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)]

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，整数線形計画法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，ワークロードへの応答時間を既存手法に対して平均 73.5%低減可能であることを確認した．

1. はじめに

RDBMS と比べて，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 [1, 2]，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 Cassandra [3], HBase[1] に代表される NoSQL データベース (Extensible Record Store [4] と呼ばれる) に着目する．

NoSQL データベースにおけるスキーマ設計はクエリ処理，更新処理集合の性能を最大化するスキーマを作成する重要なタスクである．スキーマ設計を行う際の課題として，クエリ処理と更新処理の性能のトレードオフが挙げられる．例えば，クエリの結果を実体化する Column Family[2]を定義すると，単独の 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．しかし，クエリ毎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．一方，複数のクエリの共通する部分クエリに応答できる Column Family を定義し，共通部分以外に応答できる Column Family と組み合わせて各クエリに応答するというスキーマ定義も考えられる．この場合，データの重複が減るため更新処理の負荷は低減できる一方で，部分クエリに応答する複数の Column Family をジョインする必要が生じる．しかし，多くの NoSQL データベースではデータベース側で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．

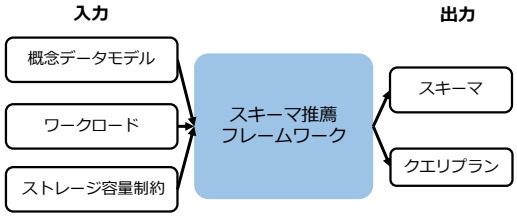

図 1: スキーマ推薦フレームワーク概念図

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，NoSQL データベー

1 大阪大学大学院情報科学研究科　〒 5650871 大阪府吹田市山田丘 1-5
2 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 8 階 802 号室
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

1 Apache HBase, http://hadoop.apache.org/hbase
2 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．

スは分散環境を想定するため，クエリプランが RDBMS とは異なり，さらには通信コストを考慮したコストモデルの設計が必要となる．したがって，RDBMS のスキーマ推薦技術 [5–7] をそのまま NoSQL データベースに適用することは困難である．

このような背景から，RDBMS を対象とするものと比べ少数ではあるが，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 [8, 9]．その中の最先端の技術として NoSQL Schema Evaluator (**NoSE**) [8] が挙げられるが，近年開発が進んできた Secondary Index [10] を活用できない問題がある．このため，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．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，最終的に整数線形計画問題としてスキーマ推薦問題を解決することにある．Secondary Index を使用することで Column Family のキー以外の属性を条件として使用するクエリを高速に実行できる．既存技術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

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なった．その結果，更新処理の負荷が高いワークロードにおいて，クエリへの応答時間の平均値を既存手法に対して 73.5%低減可能であることを確認した．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，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，評価実験において使用した Cassandra に関する概念を説明する．2.3 節では，Cassandra のデータ構造である，Column Family について説明する．また，2.4 節では Secondary Index について，2.5 節では NoSQL データベースにおけるスキーマ推薦について説明する．

2.1 Extensible Record Store

Extensible Record Store [4] のデータは属性と各レコード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，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 Cassandra

Cassandra [3] は Dynamo [11] に影響を受けた Extensible Record Store に分類される分散データベースである．マスタやスレーブを設けないことで，分散環境における単一障害点を排除しており，耐障害性が高い．Cassandra のデータ構造における最小単位はカラムであり，キーと値，タイムスタンプを持つ．そして，カラムをまとめて管理するデータ構造として Column Family を持つ．

2.3 Column Family

Column Family は複数レコードを保持するデータ構造であり，本稿では式 (1) の書式で表現する．

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

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* は各属性の値である．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．これにより，高速にクエリを処理でき，*clustering key*, *value* の属性の値を取得できる．[3]

