

# Physical design problem

## 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

Database systems perform efficiently  
only after proper tuning...



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

Databases and Data Mining 2018



DBA without adaptive indexing



## Physical Design

Sample  
Workload

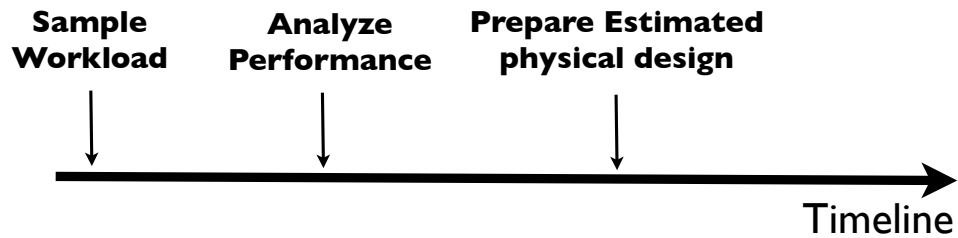


Sample  
Workload

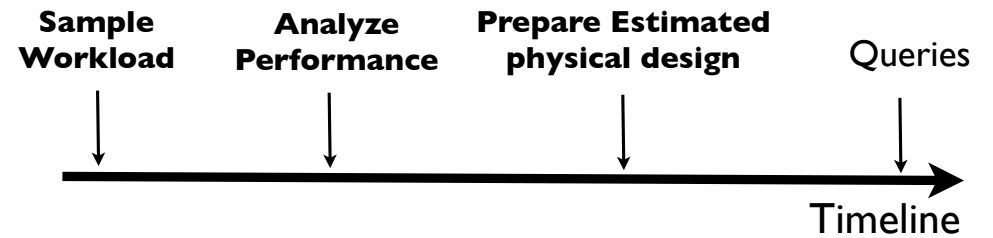
Analyze  
Performance



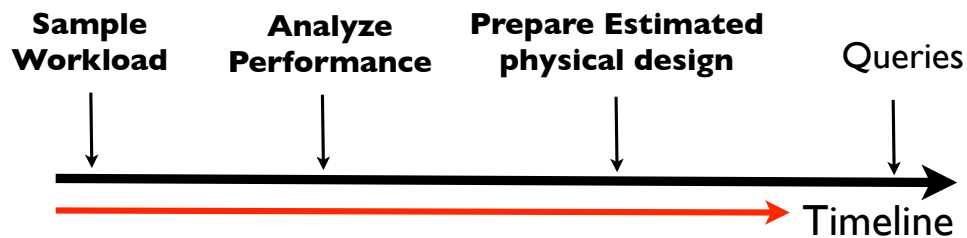
# Physical Design



# Physical Design

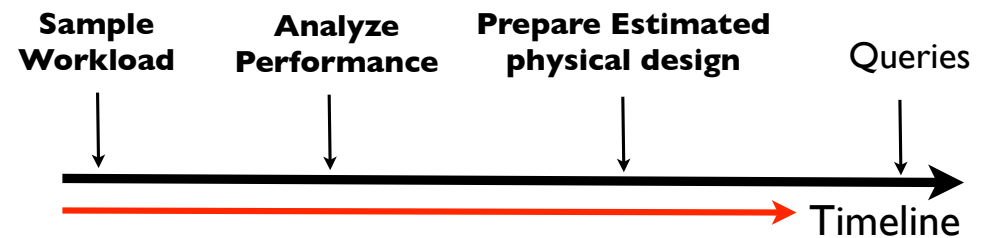


# Physical Design



**Complex and time consuming process**

# Physical Design



**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



# Adaptive Indexing

*For dynamic environments:*

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



DBA with adaptive indexing

*How?*

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



# 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

*no monitoring*

*no preparation*

*no external tools*

*no full indexes*

*no human involvement*





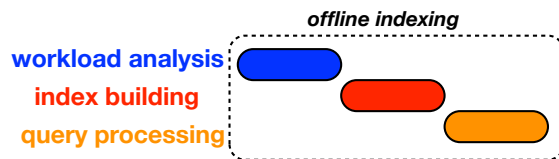
# Adaptive Indexing

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

**Continuous on-the-fly physical reorganization**



# Indexing Overview



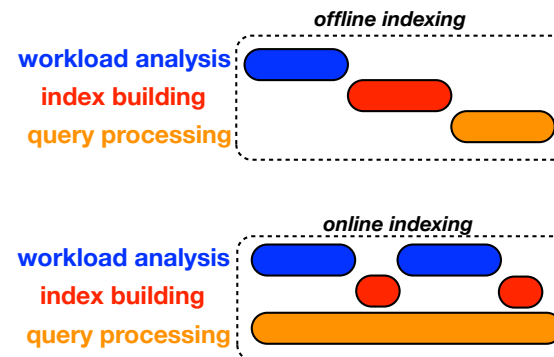
# Adaptive Indexing

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

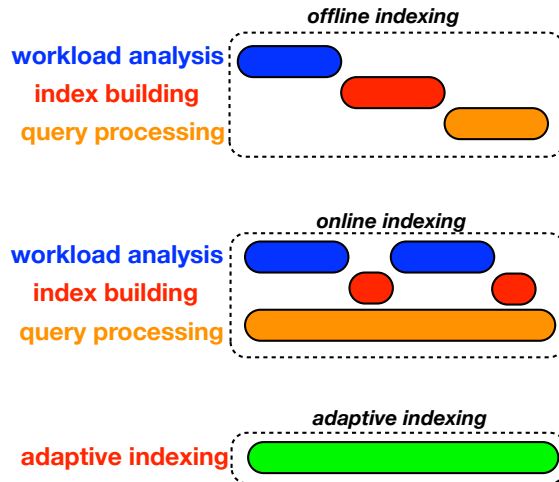
**Continuous on-the-fly physical reorganization**  
**partial, incremental, adaptive indexing**



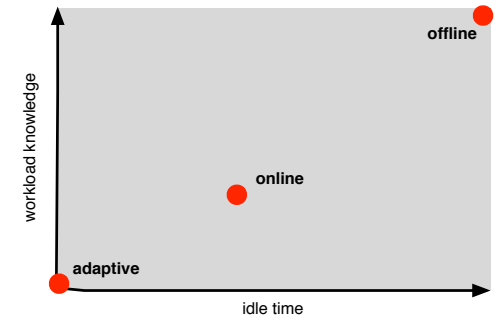
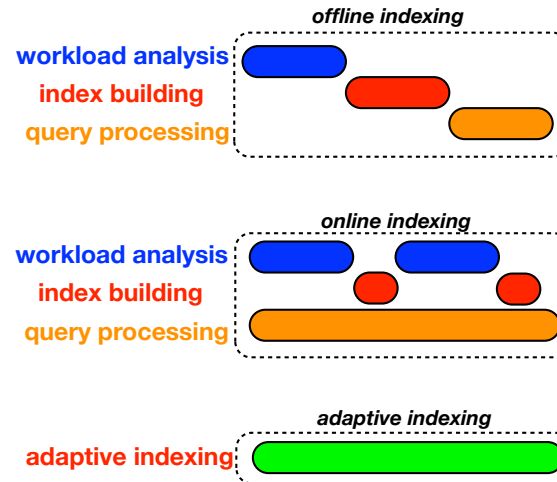
# Indexing Overview



# Indexing Overview



# Indexing Overview



Database Cracking CIDR 2007

## Cracking Example

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



### Cracking the Database Store

Martin Kersten  
CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands  
(martin.kersten, stefan.manegold@cwi.nl)

Stefan Manegold  
CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands  
(stefan.manegold@cwi.nl)

#### Abstract

Query performance strongly depends on finding an execution plan that touches as few superfluous tuples as possible. The access structure 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 run 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 idea cannot be achieved easily, because it requires upfront knowledge of the user's query intent.

In OLTP applications, all imaginable database queries are considered of equal importance for query processing. The queries mostly retrieve just a few tuples without statistically relevant intra-dependencies. This permits a physical

Reorganization to copy without for all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 2005 CIDR Conference

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 bit-indices, are the primary tools to improve performance [Raf03].

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, ACK04]. 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 data-uninteresting tuples during query evaluation, can we use that to prepare for a better future?

To illustrate, consider a simple query  $\text{select } * \text{ from } R \text{ where } R.a < 10$  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 TABLE

### Database Cracking

Stratos Idreos  
CWI Amsterdam  
The Netherlands  
Stratos.Idreos@cwi.nl

Martin L. Kersten  
CWI Amsterdam  
The Netherlands  
Martin.Kersten@cwi.nl

Stefan Manegold  
CWI Amsterdam  
The Netherlands  
Stefan.Manegold@cwi.nl

#### ABSTRACT

Database indices provide a non-discriminative navigational infrastructure 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 reorganizing data the way users request it, we can achieve fast access and the much desired self-organized behavior. We present the first system-cracking architecture and report on our implementation of cracking in the context of a full featured 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 experiments 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 viable 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 large 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 [6]. 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, 5, 14].

This article is published under a Creative Commons License Agreement (http://creativecommons.org/licenses/by/2.0). You may copy, distribute, display, or perform the work, make derivative works and make commercial use of the work, but you must attribute the work to the author and CIDR 2007.

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 subset of data, 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 speed up 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 unexplored territory and remains 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 large 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 our 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 [20]. The contributions of this paper are the following. We present the first system-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/SQL, 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 SQL microbenchmarks, 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-driven 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

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

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



# 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

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



# 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

Column A	Cracker column of A
13	4
16	9
4	2
9	7
2	1
12	3
7	8
1	6
19	13
3	12
14	11
11	16
8	19
6	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

Column A	Cracker column of A
13	4
16	9
4	2
9	7
2	1
12	3
7	8
1	6
19	13
3	12
14	11
11	16
8	19
6	14

Piece 1:  
A <= 10

Piece 2:  
10 < A < 14

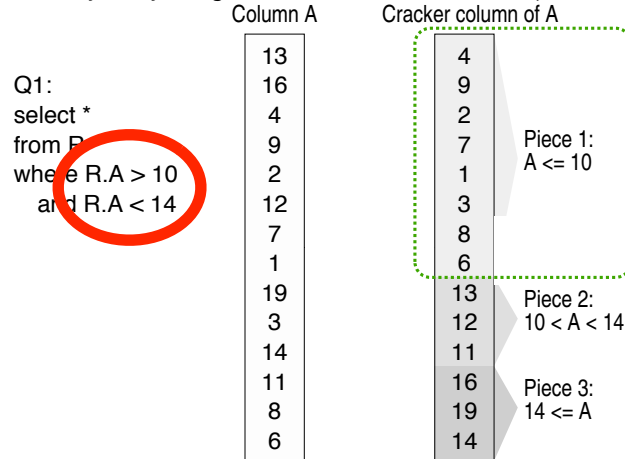
Piece 3:  
14 <= A



# Cracking Example

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

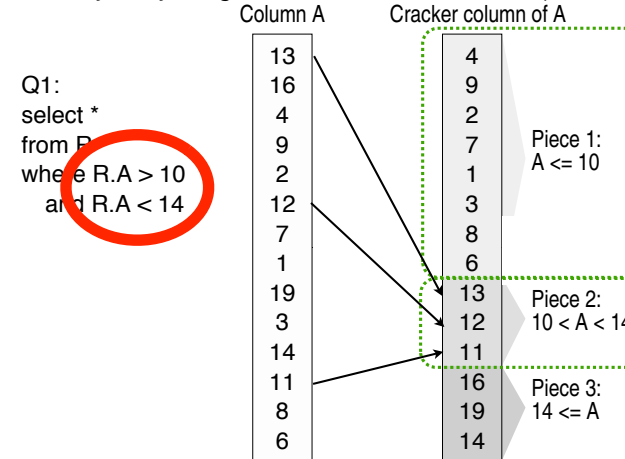
Physically reorganize based on the selection predicate



# Cracking Example

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

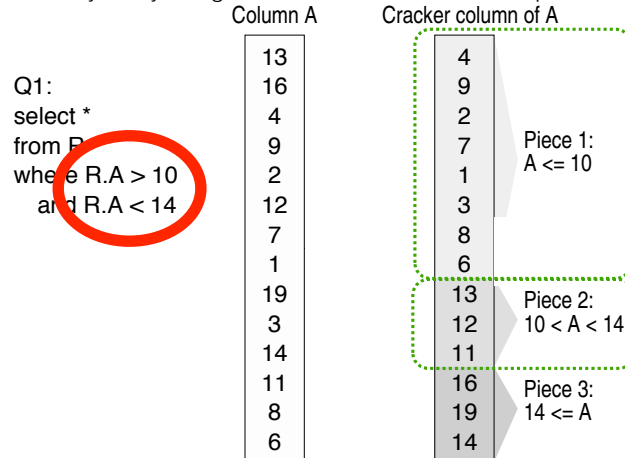
Physically reorganize based on the selection predicate



# Cracking Example

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

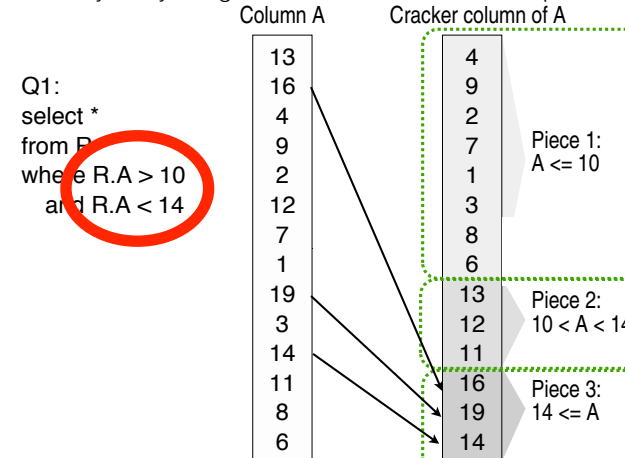
Physically reorganize based on the selection predicate



# Cracking Example

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

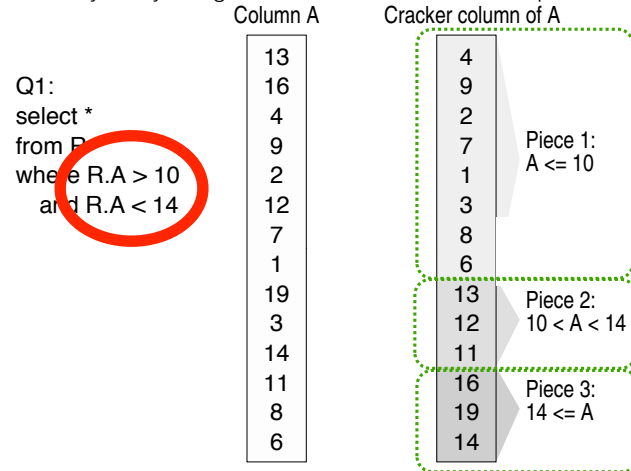
Physically reorganize based on the selection predicate



# Cracking Example

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

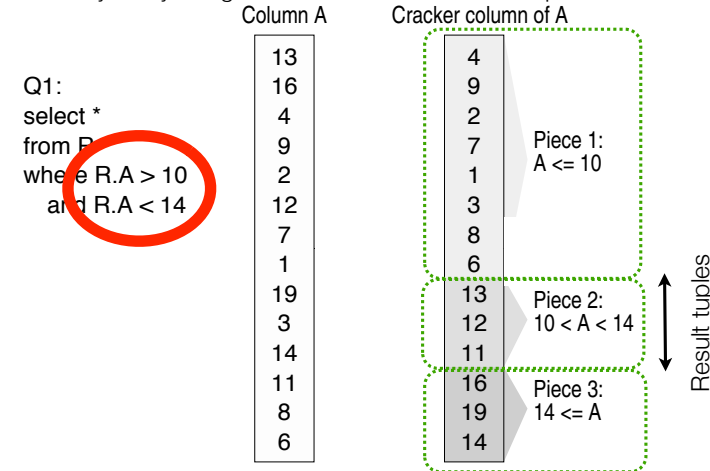
Physically reorganize based on the selection predicate



# Cracking Example

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

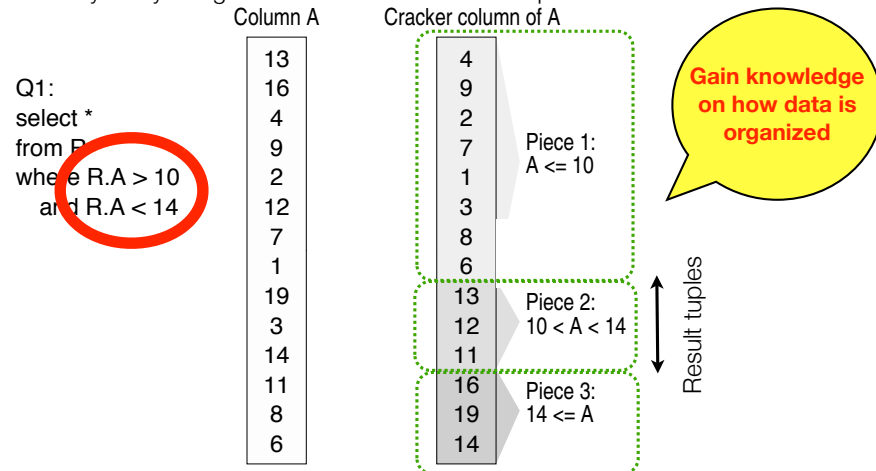
Physically reorganize based on the selection predicate



# Cracking Example

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

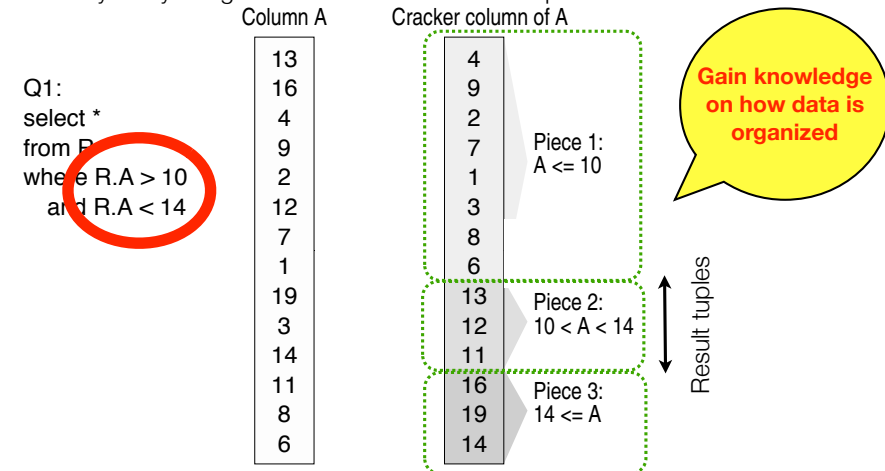
Physically reorganize based on the selection predicate



# Cracking Example

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

Physically reorganize based on the selection predicate



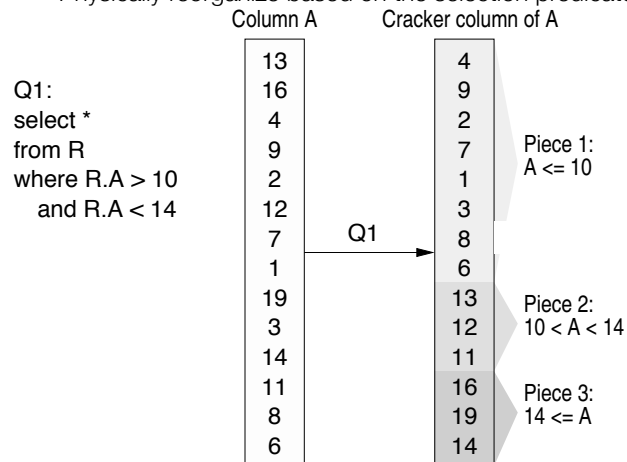
Dynamically/on-the-fly within the select-operator



# Cracking Example

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

Physically reorganize based on the selection predicate



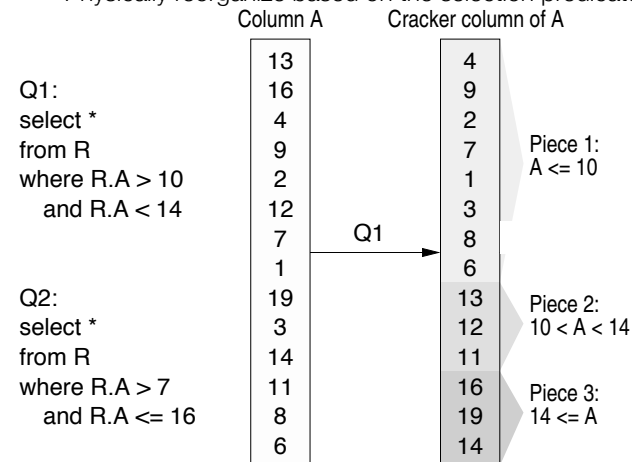
Dynamically/on-the-fly within the select-operator



# Cracking Example

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

Physically reorganize based on the selection predicate



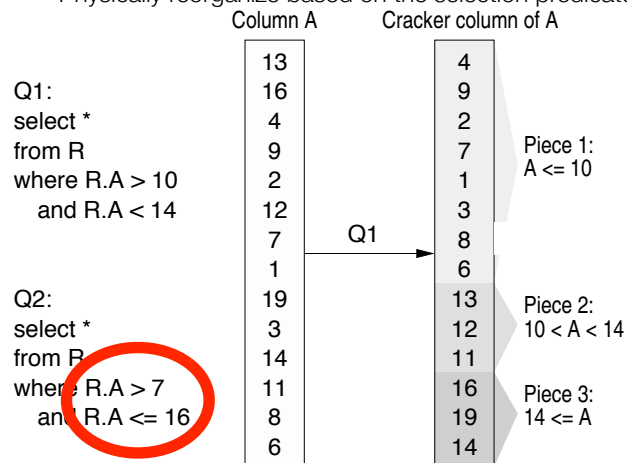
Dynamically/on-the-fly within the select-operator



# Cracking Example

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

Physically reorganize based on the selection predicate



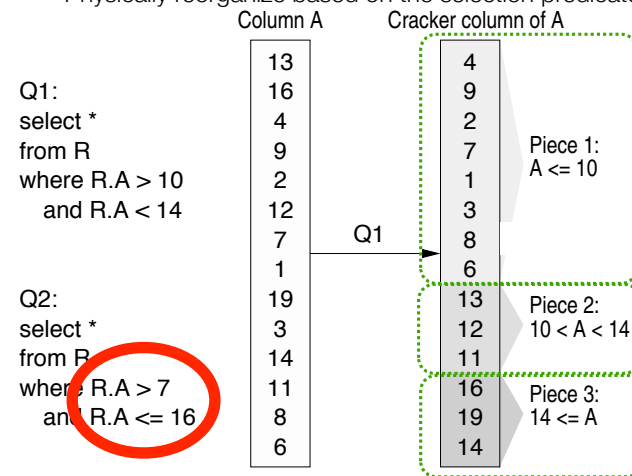
Dynamically/on-the-fly within the select-operator



