

# 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](mailto:s.manegold@liacs.leidenuniv.nl)

[e.m.bakker@liacs.leidenuniv.nl](mailto:e.m.bakker@liacs.leidenuniv.nl)

# Physical design problem

Database systems perform efficiently only after proper tuning...



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

**DBA without adaptive indexing**

# Physical Design

**Sample  
Workload**



Timeline

# Physical Design

**Sample  
Workload**

**Analyze  
Performance**





# Physical Design

**Sample  
Workload**

**Analyze  
Performance**

**Prepare Estimated  
physical design**



Timeline

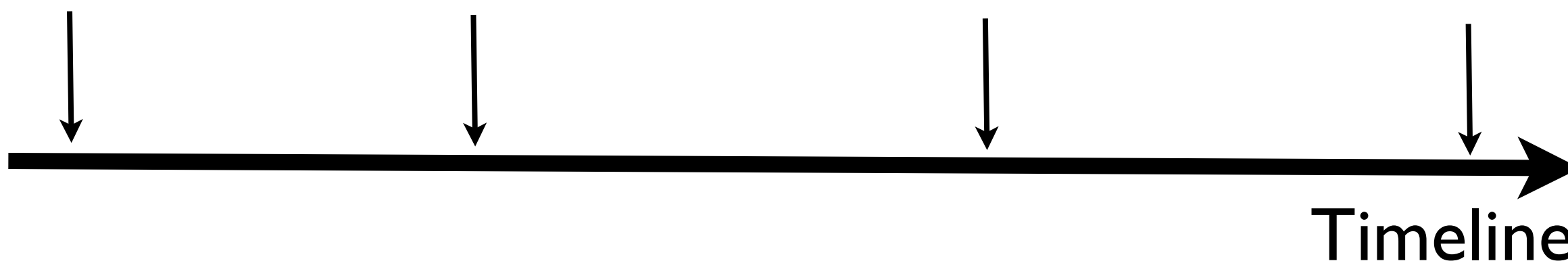
# Physical Design

**Sample  
Workload**

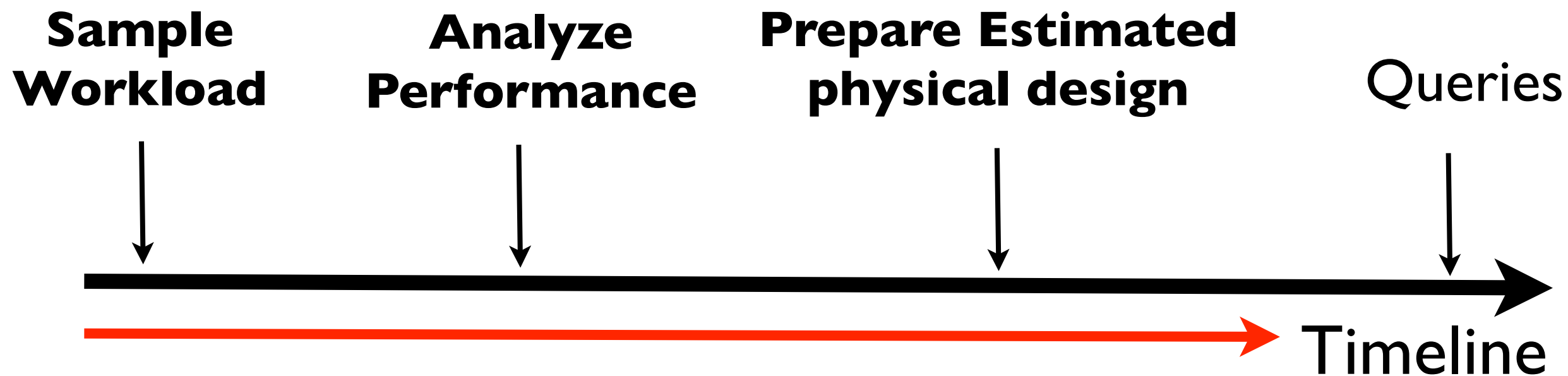
**Analyze  
Performance**

**Prepare Estimated  
physical design**

Queries

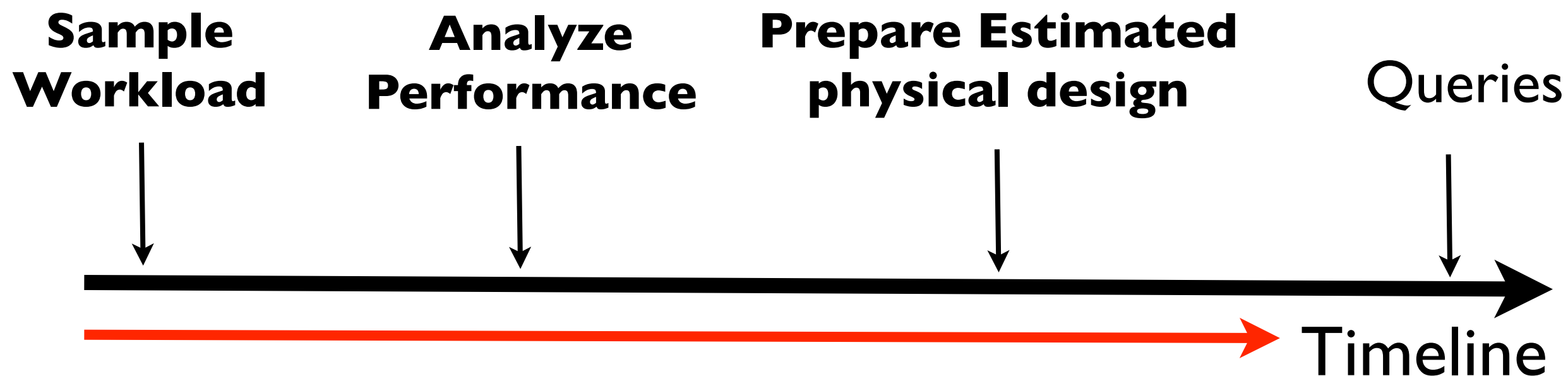


# Physical Design



**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



*How?*

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

***DBA with adaptive indexing***

# Adaptive Indexing

*no monitoring*

*no preparation*

*no external tools*

*no full indexes*

*no human involvement*

# Adaptive Indexing

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

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

# Adaptive Indexing

*no monitoring*

*no preparation*

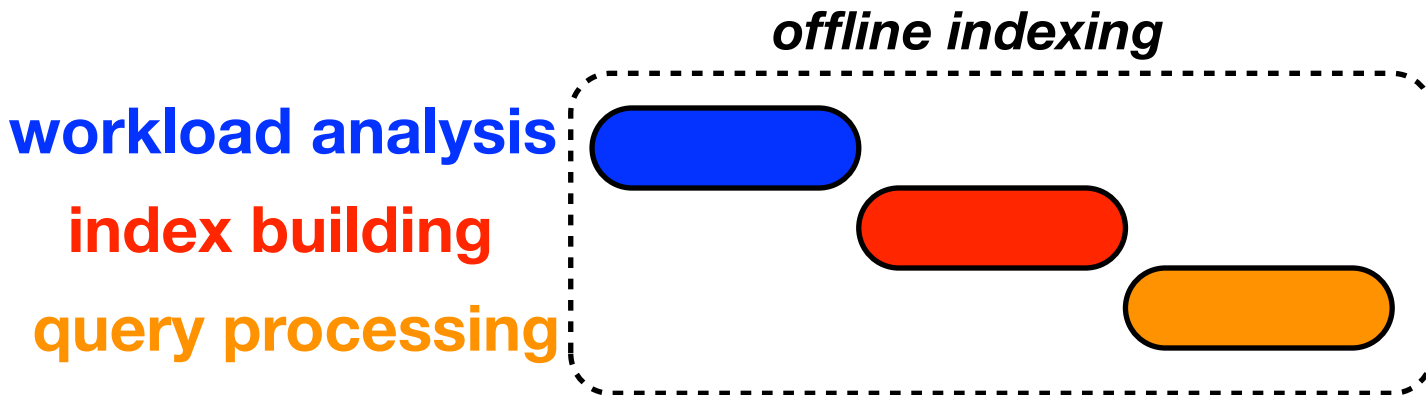
*no external tools*

*no full indexes*

*no human involvement*

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

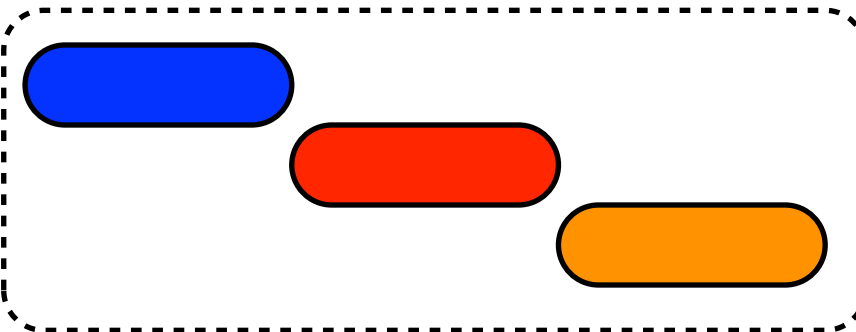
# Indexing Overview



# Indexing Overview

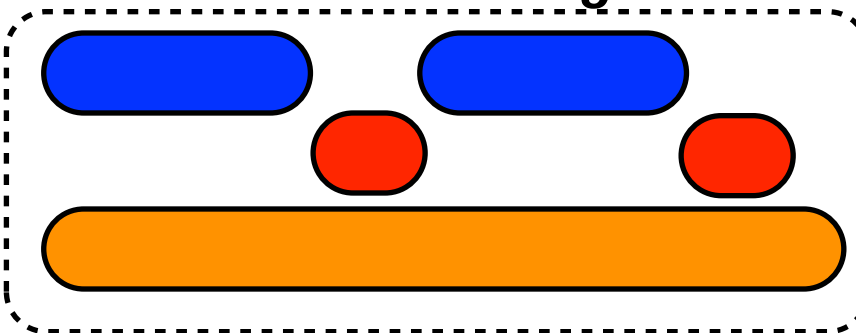
*offline indexing*

**workload analysis**  
**index building**  
**query processing**



*online indexing*

**workload analysis**  
**index building**  
**query processing**

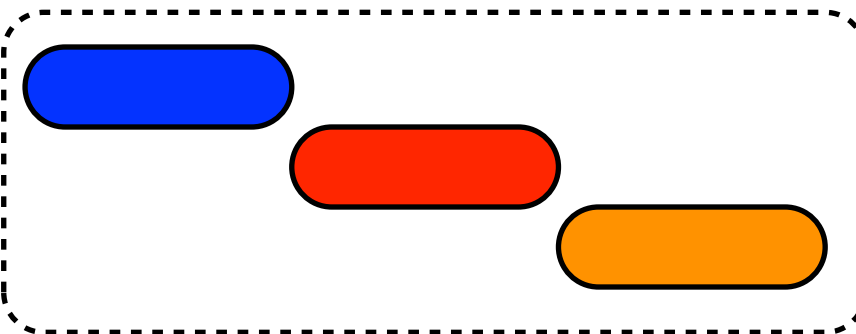




# Indexing Overview

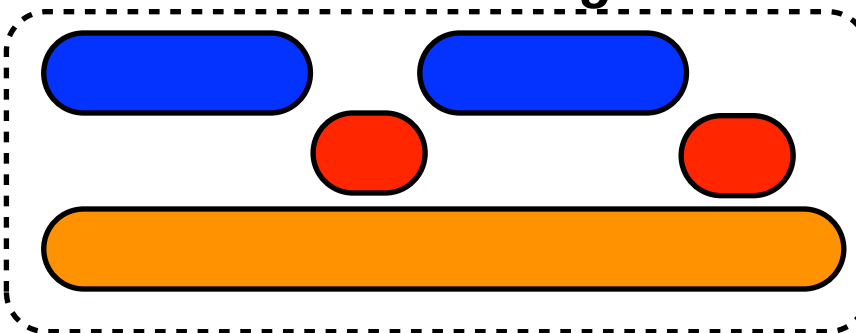
*offline indexing*

**workload analysis**  
**index building**  
**query processing**



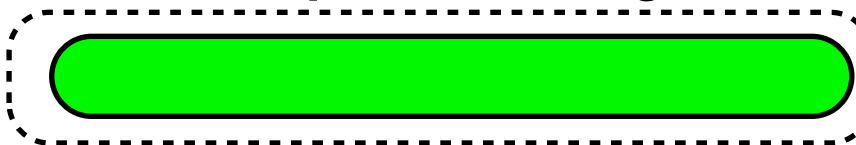
*online indexing*

**workload analysis**  
**index building**  
**query processing**

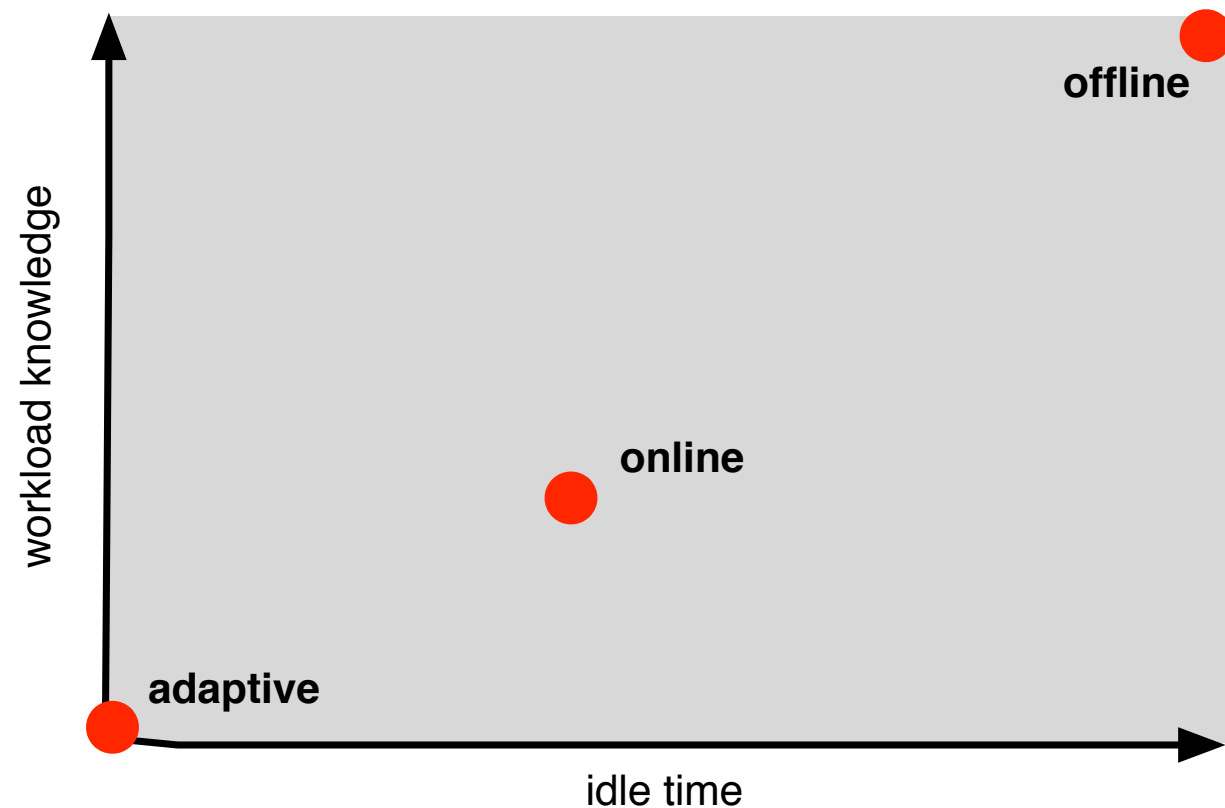
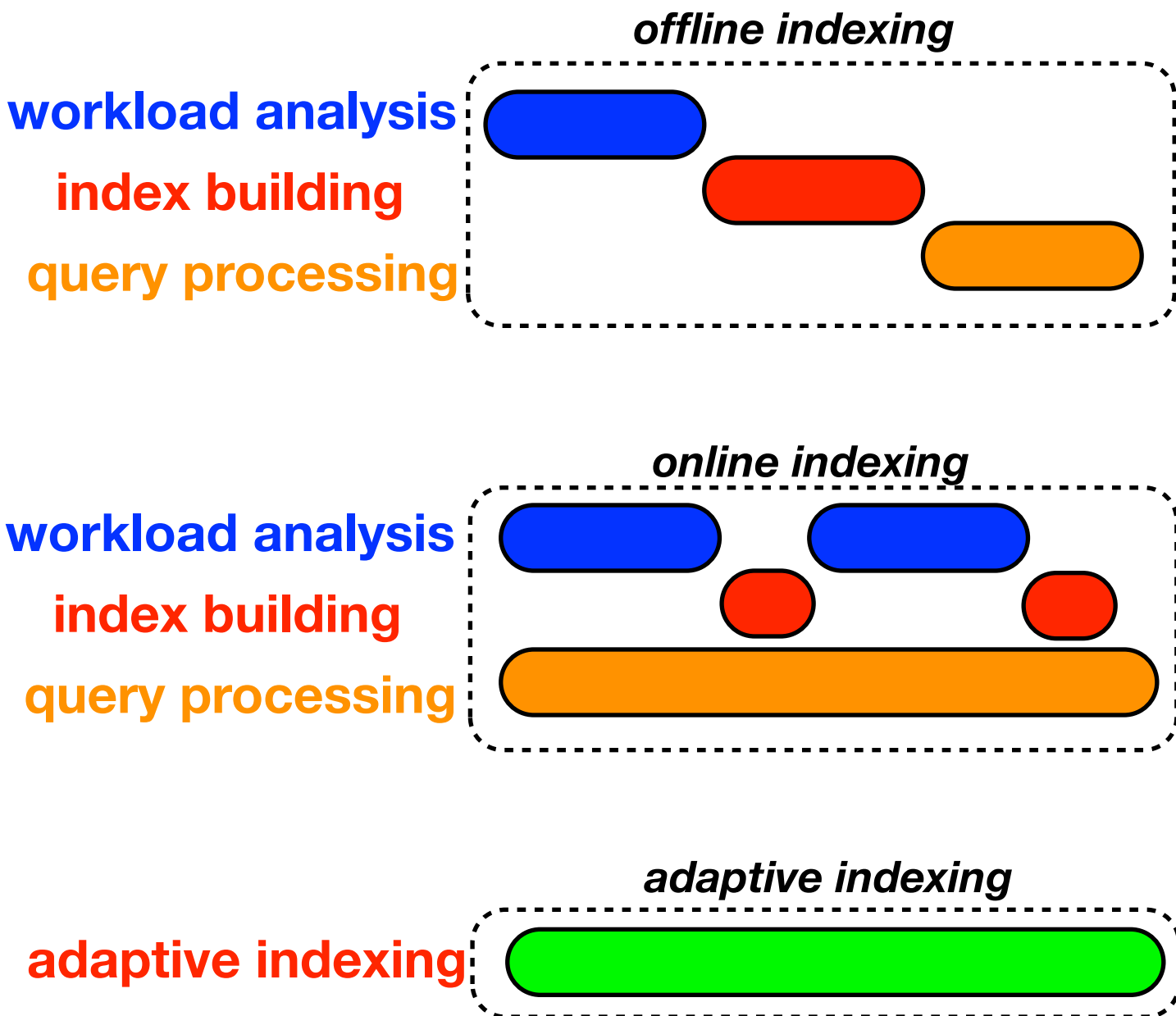


*adaptive indexing*

**adaptive indexing**



# Indexing Overview



# Cracking the Database Store

Martin Kersten

Stefan Manegold

CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

{Martin.Kersten,Stefan.Manegold}@cwi.nl

## Abstract

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

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

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

## 1 Introduction

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

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

database design centered around index accelerators for individual tables and join-indices to speed up exploration of semantic meaningful links.

In decision support applications and scientific databases, however, it is a priori less evident what subsets are relevant for answering the -mostly statistical- queries. Queries tend to be ad-hoc and temporarily localized against a small portion of the databases. Data warehouse techniques, such as star- and snowflake schemas and 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 [ZLLL01, ACK<sup>+</sup>04]. Between successive database reorganizations, a query is optimized against this static navigational access structure.

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

In this paper we explore a different route based on the hypothesis that access maintenance should be a byproduct of query processing, not of updates. A query is interpreted as both a request for a particular database subset and as an advice to *crack* the database store into smaller *pieces* augmented with an index to access them. If it is unavoidable to touch *Una*-interesting tuples during query evaluation, can we use that to prepare for a better future?

To illustrate, consider a simple query `select * from R where R.a <10` and a storage scheme that requires a full table scan, i.e. touching all tuples to select those of interest. The result produced in most systems is a stream of qualifying tuples. However, it can also be interpreted as a task to fragment the table into two pieces, i.e. apply horizontal fragmentation. This operation does not come for free, because the new table incarnation should be written back to persistent store and its properties stored in the catalog. For example, the original table can be replaced by a UNION TA-

# Database Cracking

Stratos 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 organizing data the way users request it, we can achieve fast access and the much desired self-organized behavior.

We present the first mature cracking architecture and report on our implementation of cracking in the context of a full fledged relational system. It led to a minor enhancement to its relational algebra kernel, such that cracking could be piggy-backed without incurring too much processing overhead. Furthermore, we illustrate the ripple effect of dynamic reorganization on the query plans derived by the SQL optimizer. The experiences and results obtained are indicative of a significant reduction in system complexity. We show that the resulting system is able to self-organize based on incoming requests with clear performance benefits. This behavior is visible even when the user focus is randomly shifting to different parts of the data.

