示す．クエリの頻度はトランザクションごとに入力されるため，図 5 に示した値は各トランザクションに属するクエリ処理や更新処理の平均応答時間である．NoSE やベースラインが特に応答時間が長いクエリに対して応答時間を低減している．ベースライン・NoSE・提案手法の最も応答に時間の掛かるクエリの応答時間はそれぞれ 6,051.4 秒，470.9 秒，109.6 秒であった．これにより，提案手法はスキーマ推薦において特定のクエリの応答時間に最大値が設定されるような状況においても有益であるといえる．また，ベースラインや NoSE において応答時間の短いクエリに対しては同様に短い応答時間を提供している[*11]．一部のトランザクションに関して詳細な考察を行う．4 つの SELECT 句を持つ ViewBidHistory において NoSE に対して応答時間を短縮した．NoSE によるクエリプランでは 1 つの SELECT 句においてジョインプランを用いる．一方，提案手法によるクエリプランではジョインプランを使用せず，Secondary Index プランを活用することで応答時間を短縮している．

SELECT 句を 1 つ持つ SearchItemsByRegion においても NoSE に対して応答時間を低減することに成功した．NoSE ではジョインプランを推薦しているのに対して，提案手法では実体化プランを推薦した．他のクエリにおいて Secondary Index を活用することでストレージ容量制約に余裕ができ，実体化プランを推薦できたと考えられる．

StoreComment では，NoSE に対して提案手法の応答時間が増加していた．StoreComment は 2 つの SELECT 句とそれぞれ 1 つの UPDATE 句と INSERT 句を持つ．SELECT 句，INSERT 句に対して推薦するクエリプランでは NoSE と提案手法の大きな違いは確認できなかった．しかし，UPDATE 句にともない更新処理を行う Column Family が NoSE では 3 つであるのに対して，提案手法では 6 つであった．これにより，応答時間が増加したと考えられる．更新対象の Column Family が増加した理由として，提案手法では Secondary Index を活用することで，NoSE に比べ Column Family の正規化の度合いが低い可能性が考えられる．

4.2.2　データベースのノード数に関するスケーラビリティ

図 4 より，提案手法と NoSE がベースラインに比べ十分に高速であることが確認できた．そのため，データベースのノード数に関するスケーラビリティの計測ではベースラインの計測を省いた．また，ワークロードは 4.1 節で述べた Secondary Index の評価のために拡張した RUBiS を

[*11] 本実験の結果は，NoSE の論文 [21] 中の図 17 に示される RUBiS を使用した実験結果とは応答時間が異なる．ここで，本実験で使用したワークロードは RUBiS を 4.1 節で示した方法で拡張したものである．したがって，使用したワークロードが NoSE における実験とは異なるため図 5 に示した結果を得たと考えられる．

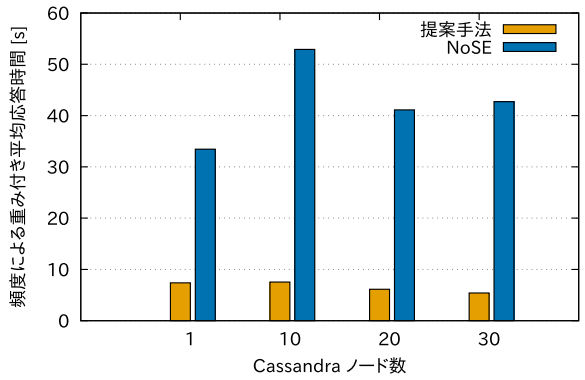


図 6　提案手法と NoSE とのデータベースのノード数に関するスケーラビリティの比較．NoSE ではノード数が増加すると応答時間が増加しているが，提案手法では応答時間は減少している

Fig. 6　Scalability of the proposed method and NoSE. The latency of NoSE increases as the number of computer nodes increases. In contrast, the latency of the proposed method decreases as the number of computer nodes increases.

使用した．Cassandra のノード数を 1，10，20，30 と変化させ，ワークロード内のクエリの頻度による重み付き平均応答時間を計測した結果を**図 6** に示す．NoSE はノード数が増加すると応答時間が増加する傾向が見られたが，提案手法ではあまり見られないことを確認できた．表 2 より，NoSE は通信回数の多いジョインプランを多く推薦していることが確認できる．そのため，ノード数の増加にともなう通信コストの増加により応答時間が増加していると考えられる．一方，ノード数を 10 から 20 に増加させた場合に NoSE の応答時間が減少している．本実験では，ノード数によらず一定数のレコードをデータベースに挿入した．したがって，データベースのノード数が増加すると各ノード内に保存されているレコード数が減少するため，クエリ処理時に各ノード内でレコードの探索に要する時間が減少する．この結果，10 ノードから 20 ノードにノード数を増加させた際に応答時間の平均値が減少したと考えられる．一方，20 ノードから 30 ノードにノード数を増やした際には，ジョインプランによって多くの通信が実行されることによるオーバヘッドが大きくなり，応答時間が増加したと考えられる．提案手法の推薦したクエリプランでは通信回数が少ないため，ノードの増加による通信量の増加の影響が小さく，短い応答時間を達成できた[*12]．

