Secondary index を活用する NoSQL スキーマ推薦による Query 処理高速化

涌田　悠佑† 善明　晃由†† 松本　拓海† 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
†† 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
E-mail: †{wakuta.yusuke,matsumoto.takumi,sasaki,onizuka}@ist.osaka-u.ac.jp,
††zenmyo_teruyoshi@cyberagent.co.jp

あらまし　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いる．そして，そのスキーマ設計は性能を引き出すために非常に重要な課題である．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化が求められている．既存技術では対応できるスキーマが限られるため，更新処理のために正規化が必要な状況下において Query の応答時間が著しく増加する問題が生じる．本稿では Secondary Index の活用を考慮した NoSQL データベースのスキーマ推薦により Query 処理の高速化を図る．具体的には Query 処理に活用できる Column Family・Secondary Index を列挙し，それらを使用する Query Plan の最適解を整数線形計画法によって探索する．これにより，更新処理を効率的に行うためにレコードの正規化が必要な状況下においても，頻度の高い Query に対しては 1 つの Column Family により応答し頻度の低い Query に対しては Secondary Index を活用することで Query 処理を高速化する．評価実験により，更新処理が多い場合に相当する厳しい容量制限下において，ワークロードの Query への応答時間の期待値を既存手法に対して 77.9%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

1 はじめに

NoSQL データベースはシステム要件に適したスキーマの使用やトランザクションの緩和により，RDBMS と比べて高い性能を達成 [1] し，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 Extensible Record Store [2] に分類される NoSQL データベースに着目する．Extensible Record Store の例としては Cassandra [3], HBase [4] が存在する．これらの NoSQL データベースでは Key-Value 型のデータを扱うため，対象システムが扱うレコードの Key-Value としてのスキーマを設計する必要がある．本稿の評価実験において使用した Cassandra においては，スキーマは Column Family を用いて設計する．

スキーマ設計はシステム要件の中で Query，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．Extensible Record Store におけるスキーマ設計の良しあしは性能に大きく影響する [5]．スキーマ設計を行う際の課題として，Query 処理と更新処理の性能のトレードオフが挙げられる．ある Query に単独で応答できる Column Family を定義すると Query の応答時間を低減できる．しかし，全ての Query にこのような Column Family を定義するとデータが非正規化されるため更新処理におけるユーザの負荷が増加する．一方，複数の Query の共通部分に応答できる 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 Query に応答するとデータが正規化できる．そのため，更新処理の負荷は低減できるが，Query の応答時間が増加する．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おり，データベースの性能を十分に発揮できていない場合が生じる．そのため，スキーマ設計を自動化する技術としてスキーマ推薦技術が提案されている．しかし，Extensible Record Store のスキーマ推薦を行う技術は RDBMS と比べて非常に少ない [6]．また，Extensible Record Store では SQL のような高機能な Query 言語をサポートしているものは少ない．さらに，キー以外の属性を等号条件とした Query 処理の応答時間が著しく増加する等，性能特性も RDBMS とは大きく異なる．したがって，RDBMS のスキーマ推薦技術 [7–10] 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．これらの問題点より，Extensible Record Store においてスキーマ設計を自動化するスキーマ推薦フレームワークが必要とされている．本稿では図 1 に示した入出力を持つスキーマ推薦フレームワークを想定する．入力で与えられるワークロードは，UPDATE, INSERT,

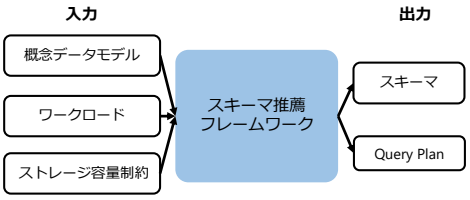

図 1　スキーマ推薦フレームワーク概念図

DELETE 句等の更新処理や，Query の集合で構成されている．

Extensible Record Store においてスキーマ推薦を行う既存技術 [11] では，更新処理が Query に比べ多い場合は，単独で応答可能な Column Family を複数の Column Family に正規化し，他の Query と共有しつつ組み合わせることで Query に応答する．ただし，各 Column Family への Query 処理は独立して行うため各 Query 処理においてクライアントとデータベース間で Query 処理結果を転送しなければならないため，Query 処理のコストが増大する．また，NoSQL データベースはデータベースでのジョイン処理を提供していないため，正規化した複数の Column Family を使用する Query Plan においてはクライアント側で入れ子ループ結合を行う．この際，1 つ目の Column Family に対して Query 処理を行った結果の各レコードを等号条件として 2 つ目の Column Family に対する Query 処理を行うため，応答時間が著しく増加するという問題がある．