## 1. INTRODUCTION

Nowadays, the challenge for database architecture design is not in achieving ultra high performance but to design systems that are *simple* and *flexible*. A database system should be able to handle *huge* sets of data and *self-organize* according to the environment, e.g., the workload, available resources, etc. A nice discussion on such issues can be found in [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, 9, 14].

This article is published under a Creative Commons License Agreement (<http://creativecommons.org/licenses/by/2.5/>). You may copy, distribute, display, and 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.

Permission to copy without fee 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.

# 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

Column A

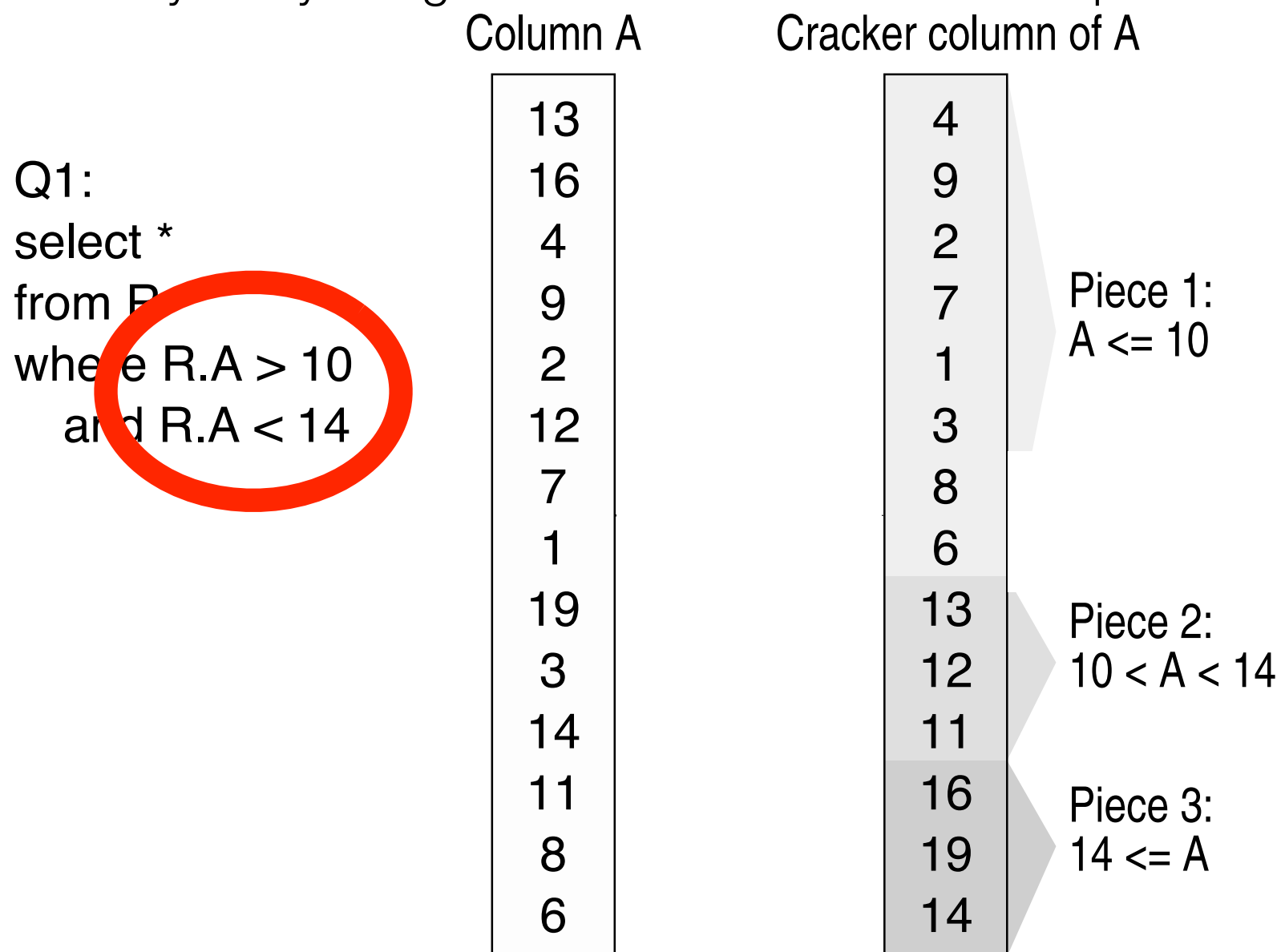
Q1:  
select \*  
from R  
where R.A > 10  
and R.A < 14

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

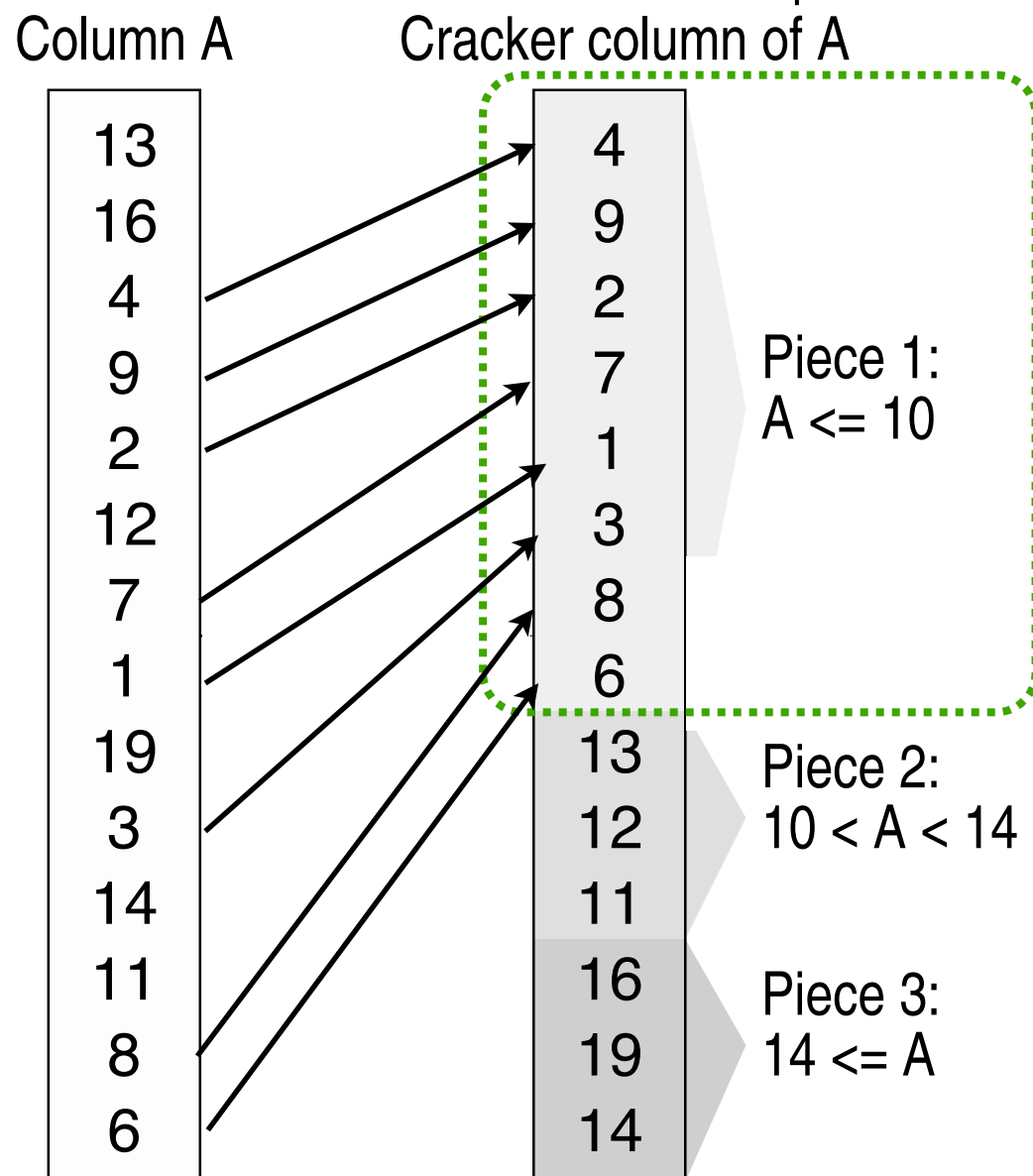


# Cracking Example

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





# Cracking Example

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

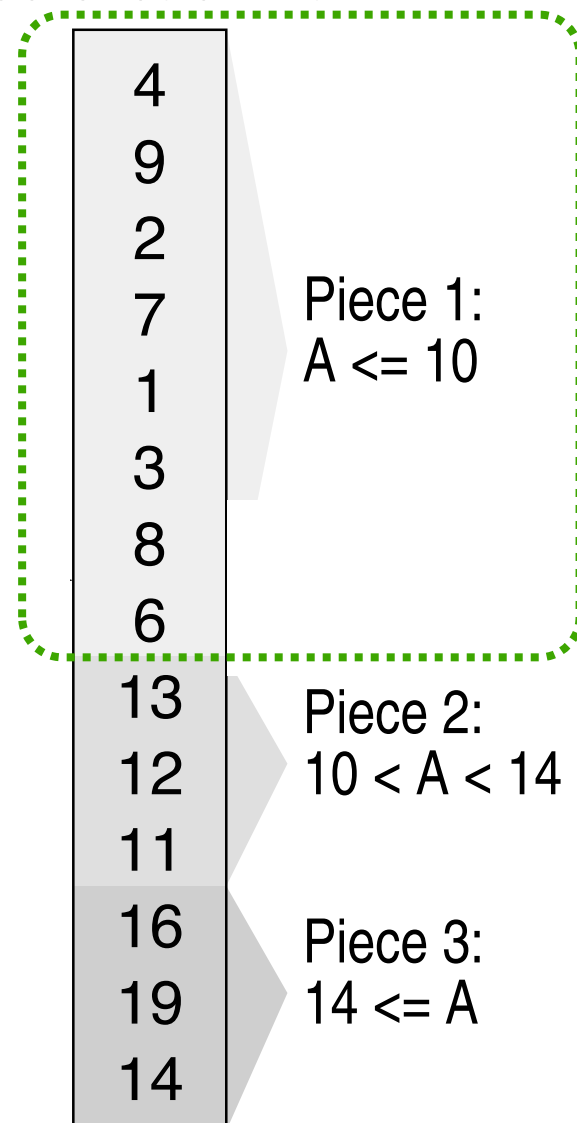
Physically reorganize based on the selection predicate

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

Cracker column of A



# Cracking Example

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

Physically reorganize based on the selection predicate

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

Cracker column of A

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

Piece 1:  
 $A \leq 10$

Piece 2:  
 $10 < A < 14$

Piece 3:  
 $14 \leq A$

# Cracking Example

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

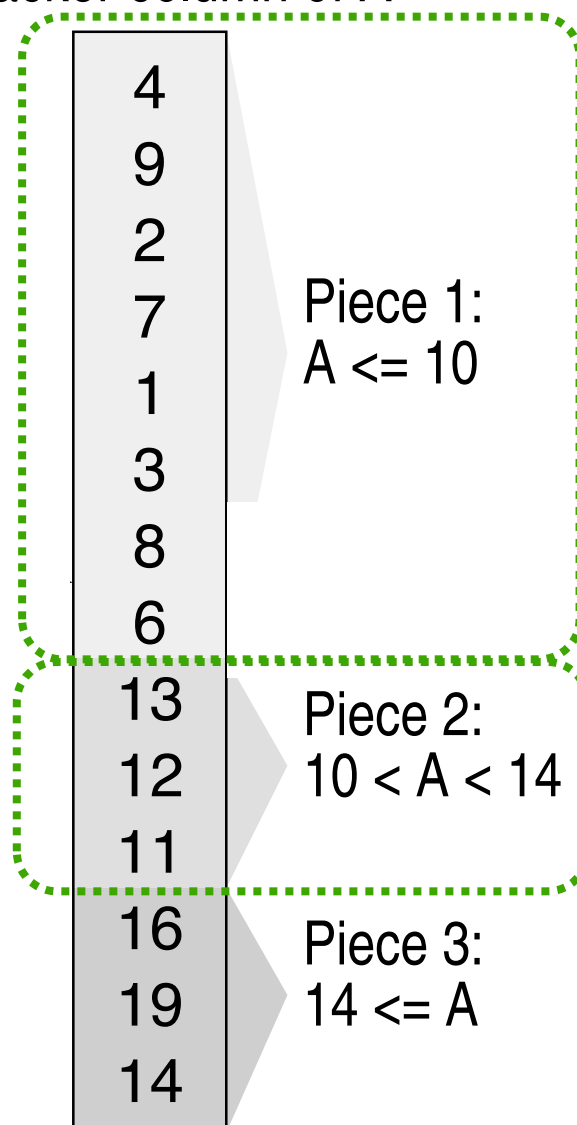
Physically reorganize based on the selection predicate

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

Column A

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

Cracker column of A



# Cracking Example

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

Physically reorganize based on the selection predicate

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

Cracker column of A

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

Piece 1:  
 $A \leq 10$

Piece 2:  
 $10 < A < 14$

Piece 3:  
 $14 \leq A$

# Cracking Example

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

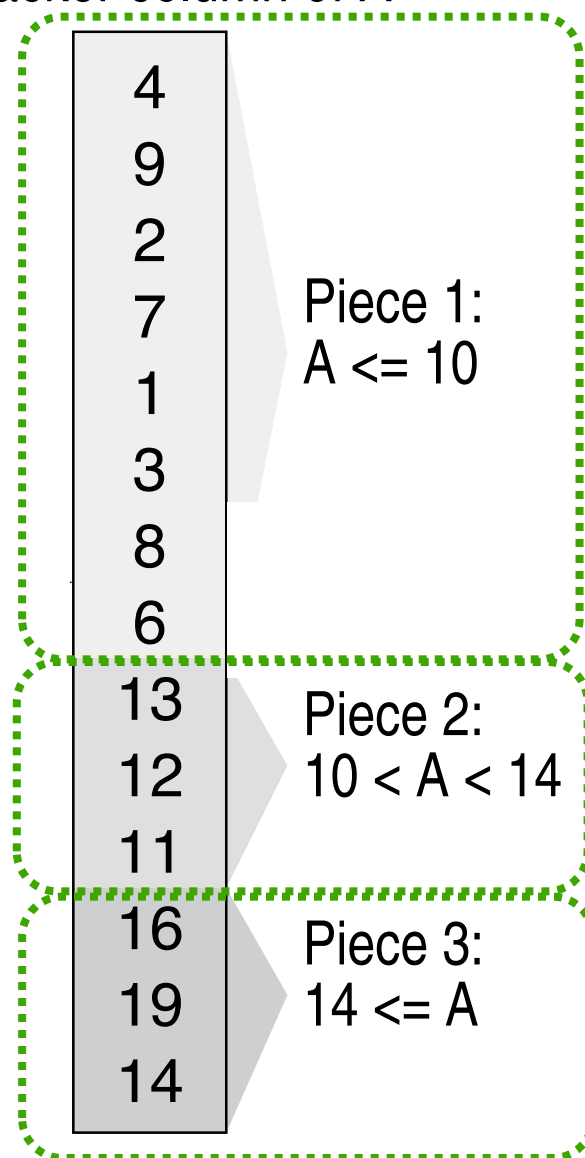
Physically reorganize based on the selection predicate

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

Column A

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

Cracker column of A



# Cracking Example

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

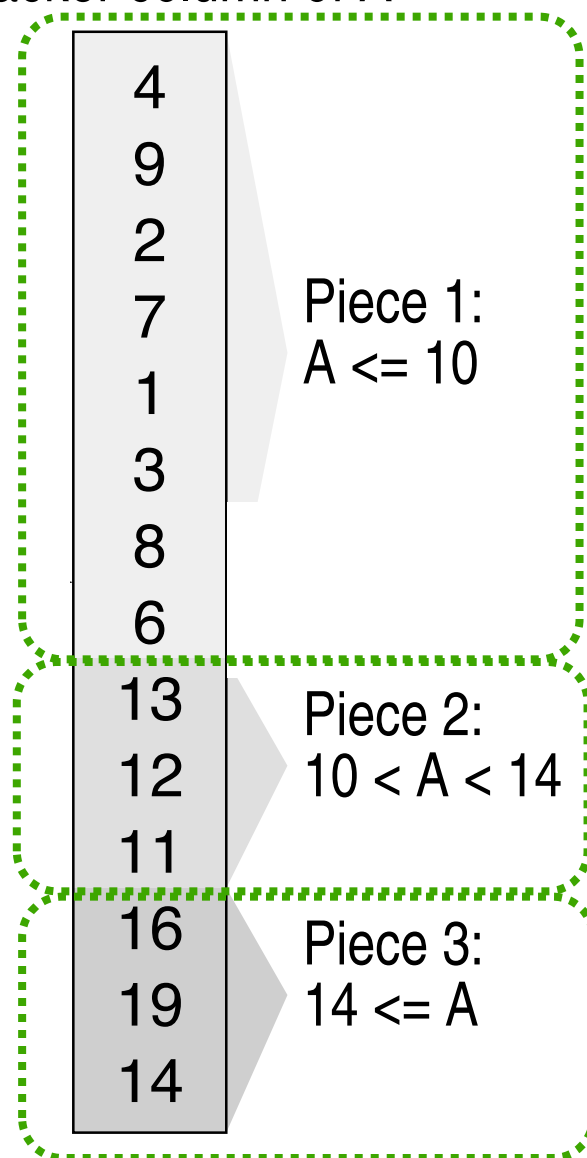
Physically reorganize based on the selection predicate

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

Cracker column of A



Piece 1:  
 $A \leq 10$

Piece 2:  
 $10 < A < 14$

Piece 3:  
 $14 \leq A$

Result tuples

# Cracking Example

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

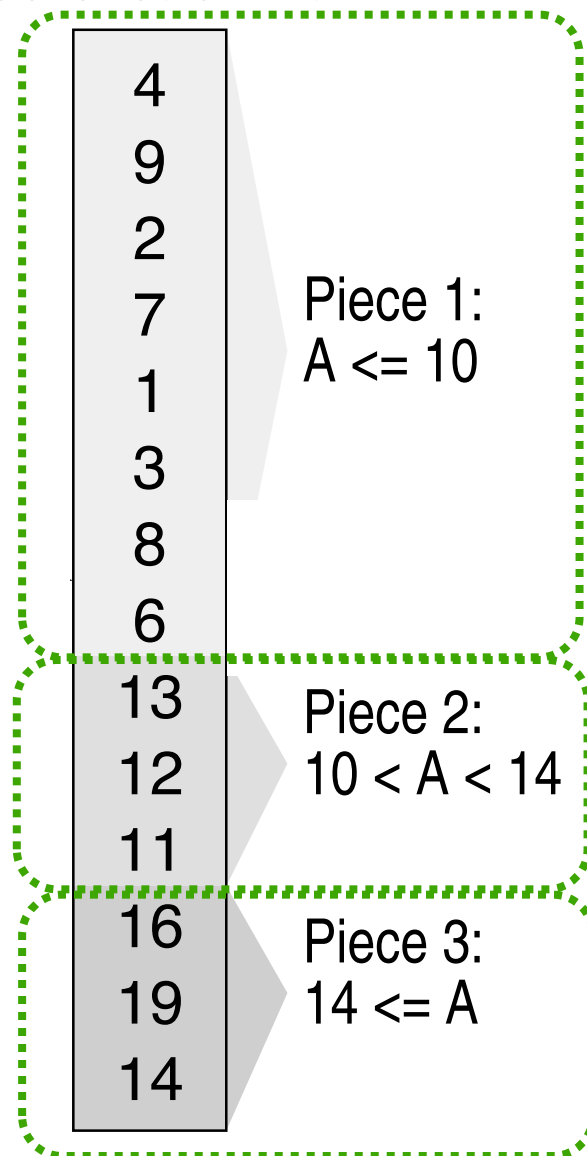
Physically reorganize based on the selection predicate

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

Column A

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

Cracker column of A



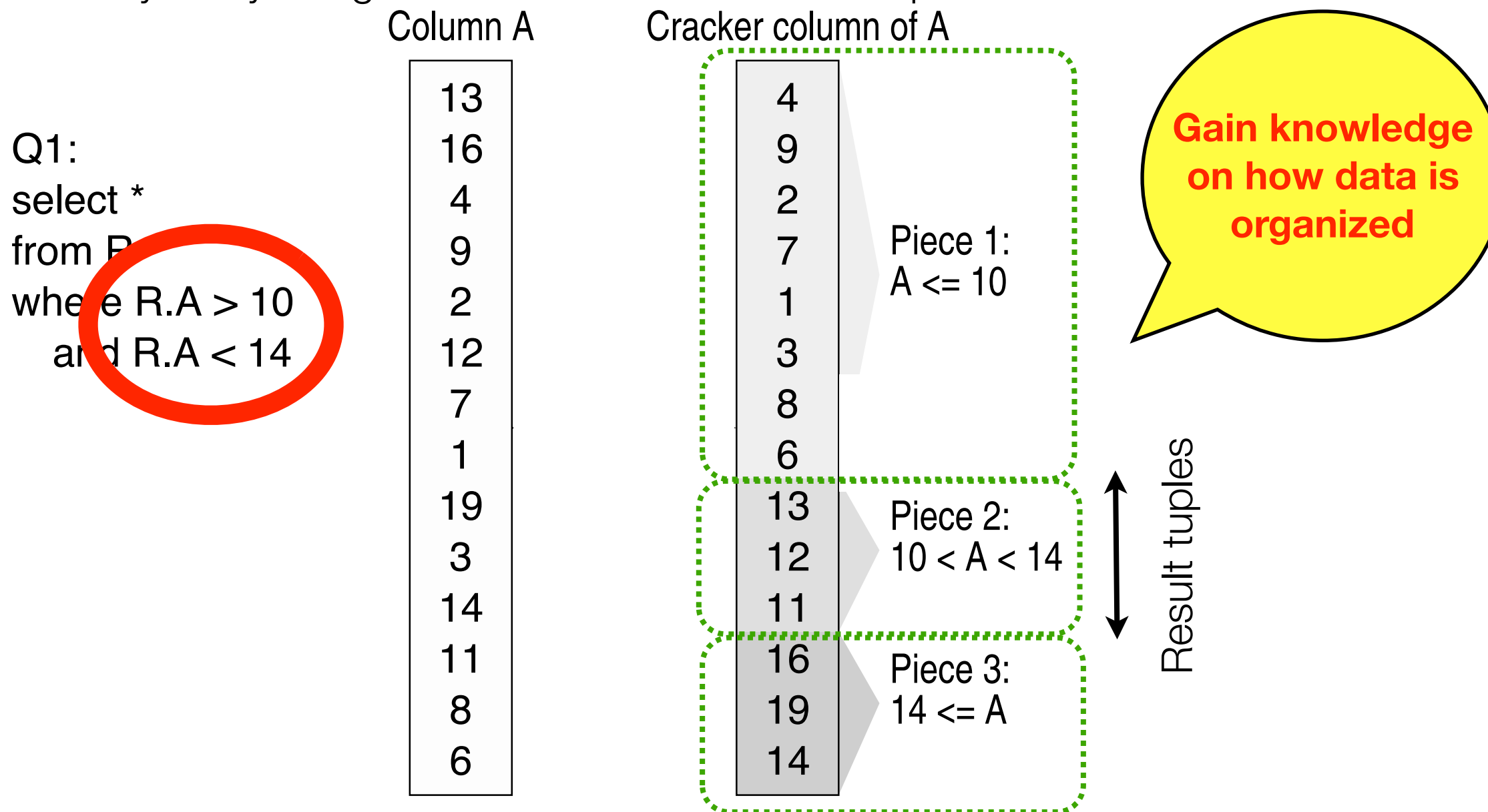
Gain knowledge  
on how data is  
organized

