社団法人 情報処理学会　研究報告
IPSJ SIG Technical Report

2004－DBS－134　(II)　(60)
2004／7／14

関係データベースシステムにおける自己再編成に関する一考察

星野 喬 †　　合田 和生 †　　喜連川 優 ‡

† 東京大学大学院 情報理工学系研究科 〒 113-0033 東京都文京区本郷 7-3-1
‡ 東京大学 生産技術研究所 〒 153-8505 東京都目黒区駒場 4-6-1

要 旨

本研究は、関係データベースシステムにおける再編成を自立化することを目的とする。再編成は、データベース管理者の主要な管理タスクの一つであり、自己再編成機構の実現は管理コスト削減に大きく寄与する。再編成は、性能劣化の要因となる表空間内のデータ配置の乱れを正すことにより性能を改善する。現状では表空間内のデータ配置の乱れを性能劣化の原因として明確に表現する指標がないため、自動的に再編成を実行することが難しい。本稿では、表空間内のデータ配置と性能の関係を直接的に視覚化するツールを開発し、2 つのアプリケーションを例にその有効性を示す。

A Study on Self-Reorganization for Relational Database Systems

Takashi Hoshino†　　Kazuo Goda†　　Masaru Kitsuregawa‡

†Graduate School of Information Science and Technology, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, 113-0033, Japan
‡Institute of Industrial Science, University of Tokyo
4-6-1 Komaba, Meguro-ku, Tokyo, 153-8505, Japan

Abstract

This research targets self-reorganization for DBMS. Reorganization is one of the major database management tasks. So self-reorganization contributes to cost reduction of database management. Disorder of data in tablespaces brings on considerable performance degradation, and reorganization replaces data in an orderly state in order to improve performance. Currently we have few explicit indicators which discribe factors of performance degradation. To realize self-reorganization, we need clear causal relationships between performance degradation and its factors. In this paper, we develop a tool that shows visually the disorder of data in tablespaces, and consider reasons of performance degradation with two application examples.

1　はじめに

近年、人類の扱うデジタルデータの容量は飛躍的に増大しており、データ管理の基盤である DBMS は益々重要になっている。一方で、その管理はより困難化しており、コスト増大が問題となっている。この問題を解決するため、DBMS を自立化させる技術が求められている。

DBMS の自立化を目指した研究は多くなされている。例えば、Self-* [1] において、多くの brick-based ストレージ を統合し、階層構造によりストレージから集約される管理情報まとめて扱える手法

の提案を行っている。このような DBMS の下位層での管理問題も重要である。また、Polus [5] において、QoS 要求を具体的な低レベルのポリシーに直すタスクを、知識と推論エンジンを使って自動的に変換する機構の提案を行っている。その他、[2] でこれからの DBMS の自立化に向けた取り組みが数多く紹介されている。

現在の DBMS で実際に動作している自立管理機構もいくつかある。オンライン表空間拡張機能はその一つであり、表空間の容量が足りなくなってきた時点で、予備領域の追加割り当てを自動的に行う機構である。また、自動バックアップも、あからじめバックアップスケジュールを与えておけば、それに従ってテープドライブに自動的にバックアップするという自立管理機構である。このように単純な機構は動作しているが、データベース管理者に高度な判断を要する管理タスクを実際に自立化する試みは、未だ実現していない自立化が難しいタスクには、データベース構成変更、性能チューニング、再編成などが挙げられる。

この中で、再編成に着目した自律化への取り組みは現時点で他に見当たらず、我々は再編成を自立化することを目標として研究に取り組んでいる。再編成は、DBMS における主要な管理業務の一つであり、性能劣化を防ぐ重要な判断を伴う。

二次記憶空間上のデータは、更新操作を繰り返すことによりその配置が乱れ、本来想定している配置と大きく乖離した状態となり、性能を大幅に劣化させる場合がある。このため、DBMS の管理者は再編成を度々行うことにより、記憶空間上のデータを再配置し、性能の劣化を予め防ぐ必要がある。現状では、性能劣化を予測し再編成を開始することの判断は難しく、また、再編成自体にかかる時間は膨大であることから、再編成は高度の管理業務と考えられており、再編成を自立化させることの意義は大きい。

