Workload-Adaptive Indexing

Erwin M. Bakker & ***Stefan Manegold***

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

s.manegold@liacs.leidenuniv.nl
e.m.bakker@liacs.leidenuniv.nl

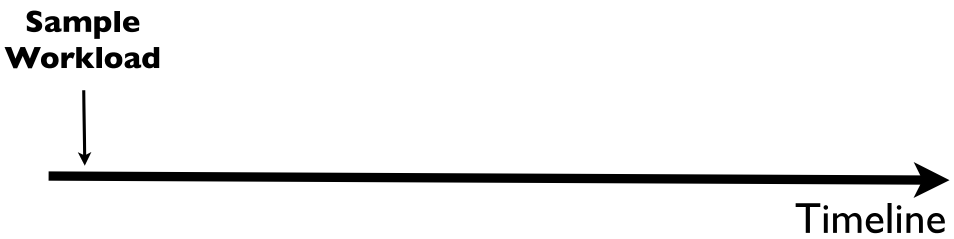
Physical design problem

Database systems perform efficiently only after proper tuning...

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

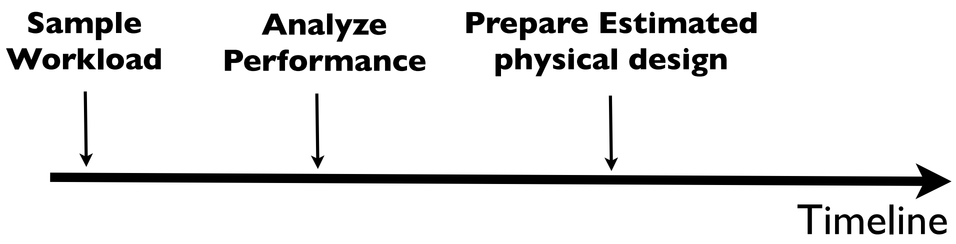
DBA without adaptive indexing

Physical Design

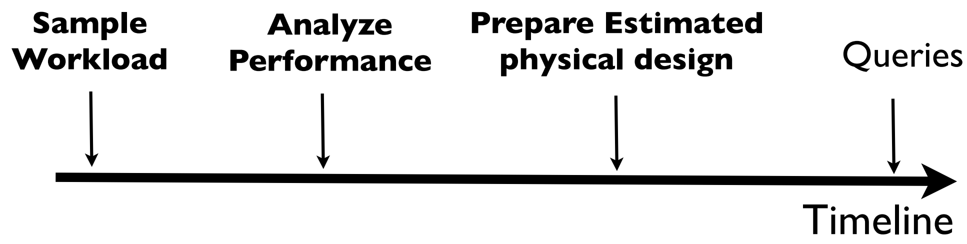


Physical Design

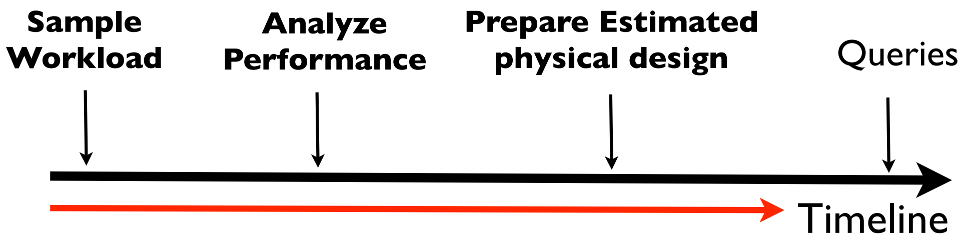
Physical Design



Physical Design

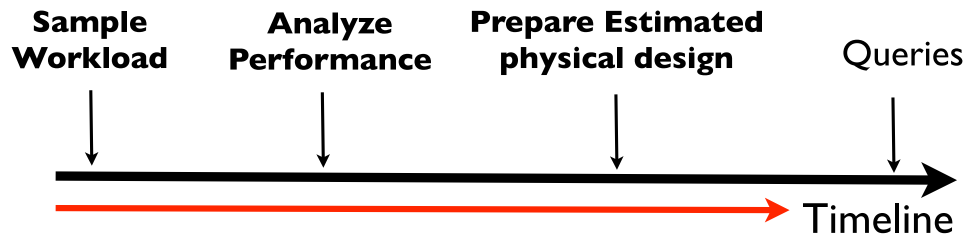


Physical Design



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

Dynamic environments

idle time **workload knowledge**

some problem cases

- Not enough idle time to finish proper tuning
- By the time we finish tuning, the workload changes
- No index support during tuning

- Not all data parts are equally useful

Adaptive Indexing

For dynamic environments:

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

DBA with adaptive indexing

How?

Design new auto-tuning kernels

(operators, plans, structures, etc.)

Adaptive Indexing

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

Adaptive Indexing

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

Continuous on-the-fly physical reorganization

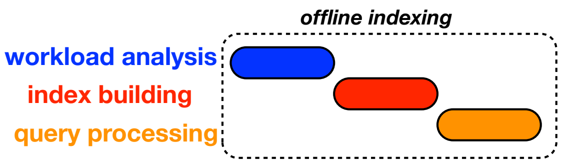
Adaptive Indexing

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

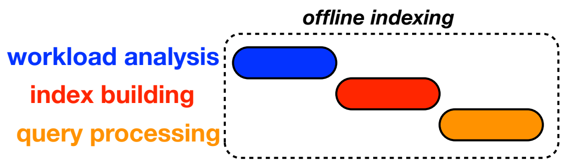
Continuous on-the-fly physical reorganization

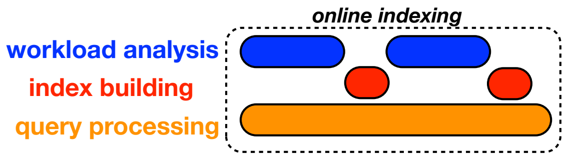
partial, incremental, adaptive indexing

Indexing Overview

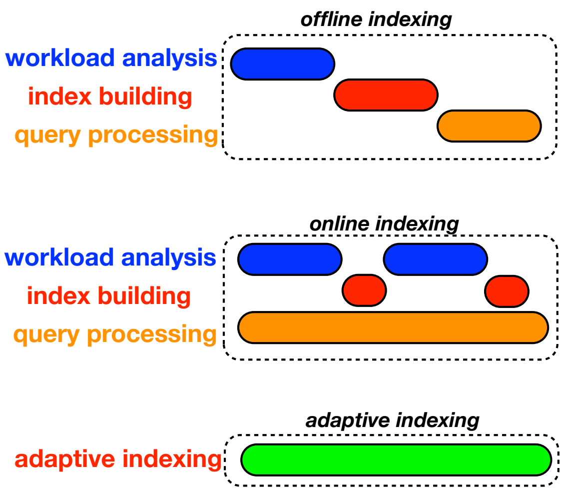


Indexing Overview

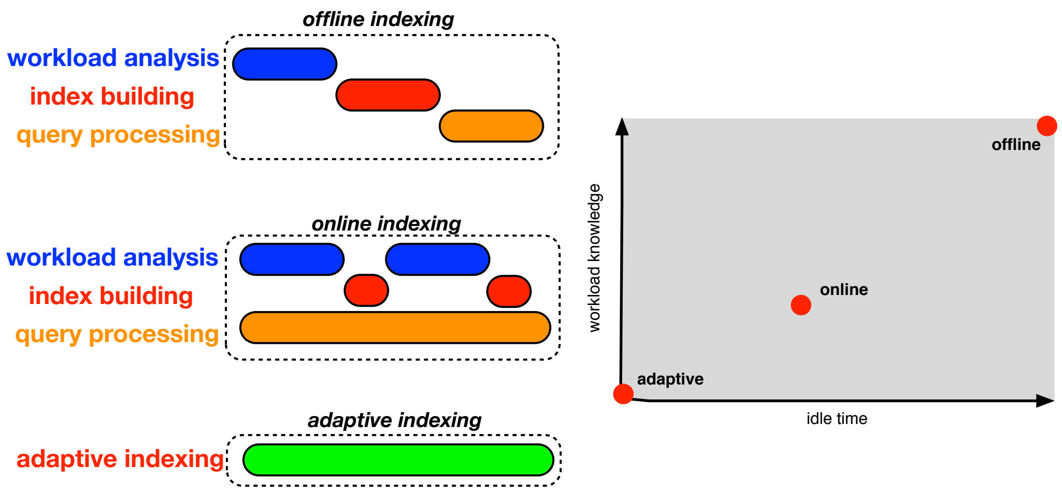




Indexing Overview



Indexing Overview



Cracking the Database Store

Martin Kersten Stefan Manegold

CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

Abstract

1 Introduction

Proceedings of the 2005 CIDR Conference

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

1. INTRODUCTION

Cracking Example

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

Cracking Example

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

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

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	Piece 1: A <= 10
16	9	
4	2	
9	7	
2	1	
12	3	
7	8	
1	6	
19	13	Piece 2: 10 < A < 14
3	12	
14	11	
11	16	Piece 3: 14 <= A
8	19	
6	14	

Cracking Example

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

Physically reorganize based on the selection predicate

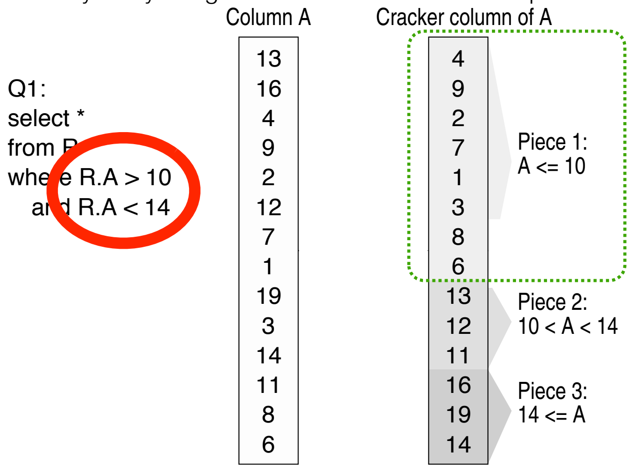
Q1:
select *
from R
where R.A > 10
and R.A < 14

Column A	Cracker column of A	
13	4	Piece 1: A <= 10
16	9	
4	2	
9	7	
2	1	
12	3	
7	8	
1	6	
19	13	Piece 2: 10 < A < 14
3	12	
14	11	
11	16	Piece 3: 14 <= A
8	19	
6	14	

Cracking Example

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

Physically reorganize based on the selection predicate

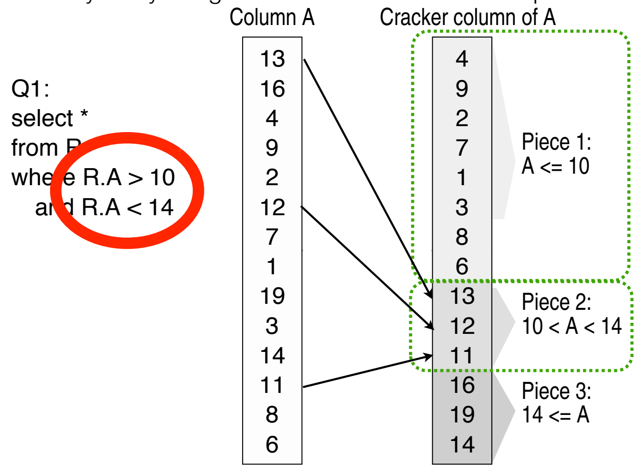


Database Cracking CIDR 2007

Cracking Example

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

Physically reorganize based on the selection predicate

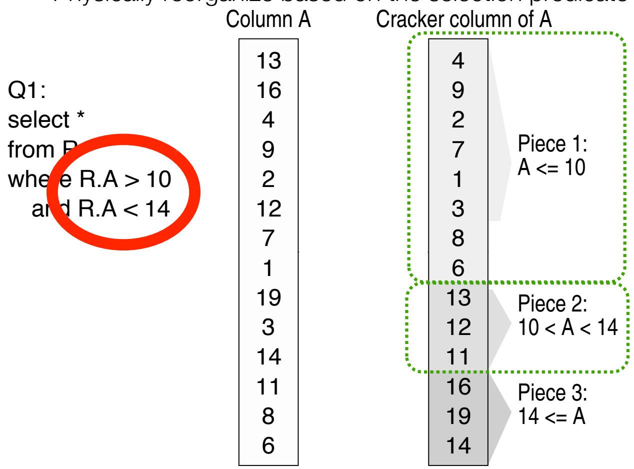


Database Cracking CIDR 2007

Cracking Example

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

Physically reorganize based on the selection predicate

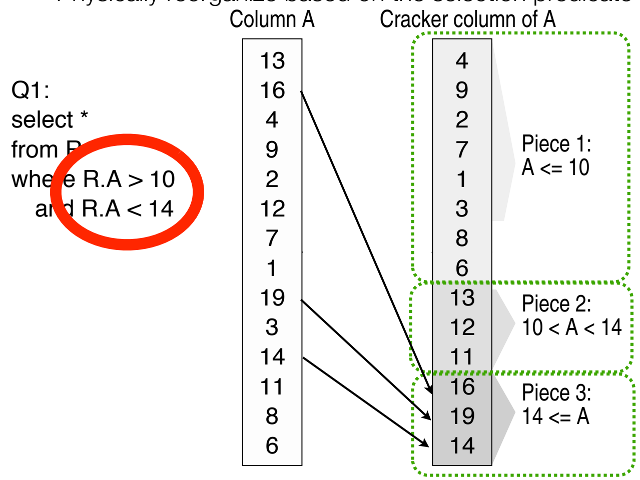


Database Cracking CIDR 2007

Cracking Example

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

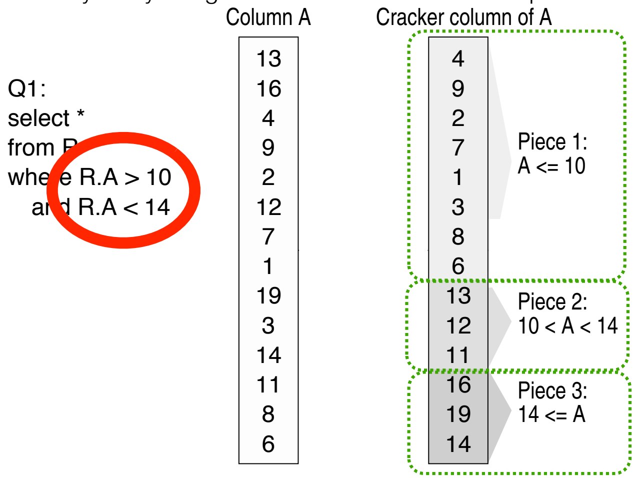
Physically reorganize based on the selection predicate



Cracking Example

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

Physically reorganize based on the selection predicate

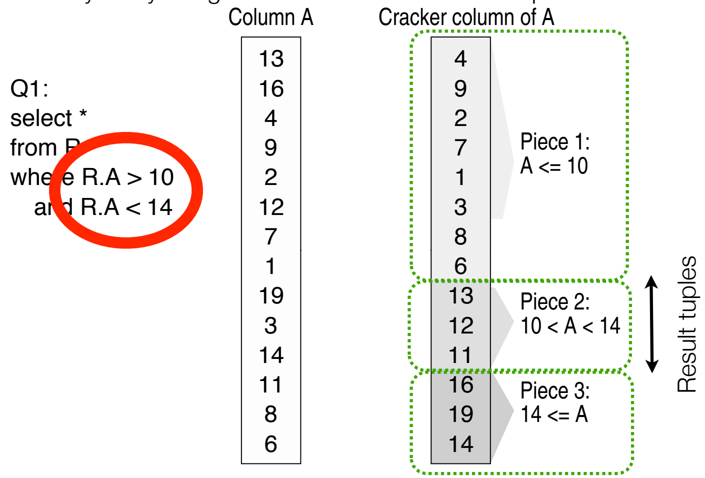


Database Cracking CIDR 2007

Cracking Example

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

Physically reorganize based on the selection predicate

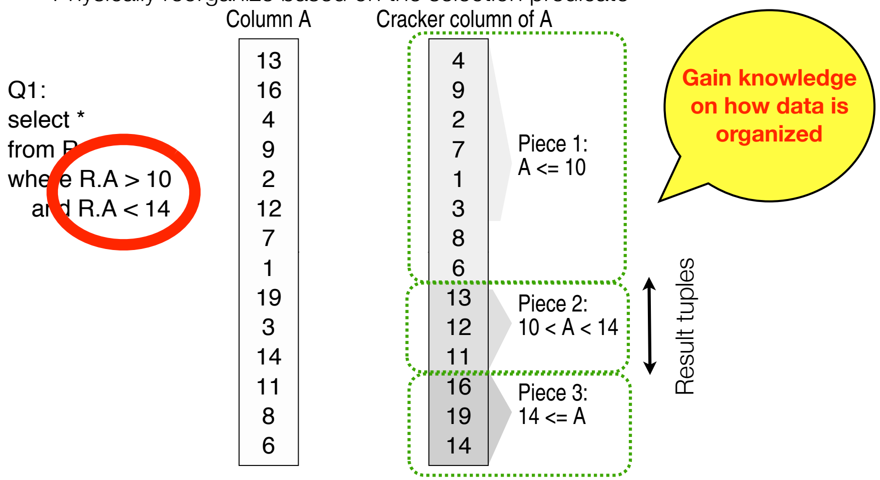


Database Cracking CIDR 2007

Cracking Example

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

Physically reorganize based on the selection predicate

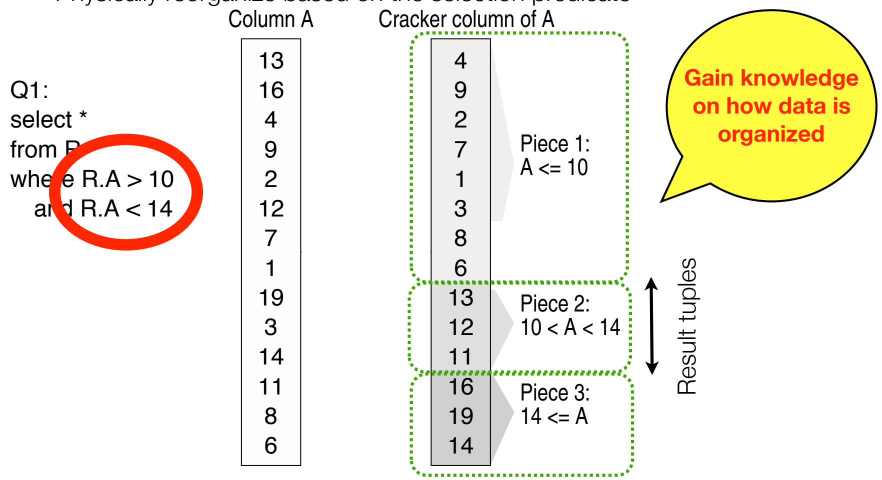


Database Cracking CIDR 2007

Cracking Example

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

Physically reorganize based on the selection predicate



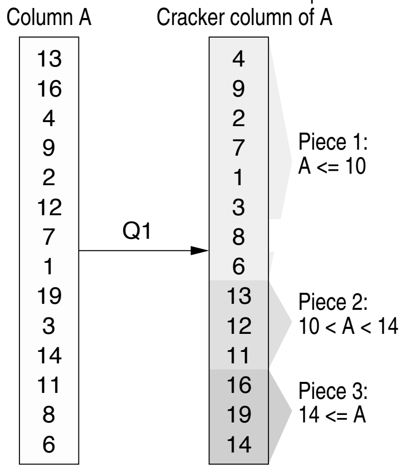
Dynamically/on-the-fly within the select-operator