Result tuples

# Cracking Example

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

Physically reorganize based on the selection predicate



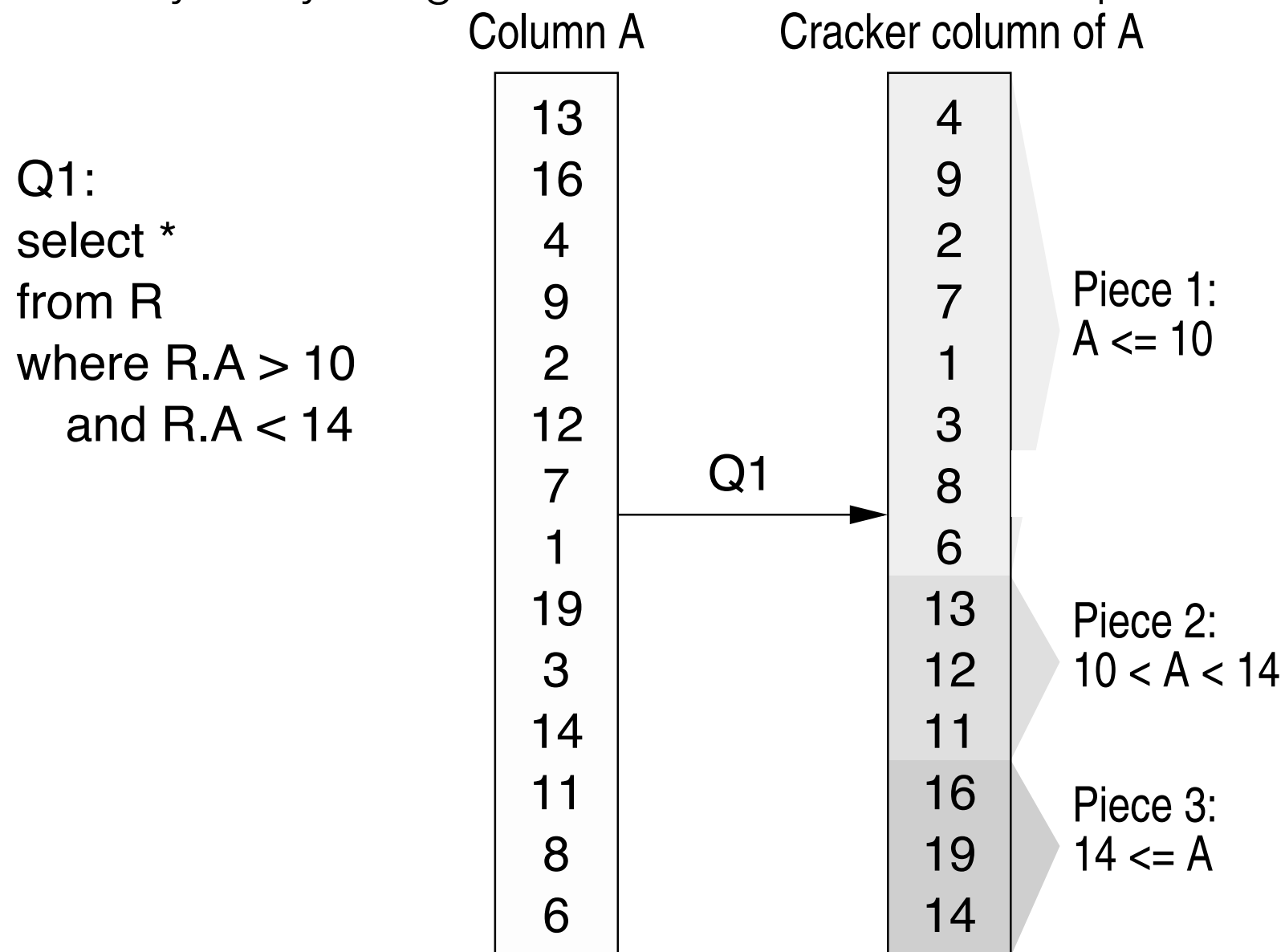
Dynamically/on-the-fly within the select-operator



