Merging What’s Cracked, Cracking What’s Merged

Adaptive Indexing in Main-Memory Column-Stores

Stratos Idreos, Stefan Manegold, (CWI)
Harumi Kuno, Goetz Graefe (HP Labs)

PVLDB 2011, 4(9)
Physical design problem

Database systems perform efficiently only after proper tuning...

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

Sample Workload

Timeline
Physical Design

Timeline

Sample Workload

Analyze Performance
Physical Design

Sample Workload → Analyze Performance → Prepare Estimated physical design → Timeline
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

Sample Workload
Analyze Performance
Prepare Estimated physical design
Timeline
Queries

Complex and time consuming process

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
Database Cracking

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 cracking
Database Cracking

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

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

Continuous on-the-fly physical reorganization
Database Cracking

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

Continuous on-the-fly physical reorganization

partial, incremental, adaptive indexing
Database Cracking

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

Continuous on-the-fly physical reorganization
partial, incremental, adaptive indexing

designed for modern column-stores
Database Cracking

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

<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

Column A

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>Piece 1: A &lt;= 10</td>
</tr>
<tr>
<td>3</td>
<td>Piece 2: 10 &lt; A &lt; 14</td>
</tr>
<tr>
<td>14</td>
<td>Piece 3: 14 &lt;= A</td>
</tr>
<tr>
<td>11</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
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

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

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
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

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

Cracker column of A

<table>
<thead>
<tr>
<th>Column A</th>
<th>Cracker column of A</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<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>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>8</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

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></td>
<td></td>
</tr>
<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 <= 16

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: $A \leq 7$
Piece 2: $7 < A \leq 10$
Piece 3: $10 < A < 14$
Piece 4: $14 \leq A \leq 16$

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

Result tuples

Column A

| 13 | 16 | 4 | 9 | 2 | 7 | 1 | 3 | 8 | 6 |

Cracker column of A

| 4 | 9 | 2 | 7 | 1 | 3 | 8 | 6 |

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

Gain knowledge on how data is organized
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

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

Cracker column of A

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

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

Piece 1: A <= 10

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

Piece 2: 10 < A < 14

Piece 3: 14 <= 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

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

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>7</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</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

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

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

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

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 <= 14

The more we crack, the more we learn.

Dynamically/on-the-fly within the select-operator.
Self-organizing behavior (TPC-H)

Response time (milli secs) vs. Query sequence for TPC-H Query 15.
Self-organizing behavior (TPC-H)

Database Cracking, SIGMOD 09
Self-organizing behavior (TPC-H)

Preparation cost
Presorted MonetDB
3-14 minutes
Self-organizing behavior (TPC-H)

- MonetDB
- Presorted
- Sel. Crack
- Sid. Crack
- MySQL
- Presorted

Response time (milli secs)

- Normal MonetDB
- Selection cracking
- MonetDB with sideways cracking

Preparation cost 3-14 minutes

TPC-H Query 15

764
420

Database Cracking, SIGMOD 09
Self-organizing behavior (TPC-H)

- Presorted MonetDB
  - Preparation cost: 3-14 minutes

- Normal MonetDB

- Selection cracking

- MonetDB with sideways cracking

- Query sequence

- TPC-H Query 15

- MonetDB
  - Presorted

- MySQL
  - Presorted

- Response time (milli secs)

- Database Cracking, SIGMOD 09
Self-organizing behavior (TPC-H)

- MonetDB
  - Presorted
- Sel. Crack
- Sid. Crack
- MySQL
  - Presorted

Query sequence

Response time (milli secs)

TPC-H Query 15

Preparation cost: 3-14 minutes

Database Cracking, SIGMOD 09
Self-organizing behavior (TPC-H)

- MonetDB
- Sel. Crack
- MySQL
- Presorted
- Sid. Crack

<table>
<thead>
<tr>
<th>Query sequence</th>
<th>Preparation cost</th>
<th>Response time (milli secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPC-H Query 15</td>
<td>3-14 minutes</td>
<td></td>
</tr>
</tbody>
</table>

