Workload-Adaptive Indexing

Erwin M. Bakker & Stefan Manegold

https://homepages.cwi.nl/~manegold/DBDM/
http://liacs.leidenuniv.nl/~bakkerem2/dbdm/

s.manegold@liacs.leidenuniv.nl
e.m.bakker@liacs.leidenuniv.nl
Physical design problem

Database systems perform efficiently only after proper tuning...

which indexes to build?
on which data parts?
and when to build them?

DBA without adaptive indexing
Physical Design

Sample Workload

Timeline
Physical Design

Sample Workload → Analyze Performance → Timeline
Physical Design

Timeline

Sample Workload
Analyze Performance
Prepare Estimated physical design
Physical Design

Sample Workload

Analyze Performance

Prepare Estimated physical design

Queries

Timeline
Physical Design

Sample Workload  Analyze Performance  Prepare Estimated physical design  Queries

Timeline

Complex and time consuming process
Physical Design

Complex and time consuming process

Sample Workload ➔ Analyze Performance ➔ Prepare Estimated physical design ➔ Queries

Timeline

Dynamic Workloads

Very Large Databases
Dynamic environments

idle time  workload knowledge
Dynamic environments

idle time   workload knowledge

some problem cases
Dynamic environments

idle time  workload knowledge

some problem cases
• Not enough idle time to finish proper tuning
Dynamic environments

idle time  workload knowledge

some problem cases

• Not enough idle time to finish proper tuning
• By the time we finish tuning, the workload changes
Dynamic environments

idle time  workload knowledge

some problem cases
• Not enough idle time to finish proper tuning
• By the time we finish tuning, the workload changes
• No index support during tuning
Dynamic environments

idle time  workload knowledge

some problem cases

• Not enough idle time to finish proper tuning
• By the time we finish tuning, the workload changes
• No index support during tuning
• Not all data parts are equally useful
Adaptive Indexing

For dynamic environments:
Remove all tuning, physical design steps but still get similar performance as a fully tuned system

How?
Design new auto-tuning kernels
(operators, plans, structures, etc.)

DBA with adaptive indexing
Adaptive Indexing

- no monitoring
- no preparation
- no external tools
- no full indexes
- no human involvement
Adaptive Indexing

- no monitoring
- no preparation
- no external tools
- no full indexes
- no human involvement

Continuous on-the-fly physical reorganization
Adaptive Indexing

- no monitoring
- no preparation
- no external tools
- no full indexes
- no human involvement

Continuous on-the-fly physical reorganization
partially, incremental, adaptive indexing
Indexing Overview

workload analysis
index building
query processing

offline indexing
Indexing Overview

**Offline Indexing**
- Workload analysis
- Index building
- Query processing

**Online Indexing**
- Workload analysis
- Index building
- Query processing
Indexing Overview

Offline indexing
- Workload analysis
- Index building
- Query processing

Online indexing
- Workload analysis
- Index building
- Query processing

Adaptive indexing
- Workload analysis
- Index building
- Query processing
- Adaptive indexing
Indexing Overview

offline indexing

workload analysis
index building
query processing

online indexing

workload analysis
index building
query processing

adaptive indexing

workload knowledge
idle time

offline

online

adaptive

CWI

monetdb
Cracking the Database Store

Martin Kersten Stefan Manegold

CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

Abstract

Query performance strongly depends on finding an execution plan that touches as few superfluous tuples as possible. The access structures deployed for this purpose, however, are non-discriminative. They assume every subset of the domain being indexed is equally important, and their structures cause a high maintenance overhead during updates. This approach often fails in decision support or scientific environments where index selection represents a weak compromise amongst many plausible plans.

An alternative route, explored here, is to continuously adapt the database organization by making reorganization an integral part of the query evaluation process. Every query is first analyzed for its contribution to break the database into multiple pieces, such that both the required subset is easily retrieved and subsequent queries may benefit from the new partitioning structure.

To study the potentials for this approach, we developed a small representative multi-query benchmark and ran experiments against several open-source DBMSs. The results obtained are indicative for a significant reduction in system complexity with clear performance benefits.

1 Introduction

The ultimate dream for a query processor is to touch only those tuples in the database that matter for the production of the query answer. This ideal cannot be achieved easily, because it requires upfront knowledge of the user’s query intent.

In OLTP applications, all imaginable database subsets are considered of equal importance for query processing. The queries mostly retrieve just a few tuples without statistically relevant access dependencies. This permits a physical database design centered around index accelerators for individual tables and join-indices to speed up exploration of semantic meaningful links.

In decision support applications and scientific databases, however, it is a priori less evident what subsets are relevant for answering the -mostly statistical- queries. Queries tend to be ad-hoc and temporarily localized against a small portion of the databases. Data warehouse techniques, such as star- and snowflake schemas and bis-indices, are the primary tools to improve performance [Ral03].

In both domains, the ideal solution is approximated by a careful choice of auxiliary information to improve navigation to the database subset of interest. This choice is commonly made upfront by the database administrator and its properties are maintained during every database update. Alternatively, an automatic index selection tool may help in this process through analysis of the (anticipated) work load on the system [ZLL01], ACK+04]. Between successive database reorganizations, a query is optimized against this static navigational access structure.

Since the choice of access structures is a balance between storage and maintenance overhead, every query will inevitably touch many tuples of no interest. Although the access structures often permit a partial predicate evaluation, it is only after the complete predicate evaluation that we know which access was in vain.

In this paper we explore a different route based on the hypothesis that access maintenance should be a byproduct of query processing, not of updates. A query is interpreted as both a request for a particular database subset and as an advice to crack the database store into smaller pieces augmented with an index to access them. If it is unavoidable to touch uninteresting tuples during query evaluation, can we prepare for a better future? To illustrate, consider a simple query select * from R where R.a < 10 and a storage scheme that requires a full table scan, i.e. touching all tuples to select those of interest. The result produced in most systems is a stream of qualifying tuples. However, it can also be interpreted as a task to fragment the table into two pieces, i.e. apply horizontal fragmentation. This operation does not come for free, because the new table incarnation should be written back to persistent store and its properties stored in the catalog. For example, the original table can be replaced by a UNION-TA-

Database Cracking

Stratos Idrees CWI Amsterdam

CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

Stefan Manegold

CWI Amsterdam

Stefan.Manegold@cwi.nl

Abstract

Database cracking provides a non-discriminative navigational interface to localize tuples of interest. Their maintenance cost is taken during database updates. In this paper, we study the complementary approach, addressing index maintenance as part of query processing using continuous physical reorganization, i.e., cracking the database into manageable pieces. The motivation is that by automatically organizing data the way users request it, we can achieve fast access and the much desired self-organized behavior.

We present the first mature cracking architecture and report on our implementation of cracking in the context of a full fledged relational system. It led to a minor enhancement to its relational algebra kernel, such that cracking could be piggy-backed without incurring too much processing overhead. Furthermore, we illustrate the ripple effect of dynamic reorganization on the query plans derived by the SQL optimizer. The experiences and results obtained are indicative of a significant reduction in system complexity.

We show that the resulting system is able to self-organize based on incoming requests with clear performance benefits. This behavior is visible even when the user focus is randomly shifting to different parts of the data.