# Cracking Example

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

Physically reorganize based on the selection predicate

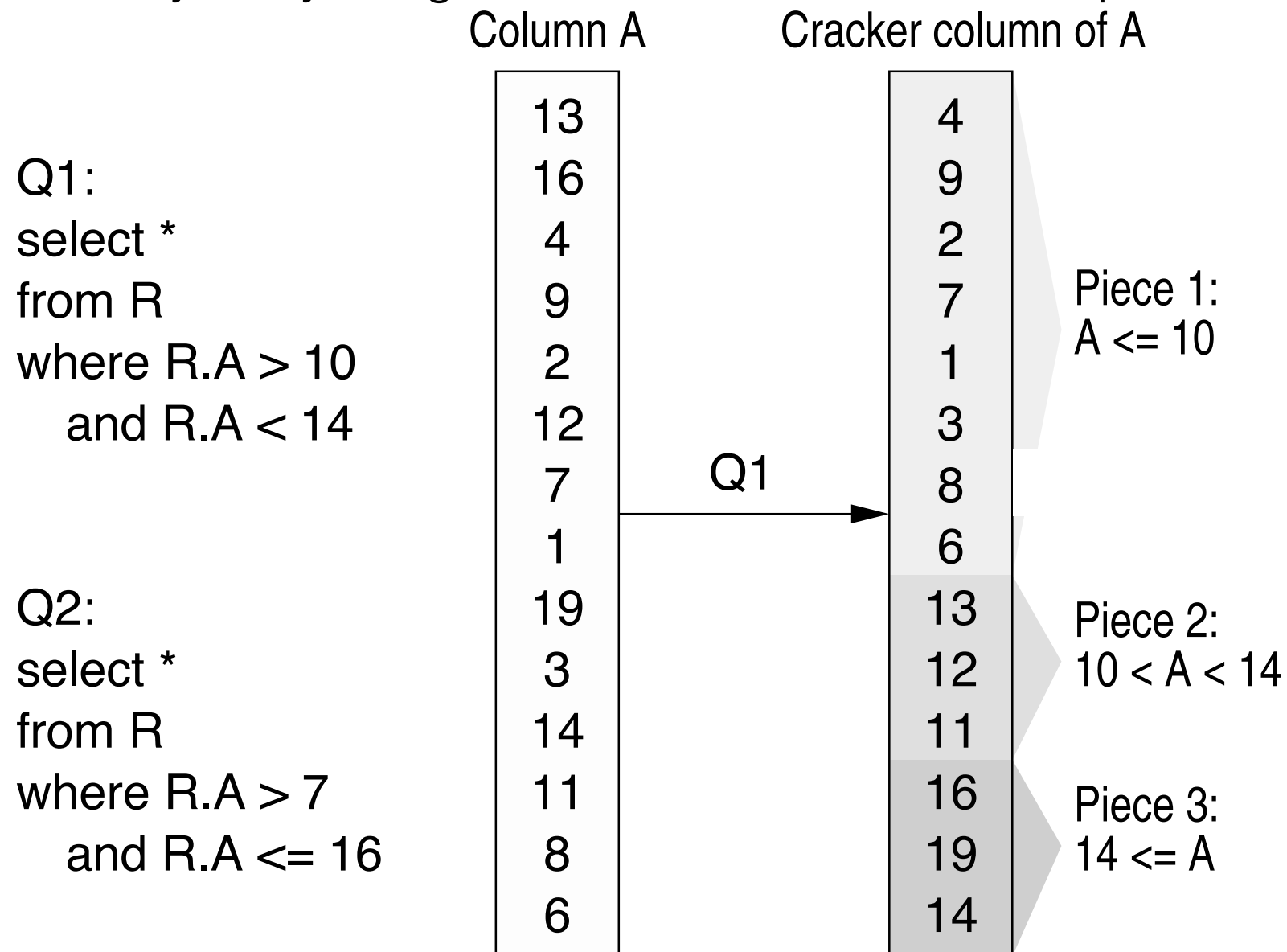


**Dynamically/on-the-fly within the select-operator**

# Cracking Example

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

Physically reorganize based on the selection predicate

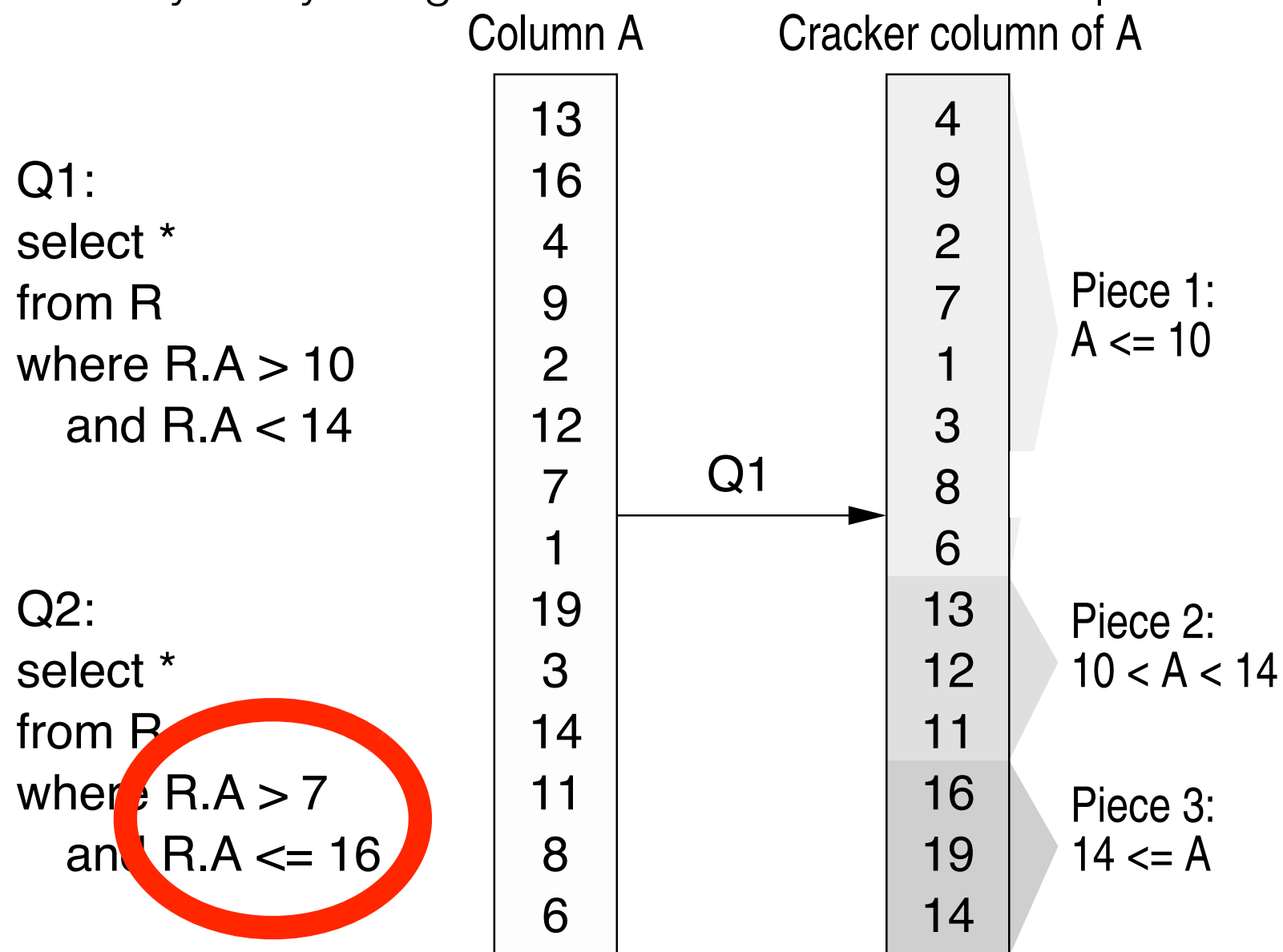


**Dynamically/on-the-fly within the select-operator**

# Cracking Example

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

Physically reorganize based on the selection predicate

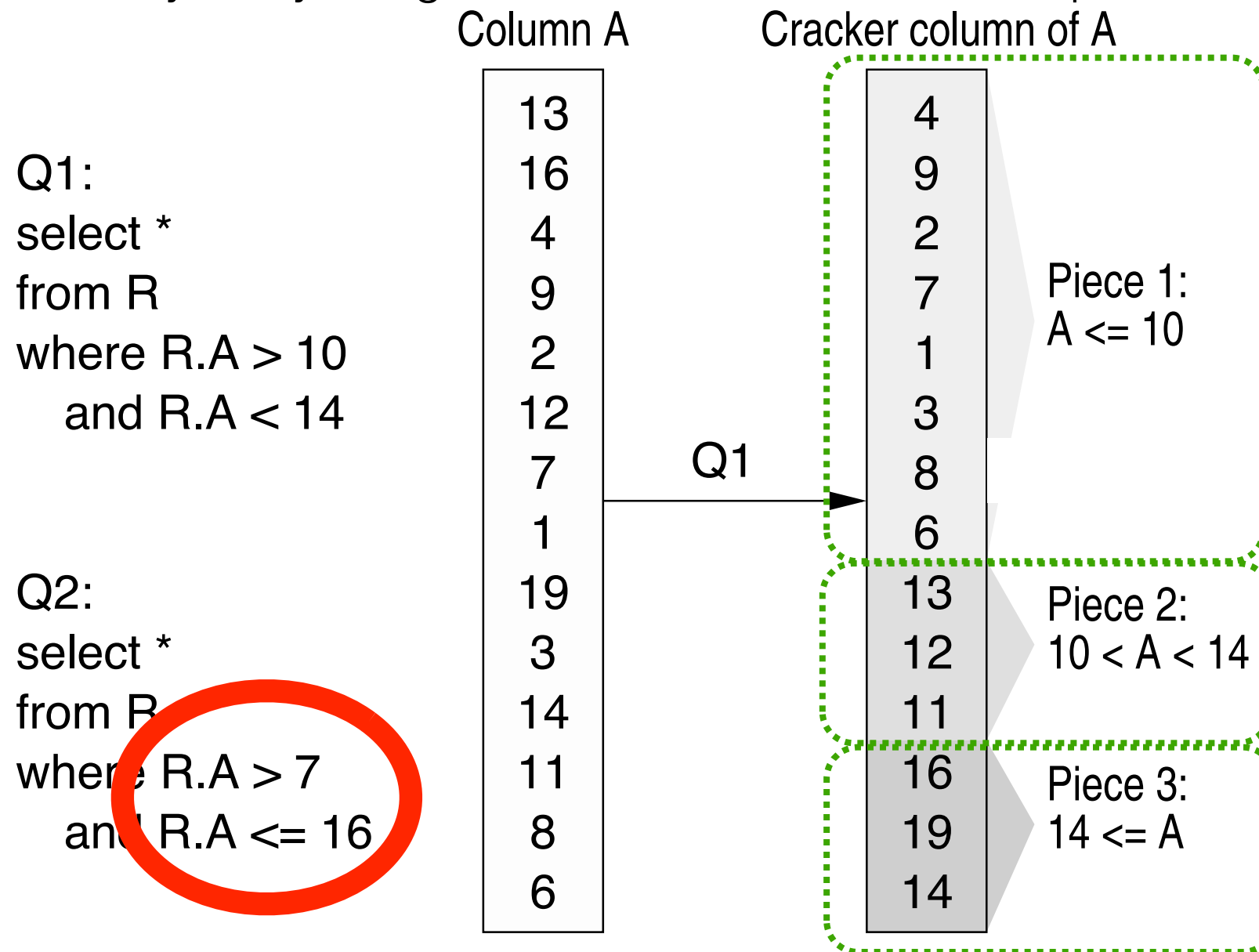


Dynamically/on-the-fly within the select-operator

# Cracking Example

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

Physically reorganize based on the selection predicate

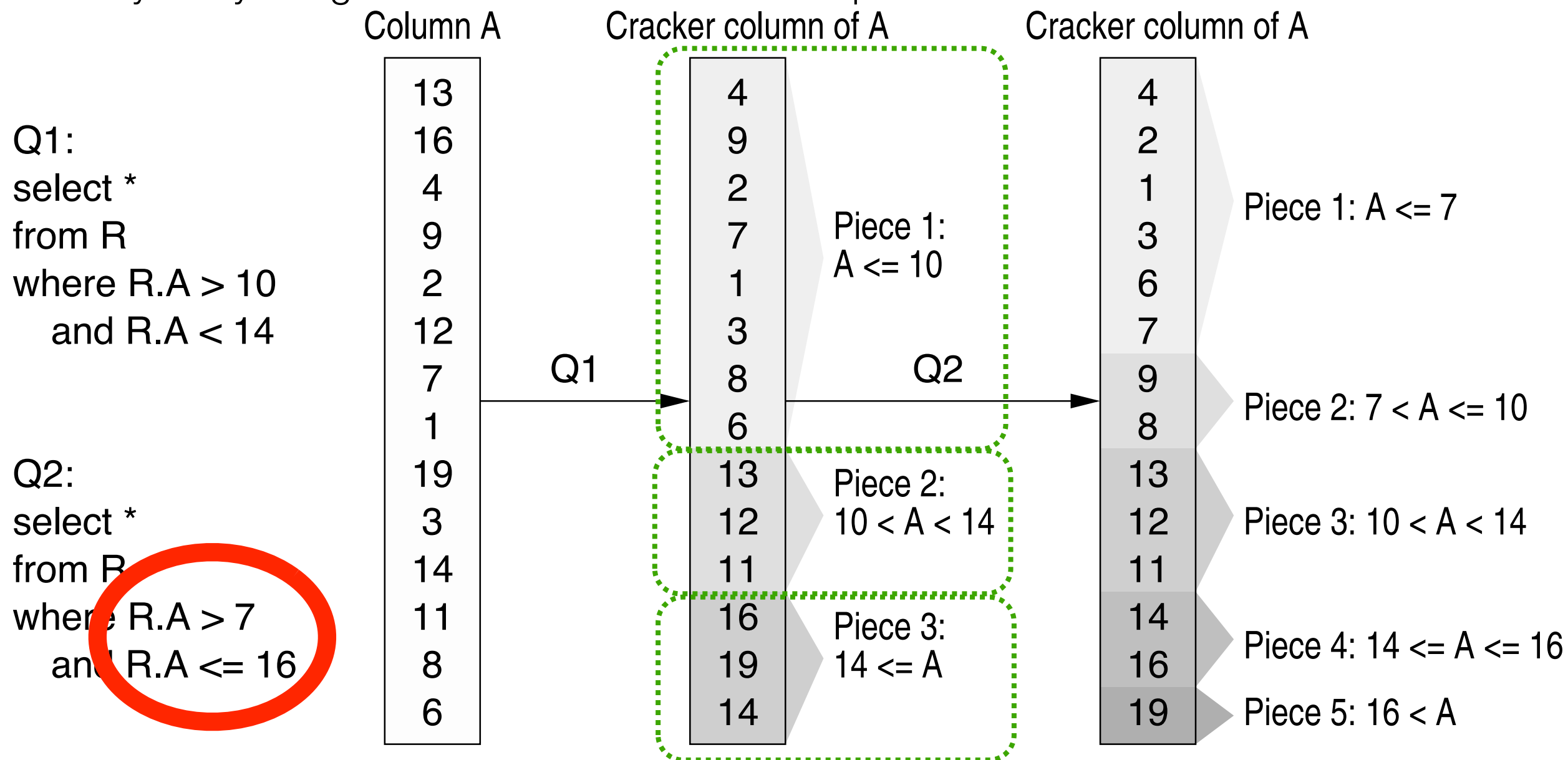


Dynamically/on-the-fly within the select-operator

# Cracking Example

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

Physically reorganize based on the selection predicate

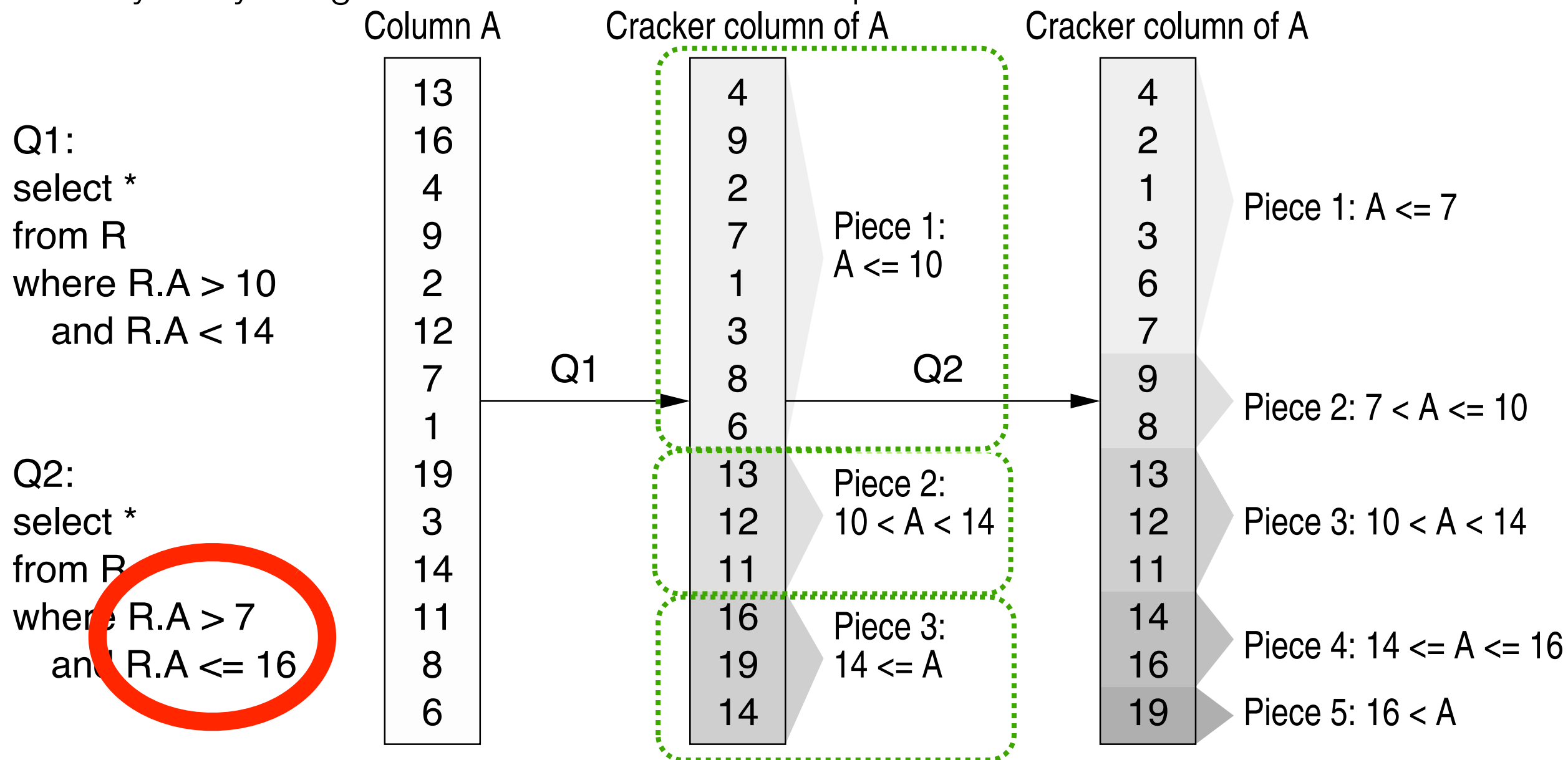


Dynamically/on-the-fly within the select-operator

# Cracking Example

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

Physically reorganize based on the selection predicate

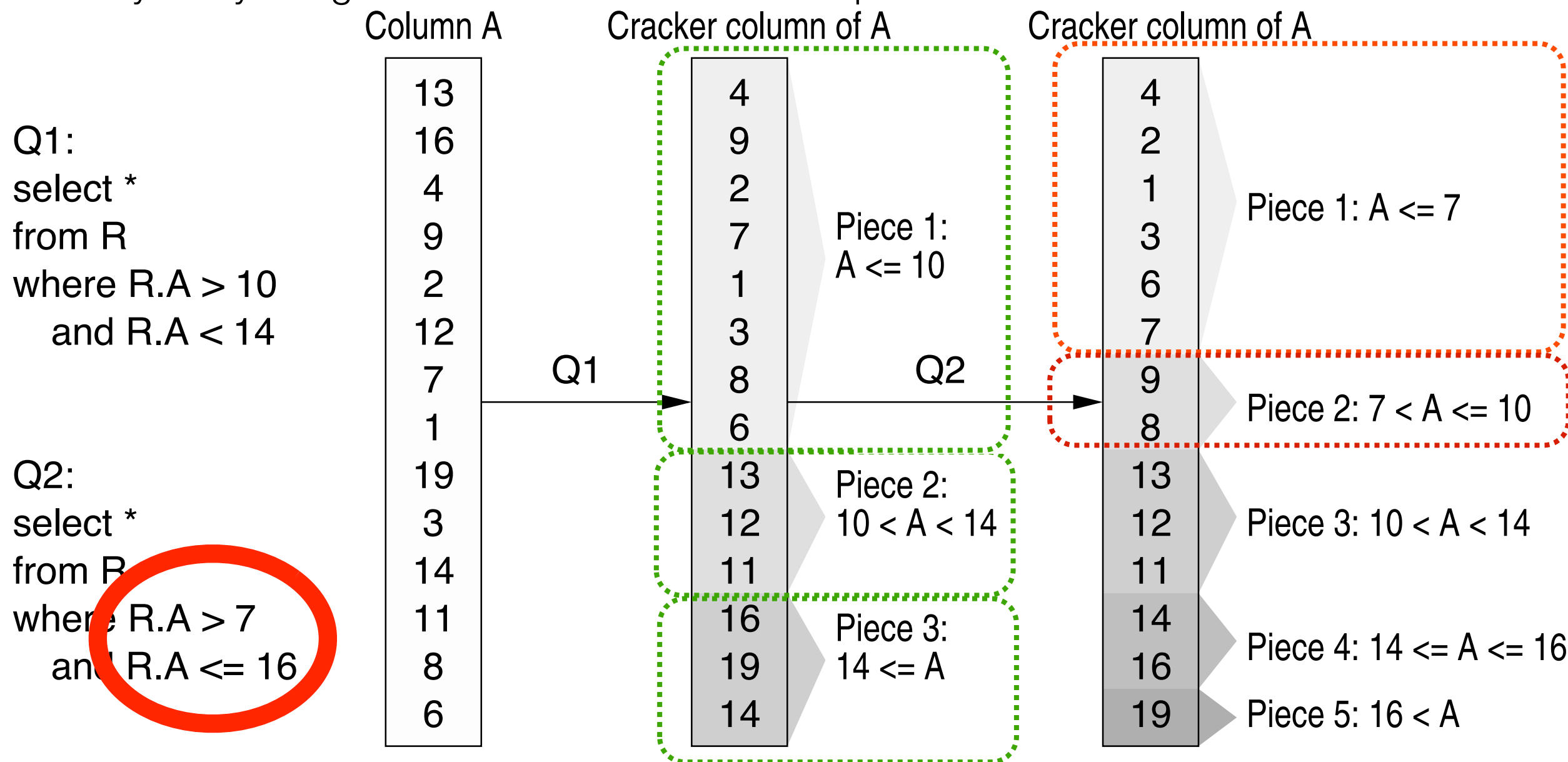


Dynamically/on-the-fly within the select-operator

# Cracking Example

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

Physically reorganize based on the selection predicate

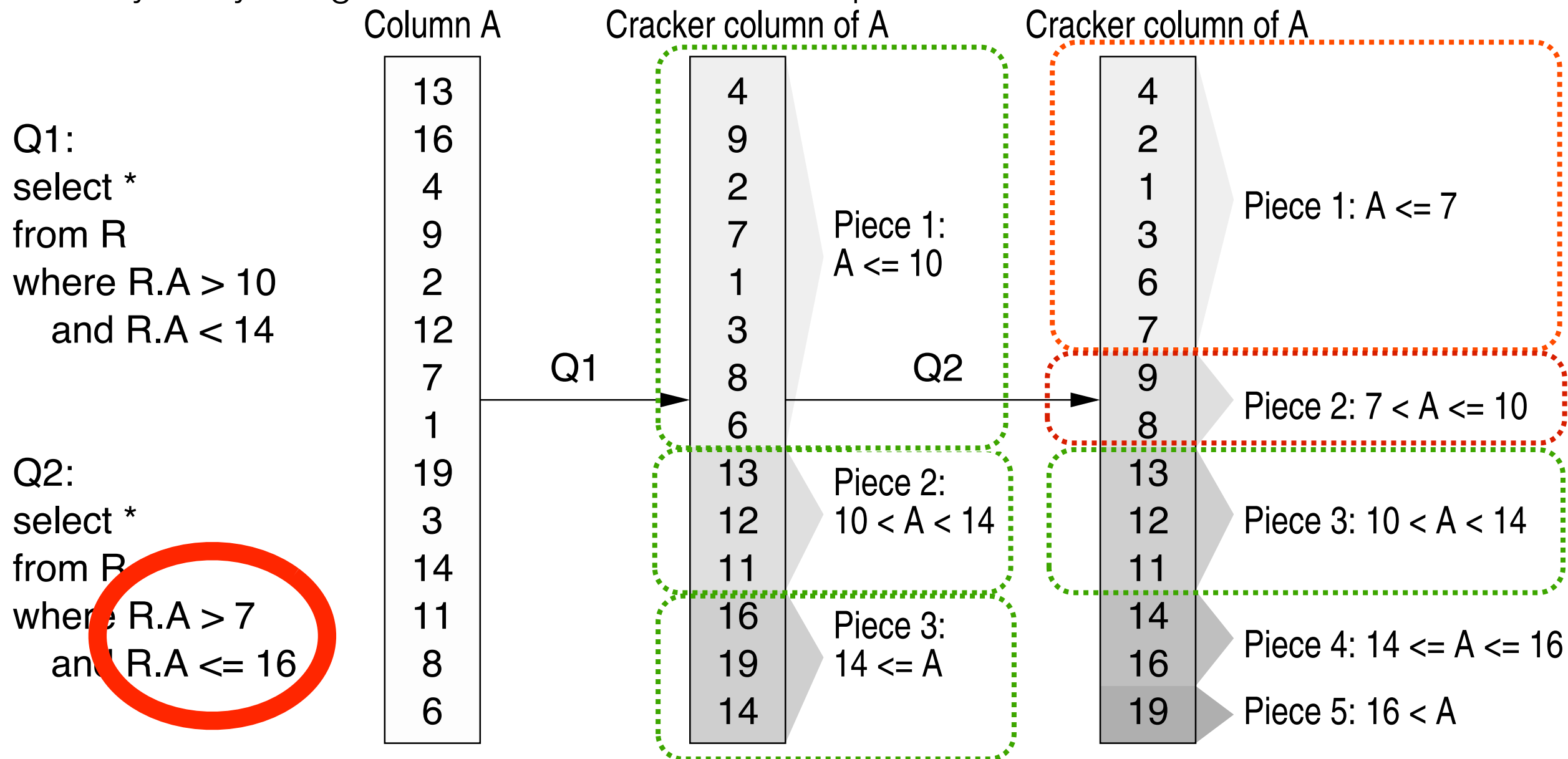


Dynamically/on-the-fly within the select-operator

# Cracking Example

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

Physically reorganize based on the selection predicate



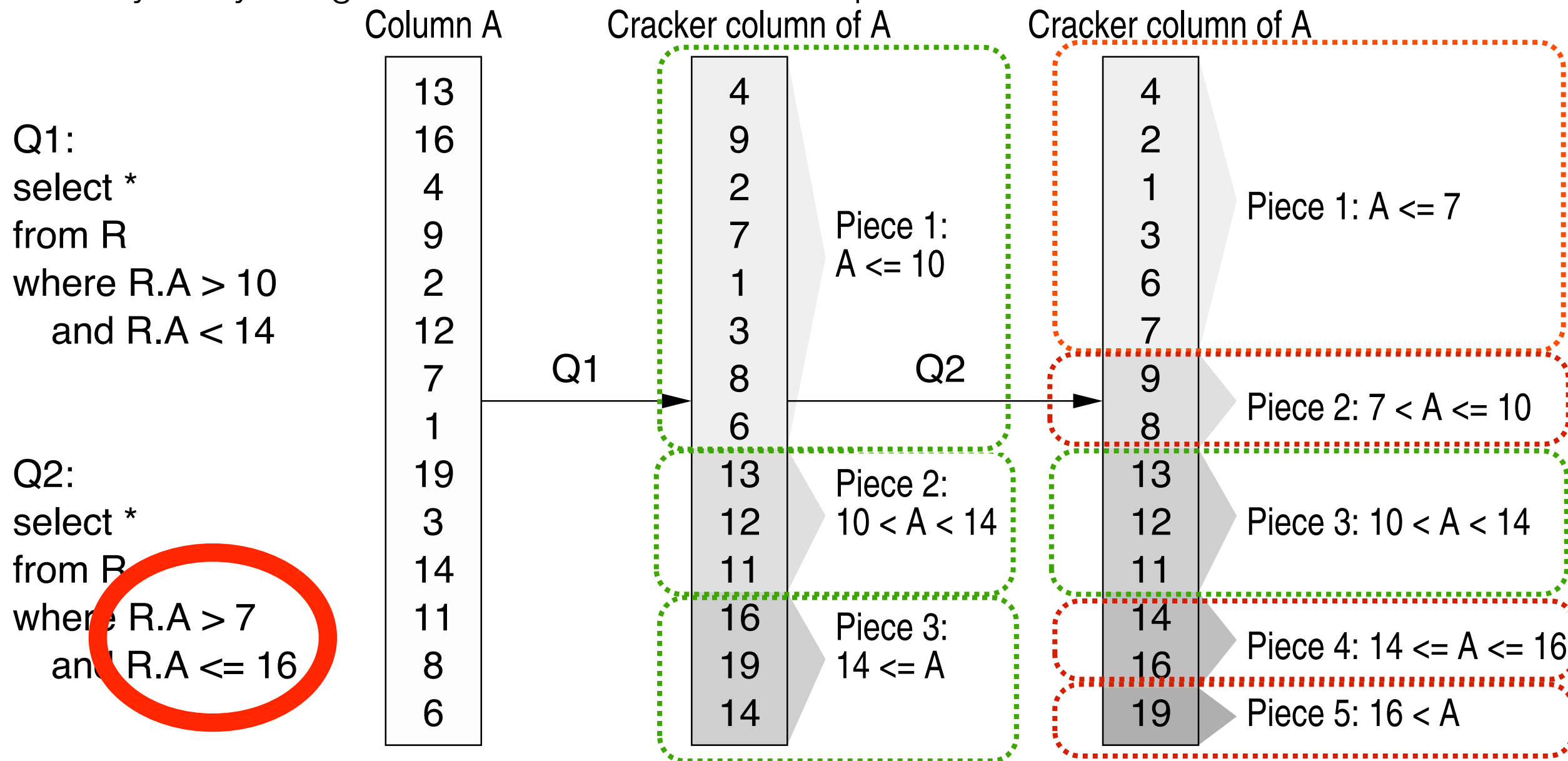
Dynamically/on-the-fly within the select-operator



