局所探索法を用いた高スケールな実体化ビュー選択

乗松　奨真†　　伊藤　竜一†　　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
E-mail: †{norimatsu.shoma,ito.ryuichi,sasaki,onizuka}@ist.osaka-u.ac.jp

あらまし　クエリ数が膨大なワークロードにおいて高速なクエリ処理のために，実体化ビューの利用が有効である．実体化する部分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ことと扱う変数の数が膨大になるため実行が困難である．本稿では，総利得の高い部分クエリの近傍は同様に総利得が高い傾向があるという性質を活用して，最適な実体化ビュー選択の問題を局所探索法を用いて解決する．これは総利得の高い部分クエリの近傍は同様に総利得が高い傾向があるためである．有効な局所探索法のためには，適切な初期解を設定して局所解に陥らないようにするため，ワークロードのおける様々な統計情報の top-k を初期解として利用する．初期解選択後，近傍探索により解候補を改善し，新たな解候補に対し整数計画問題を解き，得られた解に対して再び近傍探索を行うという処理を繰り返すことで最適解へと漸近する．実験では小規模な Join Order Benchmark と大規模な天文台データベースでの実験により，既存技術である BIGSUBS の問題点を明確にすると共に，提案手法の有効性を示した

キーワード　問い合わせ処理，データベース技術，実体化ビュー

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]．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問題となっている．

本稿では，この問題を解決するたため大量クエリ処理には多くの共通した処理が含まれているという特徴 [3] [4] を利用したクエリ処理高速化の課題に取り組む．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，実体化ビューを再利用することで，高速な検索クエリ処理を実現できるようになる [3] [4]．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．また，実体化ビューはクエリの処理結果を物理的に保存するため，容量を要する．よって，実体化ビューの保存容量をどれほど許容するかを示すストレージ容量制約が必要となる．これらのトレードオフや制限を考慮しつつ，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．このような背景のもと，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 [2] [3] [4] [5] [6]．例えば BIGSUBS [2] では，巨大なワークロードを対象としているおり，実体化ビュー選択問題とクエリが利用する実体化ビューを決定する問題を分けて考えることで，整数計画問題を高速化する．BIGSUBS では，ストレージ容量制約を課すことで，実体化ビューの容量が制限されるため，同時に更新性能の過度な劣化を防ぐことができる．この手法では，利得とストレージ容量制約の両方を考慮し，容量当たりの利得の良い実体化ビューのみを選択できる．しかし，ストレージ容量制約を使い切り利得を最大化する解から離れてしまうという問題点が存在する．また，確率を用いた実体化ビュー選択を行うため，解が振動してしまうという問題点も存在する．本稿では，解の振動の問題と，容量を使い切らない問題を解決し，実体化ビューによる利得を最大に漸近させる最適化を高速化する手法を提案する．具体的には，メタヒューリスティックである局所探索法 [7] を利用することで解の探索を高速化する．実体化ビュー選択問題に対して局所探索法を適用するには，近傍解を適切に決定することおよび初期解の選択方法が技術課題である．近傍解を適切に決定する方法においては，あるサブクエリの包含関係にあるサブクエリを元のサブクエリの近傍であると定義する．そしてクエリワークロードにおいて近傍の関係にあるサブクエリ同士はそれらの利得が類似するという特性を活用する．具体的には，包含関係にある親サブクエリの方が子サブクエリよりもクエリ表現が複雑であるため，実体化することで得られる利得がより大きく，逆に子サブクエリの方がより多くのクエリで利用される可能性が高い．この特性を活用して，局所探索法における近傍解を決定し，解が収束するまで最適解の探索を実行する．次に初期解の選択方法に関しては，利得が高い解を高速に選択できるように，サブクエリに関する情報を用いて，利得，容量当たりの利得，ワークロード内での頻度がそれぞれ上位のサブクエリを初期解とする 3 つの方法を提案する．実験は提案手法による実体化ビュー選択が高速であること，選択された実体化ビューにより高い利得が得られること，実体化ビューを利用したワークロードが高速であること，局所探索に効果があることを確認するために行った．Join Order Benchmark [8] のデータベースとワークロードを用いた低スケールな実験と天文台のデータベースとワークロードを用いて高スケールな実験を行った．このベンチマークは結合処理が多く含まれ，整数計画問題で扱う変数が多くなり，最適解を

求める時間が多くかかるため，実験に用いるワークロードとして適する．これを用いた実験により，提案手法や最新技術である BIGSUBS [2] の性能を確認する．天文台のデータベースとワークロードは実世界データを用いており，これを用いた実験により，実環境での提案手法や BIGSUBS の性能を確認する．これらの実験により，BIGSUBS の特徴と問題点を明確にすると共に，提案手法の有効性を示した．

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，局所探索を用いて整数計画問題を高速化する．整数計画問題で用いられる変数の詳細について 2.1 節において前提知識として説明する．次に，2.2 節でワークロードにおける利得を最大化する最適化問題を解くために用いられる整数計画問題の定式化を説明する．本稿では，ワークロード W に含まれるクエリ数を n，サブクエリ数を m として，W に含まれるクエリ $q_i(1 \leq i \leq n)$ 集合を Q，サブクエリ $s_j(1 \leq j \leq m)$ の集合を S とする．また，クエリの実行プランは木構造で表すことができる．

2.1 事前知識

サブクエリの利得: 実体化ビュー選択問題はワークロード全体の実行コストを下げる実体化ビュー群を適切に選択する問題である．この実体化ビューを評価するために利得を用いる．利得は実体化ビューを利用したクエリと利用しないクエリの実行コストの差で計算する．サブクエリの実行コストは，クエリ最適化で用いられる推定実行コストを用いる．BIGSUBS [2] の定義に従って利得を以下のように定義する．

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

但し，実体化ビューを利用しないで q_i を処理する際のコストを $cost(i)$，s_j を実体化したビューを利用して q_i を処理する際のコストを $cost(i|s_j)$ とする．また，以下の式のようにワークロードに含まれる各クエリを q_i とし，そのサブクエリ s_j を実体化することで得られる利得 u_{ij} の全クエリにおける和，つまり s_j のワークロード全体における総利得を U_j とする．

$$U_j = \sum_{i=1}^{n} u_{ij} \tag{2}$$

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 1 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 q_2, q_4 に共通し，s_3 がクエリ q_2, q_3 に共通している．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，再利用することで複数のクエリが高速化され，ワークロード全体の処理