DBMS における再編成は、データベース管理者が DBMS から得られる性能などの統計情報や、空間情報などからそのデータベースに再編成が必要であることを判断し、再編成を実行する。これらの情報を判断する方法は、管理者の勘と経験によるノウハウに多くを依存しており、具体的にマニュアル化されたものではない。

現状のデータベース再編成判断がこのように曖昧なものになっている主要因は、性能劣化とデータ配置の乱れの関係を記述する方法論が曖昧であること

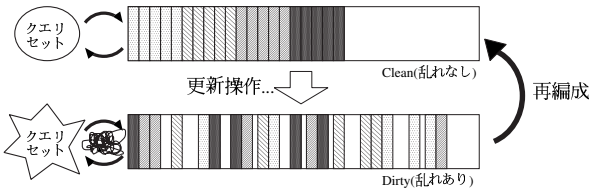


図 1: 再編成の概念図

である。再編成を起動するタイミングを自動化し、また、再編成後のデータ配置を効率化するためには、性能劣化の直接的な原因となるデータ配置の乱れを定量的な指標によって表現する必要がある。これにより再編成タスクによって能動的に性能改善を図ることが出来るようになる。将来的には、IT システム全体の自立管理へ統合可能な技術とすることを視野に入れたシステムにすることも可能である。

我々は、性能劣化とデータ配置の関係を視覚的に表現出来るツール Dirtiness Visualizer for DBMS を開発し、実験を行い、データベース更新操作による性能劣化の原因を考察した。

本稿は以下のように構成される。まず第 2 章で関係データベースにおける再編成について説明する。次に、第 3 章で我々の提案する自立再編成機構の枠組について説明する。その後、第 4 章で我々の開発したデータ配置可視化ツールを紹介し、第 5 章にてデータ配置による性能劣化を実験によって示す。最後に第 6 章においてまとめと今後の展望を述べる。

2 関係データベースにおける再編成

図 1 に、DBMS における再編成の概要を図示した。

DBMS を運用するにあたり、データを表空間にロードした直後は、データが整然と並んでいる。その後、運用によってデータベースの更新が繰り返され、徐々にデータが期待した通りに並ばなくなってくる。そのような状態になると、ロード直後と同様のデータセット、クエリセットにおいても、性能が悪化する現象が発生する。これをそのままにしておくと、要求される性能を下回るほど性能が悪化してしまう。このような状態を我々は、データベースが「汚い」と呼ぶことにする。データベースの汚さは、許容できない性能劣化を引き起こす。データベース

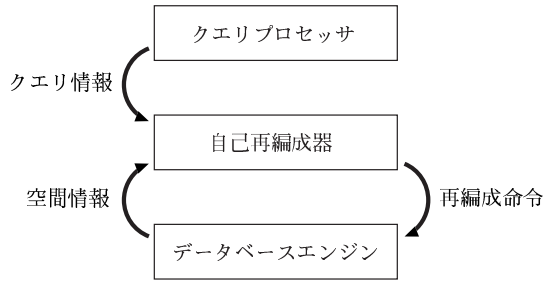


図 2: 自己再編成の枠組

が汚くなる前に再編成が必要になる。データの再配置を行い、データ配置の乱れによる性能劣化を最小限にすることが望まれる。

の場合簡単なものである。データをロードし直して、データが整然と並ぶ配置にする。ロード直後の状態は、性能要求を満たすように設計されているはずで、性能劣化の一要因である断片化やゴミブロックなども存在しない。よってほとんどの場合、この方法で想定外の性能劣化は防ぐことが出来る。