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



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

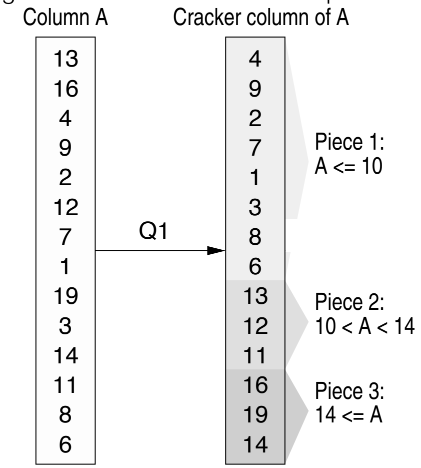
Cracking Example

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

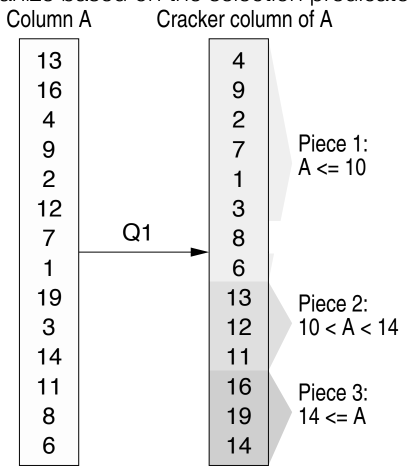
Cracking Example

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

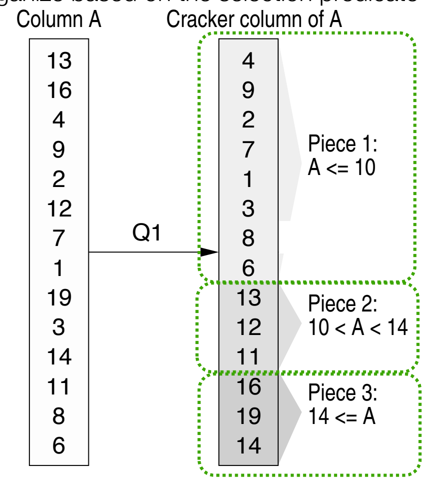
Cracking Example

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

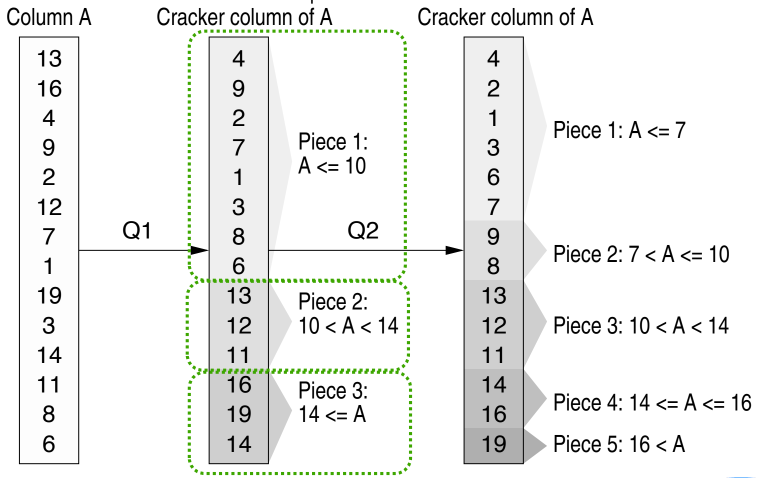
Cracking Example

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

Physically reorganize based on the selection predicate

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

Q2:
select *
from R
wher R.A > 7
an R.A <= 16



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

Database Cracking CIDR 2007

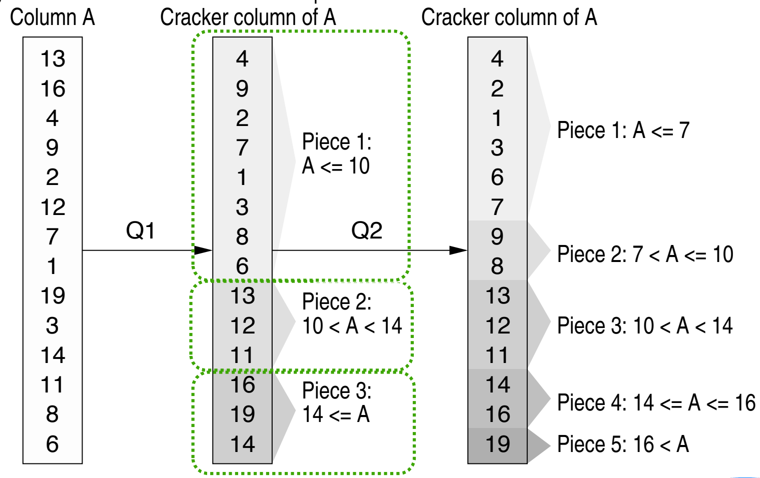
Cracking Example

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
wher R.A > 7
an R.A <= 16



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

Database Cracking CIDR 2007

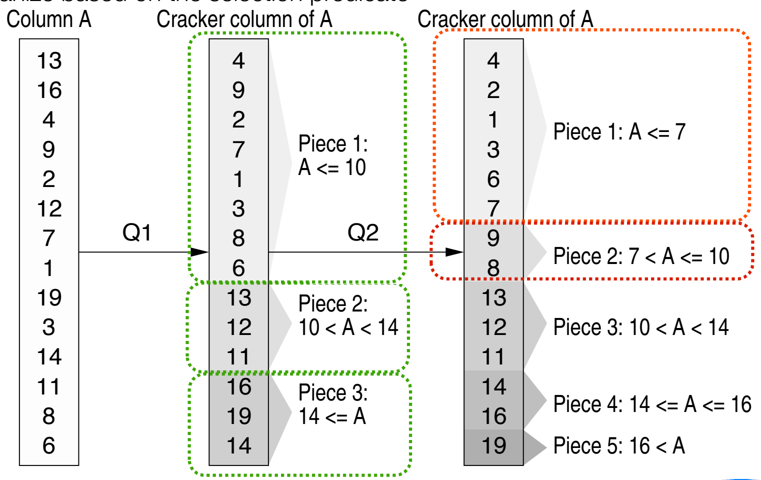
Cracking Example

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
wher R.A > 7
an R.A <= 16



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

Database Cracking CIDR 2007

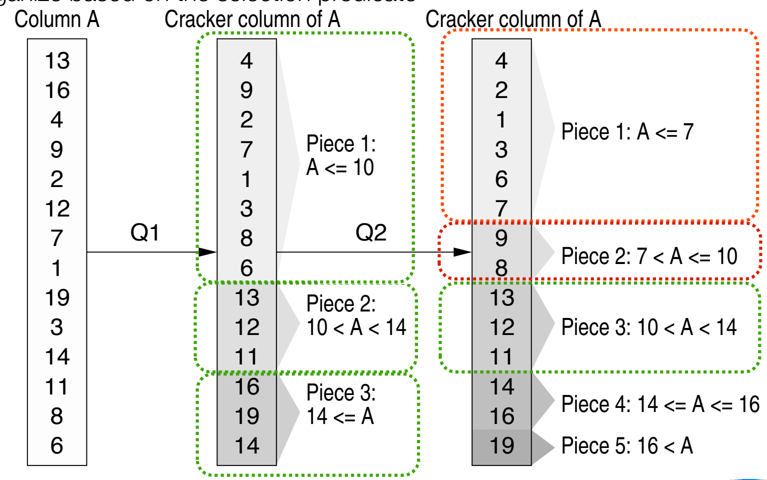
Cracking Example

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
wher R.A > 7
an R.A <= 16

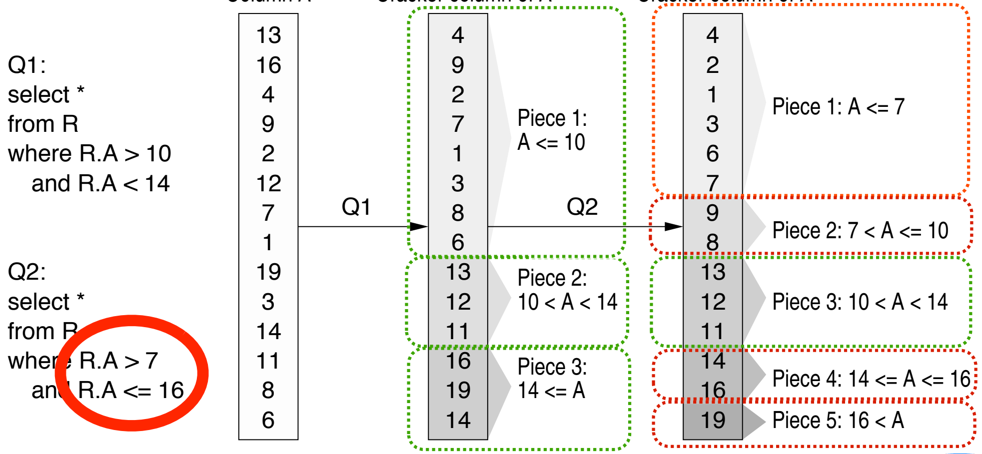


Dynamically/on-the-fly within the select-operator

Cracking Example

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

Physically reorganize based on the selection predicate

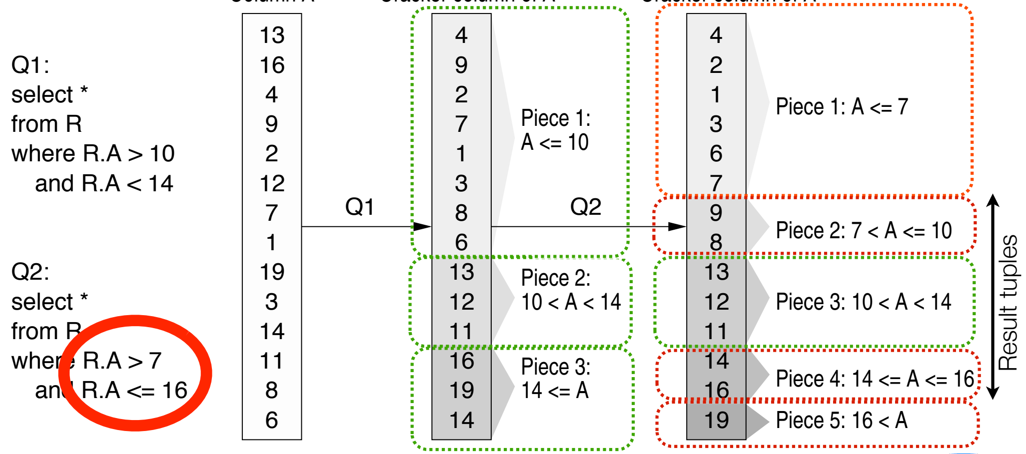


Dynamically/on-the-fly within the select-operator

Cracking Example

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

Physically reorganize based on the selection predicate



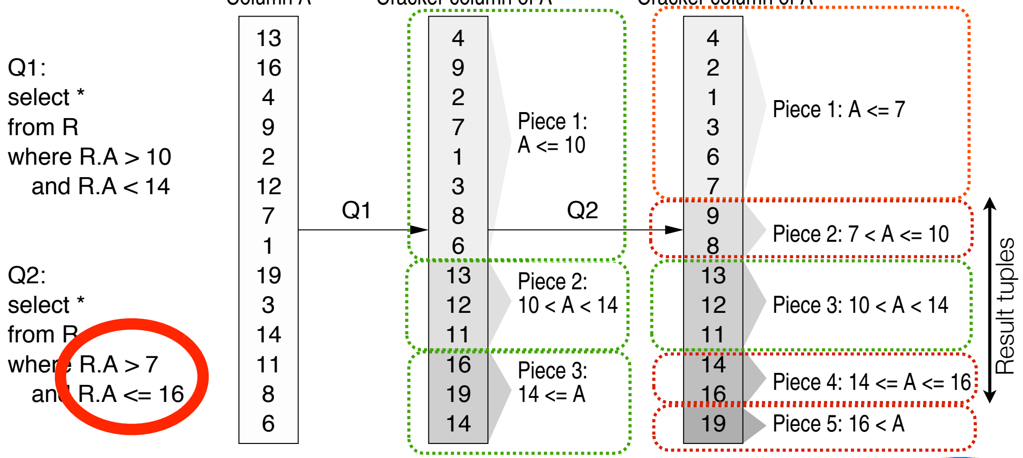
Dynamically/on-the-fly within the select-operator

Cracking Example

Each query is treated as an advice on how data shou

The more we crack, the more we learn

Physically reorganize based on the selection predicate



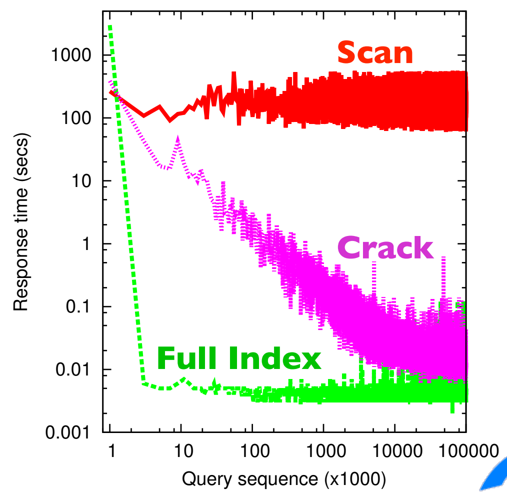
Dynamically/on-the-fly within the select-operator

Cracking Example

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

set-up

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

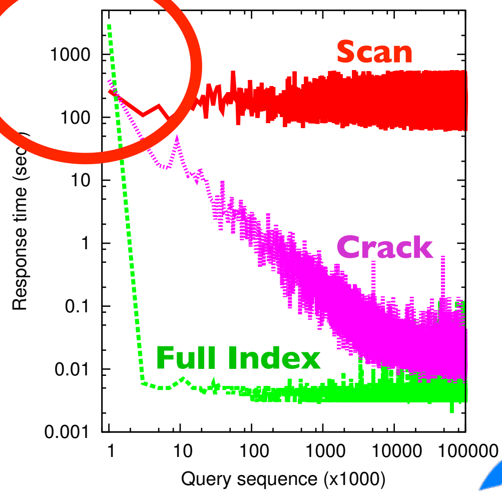


Cracking Example

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



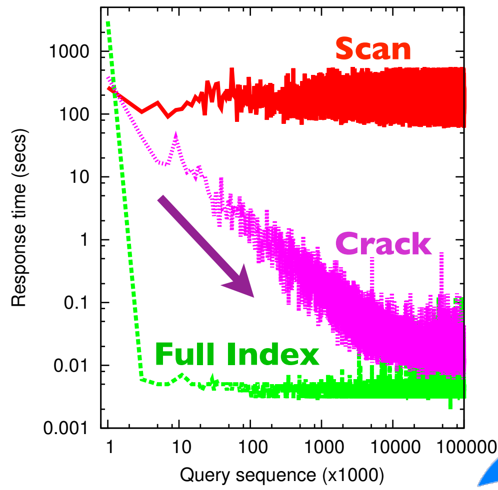
almost no initialization overhead

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

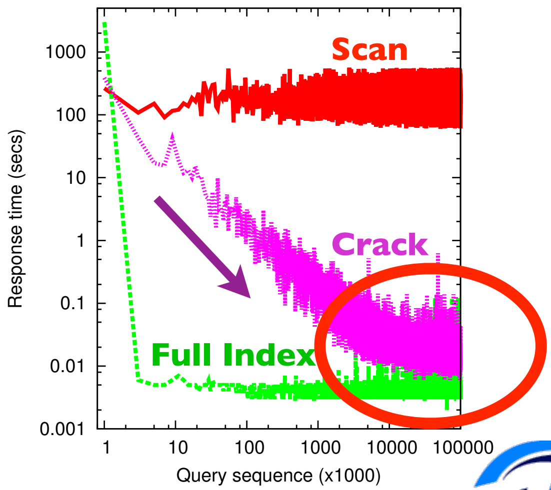
continuous improvement

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

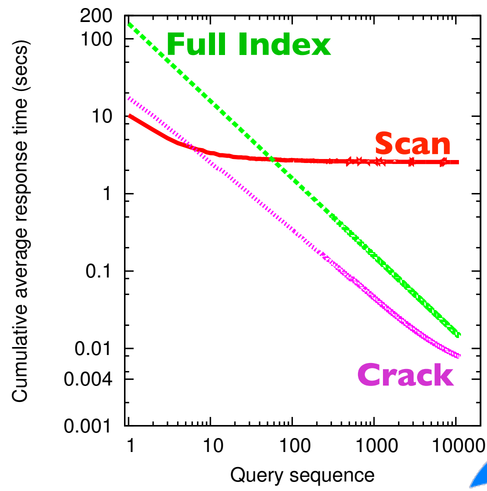
continuous improvement

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



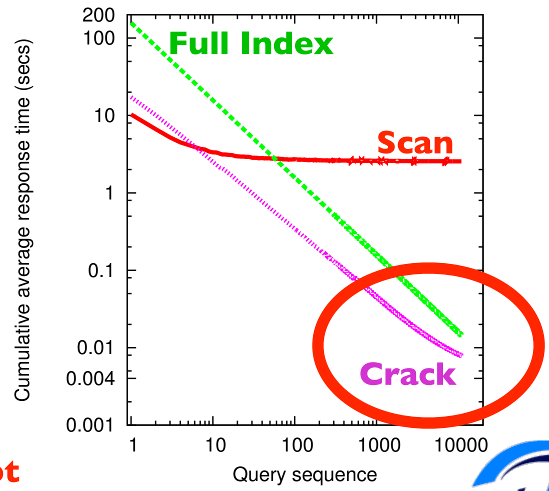
Database Cracking CIDR 2007

Cracking Example

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

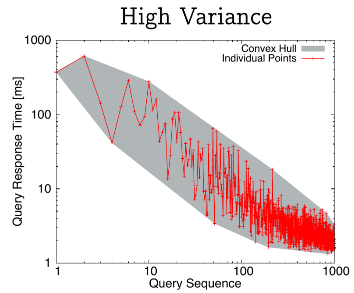
set-up

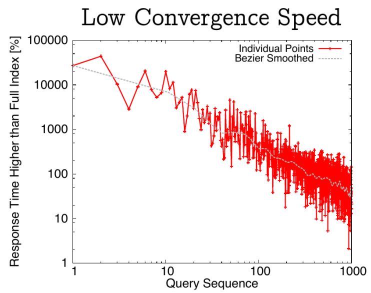
10K random selections
selectivity 10%
random value ranges
in a 30 million integer column



10K queries later,
Full Index still has not
amortized the initialization costs

Problems





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

Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores[*]

Felix Halim[*] Stratos Idreos[†] Panagiotis Karras[◊] Roland H. C. Yap[*]

[*]National University of Singapore {halim, ryap}@comp.nus.edu.sg
[†]CWI, Amsterdam idreos@cwi.nl
[◊]Rutgers University karras@business.rutgers.edu

ABSTRACT

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

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

1. INTRODUCTION

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

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

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

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

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

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

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

[*]Work supported by Singapore's MOE AcRF grant T1 251RES0807.