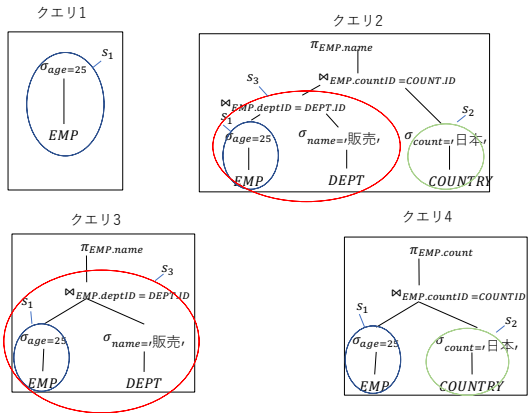

図 1　異なるクエリ間でのサブクエリの重複

が高速化される．また，このように複数クエリにまたがって同じサブクエリが存在するために，実体化ビュー選択問題は複雑になる．

2.2 整数計画問題を用いた実体化ビュー選択

この節では整数計画問題の定式化について説明する．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．しかし，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．また，多くのクエリ処理の結果が実体化ビューとして保持されていると，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．そこで，BIGSUBS [2] では，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容量制約を課すことで実体化ビューの容量を制限し，更新処理性能の低下を抑えている．これらのことを考慮しつつ，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．実体化ビュー選択問題は以下のように定義される．整数計画問題に対する入出力は以下である．

定義 1. 入力:W，u_{ij}，b_j，x_{jk}，B_{max}
u_{ij} はクエリ q_i で実体化したサブクエリ s_j を利用する際の利得，b_j はの s_j の容量，x_{jk} は s_j が s_k を包含する場合に 1 をとるバイナリ変数，B_{max} が実体化できるサブクエリの上限容量である．

定義 2. 出力: 実体化することでワークロード全体の利得を最大化できるサブクエリ群，得られる合計利得．

実体化ビュー選択問題は整数計画問題を用いて以下のように定式化できる．

$$maximize \sum_{i=1}^{n}\sum_{j=1}^{m} u_{ij} \cdot y_{ij} \tag{3}$$

$$s.t. \sum_{j=1}^{m} b_j \cdot z_j \leq B_{max} \tag{4}$$

$$y_{ik} + \frac{1}{m}\sum_{\substack{j=1 \\ j\neq k}}^{m} y_{ij} \cdot x_{jk} \leq 1 \qquad \forall i \in [1,n], k \in [1,m] \tag{5}$$

$$y_{ij} \leq z_j \qquad \forall i \in [1,n], j \in [1,m] \tag{6}$$

(3) 式がこの最適化問題の目的関数である．この目的関数は W 全体における実体化ビューの利得を最大化するものである．(4)〜 (6) 式は最適化を行う際に，満たすべき制約を示している．但し，整数計画問題扱う変数は以下である．z_j は s_j が実体化される場合に 1 をとるバイナリ変数，y_{ij} は q_i が実体化された s_j をクエリ処理の際に利用する場合に 1 をとるバイナリ変数とする．(4) 式は実体化ビューの容量を制限するストレージ制約である．この制約はストレージ制約を課している．また，ストレージ制約により実体化ビューの個数は制限され，データベース更新があった場合の実体化ビュー再作成コストの過度な増加を防いでいる．(5) 式はサブクエリの利得を重複して評価しないための制約である．この制約に関しては 2.3 節で後述する．また，(3) 式で，$\sum_{i=1}^{n} y_{ij}$ が 1 以上となる j の集合を M_j とする．この場合，$s_k(k \in M_j)$ の実体化ビューは 1 つ以上利用されている．つまり，$b_k = 1(\forall k \in M_j)$ とする必要がある．この制約式が (6) 式である．(3)〜 (6) 式は BIGSUBS [2] や Wide-deep [4] でも同様の整数計画問題が定義されているが，クエリ内に集約演算を含む場合，実体化した際の容量は更新コストを適切に表現できないため，更新処理性能の問題を解決することはできない．この観点から本稿では，ワークロードに更新クエリを含む場合，更新コストを予測し，整数計画問題に組み込む．更新方法は PostgreSQL [9] で開発されているインクリメンタルビューメンテナンス (IVM) [10] を想定する．この手法は実テーブルに発生した変化分に応じてデータベースに格納されている実体化ビューの一部分だけを更新するという手法であり，一から再計算する完全リフレッシュと比較して短時間で実体化ビューの更新が可能である [11]．本稿では，更新クエリは 1 クエリにつき，1 テーブルに対して，1 レコードを更新するものとする．以下では，A テーブルと B テーブルを用いて検索を行うサブクエリ $s_j(s_{serach}(A|B))$ の実体化ビューを $MV(s_{serach}(A|B))$ とする．B テーブルが $B+dB$ へと更新され，$MV(s_{serach}(A|B))$ が $MV(s_{serach}(A|B+dB))$ となる場合のコスト M_j を考える．この際 IVM による更新コスト評価において，考慮するものは以下の 3 つである．

実体化ビューにおける更新対象レコードを特定するコスト: このコストは B テーブルから更新対象を見つけるコストと等価であるため，クエリ ($select \ * \ from \ B \ where$ 主キー $= 1$) のコストとなる．このコストを $cost(search(B))$ とする．