# Cracking Example

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

Physically reorganize based on the selection predicate

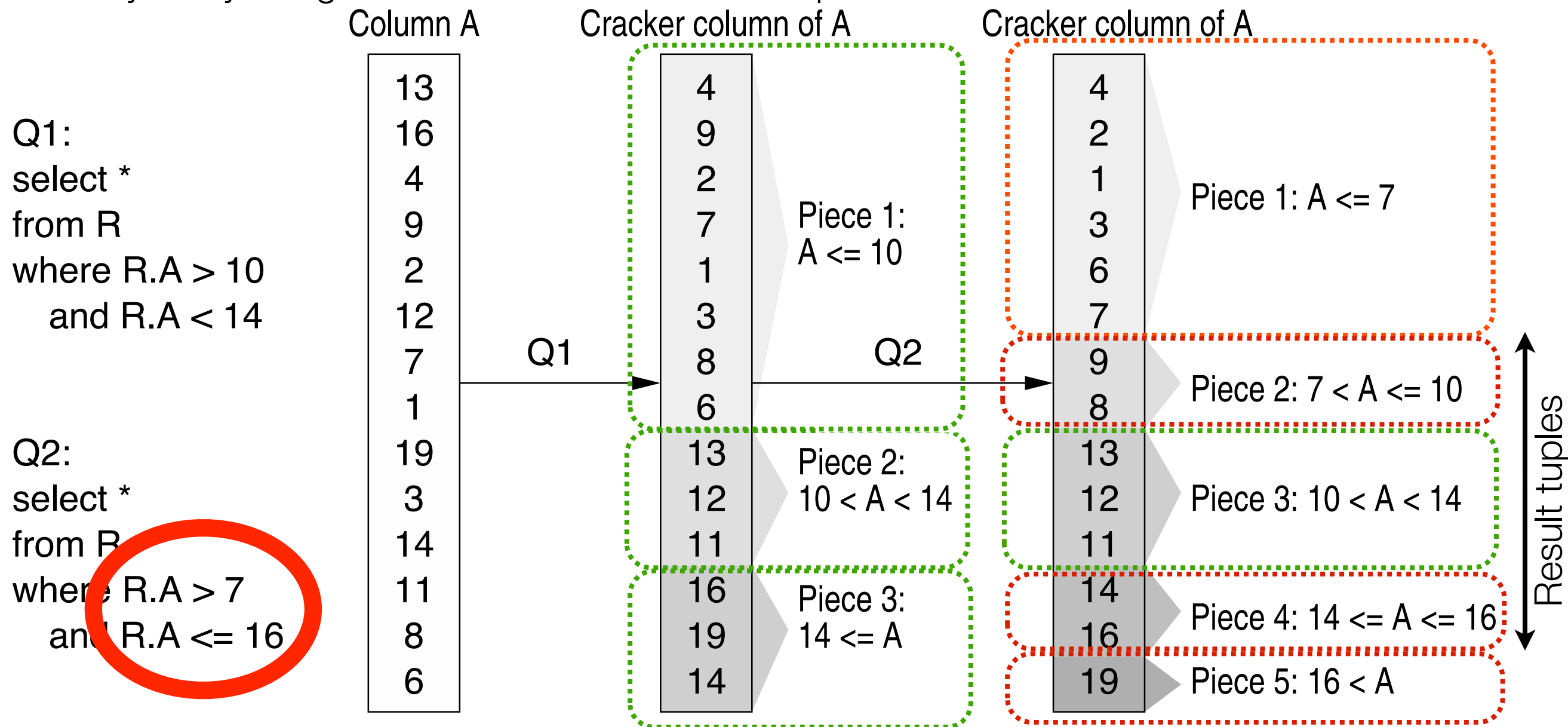


Dynamically/on-the-fly within the select-operator

# Cracking Example

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

Physically reorganize based on the selection predicate



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

# Cracking Example

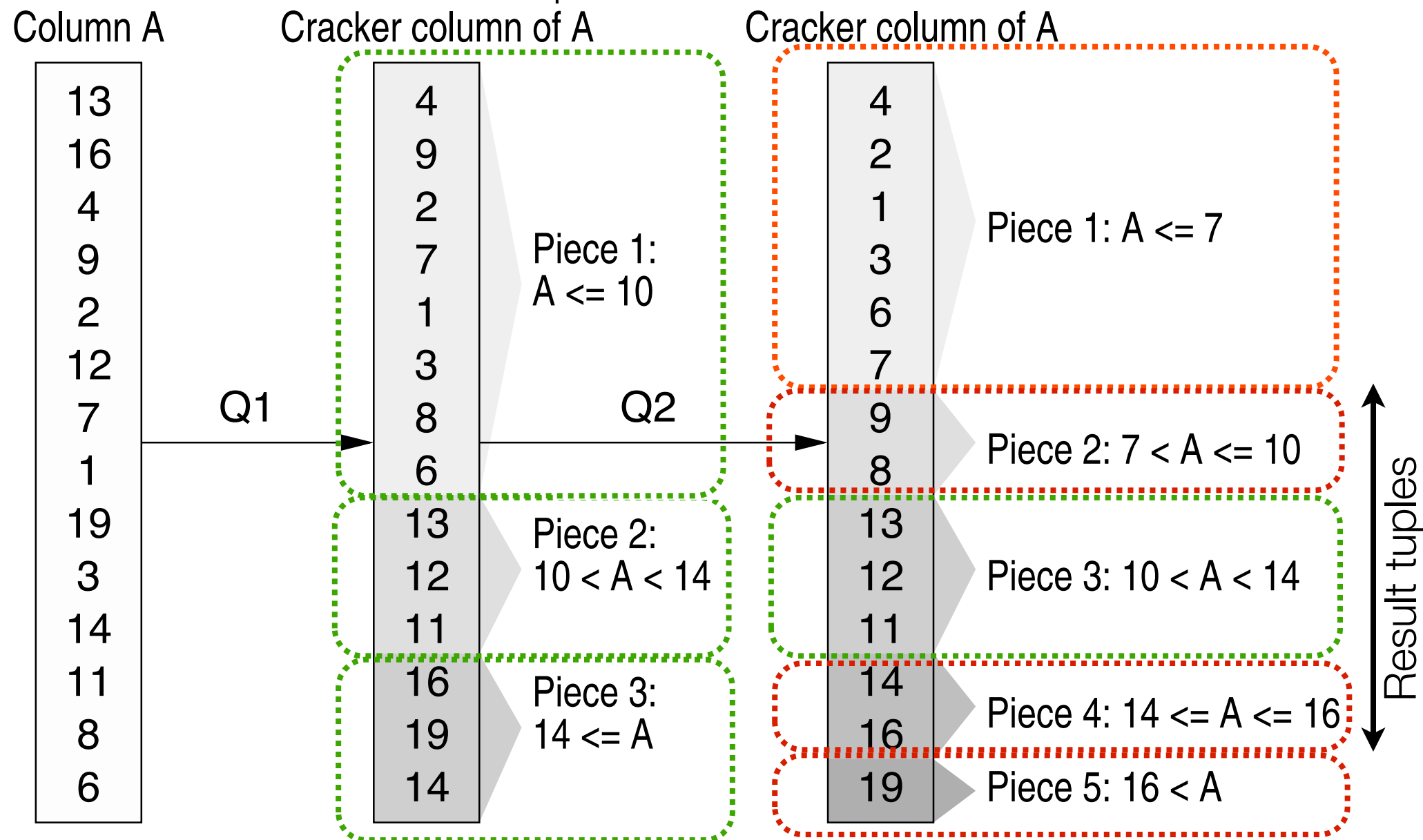
The more we crack, the more we learn

Each query is treated as an advice on how data should

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



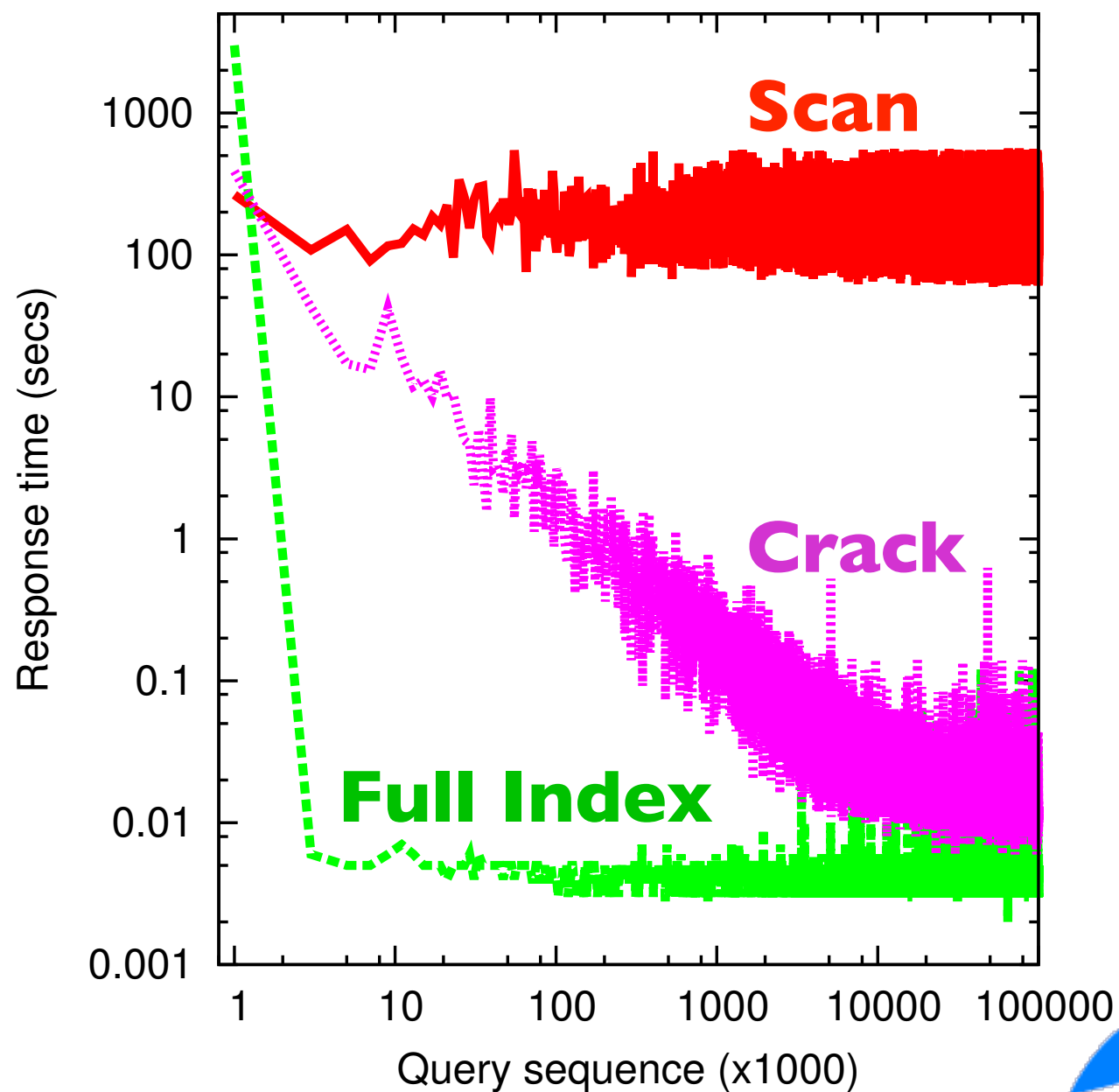
Dynamically/on-the-fly within the select-operator

# Cracking Example

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

## set-up

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



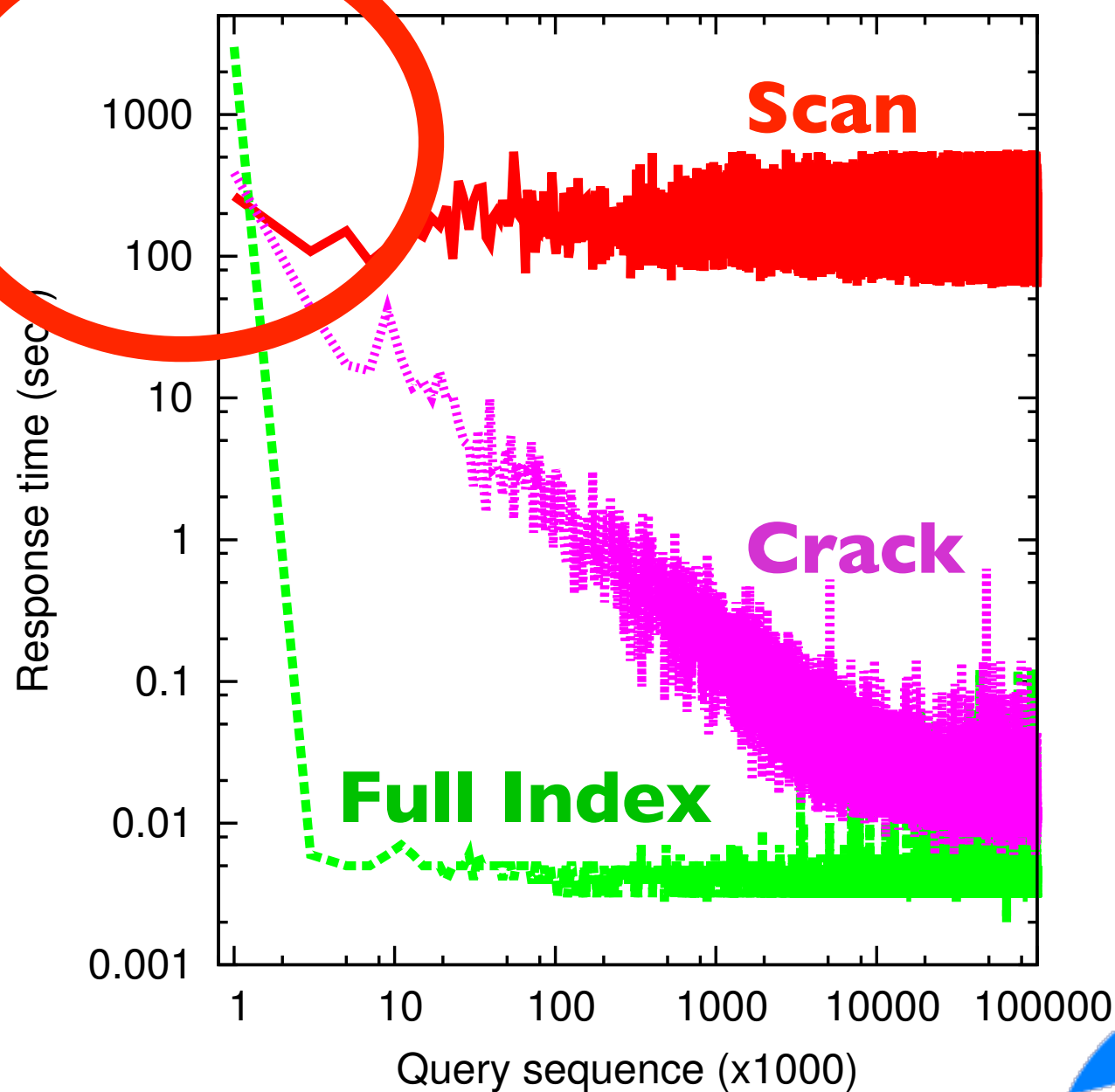
# Cracking Example

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

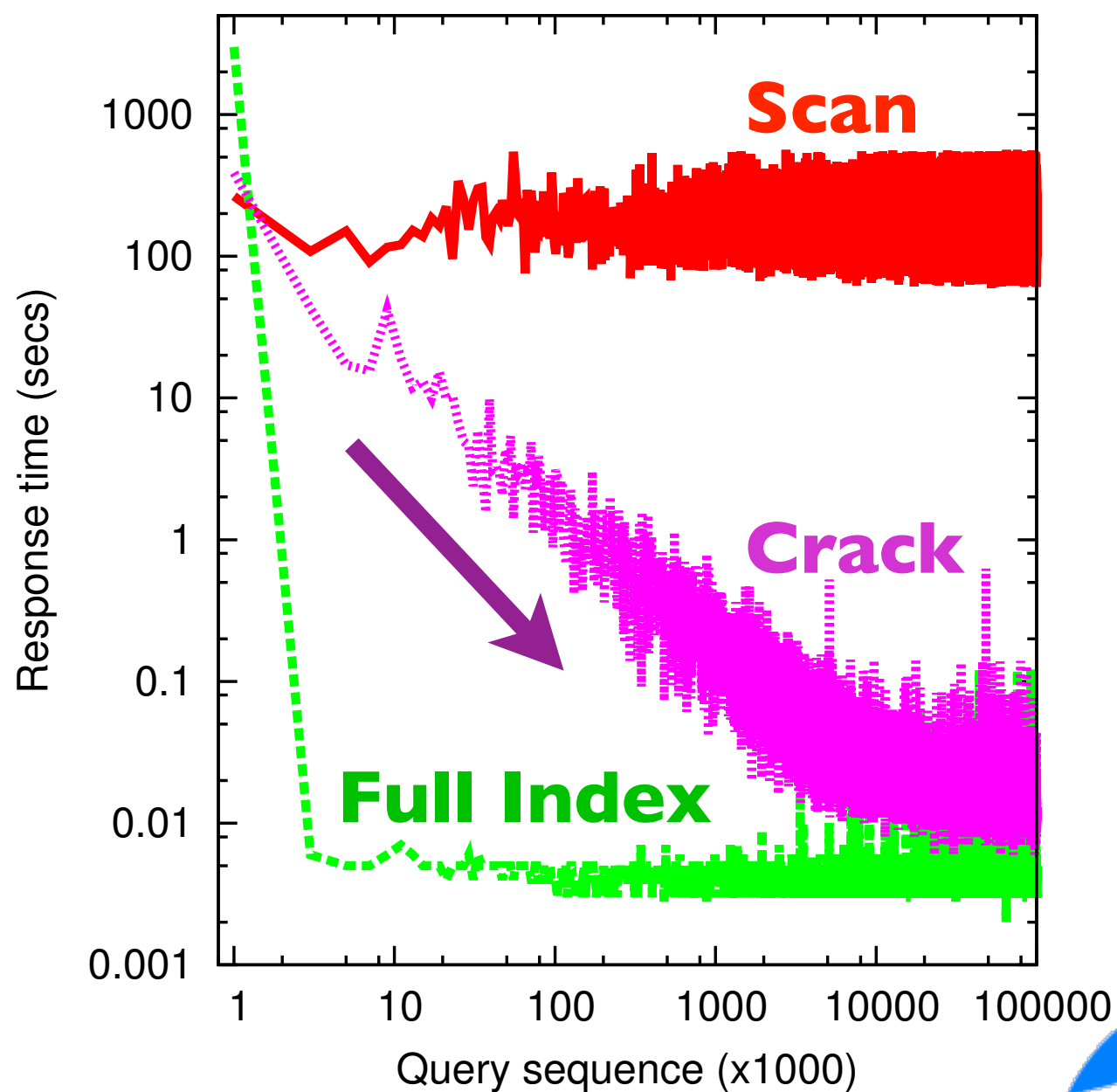
Each query is treated as an advice on how data should be stored

## set-up

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

**almost no  
initialization overhead**

**continuous improvement**



# Cracking Example

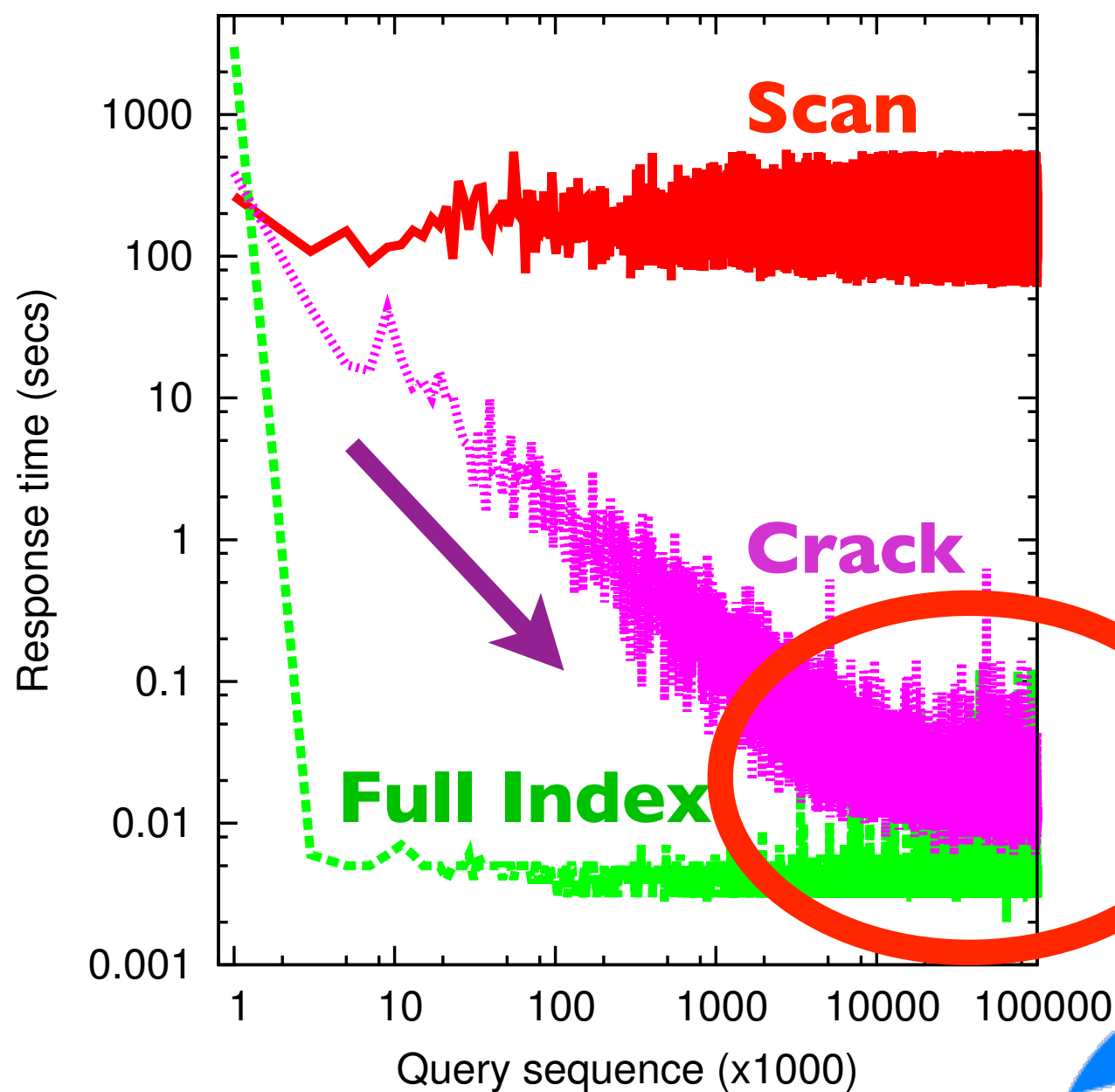
Each query is treated as an advice on how data should be stored