# Cracking Example

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

Physically reorganize based on the selection predicate



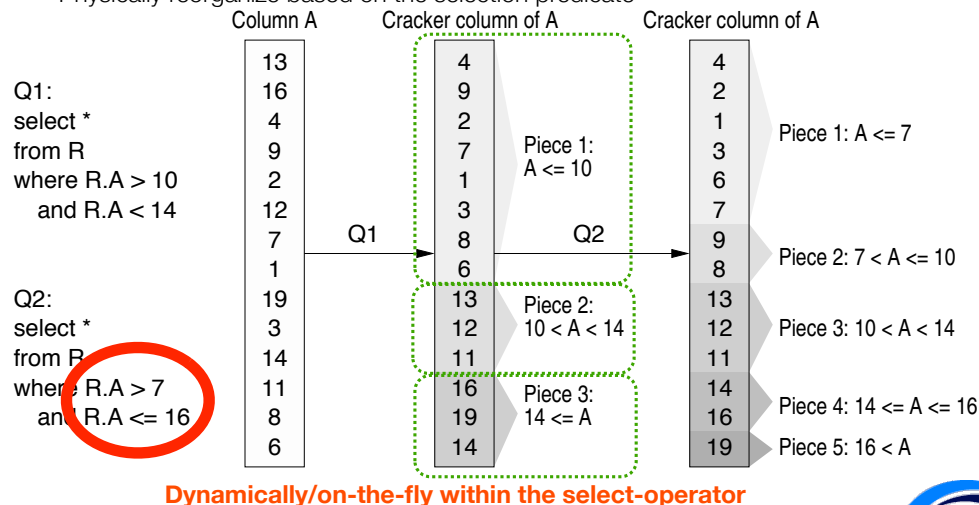
Dynamically/on-the-fly within the select-operator



# Cracking Example

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

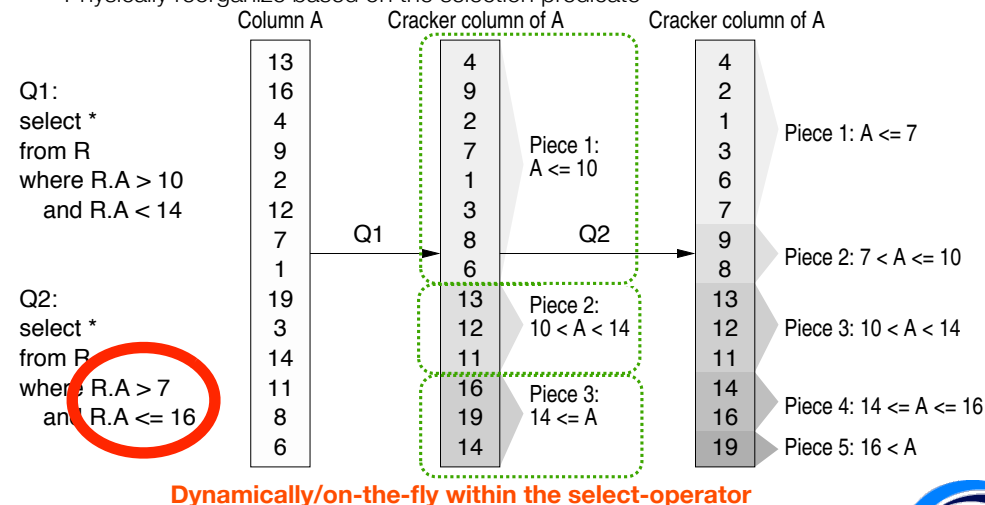
Physically reorganize based on the selection predicate



# Cracking Example

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

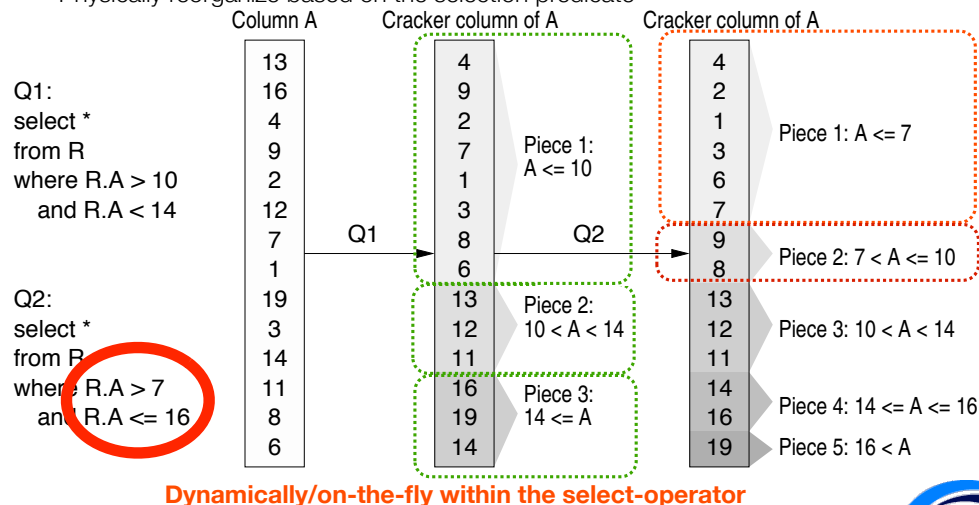
Physically reorganize based on the selection predicate



# Cracking Example

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

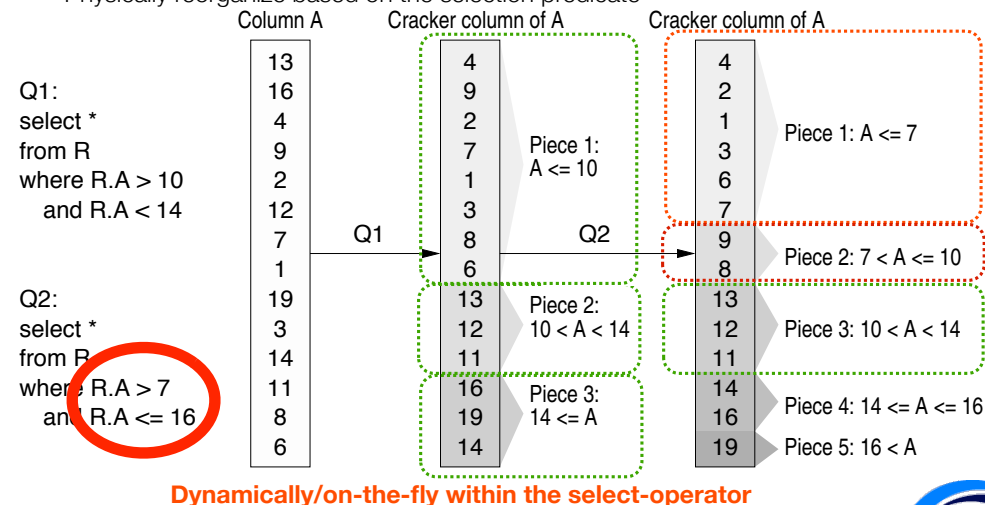
Physically reorganize based on the selection predicate



# Cracking Example

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

Physically reorganize based on the selection predicate

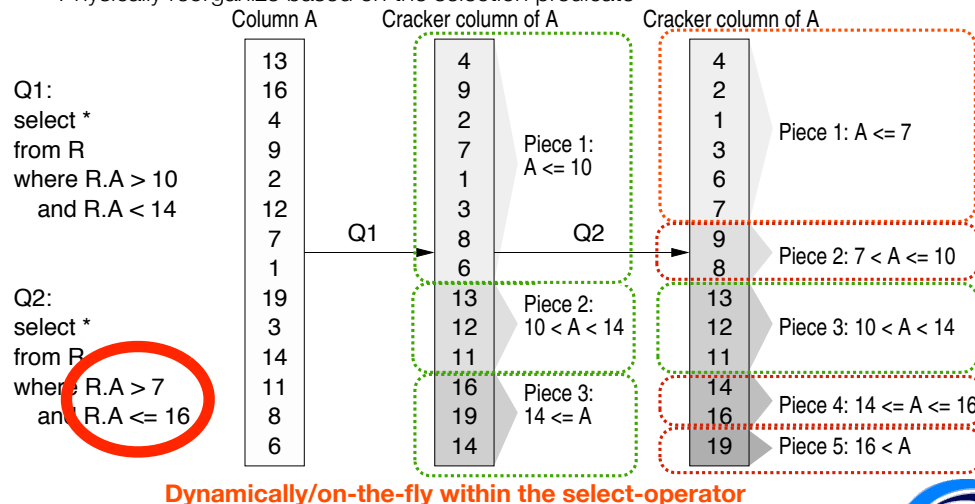




# Cracking Example

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

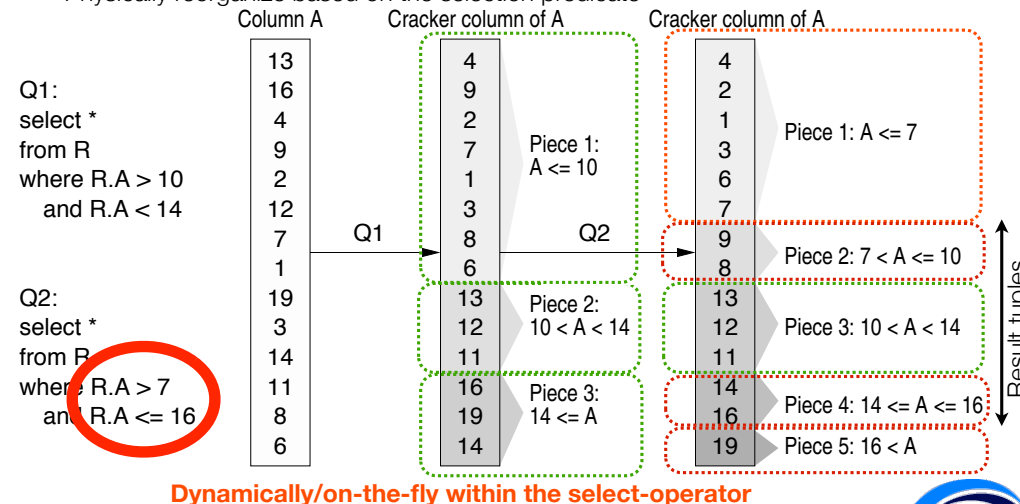
Physically reorganize based on the selection predicate



# Cracking Example

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

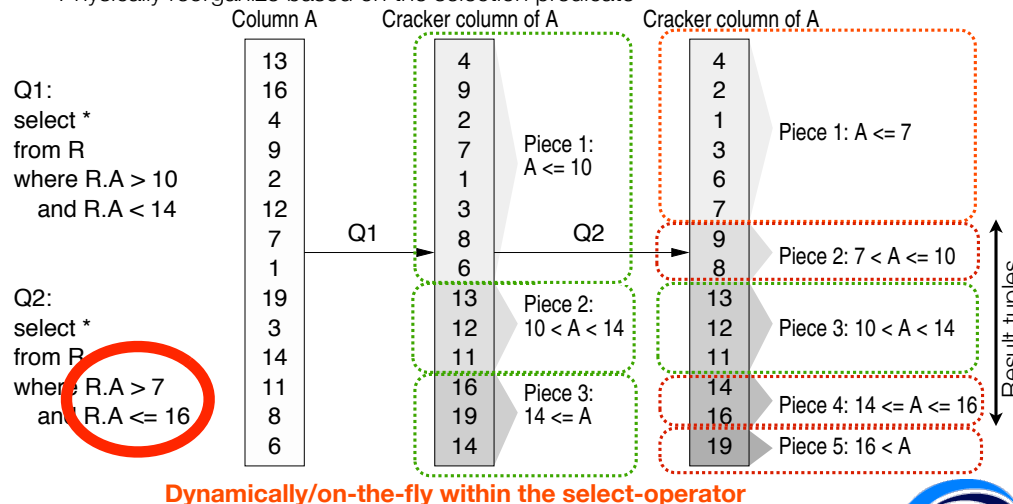
Physically reorganize based on the selection predicate



# Cracking Example

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

Physically reorganize based on the selection predicate



The more we crack, the more we learn

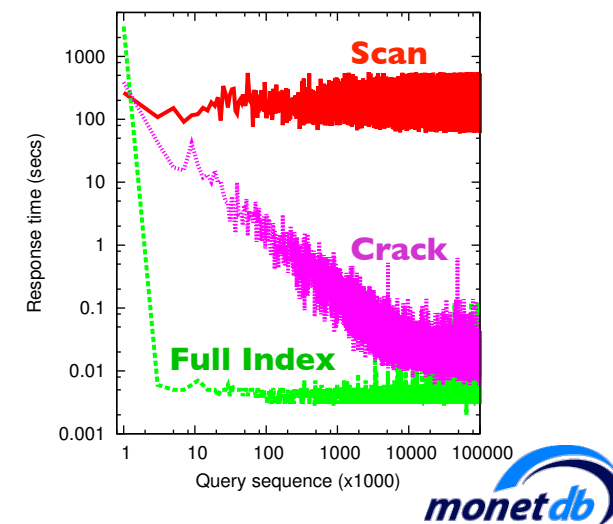


# Cracking Example

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





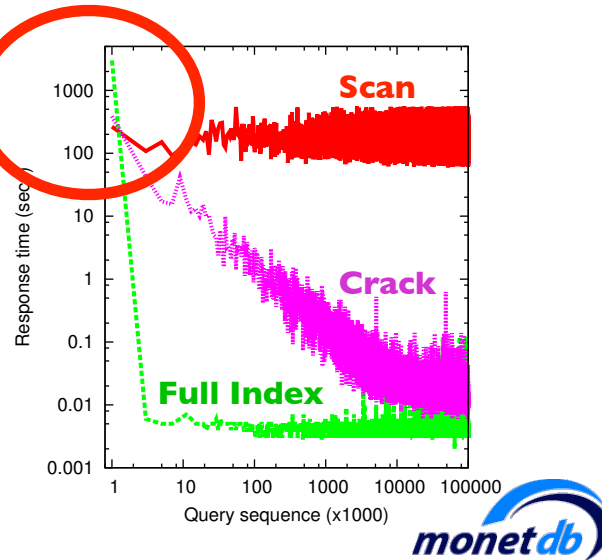
# Cracking Example

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**



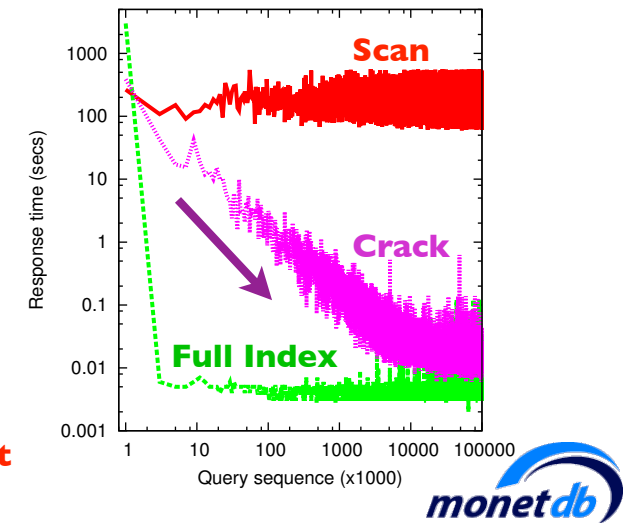
# Cracking Example

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**



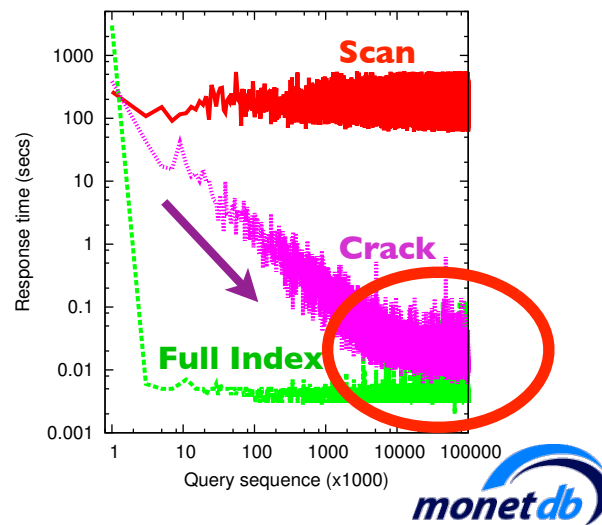
# Cracking Example

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**

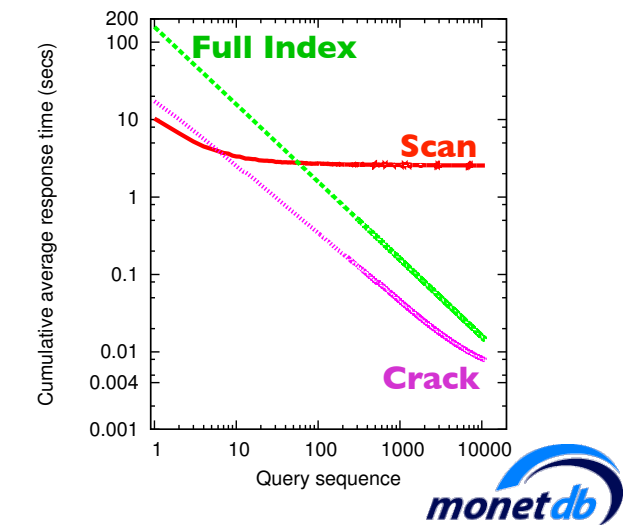


# Cracking Example

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

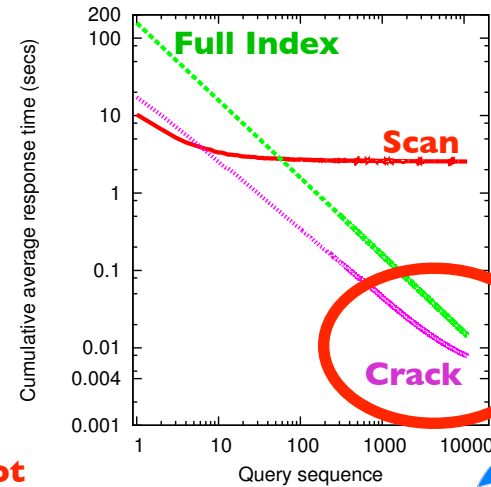


# Cracking Example

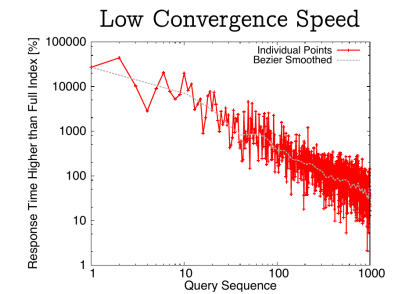
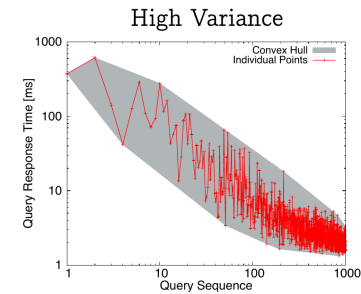
Each query is treated as an advice on how data should be stored

## 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**



[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<sup>\*</sup> Stratos Idréos<sup>†</sup> Panagiotis Karras<sup>‡</sup> Roland H. C. Yap<sup>§</sup>  
<sup>\*</sup>National University of Singapore (halim, ryap)@comp.nus.edu.sg  
<sup>†</sup>CWI, Amsterdam idreos@cwi.nl  
<sup>‡</sup>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 literally 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

<sup>\*</sup> Database research has led us to reexamine established assumptions in order to meet the new challenges posed by big data, scientific databases, highly dynamic, distributed, and multi-core CPU

<sup>†</sup>Work supported by Singapore's MOE AcRF grant T1 251RES08007.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were limited to present their results at the 38th International Conference on Very Large Data Bases, August 27th - 31st 2012, Istanbul, Turkey. *Proceedings of the VLDB Endowment*, Vol. 5, No. 6, Copyright 2012 VLDB Endowment 2150-8097/1202... \$ 10.00.

