局所探索法を用いた実体化ビュー選択によるクエリ処理高速化

乗松　奨真† 　涌田　悠佑† 　佐々木勇和† 　鬼塚　真†

† 大阪大学情報科学研究科　〒 565–0871 大阪府吹田市山田丘 1-5

E-mail: †{norimatsu.shoma,yusuke.wakuta,sasaki,onizuka}@ist.osaka-u.ac.jp

あらまし　データベースにおいて大規模なデータを分析するため高速なクエリ処理が不可欠である．データベース上では重複したクエリ処理が実行されることが多く，クエリ処理の結果を実体化して再利用することは，データベース全体でのワークロードの高速化に繋がる．ここで実体化するクエリの選択が重要となるが，クエリ数が膨大な場合，実体化するクエリの選択を整数計画問題として解くことは解の探索空間が膨大であるため実行が困難である．本稿では，整数計画問題によって局所的に最適解を探索し，得られた最適解の近傍に探索範囲を拡大し，解を改善する処理を繰り返すことで実体化するクエリを決定する手法を提案する．Join Order Benchmark に対して，通常の整数計画問題による最適化と比較して，クエリ処理時間の増加を 1.2 倍程度にとどめつつ，通常の整数計画問題で最適化できない問題に対して最適化を可能にし，通常の整数計画問題で最適化可能な問題に対しては，最適化時間を最悪でも 20% 程度にまで低減した．

キーワード　実体化ビュー，整数計画問題，データベース

1 はじめに

近年データベースシステムに対して多くのユーザが存在し，大量のクエリが実行されている [1] [2]．このことにより，一つの企業内で数十万単位で日々のクエリが実行されている場合がある [3]．そして，これらのクエリの処理速度を向上させることが大きな課題となっている．

データベース内で行われるクエリ処理には多くの共通した処理が含まれている．この共通したクエリ処理の結果を実体化ビューとして保持しておくことで，実体化ビューを再利用することで，高速にクエリ処理を実現できるようになる．

クエリ処理の性能は利用する実体化ビューに依存するため，実体化ビューとして保存する処理を適切に決めることが重要になる．この実体化ビューによってワークロードに含まれるクエリ処理がどの程度高速になるかをその実体化ビューの利得とする．最適な実体化ビューを選択する際に，考慮すべき観点には．実体化ビューを利用することによって得られる利得と，実体化ビューの更新処理の負荷が挙げられる．検索処理性能と更新処理性能はトレードオフの関係にある．処理に時間がかかるクエリ処理を実体化する場合，これを実体化して得られる利得は高くなるため，実体化ビューを多く作れば作るほど，高い利得が得られる．しかし，テーブルに対して更新が行われると，そのテーブルに関するクエリ処理の結果が変わってしまうため，クエリによりテーブルが更新された場合は関連する実体化ビューにも更新する必要があり，関連する実体化ビューが多い場合はその処理に時間がかかる．一方で保持する実体化ビューが少ない場合，得られる利得は小さいが実体化ビューの更新処理は短時間である．全ての実体化ビュー候補に対し、整数計画問題を解くことで最適な実体化ビュー群を解として得ることができるが，クエリ数が多いと，その導出に時間がかかる．このような背景から，最適な実体化ビューを高速に選択する手法が提案されている [1] [3] [4] [5] [6]．例えば BIGSUBs [3] では，巨大なワークロードを対象としているが，全てのサブクエリを実体化候補として列挙するために，最適化に時間がかかるという問題がある．

本稿では，近似解の精度を高く保ちつつ，最適化を高速化する手法を提案する．膨大な解候補を削減して最適化を行うために局所探索 [7] を用いる．局所探索を用いると最適解を見つけることが困難となるため，より最適解に近似できるようアルゴリズムを設計する．まず，実体化した際の利得が上位となるものを初期解として選択する．選ばれる初期解が少ないほど，近傍探索によって得られる整数計画問題の解候補は少なくなり，最適化は高速となる．しかし，初期解を少なくしすぎると，最終的に選ばれる実体化ビューを用いたクエリ応答時間は最適解を用いた場合に比べ長くなってしまう．このため，初期解の選び方は重要となる．次に，解の近傍を探索して解候補を列挙し，得られた解候補に対して整数計画問題を解く．この処理を利得が収束するまで反復する．実験は Join Order Bendhmark のクエリセットを用い，ストレージの容量制約を全てのクエリを実体化した際のストレージに対する割合で設定する．解候補を削減しないで行う最適化と比較して，最適解に比べ最悪でも 1.2 倍程度のクエリ処理時間を提供しつつ，0.01 ～ 40% の最適化時間を達成した．

本稿の構成は，次の通りである．2 節で事前知識，整数計画問題での定式化を説明する．3 節で提案手法の詳細について説明する．4 節で実験について述べる．5 節で関連研究について，6 節で本稿のまとめ，今後の課題を述べる．