実体化ビューを更新する際に必要な実体化ビュー再処理コスト: このコストは $MV(s_{serach}(A|B))$ の構築コストの $1/|B|$ 倍である ($|B|$ は B のレコード数)．このコストを $cost(remakeMV(s_{search}(A|B)))$ とすると，以下の式となる．

$$cost(remakeMV(s_{search}(A|B))) = cost(s_{serach}(A|B)) \cdot 1/|B| \tag{7}$$

総書き込みコスト: 更新クエリでは，delete，insert とこの二つの組み合わせである update が存在する．この insert コストと delete コストを取得することで全ての更新に対応可能であるが，本稿では，insert のみを想定する．$MV(s_{serach}(A|B))$ へと更新差分を書き込むコストを $cost(insert(MV(s_{serach}(A|B))))$ とする．このコストは B テーブルへの insert コスト ($cost(insert(B))$) と $MV(s_{serach}(A|B))$ のカラム数 $column(MV(s_{serach}(A|B)))$，B テーブルのカラム数 $column(B)$ を用いて以下の式となる．

$$cost(insert(MV(s_{serach}(A|B)))) = cost(insert(B)) \cdot column(MV(s_{serach}(A|B)))/column(B) \tag{8}$$

次に書き込みレコード数 $n_{insert(MV(s_{serach}(A|B)))}$ は $|B|$ と $MV(s_{serach}(A|B))$ のレコード数 $|MV(s_{serach}(A|B))|$ を用いて以下の式となる．

$$n_{insert(MV(s_{serach}(A|B)))} = |MV(s_{serach}(A|B))|/|B| \tag{9}$$

総書き込みコスト $cost(ALLinsert(MV(s_{serach}(A|B))))$ は insert コストと書き込みレコード数の積である．

$$cost(ALLinsert(MV(s_{serach}(A|B)))) = n_{insert(MV(s_{serach}(A|B)))} \cdot cost(insert(MV(s_{serach}(A|B)))) \tag{10}$$

以上の変数を用いて M_j は以下の式となる．

$$\begin{aligned} M_j = {} & cost(search(B)) \\ & + cost(remakeMV(s_{search}(A|B))) \\ & + cost(ALLinsert(MV(s_{serach}(A|B)))) \end{aligned} \tag{11}$$

この更新コスト M_j を用いてワークロード内に更新クエリを含む場合の整数計画問題の目的関数を以下のように定義する．

$$maximize \sum_{i=1}^{n}\sum_{j=1}^{m} u_{ij} \cdot y_{ij} - z_j \cdot M_j/n \tag{12}$$

2.3 重複評価制約

一つのクエリに対して実体化ビューの利得を複数重複して評価しないために，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同じクエリ処理において利用されないようにする必要がある．上記のようにクエリが利用する実体化ビューによる利得を適切に評価するための制限式が (5) 式である．(5) 式の制限を設けず式 (3)〜(6) の整数計画問題を解く場合サブクエリの利得を重複して評価する場合があるが，これは

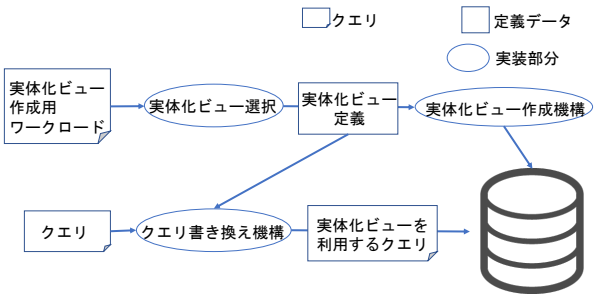

図 2　実体化ビュー利用までの流れ

適切な評価ではない．このような適切でない評価を防ぐため，y_{ik} が 1，つまり q_i が実体化された s_k を利用する場合，q_i が s_k が包含する他のサブクエリを利用できない制限を設ける．s_k が包含する全ての他のサブクエリが q_i によって利用されているか否かは $y_{ij} \cdot x_{jk}$ によって表される．y_{ik} が 1 の場合は，$y_{ij} \cdot x_{jk}$ は全て 0 である必要がある．y_{ik} が 0 の場合は，$y_{ij} \cdot x_{jk}$ は自由に選択できる．例えば，図 1 において，s_1 と s_3 を実体化する場合を考える．q_2 において，s_3 が s_1 を包含しているため，q_2 において削減できるコストは s_3 のコストのみである．ここでクエリ q_2 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り，過剰な利得が得られてしまう．ここで，(5) の制限式を設けることにより，s_3 のみの利得として評価することができる．

2.4　局所探索法

提案手法は局所探索法を用いて実体化ビュー選択を高速化するが，本節では一般的な局所探索方法 [7] について説明する．局所探索法は近似アルゴリズムの中での最も単純なアルゴリズムの枠組みの 1 つで，良質な近似解を求めることを主な目的としている．このアルゴリズムは以下のように実装される．

1. 解を 1 つランダムに生成する．
2. 現在の解の近傍の内 1 つをある条件で選び近傍解とする．
3. 定義した条件を満たすなら近傍解と現在の解を入れ替える．
4. 終了条件を満たすまで 2 以下を繰り返す．

このアルゴリズムにおいて定義するべきパラメータは以下の 4 つである．

- 近傍の定義
- 近傍解を選ぶ条件
- 近傍解と現在の解を入れ替える条件
- 終了条件

この局所探索法の提案手法への適用方法や，提案手法におけるパラメータの定義は 3 章にて詳しく説明する．

3　提 案 手 法

ワークロードの情報を用いて，実体化ビューをデータベース上に作成し，この実体化ビューを用いてクエリをデータベース上で実行する実際の流れは図 2 の通りである．まず，ワークロードの情報を用いて実体化ビュー選択技術により，実体化するサブクエリを決定する．その後，実体化ビューとして選択されたビューの定義情報を用いて実体化ビューを作成するクエリを発行し，このクエリをデータベース上で実行することで実体化ビューを作成する．そして，データベース上で実行したいクエリを，実体化ビューの定義情報を用いて実体化ビューを利用するように書き換える．このクエリをデータベース上で実行することで，実体化ビューを利用する高速なクエリ実行が可能となる．本稿において実装した部分は，提案手法である実体化ビュー選択部分，実体化ビュー作成機構，クエリ書き換え機構である．本章では，提案手法における実体化ビュー選択手法についてその概要を 3.1 節，詳細を 3.2 節で説明し，3.3 節で実体化ビュー作成とクエリ書き換えについて説明する．