5.　関連研究

概念データモデルやクエリ・更新処理集合等のワークロードから，最適なスキーマを推薦する研究は RDBMS において広く行われている [1], [6], [8], [9], [10], [13], [23], [26]．Harinarayan ら [10] は，ストレージ容量制限下で greedy なアルゴリズムにより Materialized View を選択し，OLAP クエリの応答時間を短縮している．また，Gupta ら [9] の研究と Agrawal ら [1] の研究は Materialized View だけでなく，Index も考慮した greedy なアルゴリズムを使用することでより性能の高いスキーマを実現している．特に，Agrawal ら [1] は Materialized View に対して定義する Index に加え，実テーブルに対して定義する Index も対象としている．Agrawal らは対象とする探索空間が非常に大きくなる問題を解決するために Materialized View をコストベースで枝刈りする方法についても提案している．しかし，greedy な手法では推薦したスキーマがどの程度最適か知ることができない．

Integer Linear Programming を用いて Index の推薦を行う研究では，推薦したスキーマの精度を把握することが可能となる [7], [12], [13], [23]．Papadomanolakis らの研究 [23] では，本稿のフレームワークと NoSE で用いる目的関数と同様に各 Index に関して推薦するかどうかの変数を割り当て最適化を行う．ただし，Integer Linear Programming による手法は探索に必要な時間等の観点から，数万のクエリを含むような非常に大きいワークロードに対して適用することは困難である．この問題に対する研究として BIGSUBS [13] が提案されている．BIGSUBS は複数のクエリの共有部分を探索し Materialize することでクエリの応答時間を低減する．2 部グラフのラベリング問題を用いて Integer Linear Programming による最適化問題を細分化し，並列に最適化を行うことで非常に大きなワークロードに対してもスキーマ推薦が可能となっている．

NoSQL データベースにおけるスキーマ推薦ではワークロードから Column Family の正規化の度合いや Secondary Index の使用の有無を確定することが課題となる．本稿で提案するフレームワークにおいて使用した NoSQL Schema Evaluator（NoSE）[21] は更新処理の多い環境ではジョインプランを推薦し応答時間が著しく増加するが，更新処理の少ない環境では性能の高いスキーマの提案を行う．NoSE と比較して本稿で提案するフレームワークの重要な貢献点は，BIP による最適化の候補に Secondary Index を使用するクエリプランを加え，Secondary Index の特性を考慮したコスト計算を行う点である．

6.　結論

本稿では，Secondary Index を活用する NoSQL スキーマ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 Column Family，Secondary Index を用いたクエリプランを生成し，BIP によって最適化したスキーマとクエリプランを推薦する．既存手法では，更新処理が多く存在する際には，クエリの応答時間が著しく増加する場合が存在するが，提案手法では Secondary Index を適切に使用する

[*12] 本来，スケーラビリティを実環境で評価するには，マシンの負荷を考慮した上でスループットの評価が必要であるが，本稿の評価軸はクエリの実行コストであるため，スループットの評価は今後の課題とする．

ことで応答時間の低減を達成した．推薦したスキーマをCassandra上に作成し，推薦したクエリプランを用いた評価実験を行った結果，更新処理の多いベンチマークにおいて既存手法に比べクエリの応答時間を低減することを確認できた．また，今後の課題として入れ子クエリやGROUP BY等のより幅広いクエリへの対応が考えられる．

謝辞　本研究はJSPS科研費JP17H06099およびJP18H04093の助成を受けたものです．

参考文献

[1] Agrawal, S., Chaudhuri, S. and Narasayya, V.R.: Automated Selection of Materialized Views and Indexes in SQL Databases, *VLDB*, No.5, pp.496–505 (2000).

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

[3] Cecchet, E., Marguerite, J. and Zwaenepoel, W.: Performance and scalability of EJB applications, *ACM SIGPLAN Notices*, pp.246–261 (2002).

[4] Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A. and Gruber, R.E.: Bigtable: A distributed storage system for structured data, *Trans. Computer Systems* (2008).

[5] Chaudhuri, S., Gupta, A.K. and Narasayya, V.: Compressing SQL workloads, *SIGMOD*, pp.488–499 (2002).

[6] Dageville, B., Das, D., Dias, K., Yagoub, K., Zait, M. and Ziauddin, M.: Automatic SQL Tuning in Oracle 10G, *Proc. 30th International Conference on Very Large Data Bases - Volume 30*, *VLDB*, pp.1098–1109 (2004).

[7] Dash, D., Polyzotis, N. and Ailamaki, A.: CoPhy: A Scalable, Portable, and Interactive Index Advisor for Large Workloads, *Proc. VLDB*, Vol.4, pp.362–372 (2011).

[8] Finkelstein, S., Schkolnick, M. and Tiberio, P.: Physical Database Design for Relational Databases, *ACM Trans. Database Syst.*, Vol.13, No.1, pp.91–128 (1988).