2 問題定義

提案手法では，実体化するサブクエリを決定するために，ま

ず，ワークロードに含まれるクエリの実行プランから全てのサブクエリを抽出する．その後，ストレージ制約のもと，サブクエリの重複を考慮し実体化した際に得られる利得が最も高くなるよう，サブクエリ群を選択する．解として選ばれたサブクエリの近傍のサブクエリは探索空間を探索するために用いられる．これらの詳細について 2.1 節において前提知識として説明する．次に，2.2 節で本稿で最適化問題を解くために用いられる整数計画問題での定式化を説明する．本稿では，サブクエリ集合を S，ワークロードに含まれるクエリ集合を Q とする．

2.1 事 前 知 識

サブクエリの利得:サブクエリの利得とは，実体化ビューを用いた場合と用いない場合でクエリを処理した時のコストの差である．利得の評価にはサブクエリの推定実行コストを用いる．サブクエリのコストは，クエリ実行時間から決定されるが，実際にクエリを実行してコストを抽出することは膨大に時間がかかる．そのため，コストは R.Marcus [8] らが提案するような cost estimation の技術を用いて推定する場合と実測する場合がある．

BIGSUBs [3] の定義に従って利得を以下のように定義する．クエリ q_i 内に対して，サブクエリ s_j を実体化 (Ms_j) して得られる利得 u_{ij} とは，Ms_j を利用する場合としない場合とで生じる q_i を処理するコスト差として表現する (図 3)．

$$u_{ij} = cost(i) - cost(i|s_j) \tag{1}$$

実行プラン: ワークロードのクエリはその内部の処理順によって，複数の実行プランが考えられる．具体例として 3 つのテーブル T_1，T_2，T_3 を結合するクエリ q を考える．この時，パターン 1)T_1 と T_2 を結合した後，T_3 を結合する，パターン 2)T_1 と T_3 を結合した後，T_2 を結合する，パターン 3)T_2 と T_3 を結合した後，T_1 を結合する，というように 3 通りのパターンは考えられる．

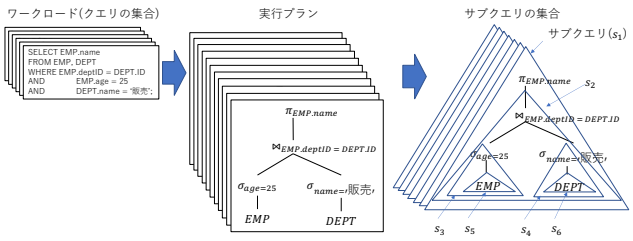

図 1　クエリワークロードの木構造実行プランへの変換，実行プラン中のサブクエリ

実体化ビュー:サブクエリを事前計算し，得られる結果を実体化ビューと呼ぶ．

サブクエリの重複: ワークロード内の異なるクエリ間で重複したサブクエリが存在する場合がある．図 2 にその例を示す．サブクエリ s_1 がクエリ q_1, q_2, q_3, q_4 に包含され，s_2 がクエリ 2, 4 に共通し，s_3 がクエリ 2, 3 に共通している．このように複数クエリのパースツリーの間で同一の部分木 (サブクエリ) が

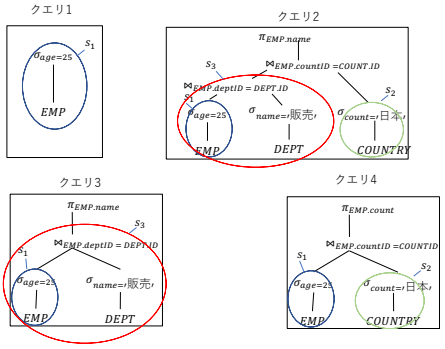

図 2　異なるクエリ間でのサブクエリの重複

存在する場合，これをサブクエリの重複があると呼ぶ．従ってこのような重複したサブクエリを実体化ビューとして保存し，再利用することで，高速なクエリ処理が可能となる．

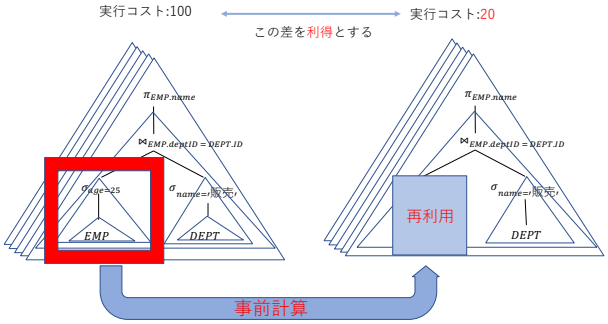

図 3　サブクエリを実体化することによって得られる利得

2.2 整数計画問題を用いたスキーマ最適化

クエリの実行プラン木に含まれるサブクエリの処理に対して，事前計算した実体化ビューを利用することで，クエリ処理の時間を削減できる．最適な実体化ビューの選択において，クエリの実行プランは全て考慮する必要がある．その理由は以下である．クエリ q_A，q_B について考える．実行プランによってクエリのコストは異なり，q_A のみ実行する場合，この合計コストが一番低い実行プランが最適である．しかし，別の q_B が存在し，q_B に対する実行プラン p_B と，A に対する実行プラン p_A に重複したサブクエリ s_i が含まれるとする．サブクエリ s_i が実体化されている場合，A， B の最適プランのコストの和よりも，s_i を再利用することによって p_A 及び p_B の実行コストの和が小さい場合がある．このことからクエリの実行プランを全て考慮する必要がある．