これまでデータベースの汚さを観測し、性能劣化を予測するのはデータベース管理者の仕事だった。近年、データベースが巨大化、高機能化し、また高い可用性を求められるようになったため、管理タスクとしての再編成がより高度で複雑な判断を必要とするようになり、自己再編成が必要になってきた。

自己再編成を実現するために、データ配置の乱れによる性能劣化を実用的な精度で予測すること、つまりデータベースの汚さを定量的に観測することが必要になる。現状でデータベース管理者の判断に使われている指標は、ページやセグメント単位での充填率、ゴミブロックの量、断片化の度合い、木構造の偏り具合などであるが、それらのパラメータの具体的な評価と再編成実行の判断は曖昧で、管理者の勘と経験に委ねられることが多い。

3 自己再編成の枠組

図 2 に自己再編成の枠組を示した。データベースの汚さは、データ配置とアクセスパターンに依存すると考えられる。アクセスパターンは、DBMS の上位のアプリケーション側から見ると、クエリ列である。上位層からは、クエリ情報を、下位層からは、データ配置の空間情報を得ることで、性能劣化の予測をし、再編成のスケジューリングを自動化することが出来る。

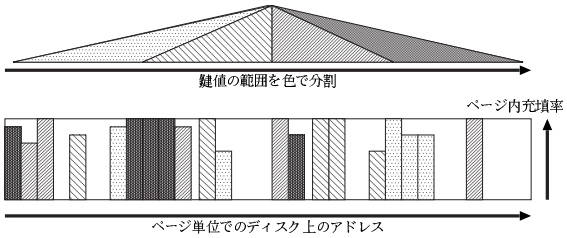


図 3: Dirtiness Visualizer の結果の表現手法

そのためには、どのようなデータ配置とアクセスパターンが性能劣化を引き起こすのか、つまりデータベースの汚さを表現する方法論を確立する必要がある。

経験則だけでは、データベースの汚さを表現するのは難しいと思われるため、直接的な因果関係となる現象を発見し、制御機構の判断基準とすることが、望ましい。

4 データ配置可視化ツール Dirtiness Visualizer

DBMS におけるデータベースの汚さを観測するために、我々は、表空間内でのデータ配置の分布を視覚的に表現するツール Dirtiness Visualizer for DBMS を開発した。ある時点でのデータベースの表空間を走査し、データ分布を解析して表示する。

結果の表示方法は、図 3 で示した。三角形の図形は鍵空間を表現しており、任意の鍵においてデータ分布を色分け表示することが出来る。色分けは、鍵の範囲を分割することによって表現している。四角形の図は表空間を表し、四角形の枠内の縦の線は、ページを表している。線の色は、そのページの中でその色の表す鍵範囲に存在する行が一番多く格納されていることを示している。

クラスタ表においては、ページ内の行は主鍵が出来るだけ近いもので構成されているため、主鍵について色分けした場合、ページ内を占める全ての行が同色であるページがほとんどであると考えることが出来る。

ページを表す縦の線の高さは、ページ内充填率を表わしている。ページの並びは二次記憶装置のアドレス (LBA) と対応しており、この並びの順番に表がアクセスされた場合、実際にディスク上でシーケ