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 Idréos, Panagiotis Karras and Roland Y. Chuan



# Workload Robustness

**Observation:**

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

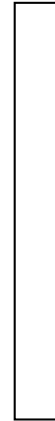
**Goal:**

Maintain adaptive behavior regardless of query input

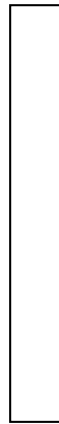
# Query patterns

column with 100 unique integers

Good pattern



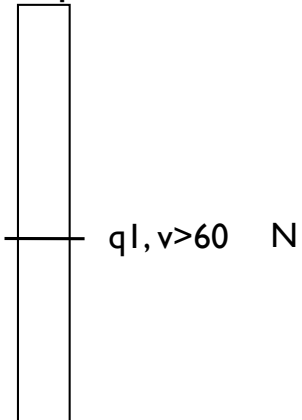
Bad pattern



# Query patterns

column with 100 unique integers

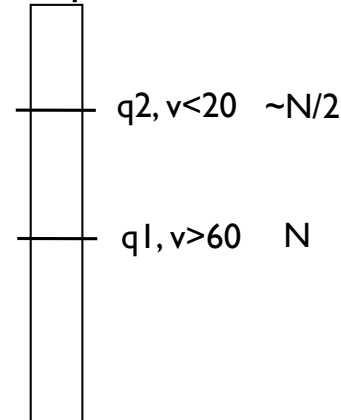
Good pattern



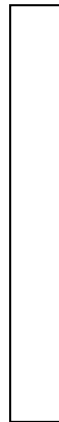
Bad pattern



Good pattern



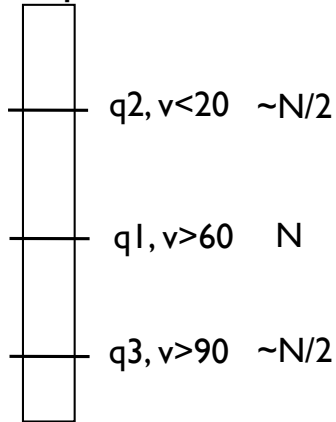
Bad pattern



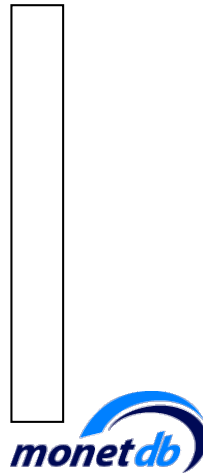
# Query patterns

column with 100 unique integers

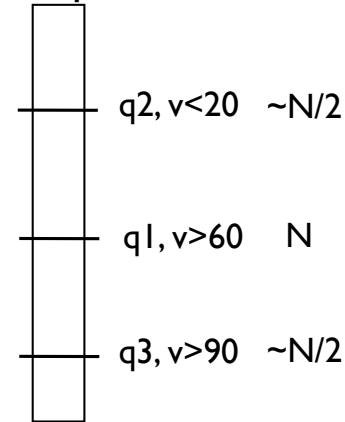
Good pattern



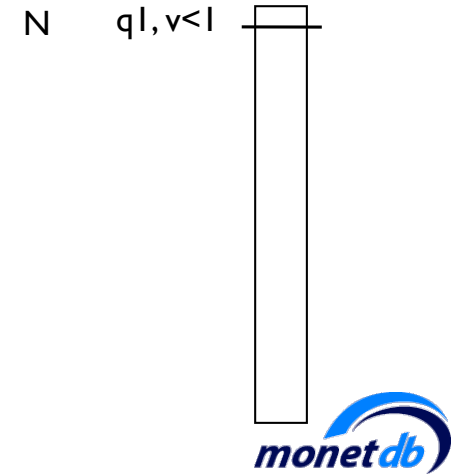
Bad pattern



Good pattern



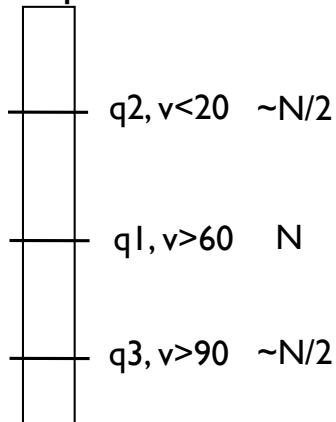
Bad pattern



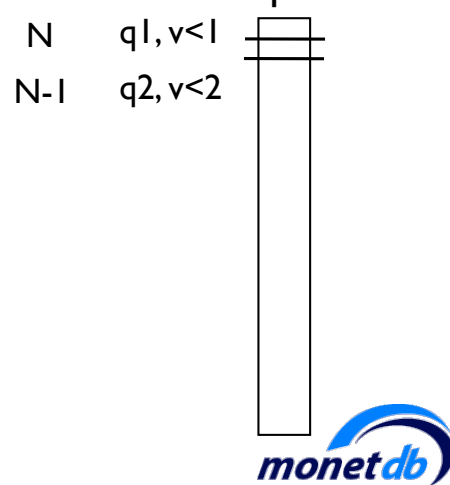
# Query patterns

column with 100 unique integers

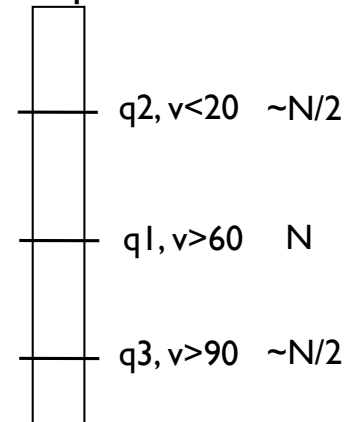
Good pattern



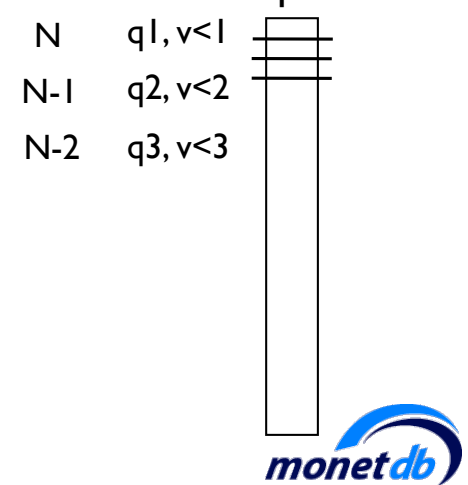
Bad pattern



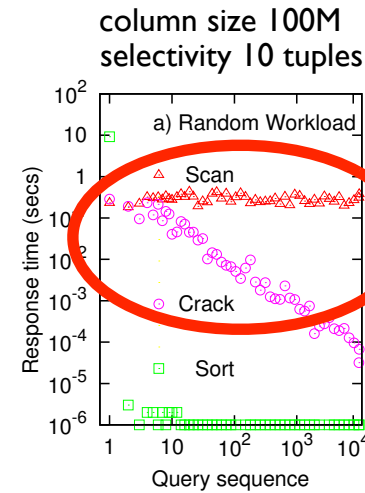
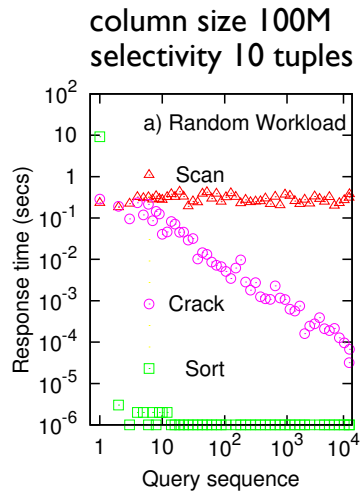
Good pattern



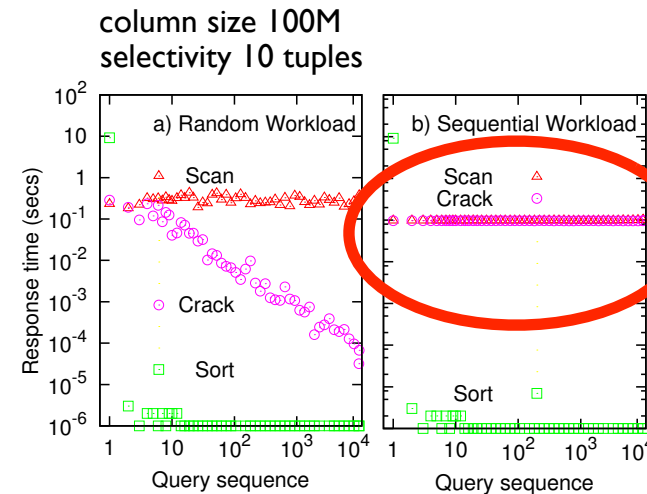
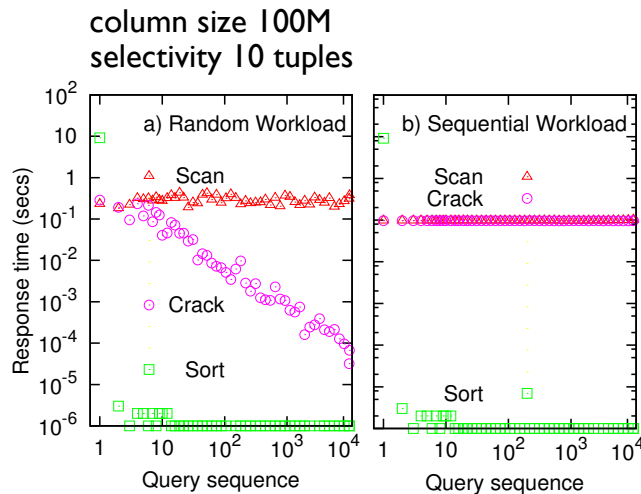
Bad pattern



# Query patterns

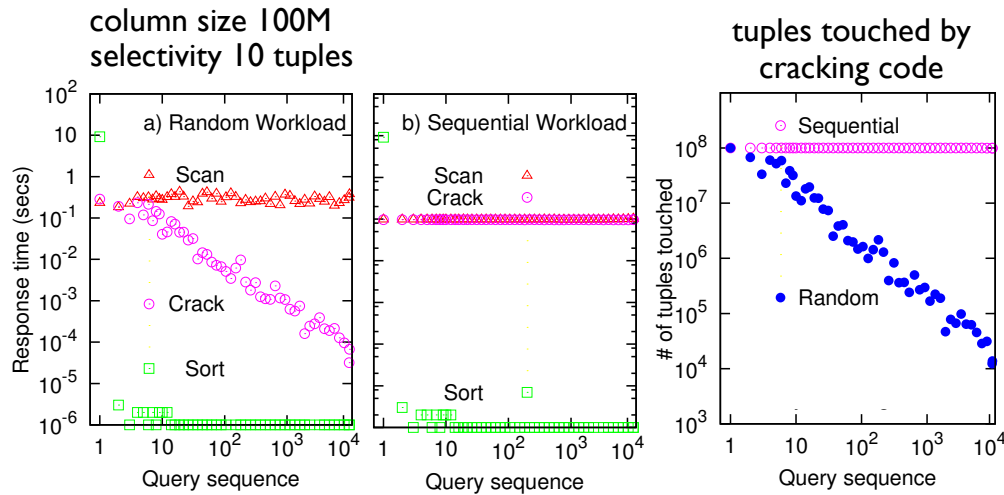


# Query patterns



performance degrades to scan

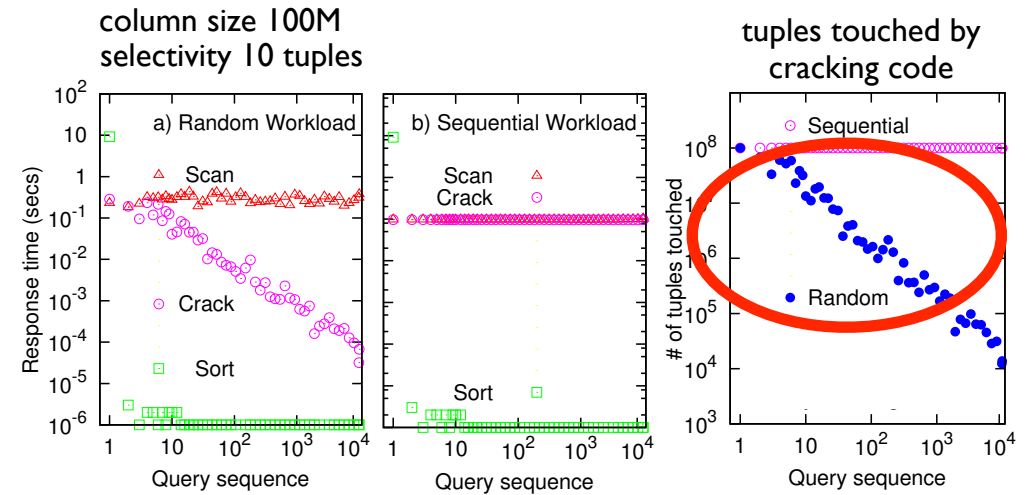
# Query patterns



performance degrades to scan



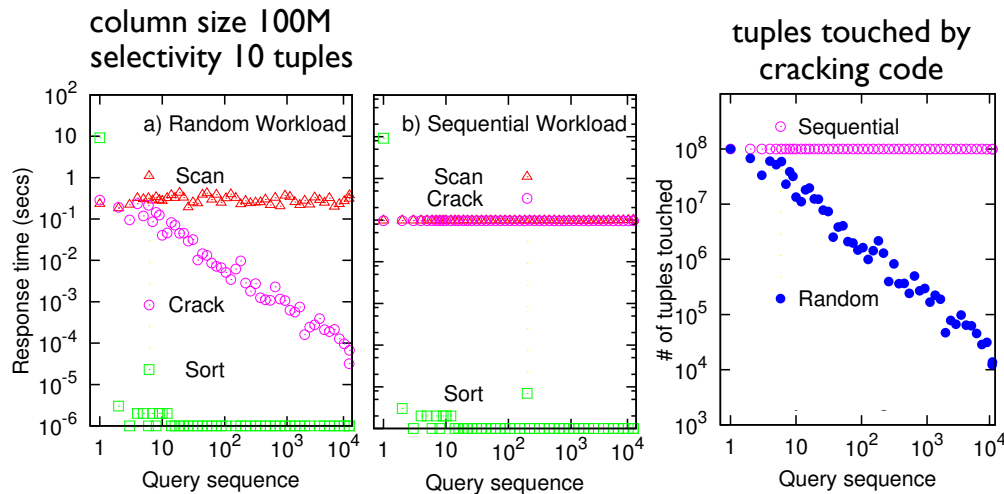
# Query patterns



performance degrades to scan



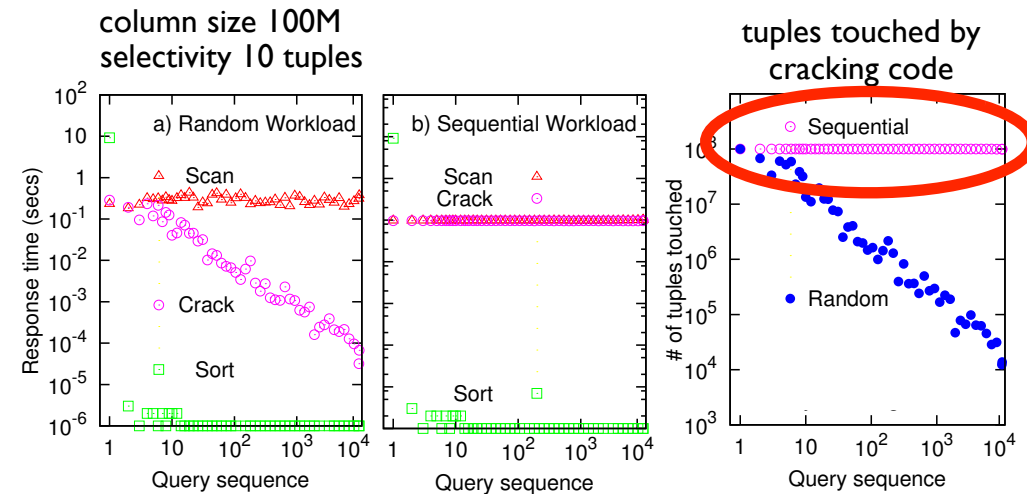
# Query patterns



performance degrades to scan



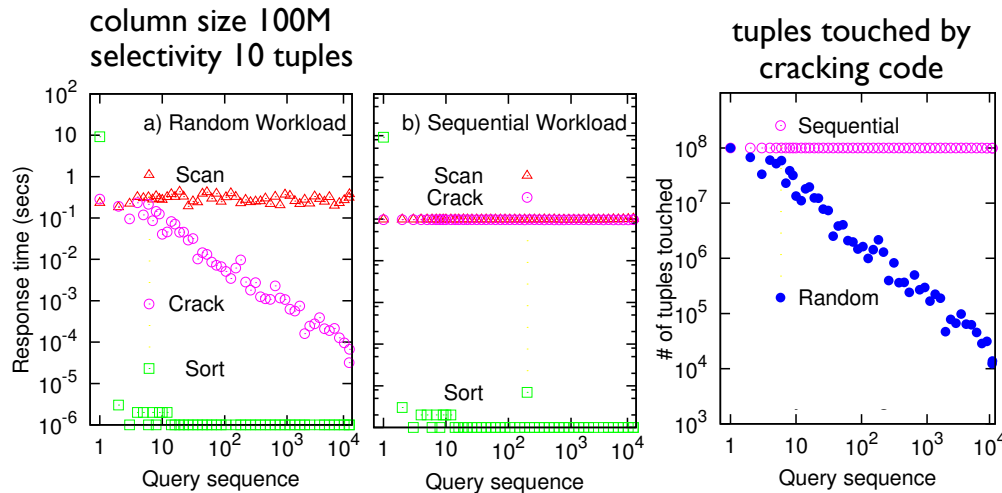
# Query patterns



performance degrades to scan



# Query patterns

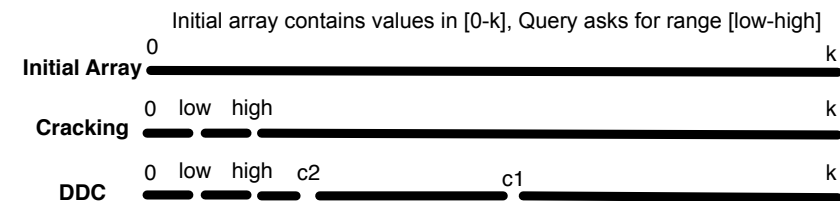
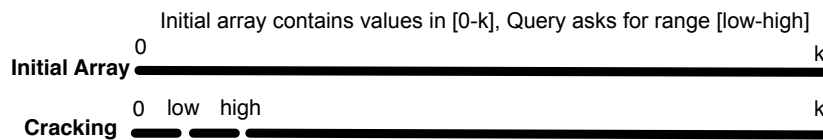


performance degrades to scan



# Stochastic Cracking

# Stochastic Cracking

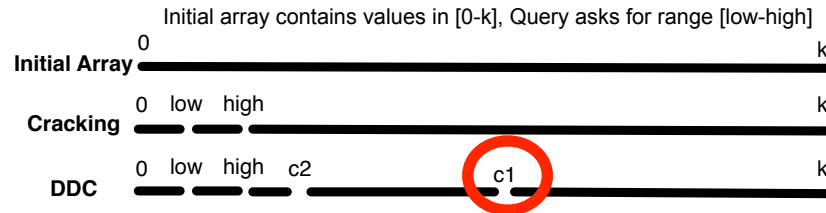


## Data Driven, Center (DDC):

1. Recursively crack a piece in exactly half until in L2 cache.
2. Then crack for the query bounds.



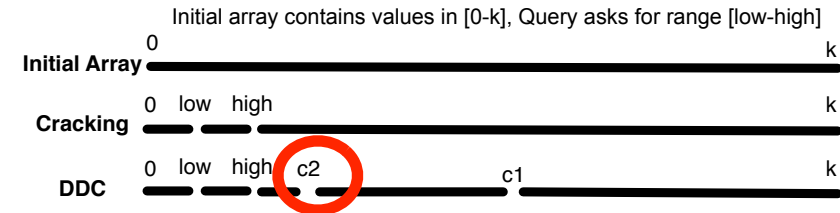
# Stochastic Cracking



## Data Driven, Center (DDC):

1. Recursively crack a piece in exactly half until in L2 cache.
2. Then crack for the query bounds.

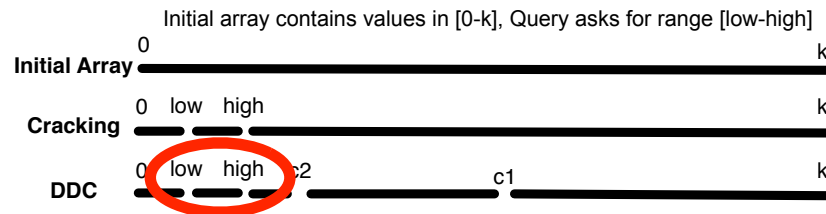
# Stochastic Cracking



## Data Driven, Center (DDC):

1. Recursively crack a piece in exactly half until in L2 cache.
2. Then crack for the query bounds.

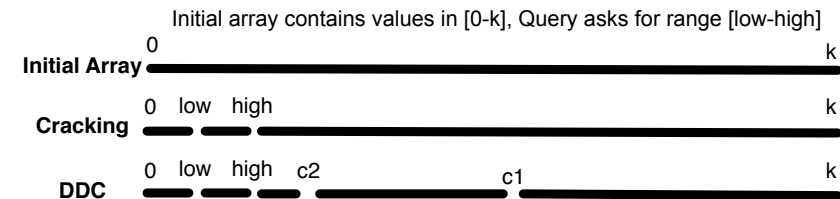
# Stochastic Cracking



## 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

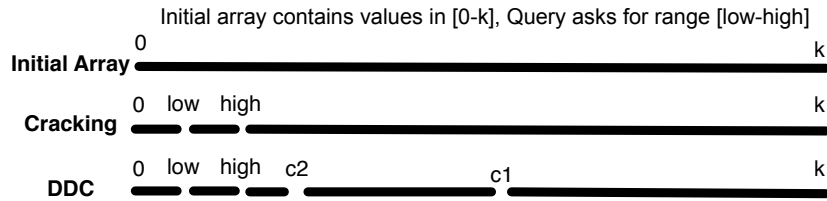


## Data Driven, Center (DDC):

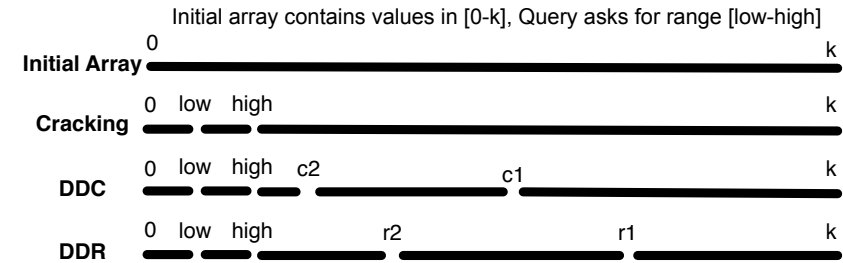
1. Recursively crack a piece in exactly half until in L2 cache.
2. Then crack for the query bounds.



# Stochastic Cracking



# Stochastic Cracking

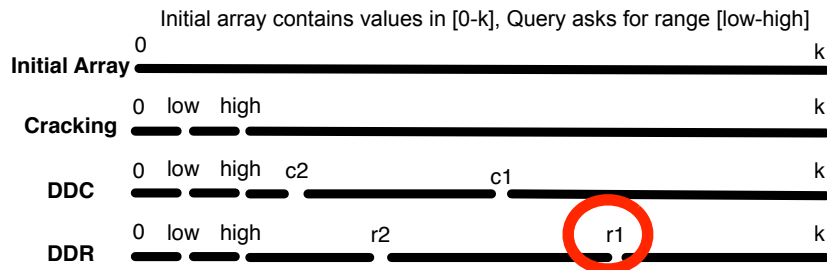


## Data Driven, Random (DDR):

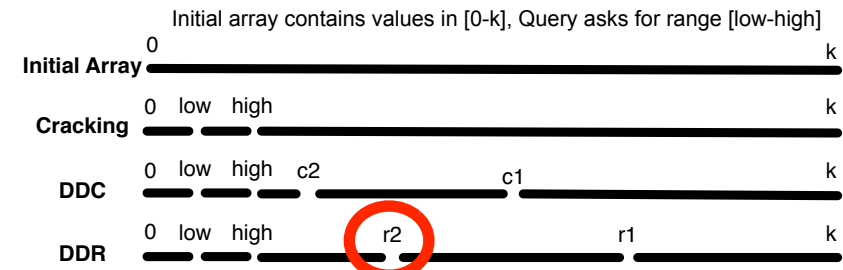
1. Recursively crack a piece randomly until in L2 cache.
2. Then crack for the query bounds.



# Stochastic Cracking



# Stochastic Cracking



## Data Driven, Random (DDR):

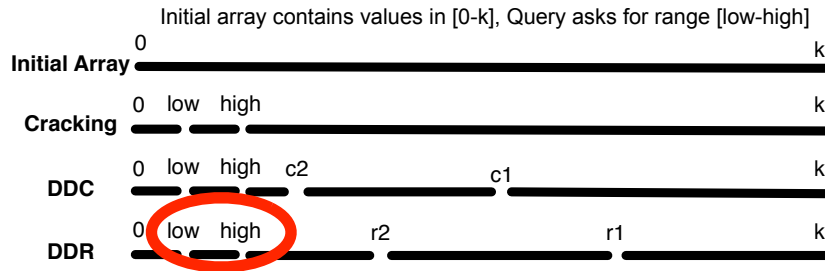
1. Recursively crack a piece randomly until in L2 cache.
2. Then crack for the query bounds.



