Adaptive Adaptive Indexing

Felix Martin Schuhknecht
Jens Dittrich
Laurent Linden

Big Data Analytics Group
bigdata.uni-saarland.de

Saarland University
Database Cracking

Stratis K. Iserman
The Netherlands
Stratis.iserman@tue.nl

Martin L. Kersten
The Netherlands
Martin.Kersten@tue.nl

Stefan Moeijendijk
The Netherlands
Stefan.moeijendijk@tue.nl

ABSTRACT

Database crackers provide a non-invasive means to tamper with database contents. They may originate from various motivations, such as financial gain or curiosity. The techniques used by crackers are similar to those used by database administrators for database maintenance. However, crackers often perform their attacks without authorization. One of the main differences between the two is the use of more powerful crackers, such as computerized networks. The crackers are more likely to perform these attacks in an automated fashion, using brute-force methods. Due to the increasing number of crackers, crackers are more likely to use automated attack strategies. Moreover, crackers often perform their attacks in an automated fashion, using brute-force methods.

1. INTRODUCTION

In this paper, we explore a method that allows crackers to tamper with database contents in an automated fashion. The method is based on the assumption that crackers can be classified into two categories: crackers who perform their attacks in an automated fashion and crackers who perform their attacks using brute-force methods. We rely on the assumption that crackers can be classified into two categories: crackers who perform their attacks in an automated fashion and crackers who perform their attacks using brute-force methods. We rely on the assumption that crackers can be classified into two categories: crackers who perform their attacks in an automated fashion and crackers who perform their attacks using brute-force methods. We rely on the assumption that crackers can be classified into two categories: crackers who perform their attacks in an automated fashion and crackers who perform their attacks using brute-force methods.
Database Cracking / Standard Cracking

Table Column

Index Column

Index Column

Index Column

sorted

?  

Q₀=[13,42)

Q₁=[6,27)

Q₂ … Qₙ

Q₀=[13,42)  

Q₁=[6,27)  

Q₂ … Qₙ

< 13

>= 13

< 42

>= 42

< 6

>= 6

< 13

>= 13

< 27

>=27

< 42

>= 42
Problems?

High Variance

Low Convergence Speed

Low Robustness

[Felix Schuhknecht, Alekh Jindal, Jens Dittrich: The Uncracked Pieces in Database Cracking, PVLDB Vol. 7, No. 2, Best Paper Award]
All-in-one?

An Adaptive Adaptive Index?
Design rules:

1. Generalize way of refinement
2. Adapt refinement effort
3. Awareness of key distributions
1. Generalize way of refinement:

\[ \text{partition-in-}k \]

\[ Q_0 \]

\[ Q_{i, i>0} \]

Base Table

- 36
- 13
- 67
- 42
- 99
- 78
- 18
- 85
- 55
- 5
- 47

Index Column (out-of-place)

- 13
- 18
- 5
- 36
- 42
- 28
- 47

Index Column (in-place)

- 13
- 18
- 5
- 28
- 36
- 42
- 47
- 67
- 55
- 99
- 78
- 85

8/20
1. Generalize way of refinement:
1. Generalize way of refinement:
2. Adapt refinement effort

\[ f(s, q) = \begin{cases} 
  b_{\text{first}} & \text{if } q = 0 \\
  b_{\text{min}} + \left( (b_{\text{max}} - b_{\text{min}}) \cdot \left(1 - \frac{s}{t_{\text{adapt}}} \right) \right) & \text{else if } s > t_{\text{adapt}} \\
  b_{\text{sort}} & \text{else if } s > t_{\text{sort}} \\
  b_{\text{min}} & \text{else.} 
\end{cases} \]

Partitioning Fanout

Out-of-place crack-in-two + In-place crack-in-two
Out-of-place radix partitioning

Input data size

32KB (L1) 256KB (L2) 2MB (Page) 10MB (L3)

Partitioning Fanout

Runtime in [s]

Partitioning Fanout

Runtime in [ms]
2. Adapt refinement effort

\[
f(s, q) = \begin{cases} 
  b_{\text{first}} & \text{if } q = 0 \\
  b_{\min} & \text{else if } s > t_{\text{adapt}} \\
  b_{\min} + \left[ (b_{\max} - b_{\min}) \cdot \left(1 - \frac{s}{t_{\text{adapt}}} \right) \right] & \text{else if } s > t_{\text{sort}} \\
  b_{\text{sort}} & \text{else.}
\end{cases}
\]
3. Awareness of key distributions: skew?