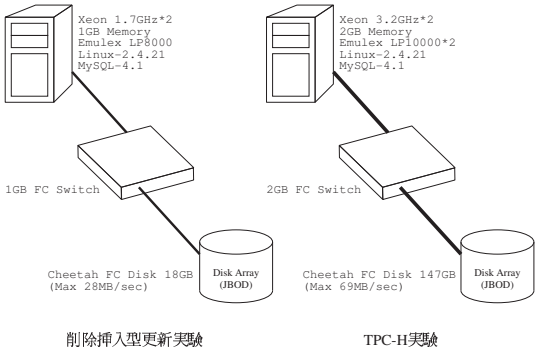


図 4: 実験環境

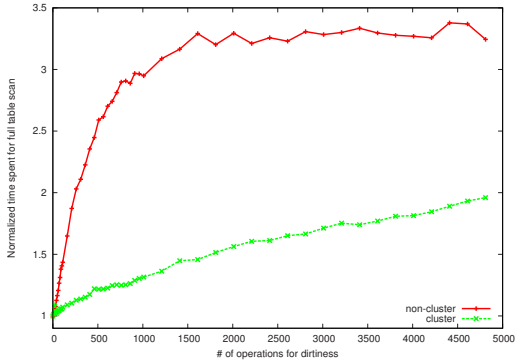
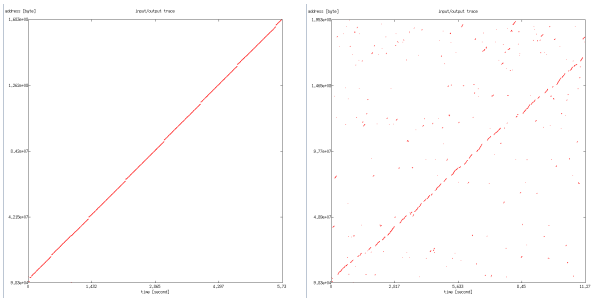


図 5: 削除挿入型更新による全表検索の性能劣化

ンシャルアクセスが発生する。

このツールは、データ配置、アクセスパターンによるデータベースの汚さを定量的に表現できる方法を確立するための評価手段として使う。また、現状での管理者による再編成判断を手助けするためのモニタリングツールとしても有用である。



a. 初期状態　　b. 最終状態

図 6: 削除挿入型更新における全表検索時の I/O(クラスタ表)

5 実験

DBMS においてデータ配置とアクセスパターンによるクエリ性能劣化を観測する実験を行った。

5.1 実験環境

図 4 に、実験環境を示した。MySQL 4.1 [3] を使い、それぞれディスク 1 台を表空間として使用した。 MySQL からのアクセスは raw デバイス経由で行なっている。

データセット、更新パターン、そしてクエリについて、2 種類のパターンについて実験を行なった。以下にその詳細を述べる。

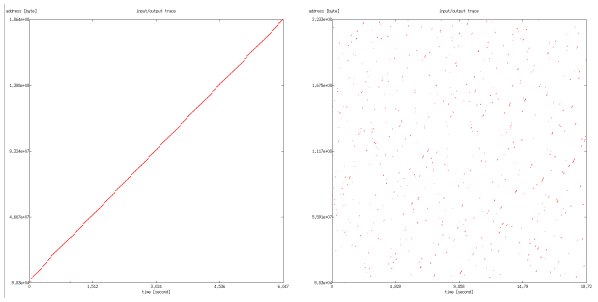
5.2 削除挿入型更新＆全表検索

単純な固定長の表 (id1 int, id2 int, txt char[255]) に [0, 500000) の主鍵 id1 を持つ 500000 行のデータセットを用意した。MySQL のページサイズはデフォルトの 16KB とし、表空間は 250MB 確保した。実験はそれぞれクラスタ表と非クラスタ表にて行った。

データセットを表空間にロードした後、以下の更新操作を約 5000 回行った。主鍵における連続行 (5000 行以下で、サイズと位置はランダム) を一度に削除し、その後全く同じ連続行を挿入する。この更新操作によりデータベースとしての状態は変化しないが、削除と挿入の実装の違いにより、表空間内のデータ配置は変化する。この実験では、データセットをロードした時に、50MB 程度の空きが出来るように表空間を定めており、更新操作が繰り返されると、最初に残っている空き領域だけでは吸収できない量の行が挿入されるようになっている。すると行挿入において更新の課程で生じた空き領域を再利用することが必要になり、表空間内のデータ配置が乱れるようになっている。

削除挿入型更新が繰り返されるデータベースに対し、全表検索クエリの性能、そのときの I/O、そして Dirtiness Visualizer の表示結果を以下に示す。

