DEIM Forum 2022 G34-3

大規模グラフ構造データの大規模トラバースに適したデータ格納方式の検討と初期評価

小澤麻由子† 西川 記史† 渡辺 聡† 坂庭 秀紀† 茂木 和彦†

† 日立製作所研究開発グループ 〒 185–8601 東京都国分寺市東恋ヶ窪 1 丁目 280

E-mail:

†{mayuko.ozawa.dj,norifumi.nishikawa.mn,satoru.watanabe.aw,hidenori.sakaniwa.hm,kazuhiko.mogi.uv}@hitachi.com

あらまし 近年、複数組織にまたがるデータ利活用促進やコスト低減に向けた DB クラウド化が進んでいる。グラフ構造データの走査と高速な分析を両立でき、クラウド化にも適応したデータベースの格納方式はまだ確立されていない。本論文では、グラフ構造データを扱う最適な手法の提案に向けて、商用 DBMS においてローストアとカラムストアの場合について、工場データトレーサビリティでのクエリを用いて評価した。B-tree 索引+カラムストアを組み合わせた方式で性能を維持するため、B-tree に検索に使用される列を保持させ、検索時にデータページにアクセスを削減する手法を取り入れた。B-tree 索引 + ローストアと B-tree 索引 + カラムストアではクエリ応答時間はほぼ同程度になるが、カラムストアの方が 30% データ容量を削減できることが確認できた。グラフ構造データの処理には B-tree 索引とカラムストア形式を選択することが適切であることが示された。

キーワード データベース、グラフ構造データ、製造業 IoT

1 はじめに

近年、情報化社会の進展に伴い地方自治体や公共機関、医療機関、民間企業などが大量にデータを保有しつつある中、新たな施策の立案やサービス創出のためにビッグデータの利活用が進んでいる。[1] [2] 例えば、自動車の自動運転技術、産業における歩留り向上技術や故障予兆診断技術、医療では遺伝子解析や製薬開発など幅広い分野でビッグデータの活用が行われつつある。ビッグデータの利活用を推進するべく、様々な RDBMS (Relational Database Management System) がデータ活用支援サービスとして開発されている。しかし、多種多様なデータが存在するのに対して、このような特徴を持つデータにはどのような特性を持つ RDBMS が適しているか、といった対応関係の研究は発展途上である。そこで今回は、ソーシャルネットワークや交通ネットワークや生物学的ネットワーク等に見られるグラフ構造データの取扱いに長けたデータベース格納方式の提案に向けて、Hitachi Advanced Data Binder [3](以下 HADB と記す)[1]を用いローストアとカラムストアの場合についてのクエリ実行の評価を行う。

2 関連研究

実表には、ローストア表とカラムストア表があり、それぞれ表データの格納形式が異なる [4]。業務内容や表の利用方法によってこれらを使い分け、表の検索性能の向上につなげている。

2.1 ローストア

データを行単位でデータベースに格納する形式のことをローストア形式と呼ぶ。ローストア形式の場合、1 行のデータが 1 レコードとしてデータベースに格納される。ローストア表の場合、データが行単位で格納されているため、次のような検索をするときに適している。

- 行単位でデータにアクセスするような検索をする場合。例えば、選択式に*を指定した SELECT 文を実行する場合や、選択式にほぼすべての列を指定するような場合。
- 検索対象のデータを B-tree インデクスを使用して絞り込むような検索をする場合。

HADB では、表の検索が実行された場合、サーバは検索対象の行が格納されているページだけを読み込む。そのため、B-tree インデクスを使用して検索対象の行を絞り込むことができる場合は、読み込むページ数を削減することができる。

2.2 カラムストア

データを列単位でデータベースに格納する形式のことをカラムストア形式と呼ぶ。カラムストア形式の場合、表の各列のデータが列ごとにまとまってデータベースに格納される。カラムストア表の場合、データが列単位でまとまって格納されているため、次のような検索をするときに適している。