## Data Driven, Random (DDR):

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

# Stochastic Cracking

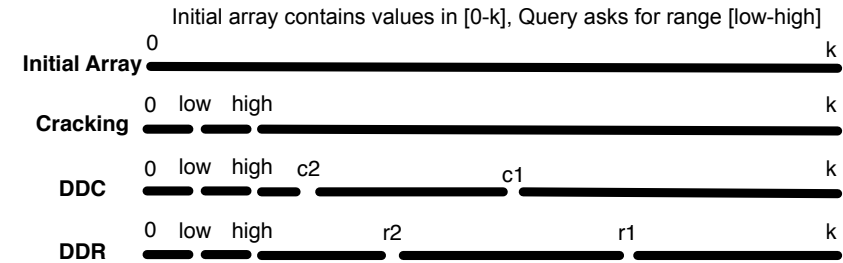


## Data Driven, Random (DDR):

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



# Stochastic Cracking

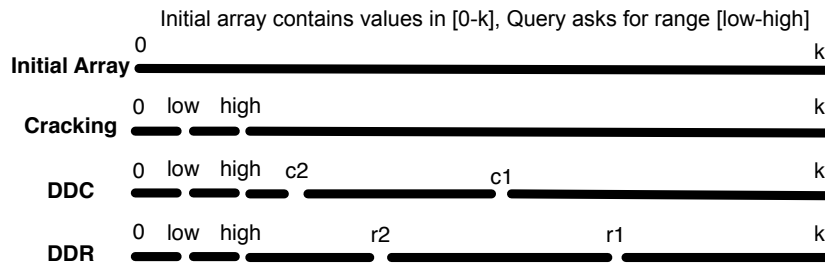


## Data Driven, Random (DDR):

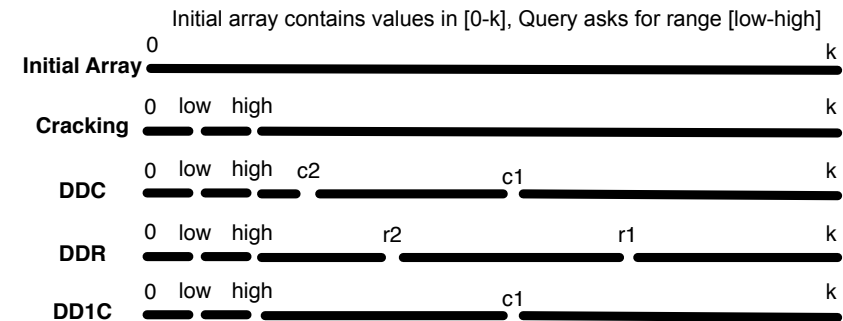
1. Recursively crack a piece randomly until in L2 cache.
2. Then crack for the query bounds.



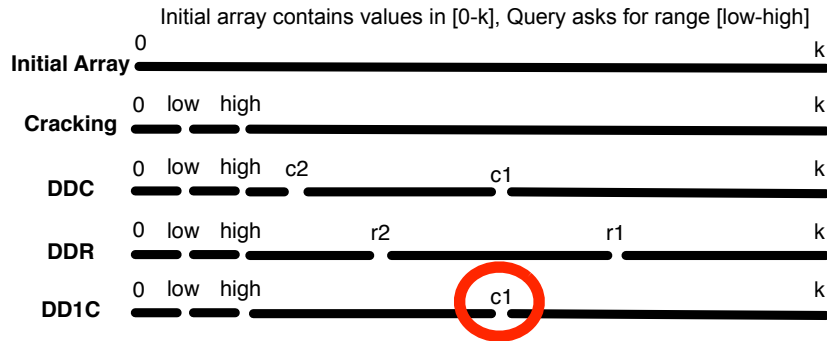
# Stochastic Cracking



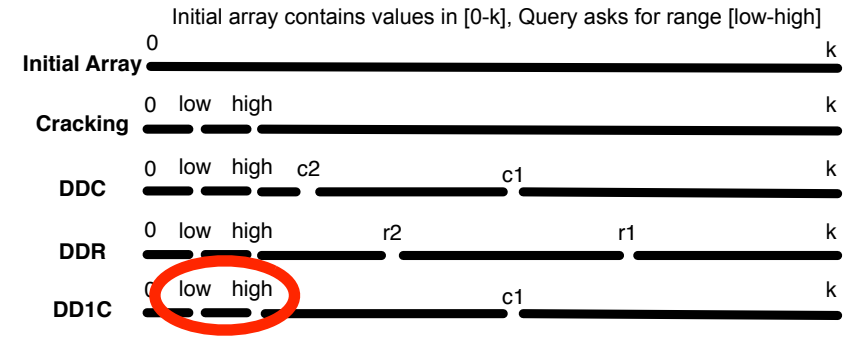
# Stochastic Cracking



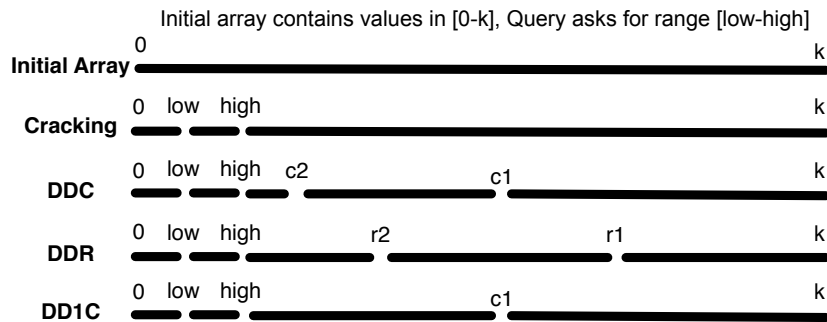
# Stochastic Cracking



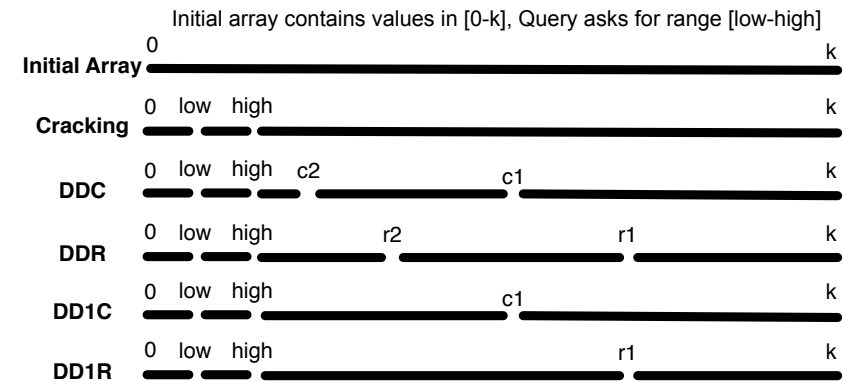
# Stochastic Cracking



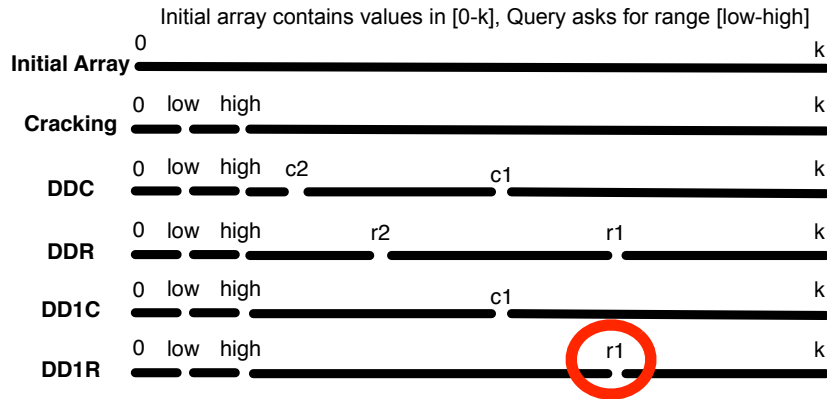
# Stochastic Cracking



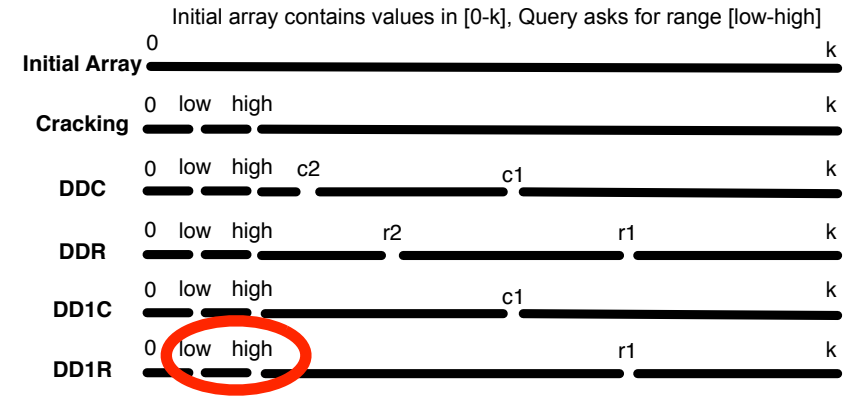
# Stochastic Cracking



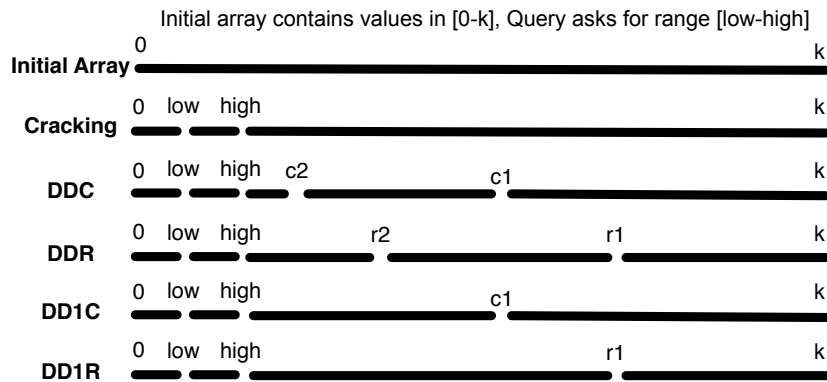
# Stochastic Cracking



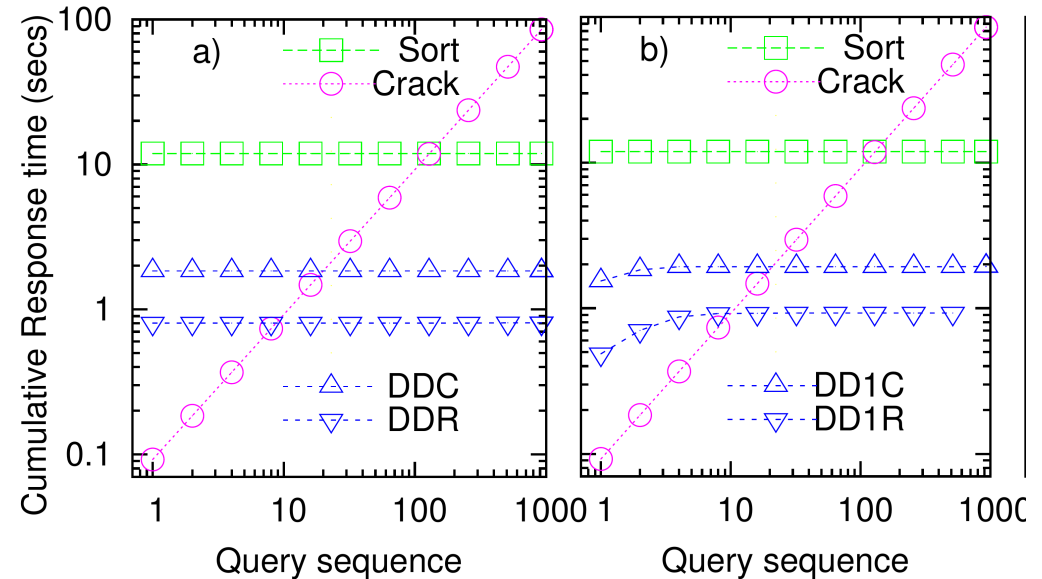
# Stochastic Cracking



# Stochastic Cracking



# Stochastic Cracking



# Adaptive Merging

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

*Incremental sort via external merge sort steps*

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



Hybrids PVLDB 2011

# Adaptive Merging

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

*Incremental sort via external merge sort steps*



Hybrids PVLDB 2011

# Adaptive Merging

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

*Incremental sort via external merge sort steps*

select(A,50,100)

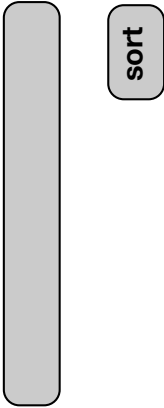


# Adaptive Merging

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

*Incremental sort via external merge sort steps*

select(A,50,100)

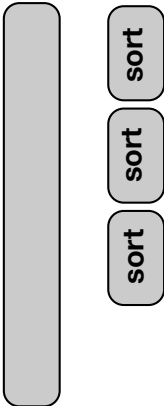


# Adaptive Merging

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

*Incremental sort via external merge sort steps*

select(A,50,100)

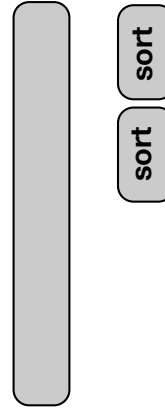


# Adaptive Merging

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

*Incremental sort via external merge sort steps*

select(A,50,100)

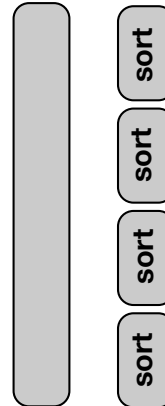


# Adaptive Merging

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

*Incremental sort via external merge sort steps*

select(A,50,100)

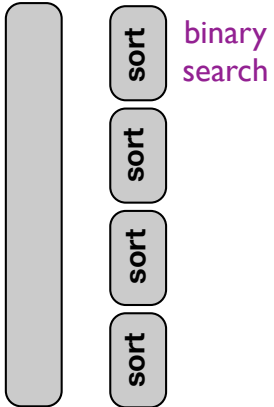


# Adaptive Merging

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

*Incremental sort via external merge sort steps*

select(A,50,100)

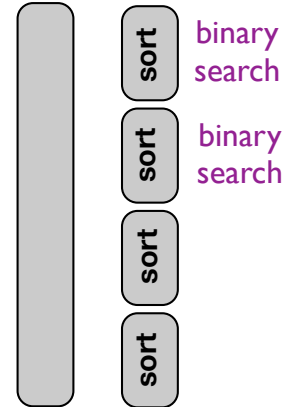


# Adaptive Merging

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

*Incremental sort via external merge sort steps*

select(A,50,100)

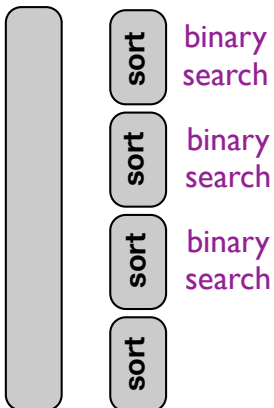


# Adaptive Merging

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

*Incremental sort via external merge sort steps*

select(A,50,100)

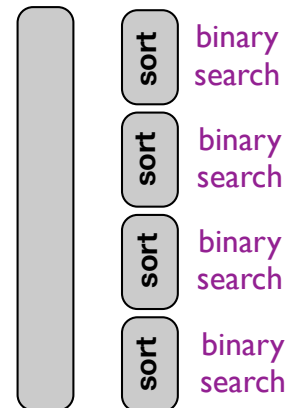


# Adaptive Merging

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

*Incremental sort via external merge sort steps*

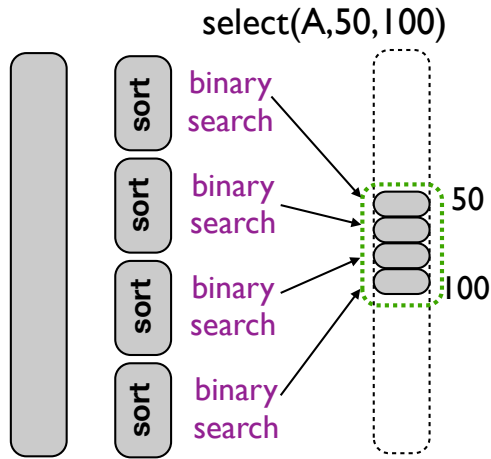
select(A,50,100)



# Adaptive Merging

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

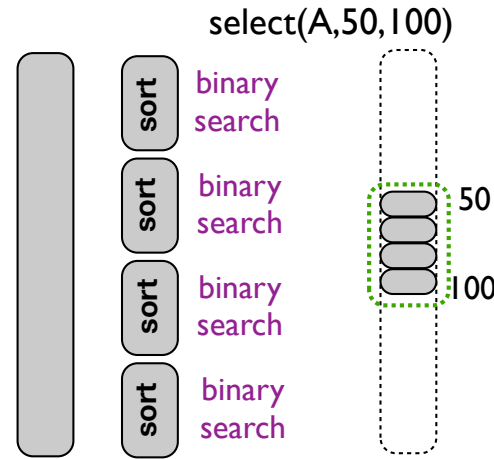
*Incremental sort via external merge sort steps*



# Adaptive Merging

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

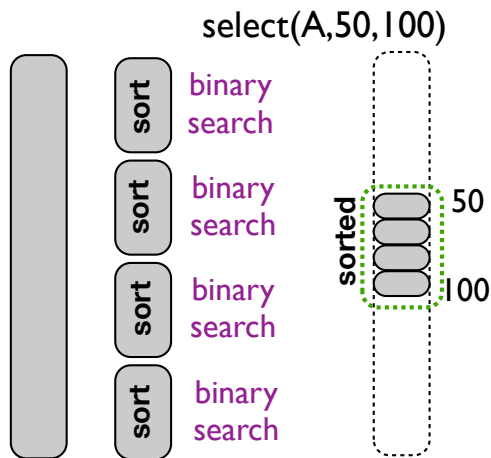
*Incremental sort via external merge sort steps*



# Adaptive Merging

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

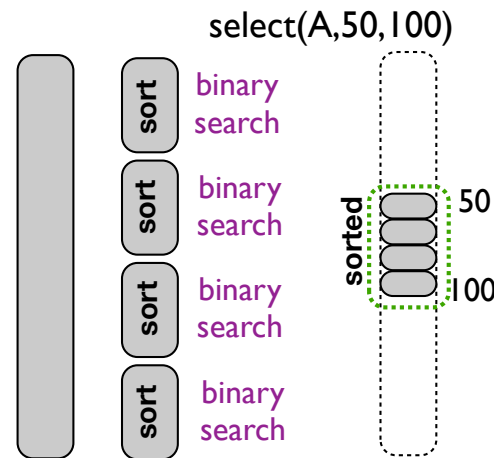
*Incremental sort via external merge sort steps*



# Adaptive Merging

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

*Incremental sort via external merge sort steps*



Initial

Final

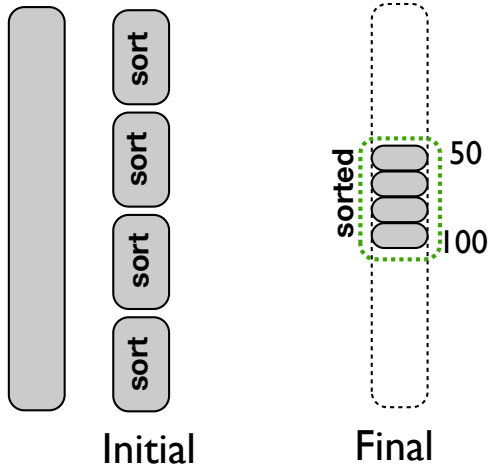


# Adaptive Merging

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

*Incremental sort via external merge sort steps*

select(A,50,100)    select(A,55,70)

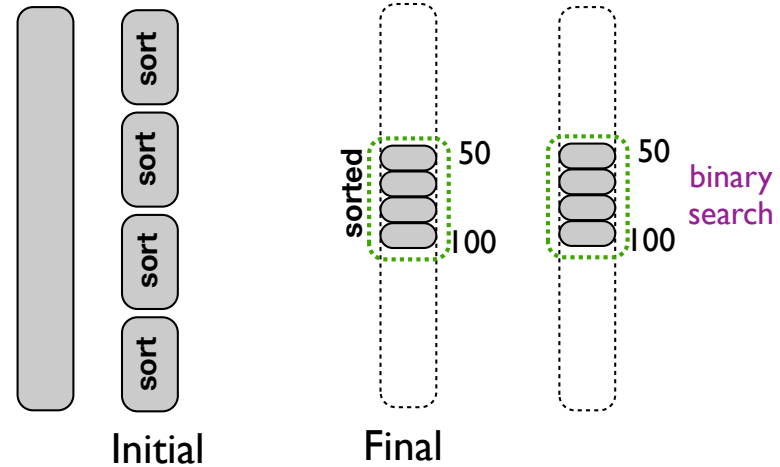


# Adaptive Merging

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

*Incremental sort via external merge sort steps*

select(A,50,100)    select(A,55,70)