全表検索クエリの性能変化を図 5 に示した。クラスタ表、非クラスタ表ともに更新が繰り返されると性能が劣化した。性能劣化は非クラスタ表の方が早い。図 6 と図 7 に全表検索時の I/O を示した。横軸が時間で、縦軸がディスクの LBA を指している。斜めの直線状のアクセスが、シーケンシャルアクセ



a. 初期状態　　b. 最終状態

図 7: 削除挿入型更新における全表検索時の I/O(非クラスタ表)

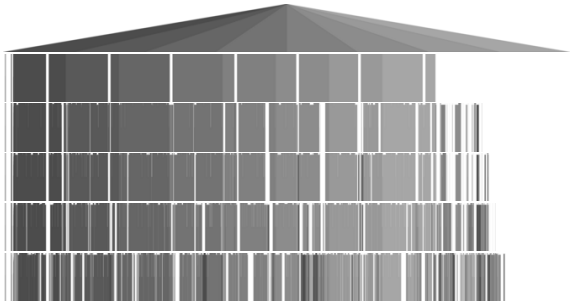


図 8: 削除挿入型更新によるデータ配置の変化 (クラスタ表) 上から順に約 0、500、1000、2000、5000 回 更新操作をした後のデータ配置

スとして、点が散在しているところはランダムアクセスとして表示されている。これを見ると、クラスタ表の方はシーケンシャルアクセスが残っているのに対し、非クラスタ表はランダムアクセスになっているのが分かる。この違いが性能劣化の違いを生んでいると結論づけることが出来る。

また、更新とともに、Dirtiness Visualizer を使い、主鍵によるデータ配置の分布を図 8 と図 9 に示した。クラスタ表は主鍵による分布が乱れるのに多くの更新を要するが、非クラスタ表は早い段階で、主キーによる分布が大きく乱れていることが分かる。クラスタ表と非クラスタ表では、行挿入の戦略に違いがあり、主鍵が近い行の近くに挿入するクラスタ表は、データ配置が乱れにくかったと考えられる。非クラスタ表においては、表空間内の後ろの空き領域を初期の挿入時に積極的に利用し、その後は、以前の削除によって生じた空き空間に行を挿入したと考えられ、データ配置が急速に乱れる様が、Dirtiness Visualizer によって示されている。

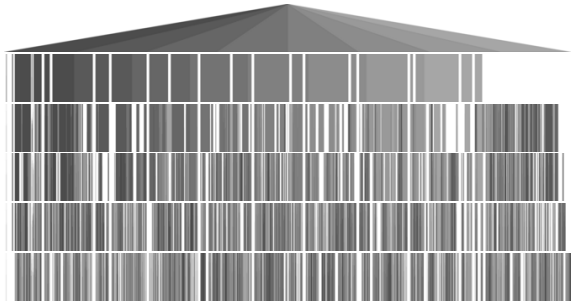


図 9: 削除挿入型更新によるデータ配置の変化 (非クラスタ表) 上から順に約 0、100、200、500、5000 回 更新操作をした後のデータ配置

5.3 TPC-H ベンチマーク

より一般的な更新操作、クエリセットである TPC-H ベンチマーク [4] を使用し、実験を行った。

スケール 0.1 (データセットは約 100MB) にて、表空間は 4GB 確保し、今回は行挿入時、新たな空き空間を確保するときに表空間の後ろの空間が自由に使える状況にした。その他のパラメータは削除挿入型更新実験と同様である。

更新に TPC-H ベンチマークの リフレッシュクエリ RF1(orders と lineitem のバルク挿入)、RF2(orders と lineitem のバルク削除) を 1 回ずつ実行することで 1 回のデータベース更新とした。このリフレッシュクエリは、100 回の更新で、orders 表と lineitem 表の全ての行が入れ替わり、400 回の更新で、論理的には初期状態と全く同じ状態に戻るようになっている。