1. Histogram

Input

<table>
<thead>
<tr>
<th>bfirst</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bmin</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

out-of-place
on

<table>
<thead>
<tr>
<th>Histogram on 00</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
</tr>
<tr>
<td>0001</td>
</tr>
<tr>
<td>0010</td>
</tr>
<tr>
<td>0011</td>
</tr>
<tr>
<td>0100</td>
</tr>
<tr>
<td>0101</td>
</tr>
<tr>
<td>0110</td>
</tr>
<tr>
<td>0111</td>
</tr>
</tbody>
</table>

in-place
on

<table>
<thead>
<tr>
<th>bmin = 4 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
</tr>
<tr>
<td>0001</td>
</tr>
<tr>
<td>0010</td>
</tr>
<tr>
<td>0011</td>
</tr>
<tr>
<td>0100</td>
</tr>
<tr>
<td>0101</td>
</tr>
<tr>
<td>0110</td>
</tr>
<tr>
<td>0111</td>
</tr>
</tbody>
</table>

Index Column

<table>
<thead>
<tr>
<th>Index Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
</tr>
<tr>
<td>0001</td>
</tr>
<tr>
<td>0010</td>
</tr>
<tr>
<td>0011</td>
</tr>
<tr>
<td>0100</td>
</tr>
<tr>
<td>0101</td>
</tr>
<tr>
<td>0110</td>
</tr>
<tr>
<td>0111</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
</tr>
<tr>
<td>0001</td>
</tr>
<tr>
<td>0010</td>
</tr>
<tr>
<td>0011</td>
</tr>
<tr>
<td>0100</td>
</tr>
<tr>
<td>0101</td>
</tr>
<tr>
<td>0110</td>
</tr>
<tr>
<td>0111</td>
</tr>
</tbody>
</table>
Putting it all together

\[ Q_0 = [35, 185] \]
\[ Q_1 = [38, 149] \]
\[ Q_2 = [48, 109] \]
\[ Q_3 = [36, 49] \]
Emulation

[Felix Martin Schuhknecht, Alekh Jindal, Jens Dittrich: The Uncracked Pieces in Database Cracking, PVLDB Vol. 7, No. 2]
Test Setup

**Test Setup**

**Frequency**

UNIFORM $[0, 2^{64})$

NORMAL ($\mu = 2^{63}, \sigma = 2^{61}$)

ZIPF $[0, 2^{64}), \alpha = 0.6$

**Key range**

**Query Sequence**

ZOOMOUTALT  ZOOMINALT

[**Felix Halim, Stratos Idreos, Panagiotis Karras, Roland H. C. Yap:**
Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores, PVLDB Vol. 5, No. 6]
Individual Query Response Times

Meta-adaptive Index (Manually configured)

DC  DD1R  HCS  Sort + Binary Search

Adaptive Adaptive Index (Manually configured)

UNIFORM [0, 2^64]

DC  DD1R  HCS  Sort + Binary Search

UNIFORM [0, 2^64]

DC  DD1R  HCS  Sort + Binary Search

UNIFORM [0, 2^64]

Random

b_{first} = 10
b_{min} = 3
b_{max} = 6
t_{adapt} = 64\,MB
t_{sort} = 256\,KB
Individual Query Response Times

DC ● DD1R ● HCS ● Sort + Binary Search ● Adaptive Adaptive Index (Manually configured)