次に，多くの実体化ビューを作成する場合には二つの問題がある．一つ目の問題はストレージ容量には制限がある場合では，無制限にクエリの実行プラン木に含まれるサブクエリを実体化ビューとして保持することはできないことである．二つ目の問題は多くのクエリ処理の結果が実体化ビュー内に保存されていると，更新処理の際，実体化ビュー内の全ての結果を更新する必要があるため，更新処理の性能が低下することである．この二つの問題を解決するために，クエリワークロードに含まれる各クエリに対して，全てのクエリを実体化できないストレージ

制約を課すことで実体化ビューのストレージを制限し，更新処理性能の低下を抑える．そして実体化ビューを利用することで得られる利得を計算し，ワークロード全体の利得を最大化，つまりクエリ応答時間が最も短くなるよう目的関数を定める．

目的関数の定義．クエリワークロード W に含まれるクエリ集合を Q とし，各クエリ $q_i \in Q$ について，q_i の実行プランを p_{it}(t は q_i の実行プランの個数だけ存在する)，p_{it} に含まれるサブクエリを表す集合 $subquery(p_{it})$，実行プラン p_{it} が q_i に対して用いられるときに 1 をとるバイナリ変数を y_t，s_j が実体化されるときに 1 をとるバイナリ変数を x_j，$cost(p_{it})$ を p_{it} の実行コストとして目的関数を最小化する．

$$minimize \sum_i \sum_t (y_t \cdot (cost(p_{it}) - \sum_{j \in subquery(p_{it})} x_j \cdot cost(j)) \tag{2}$$

第一項 $y_t \cdot (cost(p_{it})$ は利用する実行プランの総コスト，第二項 $x_j \cdot cost(j)$ は実体化するサブクエリのコストを表す．データベースのストレージサイズを考慮しつつ，実体化ビューの評価を行い，整数計画問題を解く．そのため，式 2 を以下の三つの制約下で解く必要がある．一つ目は実体化ビューの容量を制限するストレージ制約．二つ目はサブクエリのコストを重複して評価しないための制約．三つ目は一つのクエリに対して用いられる実行プランは一つのみとする制約である．

制約 1　ストレージ制約

ストレージ制約は実体化ビューを保持するストレージの容量を制限している．[1]b_j はサブクエリ s_j のストレージ容量，B_{max} は使用できるデータベース全体のストレージ容量とする．x_i は s_i が実体化される時に 1 をとり，実体化されない時は 0 であるため，$b_i \cdot x_i$ の全ての i に対する総和を取ることで，実体化ビューの総ストレージ容量を求められる．ストレージ容量に対する制約は以下の式となる．

$$\sum_i b_i \cdot x_i \leqq B_{max} \tag{3}$$

制約 2　重複評価制約

式 (2) では，サブクエリの利得を重複して評価する可能性があるために重複評価制約を設ける．ある実体化ビューをクエリ処理において利用する場合には，その実体化されたサブクエリが包含するサブクエリの実体化ビューが利用されないようにする．例えば，図 2 において，s_1 と s_3 を実体化する場合を考える．クエリ q_2，q_3 での利得は s_1, s_3 の利得を合計したものとすると，重複して利得を評価することになる．正しくは，s_3 のみの利得として評価するべきである．MIN_i を s_i に含まれるサブクエリの集合とすると，s_i を実体化する，つまり $x_i = 1$ の時，$x_j(j \in MIN_i) = 0$ とする制約を課すと，この制約は満たされる．

$$x_i + x_j \leqq 1(j \in MIN_i) \tag{4}$$

制約 3　クエリと実行プランの 1 対 1 制約

これは一つのクエリに対して用いられる実行プランは一つのみとする制約である．式 (2) では，1 つのクエリに対して y_i を全て 0 とする，つまり実行プランを一つも選ばない可能性があるため，この制約を設ける．この制約は以下のように定義できる．

$$\sum_t p_{it} = 1 \tag{5}$$

上記の制約はクエリの個数 n だけ存在するものである．

上記の制約を含んだ整数計画問題は，巨大なワークロードに対しては解の候補 (s_j) が膨大となり，解の導出に時間がかかる．そのため，解くための工夫が必要である．

3　提 案 手 法

提案手法は，図 4 のように局所探索法 [7] を用いて最適解を探索する．具体的には，サブクエリの出現頻度とそのコストを考慮してアルゴリズムの初期解を決定し，初期解の近傍に解空間を広げ，整数計画問題により解を決定する．ここでサブクエリの近傍とは，そのサブクエリを含むサブクエリ (親サブクエリ) や，そのサブクエリに含まれるサブクエリ (子サブクエリ) のことを表す．以降，近傍を解候補として整数計画問題により解を求める処理を繰り返すことで，最適解を探索する．