- 検索対象のデータをあまり絞り込まず、特定の列データ全体を検索する場合。
- 特定の範囲内（特定の年や月など）の、特定の列データにアクセスする場合。
- 特定の列データに対する値の集計（平均や合計などを求める）をすることが多い場合。

HADB では、表の検索が実行された場合、サーバは検索対象の

1：内閣府の最先端研究開発支援プログラム「超巨大データベース時代に向けた最高速データベースエンジンの開発と当該エンジンを核とする戦略的社会サービスの実証・評価」（中心研究者：喜連川東大特別教授／国立情報学研究所所長）の成果を利用。

列データが格納されているセグメントを最初に探す。その後、セグメント内の検索対象の列データが格納されているページを読み込む。特定の列データの集計をする場合は、集計対象の列データが格納されているページだけにアクセスすればよいため、読み込むページ数を削減することができる。 また、カラムストア形式と B-tree インデクスを適切に組み合わせたハイブリッドな設計により、処理性能が向上するベンチマークの結果がある。[5]

3 提案手法

3.1 本研究のアプローチ

本研究の狙いは、産業分野において代表的なワークロードで使われている工場トレーサビリティデータ [6] の検索性能の向上と DB 容量の削減である。工場トレーサビリティデータは工場の 4M (Man、 Machine、 Material、 Method) 管理で利用されるグラフデータベースである。グラフデータベースは、ノード、エッジ、プロパティの 3 要素を持ちデータ構造がネットワーク状になっていることが特徴で、繋がりのあるデータを効率的に検索できる。しかし、参照するノードやエッジ数が膨大になると処理速度が遅くなり、データベース容量が大きくなってしまうことが課題として挙げられてきた。本論文では、今までローストア形式で使用されることが多かった B-tree インデクスを、カラムストア形式に使用することを考える。B-tree インデクスをカラムストア形式に組み込むことで、アクセスの高速化とデータベース容量の削減の両立が見込める [7] [8]。カラムストア形式では、1 件アクセスするたびに都度データの解凍と列の復元を行う必要があるため非効率的なことが問題であった。そこで、B-tree 索引に参照列を全て保持することで、カラムストアへのアクセスを回避し性能を向上することが考えられる。しかし、B-tree 索引に参照列を全て保持すると B-tree 索引のサイズが増大し、DB 容量削減の効果が低減する。我々は、B-tree 索引に必要な参照列を適宜保持することでカラムストアへのアクセス回数を減らし、性能向上と DB 容量抑制の両立を図ることが可能になると考えた。従って、本論文ではカラムストアと B-tree 索引の併用において、B-tree 索引を定義すべき適切な列を明確化するための初期評価を行う。

3.2 工場トレーサビリティデータ

工場トレーサビリティデータは、生産工程における業務と、各業務で発生し点在する OT、IT データを、データ間の「つながり」で定義・連結するデータモデルによって End to End で業務とデータを紐付け、必要なデータの抽出や分析を容易にするデータ活用基盤である 。工場の生産管理では、生産に必要な要因系のアイテムである作業者 (Man)、設備 (Machine)、作業方法 (Method)、材料 (Material) の管理が必要である。工場トレーサビリティデータでは、工程毎の業務 (Activity) に 4M の情報が、Fig. 3 のようにグラフ構造で紐付いて管理される。グラフ構造では、ノード（頂点）群とノード間の連結関係を表すエッジ（枝）群で構成され、2 つのノード間の経路を探す操作に有利である。Fig. 4 のような複数の工程を管理する現場を考える。例えば、工程 2 で障害が発生した場合のその時刻の前後の関連する工程の設備の情報を取得しようとした際に、下記手順にて、グラフを辿ることにより容易に目的の情報に辿り着くことができる。

（1） 工程 2 の業務ノードの対応時刻の情報を取得
（2） 工程 2 の後工程をグラフ検索
（3） 工程 2 の前工程をグラフ検索
（4） 工程 1 の設備情報と工程 3 の設備情報を取得