1. INTRODUCTION

Nowadays, the challenge for database architecture design is not in achieving ultra high performance but to design systems that are simple and flexible. A database system should be able to handle huge sets of data and self-organize according to the environment, e.g., the workload, available resources, etc. A nice discussion on such issues can be found in [8]. In addition, the trend towards distributed environments to speed up computation calls for new architecture designs. The same holds for multi-core CPU architectures that are starting to dominate the market and open new possibilities and challenges for data management. Some notable departures from the usual paths in database architecture design include [2, 3, 9, 14].

In this paper, we explore a radically new approach in database architecture, called database cracking. The cracking approach is based on the hypothesis that index maintenance should be a byproduct of query processing, not of updates. Each query is interpreted not only as a request for a particular result set, but also as an advice to crack the physical database store into smaller pieces. Each piece is described by a query, all of which are assembled in a cracker index to speedup future search. The cracker index replaces the non-discriminative indices (e.g., B-trees and hash tables) with a discriminative index. Only database portions of past interest are easily localized. The remainder is non-indexed until a query becomes interested. Continuously reacting on query requests brings the powerful property of self-organization. The cracker index is built dynamically while queries are processed and adapts to changing query workloads.

The cracking technique naturally provides a promising basis to attack the challenges described in the beginning of this section. With cracking, the way data is physically stored self-organizes according to query workload. Even with a huge data set, only tuples of interest are touched, leading to significant gains in query performance. In case the focus shifts to a different part of the data, the cracker index automatically adjusts to that. In addition, cracking the database into pieces gives us disjoint sets of data targeted by specific queries. This information can be nicely used as a basis for high-speed distributed and multi-core query processing.

The idea of physically reorganizing the database based on incoming queries has first been proposed in [10]. The contributions of this paper are the following. We present the first mature cracking architecture (a complete cracking software stack) in the context of column-oriented databases. We report on our implementation of cracking on top of MonetDB/SQ2, a column oriented database system, showing that cracking is easy to implement and may lead to further system simplification. We present the cracking algorithms that physically reorganize the database and the new cracking operators to enable cracking in MonetDB. Using SQ2 micro-benchmarks, we assess the efficiency and effectiveness of the system at the operator level. Additionally, we perform experiments that use the complete software stack, demonstrating that cracker-aware query optimizers can successfully generate query plans that deploy our new cracking operators and thus exploit the benefits of database cracking. Furthermore, we evaluate our current implementation and discuss some promising results. We clearly demonstrate that the resulting system can self-organize according to query
Cracking Example

Each query is treated as an advice on how data should be stored
Cracking Example

Each query is treated as an advice on how data should be stored.

Q1:
select *
from R
where R.A > 10
and R.A < 14

Q2:
select *
from R
where R.A > 7
and R.A <= 16

Column A

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>16</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>3</td>
<td>14</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cracking Example

Each query is treated as an advice on how data should be stored

Physically reorganize based on the selection predicate

<table>
<thead>
<tr>
<th>Column A</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>19</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

Q1:
select *
from R
where R.A > 10
and R.A < 14
Cracking Example

Each query is treated as an advice on how data should be stored

Physically reorganize based on the selection predicate

<table>
<thead>
<tr>
<th>Column A</th>
<th>Cracker column of A</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>19</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Q1:
select *
from R
where R.A > 10
and R.A < 14

Piece 1: A <= 10
Piece 2: 10 < A < 14
Piece 3: 14 <= A
Each query is treated as an advice on how data should be stored

Physically reorganize based on the selection predicate

Q1:
select *
from R
where R.A > 10
and R.A < 14

Q2:
select *
from R
where R.A > 7
and R.A <= 16
Cracking Example

Each query is treated as an advice on how data should be stored

Physically reorganize based on the selection predicate

<table>
<thead>
<tr>
<th>Column A</th>
<th>Cracker column of A</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>14</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>19</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
</tr>
</tbody>
</table>

Q1: select * from R where R.A > 10 and R.A < 14

Piece 1: A <= 10
Piece 2: 10 < A < 14
Piece 3: 14 <= A
Cracking Example

Each query is treated as an advice on how data should be stored.

Physically reorganize based on the selection predicate.

Q1:
select *
from R
where R.A > 10
and R.A < 14

Q2:
select *
from R
where R.A > 7
and R.A <= 16

Piece 1: A <= 7
Piece 2: 7 < A <= 10
Piece 3: 10 < A < 14
Piece 4: 14 <= A <= 16

Cracker column of A
Cracking Example

Each query is treated as an advice on how data should be stored

Physically reorganize based on the selection predicate

Column A

<table>
<thead>
<tr>
<th>13</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>14</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Cracker column of A

<table>
<thead>
<tr>
<th>4</th>
<th>9</th>
<th>2</th>
<th>7</th>
<th>1</th>
<th>3</th>
<th>8</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>12</td>
<td>11</td>
<td>16</td>
<td>19</td>
<td>14</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Piece 1: A <= 10
Piece 2: 10 < A < 14
Piece 3: 14 <= A

Q1:
select *
from R
where R.A > 10
and R.A < 14

Q2:
select *
from R
where R.A > 7
and R.A <= 16
Cracking Example

Each query is treated as an advice on how data should be stored

Physically reorganize based on the selection predicate

Q1:
select *
from R
where R.A > 10
and R.A < 14

Q2:
select *
from R
where R.A > 7
and R.A <= 16

Piece 1: A <= 7
Piece 2: 7 < A <= 10
Piece 3: 10 < A < 14
Piece 4: 14 <= A <= 16

Column A

Cracker column of A

13
16
4
9
2
12
7
1
19
3
14
11
8
6

Piece 1: A <= 10
Piece 2: 10 < A < 14
Piece 3: 14 <= A
## Cracking Example

Each query is treated as an advice on how data should be stored

Physically reorganize based on the selection predicate

<table>
<thead>
<tr>
<th>Column A</th>
<th>Cracker column of A</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>14</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>19</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
</tr>
</tbody>
</table>

Q1: select * from R where R.A > 10 and R.A < 14

Piece 1: A <= 10
Piece 2: 10 < A < 14
Piece 3: 14 <= A

Physically reorganize based on the selection predicate
Cracking Example

Each query is treated as an advice on how data should be stored

Physically reorganize based on the selection predicate

Q1:
select *
from R
where R.A > 10
and R.A < 14

Q2:
select *
from R
where R.A > 7
and R.A <= 16
Cracking Example

Each query is treated as an advice on how data should be stored

Physically reorganize based on the selection predicate

<table>
<thead>
<tr>
<th>Column A</th>
<th>Cracker column of A</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>14</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>19</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
</tr>
</tbody>
</table>

Q1:
select *
from R
where R.A > 10
and R.A < 14

Q2:
select *
from R
where R.A > 7
and R.A <= 14

Gain knowledge on how data is organized

Result tuples
Cracking Example

Each query is treated as an advice on how data should be stored

Physically reorganize based on the selection predicate

Q1:
select * from R
where R.A > 10
and R.A < 14

Q2:
select *
from R
where R.A > 7
and R.A <= 16

Gain knowledge on how data is organized

Dynamically/on-the-fly within the select-operator
Cracking Example

Each query is treated as an advice on how data should be stored

Physically reorganize based on the selection predicate

Column A  Cracker column of A

Q1: select * from R
where R.A > 10 and R.A < 14

Q2: select * from R
where R.A > 7 and R.A <= 16

Q1

Dynamically/on-the-fly within the select-operator
Cracking Example

Each query is treated as an advice on how data should be stored

Physically reorganize based on the selection predicate

<table>
<thead>
<tr>
<th>Column A</th>
<th>Cracker column of A</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>

Q1:
select *
from R
where R.A > 10
    and R.A < 14

Q2:
select *
from R
where R.A > 7
    and R.A <= 16

Piece 1: 10 < A <= 14
Piece 2: 14 <= A
Piece 3: A <= 7

Dynamically/on-the-fly within the select-operator
### Cracking Example

Each query is treated as an advice on how data should be stored.