本稿では，Secondary Index [12] 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 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 Query を高速に実行できる．複数の Column Family を使用する Query Plan を Secondary Index を活用する Query Plan に置き換えることで，クライアントとデータベース間でのデータ転送回数を削減し，大幅に Query の応答時間を低減することができる [1]．Secondary Index を適切に活用するために Column Family だけでなく Secondary Index も含めたスキーマ・Query Plan 候補を生成し，コストモデルにより実行コストを評価する．そして，それぞれのスキーマ・Querya Plan に整数線形計画法を適用し最適解の探索を行う手法を提案する．提案フレームワークによって推薦されたスキーマを Cassandra 上に定義し，Query Plan を実行する際の応答時間・スループット・スケーラビリティの評価を行なった．評価実験により，更新処理が多い場合に相当する厳しい容量制限下において，ワークロードの Query への応答時間の期待値を既存技術に対して 77.9%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しても評価を行い，既存技術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，評価実験において使用した Cassandra に関する概念を説明する．2.1 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，2.4 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 は Cattel ら [2] により行われた NoSQL データベースの分類の一種である．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．また，Extensible Record Store のデータ構造は属性と各レコードにより構成されている．データベースノード群にデータを分散する際は，属性とレコードを用いて効率的な分散を行う．属性に関しては，Column Family 等の属性の集合として分割され，各データベースノードに保持される．さらに，それぞれの属性の集合に保持される各レコードはキーの値によって各データベースノードにハッシュ分散あるいは範囲分散によって割り当てられ，保持される．

2.2 Column Family

Column Family は各レコードをキーによって一意に特定可能であり，各レコードは任意の個数の属性を持つことができ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．

$$CF([partition\ keys][clustering\ keys] \rightarrow [values]) \quad (1)$$

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* によってデータベース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．Query 処理を行う際に *partition key* を等号条件として利用することで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ため，高速な Query 処理を行うことができ *clustering key*, *value* の属性の値を取得することができる．*partition key* を等号条件として持たない Query を実行する際，全ての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，Query の応答時間が非常に大きくなる．

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る．Secondary Index を使用することで Column Family の *partition key* 以外の属性が等号条件の Query を高速に実行することが可能となる．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 を使用せずに $pkey_1$ 以外の属性を等号条件として使用する Query を実行すると，応答時間が増加する．ここで，Secondary Index を活用して $value_1$ を等号条件として使用する場合について考える．本稿では式 (3) の書式で Secondary Index を表現する．

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

1 : Secondary Index はある特定の Column Family の属性に対して定義する．また，Secondary Index は定義元の Column Family とアトミックに更新が行われるので，レコードの整合性を保つことも可能となる．

ソースコード 1　Query Plan 例

```
1  --Query1
2  SELECT id, firstname, lastname, password, email
3  FROM user
4  WHERE firstname = ?
5
6  --実体化 Plan
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 Plan
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 Plan
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 --Query2
25 SELECT lastname FROM user WHERE firstname = ?
26
27 --単独 Secondary Index Plan
28 Index Lookup Secondary Index E:
29     SI([user.firstname] → [user.lastname],CF_X)
```

$value_1$ 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，$value_1$ を等号条件として使用する Query 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．

2.4　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 Query 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 Column Family, Secondary Index で構成され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．この際，推薦するスキーマのサイズに制約を課すことが可能である．さらに，推薦するスキーマに対して各 Query の実行計画 (**Query Plan**) も導出を行う．この際，Query 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 [13]．Query から Query Plan を導出する方法は以下の 4 種類の方法がある．

- 単独の Column Family のみを用いて応答する Query Plan (**実体化 Plan**)
- 正規化された複数の Column Family を用いて応答する Query Plan (**ジョイン Plan**)
- Column Family と Secondary Index を組み合わせて応答する Query Plan (**Secondar Index Plan**)
- 単独の SecondaryIndex を使用する Query Plan (**単独 Secondary Indx Plan**)