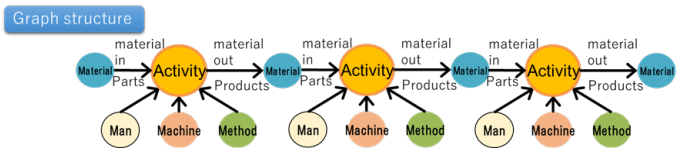

図 1 工場トレーサビリティデータのグラフ構造データ

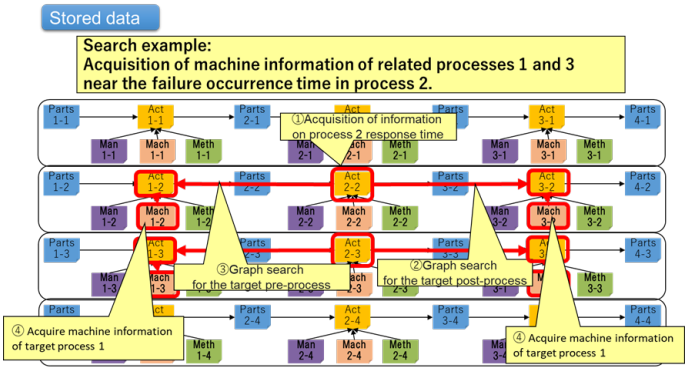

図 2 複数工程に渡るデータの検索方法

3.3 データ格納方式

HADB は、日立製 RDBMS であり、非順序型実行原理 [2] [9] [10] [11] [12] を用いて、データの要求順序とは無関係な順序に非同期的にデータを処理することにより、ハードウェアの処理性能を最大限に引き出せる特徴がある。1 つの SQL 処理を多数のスレッドに割付け並列実行できる仕組みとデータベースオブジェクトを効率良く管理するためのスキーマ表、インデックス表 (レンジ索引、B-tree 索引) を有し、論理領域と物理領域を効率良くマッピングしてハード性能を最大限に引き出す構造になっている。データフォーマットは、利用するアプリに応じてカラム形式かロー形式かを選択可能である。データが増加した際は、ノードとストレージをセットでスケールアウトするために再設計が必要になる。

4 評価方法

4.1 使用データ

本論文では 3.2 で説明した工場トレーサビリティデータの検索時の性能工場が目的であるため、評価でも工場トレーサビリティデータを使用する。

2：喜連川東大特別教授／国立情報学研究所所長・合田東大准教授が考案した原理。

4.2 クエリ

グラフ処理では、起点ノード指定条件に合致するノードを求め、前後につながったノードの終端まで辿り検索ノードに指定されたノードのインスタンス ID の一覧を求める。まず、起点ノードの情報を取得し、後工程方向に、material_in、material_out が無くなるまでエッジを辿る。Class_id が一致した業務 (Activity) のインスタンス ID を取得する。次に、前工程方向に、material_out、material_in が無くなるまで情報を辿り、Class_id が一致した Activity のインスタンス ID を取得する。このように、起点ノードの指定条件に合致する情報をノード数分繰り返す。これを SQL では (a) 起点探索として、with 句にて再帰的にエッジを辿り関連するノードを重複有で全て取得する。(b) 再帰探索として、再帰的に取得したノードから対象とする工程の情報を重複排除をして選択し抽出する（図 3）。Q1–Q4 の 4 種類のクエリパターンを用意しているが、探索する選択期間が異なる。Q1 が選択期間が最も短く、0.125 時間、Q2 は 0.25 時間、Q3 は 0.5 時間、Q4 は 1 時間となっている。アンカーメンバの選択率はこの選択期間 ×class ID の選択率 (12.5%) で算出され、Q1 で 0.000625%〜Q4 で 0.005% となる。選択期間が短い方が、テーブルのスキャンする範囲が狭まるので、高速に処理ができる可能性がある。実際の SQL としては、 WITH RECURSIVE 形式の構文での再帰クエリの記述となっている（図 4）。B-tree のようなデータ構造になっていると効率良く処理できる可能性がある。