Physically reorganize based on the selection predicate.

### Query 1 (Q1)
```
select *
from R
where R.A > 10
  and R.A < 14
```

### Cracker Column of A

<table>
<thead>
<tr>
<th>A</th>
<th>Cracker Column of A</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

**Piece 1:** $A \leq 10$

### Query 2 (Q2)
```
select *
from R
where R.A > 7
  and R.A <= 16
```

### Cracker Column of A

<table>
<thead>
<tr>
<th>A</th>
<th>Cracker Column of A</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

**Piece 2:** $10 < A \leq 16$

**Piece 3:** $10 < A < 14$

**Piece 4:** $14 \leq A$

Dynamically/on-the-fly within the select-operator.
Cracking Example

Each query is treated as an advice on how data should be stored

Physically reorganize based on the selection predicate

Column A

<table>
<thead>
<tr>
<th>13</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>19</td>
</tr>
<tr>
<td>19</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>14</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
</tr>
</tbody>
</table>

Cracker column of A

<table>
<thead>
<tr>
<th>4</th>
<th>9</th>
<th>2</th>
<th>1</th>
<th>3</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>12</td>
<td>11</td>
<td>16</td>
<td>19</td>
<td>14</td>
</tr>
</tbody>
</table>

Piece 1: \(A \leq 10\)
Piece 2: \(10 < A < 14\)
Piece 3: \(14 \leq A\)

Q1:
select *
from R
where R.A > 10
and R.A < 14

Q2:
select *
from R
where R.A > 7
and R.A <= 16

Dynamically/on-the-fly within the select-operator
Cracking Example

Each query is treated as an advice on how data should be stored

Physically reorganize based on the selection predicate

Q1:
select *
from R
where R.A > 10
and R.A < 14

Q2:
select *
from R
where R.A > 7
and R.A <= 16

Dynamically/on-the-fly within the select-operator

Cracker column of A

Piece 1: A <= 10
Piece 2: 10 < A < 14
Piece 3: 10 < A < 14
Piece 4: 14 <= A <= 16
Piece 5: 16 < A

Column A
13 16
4 9
2 7
1 8

Cracker column of A
4 9
2 7
1 8

Piece 1:
Piece 2:
Piece 3:
Piece 4:
Piece 5:
Cracking Example

Each query is treated as an advice on how data should be stored

Physically reorganize based on the selection predicate

Q1:
select *
from R
where R.A > 10
and R.A < 14

Q2:
select *
from R
where R.A > 7
and R.A <= 16

Piece 1: A <= 10
Piece 2: 7 < A <= 10
Piece 3: 10 < A < 14
Piece 4: 14 <= A <= 16
Piece 5: 16 < A

Dynamically/on-the-fly within the select-operator
Cracking Example

Each query is treated as an advice on how data should be stored

Physically reorganize based on the selection predicate

Q1:
select *
from R
where R.A > 10
and R.A < 14

Q2:
select *
from R
where R.A > 7
and R.A <= 16

Dynamically/on-the-fly within the select-operator
### Cracking Example

Each query is treated as an advice on how data should be stored.

**Physically reorganize based on the selection predicate.**

<table>
<thead>
<tr>
<th>Column A</th>
<th>Cracker column of A</th>
<th>Cracker column of A</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>13</td>
<td>19</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>14</td>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
<td>14</td>
</tr>
<tr>
<td>16</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>8</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Q1:**
- **select \***
- from **R**
- where **R.A > 10**
- and **R.A < 14**

**Q2:**
- **select \***
- from **R**
- where **R.A > 7**
- and **R.A <= 16**

**Pieces:**
- Piece 1: \[A <= 7\]
- Piece 2: \[7 < A <= 10\]
- Piece 3: \[10 < A < 14\]
- Piece 4: \[14 <= A <= 16\]
- Piece 5: \[16 < A\]

Dynamically/on-the-fly within the select-operator.
Cracking Example

Each query is treated as an advice on how data should be stored

Physically reorganize based on the selection predicate

Q1:
select *
from R
where R.A > 10
and R.A < 14

Q2:
select *
from R
where R.A > 7
and R.A <= 16

Dynamically/on-the-fly within the select-operator
Cracking Example

Each query is treated as an advice on how data should be stored

Physically reorganize based on the selection predicate

Q1:
select *
from R
where R.A > 10
and R.A < 14

Q2:
select *
from R
where R.A > 7
and R.A <= 16

Dynamically/on-the-fly within the select-operator
Cracking Example

Each query is treated as an advice on how data should be stored.

Physically reorganize based on the selection predicate

<table>
<thead>
<tr>
<th>Column A</th>
<th>Cracker column of A</th>
<th>Cracker column of A</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>16</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>14</td>
<td>16</td>
<td>14</td>
</tr>
<tr>
<td>19</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>19</td>
<td>19</td>
</tr>
</tbody>
</table>

Q1: select * from R where R.A > 10 and R.A < 14

Q2: select * from R where R.A > 7 and R.A <= 16

The more we crack, the more we learn

Physically reorganize based on the selection predicate

Dynamically/on-the-fly within the select-operator

Result tuples
Cracking Example

Each query is treated as an advice on how data should be stored

set-up

100K random selections
random selectivity
random value ranges
in a 10 million integer column
Cracking Example

Each query is treated as an advice on how data should be stored

set-up

100K random selections
random selectivity
random value ranges
in a 10 million integer column

almost no initialization overhead
Cracking Example

Each query is treated as an advice on how data should be stored

set-up

100K random selections
random selectivity
random value ranges
in a 10 million integer column

almost no
initialization overhead

continuous improvement
Cracking Example

Each query is treated as an advice on how data should be stored

set-up

100K random selections
random selectivity
random value ranges
in a 10 million integer column

almost no initialization overhead

continuous improvement
Cracking Example

Each query is treated as an advice on how data should be stored

set-up

10K random selections
selectivity 10%
random value ranges
in a 30 million integer column

Diagram:

Cumulative average response time (secs)

Query sequence

Scan

Full Index

Crack
Cracking Example
Each query is treated as an advice on how data should be stored

set-up
10K random selections
selectivity 10%
random value ranges
in a 30 million integer column

10K queries later,
Full Index still has not
amortized the initialization costs
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]
Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores*

Felix Halim† Stratos Idreos‡ Panagiotis Karras§ Roland H. C. Yap

†National University of Singapore (halim, ryap)@comp.nus.edu.sg
‡CWI, Amsterdam idreos@cwi.nl
§Rutgers University karras@business.rutgers.edu

ABSTRACT

Modern business applications and scientific databases call for inherently dynamic data storage environments. Such environments are characterized by two challenging features: (a) they have little idle system time to devote on physical design; and (b) there is little, if any, a priori workload knowledge, while the query and data workload keeps changing dynamically. In such environments, traditional approaches to index building and maintenance cannot apply. Database cracking has been proposed as a solution that allows on-the-fly physical data reorganization, as a collateral effect of query processing. Cracking aims to continuously and automatically adapt indexes to the workload at hand, without human intervention. Indexes are built incrementally, adaptively, and on demand. Nevertheless, as we show, existing adaptive indexing methods fail to deliver workload-robustness; they perform much better with random workloads than with others. This frailty derives from the inelasticity with which these approaches interpret each query as a hint on how data should be stored. Current cracking schemes blindly reorganize the data within each query’s range, even if that results into successive expensive operations with minimal indexing benefit.