[9] Gupta, H., Harinarayan, V., Rajaraman, A. and Ullman, J.D.: Index Selection for OLAP, *ICDE* (1997).

[10] Harinarayan, V., Rajaraman, A. and Ullman, J.D.: Implementing Data Cubes Efficiently, *SIGMOD*, pp.205–216 (1996).

[11] Hecht, R. and Jablonski, S.: NoSQL evaluation: A use case oriented survey, *CSC*, pp.336–341 (2011).

[12] Heeren, C., Jagadish, H.V. and Pitt, L.: Optimal Indexing Using Near-minimal Space, *Proc. 22nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, *PODS '03*, pp.244–251 (2003).

[13] Jindal, A., Karanasos, K., Rao, S. and Patel, H.: Selecting Subexpressions to Materialize at Datacenter Scale, *Proc. VLDB*, Vol.11, No.7, pp.800–812 (2018).

[14] Lakshman, A. and Malik, P.: Cassandra: A Decentralized Structured Storage System, *SIGOPS*, Vol.44, No.2, pp.35–40 (2010).

[15] Li, Y. and Manoharan, S.: A performance comparison of SQL and NoSQL databases, *PACRIM*, pp.15–19 (2013).

[16] Liu, S., Song, B., Gangam, S., Lo, L. and Elmeleegy, K.: Kodiak: Leveraging materialized views for very low-latency analytics over high-dimensional web-scale data, *Proc. VLDB*, Vol.9, No.13, pp.1269–1280 (2016).

[17] Mami, I. and Bellahsene, Z.: A survey of view selection methods, *SIGMOD Record*, Vol.41, pp.20–29 (2012).

[18] Mior, M.J.: Automated schema design for NoSQL databases, *SIGMOD PhD Symposium*, pp.41–45 (2014).

[19] Mior, M.J., Salem, K., Aboulnaga, A. and Liu, R.: NoSE: Schema design for NoSQL applications, *ICDE*, pp.181–192 (2016).

[20] Mior, M.J.: Nose - automated schema design for nosql applications (online), available from ⟨https://github.com/michaelmior/NoSE⟩.

[21] Mior, M.J., Salem, K., Aboulnaga, A. and Liu, R.: NoSE: Schema design for NoSQL applications, *TKDE*, Vol.29, No.10, pp.2275–2289 (2017).

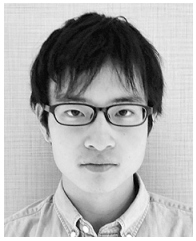
[22] Mishra, P. and Eich, M.H.: Join processing in relational databases, *CSUR*, pp.63–113 (1992).

[23] Papadomanolakis, S. and Ailamaki, A.: An Integer Linear Programming Approach to Database Design, *ICDE*, pp.442–449 (2007).

[24] Qader, M.A., Cheng, S. and Hristidis, V.: A Comparative Study of Secondary Indexing Techniques in LSM-based NoSQL Databases, *SIGMOD*, pp.551–566 (2018).

[25] Sanders, G.L. and Shin, S.: Denormalization effects on performance of RDBMS, *HICSS* (2001).

[26] Zilio, D.C., Rao, J., Lightstone, S., Lohman, G.M., Storm, A.J., Garcia-Arellano, C. and Fadden, S.: DB2 Design Advisor: Integrated Automatic Physical Database Design, *VLDB* (2004).



涌田 悠佑

2019年大阪大学工学部電子情報工学科卒業．同大学大学院情報科学研究科在学中．

善明 晃由　（正会員）

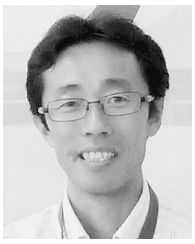
2004年東京工業大学大学院情報理工学研究科計算工学専攻修士課程修了．総合電機メーカーを経て，現在は株式会社サイバーエージェントにて大規模データ処理基盤・データマネージメントの研究開発に従事．博士（工学）．電子情報通信学会，ACM，IEEE Computer Society各会員．

松本 拓海

2018年大阪大学工学部電子情報工学科卒業．同大学大学院情報科学研究科在学中．

佐々木 勇和　（正会員）

大阪大学大学院情報科学研究科助教．2014 年大阪大学情報科学研究科博士後期課程修了．情報科学博士．データベースシステム，グラフデータ処理，都市コンピューティングに関する研究に従事．ACM，IEEE，日本データベース学会の各会員．



鬼塚 真　（正会員）

1991 年東京工業大学卒業．同年 NTT 入社．2000–2001 年ワシントン大学客員研究員，2013–2014 年電気通信大学客員教授，2012–2014 年 NTT 特別研究員．博士（工学）．現在，大阪大学大学院情報科学研究科教授．これまで主記憶オブジェクトリレーショナルデータベース LiteObject，XML ストリーム処理エンジン XMLToolkit，クラウド基盤システム CBoC type2 の研究開発に従事．現在は，大規模グラフ分析，分散処理クエリ最適化，機械学習等を用いたデータ分析処理に取り組んでいる．

（担当編集委員　田中　剛）