ソースコード 1 にそれぞれの Query Plan の例を示す．Index Lookup ステップはキーの値を等号条件とした Query 操作を表す．

実体化 Plan (6-8 行目) は 1 つの Column Family にアクセスすることで Query 結果を取得できるため応答時間が短い．ただし，各 Query に対して実体化 Plan を生成すると，データが非正規化されるため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各レコードの整合性を保つ負荷が増加する課題が生じる．

ジョイン Plan (10-14 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，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 Query Plan では Column Family B, Column Family C に関して，入れ子ループ結合 [14] に相当する処理を行う．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 Query 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 Plan (16-20 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．ここで，Secondary Index の後続の Column Family として Column Family を使用できる条件は 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．この Query Plan は，Secondary Index を用いることで，ジョイン Plan と同様にデータを正規化することができる．Secondary Index Plan を実行する際は，データベースノード内において Column Family に定義した Secondary Index のキーの属性 (**Secondary Index 属性**) を用いて 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 Query 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 Query 処理を行うことができる [2]．

単独 Secondary Index Plan は Query 2 に対して単独で応答可能な Secondary Index を用いる．この Plan を用いることで Column Family 上のレコードの重複を低減することができる [3]．

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 Query Plan を用いることで

2 : 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．

3 : 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

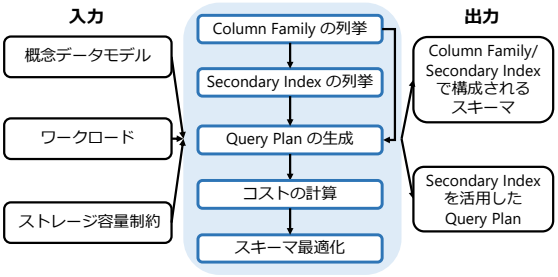

図 2　提案手法概念図

Query 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・Query Plan の生成・スキーマ最適化の際の目的関数と一部の制約は既存技術である NoSQL Schema Evaluator (**NoSE**) [11,15] 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)　概念データモデルやワークロードから，Query 処理や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, Secondary Index と Column Family の組を列挙する．

(3)　列挙した Column Family, Secondary Index を用いて Query に応答する際に使用できる Query Plan を列挙する．

(4)　Query Plan 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 Query について列挙した Query Plan 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に応答する際に使用する Column Family の候補を列挙する．この際，2.4 節において言及した単独の Column Family のみを用いて応答する 実体化 Plan において使用する Column Family だけではなく，ジョイン Plan に必要な Column Family の列挙も行う．

3.3　Secondary Index の列挙

2.4 節における Secondary Index Plan および単独 Secondary Plan で使用する Secondary Index を列挙する．Secondary Index が後続の Column Family と共にジョイン Plan 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 *partitonkey* 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，Query Plan において使用する Secondary Index と 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 (14-16 行目)

Algorithm 1: Secondary Index 列挙アルゴリズム

Input : ColumnFamily (**CF**)
Output: Secondary Index と補助用 Column Family

```
1  indexes ← []
2  foreach (nkey) ∈ Union(CF.clustering_keys, CF.value) do
3      foreach (key) ∈ ColumnFamly.partitionkeys do
4          if key != nkey.entity.primary then
5              si1 = generate_si(key,nkey.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,nkey)
15         indexes += si3
16         indexes += generate_additional_cf(si3)
17     end
18 end
19 return indexes
```

1 つ目の Secondary Index のケースは，Secondary Index Plan において使用される．ここで，Query Plan の数を削減するため，Secondary Index Plan 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 を追加で列挙する．具体的には，後続の Column Family として使用される可能性のある Column Family に関して，非キー属性に Secondary Index のキーを追加した Column Family を *generate_additional_cf* メソッドにより追加で生成する．これにより，この制約下においても Secondary Index を活用できる Query Plan 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 Plan に使用する．

3.4　Query Plan の生成

列挙した Column Family・Secondary Index を使用して，ワークロード内の Query に応答する際の Query Plan を生成する．Query Plan 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 を行うことが可能である．ソースコード 1 において述べた全ての Query Plan を列挙する．さらに，3 つ以上の Column Family が連続する Query Plan や Secondary Index の後続の Column Family の更に次に複数の Column Family が存在する場合についても再