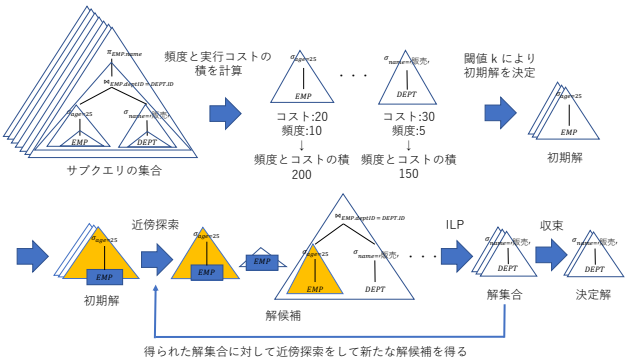

図 4　閾値による初期解の決定，近傍探索と整数計画問題の反復

以下が本稿で提案するアルゴリズムの入力と出力である．

- 入力:クエリワークロード W[2]，クエリの木構造，閾値 k，サブクエリのコスト，サブクエリのストレージ，クエリとサブクエリの対応関係．
- 出力:実体化するサブクエリ群，クエリワークロードの総実行コスト．

1：あるクエリの実体化ビューについて，実体化ビューのベーステーブルが更新された場合，実体化ビューの更新も必要となる．よって多数の実体化ビューを保持すると，更新処理の性能が劣化する．この更新処理性能の劣化が過度とならないようにこのストレージ制約条件を課している．

2：クエリの頻度情報は用いない

提案手法では 3.1.1 節に示す方法で予め解候補を削減するため，解候補に含まれるサブクエリのみの包含関係を考えれば良い．定式化は以下となる．

$$minimize \sum_i \sum_t (y_t \cdot (cost(p_{it}) - \sum_{j \in subquery(p_{it})} x_j \cdot cost(j)) \quad (6)$$

(6) 式を以下の制約の下で解く．

- 制約 1:ストレージ制約

$$\sum_i b_i \cdot x_i \leqq B_{max} \quad (7)$$

- 制約 2:重複評価制約

$$x_i + x_j \leqq 1 (j \in MIN_i) \quad (8)$$

- 制約 3:クエリと実行プランの 1 対 1 制約

$$\sum_t p_{it} = 1 \quad (9)$$

3.1 手　　法

3.1.1 初期解の列挙

局所探索法を有効に用いるために優れた初期解が必要となる．クエリワークロード全体に対して利得が上位であるサブクエリが初期解とする．この初期解を決定するために，s_j の頻度と s_j の実行コスト $cost(s_j)$ の積 e_j を定義する．e_j は s_j のワークロード全体での利得の合計を表す．q の実行プラン集合を出力する関数 $plans(q)$，実行プラン p のクエリプラン群を出力する関数 $subqueries(p)$ を用いて，サブクエリ s_j がワークロード内のクエリの全実行プラン中に現れる頻度 $freq(s_j)$ を以下のように定義する．

$$freq(s_j) = |q \in Q | s_j \in \bigcup_{p \in plans(q)} subqueries(p)| \quad (10)$$

$$e_j = freq(s_j) \cdot cost(j) \quad (11)$$

この e_j が高いサブクエリほど実体化した時，クエリワークロードの処理コストが小さくなる可能性が高い．e_j が指定の閾値を満たすサブクエリに限定し，初期解とする．

3.1.2 近傍探索による解候補の列挙

あるサブクエリと同じサブクエリが存在する場合にその子サブクエリは必ず存在し，異なる親サブクエリ内にもその子サブクエリは存在することがあるため，子サブクエリは親サブクエリより頻度が多い．しかし，親サブクエリは子サブクエリを包含した処理を行うために子サブクエリは親サブクエリより利得が小さい．逆に，親サブクエリは子サブクエリより頻度が少なく，利得が大きい．このような理由から，有用サブクエリの近傍もまた有用サブクエリだと推測して解候補に追加する．A に含まれるサブクエリの近傍のサブクエリを，次の整数計画問題の解候補として追加する．これによって得られる解候補を B_0 とする．

$$B_0 = A \cup \bigcup_{s_i \in A} neighbor\_search(a_i) \quad (12)$$

$neighbor\_search$ は近傍クエリ群を返す関数であり，以下のように定義する．$parent$ は引数の親サブクエリを返す関数，$children$ は引数の子サブクエリを返す関数である．

$$neighbor\_search(s_i) = parent(s_i) \cup children(s_i) \quad (13)$$

3.1.3 整数計画問題の繰り返しによる最適解の探索

提案手法では，整数計画問題と解候補の探索を反復する．得られた解候補 $B_n (n = 0 \cdots$ 反復回数) に対し，式 (6) の整数計画問題を解く．これによって得られる解を C_n とする．ILP は解候補に対して整数計画問題を解く関数である．

$$C_n = ILP(B_n) \quad (14)$$

得られた解の実体化ビューを用いたクエリワークロードの総実行コストが，一つ前の整数計画問題で得られた総実行コストより小さいならば，得られた解に対して近傍探索を再度実行する．

$$B_{n+1} = C_n \cup \bigcup_{c_{ni} \in C_n} neighbor\_search(c_{ni}) \quad (15)$$