In this paper, we introduce stochastic cracking, a significantly more resilient approach to adaptive indexing. Stochastic cracking also uses each query as a hint on how to reorganize data, but not blindly so; it gains resilience and avoids performance bottlenecks by deliberately applying certain arbitrary choices in its decision-making. Thereby, we bring adaptive indexing forward to a mature formulation that confers the workload-robustness previous approaches lacked. Our extensive experimental study verifies that stochastic cracking maintains the desired properties of original database cracking while at the same time it performs well with diverse realistic workloads.

1. INTRODUCTION

Database research has set out to reexamine established assumptions in order to meet the new challenges posed by big data, scientific databases, highly dynamic, distributed, and multi-core CPU environments. One of the major challenges is to create simple-to-use and flexible database systems that have the ability self-organize according to the environment [7].

Physical Design. Good performance in database systems largely relies on proper tuning and physical design. Typically, all tuning choices happen up front, assuming sufficient workload knowledge and idle time. Workload knowledge is necessary in order to determine the appropriate tuning actions, while idle time is required in order to perform those actions. Modern database systems rely on auto-tuning tools to carry out these steps, e.g., [6, 8, 13, 1, 28].

Dynamic Environments. However, in dynamic environments, workload knowledge and idle time are scarce resources. For example, in scientific databases new data arrives on a daily or even hourly basis, while query patterns follow an exploratory path as the scientists try to interpret the data and understand the patterns observed; there is no time and knowledge to analyze and prepare a different physical design every hour or even every day.

Traditional indexing presents three fundamental weaknesses in such cases: (a) the workload may have changed by the time we finish tuning; (b) there may be no time to finish tuning properly; and (c) there is no indexing support during tuning.

Database Cracking. Recently, a new approach to the physical design problem was proposed, namely database cracking [14]. Cracking introduces the notion of continuous, incremental, partial and on demand adaptive indexing. Thereby, indexes are incrementally built and refined during query processing. Cracking was proposed in the context of modern column-stores and has been hitherto applied for boosting the performance of the select operator [16], maintenance under updates [17], and arbitrary multi-attribute queries [18]. In addition, more recently these ideas have been extended to exploit a partition/merge -like logic [19, 11, 12].

Workload Robustness. Nevertheless, existing cracking schemes have not deeply questioned the particular way in which they interpret queries as a hint on how to organize the data store. They have adopted a simple interpretation, in which a select operator is taken to describe a range of the data that a discriminative cracker index should provide easy access to for future queries; the remainder of the data remains non-indexed until a query expresses interest therein. This simplicity confers advantages such as instant and lightweight adaptation; still, as we show, it also creates a problem.

Existing cracking schemes faithfully and obediently follow the hints provided by the queries in a workload, without examining whether these hints make good sense from a broader view. This approach fares quite well with random workloads, or workloads that expose consistent interest in certain regions of the data. However, in other realistic workloads, this approach can falter. For example, consider a workload where successive queries ask for consecutive items, as if they sequentially scan the value domain; we call this
Stochastic cracking

PVLDB2012, *Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main Memory Column Stores*
Felix Halim, Stratos Idreos, Panagiotis Karras and Roland Y. Chuan
Workload Robustness

**Observation:**
Queries define adaptive indexing actions
The kind of queries and the order of queries matter!

**Goal:**
Maintain adaptive behavior regardless of query input
Query patterns

column with 100 unique integers

Good pattern

Bad pattern
Query patterns

column with 100 unique integers

Good pattern

q1, v>60  N

Bad pattern
Query patterns

column with 100 unique integers

Good pattern

q1, v>60

q2, v<20 \sim \frac{N}{2}

Bad pattern

q1, v>60 \quad N
Query patterns

column with 100 unique integers

Good pattern

q1, v>60

q2, v<20 ~N/2

q3, v>90 ~N/2

Bad pattern
Query patterns

column with 100 unique integers

Good pattern

q2, v<20 ~N/2
q1, v>60 N
q3, v>90 ~N/2

Bad pattern

N q1, v<1
Query patterns

column with 100 unique integers

**Good pattern**

$q_2, v < 20 \sim N/2$

$q_1, v > 60 \quad N$

$q_3, v > 90 \sim N/2$

**Bad pattern**

$N \quad q_1, v < 1$

$N-1 \quad q_2, v < 2$
Query patterns

column with 100 unique integers

Good pattern

- q2, v<20 \sim N/2
- q1, v>60 \quad N
- q3, v>90 \sim N/2

Bad pattern

- N \quad q1, v<1
- N-1 \quad q2, v<2
- N-2 \quad q3, v<3
Query patterns

column size 100M
selectivity 10 tuples

Response time (secs)

Query sequence

a) Random Workload

Scan
Crack
Sort
Query patterns

column size 100M
selectivity 10 tuples

Response time (secs)
Query sequence

a) Random Workload
Scan
Crack
Sort

Stochastic Cracking, PVLDB 12
Query patterns

column size 100M
selectivity 10 tuples

a) Random Workload
b) Sequential Workload
Query patterns

column size 100M
selectivity 10 tuples

Response time (secs)

Query sequence

Performance degrades to scan
Query patterns

column size 100M
selectivity 10 tuples

a) Random Workload
b) Sequential Workload

tuples touched by cracking code

performance degrades to scan
Query patterns

column size 100M
selectivity 10 tuples

a) Random Workload

b) Sequential Workload

tuples touched by cracking code

Performance degrades to scan
Query patterns

column size 100M
selectivity 10 tuples

a) Random Workload

Scan
Crack
Sort

b) Sequential Workload

Scan
Crack
Sort

tuples touched by cracking code

Response time (secs)

Query sequence

performance degrades to scan
Query patterns

column size 100M
selectivity 10 tuples

- response time (secs)
  - query sequence
    - a) Random Workload
      - Scan
      - Crack
      - Sort
    - b) Sequential Workload
      - Scan
      - Crack
      - Sort

- tuples touched by cracking code
  - query sequence
  - performance degrades to scan
Query patterns

column size 100M
selectivity 10 tuples

Selectivity 10 tuples

tuples touched by cracking code

performance degrades to scan
Problem:
Blind adaptation to queries

Solution:
Query driven and data driven adaptation
Stochastic Cracking

Initial array contains values in [0-k], Query asks for range [low-high]

Initial Array

Cracking

0  low  high

k

k
Stochastic Cracking

Initial array contains values in [0-k], Query asks for range [low-high]