2.4 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

[3] *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．

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)
```

外の属性が等号条件のクエリを高速に実行することが可能となる．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．

$$CF_1 = CF([pkey_1][ckey_1] \rightarrow [value_1, value_2]) \quad (2)$$

Secondary Index を活用して $value_1$ を等号条件として使用する場合，式 (3) の書式で表現される Secondary Index を使用する．

$$SI_1 = SI([value_1] \rightarrow [pkey_1], CF_1) \quad (3)$$

$value_1$ が SI_1 のキー，$pkey_1$ が value となる．この Secondary Index を使用することで，CF_1 に対して $value_1$ を等号条件として $pkey_1, ckey_1, value_2$ を取得するようなクエリも高速に処理することが可能となる．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，レコードの整合性を保つことも可能となる．

2.5 NoSQL データベースにおけるスキーマ推薦

スキーマ推薦においては，保持するデータの概念データモデル，ワークロード，ストレージ容量制約等を入力とする．そしてこれらの入力から，ワークロード内の各クエリ処理・更新処理に対して最小のコストにより応答可能であるスキーマを推薦することがスキーマ推薦における課題となる．ワークロードの最適化を行うため，推薦するスキーマに対して各クエリプランも導出し，実行コストの計算を行う．ここで，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課題となる [12]．クエリからクエリプランを導出する方法は大まかに以下の 4 種類の方法がある．

- 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**)
- 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．但し，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，概念データモデル内の 1 つのエンティティのデータが複数の 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課題が生じる．

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減らすことができる．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 [13] 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クエリ処理を行わなければならないため，大きな実行コストを要する．

Secondary Index プラン (17-22 行目) は，Column Family E 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 Secondary Index D の**後続の Column Family** と定義する．

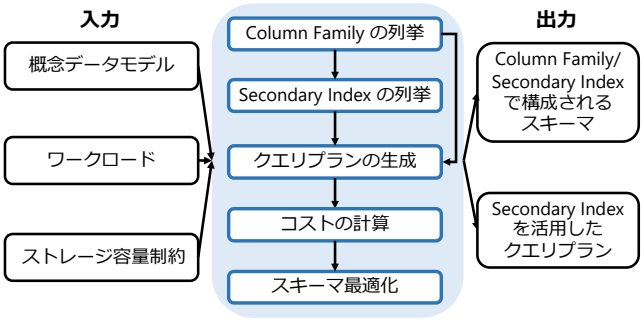

図 2: 提案手法概念図

ここで, Secondary Index の後続の Column Family は Secondary Index のキーを非 *partition key* に持ち, Secondary Index の *value* を *partiton key* に持たなければならない. 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 のキーの属性 (**Secondary Index 属性**) 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一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いため, 高速にクエリ処理を行うことができる[4].

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[5].

3. 提案手法

本稿では, Column Family に加え Secondary Index の活用も考慮した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 を活用したクエリプランを用いることでクエリの応答時間を低減することが可能となる.

3.1 フレームワークの概要

Column Family の列挙・Column Family を用いるクエリプランの生成・Column Family に関するコストモデル・スキーマ最適化の際の目的関数と一部の制約は既存手法である NoSQL Schema Evaluator (**NoSE**) [8, 14] を使用する. フレームワークの概念図を図 2 に示す. また, 処理の概要は以下の通りである.

(1) 概念データモデルおよびワークロードから, クエリ処理と更新処理を実行する際に使用する Column Family を列挙する.
(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 を用いてクエリに応答する際に使用できる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, その実行コストを見積もる.
(5) 整数線形計画法を用いることで各クエリについて列挙したクエリプランから最も合計コストが低いプランを選択する.

3.2 Column Family の列挙

入力として与えられたワークロードへの応答に使用する Column Family の候補を列挙する. この際, 2.5 節において言及した実体化プランにおいて使用する Column Family および, ジョインプランに必要な Column Family の列挙を行う. また, 列挙した Column Family の中で, *partition key*, *clustering key* の同一のものが存在すれば, それらの *value* 属性の和集合を有する新たな Column Family の生成も行う.

3.3 Secondary Index の列挙

2.5 節における 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. Secondary Index が後続の Column Family と共に Secondary Index プランとして使用される条件は, Secondary Index のキーが Column Family の非キー属性に含まれ, *value* が Column Family の *partiton key* であることである. 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, クエリプランにおいて使用する Secondary Index と後続の Column Family を複数出力する. *generate_si*() メソッドは第一引数を式 (3) の $value_1$, 第二引数を $pkey_1$ 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エンティティの 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 *generate_additional_cf* メソッドにより追加で

[4] 更新処理を行う際には 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することも可能となる.
[5] 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.

Algorithm 1: Secondary Index 列挙アルゴリズム

Input : Column Family (**CF**)
Output: Secondary Index と後続の Column Family

```
1  indexes ← []
2  foreach (extra) ∈ (CF.clustering_key + CF.value) do
3      foreach (key) ∈ (ColumnFamly.partitionkeys) do
4          if key != extra.entity.primary then
5              si1 = generate_si(key,extra.entity.primary)
6              indexes += si1
7              indexes += generate_additional_cf(si1)
8          end
9          if key != key.entity.primary then
10             si2 = generate_si(key,key.entity.primary)
11             indexes += si2
12             indexes += generate_additional_cf(si2)
13         end
14         si3 = generate_si(key,extra)
15         indexes += si3
16     end
17 end
18 return indexes
```

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: クエリプランのコスト