ここで，あるサブクエリの近傍に最適な解の集合に近似できない場合においても，近傍のさらにもう一つ先の近傍に最適な解の集合に近似できる可能性がある．近傍のもう一つ先の近傍で反復を止める理由として，もう一つ先の近傍まで探索すると，解候補が増大することが挙げられる．最悪の場合，全てのサブクエリを解候補としてしまい，最適化の高速化が出来なくなってしまう．そのため，近傍探索の回数を制限する．以上のことから，以下の条件両方を満たした時，収束したと判定して反復を終了する．

- $n-1$ 回目の処理で選ばれた実体化ビューを用いた時の総実行コストが n 回目の処理で選ばれたビューを用いた時の総実行コスト以上
- $n+1$ の処理で得られる総実行コストも $n-1$ での総実行コスト以上

.

3.2 アルゴリズム

提案手法のアルゴリズムを Algorithm1 に示す．

$select\_cand$ は閾値 k 以上のサブクエリを検索する関数，行目 $neighbor\_serach$ は近傍探索，$interaction\_check$ は解候補同士の包含関係の確認，ILP は整数計画問題の関数である．ILP は整数計画問題によって計算された目的関数の値を配列の第 1 要素として返し，実体化に選ばれた解の集合を配列の第 2 要素として返す．

まず 3 行目で 3.1.1 節に示したように初期解を閾値 k により決定する．5 ～ 7 行目で，図 4 で示した，3.1.2 節で示した解候補に対して近傍探索と整数計画問題を実行している．10 ～ 17 行目は初期解に対して，または一つ前の反復における整数計画問題でクエリロードの総実行コストが改善された場合に実行する．12 ～ 15 行目は，3.1.3 節で示したように 8 行目で格納さ

Algorithm 1 提案手法のアルゴリズム

Input: クエリワークロード (workload)，クエリの木構造，閾値 k，サブクエリのコスト，サブクエリのストレージ，クエリとサブクエリの対応関係

Output: 実体化するサブクエリ，クエリワークロードの総実行コスト

```
1: iteration_n = 1
2: precost = MAX
3: ans_cand =select_cand(workload, k)
4: while (iteration_n! = 3) do
5:    ans_cand =neighbor_serch(ans_cand)
6:    interaction =interaction_check(ans_cand)
7:    result =ILP(ans_cand, interaction)
8:    cost = result[0]
9:    ans_cand = result[1]
10:   if (iteration_n == 1) then
11:      if (cost >= precost) then
12:         pre2cost = precost
13:         pre_ans_cand = ans_cand
14:         ans_cand =neighbor_serch(ans_cand)
15:         iteration_n = 2
16:      end if
17:   end if
18:   if (interaction_n == 2) then
19:      if (cost >= pre2cost) then
20:         interaction_n = 3
21:         ans_cost = pre2cost
22:         ans_view = pre_ans_cand
23:      else
24:         interaction_n = 1
25:      end if
26:   end if
27:   precost = cost
28: end while
29: return (ans_cost, ans_view)
```

れた総実行コストが一つ前の反復の総実行コストと比較して改善されなかった場合に実行される．これは 18 行目からの処理に進むために行われる処理である．クエリワークロードの総実行コストが改善された場合は選ばれた解を再び解候補として，5 行目へと戻り，処理が繰り返される．18 ～ 25 行目は，行目は一つ前の反復における整数計画問題で総実行コストが改善されなかった場合に行われる．20 ～ 22 行目は 8 行目で格納された総実行コストが二つ前の反復の総実行コストと比較して改善されなかった場合に実行される．4 行目によりクエリの総実行コストが改善されなくなった時点でアルゴリズムの反復は終了し，29 行目へと進み決定解と総実行コストを出力する．24 行目は 8 行目で格納された総実行コストが二つ前の反復の総実行コストと比較して改善された場合に実行される．この場合，選ばれた解を再び解候補として，5 行目へと戻り，処理が繰り返される．

4 実　　験

本章では，解候補を削減しないで整数計画問題を解く手法 (N_ILP) と比較して，提案手法の有効性を評価する．クエリ数の多寡による提案手法の有効性を評価するために，クエリ数を変化させる．更新性能の劣化具合による提案手法の有効性を評価するためにストレージ容量を変化させる．初期解による提案手法の有効性を評価するために閾値を変化させる．

4.1 ベンチマーク

本稿では，インターネットムービーデータベース (IMDB) を想定したワークロードである Join-Order-Benchmark (JOB) [9] を使用する．短時間で提案手法の性能を確認するために，クエリは結合するテーブルが 5 以下のものに制限した．JOB におけるクエリ数 23，10 の 2 パターンに対し実験を行った．