- Normal MonetDB
- Selection cracking
- MonetDB with sideways cracking

Presorted MonetDB

764
420
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**
- Adaptive indexing
Indexing Overview

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

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

**Adaptive Indexing**
- Adaptive indexing

Diagram illustrating the workflow of indexing with different timelines for offline, online, and adaptive indexing.
Database Cracking
Each query is treated as an advice on how data should be stored

CIDR’07 Selection cracking
SIGMOD’07 Updates
SIGMOD’09 Sideways and partial cracking

Can be thought of as an incremental quicksort
Database Cracking
Each query is treated as an advice on how data should be stored

CIDR’07 Selection cracking
SIGMOD’07 Updates
SIGMOD’09 Sideways and partial cracking

Can be thought of as an incremental quicksort

The core cracking algorithm is extremely lazy
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*

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

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)

Initial

Final
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
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)
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*

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

```
Initial
```

```
Final
```

(sorted) 50 00

(sorted) 50 100

(sorted) 50 100
Adaptive Merging

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

*Incremental sort via external merge sort steps*

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

Initial

Final

binary search

sorted

50

100

50

100

50

150

170
Questions

• Adaptive merging in column-stores?
• Adaptive merging Vs Cracking?
• Can we learn from both AM and Cracking?
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

Cumulative Average (secs)

Query sequence

Scan
Sort
AM
Crack
Performance Analysis

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

Cumulative Average (secs)

Query sequence

Scan
Sort
AM
Crack
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

AM: high init overhead
but fast convergence
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 and Cracking are extremes
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

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)   select(A,55,70)
Crack-Crack

vary initialization and incremental steps taken

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

\[
\text{not sorted} \quad \text{crack}
\]

\[
50 \quad 00 \quad 50 \quad 100
\]
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>HSS</td>
</tr>
<tr>
<td>Radix</td>
<td>HRS</td>
</tr>
<tr>
<td>Crack</td>
<td>HCS</td>
</tr>
</tbody>
</table>

slow – convergence – fast

fast – convergence – slow

high – overhead – low

low – overhead – high
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>
<tr>
<td></td>
<td>HSR</td>
</tr>
<tr>
<td></td>
<td>HRR</td>
</tr>
<tr>
<td></td>
<td>HCR</td>
</tr>
<tr>
<td></td>
<td>HSC</td>
</tr>
<tr>
<td></td>
<td>HRS</td>
</tr>
<tr>
<td></td>
<td>HRR</td>
</tr>
<tr>
<td></td>
<td>HRC</td>
</tr>
<tr>
<td></td>
<td>HCS</td>
</tr>
<tr>
<td></td>
<td>HCR</td>
</tr>
<tr>
<td></td>
<td>HCC</td>
</tr>
</tbody>
</table>

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

low – overhead – high
# Adaptive Indexing

## Initial Partitions

<table>
<thead>
<tr>
<th>Method</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>HRC</td>
<td>HCC</td>
<td></td>
</tr>
</tbody>
</table>

## Final Partitions

- **Sort**: Fast convergence, low overhead
- **Radix**: Slow convergence, high overhead
- **Crack**: High overhead, low convergence

---

**Note**: This diagram illustrates the trade-offs between different indexing methods in terms of initial and final partitions. The methods include Sort, Radix, and Crack, with corresponding overhead and convergence characteristics.
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>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Sort</td>
<td></td>
</tr>
<tr>
<td>Radix</td>
<td></td>
</tr>
<tr>
<td>Crack</td>
<td></td>
</tr>
</tbody>
</table>

- **Sort**: fast convergence, high overhead
- **Radix**: slow convergence, low overhead
- **Crack**: adaptive indexing

- **HSS**: high convergence, high overhead
- **HSR**: high convergence, low overhead
- **HSC**: adaptive indexing
- **HRS**: slow convergence, high overhead
- **HRR**: slow convergence, low overhead
- **HRC**: adaptive indexing
- **HCS**: adaptive indexing
- **HCR**: adaptive indexing
- **HCC**: adaptive indexing
# 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>

- **Sort**: fast convergence, low overhead.
- **Radix**: slow convergence, high overhead.
- **Crack**: fast convergence, high overhead.