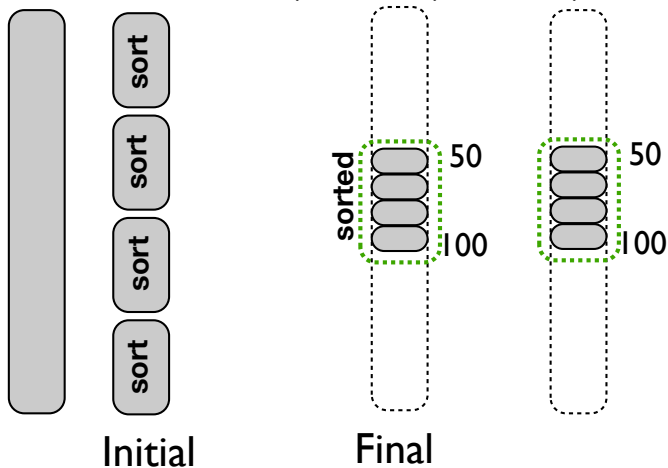


# Adaptive Merging

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

*Incremental sort via external merge sort steps*

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

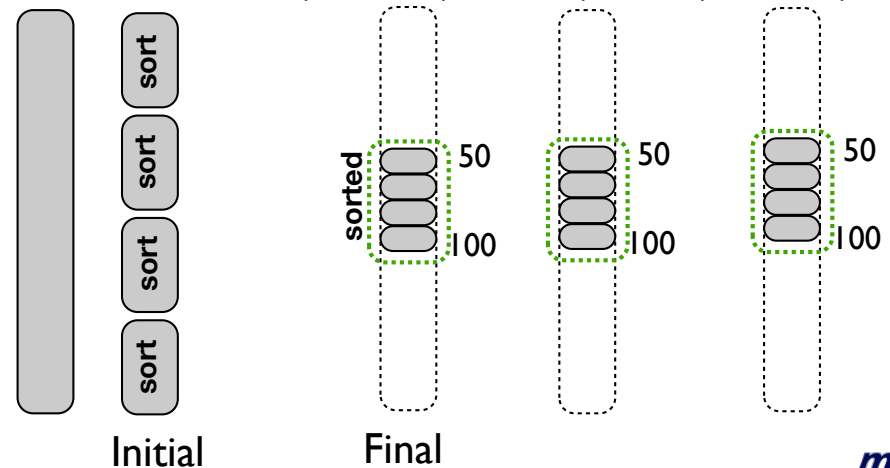


# Adaptive Merging

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

*Incremental sort via external merge sort steps*

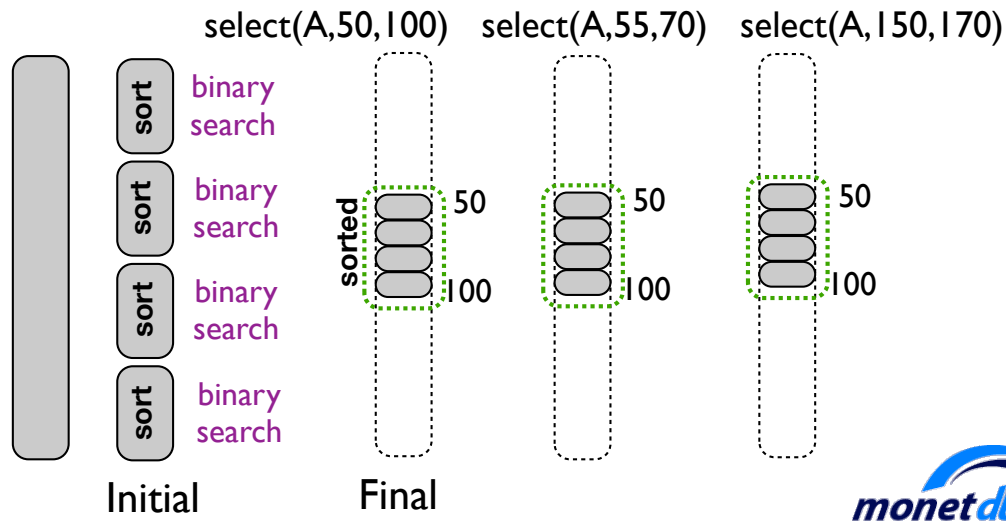
select(A,50,100)    select(A,55,70)    select(A,150,170)



# Adaptive Merging

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

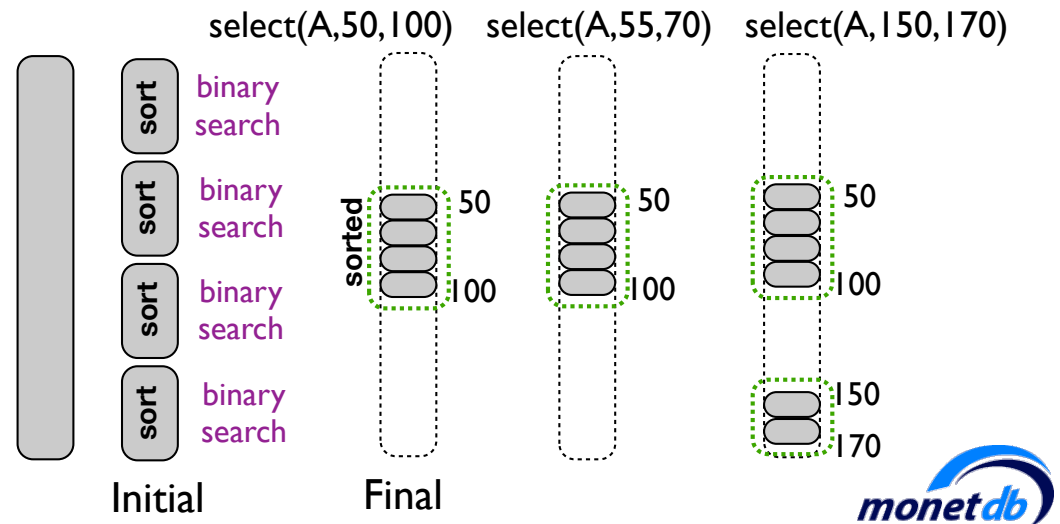
*Incremental sort via external merge sort steps*



# Adaptive Merging

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

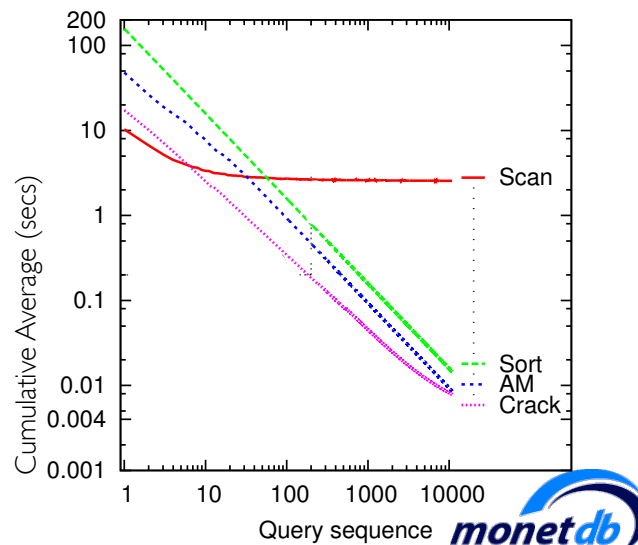
*Incremental sort via external merge sort steps*



## Performance Analysis

### set-up

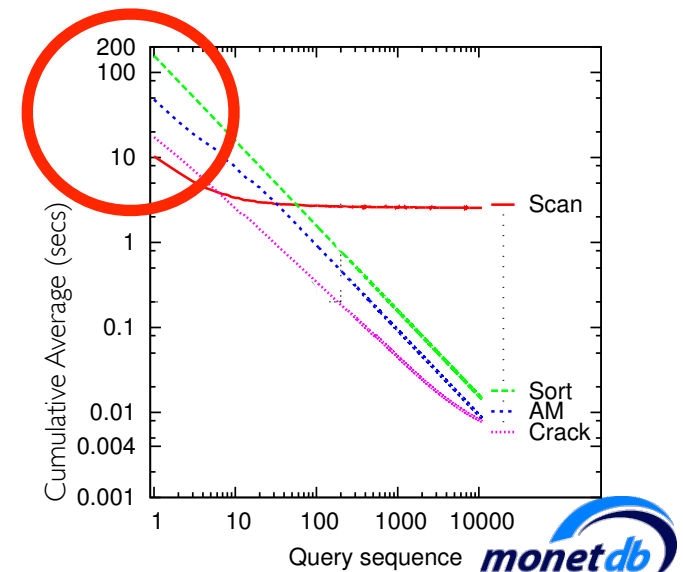
10K random selections  
selectivity 10%  
random value ranges  
in a 30 million integer column



## Performance Analysis

### set-up

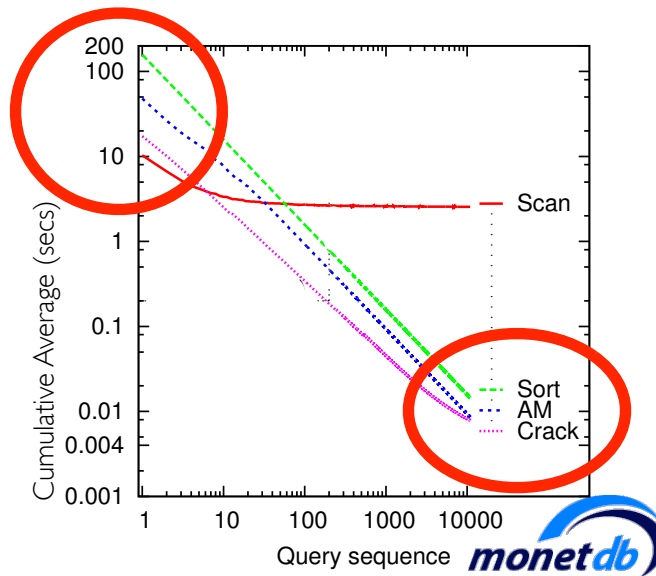
10K random selections  
selectivity 10%  
random value ranges  
in a 30 million integer column



# Performance Analysis

## set-up

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

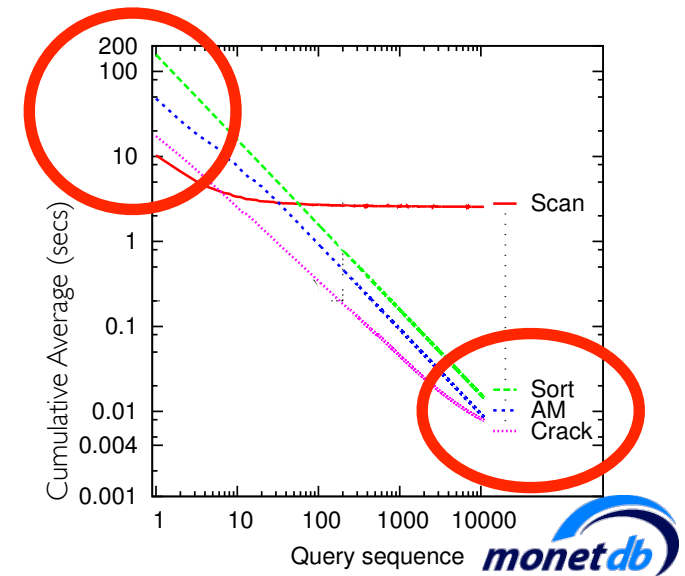


# Performance Analysis

## set-up

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

**AM: high init overhead  
but fast convergence**



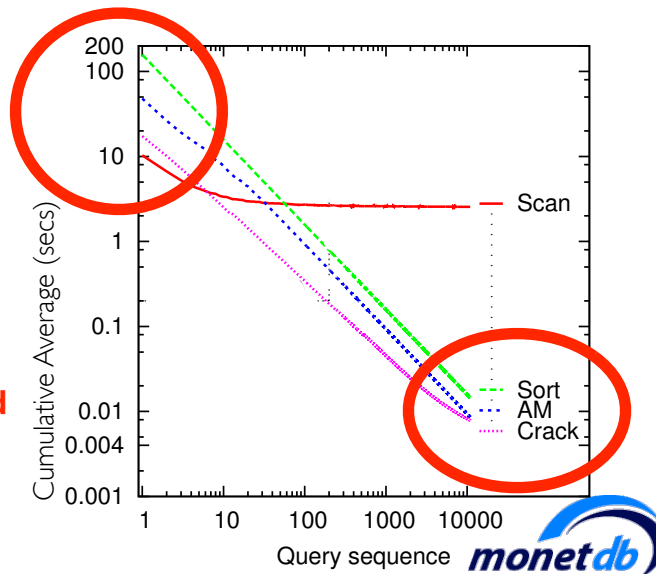
# Performance Analysis

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

# Questions

# Crack-Crack

*vary initialization and incremental steps taken*

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*



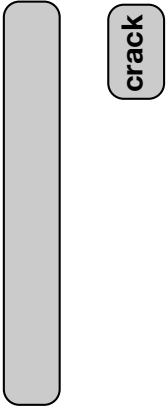
`select(A,50,100)`



# Crack-Crack

*vary initialization and incremental steps taken*

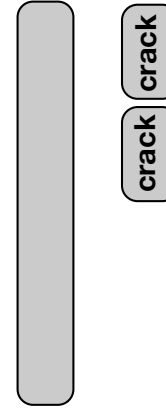
select(A,50,100)



# Crack-Crack

*vary initialization and incremental steps taken*

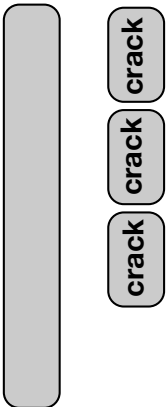
select(A,50,100)



# Crack-Crack

*vary initialization and incremental steps taken*

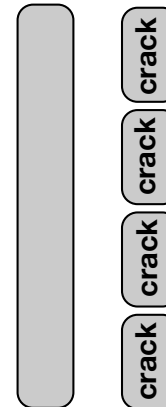
select(A,50,100)



# Crack-Crack

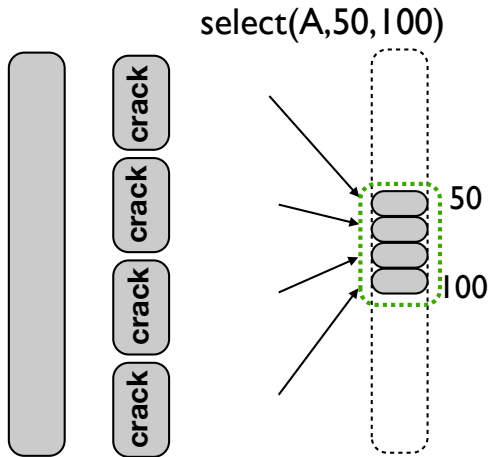
*vary initialization and incremental steps taken*

select(A,50,100)



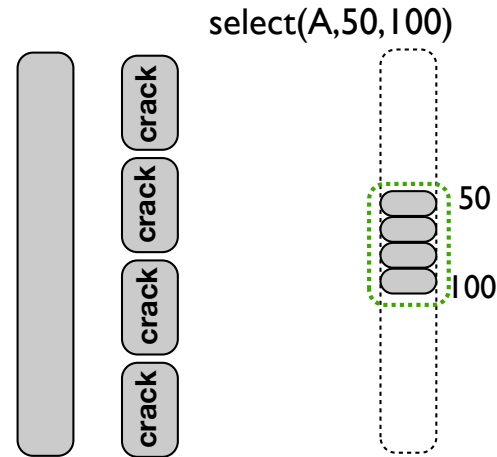
# Crack-Crack

*vary initialization and incremental steps taken*



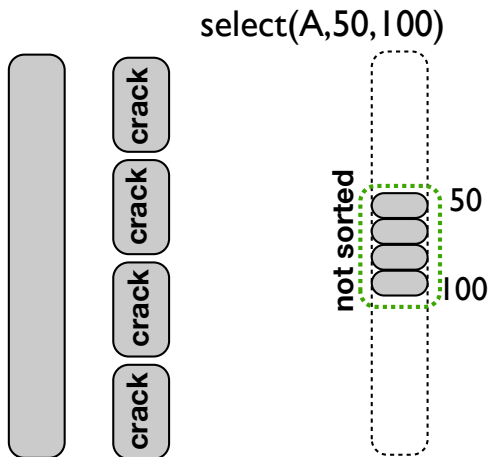
# Crack-Crack

*vary initialization and incremental steps taken*



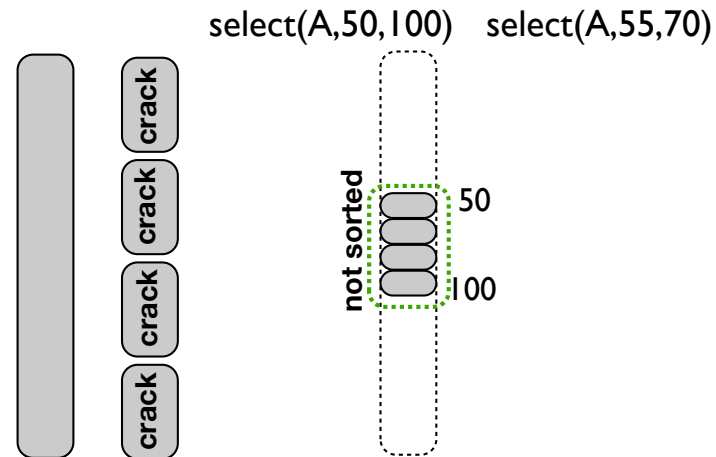
# Crack-Crack

*vary initialization and incremental steps taken*



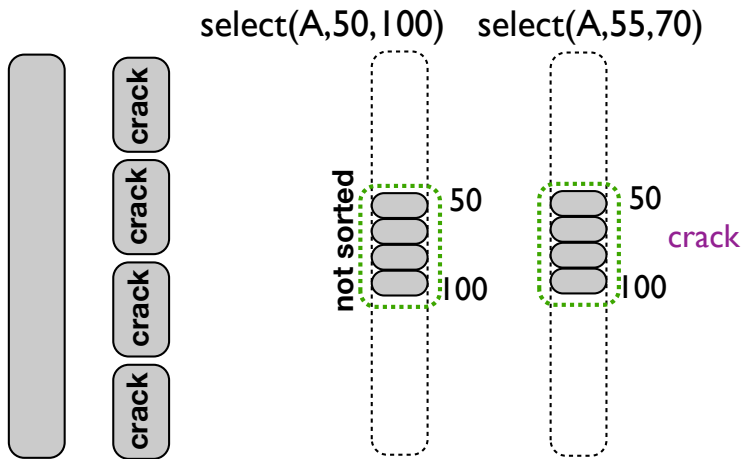
# Crack-Crack

*vary initialization and incremental steps taken*



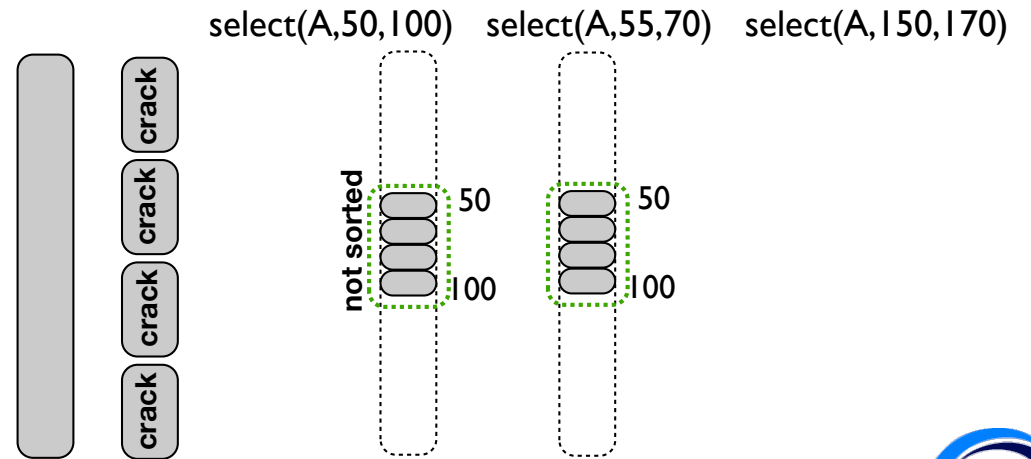
# Crack-Crack

*vary initialization and incremental steps taken*



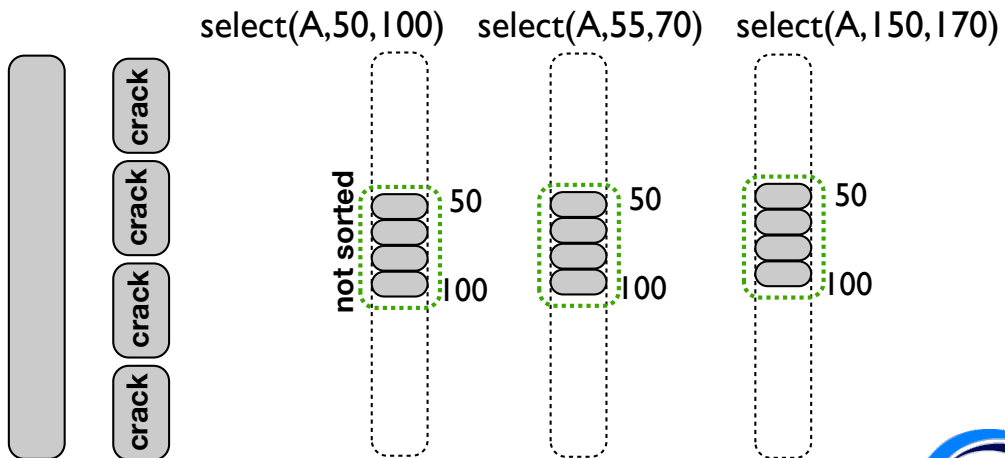
# Crack-Crack

*vary initialization and incremental steps taken*



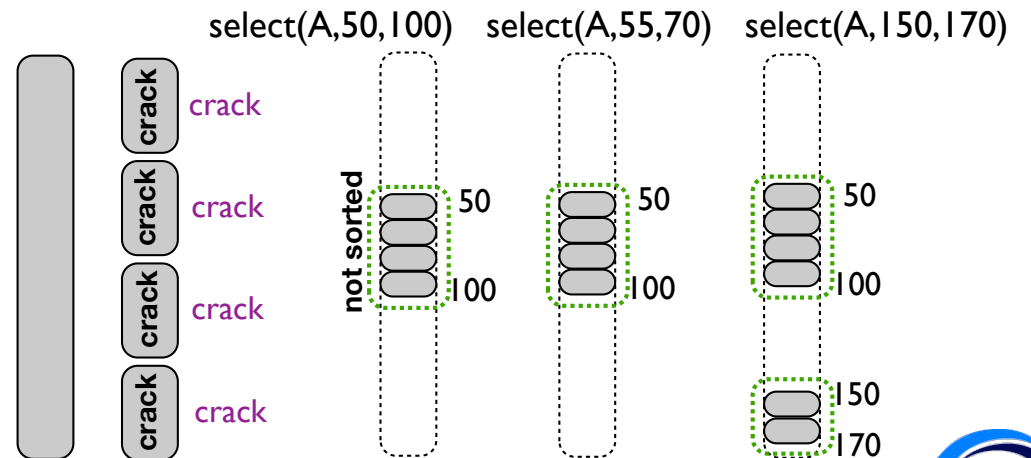
# Crack-Crack

*vary initialization and incremental steps taken*

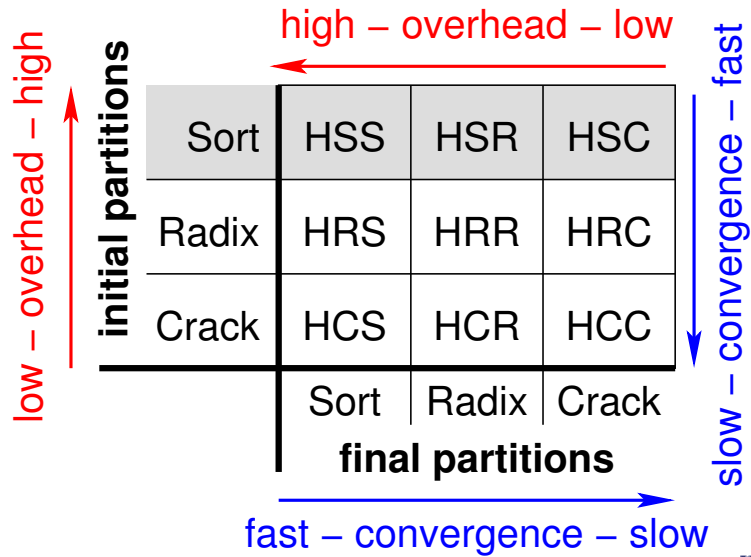


# Crack-Crack

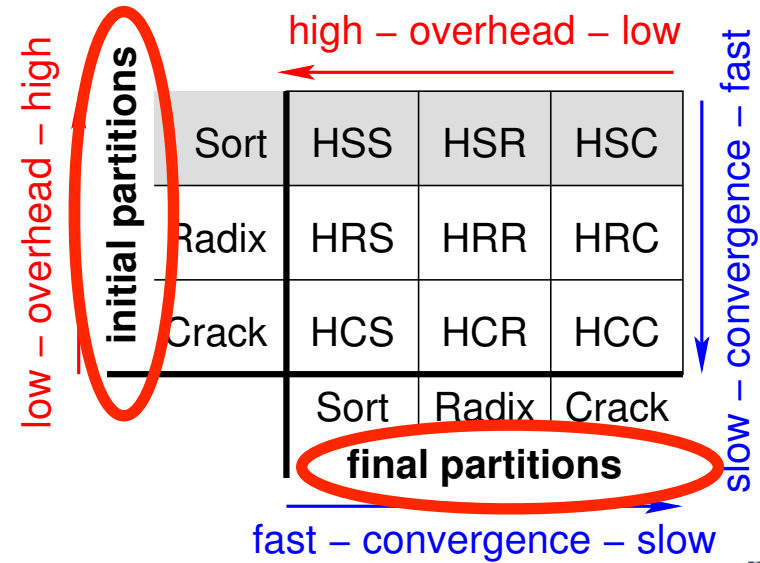
*vary initialization and incremental steps taken*