ここで，JOB のクエリ数は $1a$，$1b$，$1c$ のようなクエリは 3 つとして数えている．JOB で使用されるデータモデルは，構成するテーブル数が 21 と多い．また，ワークロードにはテーブルの結合処理を含むクエリが多数存在する．そのため，あるクエリに対して全体のワークロードで考えた時に最も最適となる実行プランを見つけるために，考慮すべき実行プランの数は多くなる．よって，整数計画問題によって処理される決定変数が多くなる．このような理由から本実験を行うワークロードとして適している．

4.2 実 験 設 定

制約条件はそれぞれ全てのクエリの処理結果を実体化するために必要なストレージの 95%，50%，35%，20% をストレージ制約と設定して実験を行った．提案手法で用いる初期解の閾値 k は初期解として選ばれるサブグラフの数が変わるように 2 パターン設定した．具体的な実験環境を表 1 に示す．

マシン	MacBook Pro (Retina, 13-inch, Early 2015)
cpu	2.7 GHz デュアルコア Intel Core i5
メモリ	8 GB 1867 MHz DDR3
solver	Gurobi9.1[3]
cost estimation	PostgreSQL[4]

表 1　実 験 環 境

4.3 実 験 結 果

表 2 には，各手法において，解候補となったサブクエリの数を示す．表 2 では，提案手法において実体化ビュー候補となったサブクエリ数として，複数の値を表記している．これは提案手法が整数計画問題をクエリ実行コストが改善されなくなるまで繰り返すため，各反復において実体化ビュー候補となったサブクエリの数を示している．また，ボールド体で示している部分は，二つ前の整数計画問題によって得られた解の近傍のもう一つ先の近傍まで解候補とした時の実体化ビュー候補の数である．

図 5，6 に実験結果を示す．クエリ数が 23，ストレージ制約

3：https://www.postgresql.jp/document/11/html/using-explain.html
4：http://www.gurobi.com

を 95% としたとき，N_ILP では 72 時間以内での最適化は不可能であった．それに対し提案手法では，許容する最適化時間，求めるクエリ応答時間に閾値を変えることで対応可能であった．

	ストレージ	閾値 (k)	実体化ビュー候補の数
クエリ数:23，N_ILP	50%		3267
提案手法		300000	1397,54,**172**,46,**165**
		800000	226,38,29,24,23,**36**,22,21,21,**32**
N_ILP	35%		3267
提案手法		300000	1397,37,**59**
		800000	226,31,25,23,21,**31**
N_ILP	20%		3267
提案手法		300000	1397,26,26,**43**
		800000	226,25,22,**37**
クエリ数:10，N_ ILP	50%		1742
提案手法		130000	826,21,**33**
		400000	328,14,**15**,14,13,**20**
N_ILP	35%		1742
提案手法		130000	826,20,20,**34**
		400000	328, 15,11,**13**,10,**15**
N_ILP	20%		1742
提案手法		130000	826,14,14,**23**
		400000	328,10,10,**16**

表 2　実体化ビュー候補となるサブクエリの数

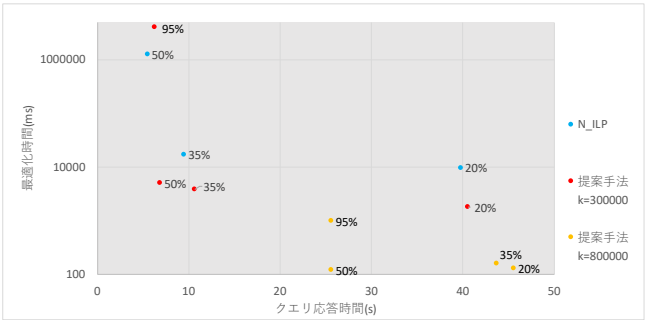

図 5　クエリ数 23 での実験結果

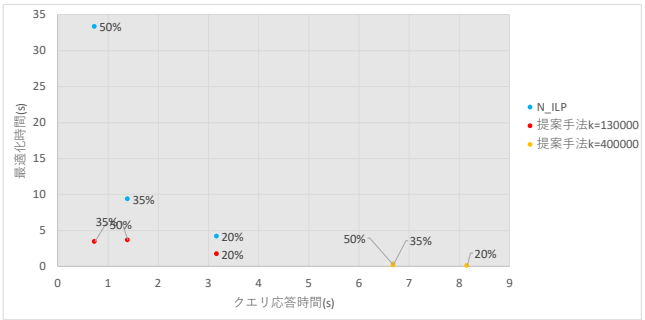

図 6　クエリ数 10 での実験結果

全てのストレージ制約，閾値においても，N_ILP に対して，提案手法は高速な最適化を達成した．特に，閾値を高くして実体化ビューとなる候補を極端に減らした場合は 0.01 ～ 4% にまで最適化時間を削減することが可能であった．このことにより，本稿で提案した実体化ビューの候補数削減は最適化時間において有効であることがわかった．