3.1　実体化ビュー選択手法の概要

提案手法は，メタヒューリスティックな手法である局所探索法 [7] による考慮する変数削減と整数計画問題を用いて，得られる解をワークロード高速化に最も寄与する実体化ビュー群である最適解に漸近する手法である．ここで実体化ビュー選択問題における解は実体化対象となるサブクエリ群である．以降，局所探索法を用いて最適な実体化ビューを選択するアルゴリズムの概要を説明する．まず s_j の総利得 U_j，容量当たりの総利得 E_j，W 内に出現するサブクエリ s_j の頻度 F_j を算出し全サブクエリの中から上位 k 件 $(top - k)$ を選択し，b_j の総和が B_{max} を満たすまで初期解として選択する．その後初期解の近傍に解候補を広げ，得られた解候補に対して整数計画問題を解くことにより解を決定する．この解の決定が局所探索法における近傍解の入れ替えである．ここでサブクエリ s_i の近傍クエリ $neighbors(s_i)$ を，s_i を含む親サブクエリおよび s_i に含まれる子サブクエリの和集合として，以下のように定義する．

定義 3. 近傍クエリ: サブクエリ s_i に対して，s_i を含む親サブクエリおよび s_i に含まれる子サブクエリの和集合として定義される．ただし，木構造における直接の親子関係以外の推移的な包含関係は考慮しないこととする．

$$neighbors(s_i) := parents(s_i) \cup children(s_i) \qquad (13)$$

但し，$parents$ は引数 s_i の親サブクエリ群を全クエリ群から探索して返す関数であり，$children$ は引数の子サブクエリ群を返す関数である．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．終了条件は実体化ビューにより得られる利得が改善しなくなる場合とする．局所探索法を利用することで，提案手法の整数計画問題は (3)〜(6) 式と比較して，解候補が削減されるため処理する変数が削減される．つまり，全ての式において解候補であるサブクエリ s_j の探索範囲が狭まり，解候補に含まれるサブクエリのみを扱えば良くなる．t 回目の反復において整数計画問題で扱う解候補群を $B_t (t \leqq$ 反復回数$)$ とすると，以下のように定式化される．

$$maximize \sum_{i=1}^{n} \sum_{j \in B_t} u_{ij} \cdot y_{ij} \tag{14}$$

$$s.t. \sum_{j \in B_t} b_j \cdot z_j \leq B_{max} \tag{15}$$

$$y_{ik} + \frac{1}{m} \sum_{\substack{j \in B_t \\ j \neq k}}^{m} y_{ij} \cdot x_{jk} \leq 1 \qquad \forall i \in [1,n], k \in [1,m] \tag{16}$$

$$y_{ij} \leq z_j \qquad \forall i \in [1,n], j \in B_t \tag{17}$$

全ての式においてサブクエリ j の範囲が解候補群 B_t 内に含まれる j に限定される．また，ワークロード内に更新クエリを含む場合の整数計画問題は以下のように定式化される．

$$maximize \sum_{i=1}^{n} \sum_{j \in B_t} u_{ij} \cdot y_{ij} - z_j \cdot M_j/n \tag{18}$$

$$s.t. \sum_{j \in B_t} b_j \cdot z_j \leq B_{max} \tag{19}$$

$$y_{ik} + \frac{1}{m} \sum_{\substack{j \in B_t \\ j \neq k}}^{m} y_{ij} \cdot x_{jk} \leq 1 \qquad \forall i \in [1,n], k \in [1,m] \tag{20}$$

$$y_{ij} \leq z_j \qquad \forall i \in [1,n], j \in B_t \tag{21}$$

3.2 実体化ビュー選択手法の詳細

3.2.1 初期解の列挙

近傍探索では局所解に陥る可能性があるが，これを初期解の選択によって解決している．図**??**のようにワークロードに含まれるクエリの統計情報であるサブクエリの利得，頻度，容量を用いて初期解を決定する．この初期解選択方法として 3 つ列挙する．

topk-U: 実体化ビューの利得が高い順に k 個を列挙する手法である．この手法では単純に，利得の高いサブクエリを選択するため，最適解に含まれるサブクエリを効率的に探索できる．ただし，容量が巨大で，非常に効率の悪いサブクエリを選択してしまうことが問題点である．

topk-E: 実体化ビューの容量当たりの利得が高い順に k 個を列挙する手法である．ワークロードに全体における容量当たりの総利得 E_j は以下のように定義される．

$$E_j = U_j/b_j \tag{22}$$

この手法は，$topk-U$ と比較して容量も考慮する手法となっており，$topk-U$ と同様に，最適解に含まれるサブクエリを効率的に探索できる．ただし，容量も考慮するため，容量が巨大であるが多くのクエリに含まれるサブクエリを探索できない可能性があることが問題点である．

topk-F: ワークロードにおける s_j の出現頻度が高い順に k 個列挙する手法である．$topk-F$ は W での頻度の高い s_j が初期解となり，多くのクエリに共通するサブクエリが初期解となることにより，後の近傍探索により広い範囲での探索が行え，得られる解が最適解に漸近しやすくなる．ただし，初期解の数が膨大となるため，整数計画問題の処理時間が長くなるという問題点がある．

3.2.2 近傍探索による解候補の列挙

クエリの木構造において，子サブクエリの方がクエリ表現が簡略であるため，親サブクエリよりワークロード内での出現頻度が高い．また，同様に親サブクエリの方がクエリ処理が複雑であるため，子サブクエリよりクエリ内での利得が高い．このような理由から，解候補となっているワークロード全体における総利得の高いサブクエリの近傍クエリもまた総利得が高いと推測して解候補に追加する．

3.2.3 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．以下の式のように解候補 $B_t(t = 0 \cdots$反復回数$)$ は一つ前の反復における解 C_{t-1} に対して近傍探索をおこなうことで得られる．

$$B_t = C_t \cup \bigcup_{c_t \in C_t} neighbors(c_t) \tag{23}$$

但し，C_0 は 3.2.1 節に示した方法で得られる初期解群とする．C_t $(t > 0)$ の場合は，以下の式によって，解候補 B_t に対し整数計画問題を解くことで得られる解であると定義する．

$$C_{t+1} = ILP(B_t) \tag{24}$$

ここで ILP は解候補に対して整数計画問題を解く関数である．この反復は得られた解の実体化ビューにより得られる総利得が改善しなくなった場合に終了とする．この手法では整数計画問題の目的関数の値は必ず収束する．その理由は，整数計画問題に与えられる変数は必ず，1 つ前の反復において得られた解を含んでいるためである．このため，1 つ前の反復における整数計画問題の目的関数と比較して，同じ値か，良い値を取ることとなる．この同じ値を取った時にアルゴリズムを終了としている．