- **HSS**: high convergence, slow overhead.
- **HSR**: high convergence, low overhead.
- **HSC**: fast convergence, low overhead.
- **HRS**: fast convergence, high overhead.
- **HRR**: slow convergence, low overhead.
- **HRC**: slow convergence, high overhead.
- **HCS**: fast convergence, high overhead.
- **HCR**: slow convergence, high overhead.
- **HCC**: fast convergence, high overhead.
Adaptive Indexing

Response time (secs)

Queries

(a)

(b)

Hybrid:
- Crack Crack
- Crack Radix
- Crack Sort

Full Index

Scan

Adaptive Merging
Adaptive Indexing

![Graph showing response time vs queries for different indexing methods.](image)

- **Adaptive Indexing**
- **Queries**: 1, 10, 100, 1000
- **Response time (secs)**: 1e-05, 0.0001, 0.001, 0.01, 0.1, 1, 10, 100

**Methods**:
- Scan
- Cracking
- Crack Crack
- Crack Radix
- Crack Sort
- Adaptive Merging
- Full Index

**Legend**:
- + Scan
- * Cracking
- * Adaptive Merging
- □ Full Index

**Graph a)** and **Graph b)** display the performance of these methods under varying query loads.
Adaptive Indexing

![Graph showing response time (secs) vs queries for different indexing methods: Scan, Cracking, Adaptive Merging, Full Index, Crack Crack, Crack Radix, Crack Sort, Hybrid. The graph highlights the performance of each method under varying query loads.](image-url)
Adaptive Indexing

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

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

Response time (secs)

(a) Scan
- + Scan
- * Crack
- x Cracking
- * Adaptive Merging
- o Full Index

(b) Hybrid:
- o Crack Crack
- * Crack Radix
- + Crack Sort

Queries

1 10 100 1000

(a) (scan)

(b) (scan)
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

none

10

100

1000
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

None

10x

5x

2x

1x

Full Index

Adaptive Merging

Ideal Hybrid

Database Cracking

Bad Hybrid

Scan

never

10

100

1000
Adaptive Indexing

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

None

Cost of first query relative to in-memory scan effort

Full Index

Adaptive Merging

Ideal Hybrid

Bad Hybrid

Database Cracking

Scan

Initialization Vs convergence tradeoff

none

10

100

1000

never
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

never

none 10 100 1000

1x

2x

5x

10x

none
Adaptive Indexing

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

Full Index
Adaptive Merging
Ideal Hybrid
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

Disk Concurrency
Adaptive Indexing

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

Cost of first query relative to in-memory scan effort

- Full Index: 10x
- Adaptive Merging: 5x
- Ideal Hybrid: 2x
- Database Cracking: CC
- Bad Hybrid: CR

Initialization vs convergence tradeoff

More active is best

Disk vs Concurrency

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

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

Cost of first query relative to in-memory scan effort:
- Full Index: 10x
- Adaptive Merging: 5x
- Ideal Hybrid: 2x
- Bad Hybrid: none

Initialization vs convergence tradeoff:
- More active is best

Disk vs Concurrency vs Updates:
- Database Cracking
- Bad Hybrid
- Scan

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

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

Initialization Vs convergence tradeoff

More active is best

More lazy is best

Cost of first query relative to in-memory scan effort

Disk
Concurrency
Updates

Ideal Hybrid

Adaptive Merging

Full Index

Database Cracking

Scan

Bad Hybrid

CC

CS

CR

How many queries before the index fully supports a random query?
More in the paper...

*Column-store design details and concerns*

*Selectivity effects*

*Concurrency control examples*

*and more...*
Ongoing and open topics

Concurrency control

Compression

Multi-cores

Workload robustness

Disk based

Aggregations

Optimizer rules

Pipelining

Row-stores

Adaptive Indexing + Auto tuning tools

...and many more...
Ongoing and open topics

- Concurrency control
- Compression
- Multi-cores
- Workload robustness
- Disk based
- Aggregations
- Optimizer rules
- Pipelining
- Row-stores
- Adaptive Indexing + Auto tuning tools
- ...and many more...

Thank you!