502

Stochastic cracking

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

Workload Robustness

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

Goal:
Maintain adaptive behavior regardless of query input

Stochastic Cracking, PVLDB 12

Query patterns

column with 100 unique integers

Good pattern

Bad pattern

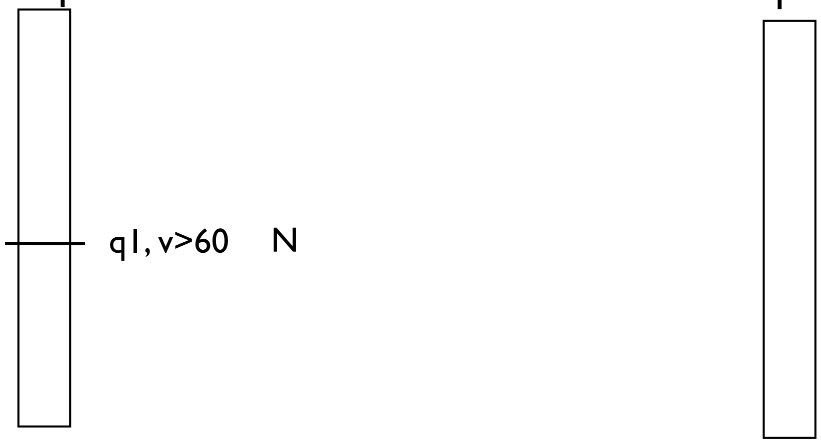
Stochastic Cracking, PVLDB 12

Query patterns

column with 100 unique integers

Good pattern

Bad pattern



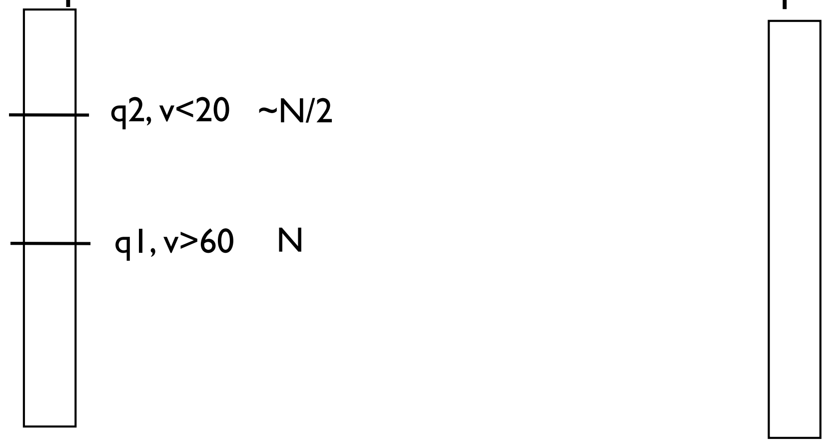
Stochastic Cracking, PVLDB 12

Query patterns

column with 100 unique integers

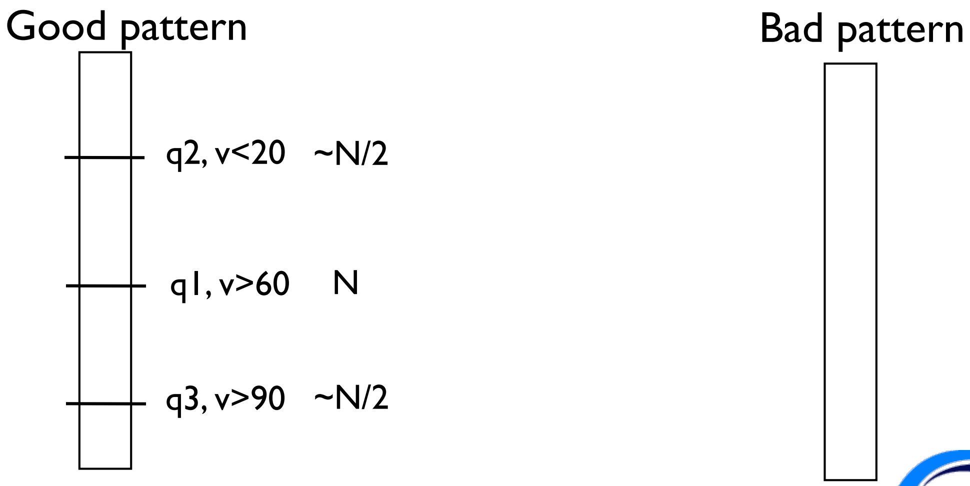
Good pattern

Bad pattern



Query patterns

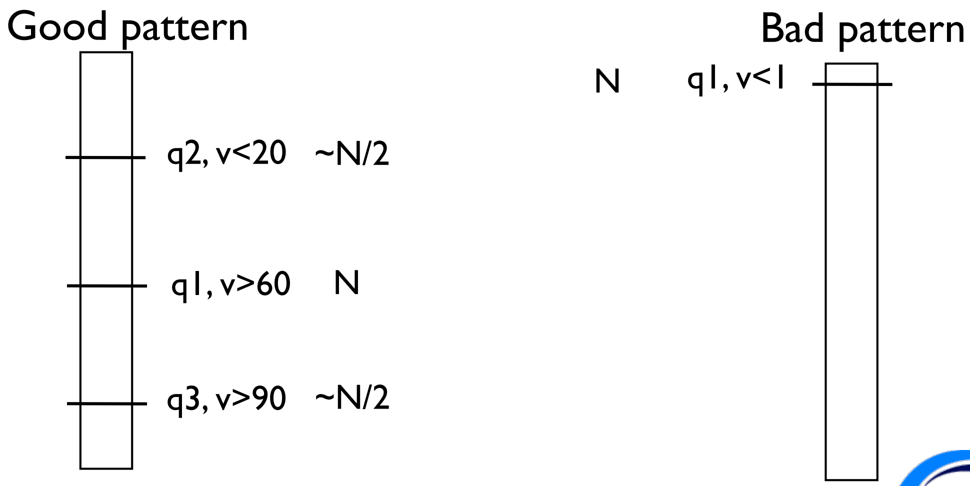
column with 100 unique integers



Stochastic Cracking, PVLDB 12

Query patterns

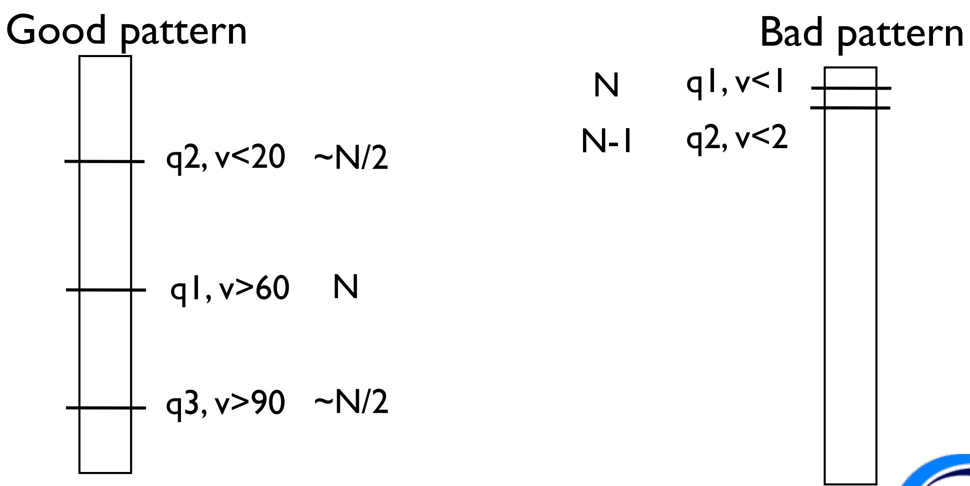
column with 100 unique integers



Stochastic Cracking, PVLDB 12

Query patterns

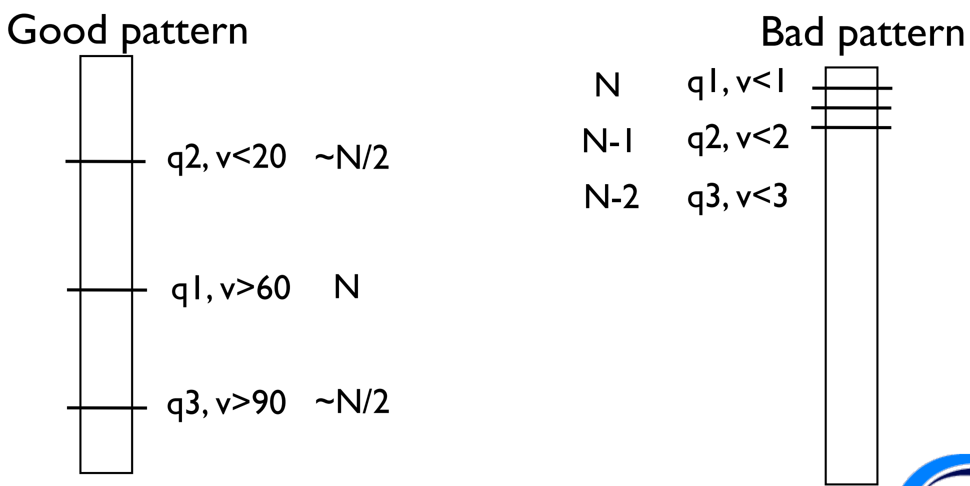
column with 100 unique integers



Stochastic Cracking, PVLDB 12

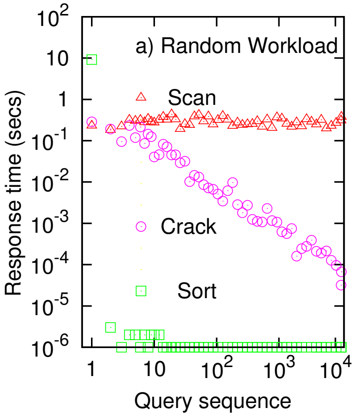
Query patterns

column with 100 unique integers



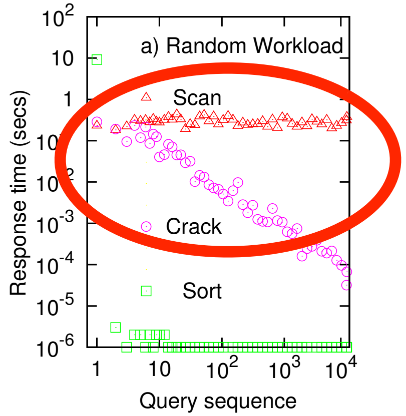
Query patterns

column size 100M
selectivity 10 tuples



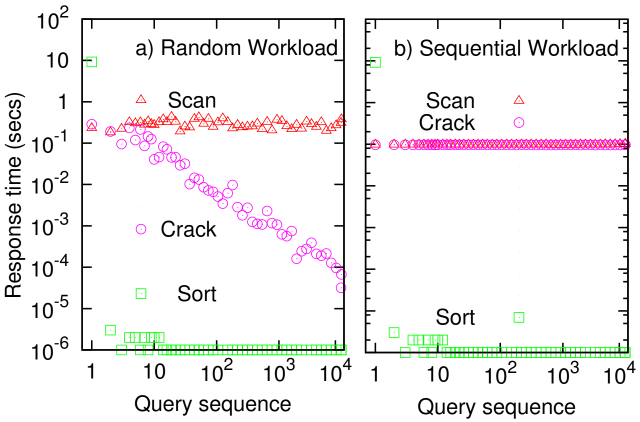
Query patterns

column size 100M
selectivity 10 tuples



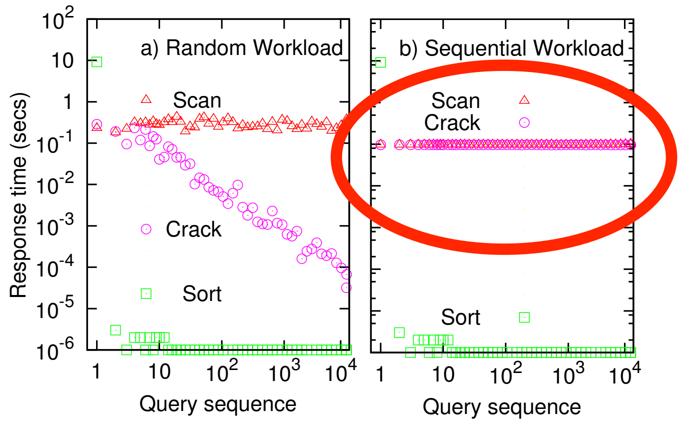
Query patterns

column size 100M
selectivity 10 tuples



Query patterns

column size 100M
selectivity 10 tuples

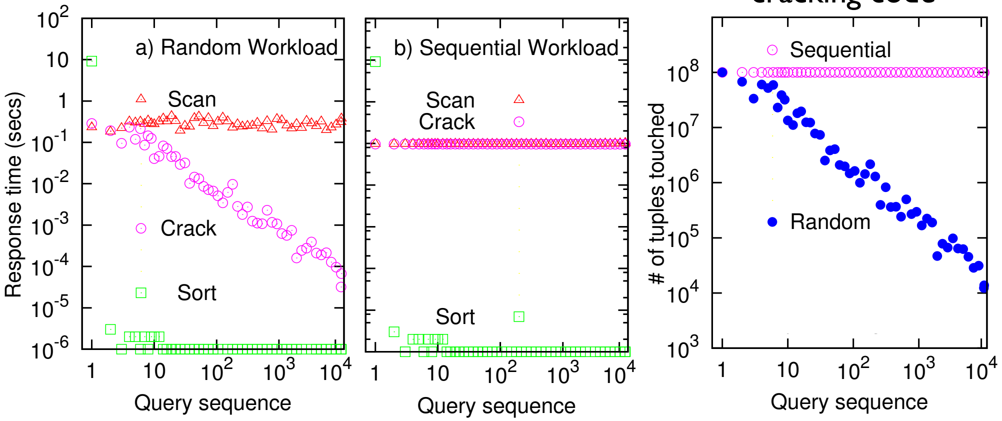


performance degrades to scan

Query patterns

column size 100M
selectivity 10 tuples

tuples touched by
cracking code



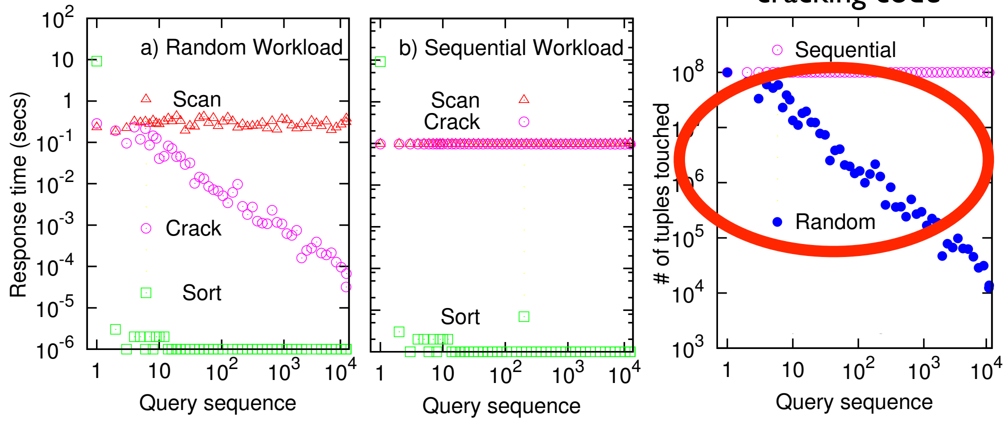
performance degrades to scan

Stochastic Cracking, PVLDB 12

Query patterns

column size 100M
selectivity 10 tuples

tuples touched by
cracking code



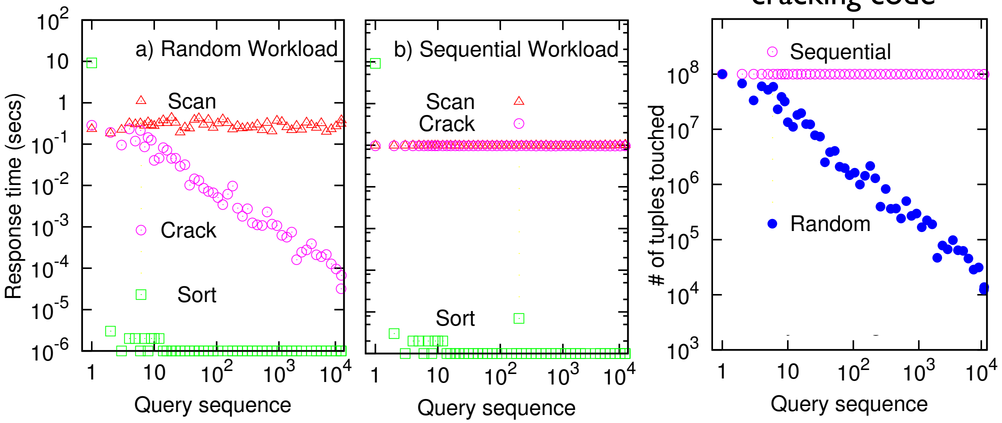
performance degrades to scan

Stochastic Cracking, PVLDB 12

Query patterns

column size 100M
selectivity 10 tuples

tuples touched by
cracking code



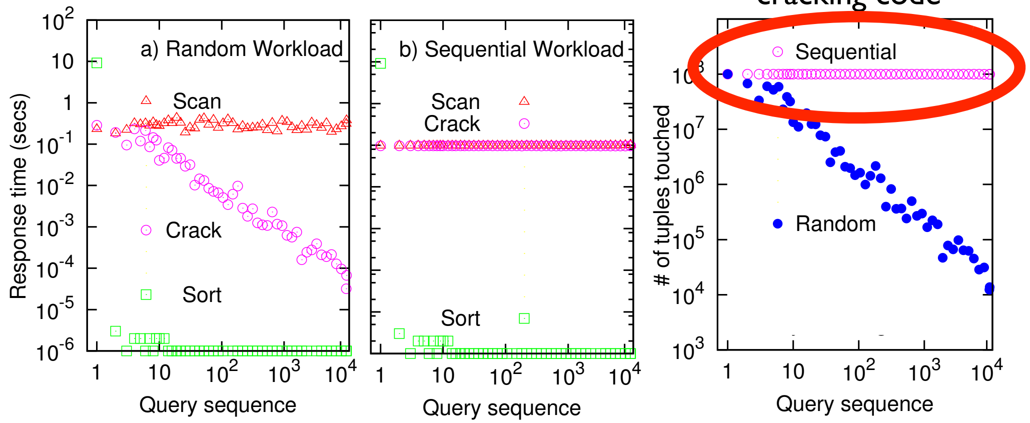
performance degrades to scan

Stochastic Cracking, PVLDB 12

Query patterns

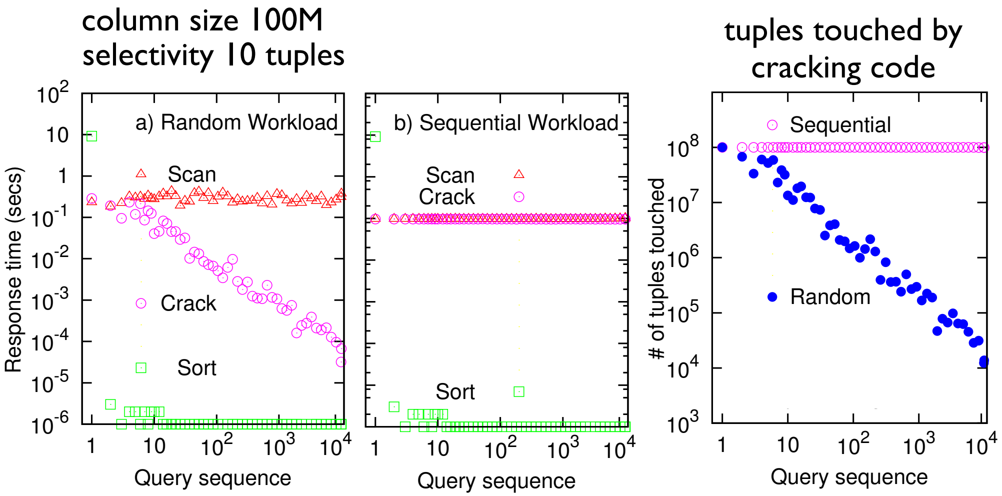
column size 100M
selectivity 10 tuples

tuples touched by
cracking code



performance degrades to scan

Query patterns



performance degrades to scan

Stochastic Cracking, PVLDB 12

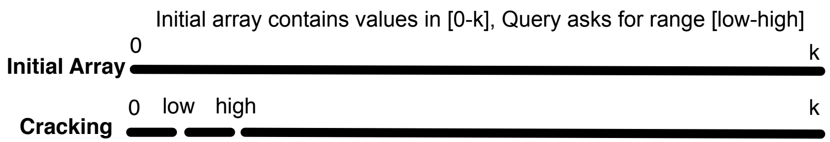
Stochastic Cracking

Problem:
Blind adaptation to queries

Solution:
Query driven and data driven adaptation

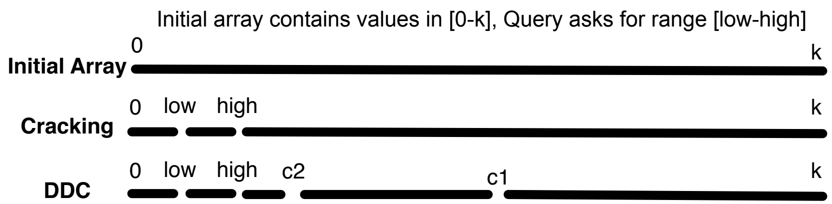
Stochastic Cracking, PVLDB 12

Stochastic Cracking



Stochastic Cracking, PVLDB 12

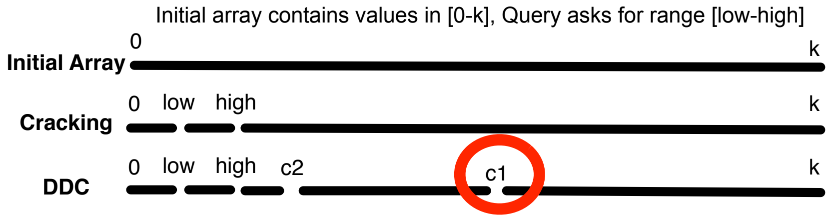
Stochastic Cracking



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

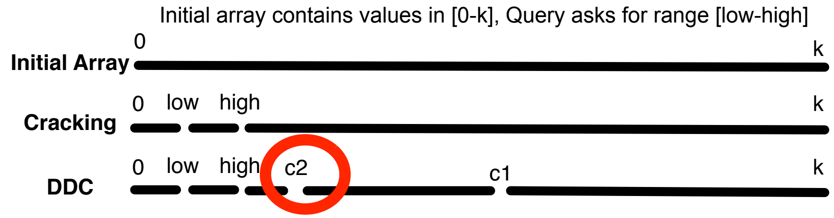


Data Driven, Center (DDC):

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

Stochastic Cracking, PVLDB 12

Stochastic Cracking

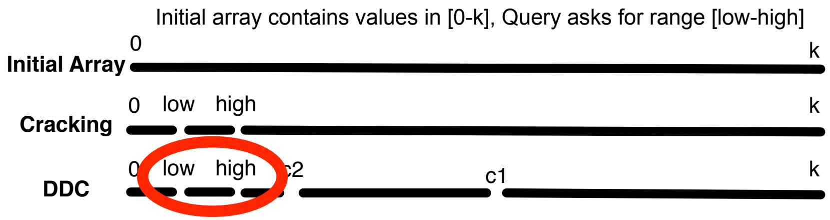


Data Driven, Center (DDC):

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

Stochastic Cracking, PVLDB 12

Stochastic Cracking

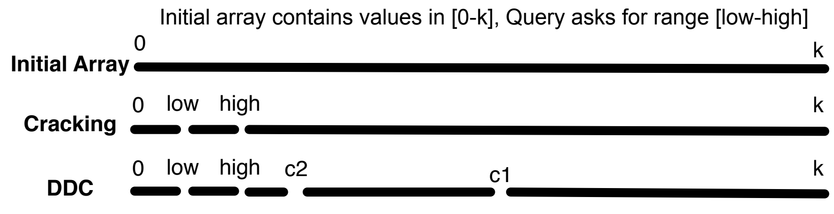


Data Driven, Center (DDC):

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

Stochastic Cracking, PVLDB 12

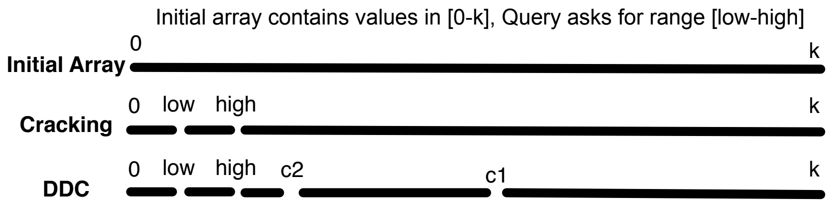
Stochastic Cracking



Data Driven, Center (DDC):

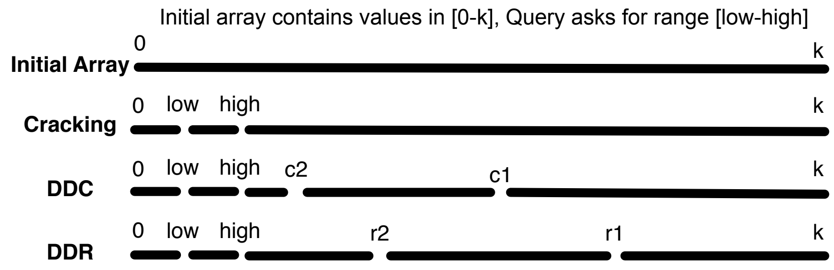
1. Recursively crack a piece in exactly half until in L2 cache.
2. Then crack for the query bounds.