3.3 実体化ビューの作成，クエリ書き換え

本稿では，解として得られた実体化ビューを実際に作成して，その実体化ビューを利用するようクエリ書き換えを行うシステムを開発した．本システムでは，PostgreSQL [9] の EXPLAIN コマンドにより出力される実行プラン内のサブクエリ群を実体化候補とする．しかし，このコマンドの仕様上，いくつかの問題点があったため，工夫を施した．その問題点と工夫について以下に示す．

LIMIT OFFSET 句: PostgreSQL の EXPLAIN コマンドでは，LIMIT OFFSET 句を含むサブクエリは OFFSET で指定される値が異なる場合でも同じ EXPLAIN 結果を返す．しかし，実際には OFFSET の値がずれている分，違うサブクエリ出力結果を返すことが必要である．EXPLAIN 結果からは違うサブクエリ出力結果であることが判別できず，実体化ビューとする時に正しい結果とならない．このため LIMIT OFFSET 句を含む場合は 1 から始まる OFFSET のみに限定することした．

Nested Loop Join: Nested Loop Join は繰り返しにより結果を出力する処理であるが，EXPLAIN コマンドではこの部分の下位のサブクエリは 1 回分しか出力されず，適当なコスト

計算はできない．このため，Nested Loop より下位のサブクエリは整数計画問題の解候補から外した．

Bitmap Index Scan: Bitmap Index Scan は実行プラン内で Bitmap を作成することで，処理を高速化する処理であるが，クエリとして Bitmap を作成することはできない．このため，Bitmap Index Scan より下位のサブクエリは整数計画問題の解候補から外した．

4 実　　験

本稿における実験の目的は BIGSUBS [2] と提案手法の性能比較を行うことと，実体化ビューの実際の効果を測定することである．実験で測定する評価指標はアルゴリズム実行時間 (実行時間)，整数計画問題によって得られる合計利得，実体化ビューが使用する容量である．また，実体化ビューの実際の効果を測定する際には，実体化ビュー作成時間と，実体化ビューを利用したワークロード実行時間を測定する．実験では，Join-Order-Benchmark(JOB) と天文台のワークロードを用いた．このワークロードに関しては 4.1 節で詳しく説明する．また，実験はワークロードと目的別に数種類行った．実験では，解候補を削減しないで行う整数計画問題 (N_ILP) により得られる最適解および BIGSUBS [2] と比較して，提案手法の有効性を評価する (実験 $1-1, 2-1$)．次に，近傍探索の有効性を評価するために，3 つの初期解に対して整数計画問題を適用した近傍探索を行わない場合との比較を行う (実験 $1-2, 2-2$)．次に，ワークロード内に更新クエリが存在する場合の実験を行う (実験 $1-3$)．JOB のワークロード内には更新クエリが存在しないため，更新クエリは自作で追加した．最後に，実体化ビューが実際にどの程度の効果をもたらすかを確認する $(3-1)$．但し，BIGSUBS では反復的アルゴリズムで最後に得られる結果が最適値とは限らないため，解の探索途中で得られた，最も性能が高い解の結果を掲載した．サブクエリの実行コストには，PostgreSQL の explain コマンドを用いてサブクエリの推定実行コストを用いる．

4.1 ベンチマーク

本稿では，二つのデータベースとワークロードを用いる．

一つ目は，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) [8] である．このワークロードは実験用の人工ワークロードである．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．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．決定変数が多いと整数計画問題を解くコストが大きくなるため，本実験を行うワークロードに選択した [1]．但し，本ワークロード内では，サブクエリを実体化した際の容量で実体化ビューの更新コストを適切に表現するためにクエリ内の SELECT 句に指定される集約演算を削除した．

マシン	MacBook Pro (Retina，Early 2015)
cpu	2.7 GHz dual Intel Core i5
メモリ	8 GB 1867 MHz DDR3
ILP solver	Gurobi9.1 [2]
データベース	IMDB
データベースサイズ	7GB
ワークロード	JOB(109 クエリ)
ワークロード予測実行コスト	200M

表 1　実験 1 環境，データベース，ワークロード

マシン	MacBook Pro (Retina，Early 2015)
cpu	2.7 GHz dual Intel Core i5
メモリ	8 GB 1867 MHz DDR3
ILP solver	Gurobi9.1 [3]
データベース	天文台
データベースサイズ	14TB
ワークロード	天文台ワークロード (5000 クエリ)
ワークロード予測実行コスト	69G

表 2　実験 2 環境，データベース，ワークロード

ワークロード実行マシン	Ubuntu16.04.1，x86˙64 GNU/Linux
ワークロード実行 cpu	2.1GHz Intel(R) Gold6130 CPU*64
ワークロード実行メモリ	1.6TB
実体化ビュー選択マシン	MacBook Pro (Retina，Early 2015)
実体化ビュー選択 cpu	2.7 GHz dual Intel Core i5
実体化ビュー選択メモリ	8 GB 1867 MHz DDR3
ILP solver	Gurobi9.1 [4]
データベース	天文台
データベースサイズ	14TB
ワークロード	天文台ワークロード (500 クエリ)
ワークロード予測実行コスト	1.7G
実測ワークロード実行時間	20.54h

表 3　実験 3 環境，データベース，ワークロード

二つ目は，天文台データベース [12] とそこで処理されているワークロードである．このデータベースは実際に天文台で運用されているものである．データベース容量は $14TB$ と高スケールとなっている．また使用するワークロード内のクエリも実際に発行されているものであるため，提案手法が実環境でも有効であるかを確認することを目的とする．