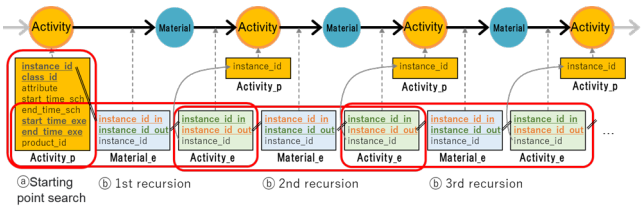

図 3　グラフ構造データの再帰的検索

```
with traversal(instance_id,class_id,instance_id_in,instance_id_out,p_instance_id) as
ⓐ (select instance_id,class_id,instance_id_in,instance_id_out,instance_id from Activity_e
     where Activity_e.edge_id=65542 and instance_id in
       (select instance_id from Activity_p where class_id ='act00036'
         and START_TIME_EXE > '2020-04-02 20:26:00' and END_TIME_EXE < '2020-04-02 20:50:00' )
   union all
ⓑ select Activity_e.instance_id,Activity_e.class_id,Activity_e.instance_id_in,Activity_e.instance_id_out,traversal.p_instance_id
    from traversal,Material_e,Activity_e
    where traversal.instance_id_out = Material_e.instance_id_in and Material_e.edge_id=65540
    and  Material_e.instance_id_out = Activity_e.instance_id_in and Activity_e.edge_id=65542)
select distinct instance_id,p_instance_id,class_id from traversal
  where traversal.class_id='act00030' or traversal.class_id='act00027' or traversal.class_id='act00085'
     or traversal.class_id='act00086' or traversal.class_id='act00024' ;
```

図 4　再帰クエリ

4.3 索引が保持する列と性能の関係

本論文では、B-tree 索引に保持する列を変えることにより検索性能やデータ容量にどのような影響が出るのかを明確化することを目的とする。B-tree 索引に保持する列を変化させるパターンの一つとして、インデクススキャンとキースキャンでの比較を行う。インデクススキャンでは、結合列のみを B-tree 索引列として定義する。キースキャンでは結合・探索条件・射影・集計に利用される全ての列を B-tree 索引列として定義する。応答時間は全ての列を B-tree 索引列として定義しているキースキャンの方が速いが、索引容量はキー長の短いインデクススキャンの方がサイズが小さくなる。従って、本論文では索引容量を小さくできるインデクススキャンを用い処理速度を向上するため、インデクススキャン・キースキャンでのクエリ実行時の挙動を観察する。

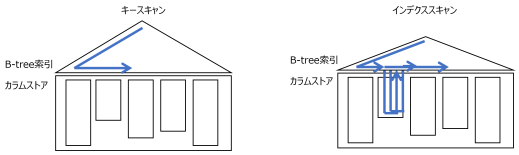

図 5　キースキャンとインデクススキャン

5 提案方式の評価

5.1 データベース容量・ロード時間

1 日分、10 日分、100 日分の 3 種類の工場トレーサビリティデータを利用した。各サイズは下記となる。

- 1 日分：CSV gz 圧縮で約 0.71GB（非圧縮約 11GB）
- 10 日分：CSV gz 圧縮で約 7.1GB（非圧縮約 110GB）
- 100 日分：CSV gz 圧縮で約 71GB（非圧縮約 1100GB）

DBMS のデータ領域は、一般的にはユーザデータ領域とシステム領域に分かれる。ユーザデータ領域には、ユーザが用意したテーブルデータが格納される。システム領域には、DBMS のディクショナリやログデータなどが格納される。ディクショナリ管理として、スキーマ、テーブルスペース、テーブル、インデックス、ビューといった DBMS 内の格納構造を参照するための情報がある。図 6 に DB サイズを示す。元の圧縮データに比べて、HADB のカラムストア形式で 24 倍程度、ローストア形式で 35 倍程度の DB 容量を管理している。このような DB 容量になった原因は、ディクショナリ管理サイズが大きいことや、B-tree 索引の圧縮が効かないことなどが挙げられる。