Stochastic Cracking



Stochastic Cracking, PVLDB 12

Stochastic Cracking

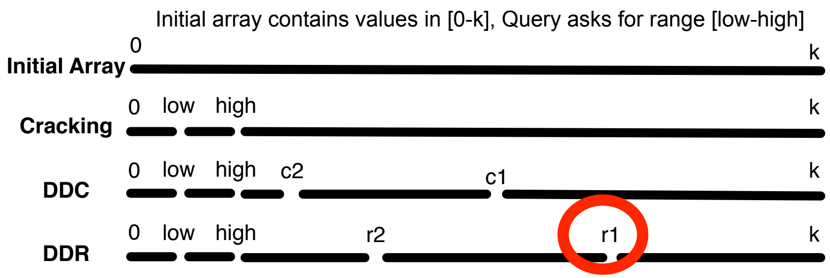


Data Driven, Random (DDR):

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

Stochastic Cracking, PVLDB 12

Stochastic Cracking

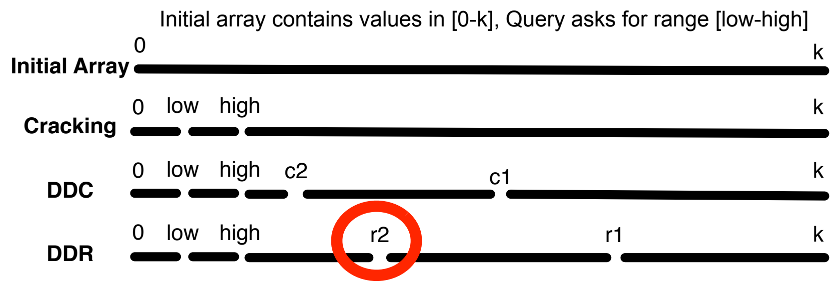


Data Driven, Random (DDR):

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

Stochastic Cracking, PVLDB 12

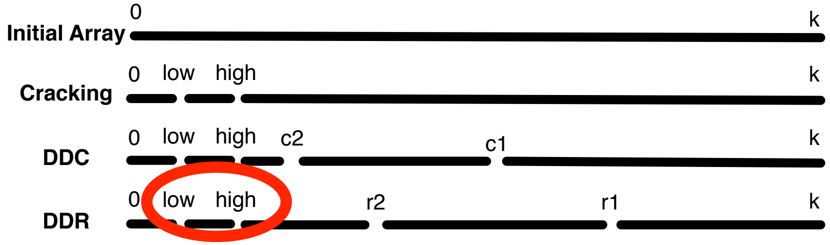
Stochastic Cracking



Data Driven, Random (DDR):

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

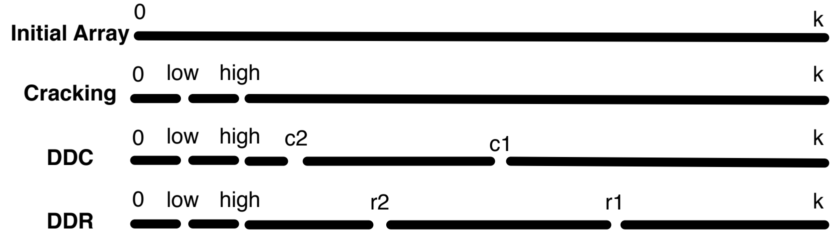
Stochastic Cracking



Data Driven, Random (DDR):

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

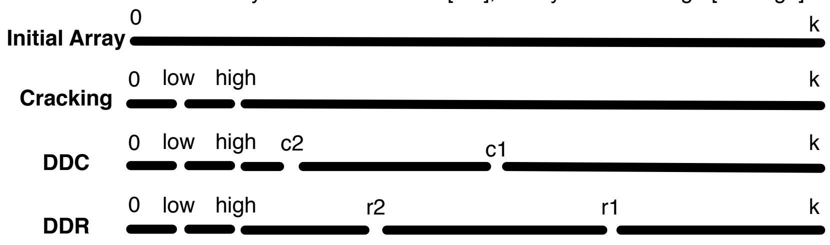
Stochastic Cracking



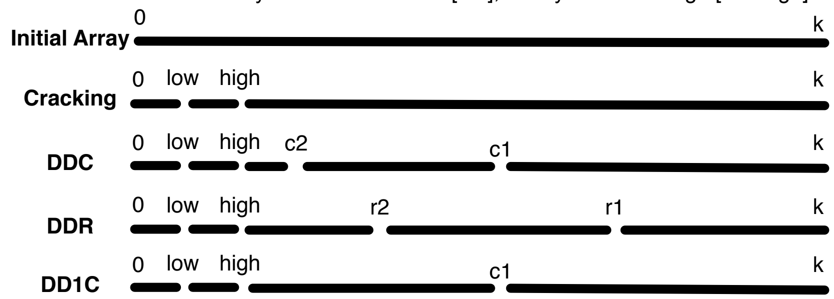
Data Driven, Random (DDR):

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

Stochastic Cracking

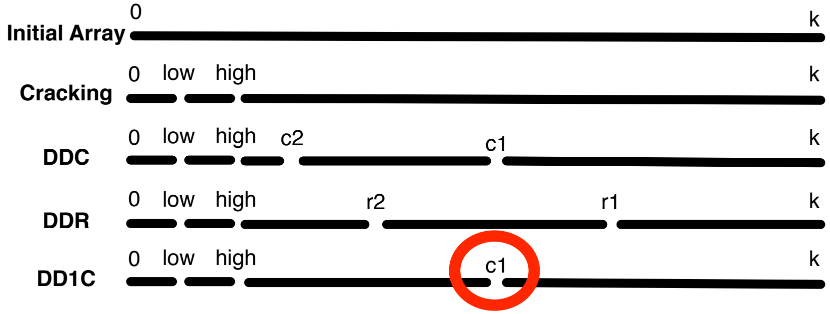


Stochastic Cracking



Stochastic Cracking

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

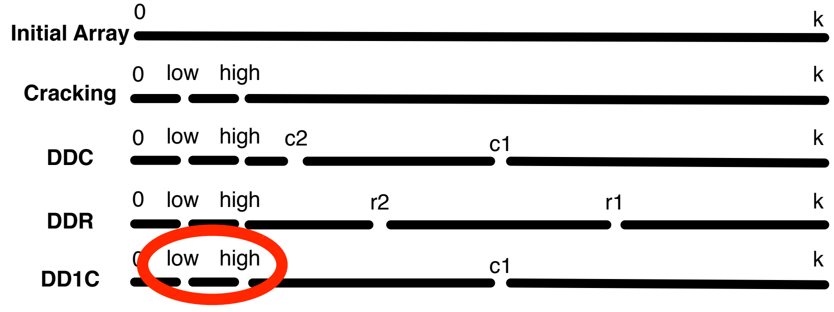


monetdb

Stochastic Cracking, PVLDB 12

Stochastic Cracking

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

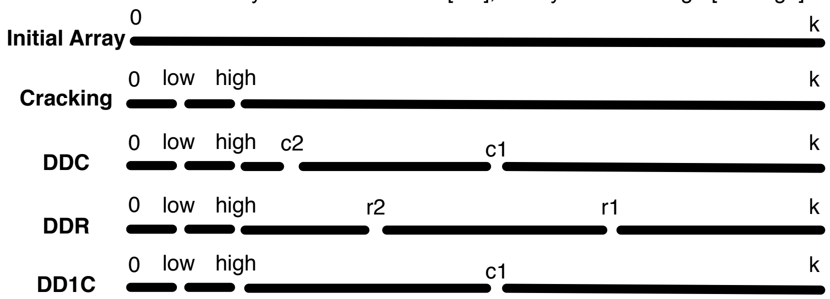


monetdb

Stochastic Cracking, PVLDB 12

Stochastic Cracking

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

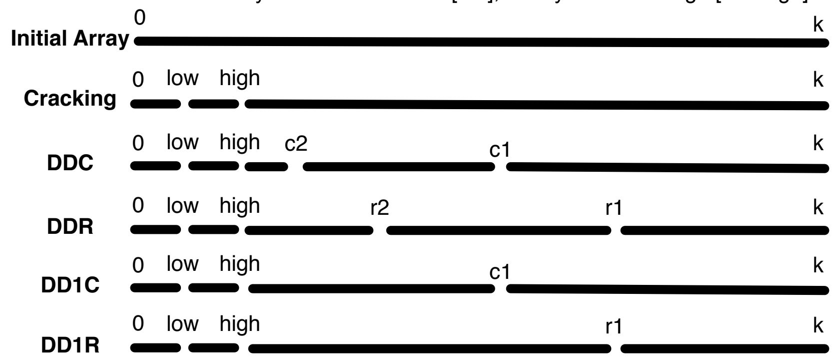


monetdb

Stochastic Cracking, PVLDB 12

Stochastic Cracking

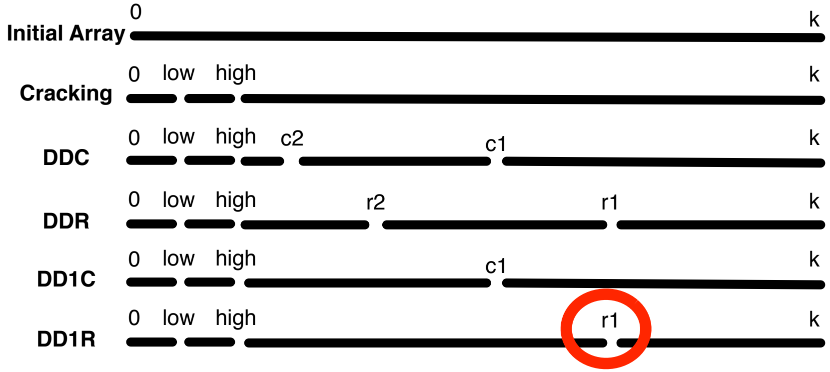
Initial array contains values in [0-k], Query asks for range [low-high]



monetdb

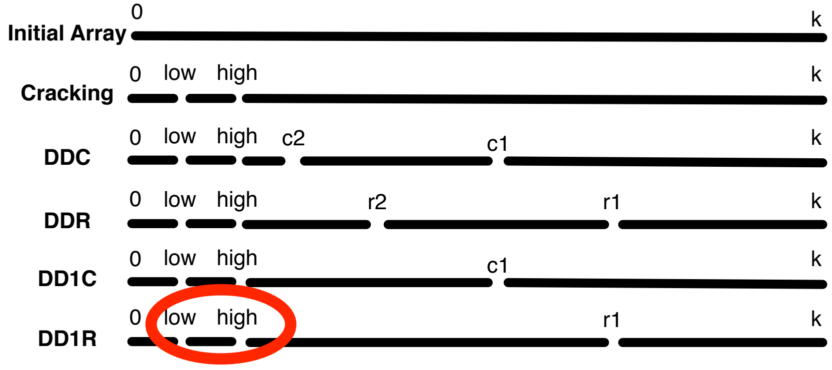
Stochastic Cracking

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



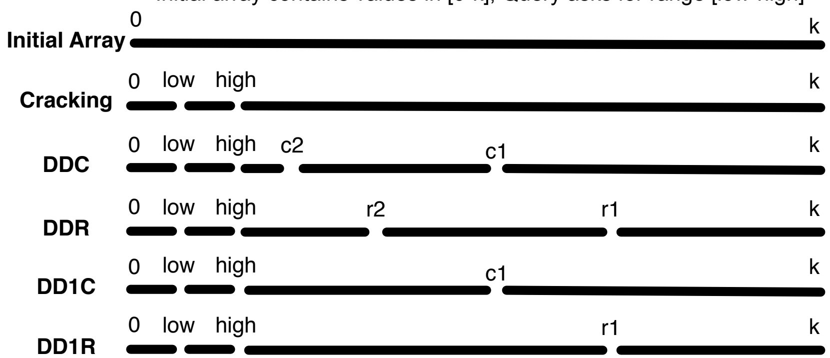
Stochastic Cracking

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



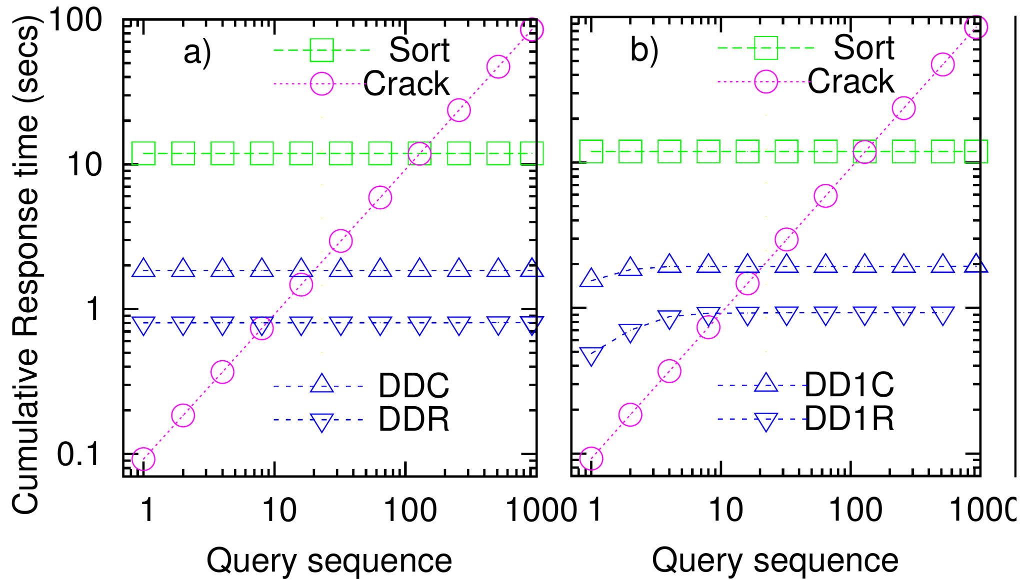
Stochastic Cracking

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



Database Architectures

Stochastic Cracking



Hybrids

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

Adaptive Merging

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

Incremental sort via external merge sort steps

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

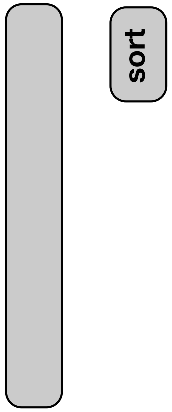
select(A,50,100)

Adaptive Merging

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

Incremental sort via external merge sort steps

select(A,50,100)



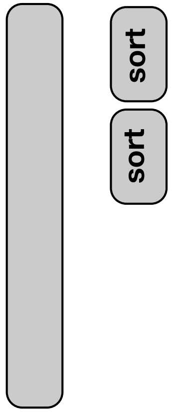
Hybrids PVLDB 2011

Adaptive Merging

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

Incremental sort via external merge sort steps

select(A,50,100)



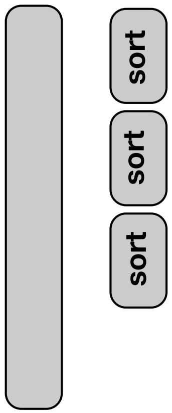
Hybrids PVLDB 2011

Adaptive Merging

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

Incremental sort via external merge sort steps

select(A,50,100)



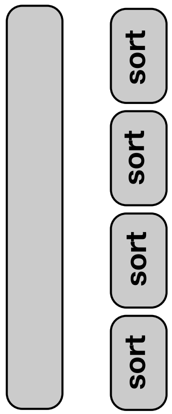
Hybrids PVLDB 2011

Adaptive Merging

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

Incremental sort via external merge sort steps

select(A,50,100)

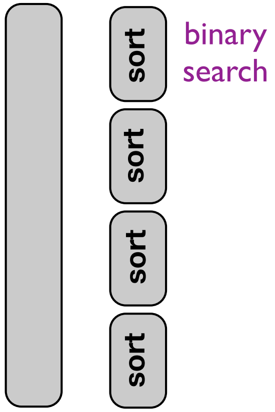


Adaptive Merging

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

Incremental sort via external merge sort steps

select(A,50,100)



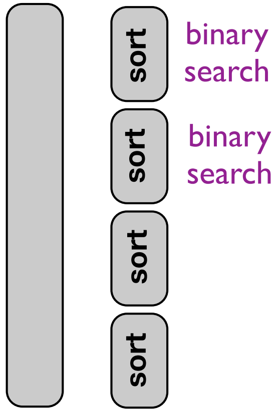
Hybrids PVLDB 2011

Adaptive Merging

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

Incremental sort via external merge sort steps

select(A,50,100)



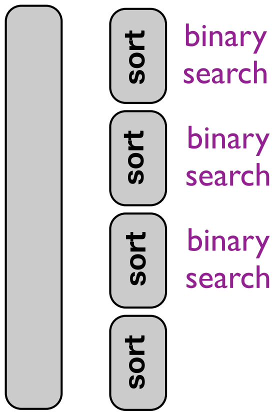
Hybrids PVLDB 2011

Adaptive Merging

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

Incremental sort via external merge sort steps

select(A,50,100)



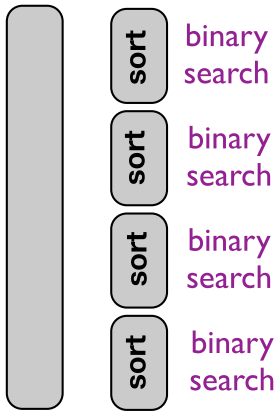
Hybrids PVLDB 2011

Adaptive Merging

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

Incremental sort via external merge sort steps

select(A,50,100)

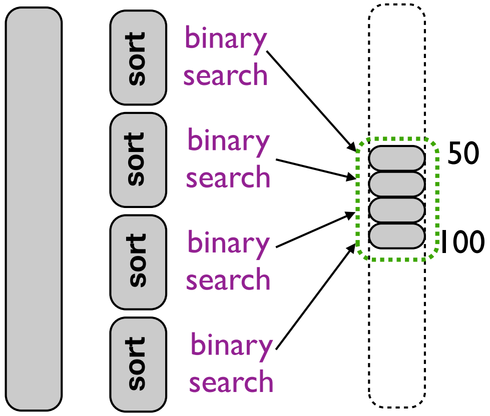


Adaptive Merging

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

Incremental sort via external merge sort steps

select(A,50,100)



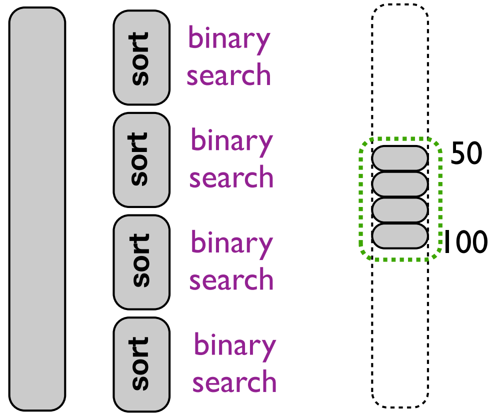
Hybrids PVLDB 2011

Adaptive Merging

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

Incremental sort via external merge sort steps

select(A,50,100)



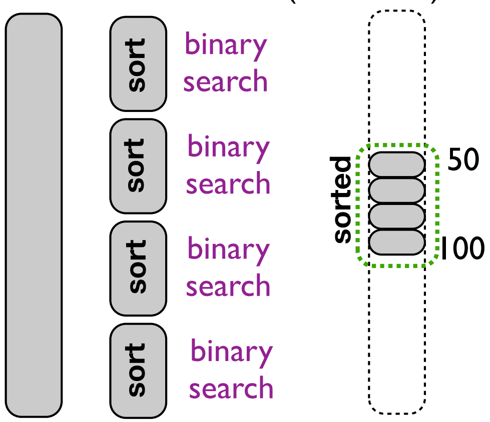
Hybrids PVLDB 2011

Adaptive Merging

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

Incremental sort via external merge sort steps

select(A,50,100)



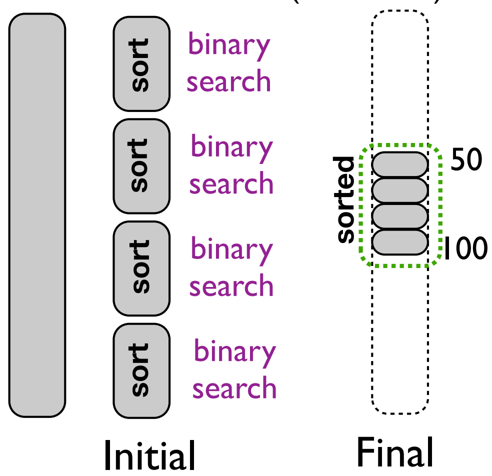
Hybrids PVLDB 2011

Adaptive Merging

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

Incremental sort via external merge sort steps

select(A,50,100)

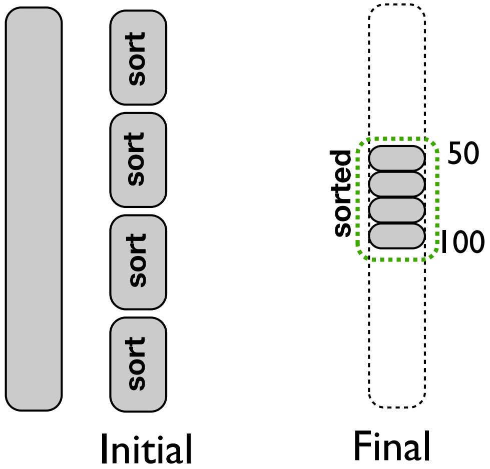


Adaptive Merging

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

Incremental sort via external merge sort steps

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

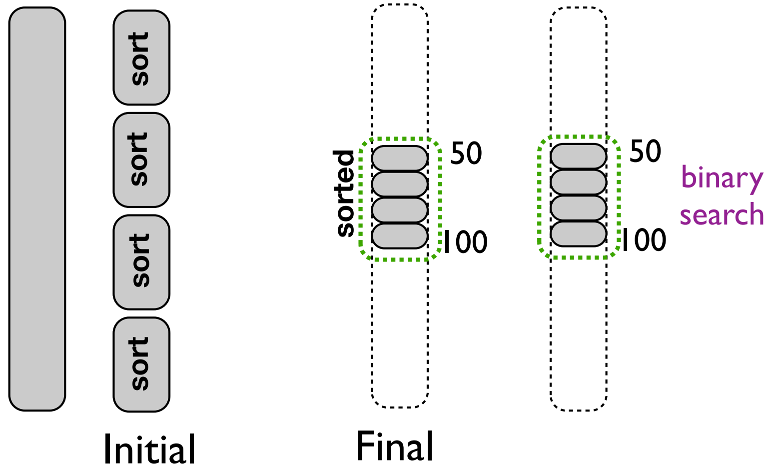


Adaptive Merging

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

Incremental sort via external merge sort steps

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

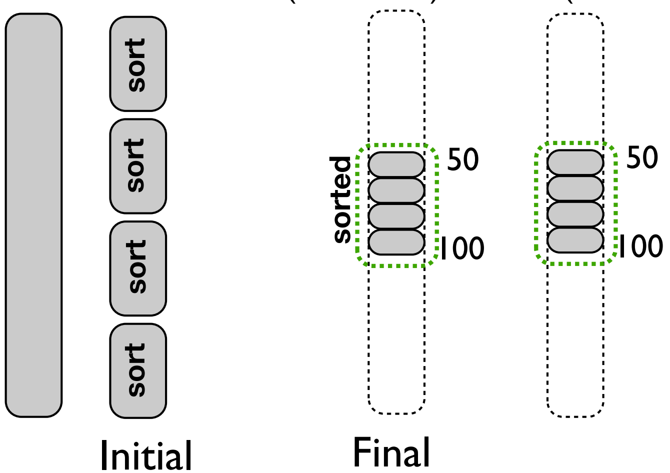


Adaptive Merging

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)