## set-up

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

**almost no  
initialization overhead**

**continuous improvement**



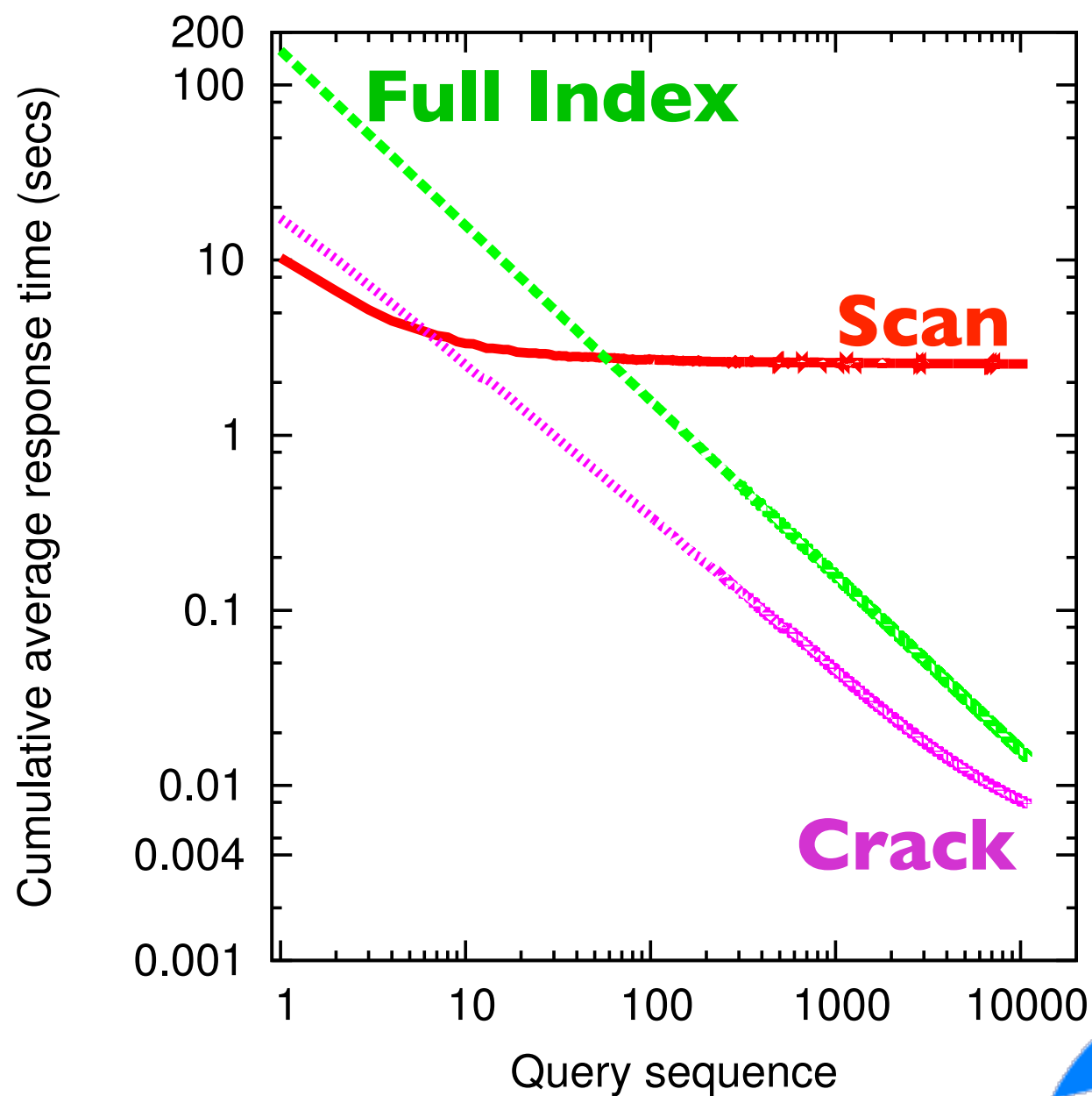


# Cracking Example

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



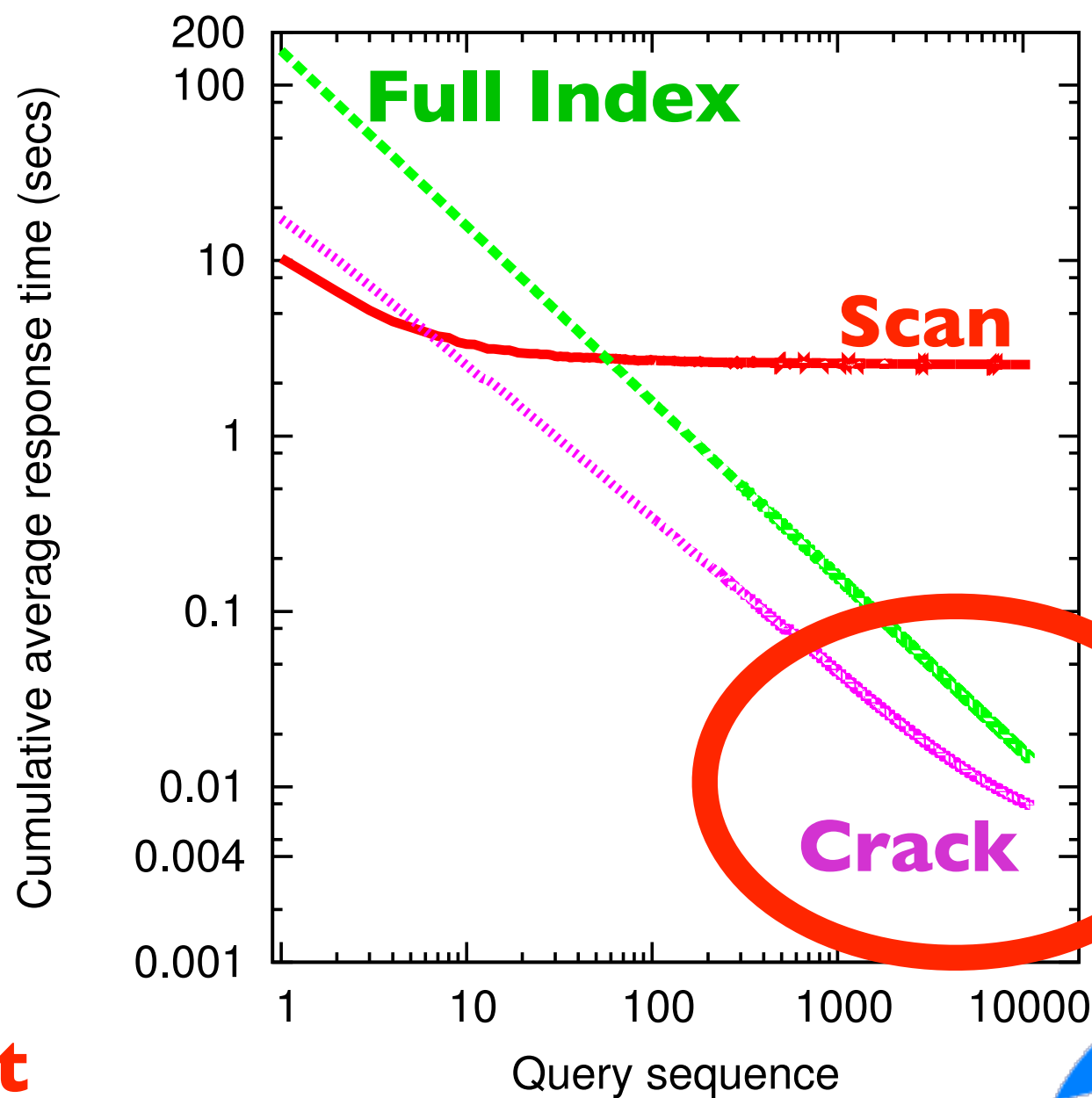


# Cracking Example

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

## set-up

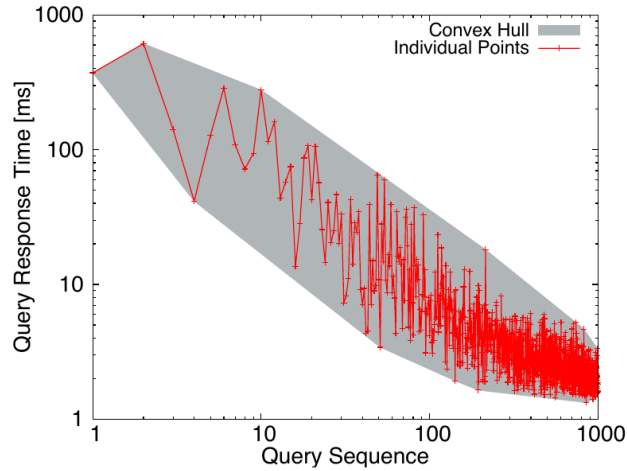
10K random selections  
selectivity 10%  
random value ranges  
in a 30 million integer column



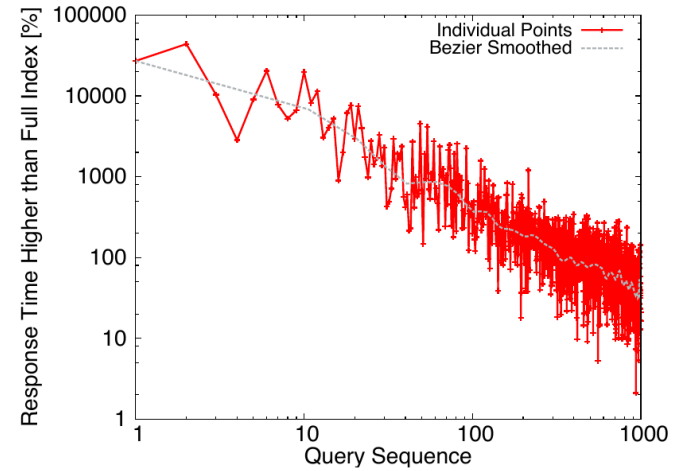
**10K queries later,  
Full Index still has not  
amortized the initialization costs**

# Problems

### High Variance



### Low Convergence Speed



### Low Robustness



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

Felix Halim\*

\*National University of Singapore  
{halim, ryap}@comp.nus.edu.sg

Stratos Idreos†

†CWI, Amsterdam  
idreos@cwi.nl

Panagiotis Karras◇

◇Rutgers University  
karras@business.rutgers.edu

Roland H. C. Yap\*

## 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

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

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 invited 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 2150-8097/12/02... \$ 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 Idreos, Panagiotis Karras and Roland Y. Chuan

# Workload Robustness

## **Observation:**

Queries define adaptive indexing actions

The kind of queries and the order of queries matter!