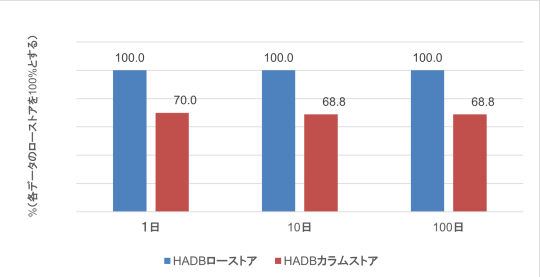

図 6　HADB のローストア/カラムストアの DB サイズ比較

図 7 にインデックス容量とテーブル容量の測定値を示す。インデックスは高速化に貢献するが、ロー形式ではテーブルサイズの 85% 程度、カラム形式では 1.5 倍近くインデックスのために容量を使っていることがわかる。コストを抑えるためには DB 容量も小さくすることも有効である。HADB ではインデックスサイズの削減化も検討が必要であると考えられる。現状の

HADB では、カラム形式の方がテーブルデータの圧縮率が高いので、全体の DB 容量は抑えられていることもわかる。クエリ処理速度でロー形式の方が処理が速くなっているが、DB 容量はカラム形式の方が圧縮が効くので、DB 容量が小さく抑えられている。従って、DB 容量の観点では、工場トレーサビリティデータを始めとするグラフ構造データの取り扱いにはカラム形式が良いと考えられる。

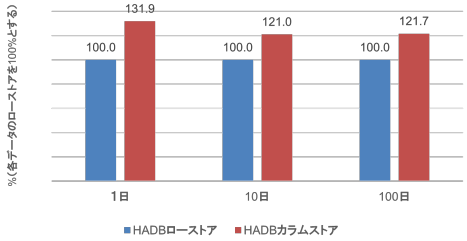

図 7　ローストア/カラムストアのロード時間の比較

5.2　工場トレーサビリティデータ性能測定

工場トレーサビリティデータを用いた HADB のローストア形式及びカラムストア形式、インデクススキャン形式及びキースキャン方式の性能評価結果（100 日分）(図 8) と表容量 (図 9) を示す。グラフの横軸は応答時間（相対値）、縦軸は形式（col-is: カラムストア、インデクススキャン、col-ks: カラムストア、キースキャン、row-is: ローストア、インデクススキャン、row-ks: ローストア、キースキャン）である。計測結果より応答時間はカラムストアとローストアでほぼ同程度になっていることが分かった。また、キースキャンの方がインデクススキャンより約 1.2 倍高速になった。索引容量はカラムストアはローストアの約半分になり、索引容量はインデクススキャンがキースキャンの約 1/4 となった。インデクススキャン用索引のサイズが小さいのはキー長が短いためである。提案手法であるカラムストアと B-tree 索引の組合せは、従来のローストアと B-tree 索引の組合せとほぼ同等の性能を維持しつつ DB 容量を削減できていることがわかった。