Initial Array

| 0 | low | high | k |

Cracking

| 0 | low | high | c2 | c1 | k |

DDC

| 0 | low | high | c2 | c1 | k |

Data Driven, Center (DDC):
1. Recursively crack a piece in exactly half until in L2 cache.
2. Then crack for the query bounds.
Stochastic Cracking

Data Driven, Center (DDC):
1. Recursively crack a piece in exactly half until in L2 cache.
2. Then crack for the query bounds.
**Stochastic Cracking**

Initial array contains values in [0-k], Query asks for range [low-high]

<table>
<thead>
<tr>
<th>Initial Array</th>
<th>0</th>
<th>low</th>
<th>high</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cracking</td>
<td>0</td>
<td>low</td>
<td>high</td>
<td>k</td>
</tr>
<tr>
<td>DDC</td>
<td>0</td>
<td>low</td>
<td>high</td>
<td>c2</td>
</tr>
</tbody>
</table>

**Data Driven, Center (DDC):**
1. Recursively crack a piece in exactly half until in L2 cache.
2. Then crack for the query bounds.
Stochastic Cracking

Initial array contains values in [0-k], Query asks for range [low-high]

Data Driven, Center (DDC):
1. Recursively crack a piece in exactly half until in L2 cache.
2. Then crack for the query bounds.
Stochastic Cracking

Initial array contains values in [0-k], Query asks for range [low-high]

<table>
<thead>
<tr>
<th>Initial Array</th>
<th>0</th>
<th>low</th>
<th>high</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cracking</td>
<td>0</td>
<td>low</td>
<td>high</td>
<td>k</td>
</tr>
<tr>
<td>DDC</td>
<td>0</td>
<td>low</td>
<td>high</td>
<td>c2</td>
</tr>
</tbody>
</table>

**Data Driven, Center (DDC):**
1. Recursively crack a piece in exactly half until in L2 cache.
2. Then crack for the query bounds.
## Stochastic Cracking

Initial array contains values in [0-k], Query asks for range [low-high]

<table>
<thead>
<tr>
<th>Array</th>
<th>Low</th>
<th>High</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial Array</td>
<td>0</td>
<td>k</td>
<td></td>
</tr>
<tr>
<td>Cracking</td>
<td>0</td>
<td>low</td>
<td>high</td>
</tr>
<tr>
<td>DDC</td>
<td>0</td>
<td>low</td>
<td>high</td>
</tr>
</tbody>
</table>
Stochastic Cracking

Initial array contains values in [0-k], Query asks for range [low-high]

**Initial Array**

| 0 | low | high | k |

**Cracking**

| 0 | low | high | k |

**DDC**

| 0 | low | high | c2 | c1 | k |

**DDR**

| 0 | low | high | r2 | r1 | k |

**Data Driven, Random (DDR):**

1. Recursively crack a piece randomly until in L2 cache.
2. Then crack for the query bounds.
Stochastic Cracking

Initial array contains values in [0-k], Query asks for range [low-high]

Initial Array

0 1 2 3 4 5 6 7 8 9 k

Cracking

0  low  high 1 2 3 4 5 6 7 8 9 k

DDC

0  low  high  c2 1 2 3 4 5 6 7 8 9 k

DDR

0  low  high  r2 1 2 3 4 5 6 7 8 9 k

Data Driven, Random (DDR):
1. Recursively crack a piece randomly until in L2 cache.
2. Then crack for the query bounds.
Stochastic Cracking

Initial array contains values in \([0-k]\), Query asks for range \([\text{low}-\text{high}]\)

- **Initial Array**
  - 0
  - \([0\ \text{low}\ \text{high}\ k]\)

- **Cracking**
  - 0
  - \([0\ \text{low}\ \text{high}\ k]\)

- **DDC**
  - 0
  - \([0\ \text{low}\ \text{high}\ c2\ c1\ k]\)

- **DDR**
  - 0
  - \([0\ \text{low}\ \text{high}\ r2\ r1\ k]\)

**Data Driven, Random (DDR):**
1. Recursively crack a piece randomly until in L2 cache.
2. Then crack for the query bounds.
Stochastic Cracking

Initial array contains values in [0-k], Query asks for range [low-high]

Initial Array

<table>
<thead>
<tr>
<th>0</th>
<th>low</th>
<th>high</th>
<th>k</th>
</tr>
</thead>
</table>

Cracking

| 0 | low | high | c2 | c1 | k |

DDC

| 0 | low | high | r2 | r1 | k |

Data Driven, Random (DDR):
1. Recursively crack a piece randomly until in L2 cache.
2. Then crack for the query bounds.
Data Driven, Random (DDR):
1. Recursively crack a piece randomly until in L2 cache.
2. Then crack for the query bounds.
Stochastic Cracking

Initial array contains values in [0-k], Query asks for range [low-high]

- **Initial Array**
  - 0 low high k

- **Cracking**
  - 0 low high k

- **DDC**
  - 0 low high c2 c1 k

- **DDR**
  - 0 low high r2 r1 k
Stochastic Cracking

Initial array contains values in [0-k], Query asks for range [low-high]

Initial Array

Cracking

DDC

DDR

DD1C

DD2R

MDD1R
## Stochastic Cracking

Initial array contains values in \([0-k]\), Query asks for range \([\text{low}-\text{high}]\)

<table>
<thead>
<tr>
<th></th>
<th>Initial Array</th>
<th></th>
<th>Cracking</th>
<th></th>
<th>DDC</th>
<th></th>
<th>DDR</th>
<th></th>
<th>DD1C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>low</td>
<td></td>
<td>low</td>
<td></td>
<td>low</td>
<td></td>
<td>low</td>
<td></td>
<td>low</td>
</tr>
<tr>
<td></td>
<td>high</td>
<td></td>
<td>high</td>
<td></td>
<td>high</td>
<td></td>
<td>r2</td>
<td></td>
<td>high</td>
</tr>
<tr>
<td></td>
<td></td>
<td>k</td>
<td></td>
<td>k</td>
<td></td>
<td>r1</td>
<td></td>
<td>k</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>k</td>
<td></td>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td>k</td>
<td></td>
</tr>
</tbody>
</table>

- \(c1\) and \(c2\) are specific values within the range.
- \(r1\) and \(r2\) are also specific values within the range.
- The diagram highlights \(c1\) as a critical value.
Stochastic Cracking

Initial array contains values in $[0-k]$, Query asks for range $[\text{low}-\text{high}]$.