```
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          return
5      end
6      if step is secondary_index then
7          width_cost = width_cost × ((si_query_cost −
             cf_query_cost)/(cf_query_cost × width))
8      end
9      else if step.prev is secondary_index then
10         query_cost = query_cost × ((si_query_cost −
             cf_query_cost)/(cf_query_cost × width))
11     end
12     step.cost = base_cost +
         query_num × query_cost + width × width_cost
13 end
```

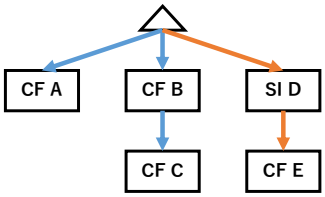

図 3: クエリ 1 に対するクエリプラン概念図．根から葉までの経路が 1 つのクエリプランに対応し，各節点が Secondary Index や後続の Column Family に対する Index Lookup であるステップを表している．

生成する (7,12 行目)．具体的には，非 *partition key* 属性に Secondary Index のキーを追加し，*partition key* に Secondary Index の *value* を持つ Column Family を与えられた Secondary Index を元に生成する．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 のケースは単独 Secondary Index プランに使用する．

3.4 クエリプランの生成

列挙した Column Family と Secondary Index を使用して，ワークロード内のクエリ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 Index Lookup，更新処理等の手順を表すステップで構成される．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．

ソースコード 1 において述べた全てのクエリプランを列挙する．ソースコード 1 のクエリとそのクエリプランを列挙する際の概念図を図 3 に示す．クエリプランの列挙は各クエリに関して，根から葉までの 1 つの経路が 1 つのクエリプランに対応する木構造のクエリプランを生成することに対応する．また，それぞれの節点がステップを表している．さらに，3 つ以上の Column Family を使用するジョインプランや，2 つ以上の Column Family を使用する Secondary Index プランにおいては 2.5 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 *partititon key* であり，*value* が後続の Column Family の *partition key* であるという特性を考慮して列挙する．そして，実行時にこのクエリプランの Secondary Index に対する Index Lookup と後続の Column Family に対する Index Lookup を，後続の Column Family の Secondary Index 属性に対しての 1 つの Index Lookup として扱う．また，単独 Secondary Index プランでは Secondary Index の後続の Column Family は別のクエリプランの中に存在する．ここで推薦される Secondary Index が必ず後続の Column Family を持つことは最適化を行う際の制約条件によって保証する．

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．まず，Column Family に対する Index Lookup のコストを取得するために，列挙した各クエリプランの Index

Lookup ステップに対して，式 (4) でコストの計算を行う．

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

式 (4) において，1 項目の $base\_cost$ は Index Lookup を行う際に常に発生するコスト，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属性の数，$width\_cost$ は属性数に対応したコストを表す．ジョインプランでは，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．

次に, 単独 Secondary Index プラン, Secondary Index プランのコストを計算する．詳細な処理を Algorithm 2 に示す．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．cf_query_cost は Column Family に対して Index Lookup を行うクエリの応答時間である．また，si_query_cost は Column Family に対して，Secondary Index 属性を等号条件として使用するクエリを実行する際の応答時間である．

まず，式 (5) により単独 Secondary Index プランのコスト計算を行う (2-5 行目).

$$cost = cf\_cost \times (si\_query\_cost / cf\_query\_cost) \quad (5)$$

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掛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 (6-11 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．これより，対象とするクエリプランが Secondary Index プランである場合に該当する処理のコストを小さく見積もる．

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踏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 (7 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 (10 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 (12 行目).

次に，更新処理のコストの計算を式 (6) によって行う．

$$cost = width \times write\_cost \quad (6)$$

定数である $write\_cost$ は各処理ごとにパラメータとして入力する．ここで, Secondary Index の後続の Column Family は，更新処理において Column Family 内のレコードと該当する Secondary Index 内のレコード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，更新処理のコスト計算では Secondary Index は key と $value$ の 2 つのみのフィールドをもつ Column Family として，Column Family と同様にコストの計算を行う．

3.6 スキーマ最適化

列挙したスキーマ候補とクエリプラン候補を用いて整数線形計画法による目的関数の定式化を行う．ここでの最適化により，ワークロード内のそれぞれのクエリについて生成した図 3 に相当するクエリプランの木構造から，1 つの経路を選択する．そして目的関数を最適化することで，ワークロードの問い合わせへの応答コストを最小化するスキーマ・クエリプランを特定する．具体的な目的関数を式 (7) に示す.

$$\text{minimize} \sum_i \sum_j f_i C_{ij} \delta_{ij} + \sum_m \sum_n f_m C'_{mn} \delta_n \quad (7)$$

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，Secondary Index 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，Secondary Index 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，Secondary Index を更新する際のコストである．また，δ_n は n 番目の Column Family，Secondary Index 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 (7) に対して式 (8),(9) の制約条件を追加する．

$$\delta_{ij} \leq \delta_j, \quad \forall i, j \tag{8}$$

$$\sum_j s_j \delta_j \leq S \tag{9}$$

制約条件 (8) は使用される Column Family は全て推薦されなければならないという制約を表す．この制約を設定しない場合では，ある推薦するクエリプランにおいて使用している Column Family, Secondary Index が推薦するスキーマの中に含まれておらず使用できない可能性が生じる．制約条件 (9) は各 Column Family の想定されるサイズ s_j の総和はストレージ容量制約 S 以下でなければならないという制約を表す．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，あるクエリプランを推薦する際は，そのクエリプランを構成する Column Family が全て推薦されなければならないという制約も加える．具体例として，クエリ 1 に対して，ソースコード 1 に上げた 3 種類のクエリプランが列挙されている場合を考える．この場合，式 (10) の制約条件を式 (7) に対して追加する．

$$\begin{aligned} \delta_{1,A} + \delta_{1,B} + \delta_D &= 1 \\ \delta_{1,A} + \delta_{1,C} + \delta_E &= 1 \\ \delta_{1,C} &\leq \delta_{1,B} \\ \delta_{1,E} &\leq \delta_{1,D} \end{aligned} \tag{10}$$

これにより Secondary Index D を使用する場合では δ_D が 1 であるから，δ_E も 1 を取ることになり，1 つのクエリプランが確定する．最後に，Secondary Index 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 Column Family と Secondary Index を組み合わせて応答するクエリプランについて式 (11) の制約条件を式 (7) に対して追加する．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．

$$\delta_D \leq \delta_E + \delta_F \tag{11}$$

式 (11) は Secondary Index D を推薦するならば，Column Family E, Column Family F のいづ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．この際，Secondary Index の後続の Column Family は最適化時に確定するため，最適化後に後続の Column Family の設定を行う．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．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ため推薦を行う．

ソースコード 2: 入力例