4.2 実 験 環 境

実験において利用する三つの実験環境およびデータベース，ワークロードを表 1，2，3 に示す．各実験がいずれの実験環境を使用しているかは各実験結果を示す節の最初に記す．ワークロード予測実行コストはワークロードに含まれる全てのクエリの予測実行コストの和のことである．

4.3 実 験 結 果

ストレージ容量制約の大小によって実体化ビューに選ばれるサブクエリは変化する．提案手法の有効性のストレージ容量制

1：postgreSQL による cost 予測値，容量予測値が極端に大きいクエリを外し，109 クエリで実験を行った．

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

容量制約:0.1GB	実行時間 (s)	合計利得 (M)	使用容量 (%)
N_ILP	485	39.8 (最適値)	99.9
BIGSUBS	1.28	31.5	24.3
最適値との比較 (%)		79.1	
提案手法 ($topk-E$)	3.12	37.1	99.9
最適値との比較 (%)		93.2	
提案手法 ($topk-U$)	2.88	28.0	99.9
最適値との比較 (%)		70.4	
提案手法 ($topk-F$)	5.39	31.9	99.9
最適値との比較 (%)		80.2	

表 4　実験 1 – 1 結果 (容量制約:0.1GB)

容量制約:10GB	実行時間 (s)	合計利得 (M)	使用容量 (%)
N_ILP	523	60.1 (最適値)	99.9
BIGSUBS	10.87	54.8	9.2
最適値との比較 (%)		91.2	
提案手法 ($topk-E$)	8.12	46.4	99.9
最適値との比較 (%)		77.2	
提案手法 ($topk-U$)	6.05	56.1	99.9
最適値との比較 (%)		93.3	
提案手法 ($topk-F$)	7.61	45.0	99.9
最適値との比較 (%)		74.9	

表 5　実験 1 – 1 結果 (容量制約:10GB)

約の変化に応じてどのように変化するか確かめるために，ストレージ容量制約である B_{max} を変化させて実験を行った．

4.3.1　実験 1 – 1: 最適解，BIGSUBS との比較

この実験は N_ILP により得られる総利得の最適値，BIGSUBS と提案手法により得られる総利得とアルゴリズム実行時間，使用容量を比較するために行った．実験環境は表 1 である．それぞれの結果を表 4，5 に示す． BIGSUBS は容量制約を使い切っていないことから，最適値には近似できない．上記に対して提案手法では容量を使い切ることで，どのストレージ制約の場合でも N_ILP により得られる最適値と比べて 75% ～ 97% 程度の精度を達成した．

4.3.2　実験 1 – 2: 更新クエリがある場合の最適解，BIGSUBS との比較

この実験はワークロード内に更新クエリが存在する場合の提案手法の有効性を確かめるために行った．ストレージ容量制約は $10GB$ で，ワークロード内における検索クエリと更新クエリの比率を変化させた際の結果を確かめた．具体的には 検索 : 更新 = 109 : 10，検索 : 更新 = 109 : 1000 での 2 つで実験を行った．実験は JOB を用いて行なったが，このワークロードには更新クエリは存在していないため，更新クエリは手動で作成した．実験環境は表 1 である．それぞれの結果を表 6，7 に示す． 更新クエリ数が増加していくにつれて，最適値への近似が難しくなっていることがわかった．これは最適値へと近似するために実体化するべきサブクエリは多くのテーブルの処理を含まない小さなサブクエリを多く含むからであり，BIGSUBS や提案手法ではこの小さなサブクエリまで探索できていないと考えられる．BIGSUBS と提案手法で比較すると，ワークロードに更新クエリを含む場合でも，アルゴリズム実行時間，合計利得共に，提案手法が優れている傾向にあった．

容量制約:10GB，更新クエリ数:10	実行時間 (s)	合計利得 (M)	使用容量 (%)
N_ILP	559	59.5 (最適値)	99.9
BIGSUBS	4.83	43.0	41.1
最適値との比較 (%)		72.3	
提案手法 ($topk-E$)	3.24	52.9	99.9
最適値との比較 (%)		88.9	
提案手法 ($topk-U$)	4.93	54.9	99.9
最適値との比較 (%)		92.3	
提案手法 ($topk-F$)	9.26	45.3	97.7
最適値との比較 (%)		76.2	

表 6　実験 1 – 2 結果 (容量制約:10GB，更新クエリ数:10)

容量制約:10GB，更新クエリ数:1000	実行時間 (s)	合計利得 (M)	使用容量 (%)
N_ILP	469	57.3 (最適値)	99.9
BIGSUBS	3.98	39.6	17.9
最適値との比較 (%)		69.1	
提案手法 ($topk-E$)	1.99	45.6	95.9
最適値との比較 (%)		86.8	
提案手法 ($topk-U$)	1.20	49.8	99.6
最適値との比較 (%)		86.8	
提案手法 ($topk-F$)	3.85	38.7	99.4
最適値との比較 (%)		67.6	

表 7　実験 1 – 2 結果 (容量制約:10GB，更新クエリ数:1000)

容量制約:10GB	実行時間 (s)	合計利得 (G)	使用容量 (%)
BIGSUBS	7021	28.5	39.2
提案手法 ($topk-E$)	55.0	27.6	98.6
提案手法 ($topk-U$)	63.0	27.8	96.6
提案手法 ($topk-F$)	593.1	27.4	99.9

表 8　実験 1 – 3 結果 (容量制約:10GB)

容量制約:1000GB	実行時間 (s)	合計利得 (G)	使用容量 (%)
BIGSUBS	7564	56.3	39.9
提案手法 ($topk-E$)	66.9	57.3	72.8
提案手法 ($topk-U$)	52.3	57.5	79.0
提案手法 ($topk-F$)	789.9	53.1	49.3

表 9　実験 1 – 3 結果 (容量制約:1000GB)