<table>
<thead>
<tr>
<th>Initial Array</th>
<th>Cracking</th>
<th>DDC</th>
<th>DDR</th>
<th>DD1C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>low</td>
<td>low</td>
<td>low</td>
<td>low</td>
<td>low</td>
</tr>
<tr>
<td>high</td>
<td>high</td>
<td>high</td>
<td>high</td>
<td>high</td>
</tr>
</tbody>
</table>
Stochastic Cracking

Initial array contains values in [0-k], Query asks for range [low-high]

<table>
<thead>
<tr>
<th>Initial Array</th>
<th>0</th>
<th>low</th>
<th>high</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cracking</td>
<td>0</td>
<td>low</td>
<td>high</td>
<td>k</td>
</tr>
<tr>
<td>DDC</td>
<td>0</td>
<td>low</td>
<td>high</td>
<td>c2</td>
</tr>
<tr>
<td>DDR</td>
<td>0</td>
<td>low</td>
<td>high</td>
<td>r2</td>
</tr>
<tr>
<td>DD1C</td>
<td>0</td>
<td>low</td>
<td>high</td>
<td>c1</td>
</tr>
</tbody>
</table>
Stochastic Cracking

Initial array contains values in [0-k], Query asks for range [low-high]

| Initial Array | 0 | low | high | k |
| DDC           | 0 | low | high | c2 | c1 | k |
| DDR           | 0 | low | high | r2 | r1 | k |
| DD1C          | 0 | low | high | c1 | k |
| DD1R          | 0 | low | high | r1 | k |
Stochastic Cracking

Initial array contains values in [0-k], Query asks for range [low-high]

- **Initial Array**
  - Range: 0 to k

- **Cracking**
  - Range: 0 to k

- **DDC**
  - Range: 0 to k
  - c1, c2

- **DDR**
  - Range: 0 to k
  - r1, r2

- **DD1C**
  - Range: 0 to k
  - c1

- **DD1R**
  - Range: 0 to k
  - r1
# Stochastic Cracking

Initial array contains values in [0-k], Query asks for range [low-high]

<table>
<thead>
<tr>
<th>Initial Array</th>
<th>0</th>
<th>low</th>
<th>high</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cracking</td>
<td>0</td>
<td>low</td>
<td>high</td>
<td>k</td>
</tr>
<tr>
<td>DDR</td>
<td>0</td>
<td>low</td>
<td>high</td>
<td>r2</td>
</tr>
<tr>
<td>DD1R</td>
<td>0</td>
<td>low</td>
<td>high</td>
<td></td>
</tr>
</tbody>
</table>

DDC

<table>
<thead>
<tr>
<th>0</th>
<th>low</th>
<th>high</th>
<th>c2</th>
<th>c1</th>
<th>k</th>
</tr>
</thead>
</table>

DD1C

| 0 | low | high | c1 | k |

DD1R
Stochastic Cracking

Initial array contains values in \([0-k]\), Query asks for range \([\text{low}-\text{high}]\)

| Initial Array | 0 | low | high | k |
| DDC | 0 | low | high | c2 | c1 | k |
| DDR | 0 | low | high | r2 | r1 | k |
| DD1C | 0 | low | high | c1 | k |
| DD1R | 0 | low | high | r1 | k |
Stochastic Cracking

(a) Sort Crack

(b) Sort Crack

Cumulative Response time (secs)

Query sequence

DDC, DDR

DD1C, DD1R

monetdb
Hybrids

PVLDB11, Cracking what’s marged. Merging what’s cracked. Adaptive Indexing in Main-Memory Column-Stores
Stratos Idreos, Stefan Manegold, Harumi Kuno and Goetz Graefe
Adaptive Merging

EDBT'10, SMDB'10, Goetz Graefe and Harumi Kuno

Incremental sort via external merge sort steps
Adaptive Merging

EDBT’10, SMDB’10, Goetz Graefe and Harumi Kuno

Incremental sort via external merge sort steps
Adaptive Merging

EDBT’10, SMDB’10, Goetz Graefe and Harumi Kuno

Incremental sort via external merge sort steps

select(A,50,100)
Adaptive Merging

EDBT’10, SMDB’10, Goetz Graefe and Harumi Kuno

Incremental sort via external merge sort steps

select(A,50,100)
Adaptive Merging

EDBT’10, SMDB’10, Goetz Graefe and Harumi Kuno

*Incremental sort via external merge sort steps*

select(A,50,100)
Adaptive Merging

**EDBT’10, SMDB’10, Goetz Graefe and Harumi Kuno**

*Incremental sort via external merge sort steps*

select(A,50,100)
Adaptive Merging

EDBT’10, SMDB’10, Goetz Graefe and Harumi Kuno

Incremental sort via external merge sort steps

select(A,50,100)
Adaptive Merging

EDBT’10, SMDB’10, Goetz Graefe and Harumi Kuno

Incremental sort via external merge sort steps

select(A,50,100)
Adaptive Merging

EDBT’10, SMDB’10, Goetz Graefe and Harumi Kuno

Incremental sort via external merge sort steps

select(A,50,100)
Adaptive Merging

EDBT’10, SMDB’10, Goetz Graefe and Harumi Kuno

Incremental sort via external merge sort steps

select(A,50,100)
Adaptive Merging

EDBT’10, SMDB’10, Goetz Graefe and Harumi Kuno

*Incremental sort via external merge sort steps*

select(A, 50, 100)
Adaptive Merging

EDBT’10, SMDB’10, Goetz Graefe and Harumi Kuno

*Incremental sort via external merge sort steps*

```
select(A,50,100)
```
Adaptive Merging

**EDBT’10, SMDB’10**, Goetz Graefe and Harumi Kuno

*Incremental sort via external merge sort steps*

```
select(A, 50, 100)
```
Adaptive Merging

**EDBT’10, SMDB’10, Goetz Graefe and Harumi Kuno**

*Incremental sort via external merge sort steps*

\[ \text{select}(A, 50, 100) \]
Adaptive Merging

EDBT’10, SMDB’10, Goetz Graefe and Harumi Kuno

Incremental sort via external merge sort steps

select(A,50,100)

Initial

Final

binary search

sorted

50

00
Adaptive Merging

EDBT’10, SMDB’10, Goetz Graefe and Harumi Kuno

Incremental sort via external merge sort steps

select(A,50,100)  select(A,55,70)
Adaptive Merging

**EDBT’10, SMDB’10**, Goetz Graefe and Harumi Kuno

*Incremental sort via external merge sort steps*

select(A, 50, 100)  select(A, 55, 70)

Initial  Final

sorted

binary search
Adaptive Merging

EDBT'10, SMDB'10, Goetz Graefe and Harumi Kuno

Incremental sort via external merge sort steps

select(A, 50, 100) select(A, 55, 70) select(A, 150, 170)

Initial

Final
Adaptive Merging

EDBT'10, SMDB'10, Goetz Graefe and Harumi Kuno

Incremental sort via external merge sort steps

\[
\text{select}(A, 50, 100) \quad \text{select}(A, 55, 70) \quad \text{select}(A, 150, 170)
\]

Initial  Final

sorted

monetdb
Adaptive Merging

EDBT’10, SMDB’10, Goetz Graefe and Harumi Kuno

Incremental sort via external merge sort steps

select(A,50,100)  select(A,55,70)  select(A,150,170)
Adaptive Merging

EDBT’10, SMDB’10, Goetz Graefe and Harumi Kuno

Incremental sort via external merge sort steps

select(A,50,100)  select(A,55,70)  select(A,150,170)

Initial  Final  Final
Performance Analysis

set-up

10K random selections
selectivity 10%
random value ranges
in a 30 million integer column
Performance Analysis

set-up

10K random selections
selectivity 10%
random value ranges
in a 30 million integer column

Cumulative Average (secs)

Query sequence
Performance Analysis

set-up
10K random selections
selectivity 10%
random value ranges
in a 30 million integer column

10K random selections
selectivity 10%
random value ranges
in a 30 million integer column

hybrids pvldb 2011
Performance Analysis

**set-up**

10K random selections
selectivity 10%
random value ranges
in a 30 million integer column

AM: high init overhead
but fast convergence

---

**Graph:**

- **Axes:**
  - Y-axis: Cumulative Average (secs)
  - X-axis: Query sequence

- **Lines:**
  - Red: Scan
  - Green: Sort
  - Blue: AM
  - Pink: Crack

**Legend:**

- Hybrids PVLDB 2011
Performance Analysis

set-up
10K random selections
selectivity 10%
random value ranges
in a 30 million integer column

AM: high init overhead but fast convergence
Crack: low init overhead but slow convergence
Questions

• Adaptive merging in column-stores?