Algorithm 2: Query Plan 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: Query Plan 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)
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)
11     end
12     step.cost = base_cost +
         query_num × query_cost + width × width_cost
13 end
```

帰的に 2.4 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 Plan では，ある Secondary Index に対して Index Lookup を行うステップの次はその Secondary Index の定義元の Column Family に対する Index Lookup を行うように Query Plan を作成する．そして，実行時にこの Query Plan の Secondary Index に対する Index Lookup と定義元の Column Family に対する Index Lookup を定義元の Column Family の Secondary Index のキーとして用いた属性に対しての 1 つの Index Lookup として扱う．また，Query に対して単独の Secondary Index で応答する Query Plan では Secondary Index の定義元の Column Family は別の Query Plan の中に存在する．ここで Secondary Index が必ず定義元の 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作成することで，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 Query Plan 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 を行う場合に常に発生するコスト，2 項目は Query 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 Query 回数，$query\_cost$ は各 Query 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．Column Family を組み合わせて使用する Query Plan 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 $query\_num$ に代入することでコストを評価する．

次に，Secondary Index を使用する Query Plan のコストの計算を行う．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 Query 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 Query 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 Plan のコスト計算を行う (2-5 行目)．

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

Column Family を直接使用する場合に比べ，Secondary Index を使用するコストが増加する．そのため，Column Family を直接使用する際のコストに対して，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 Plan のコスト計算を行う (6-11 行目)．Column Family を組み合わせて使用する Query Plan に比べ，データの転送回数を削減できることを踏まえたコスト計算を行う．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 である場合は結果をクライアントに再送しないため，$width\_cost$ を Secondary Index を使用する分低く見積もる (7 行目)．

該当するステップの 1 つ前のステップが Secondary Index である場合はクライアントから Query 処理を取得しないため，$query\_cost$ を Secondary Index を使用する分低く見積もる (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，Secodnary Index を定義した定義元の 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定義元の 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 $value$ の二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 Query Plan 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．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・Query Plan を特定する．具体的な目的関数を式 (7) に示す．

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

1 項目は Query 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 Query 内の i 番目の Query の頻度，C_{ij} は i 番目の Query に対して j 番目の Column Family, Secondary Index を使用する際のコスト，δ_{ij} は i 番目の Query 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．また第 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，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保持する 0 もしくは 1 の値を取る変数である．また，式 (7) に対して式 (8),(9) の制約条件を追加する．

$$\delta_{ij} \leqq \delta_j, \quad \forall i, j \qquad (8)$$

$$\sum_j s_j \delta_j \leqq S \qquad (9)$$

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 Query Plan 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することができる．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することができる．また，ある Query Plan を推薦する際は，その Query Plan 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，Query 1 に対して，ソースコード 1 に上げた 3 種類の Query Plan 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．