Key Range

Query Sequence

Individually Query Response Times

b_{first}=10
b_{min}=3
b_{max}=6
t_{adapt}=64MB
t_{sort}=256KB
Accumulated Query Response Times

![Bar chart showing accumulated query response times for different query workloads and query sequence configurations. The chart compares the performance of DC, DD1R, HCS, and adaptive adaptive indexes (Manually configured and Simulated annealing configured). The x-axis represents different query sequences: RANDOM, SKEW, PERIODIC, SEQUENTIAL, ZOOMOUTALT, ZOOMINALT. The y-axis represents accumulated query response time in seconds. The chart also includes key range and query sequence visualizations.]

Frequency
UNIFORM [0,2^{64}]

Key range

b_{first} = 10
b_{min} = 3
b_{max} = 6
t_{adapt} = 64\,MB
t_{sort} = 256\,KB
Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores

Falko Heiden
Stefan Kramer
Ranjidhara Amarasinghe
Robert H. C. Ng
Nagariah W. S. W. E. Wijesinghe
kramer@cs.nott.ac.uk
amarasinghe@cs.nott.ac.uk
ng@cs.nott.ac.uk
wijesinghe@cs.nott.ac.uk

ABSTRACT

Database applications and information retrieval systems rely on high performance and data organization. Such environments are characterized by high data rates and access patterns that are highly skewed, with a significant number of reads but few writes. This poses a challenge to database systems, which are typically optimized for inserts and updates. A common approach to address this issue is to use adaptive indexing, which dynamically adjusts the index structure based on the access patterns. However, this approach requires a constantly changing index structure, which can lead to performance degradation.

This paper presents a novel approach to adaptive indexing that addresses this issue. Our approach, called Stochastic Database Cracking, uses a stochastic model to predict the access patterns and adjusts the index structure accordingly. We evaluate our approach on a real-world data set and show that it significantly improves performance.

1. INTRODUCTION

2. Merging What’s Cracked, Cracking What’s Merged: Adaptive Indexing in Main-Memory C-Sorts

Stefan Kramer
Harsh P. Kulkarni
kramer@cs.nott.ac.uk
HarshP@cs.nott.ac.uk

ABSTRACT

As an increasing amount of data is stored in main-memory storage systems, the efficient management of this data becomes increasingly important. One of the key aspects of managing data in main-memory is the use of indexes. Indexes are used to speed up data retrieval operations, such as queries and updates. However, traditional indexes, such as B-trees, can become inefficient when used in main-memory environments.

In this paper, we propose a novel approach to indexing in main-memory systems, called Adaptive Indexing. Our approach dynamically adjusts the index structure based on the access patterns, which allows it to adapt to the changing data workload. We evaluate our approach on a real-world data set and show that it significantly improves performance.

1. INTRODUCTION

2. Merging What’s Cracked, Cracking What’s Merged: Adaptive Indexing in Main-Memory C-Sorts

Strategic Tuple Reconstruction in Column-Stores

Stefan Kramer
Martin L. Graefe
Stefan Manegold
kramer@cs.nott.ac.uk
graefe@cs.nott.ac.uk
manegold@cs.nott.ac.uk

ABSTRACT

In column-oriented database systems, data is stored in separate columns, each containing a set of related values for a specific attribute. This allows for efficient data storage and retrieval, but also introduces challenges in terms of data management and query processing. One such challenge is the need for efficient tuple reconstruction, i.e., the process of generating a complete tuple from its constituent columns.

In this paper, we propose a novel approach to tuple reconstruction that leverages the column-oriented storage model. Our approach, called Strategic Tuple Reconstruction, is designed to improve the performance of tuple reconstruction operations by optimizing the selection of columns to be retrieved.

1. INTRODUCTION

2. Optimizing Tuple Reconstruction

3. Experimental Evaluation

4. Conclusion

Keywords: Database indexing, tuple reconstruction, column-stores, column-oriented databases.