• Adaptive merging Vs Cracking?

• Can we learn from both AM and Cracking?
Questions

Adaptive merging and Cracking are extremes

What is there in between?
Crack-Crack

vary initialization and incremental steps taken
Crack-Crack

vary initialization and incremental steps taken
Crack-Crack

vary initialization and incremental steps taken

select(A,50,100)
Crack-Crack

vary initialization and incremental steps taken

select(A,50,100)
Crack-Crack

vary initialization and incremental steps taken

select(A,50,100)
Crack-Crack

vary initialization and incremental steps taken

select(A, 50, 100)
Crack-Crack

vary initialization and incremental steps taken

select(A,50,100)
Crack-Crack

*vary initialization and incremental steps taken*

```
select(A, 50, 100)
```
Crack-Crack

*vary initialization and incremental steps taken*

\[
\text{select}(A, 50, 100)
\]
Crack-Crack

vary initialization and incremental steps taken

select(A, 50, 100)

not sorted

50

00
Crack-Crack

*vary initialization and incremental steps taken*

\[
\text{select}(A,50,100) \quad \text{select}(A,55,70)
\]

not sorted

50
00

Hybrids PVLDB 2011
Crack-Crack

vary initialization and incremental steps taken

select(A,50,100)  select(A,55,70)

not sorted

50 00

50 100

Crack-Crack
Crack-Crack

*vary initialization and incremental steps taken*

select(A, 50, 100)  select(A, 55, 70)  select(A, 150, 170)
Crack-Crack

vary initialization and incremental steps taken

select(A,50,100)  select(A,55,70)  select(A,150,170)
Crack-Crack

vary initialization and incremental steps taken

select(A,50,100)  select(A,55,70)  select(A,150,170)
Adaptive Indexing

<table>
<thead>
<tr>
<th>initial partitions</th>
<th>final partitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sort</td>
<td>Sort</td>
</tr>
<tr>
<td>Radix</td>
<td>Radix</td>
</tr>
<tr>
<td>Crack</td>
<td>Crack</td>
</tr>
<tr>
<td>HSS</td>
<td>HSS</td>
</tr>
<tr>
<td>HSR</td>
<td>HSR</td>
</tr>
<tr>
<td>HSC</td>
<td>HSC</td>
</tr>
<tr>
<td>HRS</td>
<td>HRS</td>
</tr>
<tr>
<td>HRR</td>
<td>HRR</td>
</tr>
<tr>
<td>HRC</td>
<td>HRC</td>
</tr>
<tr>
<td>HCS</td>
<td>HCS</td>
</tr>
<tr>
<td>HCR</td>
<td>HCR</td>
</tr>
<tr>
<td>HCC</td>
<td>HCC</td>
</tr>
</tbody>
</table>

high – overhead – low
slow – convergence – fast

Hybrids PVLDB 2011
Adaptive Indexing

<table>
<thead>
<tr>
<th></th>
<th>Sort</th>
<th>HSS</th>
<th>HSR</th>
<th>HSC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radix</td>
<td>HRS</td>
<td>HRR</td>
<td>HRC</td>
<td></td>
</tr>
<tr>
<td>Crack</td>
<td>HCS</td>
<td>HCR</td>
<td>HCC</td>
<td></td>
</tr>
</tbody>
</table>

initial partitions

final partitions

- low – overhead – high
- slow – convergence – fast
- fast – convergence – slow

HCC
HCR
HCS
HSS
HSR
HRC
Adaptive Indexing

<table>
<thead>
<tr>
<th>Initial Partitions</th>
<th>Final Partitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sort</td>
<td>HSS</td>
</tr>
<tr>
<td>Radix</td>
<td>HRS</td>
</tr>
<tr>
<td>Crack</td>
<td>HCS</td>
</tr>
</tbody>
</table>

- High - overhead - low
- Slow - convergence - fast
- Fast - convergence - slow
Adaptive Indexing

<table>
<thead>
<tr>
<th>initial partitions</th>
<th>Sort</th>
<th>Radix</th>
<th>Crack</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSS</td>
<td>HRS</td>
<td>HCS</td>
<td></td>
</tr>
<tr>
<td>HSR</td>
<td>HRR</td>
<td>HCR</td>
<td></td>
</tr>
<tr>
<td>HSC</td>
<td></td>
<td></td>
<td>HCC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>final partitions</th>
<th>Sort</th>
<th>Radix</th>
<th>Crack</th>
</tr>
</thead>
</table>

Hybrids PVLDB 2011
Adaptive Indexing

<table>
<thead>
<tr>
<th>initial partitions</th>
<th>final partitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sort</td>
<td>HSS</td>
</tr>
<tr>
<td>Radix</td>
<td>HRS</td>
</tr>
<tr>
<td>Crack</td>
<td>HCS</td>
</tr>
</tbody>
</table>

- high – overhead – low
- slow – convergence – fast
- fast – convergence – slow

Hybrids PVLDB 2011
Adaptive Indexing

Response time (secs)

(a) (scan)

Hybrid:
- Crack Crack
- Crack Radix
- Crack Sort

(b) (scan)

Full Index

Queries: 1, 10, 100, 1000
Adaptive Indexing

Hybrids PVLDB 2011
Adaptive Indexing

Response time (secs)

(a) Hybrid: Scan, Cracking, Adaptive Merging, Full Index

(b) Hybrid: Crack Crack, Crack Radix, Crack Sort

Queries

Hybrids PVLDB 2011
Adaptive Indexing

How many queries before the index fully supports a random query?

- Full Index
- Adaptive Merging
- Ideal Hybrid
- CR
- CC
- Bad Hybrid
- Database Cracking
- Scan

Initialization Vs convergence tradeoff

Cost of first query relative to in-memory scan effort

- 10x
- 5x
- 2x
- 1x
- none
- 10
- 100
- 1000

Hybrids PVLDB 2011
Adaptive Indexing

How many queries before the index fully supports a random query?

Initialization Vs convergence tradeoff

Cost of first query relative to in-memory scan effort

Full Index

Adaptive Merging

Ideal Hybrid

Database Cracking

Bad Hybrid

Scan

none

10

100

1000

never

1x

2x

5x

10x
Adaptive Indexing

How many queries before the index fully supports a random query?

Cost of first query relative to in-memory scan effort

Initialization Vs convergence tradeoff

Full Index

Adaptive Merging

Ideal Hybrid

Bad Hybrid

Database Cracking

Scan

none

10

100

1000

never

1x

2x

5x

10x
Adaptive Indexing

How many queries before the index fully supports a random query?