クエリ数が 23 でストレージ制約を 50% とした時，N_ILP では最適化のために 1262(s) と長時間を要している．これは整数計画問題の性質上，決定変数の個数が多くなるほど問題が難しくなり，制約条件が緩くなるほど，考えられる解の組み合わせが膨大となり，最適解を見つけるための時間がかかる傾向があるためである．同じクエリ数，制約条件下でも提案手法では，N_ILP の 0.01% という短時間で最適化を行うことができた．このことから特に，実体化ビュー候補の数が膨大である時，つまりワークロードが巨大である場合に提案手法が有効であるとわかった．N_ILP は，全てのサブクエリを実体化ビュー候補としているため，理想的な実体化ビューを選択する．そのため，N_ILP によって選ばれた実体化ビューを用いたクエリ応答速度が最も短い時間となっている．提案手法では，初期実体化ビュー候補を減らしすぎないように閾値を選択すると，クエリ数が小さい場合は同等のクエリ応答速度を達成した．しかし，クエリ数が大きい場合には，ストレージ制約が大きい方から順に，1.24 倍，1.12 倍，1.02 倍と制約が緩い方が応答時間が増加した．制約条件が厳しい場合では，N_ILP に比べて，最適化が 18% の時間で終了することに加え，クエリ応答速度は 1.02 倍の劣化に止めることができた．この原因として，提案手法の解候補の削減によって得られる初期解の特性がある．クエリ数 23，ストレージ制約 50% における二つの手法の例を示す．N_ILP で得られる解には，全てのクエリに関する実体化ビューが存在する．対して，提案手法で得られる解は，q_0，q_1，q_2，q_3 に関する実体化ビューは含まれない．これらの実体化ビューを以下では q_{out} と呼ぶことにする．q_{out} はプランの総実行コストが 50000 前後となっており，閾値によって q_{out} のサブクエリは解候補から外れてしまうからである．q_{out} はストレージ制約を 35%，20% とした場合には実体化ビューには選ばれない．よって，q_{out} はストレージ制約 50% において，N_ILP で得られる実体化ビュー群の中では，その利得は上位ではないがストレージに余裕があれば最適解となるクエリである．q_{out} のようなクエリを提案手法では無視してしまうため，ストレージ制約が緩い中では最適解から外れた結果を示すと考えられる．このことから，制約条件がより厳しい中で，提案手法はより最適解に近い実体化ビュー群を選択できることがわかった．

閾値を高くした場合は，実体化ビュー候補の数が減少するために，高速な最適化時間を提供できる．しかし，この最適化によって得られる実体化ビューを用いたクエリ応答速度は N_ILP に比べて，最悪で 4.8 倍，最良でも 1.15 倍となっている．これは，初期候補が少なくなっていることから，より良い解にたどり着くまでに，目的関数が改善しなくなり，局所解に陥ってしまうことによる．

5　関 連 研 究

データベース上のクエリ処理を高速化するために適切な実体化ビューを選択する研究が行われている [10], [11] [12], [3], [13], [4].

Harinarayan ら [10] の研究ではストレージ制約の条件下で貪欲アルゴリズムを用いて実体化ビューを選び．クエリの処理時

間を低減している．

T．Dokeroglu ら [11] は分枝限定法，HillClimbinf，Genetic，Hybrid Genetic-Hill Climbing アルゴリズムを含むヒューリスティックな手法を採用している．サブクエリが共通のタスクを共有する機会を増やすために各クエリの代替クエリプランを生成する．このプラン生成器はコストモデルと相互作用し，サブクエリを再利用する価値を導出する．そしてこれらの代替案の中からより低いコストのプランを提供する．

D.C.Zilo ら [12] は類似のビューをマージする，選択条件を変更する，group by 句を追加することにより抽出されたビューを一般化する．これによりストレージの消費を抑えてより多くの実体化ビューを作ることで，多くのクエリがビューを利用する．

BIGSUBs [3] では二部グラフのラベリング問題を用いることで，整数計画問題を細分化し，確率的なアルゴリズムを導入している．このアルゴリズムでは二つのステップを反復実行する．一つ目のステップではあるビューが実体化されるべきかどうかを確率的に判断する．この確率は (1) そのビューにおけるディスクの容量，(2) そのビューによる利得の二つにより計算される．2 つ目のステップでは細分化した整数計画問題を解く．これらの反復処理を整数計画問題におけるコスト関数が収束するまで繰り返す．この処理により実体化ビューを選択する．

BIGSUBs と類似した手法として CloudViews [4] がある．ワークロードに応じてコンパイル時間と実行時間の統計情報を調整し，各サブクエリの利得とコストの正確な値を収集するフィードバックループを実装することで実体化ビューを選択する．定期的なワークロードに焦点を当てているため，異なるパターンや分布を持つ新しいワークロードには迅速に対応できないことが欠点となる．

Wide-deep [13] 実体化するサブクエリのコストも課題とし，コスト推定モデルを提案している．ビュー選択問題では強化学習を用いている．

6 ま と め

提案手法はワークロード内のサブクエリの出現頻度とそのコストを考慮して解候補を削減し，整数計画問題を近傍探索を行いながら繰り返し解く．この手法により，解候補を削減せずに行う整数計画問題と比較して，高速な最適化をもたらすことができた．クエリ数が少ない場合と，ストレージ制約が厳しい場合に，N_ILP に対し，1 ～ 1.02 倍のクエリ応答時間を提供する実体化ビューを選択できた．