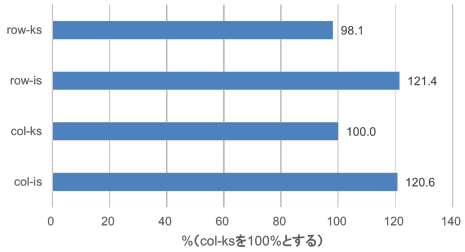

図 8　100 日分のデータでの HADB のローストア/カラムストア、インデクススキャン/キースキャンの性能評価比較

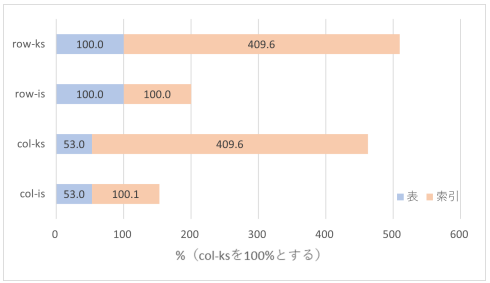

図 9　100 日分のデータでの HADB のローストア/カラムストア、インデクススキャン/キースキャンの表容量比較

また、ローストア形式及びカラムストア形式、インデクススキャン形式及びキースキャン方式のクエリ実行に要した論理 I/O 数、物理 I/O 数 (図 10、図 11) を示す。グラフの横軸は I/O 数（相対値）、縦軸は形式（col-is: カラムストア、インデクススキャン、col-ks: カラムストア、キースキャン、row-is: ローストア、インデクススキャン、row-ks: ローストア、キースキャン）である。結果より、論理 I/O 数、物理 I/O 数ともインデクススキャンの方が少なく、索引に対する I/O が大半を占めていることがわかった。また、キースキャン用索引のキー長はインデクススキャン用索引の約 4 倍である (図 12)。インデクススキャンの方がキー長が小さいため、B-tree の段数が少なくなる。従って、インデクススキャンの B-tree 部分のアクセス時間は、キースキャンの B-tree 部分のアクセス時間と同等になるか短縮できる可能性がある。よって、インデクススキャンを用いることで、キースキャンに比べて応答時間を大きく低下させることなく、DB 容量を削減できる可能性があることがわかった。また、カラムストアアクセス 1 件ごとに都度データの解凍と行の構築が必要であることが、インデクススキャンの方がキースキャンよりも遅い要因の一つの可能性がある。今後はこの推測を確認し、改善につなげていくことが課題である。

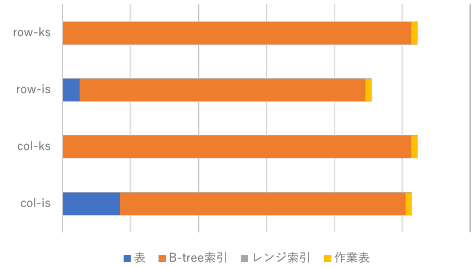

図 10　100 日分のデータでの HADB のローストア/カラムストア、インデクススキャン/キースキャンのクエリ実行に要した論理 I/O 数

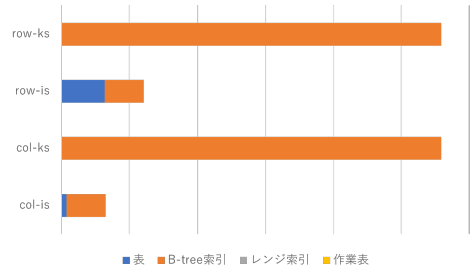

図 11　100 日分のデータでの HADB のローストア/カラムストア、インデクススキャン/キースキャンのクエリ実行に要した物理 I/O 数

■ 表定義

```
CREATE TABLE "Activity_p" (
  "instance_id"        VARCHAR(59)
 ,"class_id"           VARCHAR(26)
 ,"individual_id"      CHAR(32)
 ,"attribute"          VARCHAR(4096)
 ,"end_time_sch"       TIMESTAMP(0)
 ,"activity_class_id" VARCHAR(100)
 ,"end_time_exe"       TIMESTAMP(0)
 ,"product_id"         VARCHAR(100)
 ,"start_time_exe"     TIMESTAMP(0)
 ,"start_time_sch"     TIMESTAMP(0)
) IN ADBUTBL01 CHUNK=24 STORAGE FORMAT COLUMN;
```

■ **B-tree**索引定義（インデクススキャン）

```
CREATE INDEX "Activity_p_exe_time"
 ON "Activity_p"(
  "class_id" ASC
 ,"start_time_exe" ASC
 ,"end_time_exe" ASC
) IN ADBUIDX01 PCTFREE=0
  EMPTY INDEXTYPE BTREE;
```