- Full Index
- Adaptive Merging
- Bad Hybrid
- Database Cracking
- Scan

Initialization Vs convergence tradeoff

Cost of first query relative to in-memory scan effort

- 10x
- 5x
- 2x
- 1x
- none
- 10
- 100
- 1000
- never

Hybrids PVLDB 2011
Progressive Indexing

Self-organizing Tuple Reconstruction In Column-stores

I. INTRODUCTION

Database papers have a unique opportunity to explore the potential of main-memory column-stores in the context of progressive indexing. This involves the development of algorithms and techniques that can efficiently and effectively manage the indexing of large datasets stored in column stores. The focus is on how progressive indexing can be achieved in this environment, leveraging the strengths of column-stores to provide fast access and low storage requirements.

ABSTRACT

The main challenges in creating efficient and flexible database systems with the ability to support adaptive indexing are how to manage indexing costs and how to leverage the scalability and efficiency of column stores. To address these challenges, we propose the development of algorithms that can dynamically adjust indexing strategies based on usage patterns and data characteristics. This approach, referred to as self-organizing tuple reconstruction, enables efficient indexing in column stores while maintaining the scalability and performance benefits of these systems.

Categories and Subject Descriptors

Database systems, adaptive indexing, query optimization.

Keywords

Database papers, adaptive indexing, query optimization.
Adaptive Indexing: 1\textsuperscript{st} Query Costs

<table>
<thead>
<tr>
<th>Method</th>
<th>Query Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scan</td>
<td>0.1</td>
</tr>
<tr>
<td>Prog Stoch</td>
<td>0.5</td>
</tr>
<tr>
<td>Cracking</td>
<td>1.0</td>
</tr>
<tr>
<td>Stochastic</td>
<td>2.0</td>
</tr>
<tr>
<td>Coarse</td>
<td>2.0</td>
</tr>
<tr>
<td>Index</td>
<td>4.0</td>
</tr>
</tbody>
</table>
Progressive Indexing

Can we / how to:

• Reduce / limit 1st query cost / overhead?
• Improve query performance predictability and robustness?
• Ensure convergence towards full index?

Yet unexplored “dimensions”:

• Other sorting algorithms than quick-sort
• Suspend/resume steps / iterations
Progressive Indexing
Progressive Quick-Sort

Initialize

13
16
4
9
2
? ?
? ?
12
16
13

≤ 9

Refine

4  9  2
4  9  2  7  1  3
4  3  2  1  7  9

≤ 9

≤ 4

> 4

> 9

≤ 9

> 9

> 9
Progressive Merge-Sort

<table>
<thead>
<tr>
<th>Initialize</th>
<th>Refine</th>
</tr>
</thead>
<tbody>
<tr>
<td>13 16 4 9 2 7 1 19 3 14 11</td>
<td>2 4 9 12 13 16</td>
</tr>
</tbody>
</table>
Progressive Bucket-Sort

Initialize

Merge

Uninitialized

13 16 4 9 2 12 7 1 19 3 14 11

Initialize

4 9 2

[−∞, 10)

Merge

11 14 19 12 16 13

[10, ∞]

Uninitialized

1 2 3 4 7 9

[−∞, 10)
Progressive Radix-Sort

```
13 16 4 9 2 12 7 1 19 3 14 11
```

LSD

```
2 12 13 4 16 9
```

Next Digit

```
1 11 2 12 13 3 4 14 16 7 9 19
```

```
1 2 3 11 12 13
```
Experimental Setup

- **Software:**
  - stand-alone C++ program, g++ -O3
  - Fedora 26

- **Hardware:**
  - Intel Core i7-2600K CPU @ 3.40 GHz, 8 cores, 8 MB L3 cache
  - 16 GB main memory

- **Data:**
  - 8-byte integers
  - $10^8$ uniformly distributed values

- **Queries:**
  - SELECT SUM(R.A) FROM R WHERE R.A BETWEEN V1 AND V2

- **Experiments:**
  - repeat entire workload 10 times
  - report median runtime per query
  - Default: 1000 queries, 10% selectivity, random workload
Random Workload
Varying $\delta$: 1st Query Cost

- Bucket
- Merge
- Quick
- Radix

Query Time (s)

$\delta$
Varying $\delta$: # Queries until Pay-off
Varying $\delta$:
# Queries until Convergence

![Graph showing the number of queries until convergence varying $\delta$.]
Varying $\delta$: Entire Workload Cost

![Graph showing varying $\delta$ and total time (s)]
Chosen $\delta$: 1

$^{st}$ Query $\sim= 2x$ Scan

<table>
<thead>
<tr>
<th>Indexing Method</th>
<th>$\delta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bucketsort</td>
<td>0.009</td>
</tr>
<tr>
<td>Mergesort</td>
<td>0.05</td>
</tr>
<tr>
<td>Quicksort</td>
<td>0.22</td>
</tr>
<tr>
<td>Radixsort</td>
<td>0.08</td>
</tr>
</tbody>
</table>
Comparison: 1\textsuperscript{st} Query
Comparison:
Entire Workload

Cumulative Time (s)

Query (#)

- Bucket
- Merge
- Quick
- Radix
- Stochastic
- Cracking
- Prog Stoch
- Coarse
- Scan
- Index
Comparison: Adaptive Indexing

![Graph showing query time comparison for different indexing methods. The x-axis represents query number, and the y-axis represents query time (log(s)). The methods compared are Coarse, Cracking, Prog Stoch, and Stochastic.]
Comparison: Progressive Indexing

![Graph showing comparison between different methods for Indexing and Scan operations. The x-axis represents Query (#), and the y-axis represents Query Time (log(s)). The methods compared are Bucket, Merge, Quick, and Radix.]
Random Workload

Query Range

Query (#)

$10^8$
Sequential Workload
Skewed Workload
Different Workloads

Different Workloads with various query time graphs for Random, Sequential, and Skewed workloads.

- **Random**: Coarse, Cracking, Prog Stoch, Stochastic
- **Sequential**: Coarse, Cracking, Prog Stoch, Stochastic
- **Skewed**: Bucket, Merge, Quick, Radix
# Queries until Pay-off

<table>
<thead>
<tr>
<th>Indexing Method</th>
<th>Random</th>
<th>Sequential</th>
<th>Skewed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full Index</td>
<td>56</td>
<td>56</td>
<td>56</td>
</tr>
<tr>
<td>Standard Cracking</td>
<td>28</td>
<td>63</td>
<td>22</td>
</tr>
<tr>
<td>Stochastic Cracking</td>
<td>69</td>
<td>40</td>
<td>49</td>
</tr>
<tr>
<td>Progressive Stochastic</td>
<td>67</td>
<td>47</td>
<td>48</td>
</tr>
<tr>
<td>Coarse Granular Index</td>
<td>42</td>
<td>76</td>
<td>38</td>
</tr>
<tr>
<td>Bucket sort</td>
<td>258</td>
<td>261</td>
<td>257</td>
</tr>
<tr>
<td>Mergesort</td>
<td>113</td>
<td>114</td>
<td>114</td>
</tr>
<tr>
<td>Quicksort</td>
<td>136</td>
<td>128</td>
<td>139</td>
</tr>
<tr>
<td>Radixsort</td>
<td>200</td>
<td>200</td>
<td>200</td>
</tr>
</tbody>
</table>
Progressive Indexing

- Robust & predictable query performance under various workloads

- Balance between
  - Fast convergence to full index
  - Small overhead for 1\textsuperscript{st} query

- Various basic sorting algorithms
  - Quick-sort
  - Merge-sort
  - Bucket-sort
  - Radix-sort