4.3.3　実験 1 – 3: 高スケールにおける BIGSUBS との比較

この実験は高スケールなデータベース，ワークロードにおいて，N_ILP により得られる総利得の最適値，BIGSUBS と提案手法により得られる総利得とアルゴリズム実行時間，使用容量を比較するために行った．つまり，実験 1 – 1(4.3.1 節) を高スケールで行ったものである．天文台データベースを用いる実験では N_ILP による最適解は導出していない．これは，天文台データベース，ワークロードは共に高スケールであり，N_ILP で解く場合整数計画問題で扱う変数が膨大となり，短時間で最適解を得ることはできないためである．また，高スケールな実験の場合，BIGSUBS の確率的アルゴリズムによる結果の振動が大きく現れていたために，10 回実験を行い，その平均値を提案手法と比較する．実験環境は表 2 である．提案手法と BIGSUBS の平均の比較結果を表 8，9 に示す． 高スケールな実験では，実行時間に着目すると BIGSUBS と提案手法では大きな差が存在した．提案手法の ($topk-E$, $topk-U$) においては，BIGSUBS の 1/100 程度の実行時間であった．提案手法の $topk-F$ は他の提案手法 2 に比べて，実行時間が 10 倍程度となっている．これはデータベース，ワークロードが高ス

容量制約:0.1GB	実行時間 (s)	合計利得 (M)	使用容量 (%)
近傍探索なし ($topk-E$)	0.01	25.7	99.9
近傍探索なし ($topk-U$)	0.01	19.2	99.9
近傍探索なし ($topk-F$)	0.01	11.2	99.9
提案手法 ($topk-E$)	3.12	37.1	99.9
提案手法 ($topk-U$)	2.88	28.0	99.9
提案手法 ($topk-F$)	5.39	31.9	99.9

表 10　実験 2 – 1 結果 (容量制約:0.1GB)

容量制約:10GB	実行時間 (s)	合計利得 (M)	使用容量 (%)
近傍探索なし ($topk-E$)	0.03	26.0	99.9
近傍探索なし ($topk-U$)	0.03	46.5	99.9
近傍探索なし ($topk-F$)	0.03	37.7	99.9
提案手法 ($topk-E$)	8.12	46.4	99.9
提案手法 ($topk-U$)	6.05	56.1	99.9
提案手法 ($topk-F$)	7.61	45.0	99.9

表 11　実験 2 – 1 結果 (容量制約:10GB)

容量制約:10GB	実行時間 (s)	合計利得 (G)	使用容量 (%)
近傍探索なし ($topk-E$)	10.1	26.0	100
近傍探索なし ($topk-U$)	8.1	26.0	100
近傍探索なし ($topk-F$)	54.2	22.1	100
提案手法 ($topk-E$)	55.0	27.6	98.6
提案手法 ($topk-U$)	63.0	27.8	96.6
提案手法 ($topk-F$)	593	27.4	99.9

表 12　実験 2 – 2 結果 (容量制約:10GB)

ケールな場合，頻度の高いサブクエリは多くのクエリ内で共有されていることなり，整数計画問題へと入力される変数が多くなるためだと考えられる．

また，利得に着目すると，容量制約 $10GB$ では BIGSUBS が優位であった．このことから BIGSUBS は高スケールな環境で良い性能を持つことがわかった．しかし，他の容量制約では提案手法が優位であった．高スケールな環境においても提案手法は良い性能を持つことがわかった．BIGSUBS では解が振動することや，利得，実行時間の観点から総合的に見ると，高スケールな環境においても提案手法は BIGSUBS より優位であると言える．

4.3.4　実験 2 – 1: 近傍探索の効果確認

この実験は提案手法における近傍探索の有効性を確かめるために行った．初期解のみに整数計画問題を適用した総利得と，近傍探索と整数計画問題を繰り返すことにより得られる総利得を比較する．実験環境は表 2 である．それぞれの結果を表 10, 11 に示す． 各近傍探索なし $topk$ と比較して大幅に利得を改善していることから提案手法における局所探索法の有用性が示された．初期解群に注目すると，必ずしも同じ種類の $topk$ が一番良い結果を示すとは限らないことも分かった．このため，適切な $top-k$ を選択する手法，もしくは初期解群選択手法を選択する必要がある．

4.3.5　実験 2 – 2: 高スケールにおける近傍探索の効果確認

この実験は高スケールなデータベース，ワークロードにおいて，提案手法における近傍探索の有効性を確かめるために行った．つまり，実験 2 – 1(4.3.4 節) を高スケールで行ったものである．**??**節と同様の理由でこの実験では，N_ILP による最適解は導出していない．実験環境は表 1 である．それぞれの結果をそれぞれの結果を表 12，13 に示す．

容量制約:1000GB	実行時間 (s)	合計利得 (G)	使用容量 (%)
近傍探索なし ($topk-E$)	13.3	56.6	100
近傍探索なし ($topk-U$)	12.2	56.6	100
近傍探索なし ($topk-F$)	57.6	44.2	100
提案手法 ($topk-E$)	66.9	57.3	72.8
提案手法 ($topk-U$)	52.3	57.5	79.0
提案手法 ($topk-F$)	789	53.1	49.3

表 13　実験 2 – 2 結果 (容量制約:1000GB)

	提案手法 (topk-E)	BIGSUBS
最適化時間 (s)	3.39	14.26
利得	1.17G	1.00G
実体化ビュー作成時間 (h)	13.11	3.27
実体化ビュー使用時のワークロード実行時間 (h)	7.57	8.77
削減実行時間 (h)	12.98	11.77

表 14　実験 3 – 1 結果 (容量制約:100GB)

高スケールな環境においても，近傍探索を行うことにより，性能改善がされていることを確認できた．特に，容量制約が 10 , $100GB$ の実験においては近傍探索により大きく性能改善が可能であった．しかし，容量制約が $1000GB$ の実験においては他 2 つの容量制約下の実験と比較して改善幅は少なかった．この容量制約下では近傍探索を行なった場合，指定された容量を使い切っていない．このことから，初期解を選択した時点で大量の冗長なサブクエリが選択されており，近傍探索では，初期解にはない少量のサブクエリを探索するのみで，利得の改善が他に比べて少なかったものと考えた．