キー長：8+7+7=22

■ **B-tree**索引定義（キースキャン）

```
CREATE INDEX "Activity_p_exe_time"
 ON "Activity_p"(
  "class_id" ASC
 ,"start_time_exe" ASC
 ,"end_time_exe" ASC
 ,"instance_id"    /* for key scan */
 ,"individual_id" /* for key scan */
) IN ADBUIDX01 PCTFREE=0
  EMPTY INDEXTYPE BTREE;
```

キー長：8+7+7+39+32=93

図 12　インデクススキャン用索引・キースキャン用索引の定義

6 ま と め

グラフ構造データを扱う最適な手法の提案に向けて、HADB においてローストアとカラムストアの場合について、工場データトレーサビリティでのクエリを使った評価を行った。カラムストアはローストアと比べてデータ容量を抑えることができるが、1 件アクセスするたびに都度データの解凍と列の復元の必要があるため非効率ということがネックになっていた。カラムストアへのアクセスを回避するために B-tree 索引に参照列を全て保持する方式では、性能は向上するが B-tree 索引のサイズが増大し、DB 容量削減の効果が低減する。我々は、B-tree 索引に必要な参照列を保持する方式を用いることでカラムストアへのアクセス回数を減らし、性能向上と DB 容量抑制の両立を図ることが可能になると考えた。本論文ではカラムストアと B-tree 索引の併用において、B-tree 索引を定義すべき適切な列を明確化するための初期評価を行った。B-tree 索引 + ローストアと B-tree 索引 + カラムストアではクエリ応答時間はほぼ同程度になり、カラムストアの方が 30% データ容量を削減できることが確認できた。また、カラムストア方式に結合列のみを B-tree 索引列として定義する方式を合わせることで、結合・探索条件・射影・集計に利用される全ての列を B-tree 索引列として定義する方式に比べて応答時間を大きく低下させることなく、DB 容量を削減できる可能性があることがわかった。B-tree 索引 + カラムストアの方式を使うことによって、クエリ応答時間についての性能を大きく低下させることなく、DB 容量を削減できる可能性があることが分かった。

文　献

[1] S. Klaus, “The Fourth Industrial Revolution,” 2016.

[2] 経済産業省, “通商白書 2017 概要,” https://www.meti.go.jp/report/tsuhaku2017,2017

[3] “超高速データベースエンジン Hitachi Advanced Data Binder,” https://www.hitachi.co.jp/products/it/bigdata/platform/data-binder/index.html.

[4] D. J. Abadi, S. R. Madden, N. Hachem, “Column-stores vs. row-stores: how different are they really?” 2008.

[5] A. Dziedzic, J. Wang, S. Das, B. Ding, V. R. Narasayya “Columnstore and B+ tree – Are Hybrid Physical Designs Important?” 2018.

[6] “生産現場を改善し続けるデジタルツイン技術：工場 IoT プラットフォームの構築へ向けて,” https://www.hitachihyoron.com/jp/archive/2020s/2020/03/03a05/index.html.

[7] A. Halverson, J. L. Beckmann, J. F. Naughton, D. J. Dewitt, “A Comparison of C-Store and Row-Store in a Common Framework” 2006.

[8] G. Graefe, “Efficient columnar storage in B-trees” 2007.

[9] M. Kitsuregawa, K. Goda “アウトオブオーダ型データベースエンジン OoODE の構想と初期実験,” 日本データベース学会論文誌, Vol.8, No. 1, pp.131–136, 2009.

[10] K. Goda, M.Toyoda, M. Kitsuregawa, “アウトオブオーダ型データベースエンジン OoODE の試作実装と小規模実験環境におけるソフトウェア実行挙動の観測,” 日本データベース学会論文誌, Vol. 12, No. 1, pp.25–30, 2013.

[11] A. Shimizu, K. Mogi, K. Goda, M. Kitsuregawa, “非順序型実行原理に基づく超高速データベースエンジンの詳細分析処理における性能評価” 日立評論イノベイティブ R&D レポート, pp.83–89, 2014.

[12] M. Kitsuregawa, “Out-of-order Execution of Query Processing and New Advances in COVID-19 Information,” VLDB2020: Keynote3.