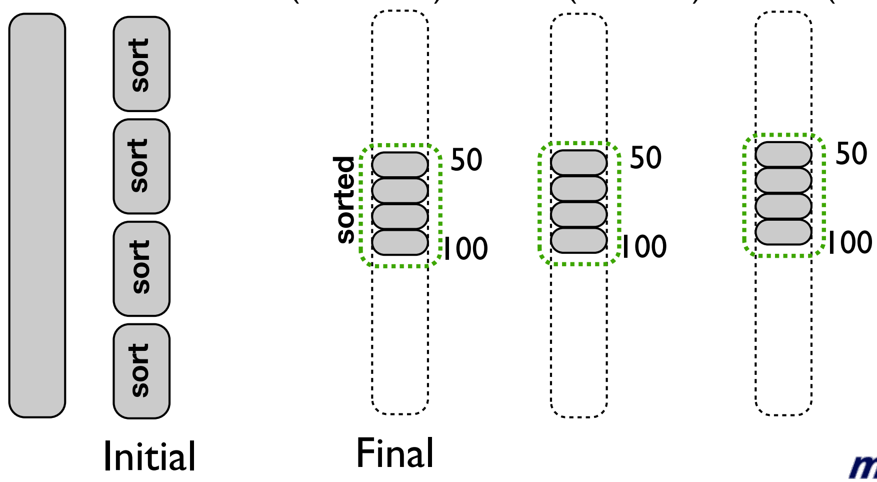


Adaptive Merging

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

Incremental sort via external merge sort steps

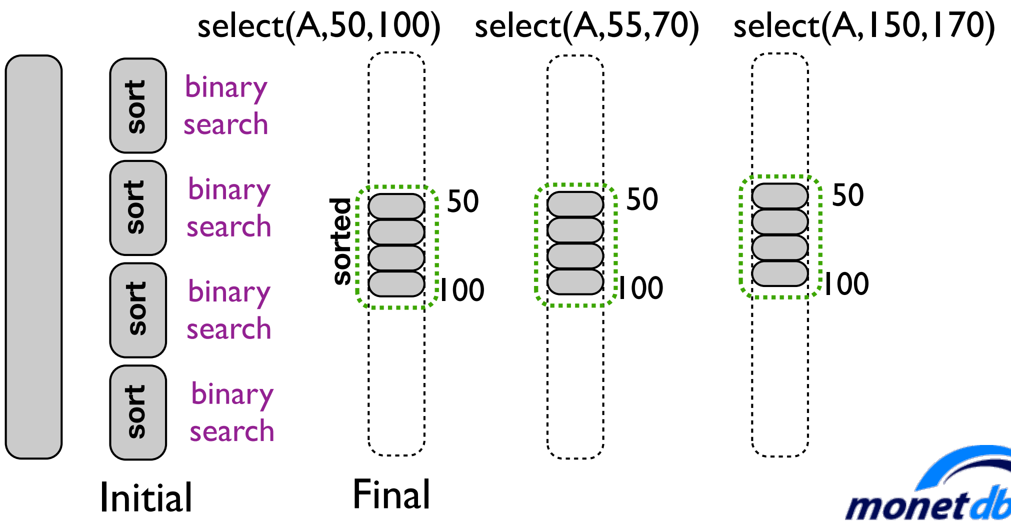
select(A,50,100)   select(A,55,70)   select(A,150,170)



Adaptive Merging

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

Incremental sort via external merge sort steps

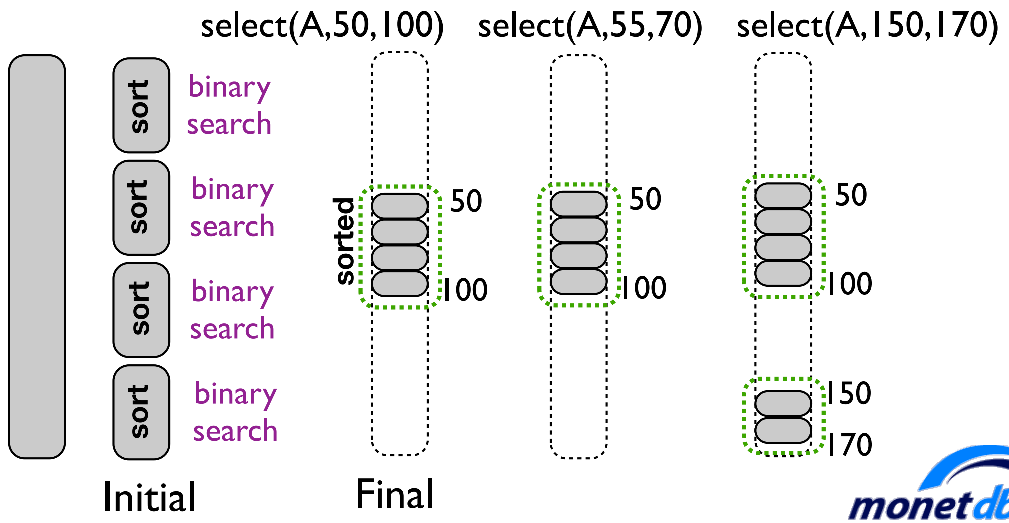


Hybrids PVLDB 2011

Adaptive Merging

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

Incremental sort via external merge sort steps

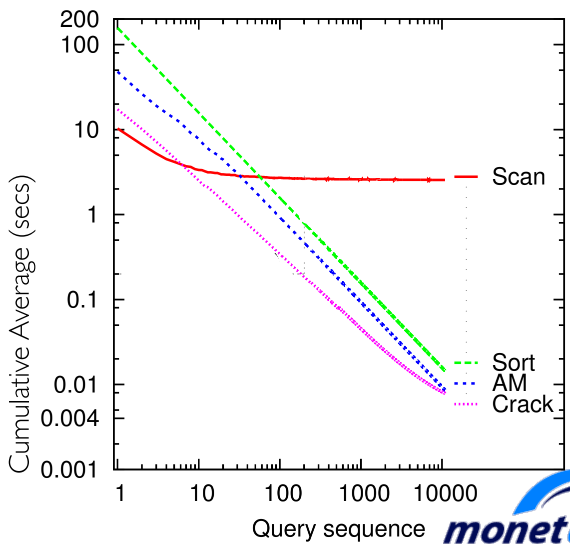


Hybrids PVLDB 2011

Performance Analysis

set-up

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

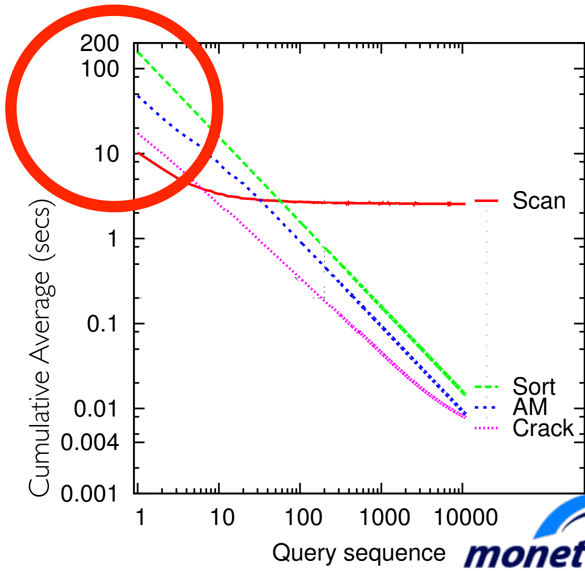


Hybrids PVLDB 2011

Performance Analysis

set-up

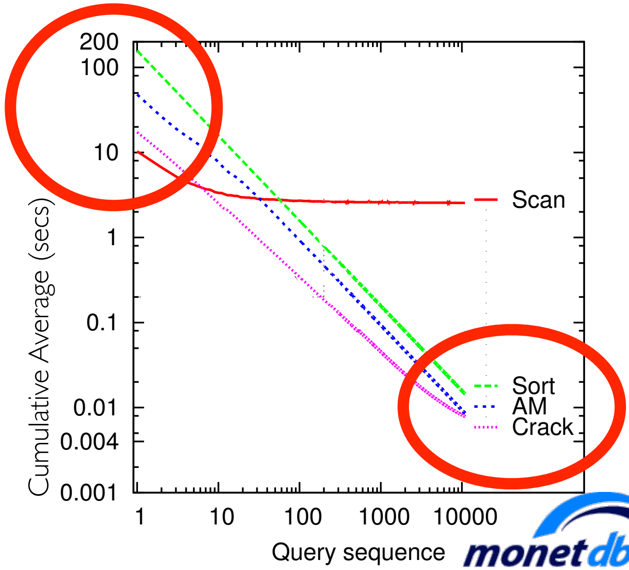
10K random selections
selectivity 10%
random value ranges
in a 30 million integer column



Performance Analysis

set-up

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

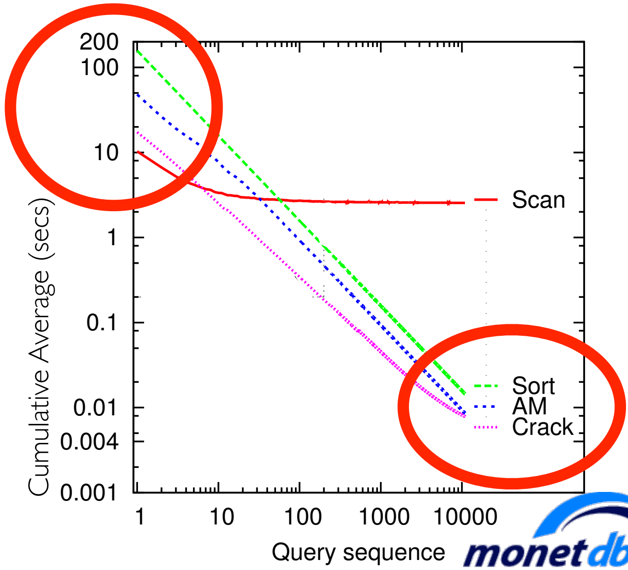


Performance Analysis

set-up

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

AM: high init overhead but fast convergence



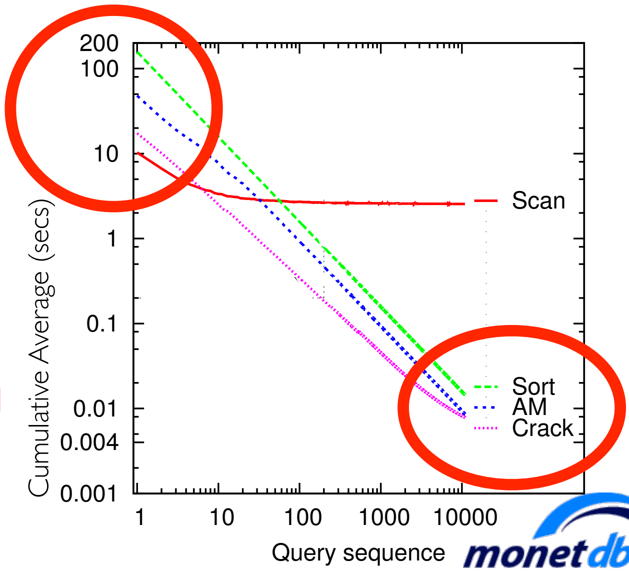
Performance Analysis

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

Adaptive merging and Cracking are extremes

What is there in between?

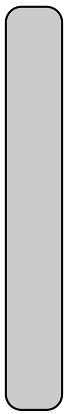
Crack-Crack

vary initialization and incremental steps taken

Hybrids PVLDB 2011

Crack-Crack

vary initialization and incremental steps taken

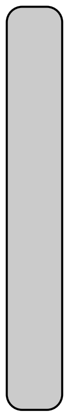


Hybrids PVLDB 2011

Crack-Crack

vary initialization and incremental steps taken

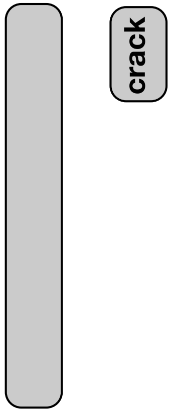
select(A,50,100)



Crack-Crack

vary initialization and incremental steps taken

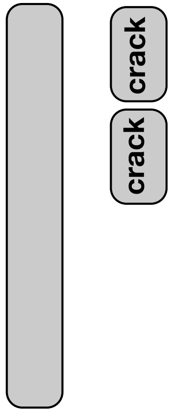
select(A,50,100)



Crack-Crack

vary initialization and incremental steps taken

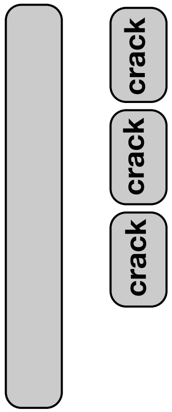
select(A,50,100)



Crack-Crack

vary initialization and incremental steps taken

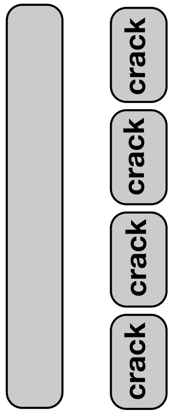
select(A,50,100)



Crack-Crack

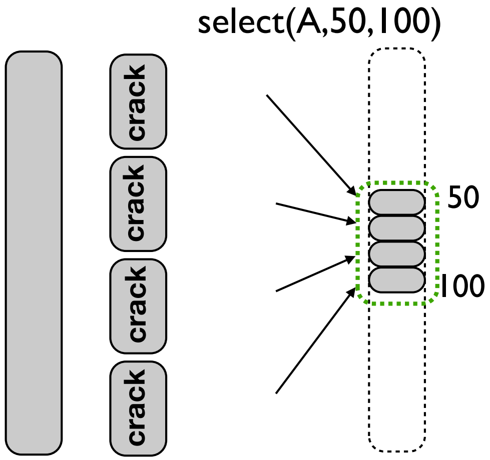
vary initialization and incremental steps taken

select(A,50,100)



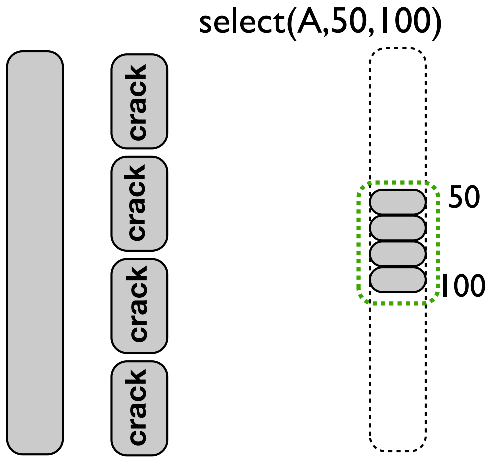
Crack-Crack

vary initialization and incremental steps taken



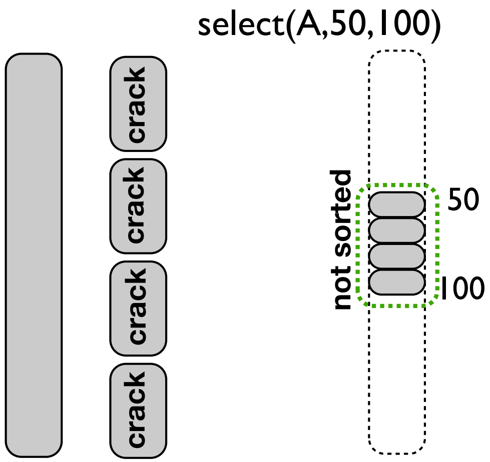
Crack-Crack

vary initialization and incremental steps taken



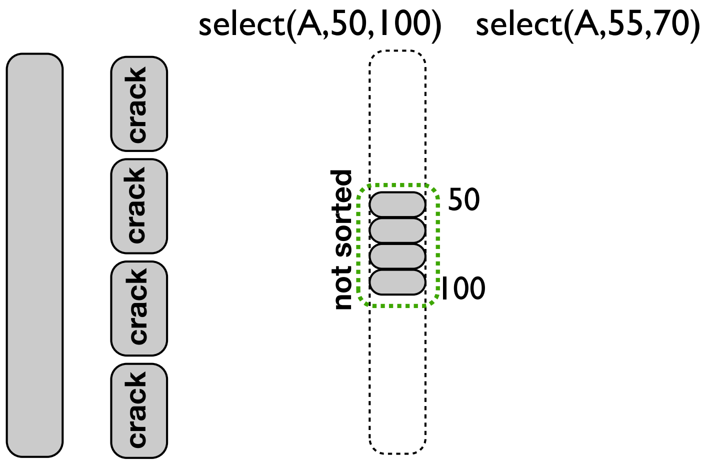
Crack-Crack

vary initialization and incremental steps taken



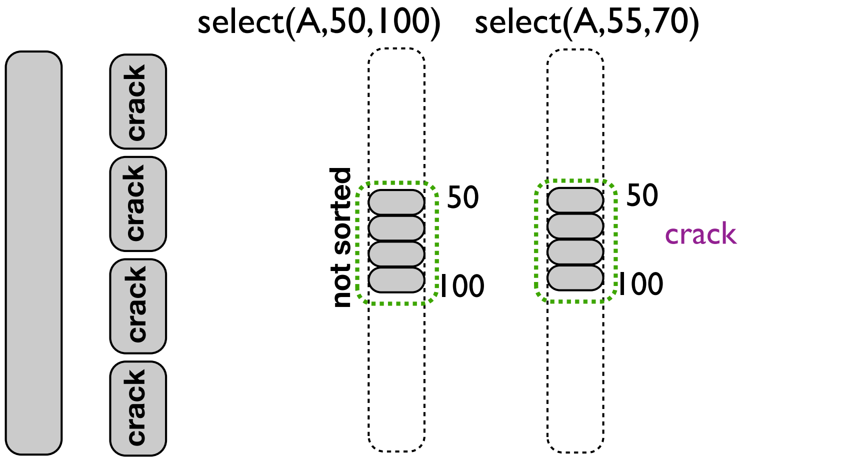
Crack-Crack

vary initialization and incremental steps taken



Crack-Crack

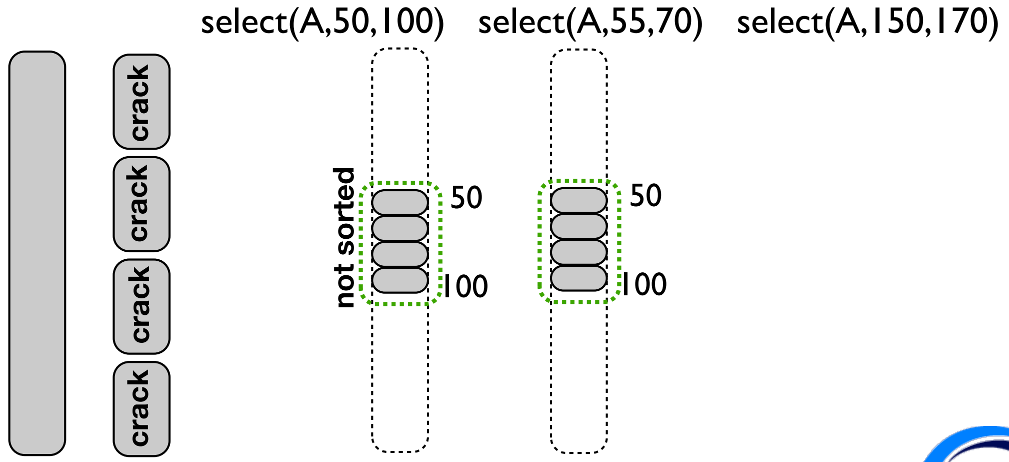
vary initialization and incremental steps taken



Hybrids PVLDB 2011

Crack-Crack

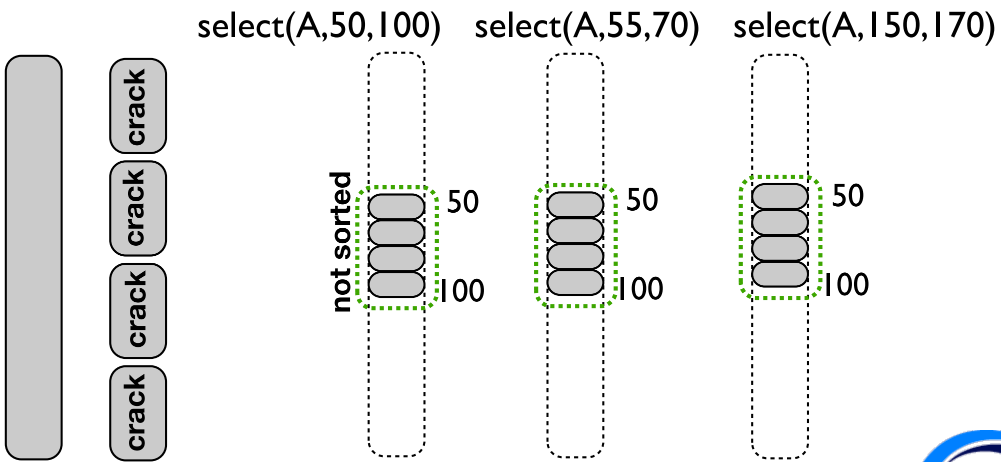
vary initialization and incremental steps taken