4.3.6　実験 3 – 1: 実体化ビューの実際の効果確認

この実験は，高スケールなデータベースにおいて，提案手法や BIGSUBS によって選ばれた実体化ビューの効果を実際に測定するために行った．提案手法はこれまでの実験で良い結果を残す傾向にあった $topk-E$ を用いる．両方の手法における最適化時間，予測削減コストと，実体化ビュー作成時間，実体化ビュー使用時のワークロード実行時間，削減実行時間を測定した．実験環境は表 3 である．実験結果を表 14 に示す．天文台データベースを用いて，500 クエリで整数計画問題を解いた場合でも，提案手法が BIGSUBS より高い利得を得ることができた．実体化ビュー作成時間では，BIGSUBS はその容量を使い切らないという特徴から，実体化するサブクエリの数が少なく，短い実体化ビュー作成時間を達成している．BIGSUBS では実体化するサブクエリが少ないにも関わらず，高い利得を得ている．これは BIGSUBS が重複を優先して効率の良いサブクエリを選択しているためだと考えた．実体化ビュー使用時のワークロード実行時間と削減実行時間で見ると，提案手法が利得と同様に良い結果を得ている．ワークロードが複数回実行される環境を想定すれば，実体化ビュー作成時間，実体化ビュー使用時のワークロード実行時間の和を考えても提案手法が有利になるだろう．

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おり，提案手法と同様に整数計画問題を用いた手法 [2] [4] が提案されている．

BIGSUBS [2] 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することで整数計画問題で扱う変数を削減する．まず，確率的な手法を用いて，実体化するサブクエリに 1，実体化しないサブクエリに 0 のラベル付けを行う．この確率は，(1) サブクエリの処理結果を保持するために必要な容量が少ないほど，(2) サブクエリを実体化した場合のコスト削減程度が大きいほど，高くなるように定義されている．次に，クエリワークロード内のクエリが，1 のラベルが付いたどのサブクエリを利用するかを整数計画問題を利用して決定する．つまり，式 (4) を満たすように z_j を確率を用いて決定し，式 (3) の目的関数を (5) と (6) を満たすように q_i ごとにそれぞれ整数計画問題を解く．この確率的なラベル付けと整数計画問題を削減コストが向上しなくなるまで繰り返すことで最終的な実体化ビューを決定する．この手法では整数計画問題において，処理される変数の数を大幅に削減しているために整数計画問題の処理時間を低減できる．そのため，この手法は大規模なクエリワークロードに対しても有効な手法とされており，実際に論文内ではこれまでの研究では考慮されていない規模のクエリワークロードに対しても適用可能であるとされている．

Wide-deep [4] は上記の BIGSUBS 改良版とされており，実体化ビュー選択問題について二つの課題を挙げている．一つ目はあるクエリに対して実体化ビューを用いることによって得られる利得をどう評価するか，二つ目は実体化ビューとするサブクエリをどう選択するかである．一つ目の解決策としてクエリに対して実体化ビューを利用した利得を推定するためにニューラルネットワークベースの手法を提案している．具体的には，オペレーターや変数などクエリの重要な特徴を抽出し，これらの特徴を効率的に hidden representation に変換するための encoding model を設計している．二つ目は BIGSUBs [2] と同様に整数計画問題と確率的なラベル付を行うが，この方法では解の収束を保証できないとし，この問題をマルコフ決定過程を用いた深層強化学習モデルを使用して解決している．エージェントは実体化ビュー群を管理しており，全ての実体化候補の中からフリップ (0 と 1 を入れ替える) る z_j を選択しする．選択された z_j をフリップした後，y_{ij} を整数計画問題によって求める．こうすることで得られた新たな利得を reward として学習を繰り返す．

6 結　　論

提案手法はワークロード内のサブクエリの利得，容量，頻度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，最新技術である BIGSUBS と比較して，高速に，高い利得をもたらすことができた．また，実際に実体化ビューを作成することで実用した際に得られる効果を確認することができた．今後の課題として，以下の点が考えられる．

- ワークロードを拡張した場合での実体化ビュー効果
- 最適実行プラン以外の実行プラン内のサブクエリ

実験 4.3.6 で行なった実験はクエリ数が 500 とまだ拡張の余地があるものとなっている．このクエリ数に応じて，提案手法，BIGSUBS の有効性がどのように変化するか確認する必要がある．提案手法ではクエリの実行プランは QueryOptimizer から出力される最適な実行プランしか考慮していない．別の実行プランも考慮することでより高い利得が得られると考えられる．しかし，入力するサブクエリの個数が増加すると，整数計画問題に要する時間も増加する．このトレードオフも考慮した実行プランの列挙を行いたい．

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）の委託業務（JPNP16007) の結果得られたものです．

文　　献

[1] H.Lan, Z.Bao, and Y.Peng. A survey on advancing the dbms query optimizer: Cardinality estimation, cost model, and plan enumeration. In *Data Science and Engineering*, pp. 86–101, 2021.

[2] Alekh Jindal, Konstantions Karanasos, Siram Rao, and Hiren Patel. Selecting subexpressions to materialize at datacenter scale. In *PVLDB*, pp. 800–812, 2018.

[3] R.Ramakrishnan, B.Sridharan, J.R. 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. In *SIGMOD*, pp. 51–63, 2017.

[4] Haitao Yuan, Guoliang Li, Ling Feng, Ji Sun, and Yue Han. Automatic view generation with deep learning and reinforcement learning. In *ICDE*, pp. 1501–1512, 2020.

[5] Y.N.Silva, P.Larson, and J.Zhou. Exploiting commonsubexpressions for cloud query processing. In *ICDE*, pp. 1337–1348, 2012.

[6] J.Zhou, P.Larson, J.C.Freytag, and W. Lehner. Efficient exploitation of similar subexpressions for query processing. In *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] Viktor Leis, rey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? In *VLDB*, pp. 204–215, 2015.

[9] Postgresql: The world's most advanced open source database. In *https://www.postgresql.org/*.

[10] Ivm (incremental view maintenance) development for postgresql. In *https://github.com/sraoss/pgsql-ivm/*.

[11] 長田悠吾, 星合拓馬, 石井達夫, 三島健, 増永良文. Postgresql におけるビューの増分メンテナンス機能の実装とその評価. In *DEIM*, 2021.

[12] https://hsc-release.mtk.nao.ac.jp/doc/index.php/database-2/.