$$\begin{aligned} \delta_{1,A} + \delta_{1,B} + \delta_D &= 1 \\ \delta_{1,A} + \delta_{1,C} + \delta_E &= 1 \\ \delta_{1,C} &\leqq \delta_{1,B} \\ \delta_{1,E} &\leqq \delta_{1,D} \end{aligned} \qquad (10)$$

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 Query Plan が確定する．最後に，Secondary Index を推薦する際は，その定義元となる 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 Query Plan 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 の定義元の Column Family として使用できる場合を考える．

$$\delta_D \leqq \delta_E + \delta_F \qquad (11)$$

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 は必ず定義対象として Column Family を持つ．この際，Secondary Index の定義元となる Column Family は最適化時に確定するため，最適化後に定義元として使用する Column Family の設定を行う．また，更新処理において Secondary Index は，定義元の 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 Query Plan の推薦は行わない．スキーマ最適化の結果，各 Query に対して使用するスキーマと Query Plan が確定するため推薦を行う．

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した．

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 に加え，Query の等号条件から Column Family の *partition key* を取得できる Column Family を作成する．そして，これらの Column Family を複数組み合わせて Query に応答する．評価項目として各手法によって推薦されたスキーマを Cassandra 上に構築し，Query Plan を実行する際の応答時間，そしてそ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ベンチマークである Rice University Bidding System (**RUBiS**) [16] 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 Query，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 Query は Secondary Index の有用性を確認するために最適ではないた

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
動作周波数 (GHz)	2.90
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　Cassandra 設定内容

設定項目	設定値
バージョン	3.11
ノード数	1, 10, 20
Replication Factor	$\begin{cases} 1 & (\text{ノード数} < 3) \\ 3 & (otherwise) \end{cases}$
Replication Strategy	SimpleStrategy

め，WHERE 句に 1 つのみ等号条件を持つ Query 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 Query を追加する．追加で生成した Query の実行頻度は複製元の Query 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 のレコードをそれぞれのスキーマに変換し，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，使用した Cassandra の設定内容を表 2 に示す．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 Query 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．また，提案するフレームワークでは，Cassandra のキャッシュの影響を考慮していないため，Column Family の partition キーキャッシュ・行キャッシュは無効化して実験を行った．

4.2 実験結果

4.2.1 応答時間

図 3 にそれぞれのスキーマに対して Query 処理を行った際の応答時間の期待値を示す．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，ベースラインに対して，97.7%，NoSE に対して，77.9% の応答時間の低減を達成した．ベースライン・NoSE は更新処理の多い環境に対応する容量制約下において正規化した Column Family を複数用い Query の応答が完了するまで長時間要している．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 4 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．NoSE やベースラインが特に応答時間が増加している Query に対して応答時間を低減できていることが確認できる．ベースライン・NoSE・提案手法の最も応答に時間の掛かる Query の応答時間はそれぞれ，6051.4 秒, 470.9 秒, 109.6 秒であった．また，ベースラインや NoSE において応答時間の短い Query 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．

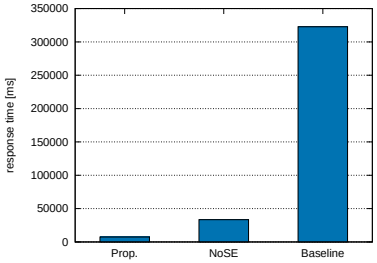

図 3　Query の応答時間の期待値．特に応答時間の長い Query において Secondary Index を活用することで全体の応答時間の期待値を低減している．

4.3 スケーラビリティ

図 3 より，提案手法・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ため，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．スケーラビリティの計測結果を図 5 に示す．Cassandra のノードを 1, 10, 20 ノードと増加させ，ワークロード内の Query の応答時間の期待値を計測した．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，提案手法ではほとんど見られないことを確認できた．これは，NoSE は Column Family を組み合わせて Query に応答する Query Plan を使用する際に通信回数が大幅に増加するため，ノードが増加した際に応答時間が増加する．また，NoSE の応答時間が 10 ノードから 20 ノードに増加させた場合に応答時間が増加しているのは，レコード数が固定であるため，各データベース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ものによると考えられる．一方，提案手法では通信回数が低減されるため，ノードの増加による通信量の増加の影響を受けにく，応答時間がノード数の影響を受けにくいものと考えられる．