Hybrids PVLDB 2011

Crack-Crack

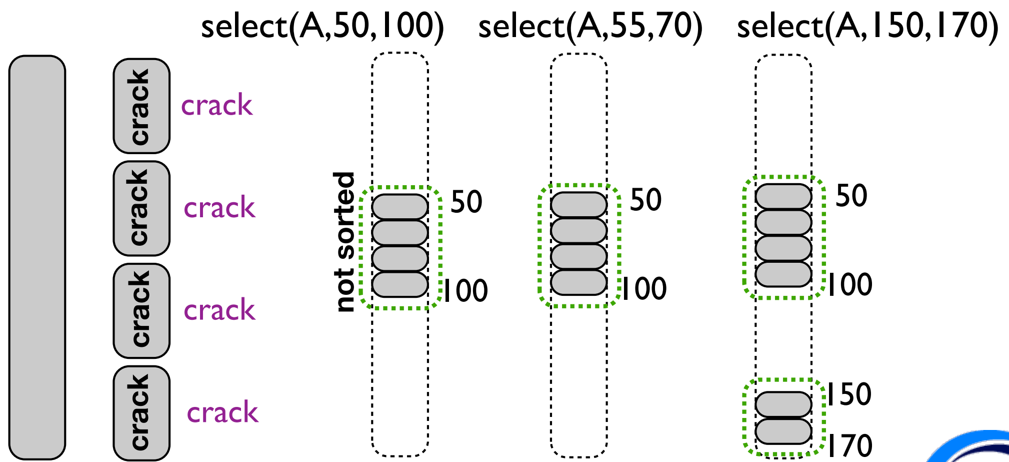
vary initialization and incremental steps taken



Hybrids PVLDB 2011

Crack-Crack

vary initialization and incremental steps taken



Adaptive Indexing

high – overhead – low

low – overhead – high

slow – convergence – fast

initial partitions \ final partitions	Sort	Radix	Crack
Sort	HSS	HSR	HSC
Radix	HRS	HRR	HRC
Crack	HCS	HCR	HCC

fast – convergence – slow

Hybrids PVLDB 2011

Adaptive Indexing

high – overhead – low

low – overhead – high

slow – convergence – fast

initial partitions \ final partitions	Sort	Radix	Crack
Sort	HSS	HSR	HSC
Radix	HRS	HRR	HRC
Crack	HCS	HCR	HCC

fast – convergence – slow

Hybrids PVLDB 2011

Adaptive Indexing

high – overhead – low

low – overhead – high

slow – convergence – fast

initial partitions \ final partitions	Sort	Radix	Crack
Sort	HSS	HSR	HSC
Radix	HRS	HRR	HRC
Crack	HCS	HCR	HCC

fast – convergence – slow

Hybrids PVLDB 2011

Adaptive Indexing

high – overhead – low

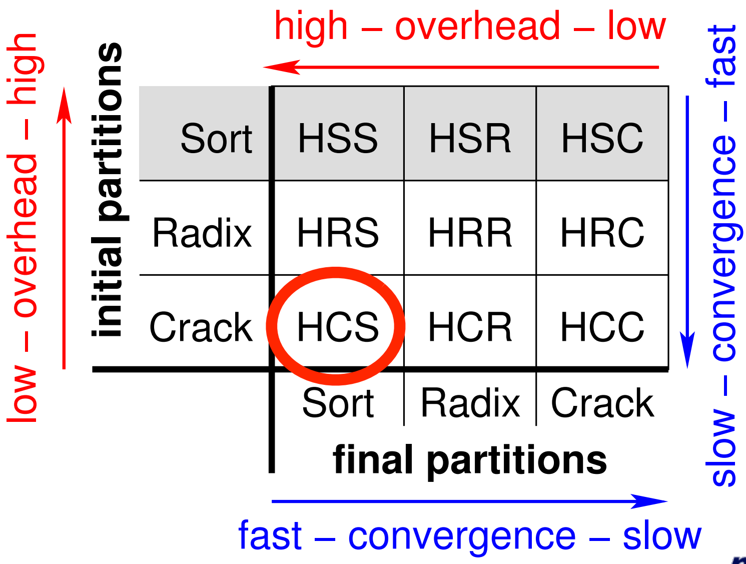
low – overhead – high

slow – convergence – fast

initial partitions \ final partitions	Sort	Radix	Crack
Sort	HSS	HSR	HSC
Radix	HRS	HRR	HRC
Crack	HCS	HCR	HCC