```
1  --user エンティティ
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: 容量制約を設定しない場合の出力例

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

3.7 入出力例

ソースコード 2 で定義される user エンティティ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．容量制約を設定しない場合はソースコード 3 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各クエリにおいて 1 度のみ Colum Family への Index lookup を行うため，高速にクエリに応答できる．一方，容量制約を設定する場合はソースコード 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 Column Family と Secondary Index を組み合わせることで，使用する Column Family を 1 つに低減し，ストレージ容量を低減している．また， Secondary Index を活用することで，複数の Column Family をクライアント側でジョインする場合に比べ，応答時間を大幅に低減している．

ソースコード 4: 容量制約を設定場合の出力例

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

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効性を評価する．

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．ベースラインでは，入力された概念データモデルの各エンティティに対応する 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，各エンティティに対応する 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．各手法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**) [15] を使用する．RUBiS は 7 つのエンティティ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．また，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属性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．ユーザエンティティのレコード数が 200,000 件で作成した RUBiS のレコードを各

表 1: AWS の実験環境

設定項目	設定値
Amazon マシンイメージ (AMI)	ami-076e276d85f524150
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 @ 2.90GHz
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

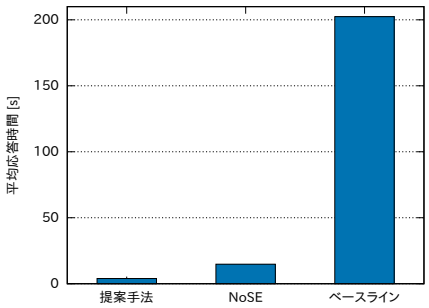

図 4: クエリの応答時間の平均値．応答時間の長いクエリにおいて Secondary Index を活用し，全体の応答時間の平均値を低減している．

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．

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，各マシン毎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 を使用した．ネットワークの通信状況による影響を低減するため，Cassandra に対してクエリを実行するクライアントも Cassandra と同一のリージョンに作成した．