性能測定には、Q1、Q2、Q4、Q6、Q10、Q101(orders 表の全表検索)、Q102(lineitem 表の全表検索) を使用した。

削除挿入型更新のときと同様、クラスタ表と非クラスタ表に対して実験を行なった。

図 10 と図 11 に性能劣化を示した。orders 表と lineitem 表しか更新しないので、この 2 つ以外のみ読むクエリは性能が変化しない。性能が劣化するクエリについて、クラスタ表の方が、非クラスタ表よりも性能劣化が早いことが、削除挿入型更新の実験とは逆である。また、クエリによってふるまいに違いが見られる。Lineitem 表の全表検索 (Q102) においては、クラスタ表では性能劣化が一番大きいのに対し、非クラスタ表では逆に性能改善している傾向が見られる。

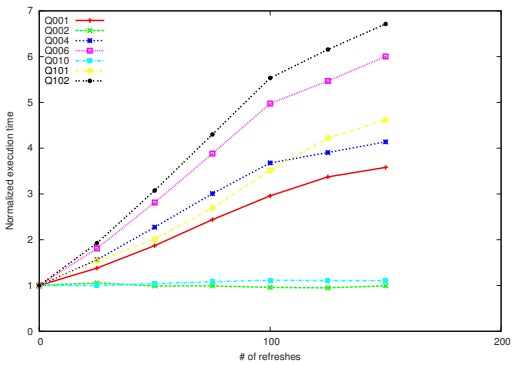


図 10: TPC-H のリフレッシュクエリによるベンチマークの性能劣化 (クラスタ表)

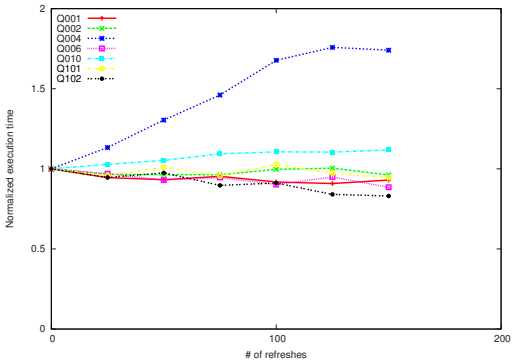
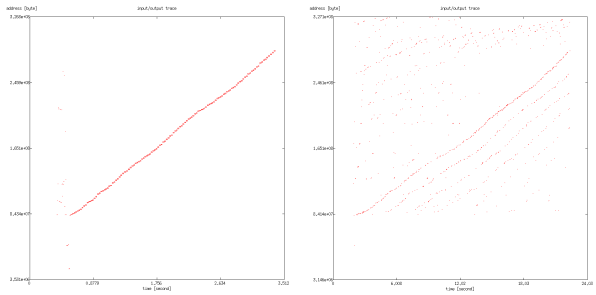


図 11: TPC-H のリフレッシュクエリによるベンチマークの性能劣化 (非クラスタ表)

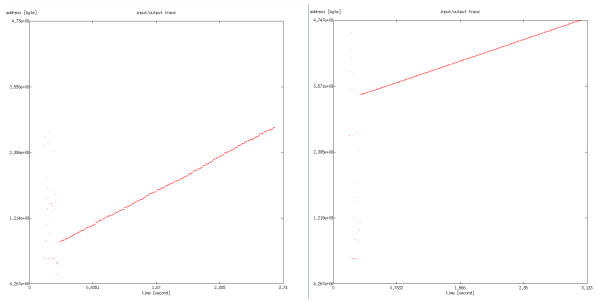
以後の結果は、表空間の多くを占める lineitem 表に注目して表示した。lineitem 表の全表検索時の I/O を図 12 と図 13 に示した。これによると、非クラスタ表の方がクラスタ表に比べてシーケンシャルアクセスが残っていることが分かり、性能劣化が非クラスタ表より速いクラスタ表という結果と一致する。