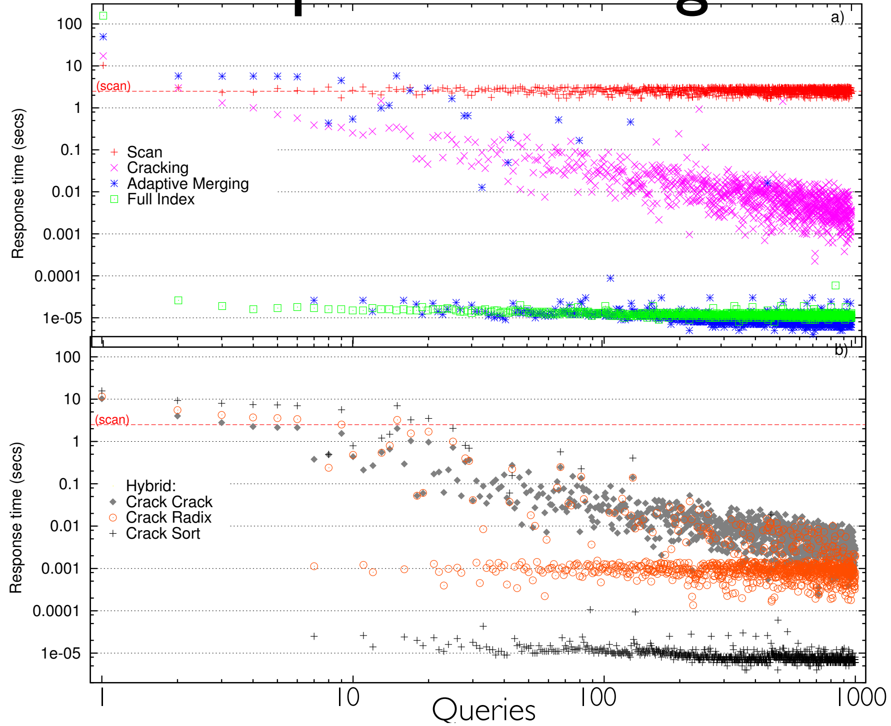
fast – convergence – slow

Adaptive Indexing



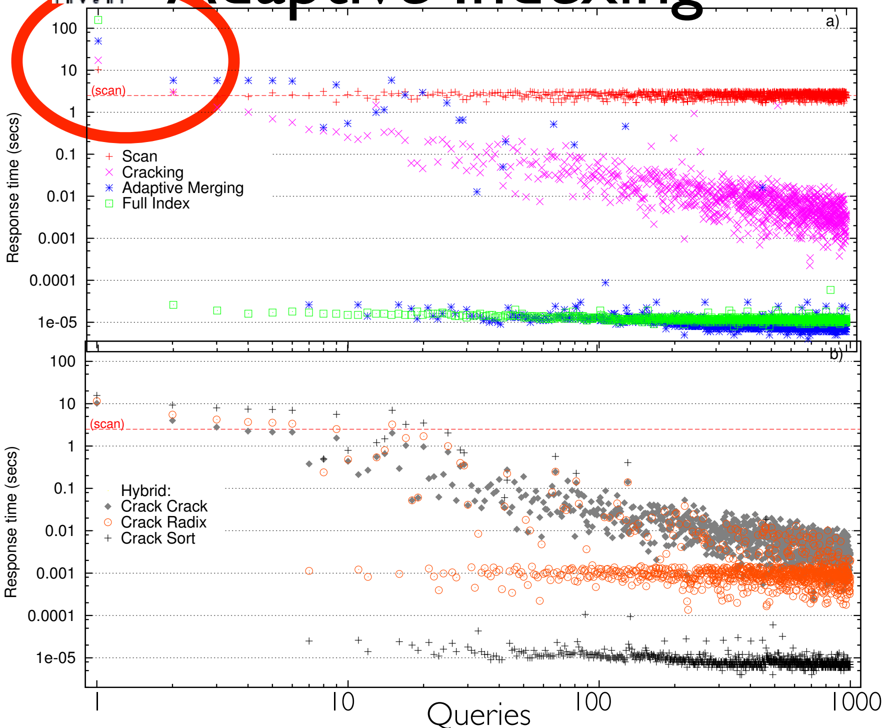
Hybrids PVLDB 2011

Adaptive Indexing



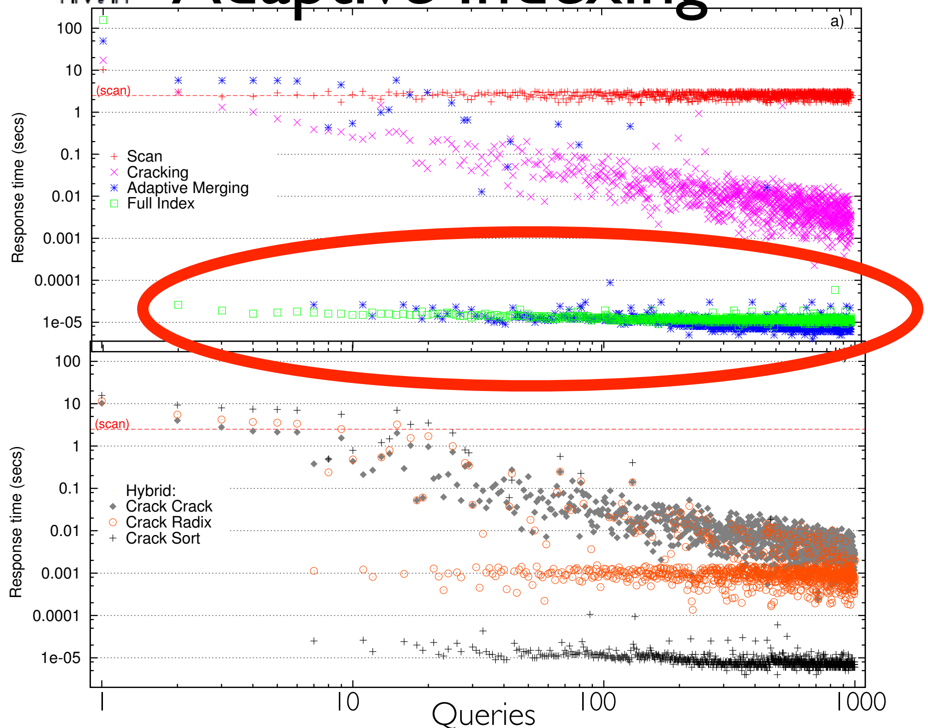
Hybrids PVLDB 2011

Adaptive Indexing

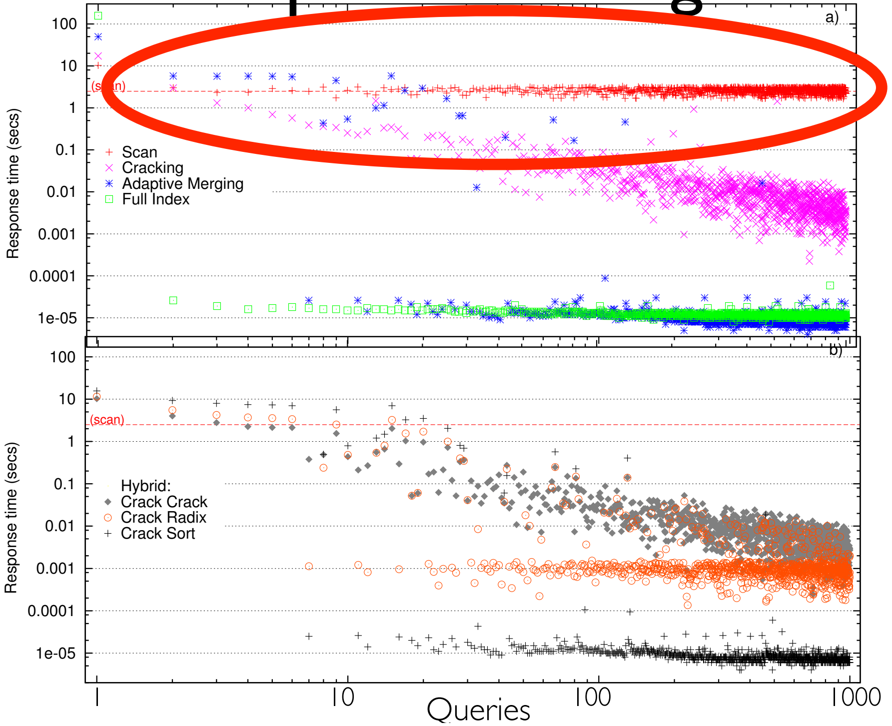


Hybrids PVLDB 2011

Adaptive Indexing

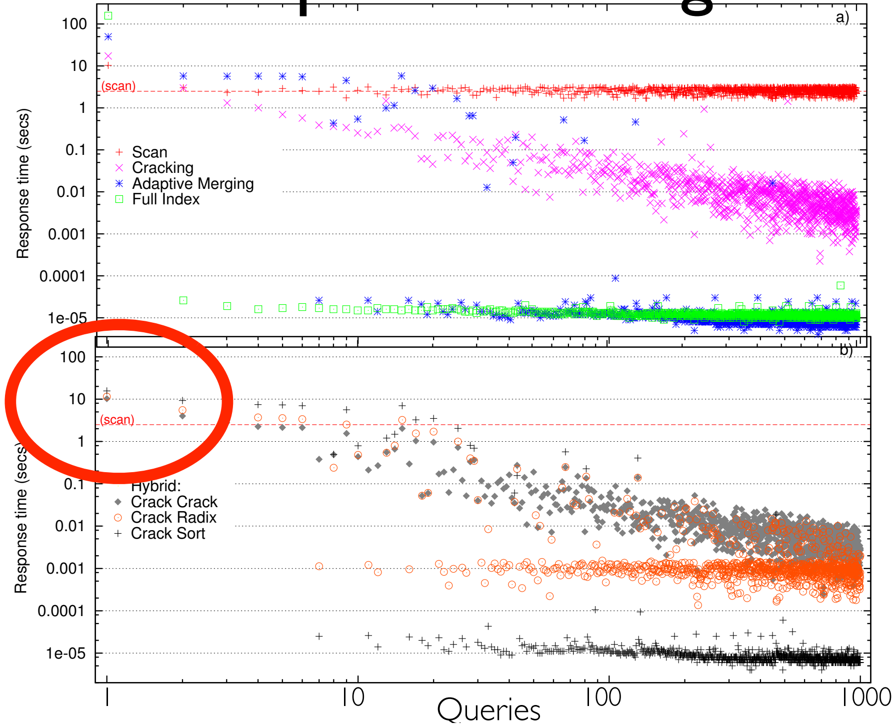


Adaptive Indexing



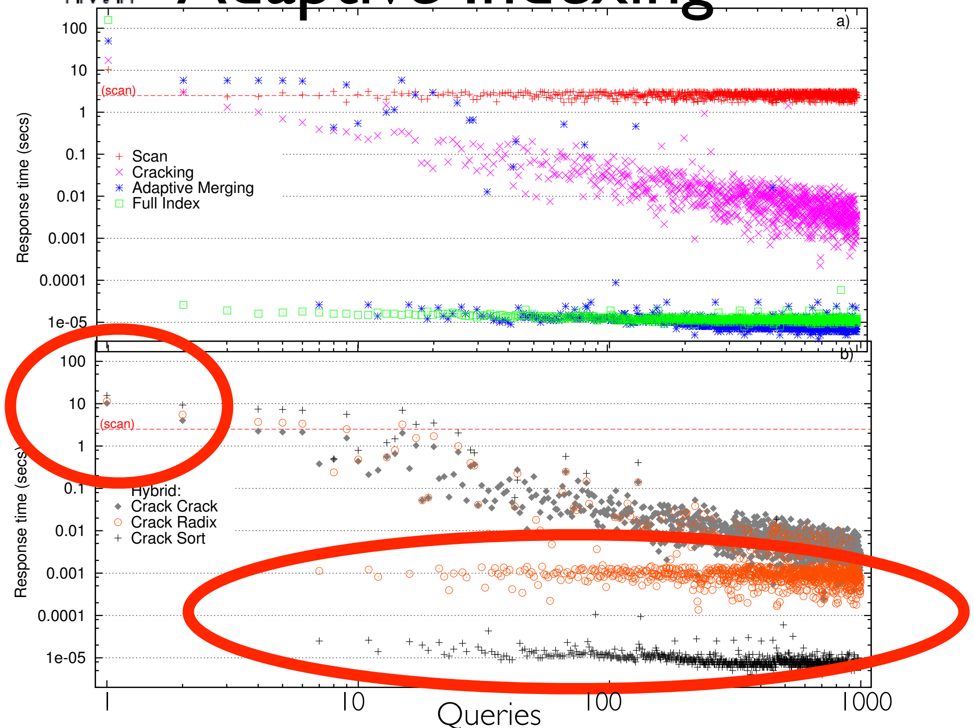
Hybrids PVLDB 2011

Adaptive Indexing



Hybrids PVLDB 2011

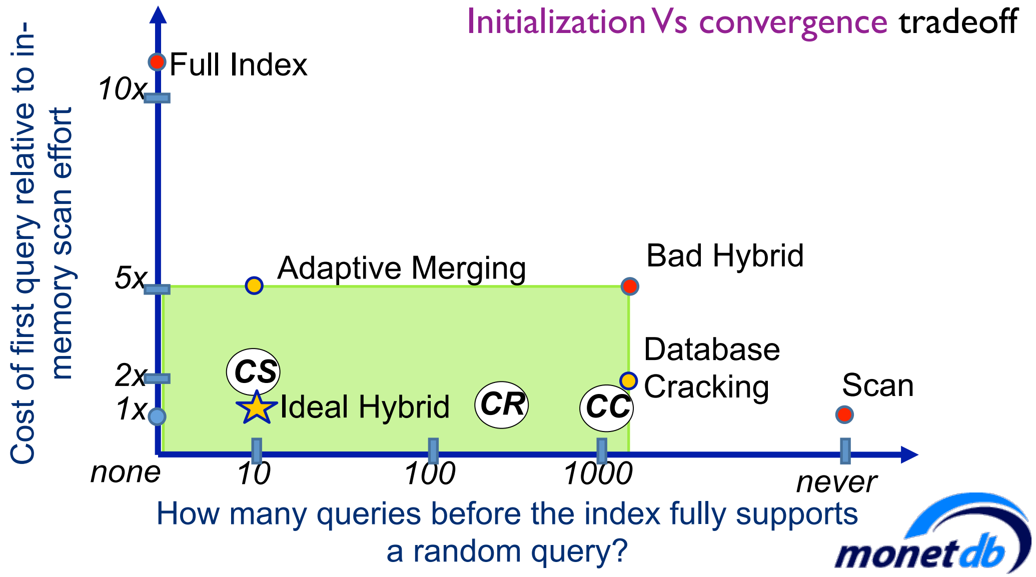
Adaptive Indexing



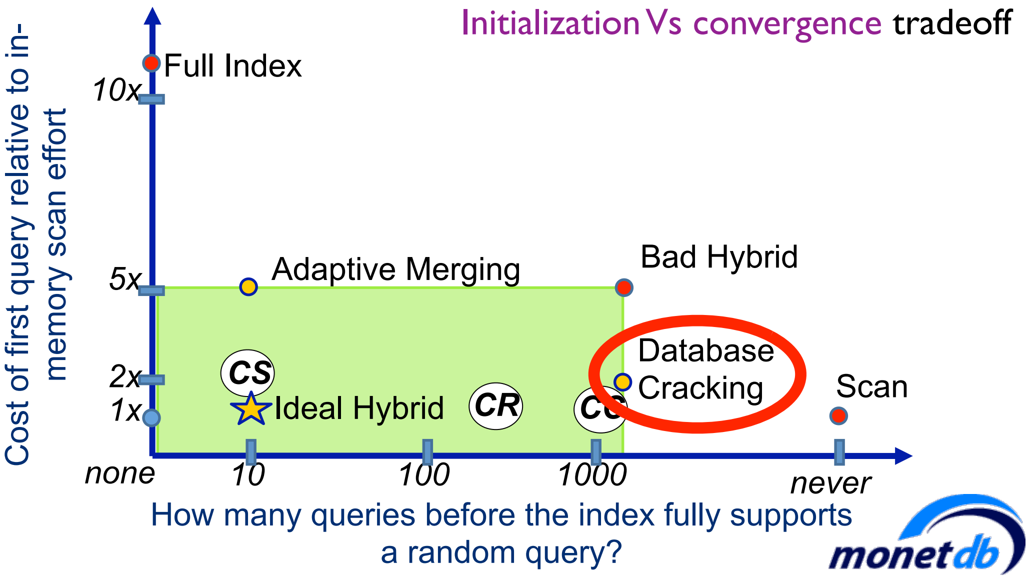
Hybrids PVLDB 2011

Adaptive Indexing

Initialization Vs convergence tradeoff

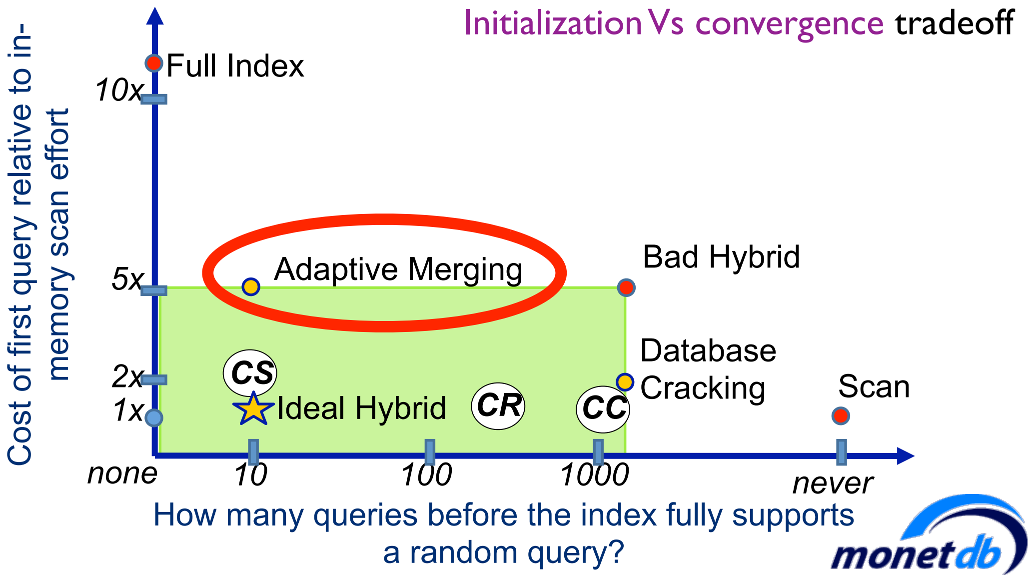


Adaptive Indexing



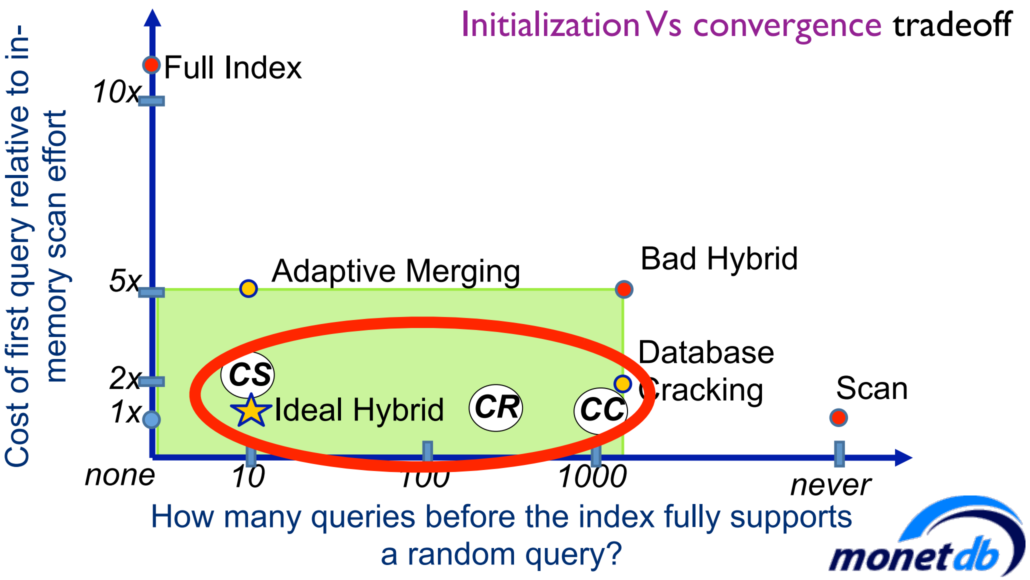
Hybrids PVLDB 2011

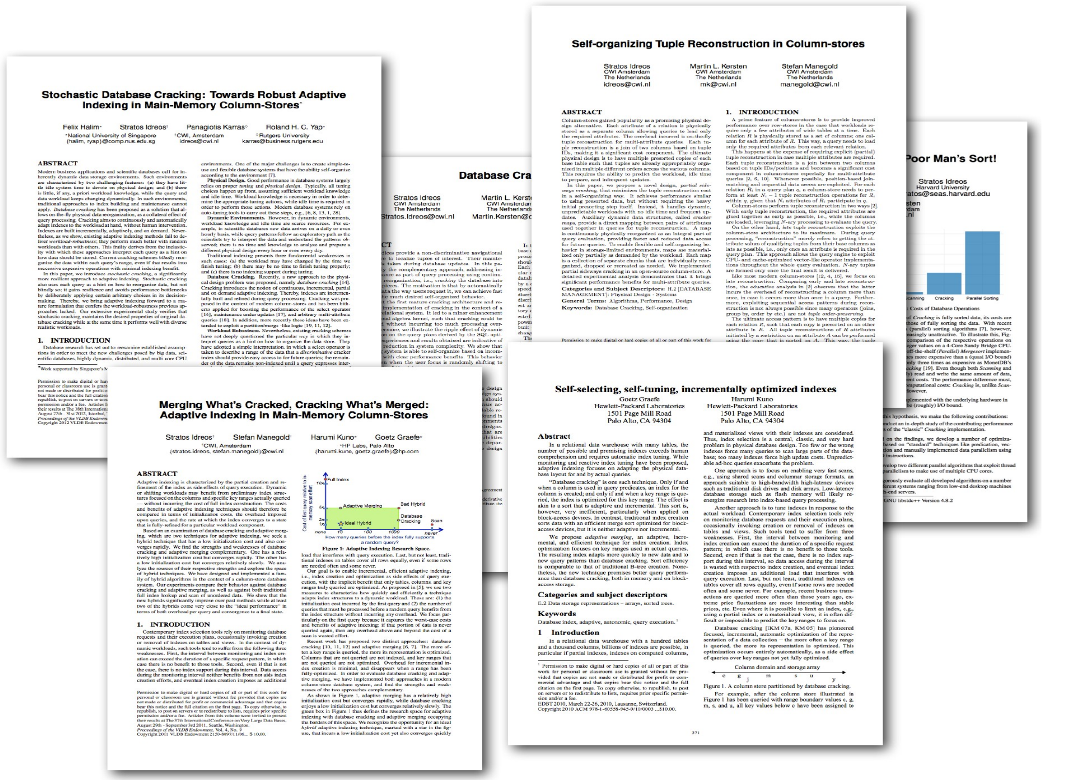
Adaptive Indexing



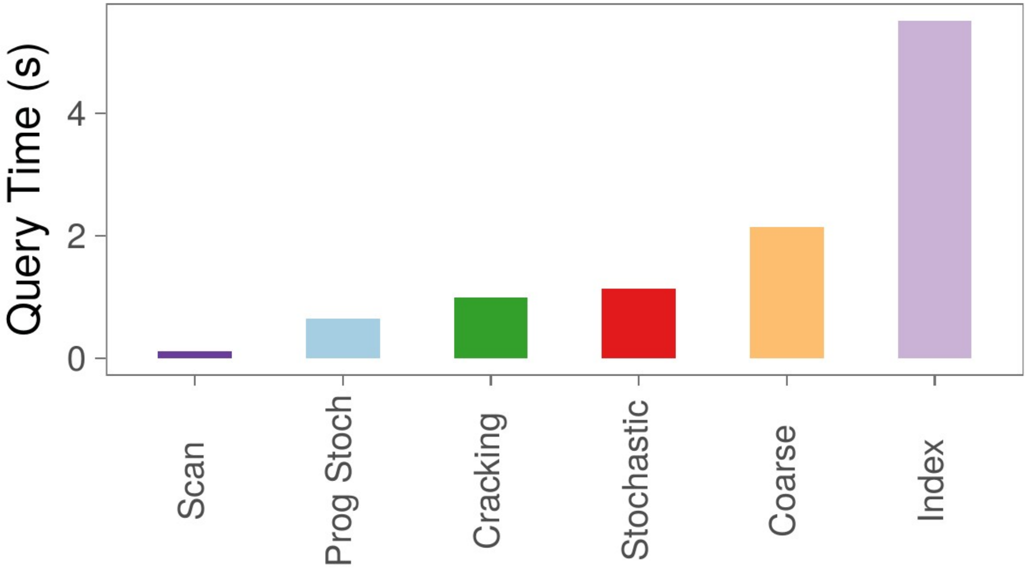
Hybrids PVLDB 2011

Adaptive Indexing





Adaptive Indexing: 1st Query Costs



Progressive Indexing

Can we / how to:

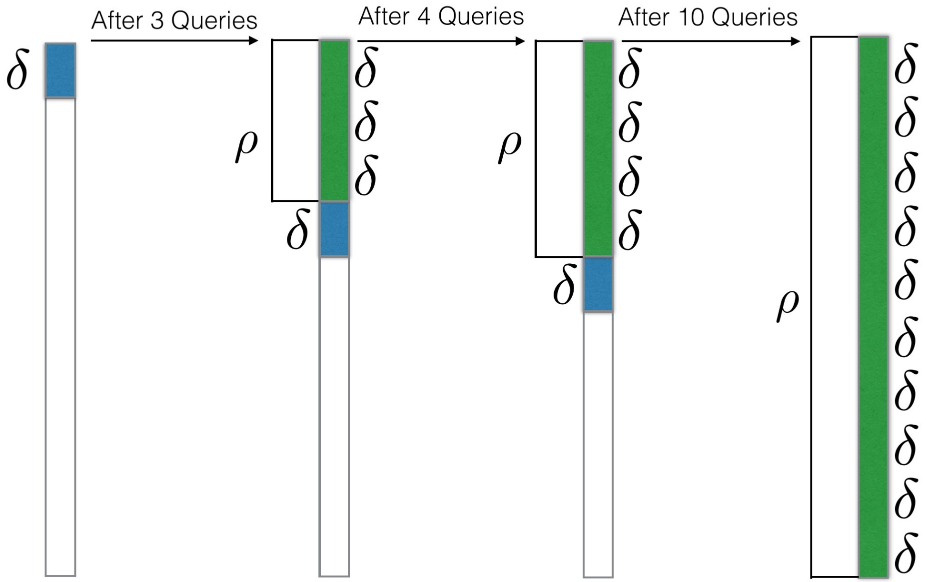
- Reduce / limit 1st query cost / overhead?
- Improve query performance predictability and robustness?
- Ensure convergence towards full index?

Yet unexplored “dimensions”:

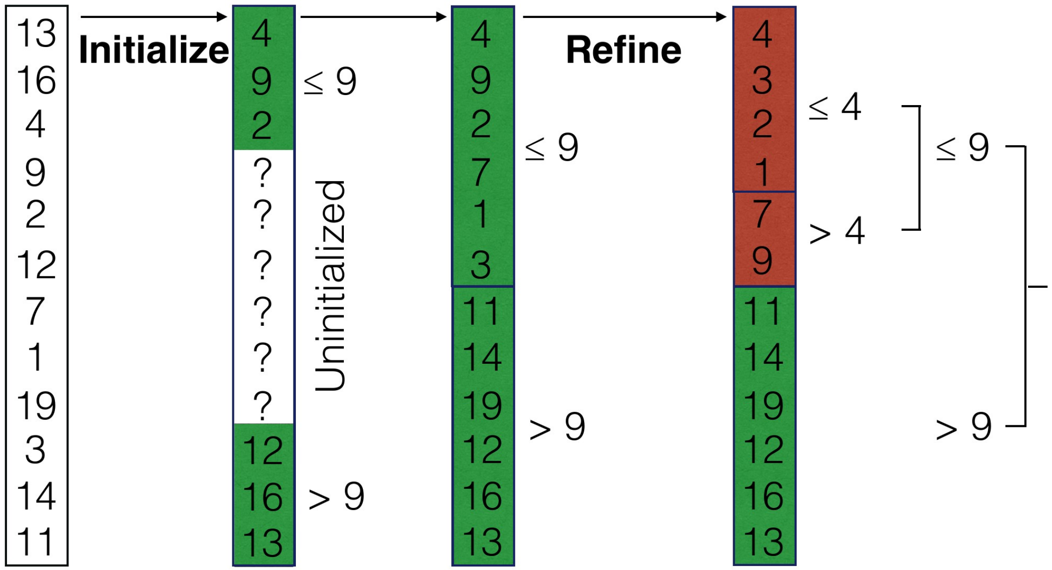
- Other sorting algorithms than quick-sort
- Suspend/resume steps / iterations