4.2 実験結果

4.2.1 応答時間

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応答時間の平均値を示す．Cassandra のデータベースノードは 1 つで測定を行った．評価結果として，ベースラインと NoSE に対して，それぞれ 98.1%, 73.5% の応答時間の低減を達成した．ベースライン・NoSE はジョインプランを複数推薦し，クエリの応答時間が増加している．一方，提案手法では正規化した Column Family に加え，Secondary Index 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

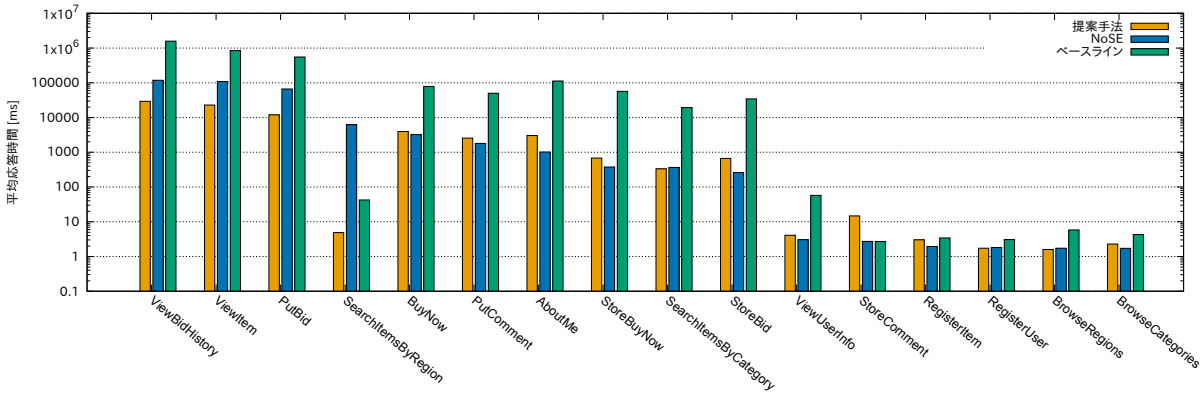

図 5: 各問い合わせ集合ごとの応答時間．NoSE やベースラインの応答時間の長いトランザクションにおいて応答時間を低減し，ベースラインや NoSE で応答時間の短いクエリには同様に短い応答時間を達成している．

トランザクションの応答時間を示す．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できていることが確認できた．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 6051.4 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でより有益であると言える．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ことも確認できた．

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 に対して平均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 つのみ持つ SearchItemsByRegion においても NoSE に対して平均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制約に余裕を持つことができたため，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平均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 句に伴い更新処理を行う 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．計測結果を図 6 に示す．Cassandra のノード数を 1, 10, 20 と変化させ，ワークロード内のクエリの応答時間の平均値を計測した．結果として，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．NoSE は通信回数の多いジョインプランを多く推薦する．そのため，ノード数の増加に伴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．また，replication factor の違いによる応答時間の増加も考えられる．ノード数が 1 の場合では replication factor は 1 であるが，ノード数が 10, 20 である場合では，replication factor は 3 であるため更新処理におけるコストが増加している可能性が考えられる．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．これは，レコード数が固定であるため，ノード数が増加すると各ノードのレコード数が減少し，レコードの探索に要する時間が減少したことが原因と考えられる．一方，提案手法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．

5. 関連研究

5.1 NoSQL データベースにおけるスキーマ設計

NoSQL データベースにおけるスキーマ推薦を行う研究は RDBMS と比べて少数ではあるが存在する [8].

本稿で提案するフレームワークの一部で使用した NoSE は本フレームワークと同じ入力から Column Family で構成されるスキーマとそのスキーマを使用するクエリプランを出力するシステムを提案している．本稿で特に課題とし

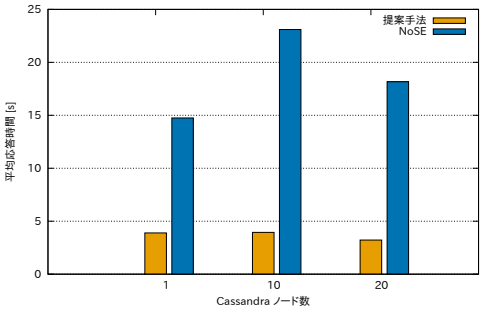

図 6: 提案手法と NoSE とのデータベースのノード数に関するスケーラビリティの比較．NoSE ではノード数が増加すると応答時間が増加しているが，提案手法では応答時間は減少している．