今後の課題として，以下の点が考えられる．

- ワークロードの拡張
- BIGSUBs [3] との比較
- 更新処理の性能を厳密に評価
- 初期解の選び方

提案手法の主な貢献点は，実体化ビューの品質を保ちつつ，最適化時間を低減することである．最適化時間が長く問題となるような場合に，提案手法は有効である．よって，提案手法は対象とするワークロードが巨大な場合により有効となる．そのために，巨大なワークロードを用いた評価実験も必要であると考える．

上記の課題に加え，巨大なワークロードに対応した，既存手法である BIGSUBs [3] との比較が必要であると考える．BIGSUBs と比較して，どのような場合に，どれ程，有用な成果を達成でき，現実世界のどのようなワークロードに有用であるかを調べる必要がある．

提案手法では，更新性能がストレージ制約によるものでしか考慮されていない．更新処理が頻繁に行われる場合を想定すると，更新処理の性能も厳密に考慮されるべき点となる．更新処理の性能の劣化の評価方法も重要である．利得が高いサブクエリは更新性能を劣化させる程度が大きいことが予想される．そのため，検索性能と更新処理の性能のトレードオフを厳密に評価し，トレードオフの中で最適な解を導き出すために，更新処理の性能も厳密に考慮した整数計画問題の定式化が必要であると考える．

提案手法では，初期解の選び方を頻度と実行コストの積を閾値とした解候補の削減を提案している．しかし，実験結果からも，この手法では局所解に陥ってしまうことがわかった．提案手法の場合，初期値として実行プランの木構造の根が初期解として選ばれやすくなっている．この場合，葉が探索されづらくなっているため，局所解に陥っている可能性が高い．局所解に陥るための対策を提案手法に実装し，最適解に漸近することはできた．しかし，まだ局所解に陥ることがわかった．より良い初期解を列挙し，より最適解に近い解集合を提案する方法が必要である．

7 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）の助成事業の結果得られたものです。

文　　献

[1] R.Ramakrishnan, B.Sridharan, J. Douceur, P.Kasturi, B.Krishnamachari-Sampath, K.Krishnamoorthy, P.Li, M.Manu, S.Michaylov, R.Ramon, N.Sharman, Z.Xu, Y.Barakat, C.Douglas, R.Draves, S.S.Naidu, S.Shastry, A.Sikaria, S.Sun, and R.Venkatesan, “Azure data lake store: A hyperscale distributed fileservice for big data analytics,” SIGMOD, pp.51–63, 2017.

[2] A.Y.Halevy, “Answering queries using views: A survey,” VLDB, pp.270–294, 2001.

[3] A. Jindal, K. Karanasos, S. Rao, and H. Patel, “Selecting subexpressions to materialize at datacenter scale,” PVLDB, pp.800–812, 2018.

[4] A. Jindal, S. Qiao, H. Patel, Z. Yin, J. Di, M. Bag, M. Friedman, Y. Lin, K. Karanasos, and S. Rao, “Computation reuse in analytics job service at microsoft,” SIGMOD, pp.192–203, 2018.

[5] Y.N.Silva, P.Larson, and J.Zhou, “Exploiting common-subexpressions for cloud query processing,” ICDE, pp.1337–1348, 2012.

[6] J.Zhou, P.Larson, J.C.Freytag, and W. Lehner, “Efficient exploitation of similar subexpressions for query processing,” SIGMOD, pp.533–544, 2007.

[7] 野々部宏司，柳浦睦憲，“局所探索法とその拡張-タブー探索法を中

心として," 特集 メタヒューリスティクスの新潮流, pp.493–499, 2009.
[8] R.Marcus, and O.Papaemmanouil, "Plan-structured deep neural network models for query performance prediction," VLDB, pp.1733–1746, 2019.
[9] V. Leis, rey Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?," VLDB, pp.204–215, 2015.
[10] V.Harinarayan, A.Rajaraman, and Ullman.J.D., "Implementing data cubes efficiently," SIGMOD, pp.205–216, 1996.
[11] T. Dokeroglu, M.A. Bayir, , and A. Cosar, "Robust heuristic algorithms for exploiting the common tasks of relational cloud database queries," Applied Soft Computing, pp.72–82, 2015.
[12] D.C. Zilio, C. Zuzarte, S. Lightstone, W. Ma, G.M. Lohman, R. Cochrane, H. Pirahesh, L.S. Colby, J. Gryz, E. Alton, D. Liang, and G. Valentin, "Recommending materialized views and indexes with ibm db2 design advisor," ICAC, pp.180–188, 2004.
[13] H. Yuan, G. Li, L. Feng, J. Sun, and Y. Han, "Automatic view generation with deep learning and reinforcement learning," ICDE, pp.1501–1512, 2020.