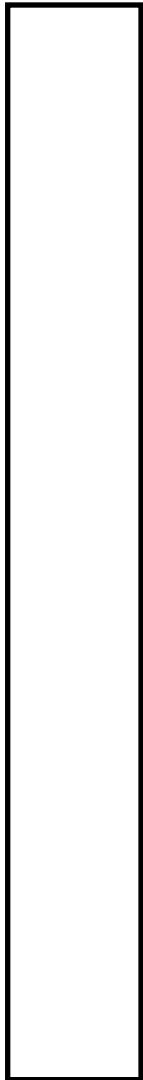
## **Goal:**

Maintain adaptive behavior regardless of query input

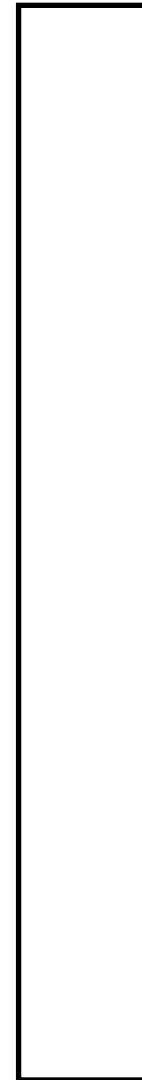
# Query patterns

column with 100 unique integers

Good pattern



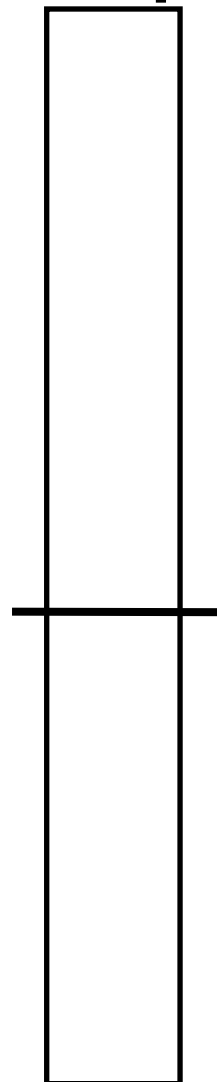
Bad pattern



# Query patterns

column with 100 unique integers

Good pattern



$q_l, v > 60$  N

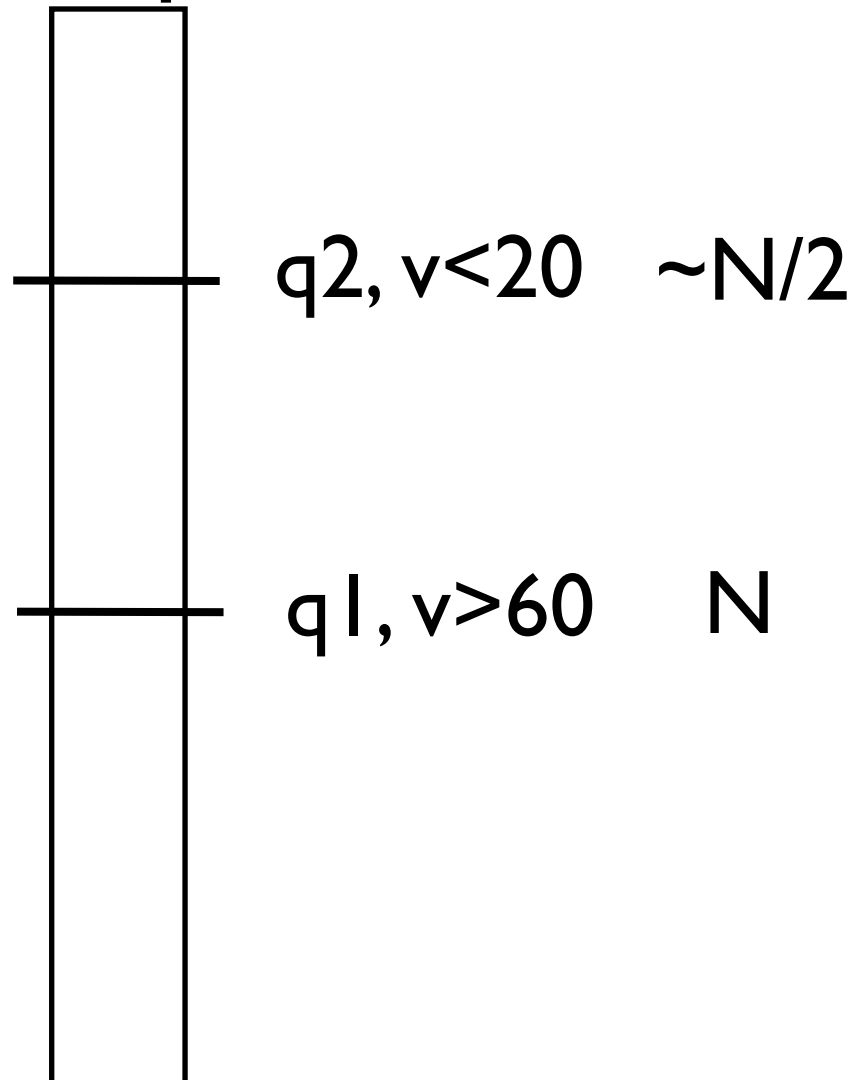
Bad pattern



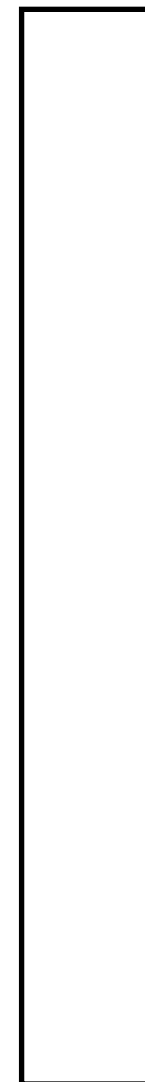
# Query patterns

column with 100 unique integers

Good pattern



Bad pattern

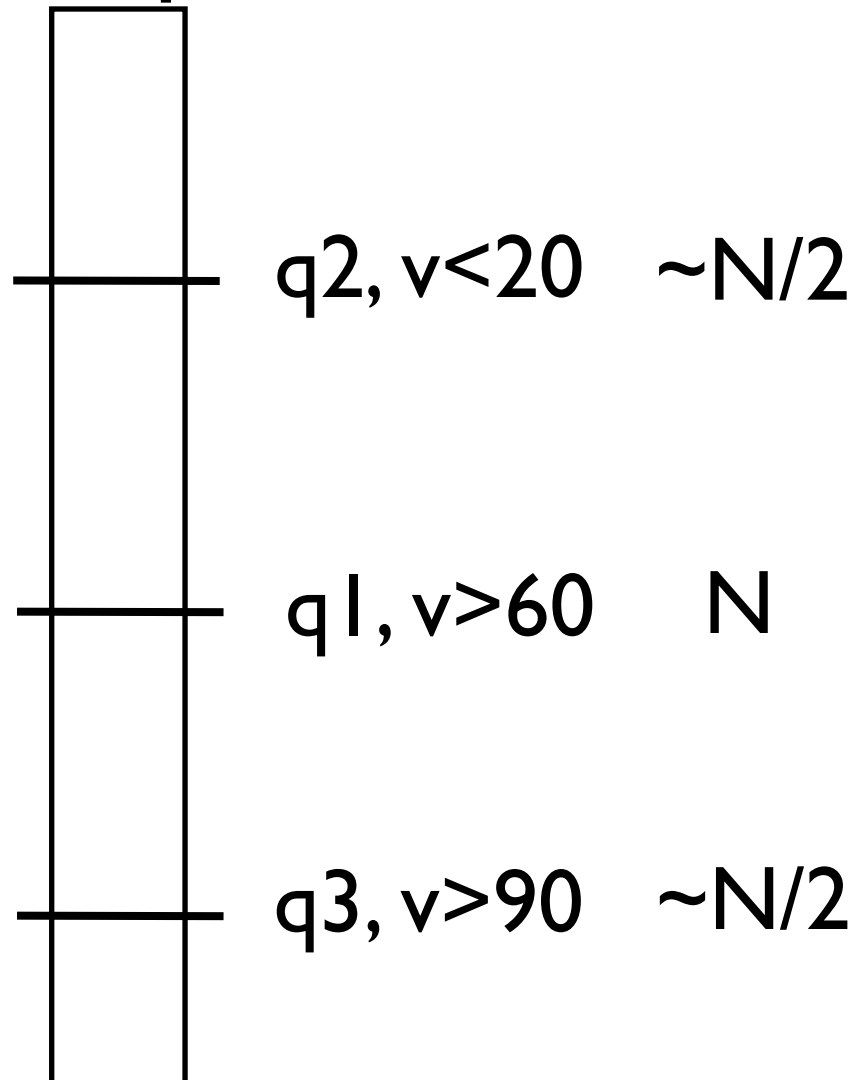




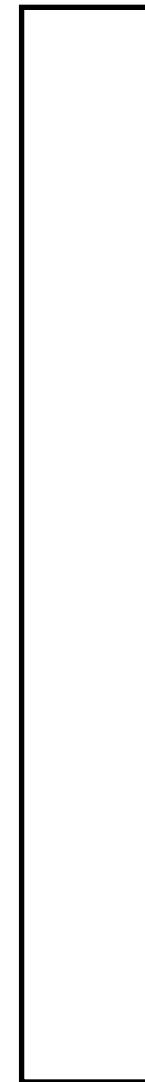
# Query patterns

column with 100 unique integers

Good pattern



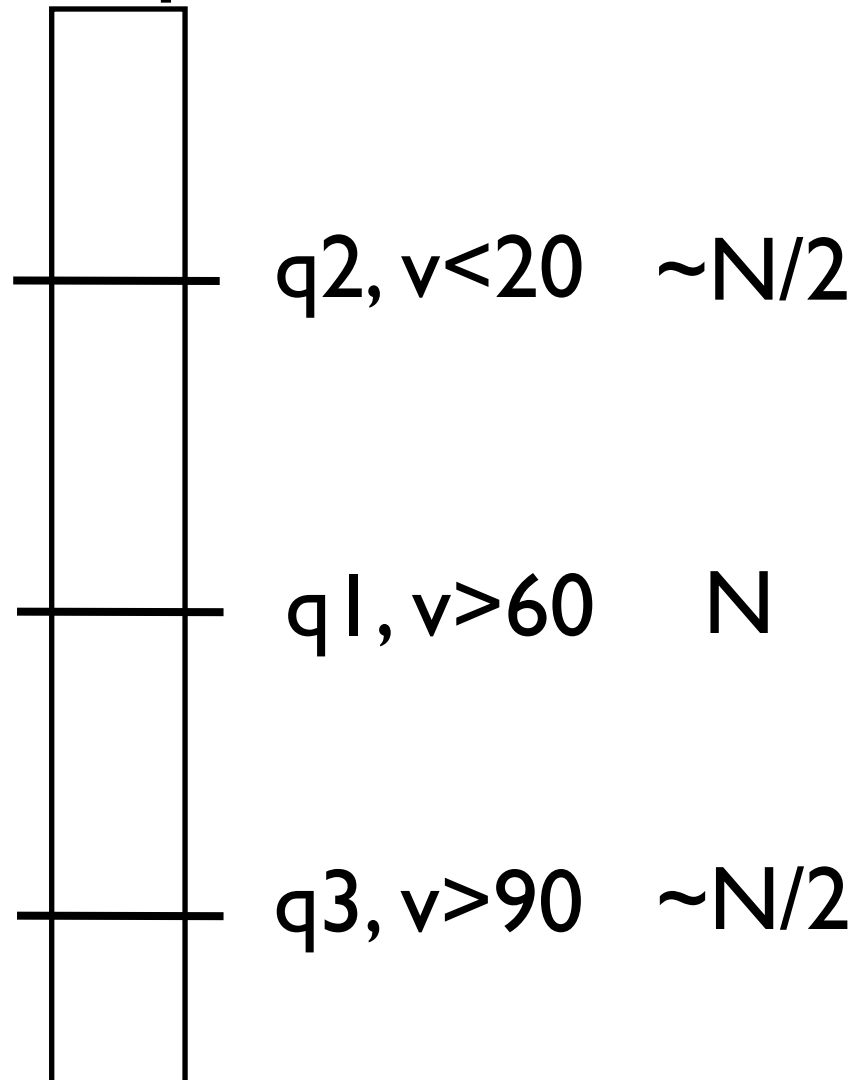
Bad pattern



# Query patterns

column with 100 unique integers

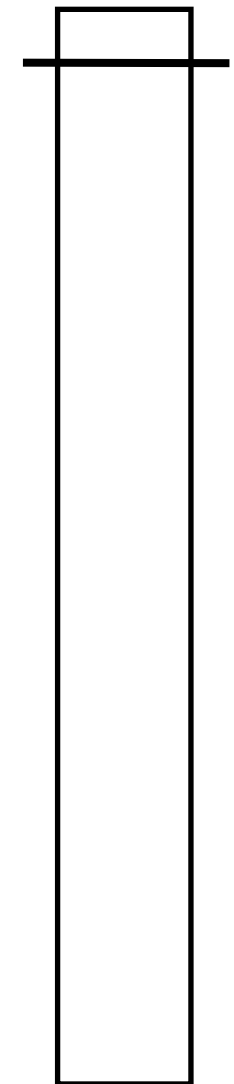
Good pattern



$N$

$q_1, v < 1$

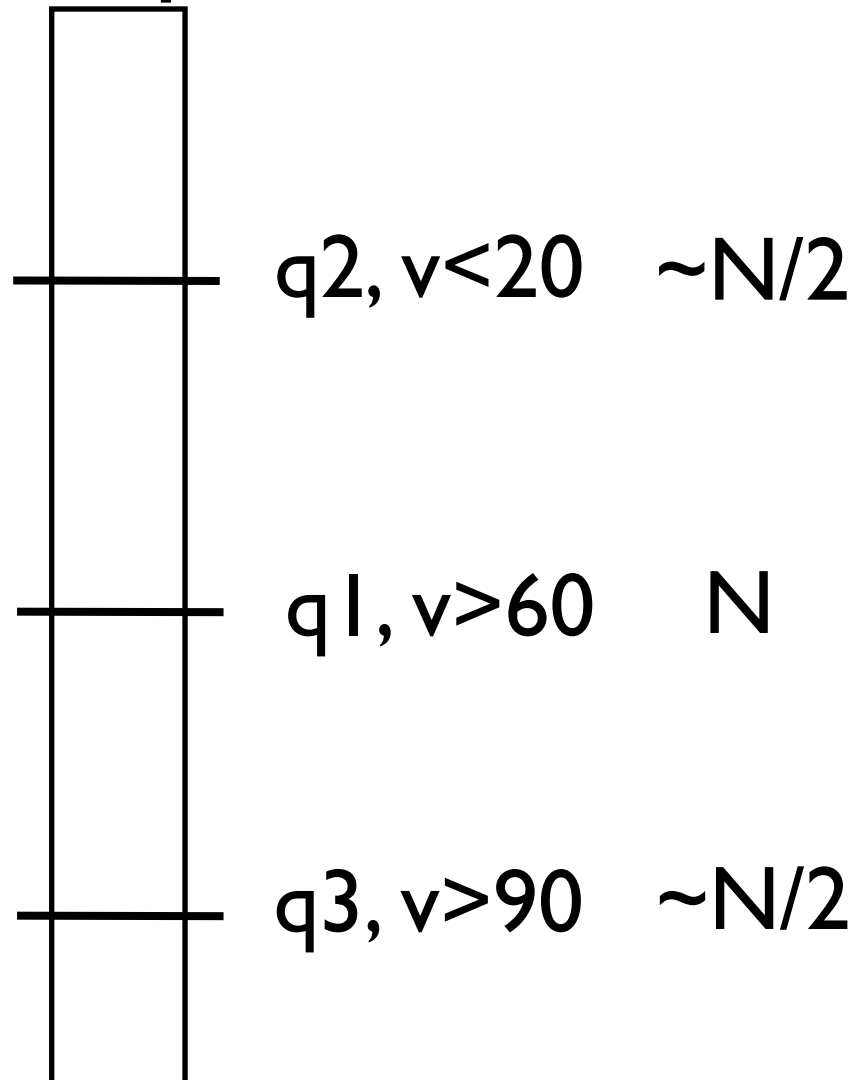
Bad pattern



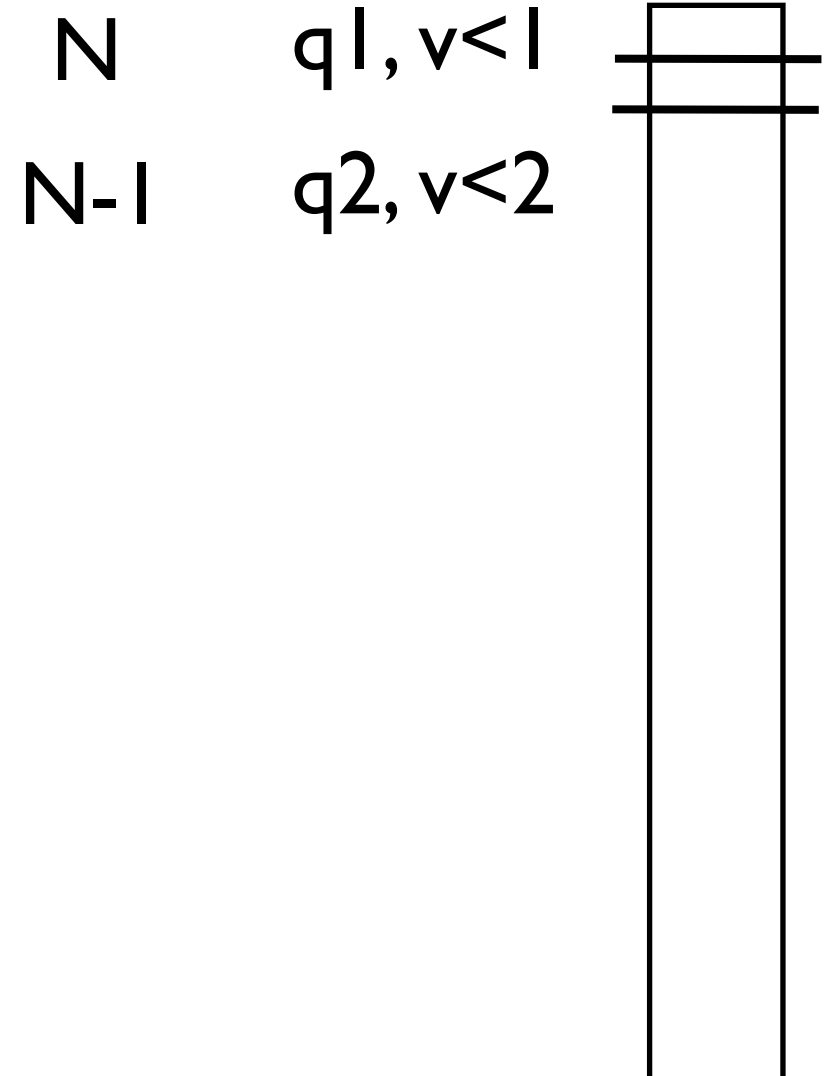
# Query patterns

column with 100 unique integers

Good pattern