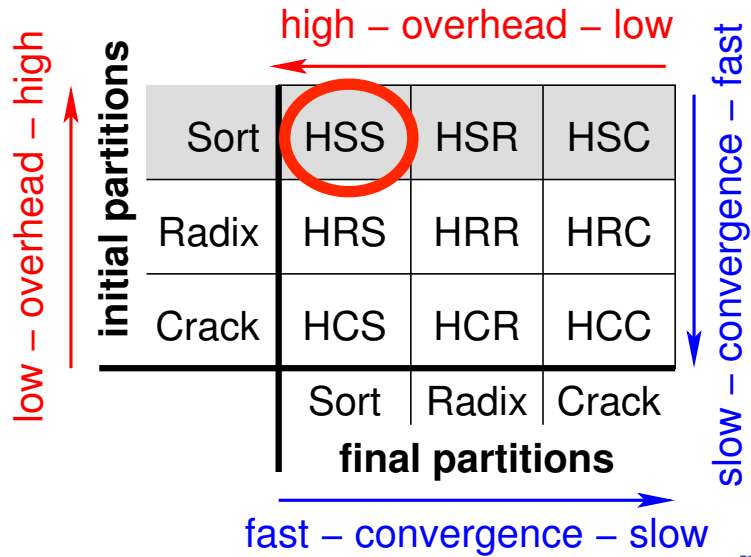
# Adaptive Indexing



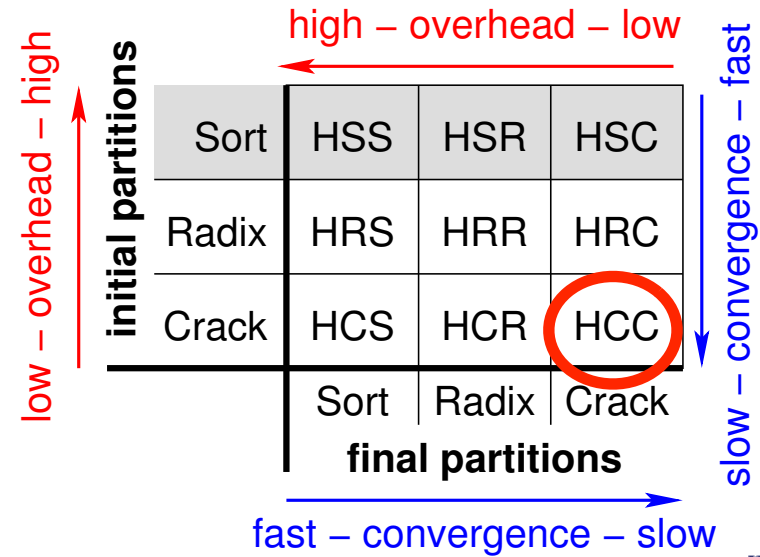
# Adaptive Indexing



# Adaptive Indexing

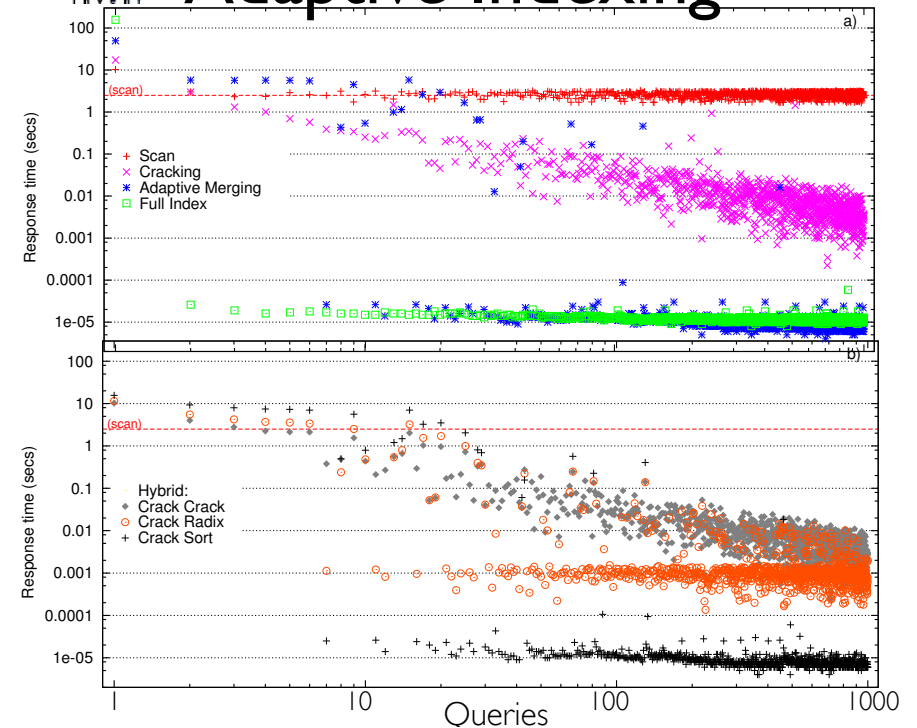
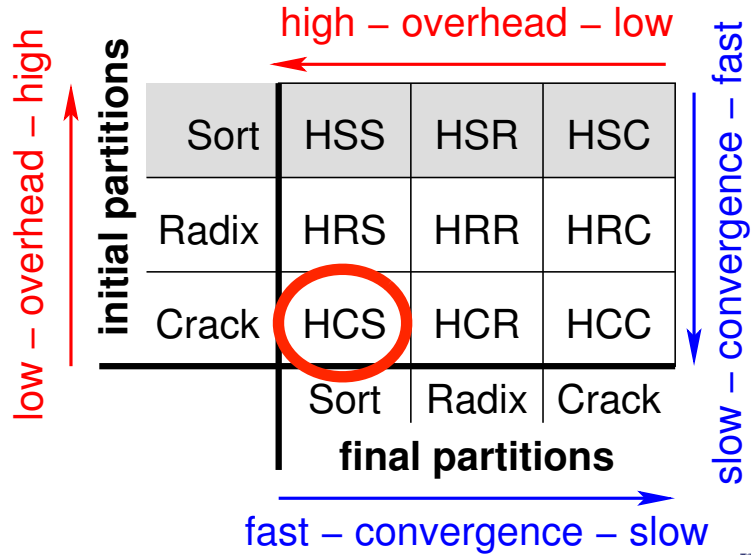


# Adaptive Indexing

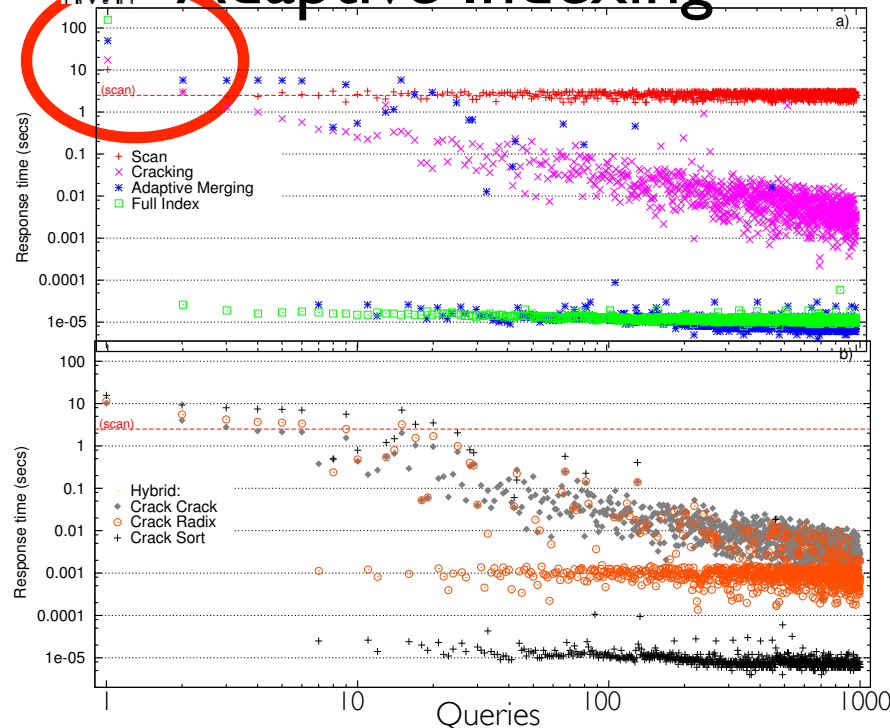




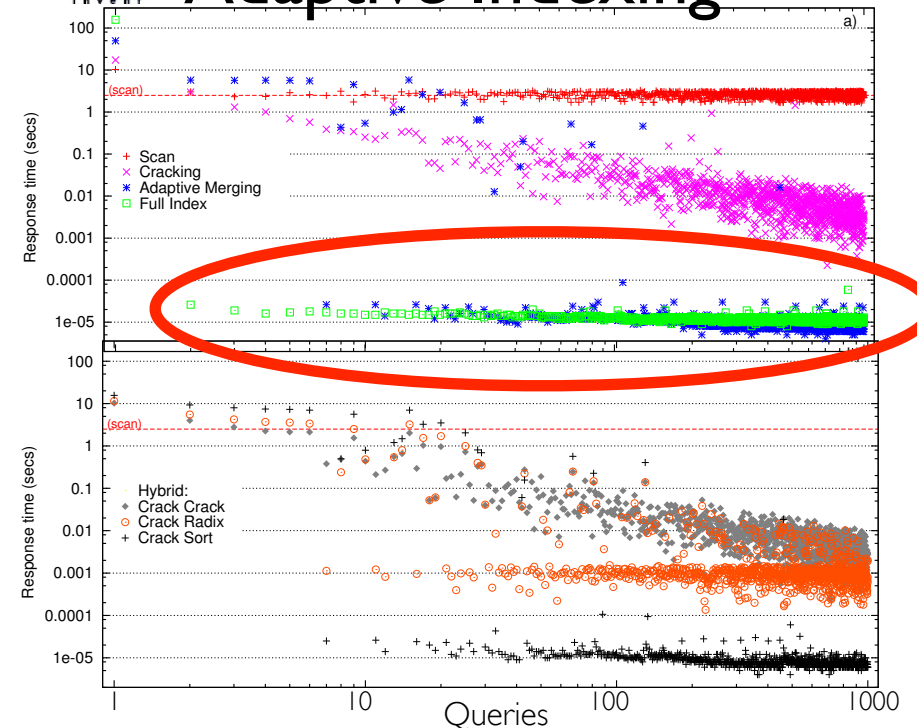
## Adaptive Indexing



## Adaptive Indexing



## Adaptive Indexing

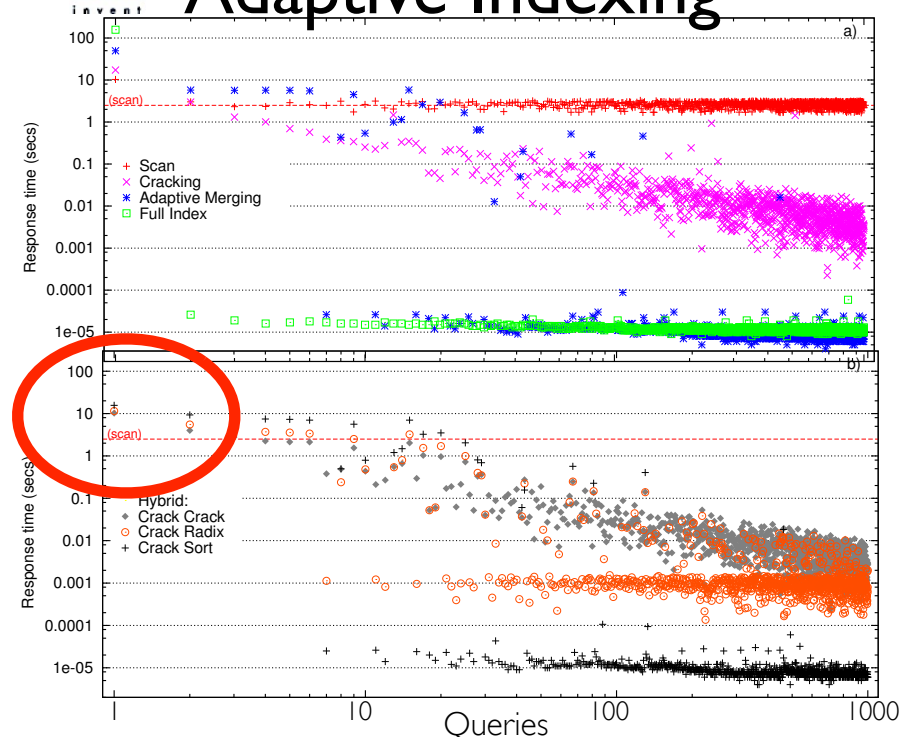
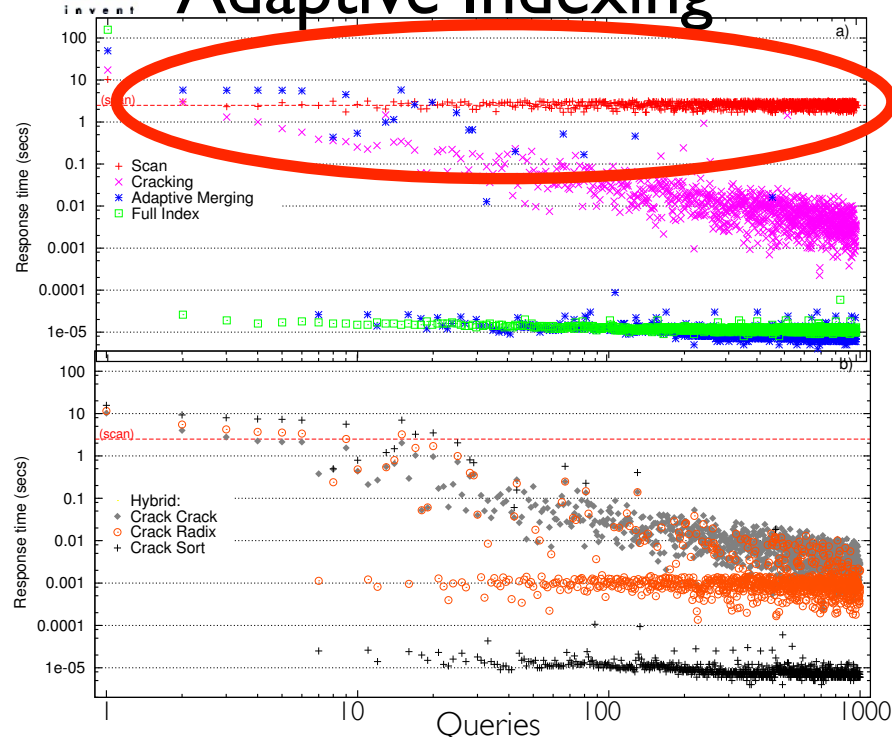


## Adaptive Indexing

Hybrids PVLDB 2011

## Adaptive Indexing

Hybrids PVLDB 2011

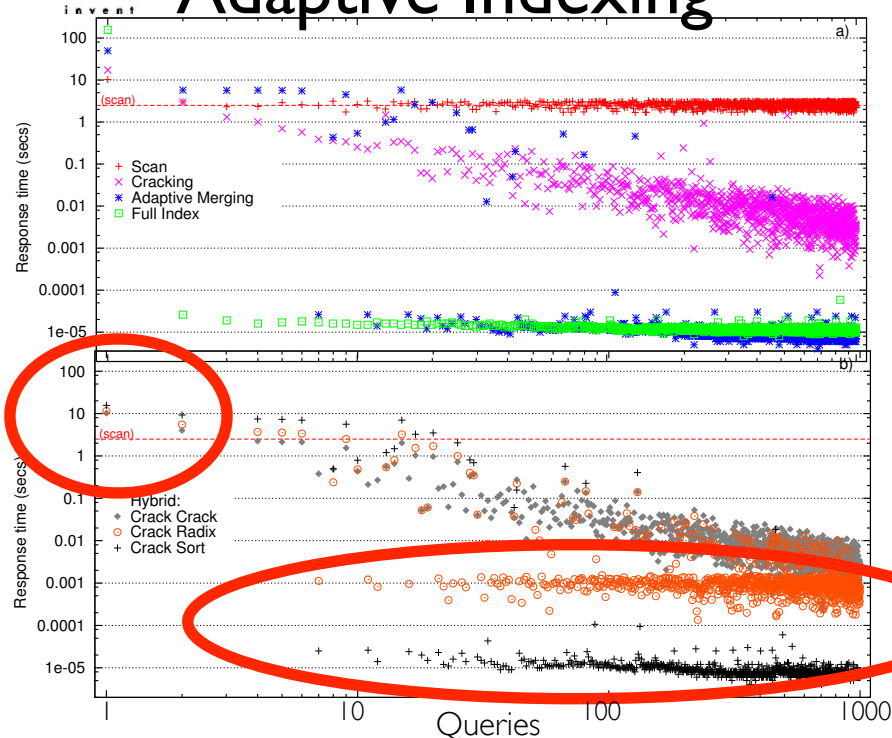


## Adaptive Indexing

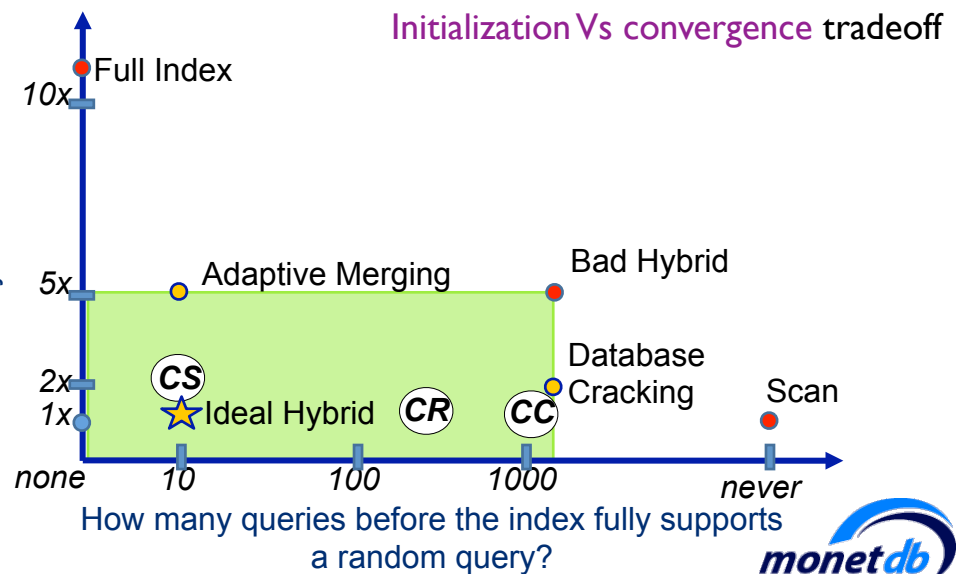
Hybrids PVLDB 2011

## Adaptive Indexing

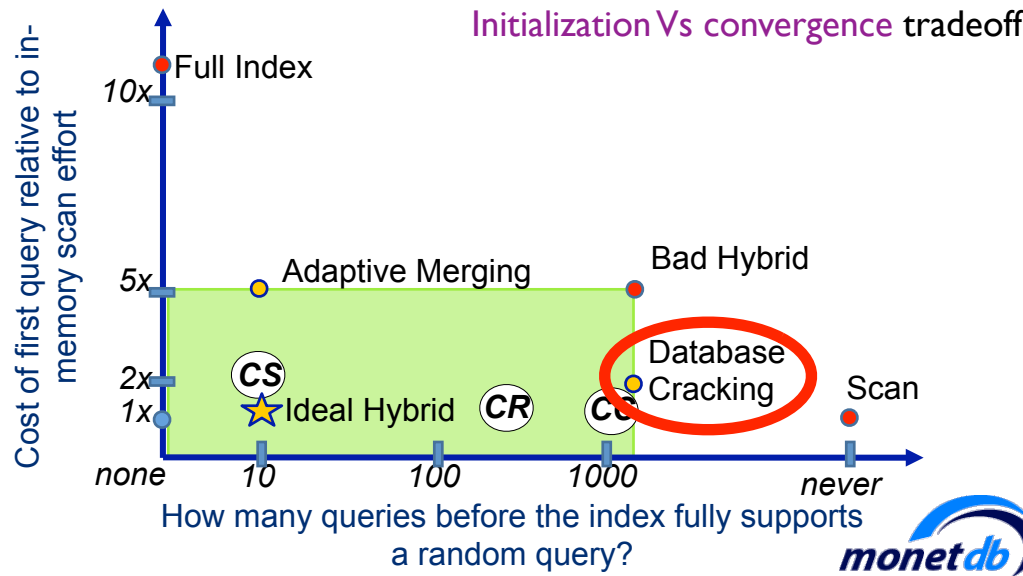
Hybrids PVLDB 2011



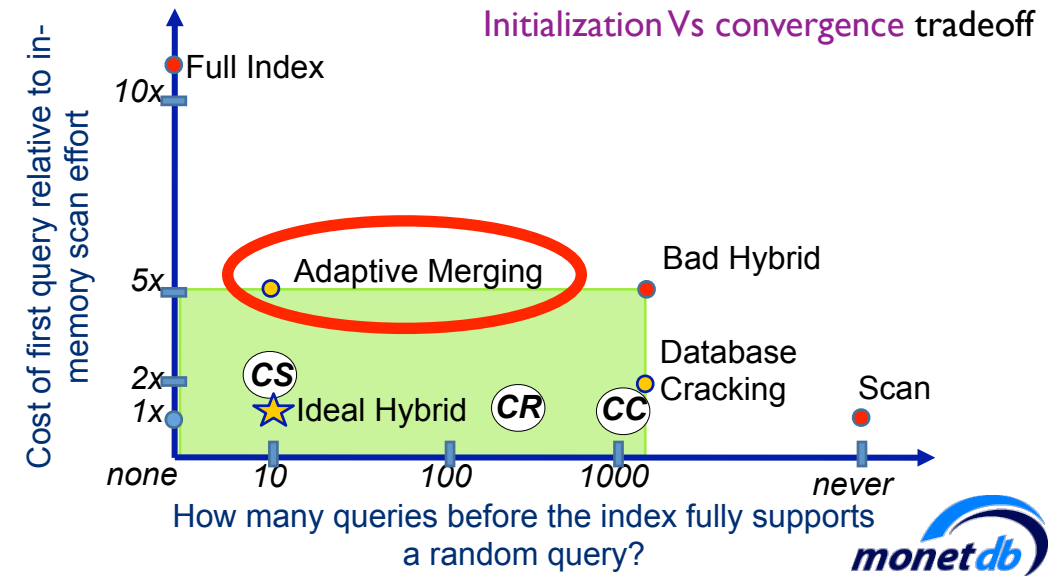
Cost of first query relative to in-memory scan effort