5 関連研究

5.1 NoSQL データベースにおけるスキーマ推薦

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 [11]．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 Query Plan 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て注目した更新処理の多い環境では，Column Family を組み合わせた Query Plan を生成するため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効率の良いスキーマの提案を行う．

5.2 RDBMS におけるスキーマ推薦技術

RDBMS におけるスキーマ推薦技術は数多く存在し，これ

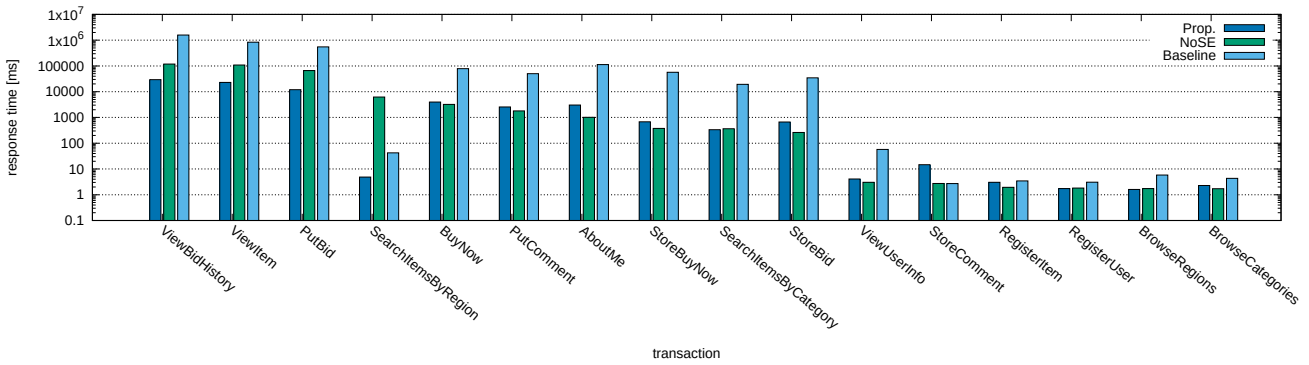

図 4　各問い合わせ集合ごとの応答時間．NoSE やベースラインの応答時間の長いトランザクションにおいて応答時間を低減しているだけでなく，ベースラインや NoSE で応答時間の短い Query には同様に短い応答時間を達成している．

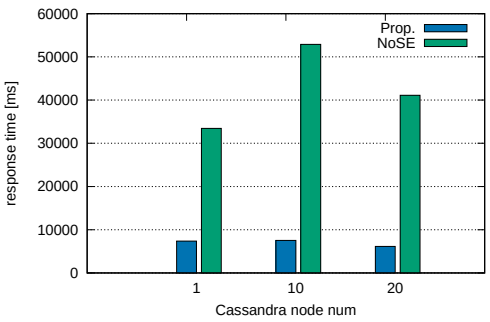

図 5　提案手法と NoSE とのスケーラビリティの比較．提案手法はノード数の多い環境でも短い応答時間を維持している．

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 [10] 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 Query 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．

6 ま と め

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 Query Plan を生成し，整数線形計画法によって最適化したスキーマを推薦する．既存手法では，更新処理が多く存在する際には，Query の応答時間が著しく増加する場合が存在するが，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 Query Plan 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 Query の応答時間を低減することを確認できた．

7 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．

文　　献

[1] Y. Li, and S. Manoharan, “A performance comparison of SQL and NoSQL databases,” IEEE Pacific RIM Conference on Communications, Computers, and Signal Processing, pp.15–19, 2013.
[2] R. Cattell, “Scalable SQL and NoSQL data stores,” ACM SIGMOD Record, vol.39, no.4, p.12, 2011.
[3] A. Lakshman, and P. Malik, “Cassandra: A decentralized structured storage system,” SIGOPS, vol.44, no.2, pp.35–40, 2010.
[4] “Apache HBase,” http://hadoop.apache.org/hbase/.
[5] S. Scherzinger, E.C. De Almeida, F. Ickert, and M.D. Del Fabro, “On the necessity of model checking NoSQL database schemas when building SaaS applications,” TTC, pp.1–6, 2013.
[6] M.J. Mior, “Automated schema design for NoSQL databases,” SIGMOD PhD Symposium, pp.41–45, 2014.
[7] I. Mami, and Z. Bellahsene, “A survey of view selection methods,” ACM SIGMOD Record, vol.41, no.1, p.20, 2012.
[8] S. Liu, B. Song, S. Gangam, L. Lo, and K. Elmeleegy, “Kodiak: leveraging materialized views for very low-latency analytics over high-dimensional web-scale data,” VLDB Endow., vol.9, no.13, pp.1269–1280, 2016.
[9] S. Agrawal, S. Chaudhuri, and V.R. Narasayya, “Automated Selection of Materialized Views and Indexes in SQL Databases,” VLDB, no.5, pp.496–505, 2000.
[10] A. Jindal Konstantinos Karanasos Sriram Rao Hiren Patel Microsoft, A. Jindal, K. Karanasos, S. Rao, H. Patel, and H.P. Microsoft, “Selecting Subexpressions to Materialize at Datacenter Scale,” Pvldb, vol.11, no.7, pp.800–812, 2018.
[11] “NoSE: Schema design for NoSQL applications,” TKDE, vol.29, no.10, pp.2275–2289, 2017.
[12] M.A. Qader, S. Cheng, and V. Hristidis, “A Comparative Study of Secondary Indexing Techniques in LSM-based NoSQL Databases,” SIGMOD, pp.551–566, 2018.
[13] G.L. Sanders, and S. Shin, “Denormalization effects on performance of rdbms,” HICSS, Jan 2001.
[14] P. Mishra, and M.H. Eich, “Join processing in relational databases,” ACM Computing Surveys, 1992.
[15] M.J. Mior, K. Salem, A. Aboulnaga, and R. Liu, “Nose: Schema design for nosql applications,” ICDE, pp.181–192, 2016.
[16] E. Cecchet, J. Marguerite, and W. Zwaenepoel, “Performance and scalability of EJB applications,” ACM SIGPLAN Notices, 2002.