て注目した更新処理の多い環境では，ジョインプランを生成するため応答時間が著しく増加する．しかし，更新処理の少ない環境では効率の良いスキーマの提案を行う．

善明らの研究 [16] では，一元的に管理したスキーマ定義に基づいてアプリケーションデータや問い合わせを Key-Value Store への読み書き操作に変換する手法を提案し，HBase を対象とした実装を行っている．この手法により，HBase のデータ構造を意識せずアプリケーション開発を行うことを可能としている．

5.2 RDBMS におけるスキーマ設計

RDBMS におけるスキーマ推薦技術は数多く存在し [5–7]，これらの多くは実データを保持するテーブルに対してどのように Materialized View や Index を作成するかに着目している．

BIGSUBS [17] は非常に大きなワークロードを対象とし，ワークロード内の重複する処理を Materialize することでクエリ処理の高速化を行う．整数線形計画法により最適化する問題を二部グラフのラベリング問題を用いて細分化することで，並列に最適化を行うことの可能なスケーラビリティの高い手法を提案している．

Kodiak [7] は，大規模なデータを扱う際にスライシングやダイシングを用いた高次元データの解析を短い応答時間でサポートすることを目的とした研究である．HDFS や MySQL のクラスタ，クエリサーバで構成されるシステムを提案しており，MySQL 上の View に定期的に一括で更新処理を行う．また，Materialize する View を選択する際はクエリの平均応答時間を最小化する目的関数を設けて最適化を行う．

これらの手法は RDBMS を想定して作成された手法であり，RDBMS と NoSQL データベースの性能特性の差異により，NoSQL データベースのスキーマ推薦に適用することは困難である．

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，整数線形計画法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．

参考文献

[1] Y. Li, S. Manoharan, “A performance comparison of SQL and NoSQL databases,” PACRIM, pp.15–19, 2013.
[2] R. Hecht, S. Jablonski, “NoSQL evaluation: A use case oriented survey,” CSC, pp.336–341, 2011.
[3] A. Lakshman, P. Malik, “Cassandra: A decentralized structured storage system,” SIGOPS, vol.44, no.2, pp.35–40, 2010.
[4] R. Cattell, “Scalable SQL and NoSQL data stores,” SIGMOD Record, vol.39, pp.12–27, 2011.
[5] I. Mami, Z. Bellahsene, “A survey of view selection methods,” SIGMOD Record, vol.41, pp.20–29, 2012.
[6] S. Chaudhuri *et al.*, “Compressing SQL workloads,” SIGMOD, pp.488–499, 2002.
[7] S. Liu *et al.*, “Kodiak: leveraging materialized views for very low-latency analytics over high-dimensional web-scale data,” PVLDB, vol.9, no.13, pp.1269–1280, 2016.
[8] M. J. Mior *et al.*, “NoSE: Schema design for NoSQL applications,” TKDE, vol.29, no.10, pp.2275–2289, 2017.
[9] M. J. Mior, “Automated schema design for NoSQL databases,” SIGMOD PhD Symposium, pp.41–45, 2014.
[10] M. A. Qader *et al.*, “A Comparative Study of Secondary Indexing Techniques in LSM-based NoSQL Databases,” SIGMOD, pp.551–566, 2018.
[11] G. DeCandia *et al.*, “Dynamo: Amazon’s highly available key-value store,” SIGOPS, pp.205–220, ACM, 2007.
[12] G. L. Sanders, S. Shin, “Denormalization effects on performance of RDBMS,” HICSS, Jan 2001.
[13] P. Mishra, M. H. Eich, “Join processing in relational databases,” CSUR, pp.63–113, 1992.
[14] M. J. Mior *et al.*, “NoSE: Schema design for NoSQL applications,” ICDE, pp.181–192, 2016.
[15] E. Cecchet *et al.*, “Performance and scalability of EJB applications,” ACM SIGPLAN Notices, pp.246–261, 2002.
[16] 善明 晃由, 津田均, “スキーマ定義に基づく SQL ライクな Key-Value ストアクライアント,” DEIM, pp.2–7, 2016.
[17] A. Jindal Konstantinos Karanasos Sriram Rao Hiren Patel Microsoft *et al.*, “Selecting Subexpressions to Materialize at Datacenter Scale,” PVLDB, vol.11, no.7, pp.800–812, 2018.