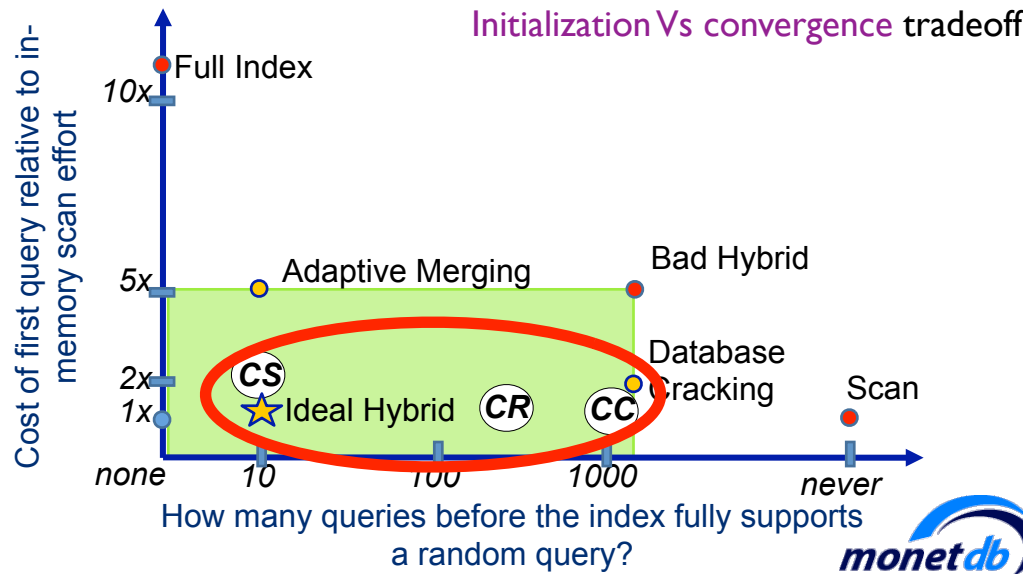
# Adaptive Indexing



# Adaptive Indexing



# Adaptive Indexing



### Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores

Felix Hahn, Statos Iliadis, Parag Mehta, and Roland H. C. Yap

**ABSTRACT**

Modern database applications and scientific databases can be highly sensitive to data access patterns. This sensitivity is often due to the fact that the data is stored in a columnar format, which is not directly accessible by the application. This paper presents a novel approach to indexing in columnar databases, called Stochastic Database Cracking (SDC). SDC is a novel indexing technique that allows the database to adaptively index the data based on the access patterns. It is designed to be robust to changes in the access patterns and to be able to handle a wide range of data access patterns. The paper describes the SDC algorithm and its implementation in the MonetDB database. The results show that SDC is able to adaptively index the data and to handle a wide range of data access patterns. The paper also discusses the challenges of indexing in columnar databases and the future work in this area.

### Self-organizing Tuple Reconstruction in Column-stores

Statos Iliadis, Martin L. Kersten, and Stefan Manegold

**ABSTRACT**

Column-stores are becoming increasingly popular in data warehousing and business intelligence applications. However, they often suffer from the problem of tuple reconstruction. This is because the data is stored in a columnar format, which is not directly accessible by the application. This paper presents a novel approach to tuple reconstruction in columnar databases, called Self-organizing Tuple Reconstruction (STR). STR is a novel indexing technique that allows the database to adaptively index the data based on the access patterns. It is designed to be robust to changes in the access patterns and to be able to handle a wide range of data access patterns. The paper describes the STR algorithm and its implementation in the MonetDB database. The results show that STR is able to adaptively index the data and to handle a wide range of data access patterns. The paper also discusses the challenges of indexing in columnar databases and the future work in this area.

### Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores

Statos Iliadis, Stefan Manegold, Harumi Kuno, and Götz Graefe

**ABSTRACT**

Modern database applications and scientific databases can be highly sensitive to data access patterns. This sensitivity is often due to the fact that the data is stored in a columnar format, which is not directly accessible by the application. This paper presents a novel approach to indexing in columnar databases, called Merging What's Cracked, Cracking What's Merged (MCC). MCC is a novel indexing technique that allows the database to adaptively index the data based on the access patterns. It is designed to be robust to changes in the access patterns and to be able to handle a wide range of data access patterns. The paper describes the MCC algorithm and its implementation in the MonetDB database. The results show that MCC is able to adaptively index the data and to handle a wide range of data access patterns. The paper also discusses the challenges of indexing in columnar databases and the future work in this area.

### Self-selecting, self-tuning, incrementally optimized indexes

Statos Iliadis, Martin L. Kersten, and Stefan Manegold

**ABSTRACT**

Modern database applications and scientific databases can be highly sensitive to data access patterns. This sensitivity is often due to the fact that the data is stored in a columnar format, which is not directly accessible by the application. This paper presents a novel approach to indexing in columnar databases, called Self-selecting, self-tuning, incrementally optimized indexes (SSIO). SSIO is a novel indexing technique that allows the database to adaptively index the data based on the access patterns. It is designed to be robust to changes in the access patterns and to be able to handle a wide range of data access patterns. The paper describes the SSIO algorithm and its implementation in the MonetDB database. The results show that SSIO is able to adaptively index the data and to handle a wide range of data access patterns. The paper also discusses the challenges of indexing in columnar databases and the future work in this area.

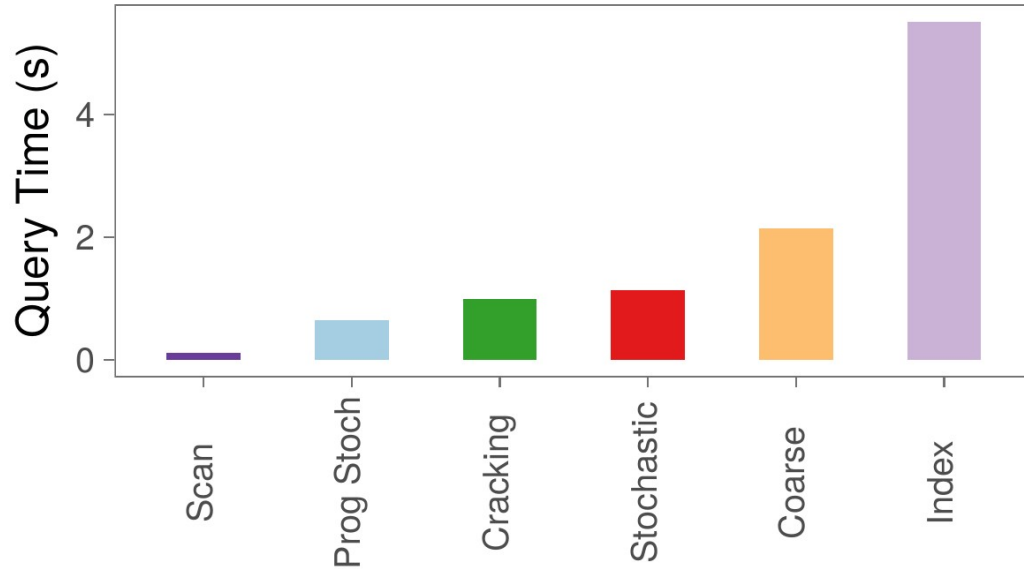
### Poor Man's Sort!

Statos Iliadis, Martin L. Kersten, and Stefan Manegold

**ABSTRACT**

Modern database applications and scientific databases can be highly sensitive to data access patterns. This sensitivity is often due to the fact that the data is stored in a columnar format, which is not directly accessible by the application. This paper presents a novel approach to indexing in columnar databases, called Poor Man's Sort (PMS). PMS is a novel indexing technique that allows the database to adaptively index the data based on the access patterns. It is designed to be robust to changes in the access patterns and to be able to handle a wide range of data access patterns. The paper describes the PMS algorithm and its implementation in the MonetDB database. The results show that PMS is able to adaptively index the data and to handle a wide range of data access patterns. The paper also discusses the challenges of indexing in columnar databases and the future work in this area.

# Adaptive Indexing: 1<sup>st</sup> Query Costs



Can we / how to:

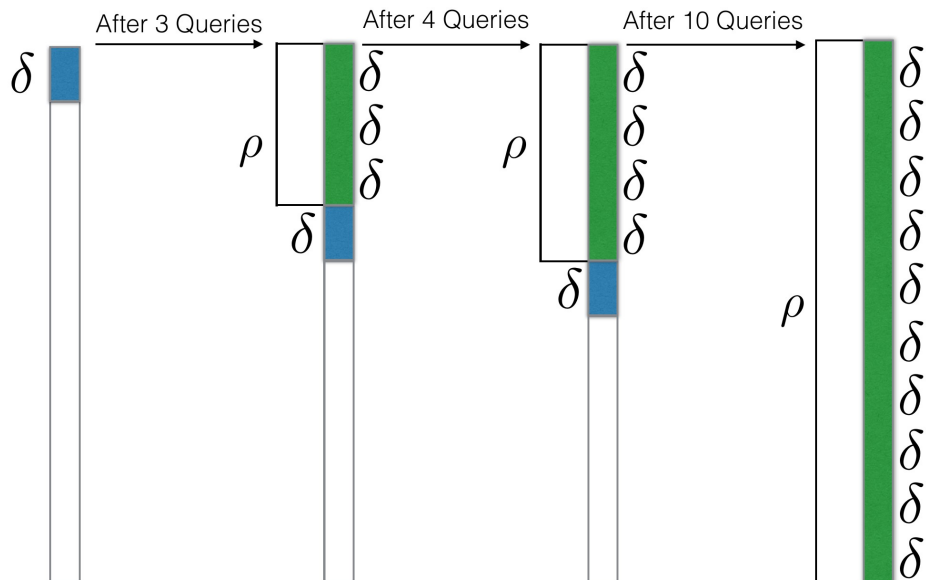
- Reduce / limit 1<sup>st</sup> query cost / overhead?
- Improve query performance predictability and robustness?
- Ensure convergence towards full index?

Yet unexplored “dimensions”:

- Other sorting algorithms than quick-sort
- Suspend/resume steps / iterations

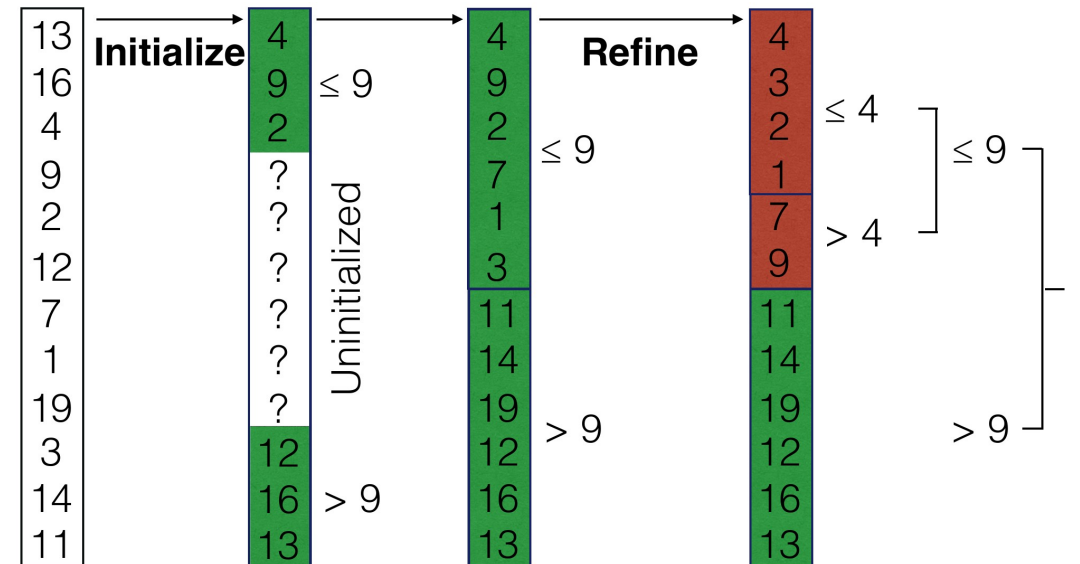
## Progressive Indexing

## Progressive Indexing



Mark Raasveldt, Pedro Holanda, Hannes Mühleisen

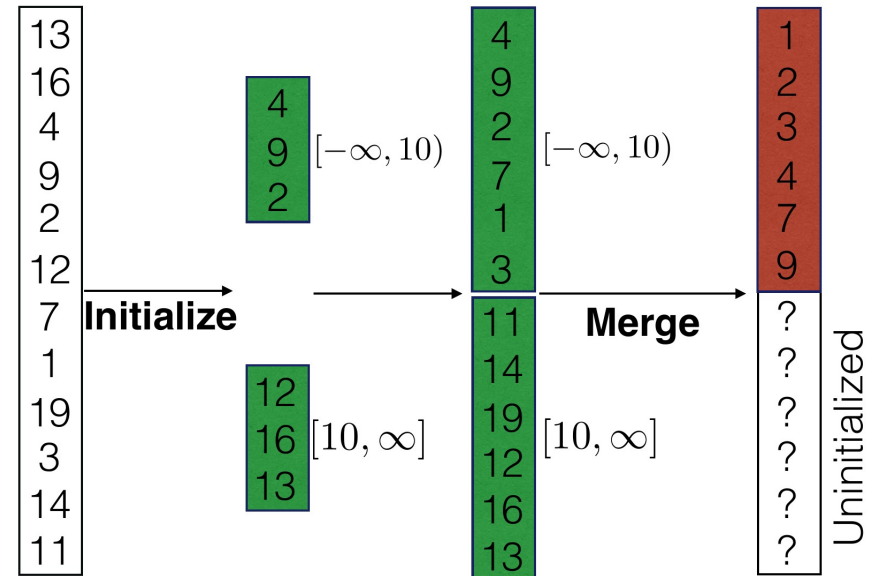
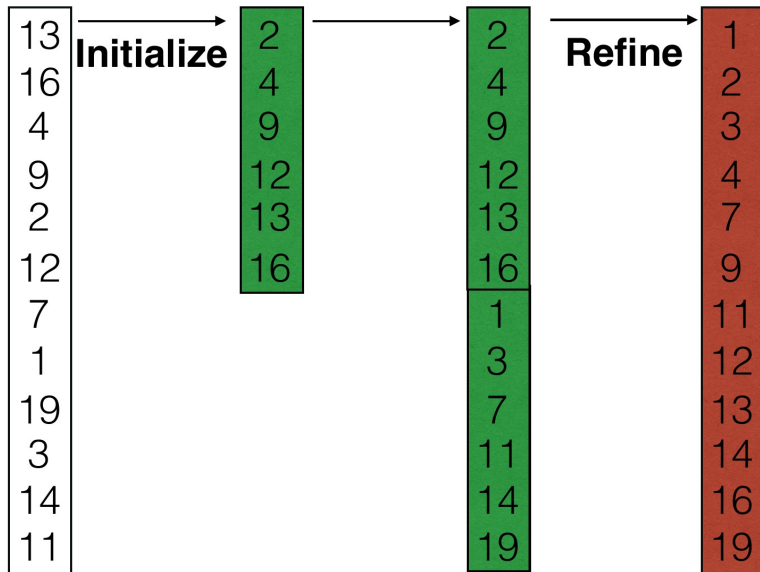
## Progressive Quick-Sort





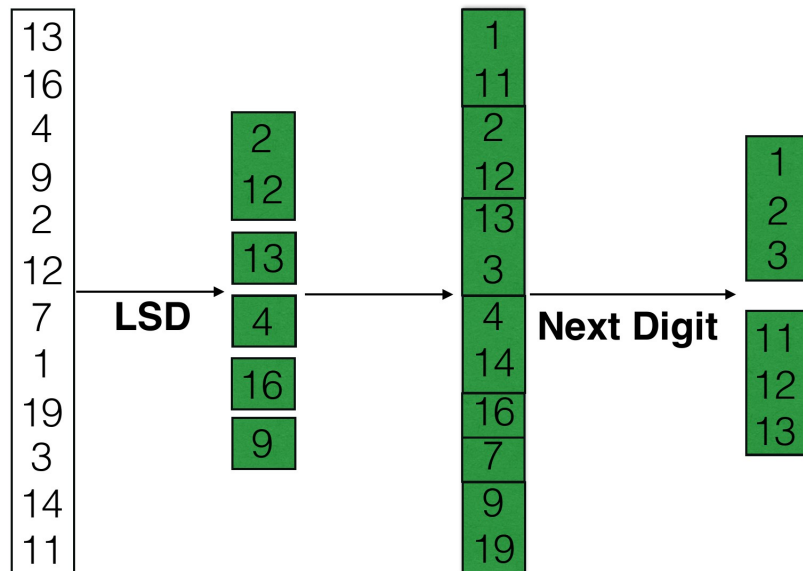
## Progressive Merge-Sort

## Progressive Bucket-Sort



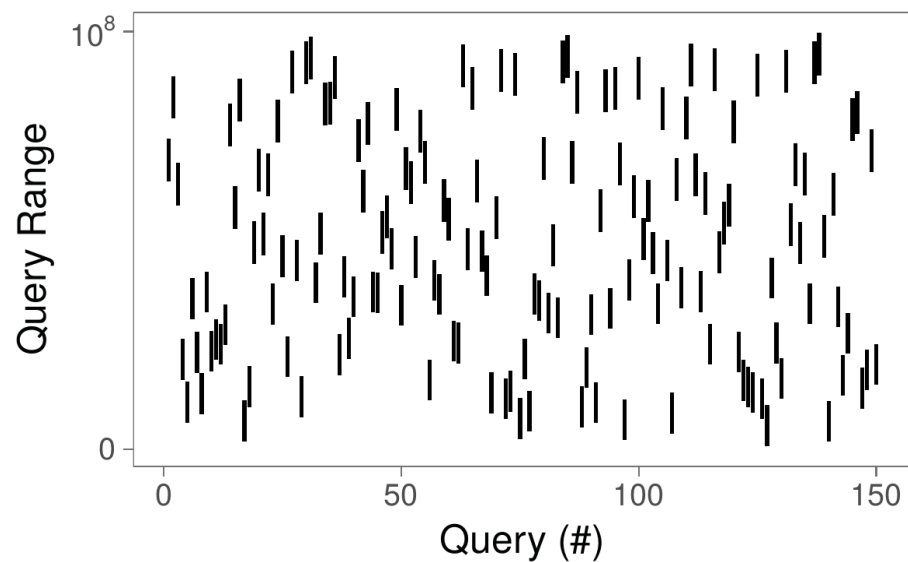
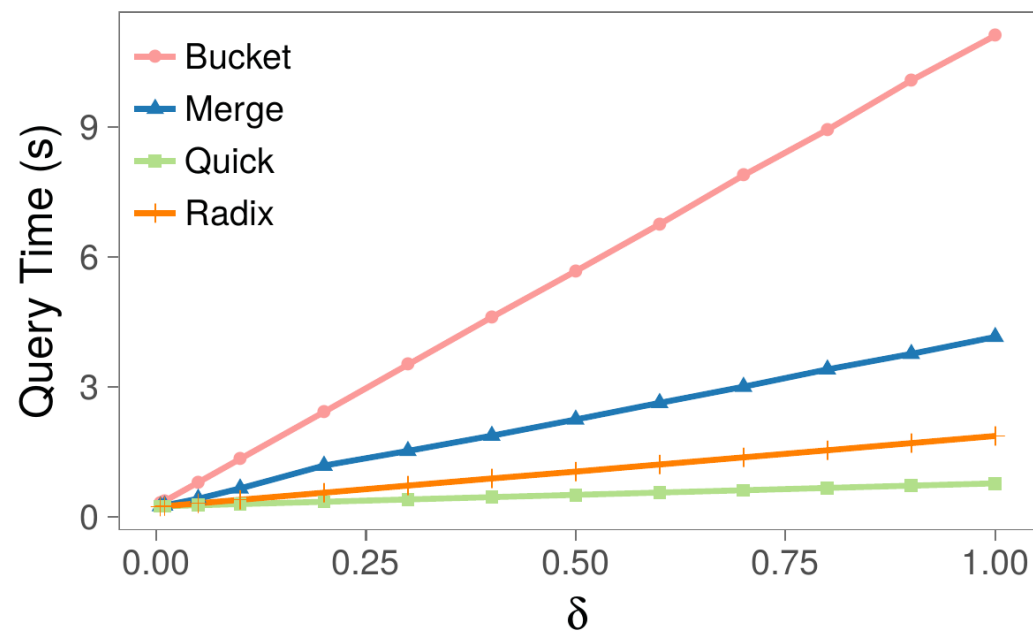
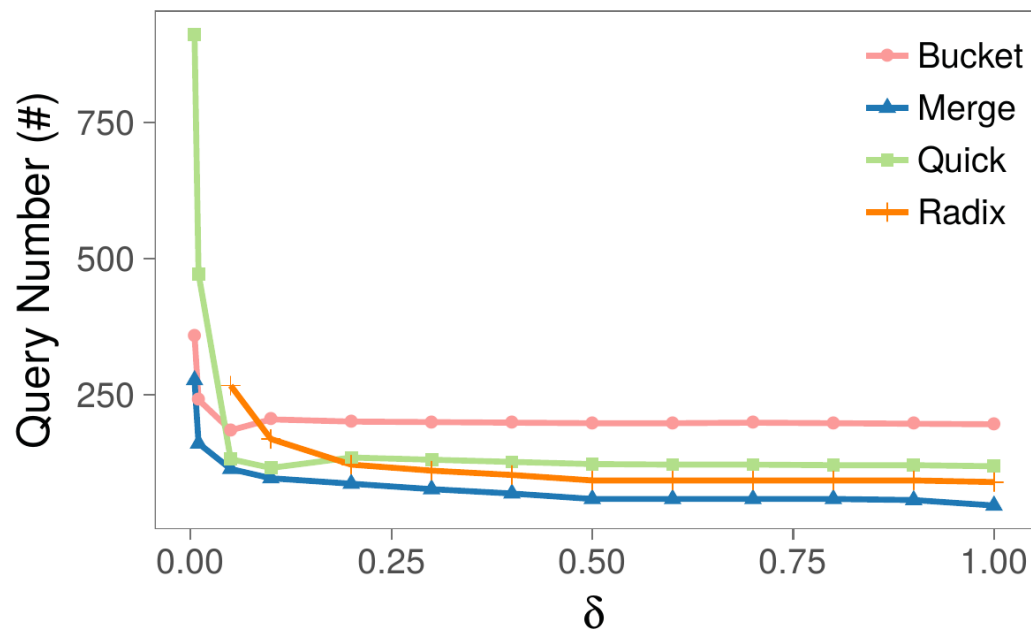
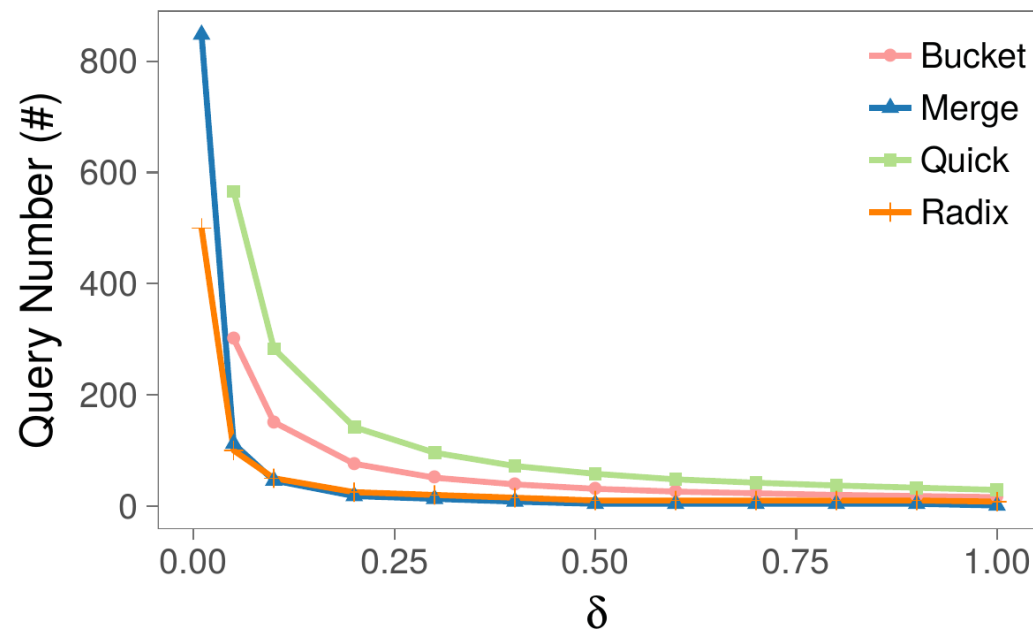
## Progressive Radix-Sort