Mark Raasveldt, Pedro Holanda, Hannes Mühleisen

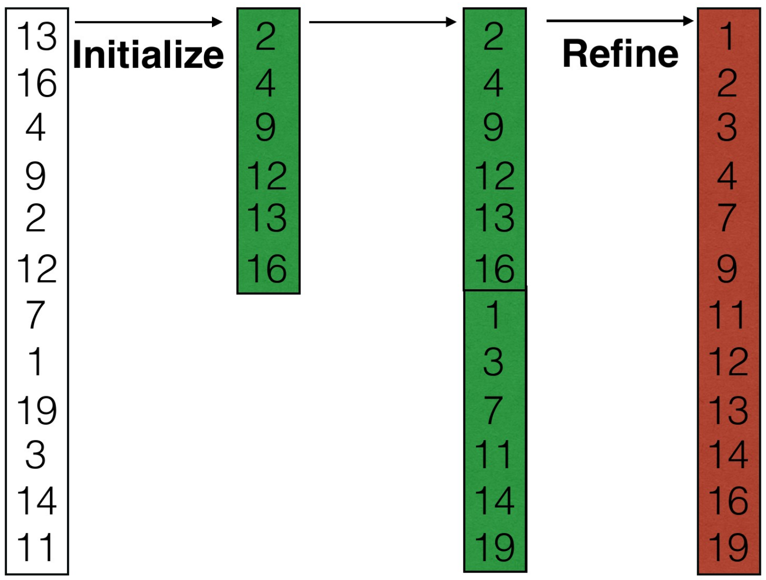
Progressive Indexing



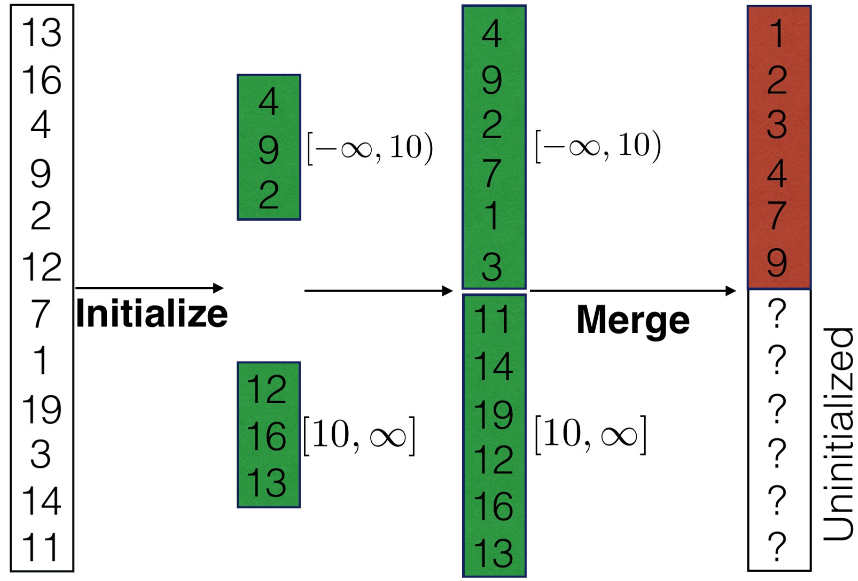
Progressive Quick-Sort



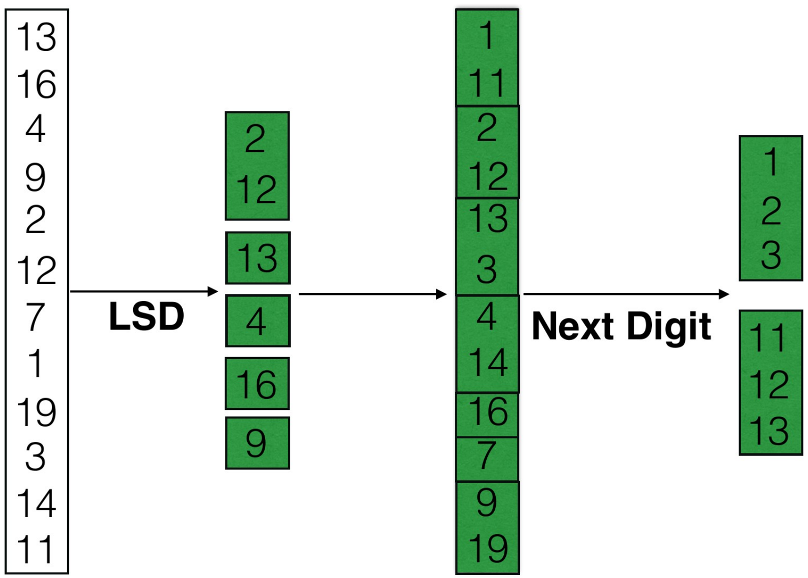
Progressive Merge-Sort



Progressive Bucket-Sort



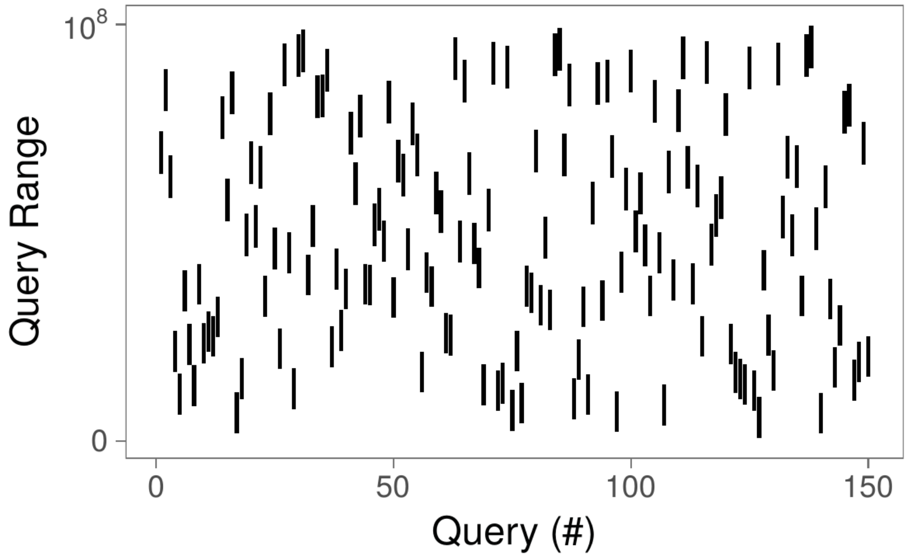
Progressive Radix-Sort



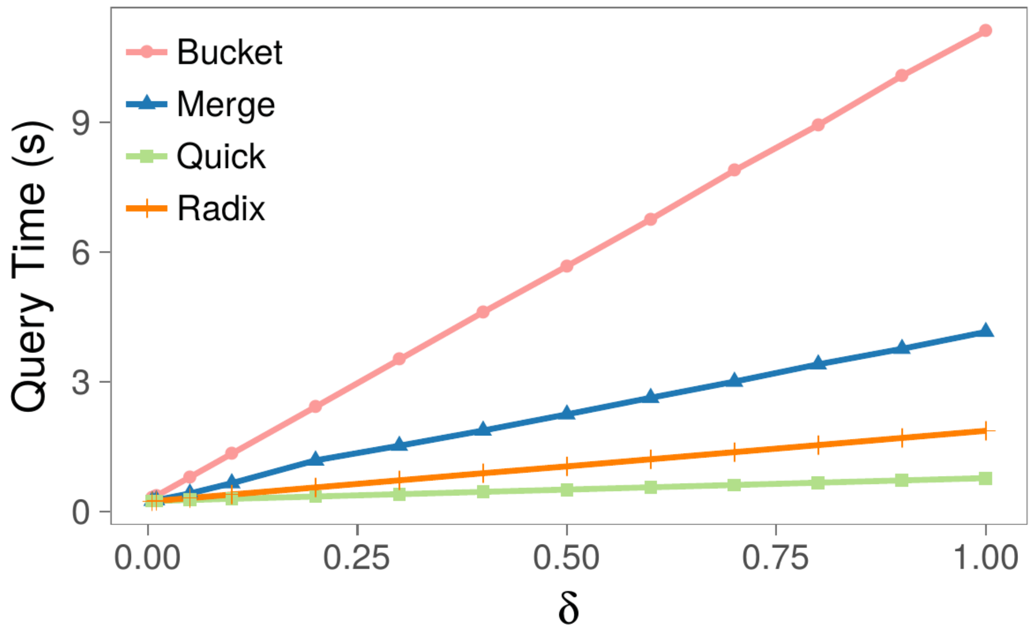
Experimental Setup

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

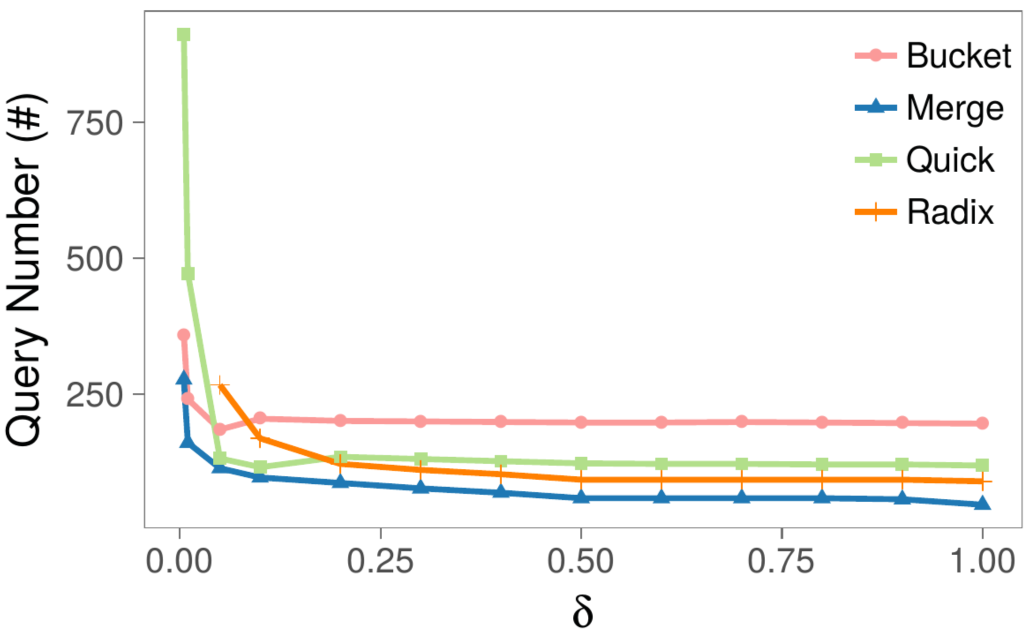
Random Workload



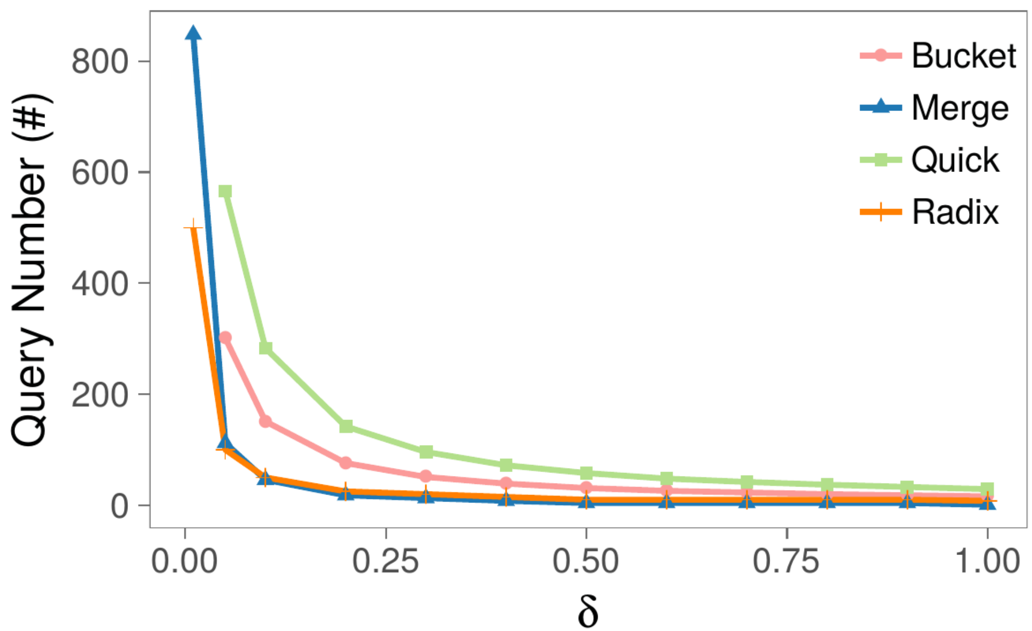
Varying δ: 1st Query Cost



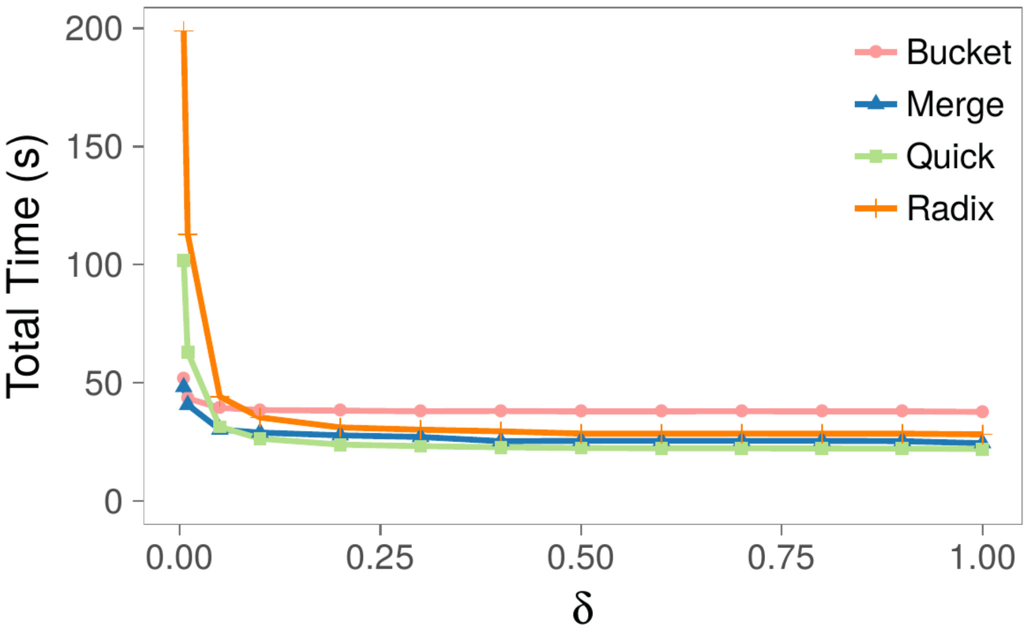
Varying δ: # Queries until Pay-off



Varying δ: # Queries until Convergence



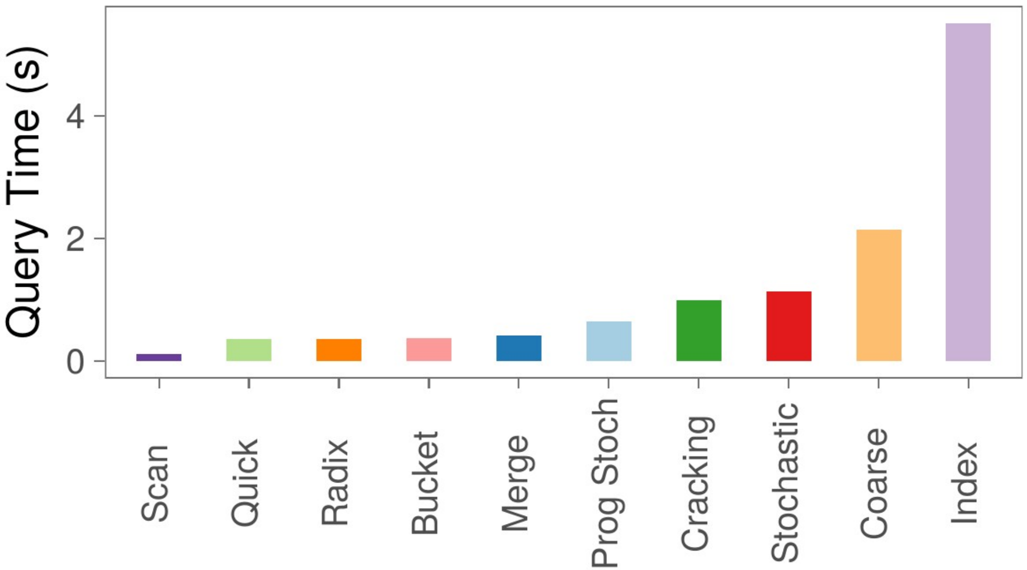
Varying δ: Entire Workload Cost



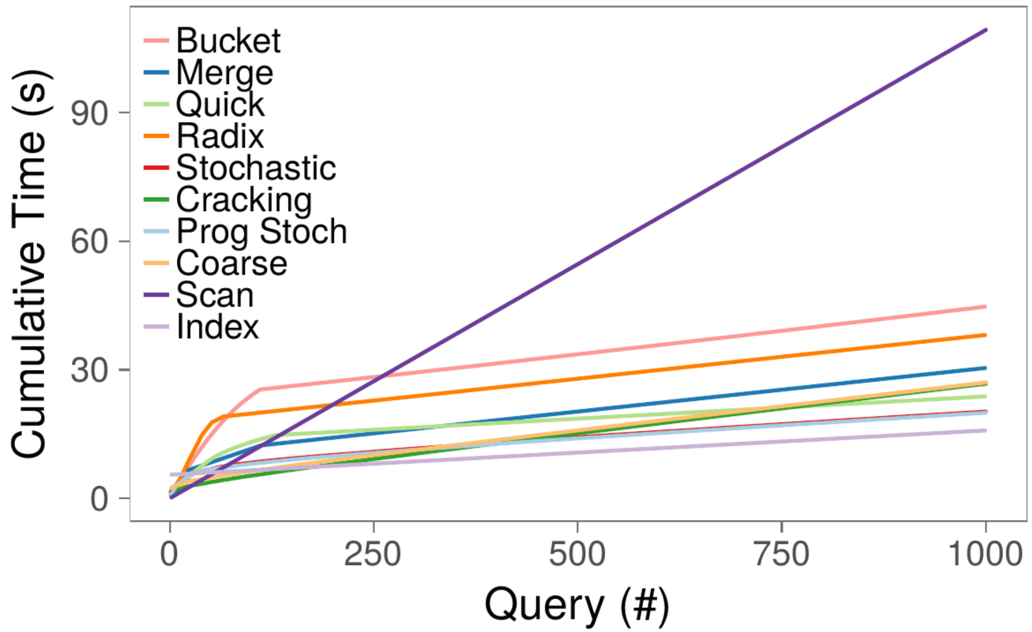
Chosen δ: 1^{st} Query ~= 2x Scan

Indexing Method	δ
Bucketsort	0.009
Mergesort	0.05
Quicksort	0.22
Radixsort	0.08

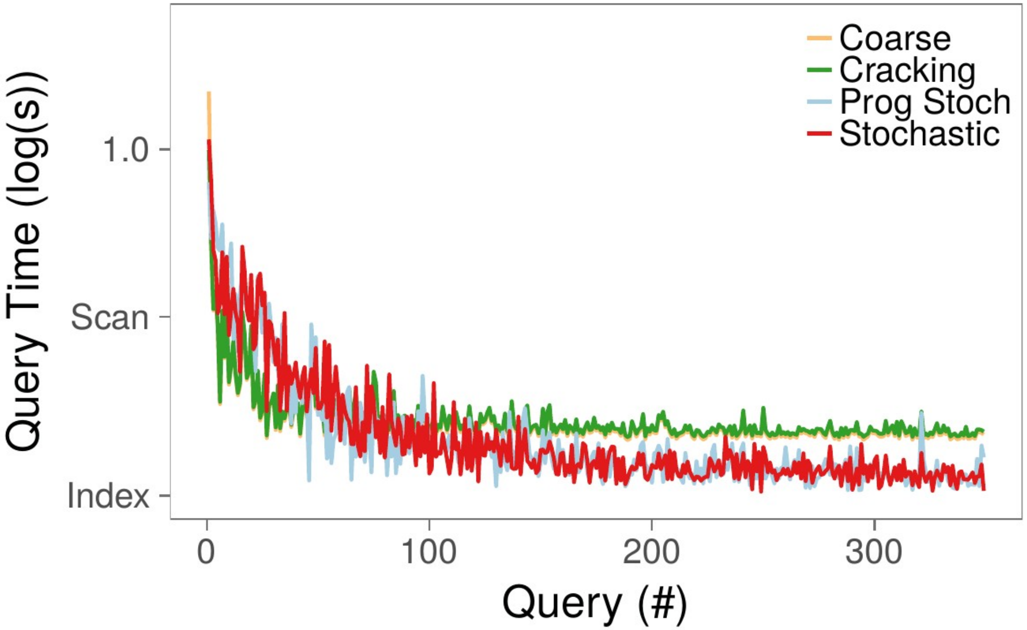
Comparison: 1^{st} Query



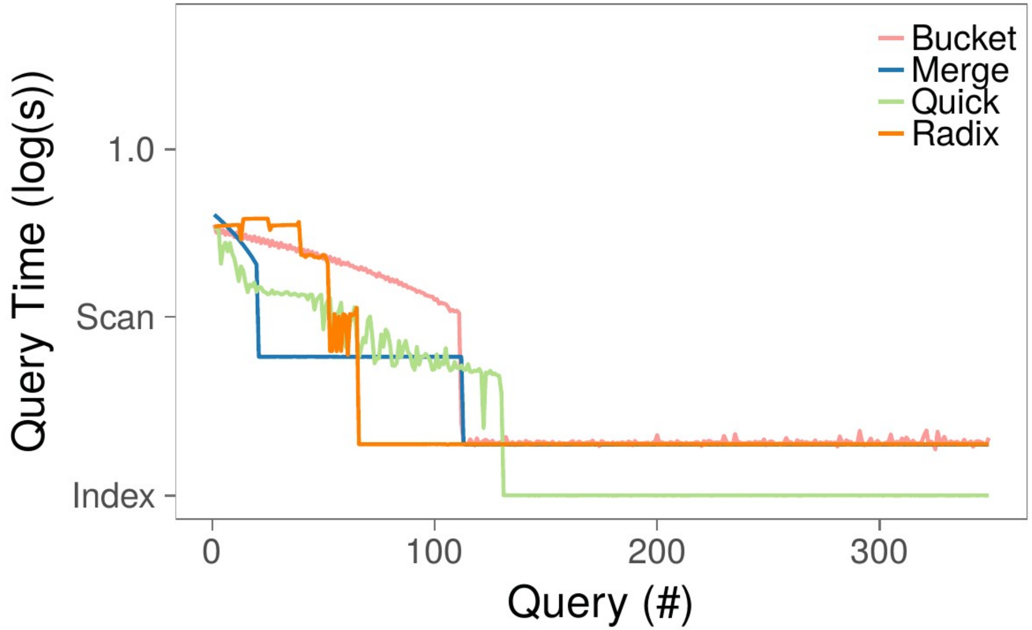
Comparison: Entire Workload



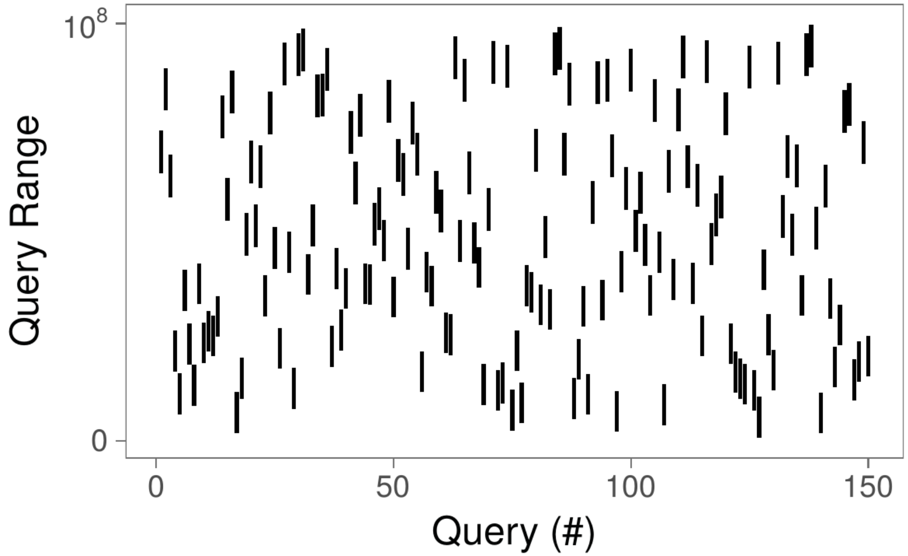
Comparison: Adaptive Indexing



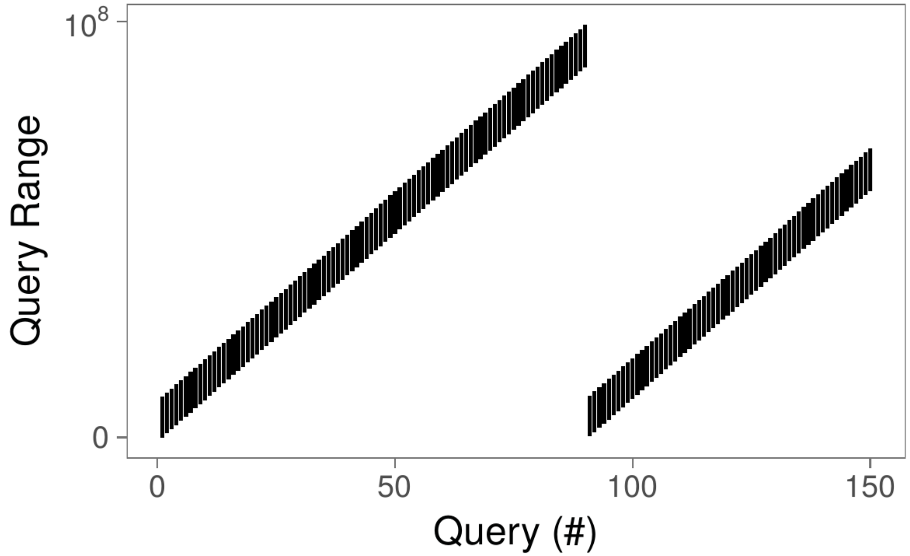
Comparison: Progressive Indexing



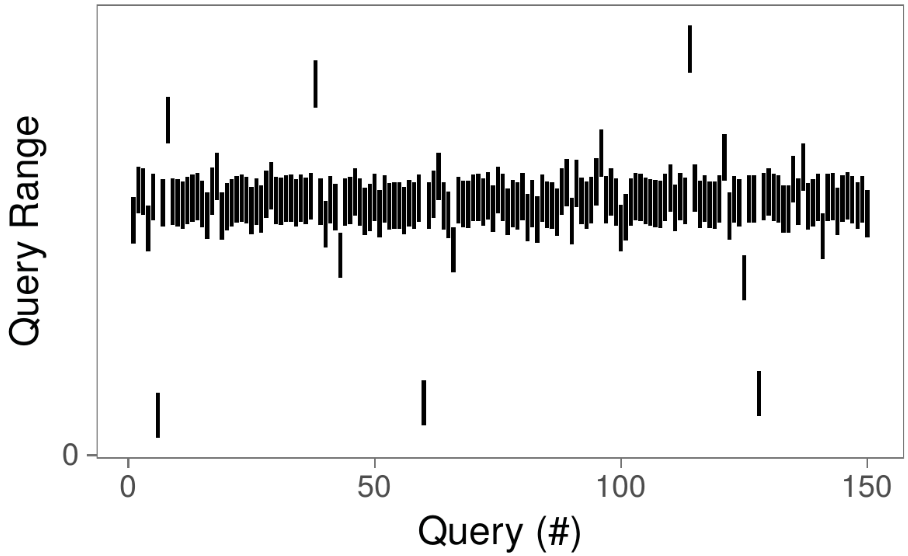
Random Workload



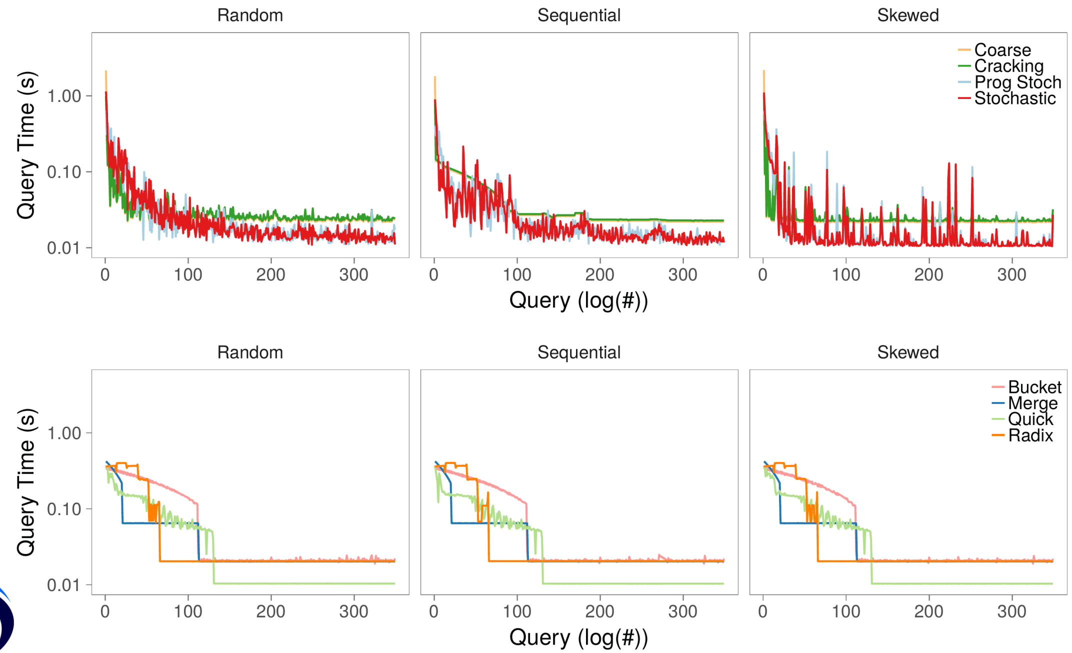
Sequential Workload



Skewed Workload



Different Workloads



Queries until Pay-off

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

Progressive Indexing

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