図 14 と図 15 に示した、lineitem 表の主鍵による色分け分布を見ると、非クラスタ表については、実験の最終状態において、初期状態よりも整然とデータが並んでいることが分かり、lineitem 表の全表検索に関してむしろ性能が良くなっている結果が説明できる。これは、TPC-H のリフレッシュクエリがベンチマークであるが故に、更新に規則性があり、鍵の順番に削除、挿入を繰り返していることが原因であると考えられる。実際のデータベース運用において、ここまで規則的なデータ配置変更が起き、性能劣化しないとは考えにくい。実運用においてどのようなクエリセットが使われるのかを考察することが必要であろう。



a. 初期状態　　b. 最終状態

図 12: TPC-H における lineitem 表の全表検索時の I/O(クラスタ表)



a. 初期状態　　b. 最終状態

図 13: TPC-H における lineitem 表の全表検索時の I/O(非クラスタ表)

6 おわりに

我々は、自己再編成を実現するために、データベースの汚さを定量的に示す方法を確立しようとしている。そのために、二次記憶上でのデータ配置の分布を視覚的に表示するツールを開発した。その後、更新操作により性能劣化を起こす 2 つの実験を行い、表空間内のデータ配置が乱れる様を表現し、性能劣化との因果関係について考察した。

今後は、より多くのケーススタディについて実験することにより、データを集め、多くの事例を網羅するデータベースの汚さについて考察したいと考えている。また、表空間内のデータ配置だけでなく、クエリプランやその他の指標についても考察し、より包括的な自己再編成の枠組を確立していきたい。

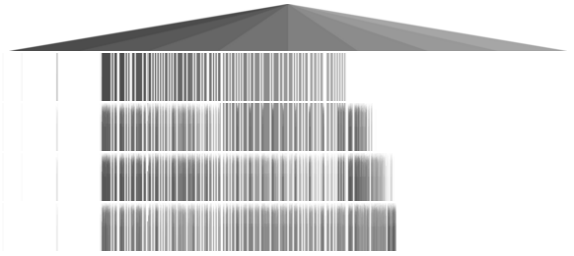


図 14: TPC-H における lineitem 表データ配置の変化 (クラスタ表) 上から順に約 0、50、100、150 回更新操作をした後のデータ配置

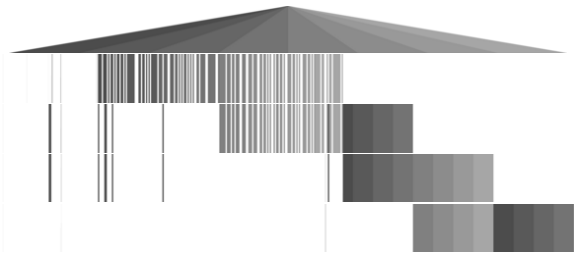


図 15: TPC-H における lineitem 表データ配置の変化 (非クラスタ表) 上から順に約 0、50、100、150 回 更新操作をした後のデータ配置

謝辞

本研究の一部は、文部科学省リーディングプロジェクト e-society 基盤ソフトウェアの総合開発「先進的なストレージ技術」の助成により行われた。協力企業である日立製作所より多くの有益なコメントを頂戴した。深謝する次第である。

参考文献

[1] Andrew J. Klosterman Gregory R. Ganger, John D. Strunk. Self-* Storage: Brick-based Storage with Automated Administration. Technical Report CMU-CS-03-178, Carnegie Mellon University, August 2003.

[2] Sam Lightstone, Berni Schiefer, Danny Zilio, and Jim Kleewein. Autonomic Computing for Relational Databases: The Ten Year Vision. In *Proceedings of Workshop on Autonomic Computing Principles and Architectures (AUCOPA2003)*, August 2003.

[3] MySQL: The World's Most Popular Open Source Database. http://www.mysql.com/.

[4] TPC: Transaction Processing Performance Council . http://www.tpc.org/.

[5] Sandeep Uttamchandani, Kaladhar Voruganti, Sudarshan Srinivasan, John Palmer, and David Pease. Polus: Growing Storage QoS Management Beyond a "Four-year Old Kid". In *Proceedings of FAST2004*, April 2004.