## Experimental Setup

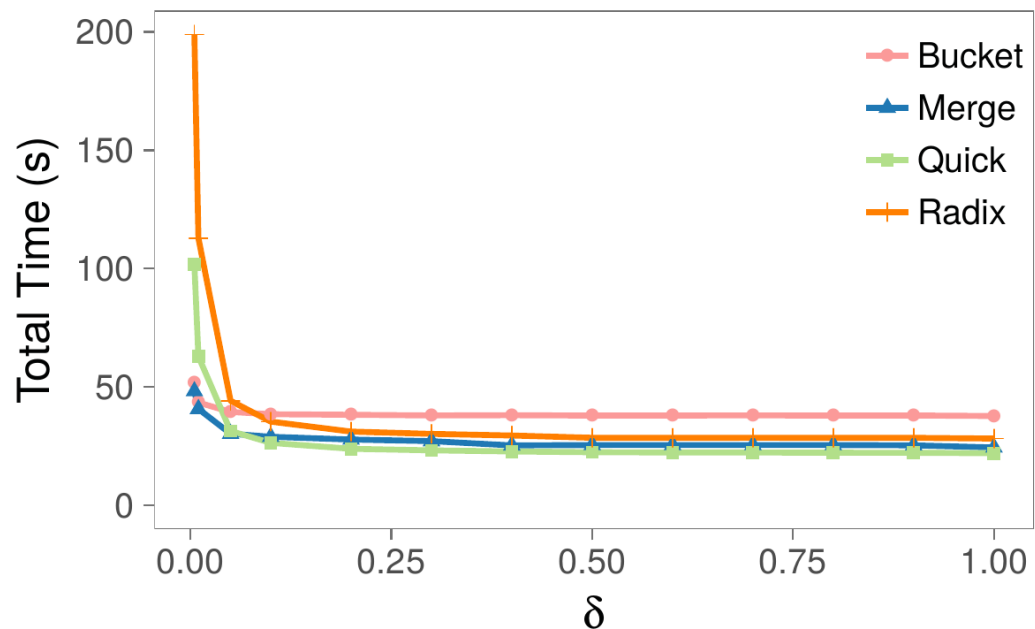


- Software:**
  - stand-alone C++ program, g++ -O3
  - Fedora 26
- Hardware:**
  - Intel Core i7-2600K CPU @ 3.40 GHz, 8 cores, 8 MB L3 cache
  - 16 GB main memory
- Data:**
  - 8-byte integers
  - $10^8$  uniformly distributed values
- Queries:**
  - SELECT SUM(R.A) FROM R WHERE R.A BETWEEN V1 AND V2
- Experiments:**
  - repeat entire workload 10 times
  - report median runtime per query
  - Default: 1000 queries, 10% selectivity, random workload

## Random Workload

Varying  $\delta$ :  
1<sup>st</sup> Query CostVarying  $\delta$ :  
# Queries until Pay-offVarying  $\delta$ :  
# Queries until Convergence

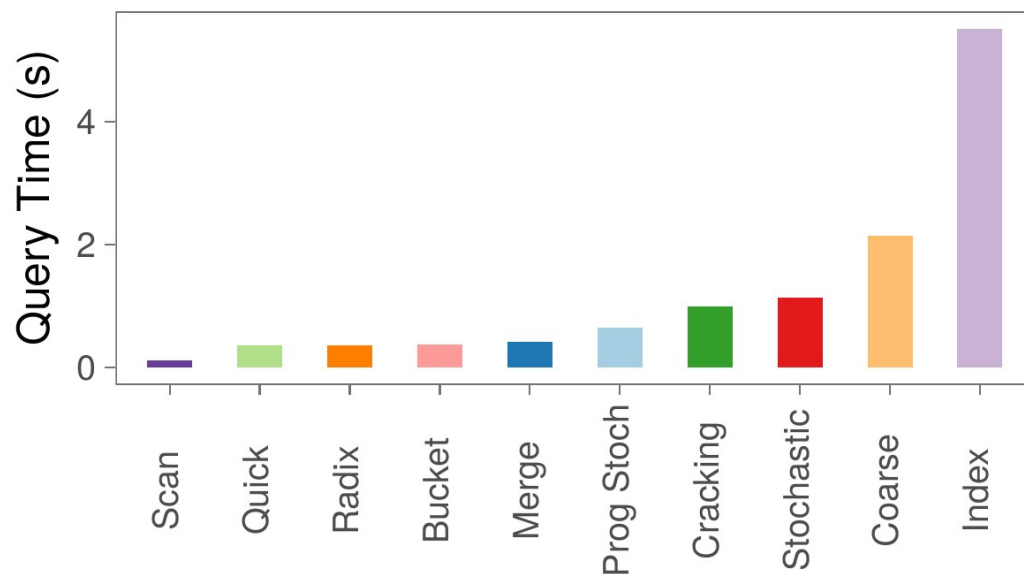
## Varying $\delta$ : Entire Workload Cost



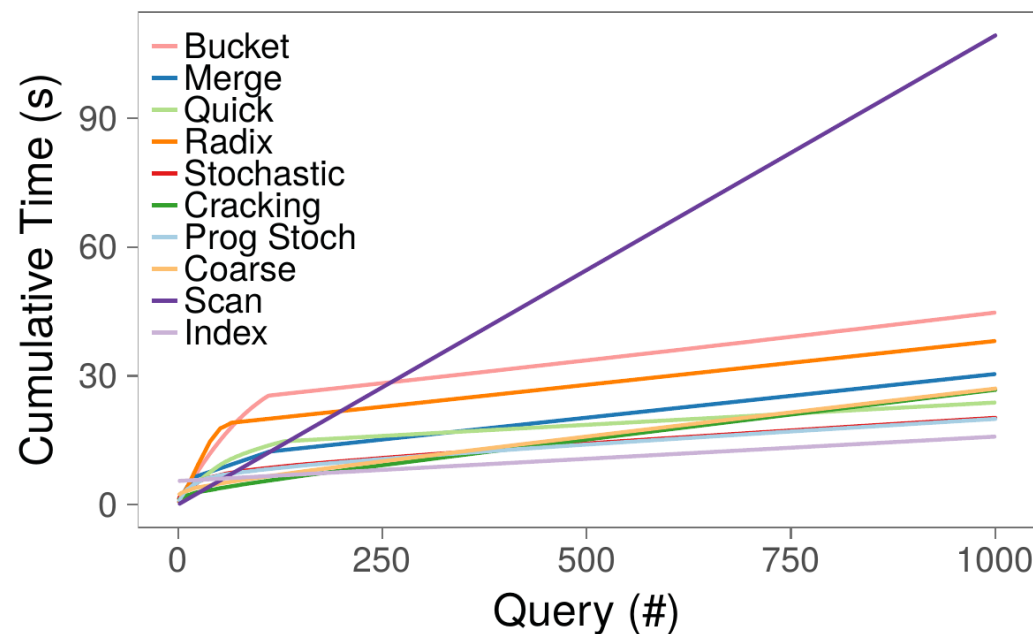
## Chosen $\delta$ : 1<sup>st</sup> Query $\approx$ 2x Scan

Indexing Method	$\delta$
Bucketsort	0.009
Mergesort	0.05
Quicksort	0.22
Radixsort	0.08

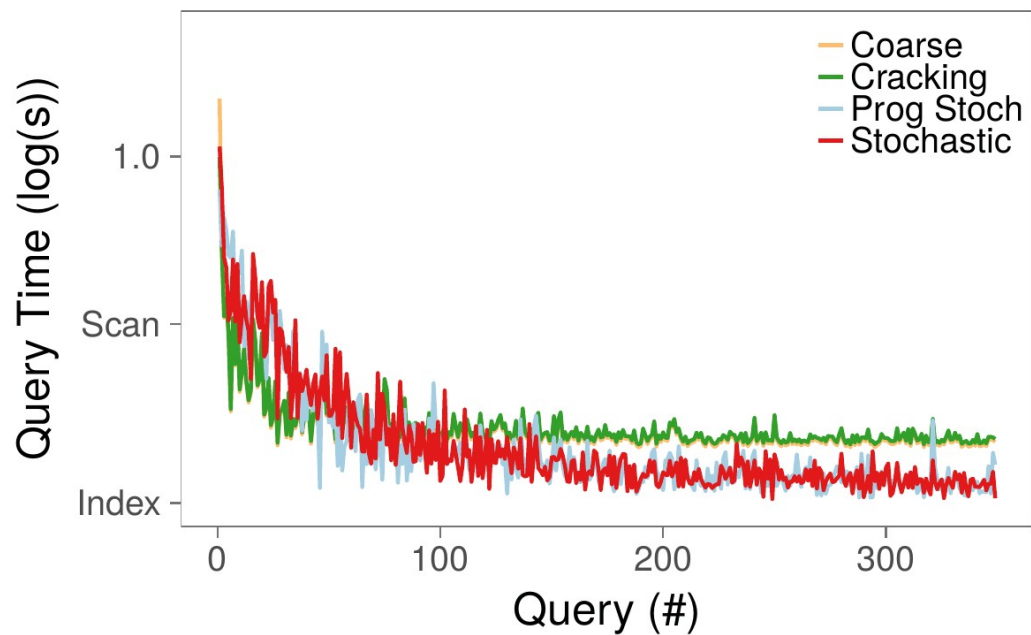
## Comparison: 1<sup>st</sup> Query



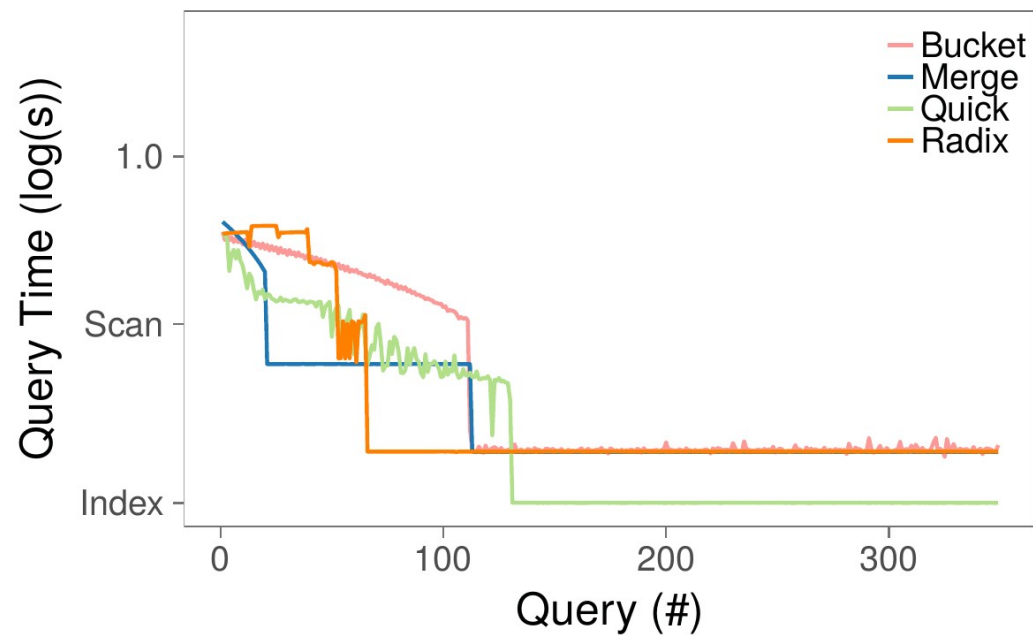
## Comparison: Entire Workload



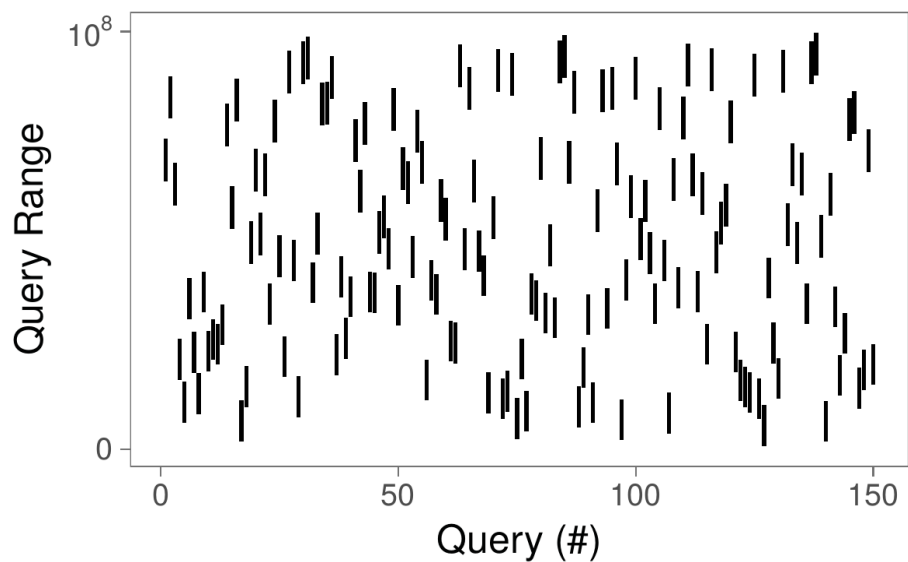
## Comparison: Adaptive Indexing



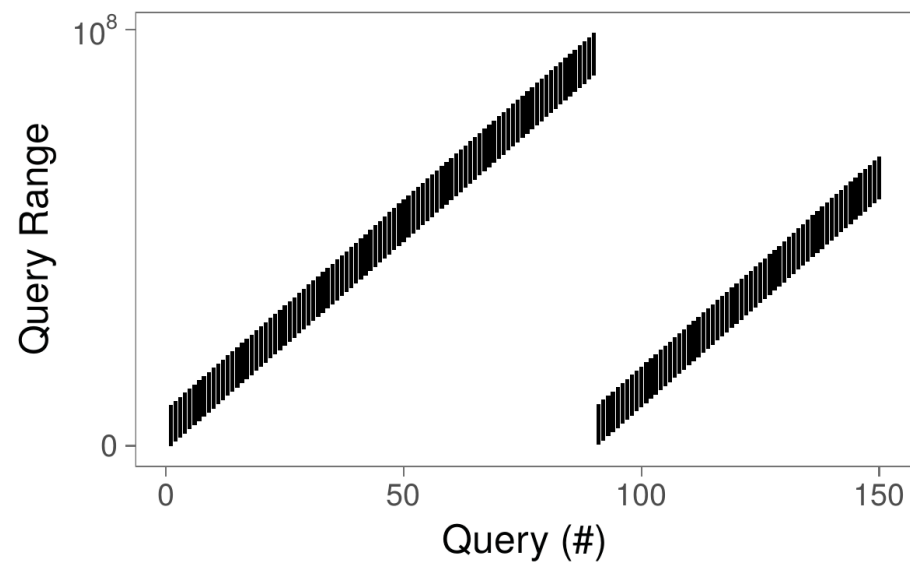
## Comparison: Progressive Indexing



## Random Workload

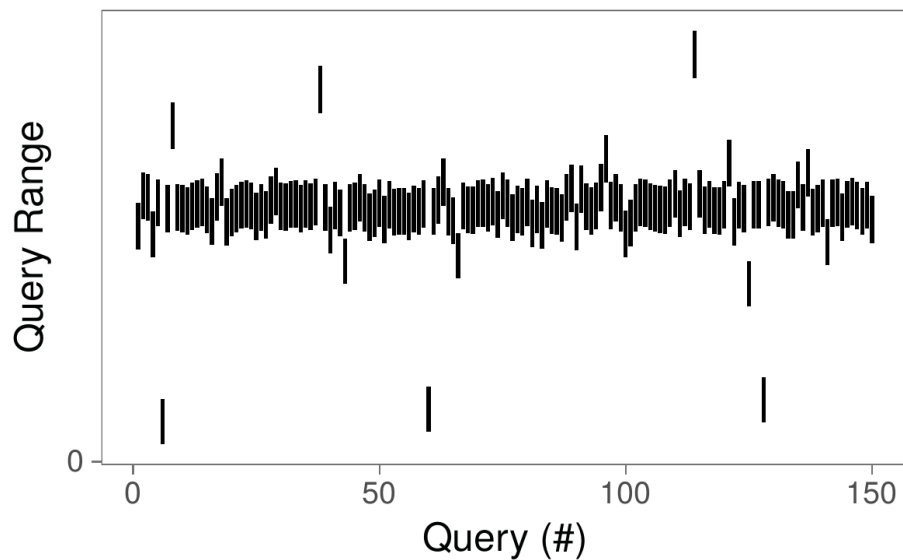


## Sequential Workload

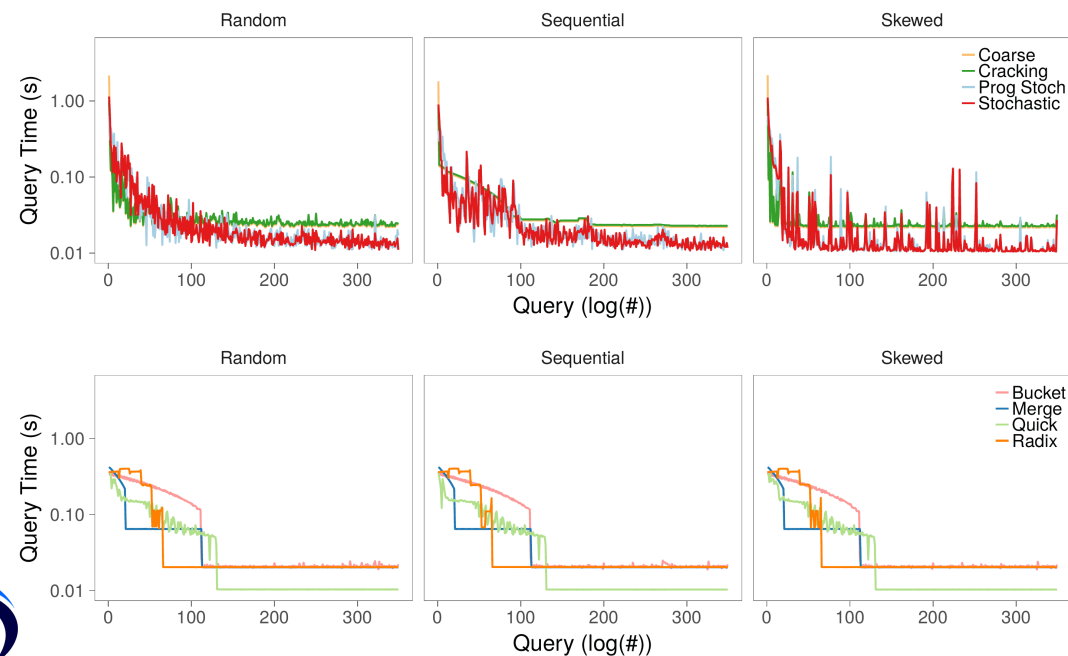




## Skewed Workload



## Different Workloads



## # Queries until Pay-off

## Progressive Indexing

Indexing Method	Random	Sequential	Skewed
Full Index	56	56	56
Standard Cracking	28	63	22
Stochastic Cracking	69	40	49
Progressive Stochastic	67	47	48
Coarse Granular Index	42	76	38
Bucketsort	258	261	257
Mergesort	113	114	114
Quicksort	136	128	139
Radixsort	200	200	200

- Robust & predictable query performance under various workloads
- Balance between
  - Fast convergence to full index
  - Small overhead for 1<sup>st</sup> query
- Various basic sorting algorithms
  - Quick-sort
  - Merge-sort
  - Bucket-sort
  - Radix-sort