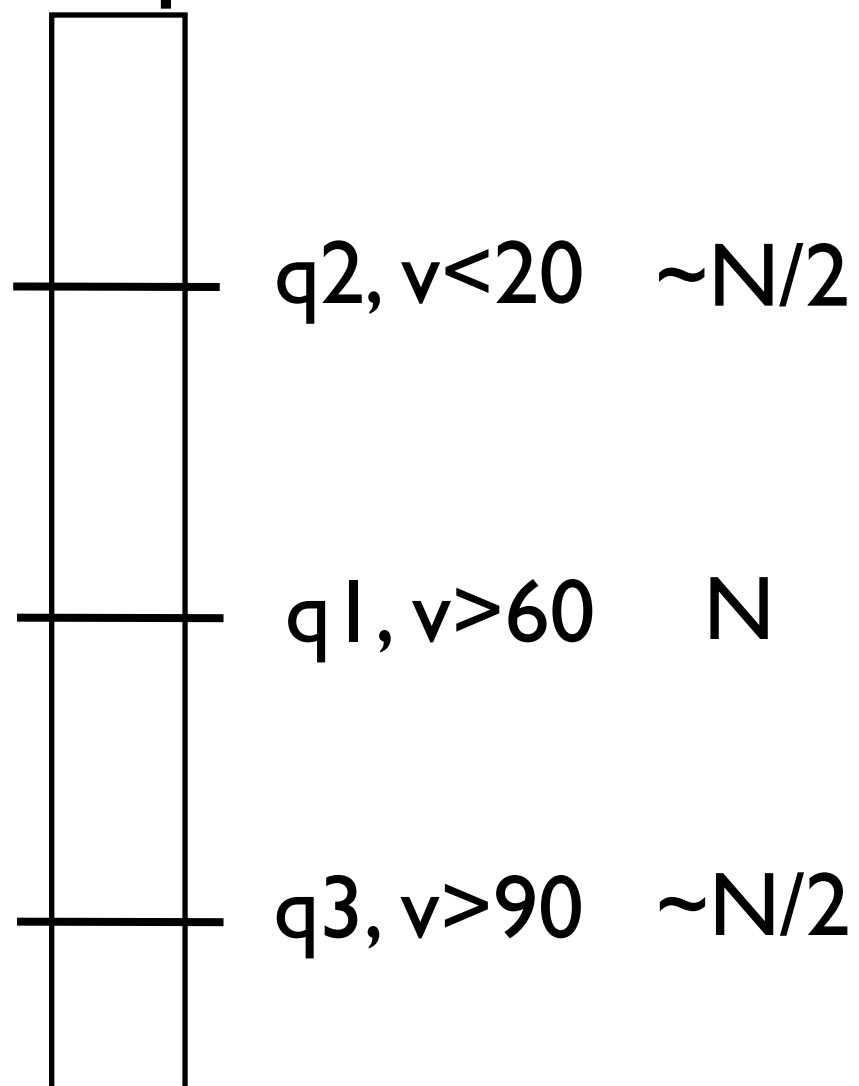
Bad pattern



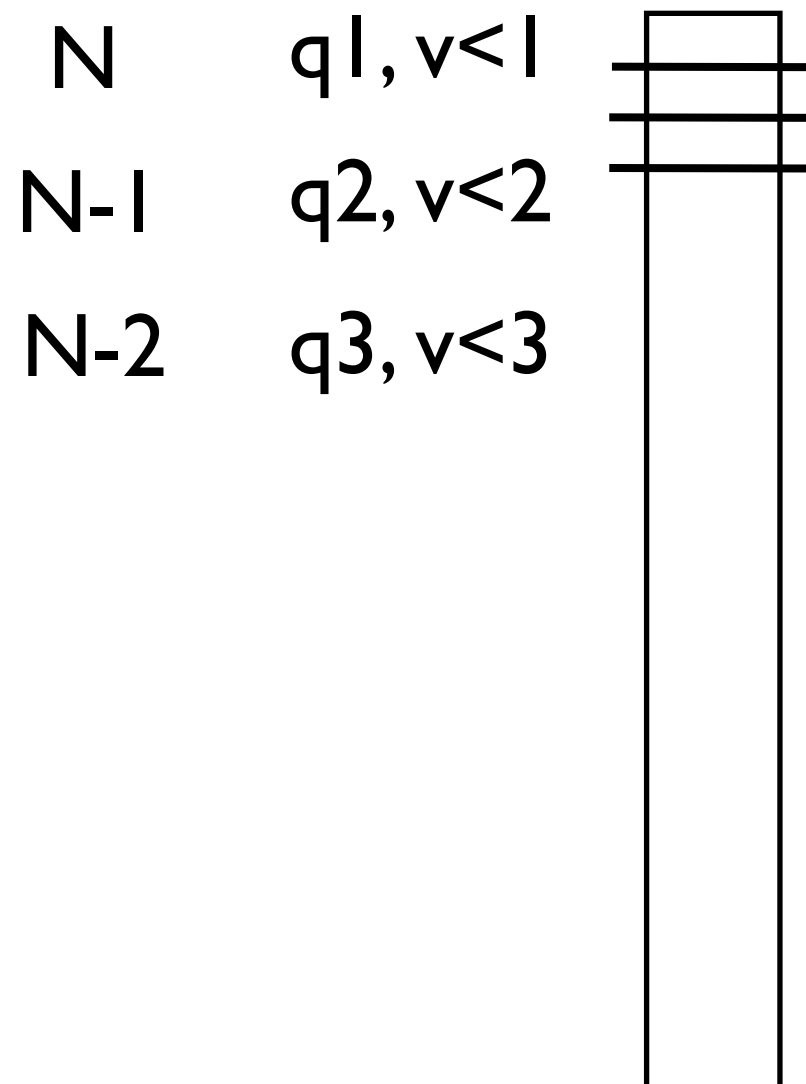
# Query patterns

column with 100 unique integers

Good pattern

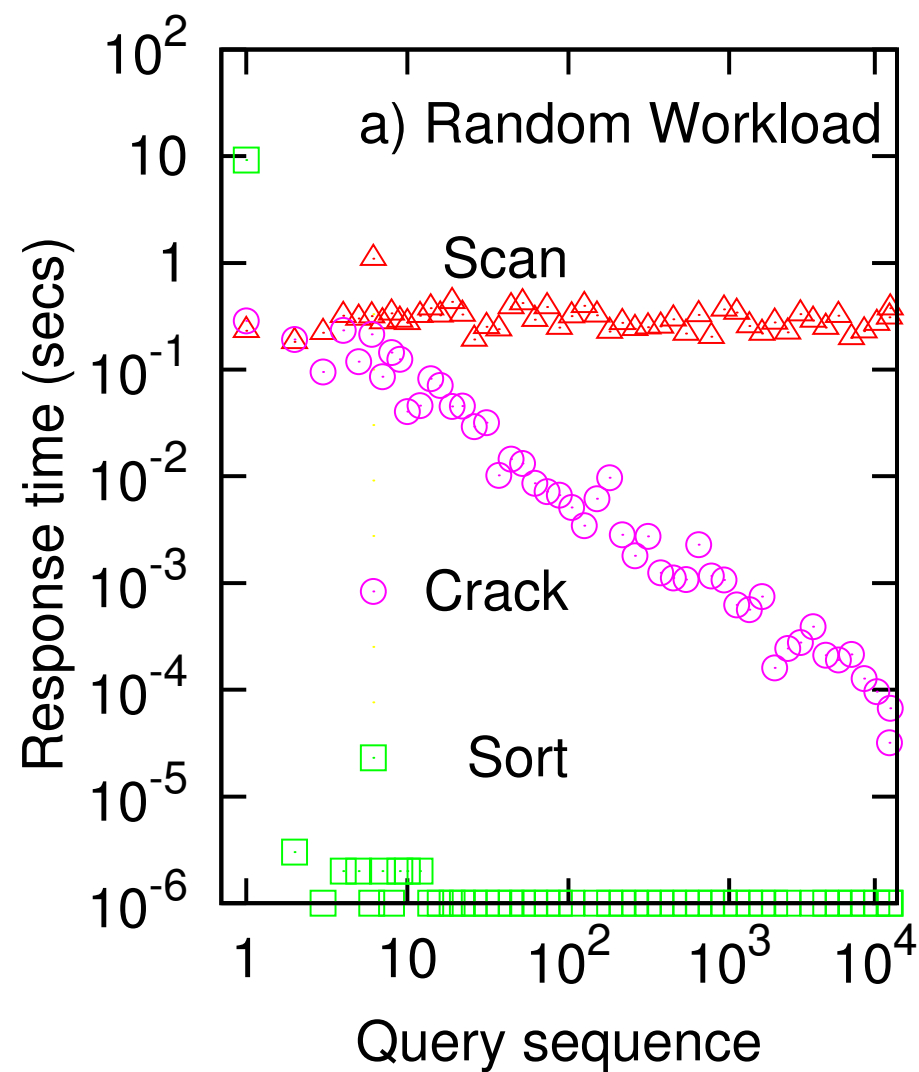


Bad pattern



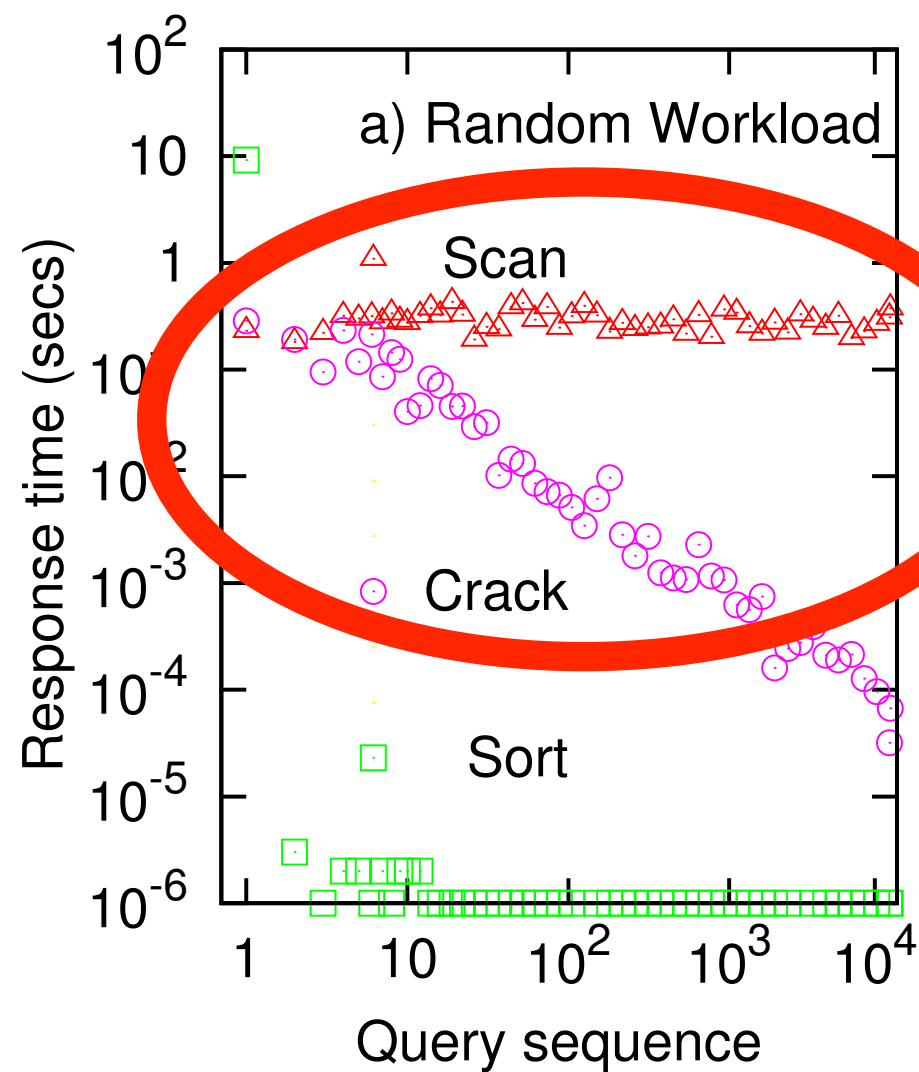
# Query patterns

column size 100M  
selectivity 10 tuples



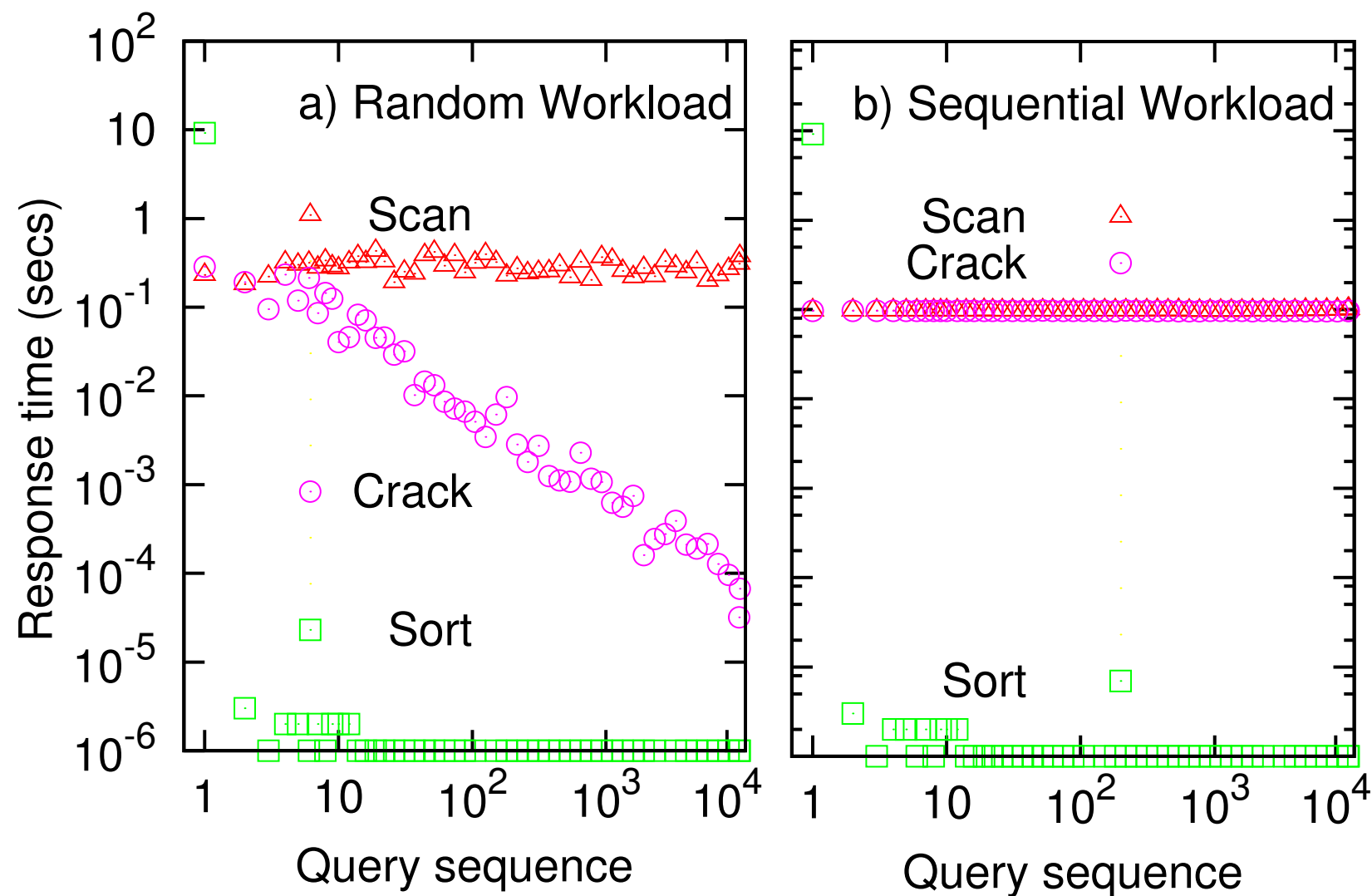
# Query patterns

column size 100M  
selectivity 10 tuples



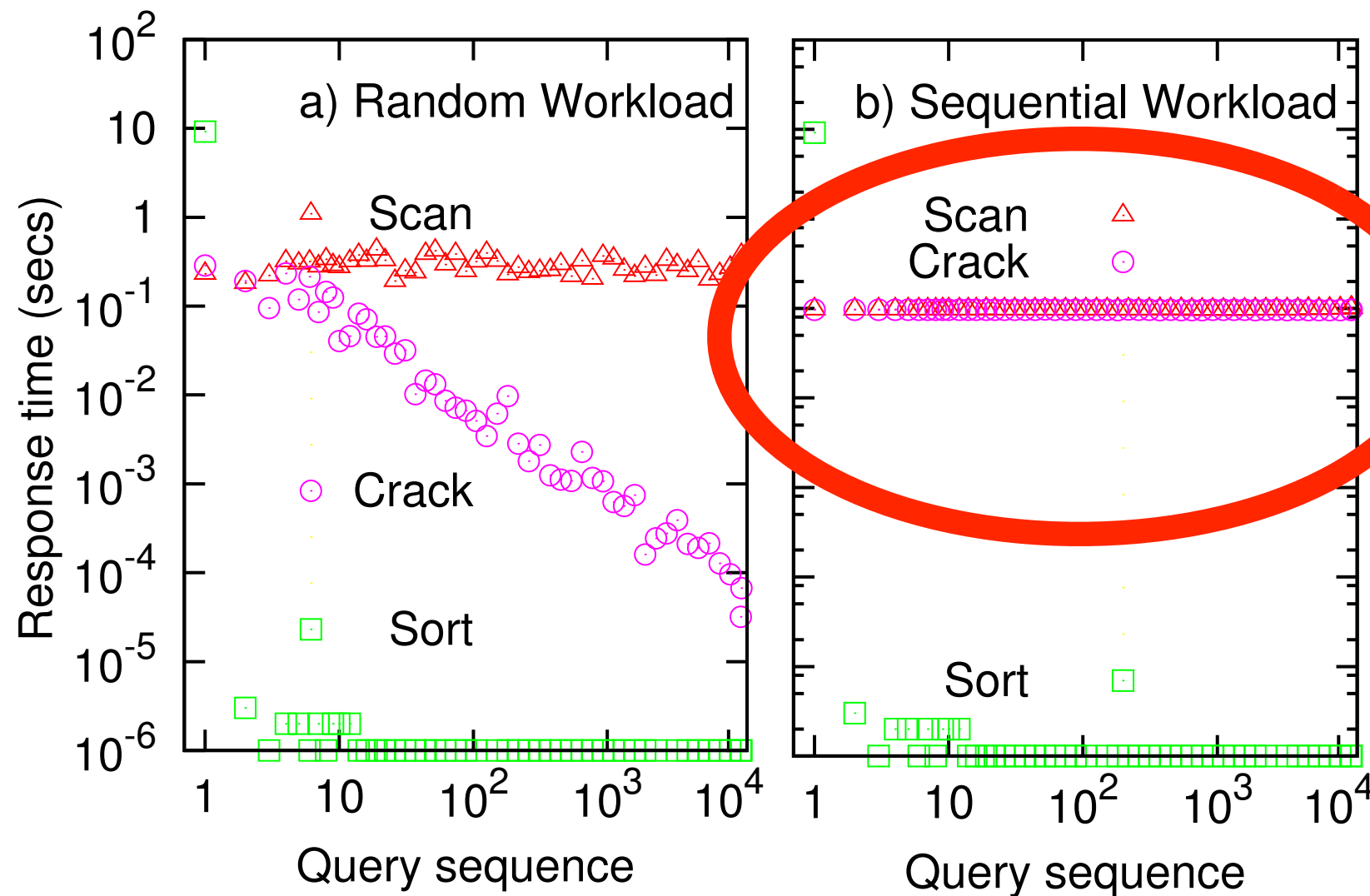
# Query patterns

column size 100M  
selectivity 10 tuples



# Query patterns

column size 100M  
selectivity 10 tuples

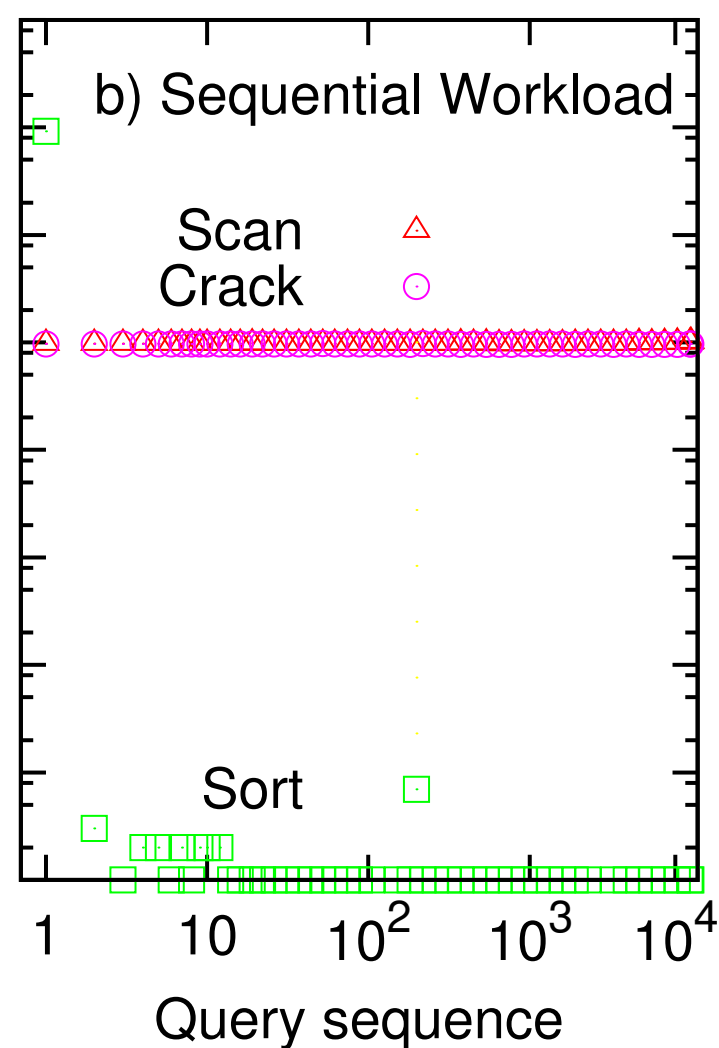
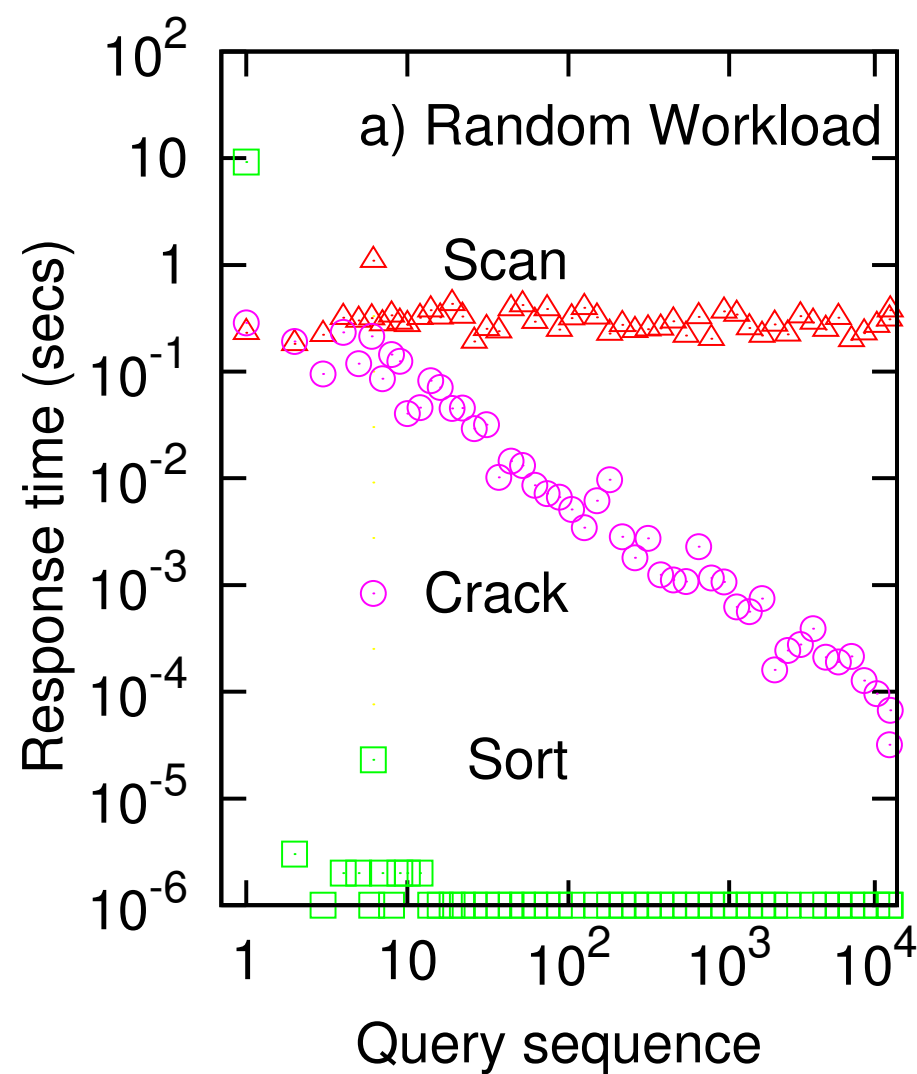


performance degrades to scan

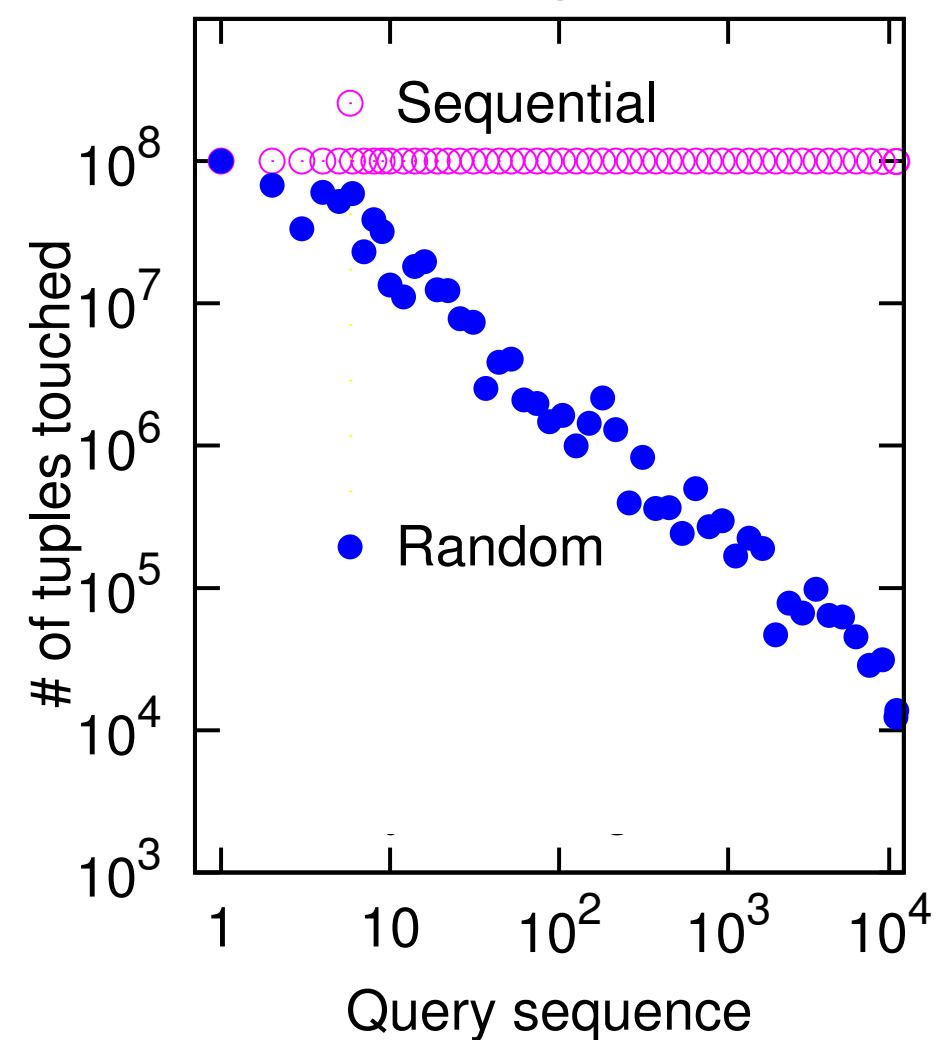


# Query patterns

column size 100M  
selectivity 10 tuples



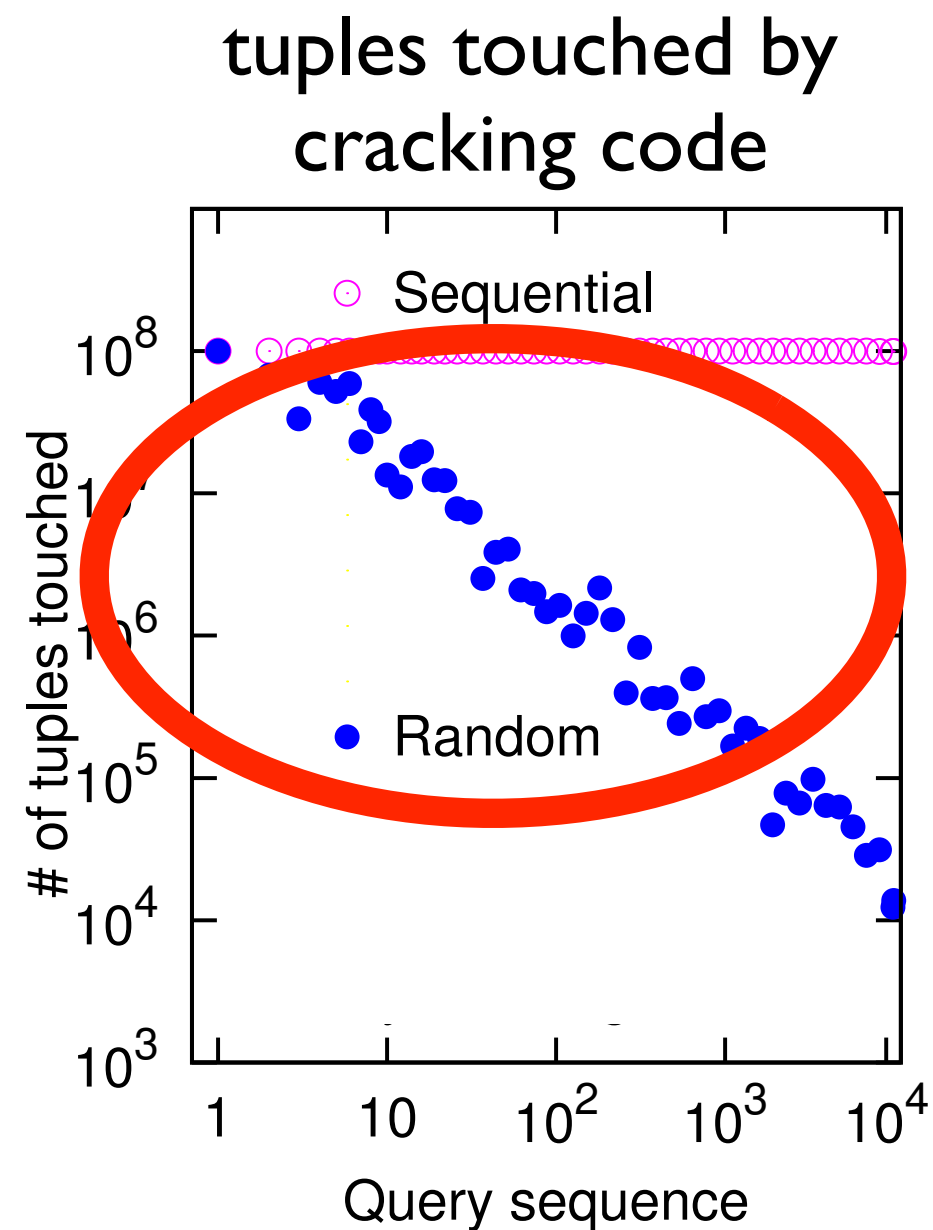
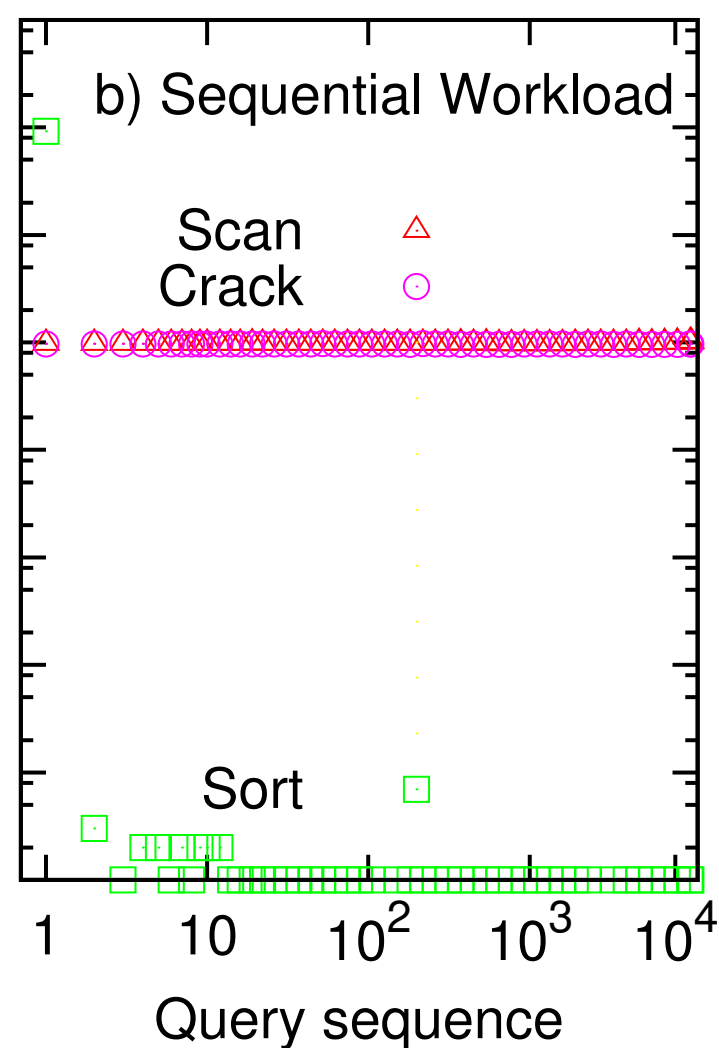
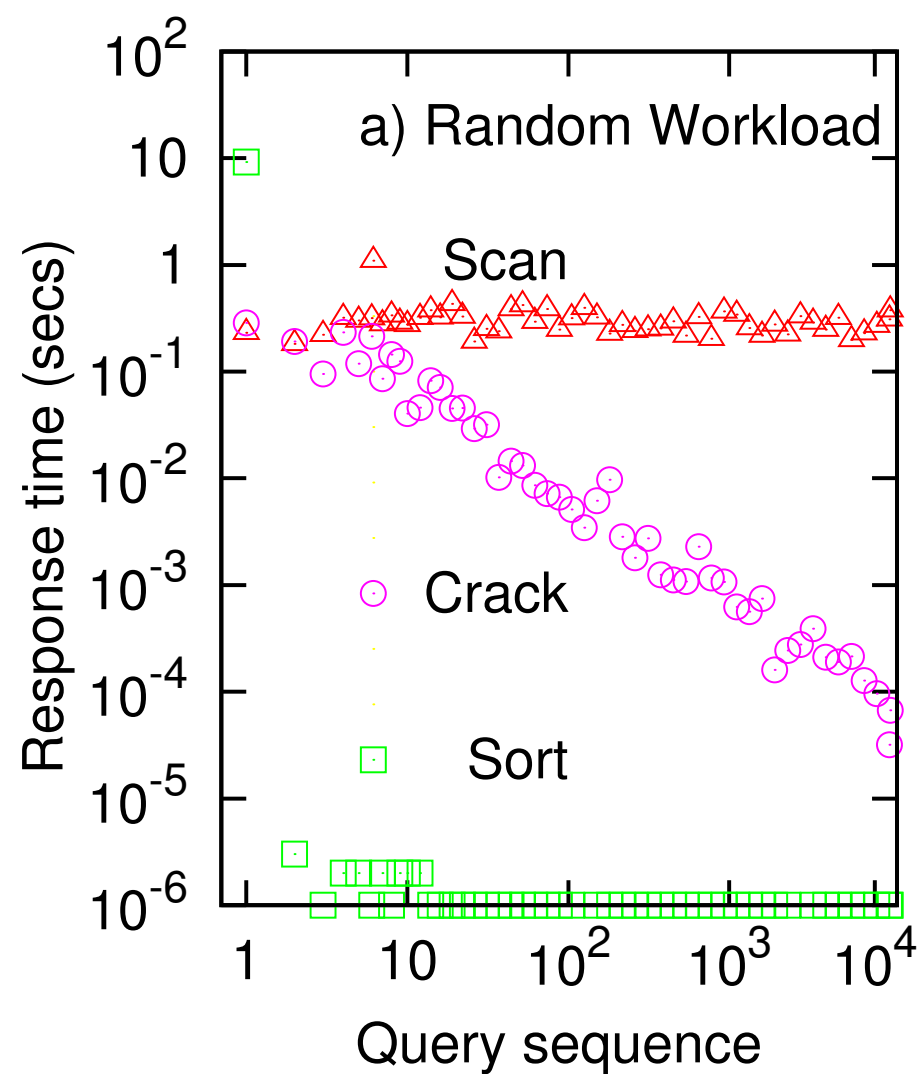
tuples touched by  
cracking code



performance degrades to scan

# Query patterns

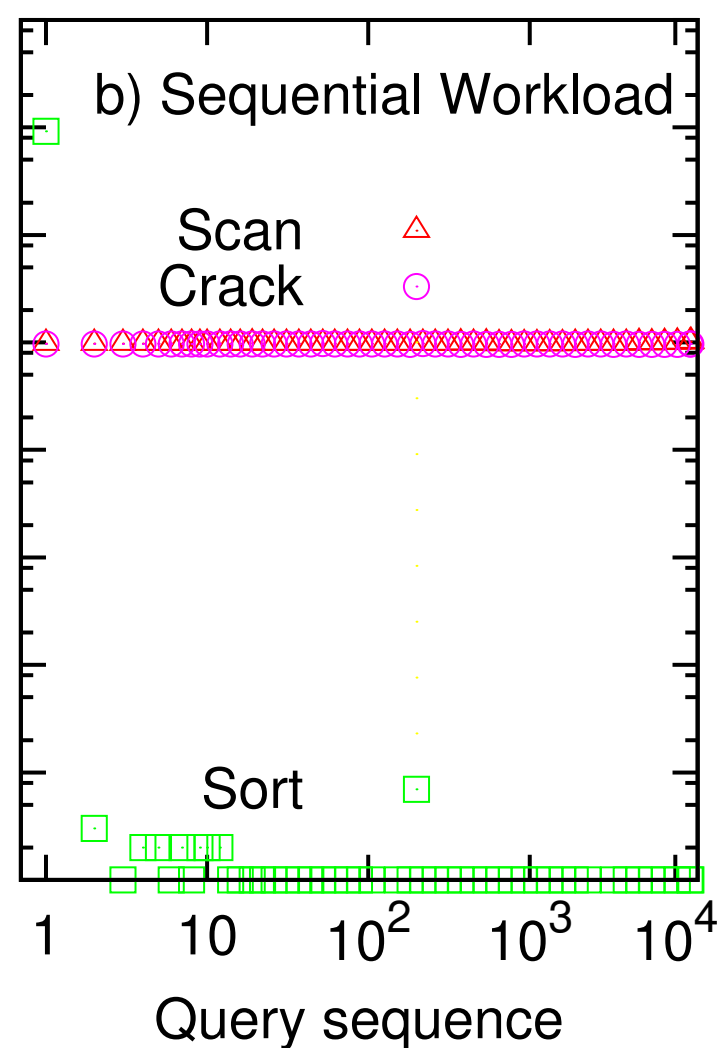
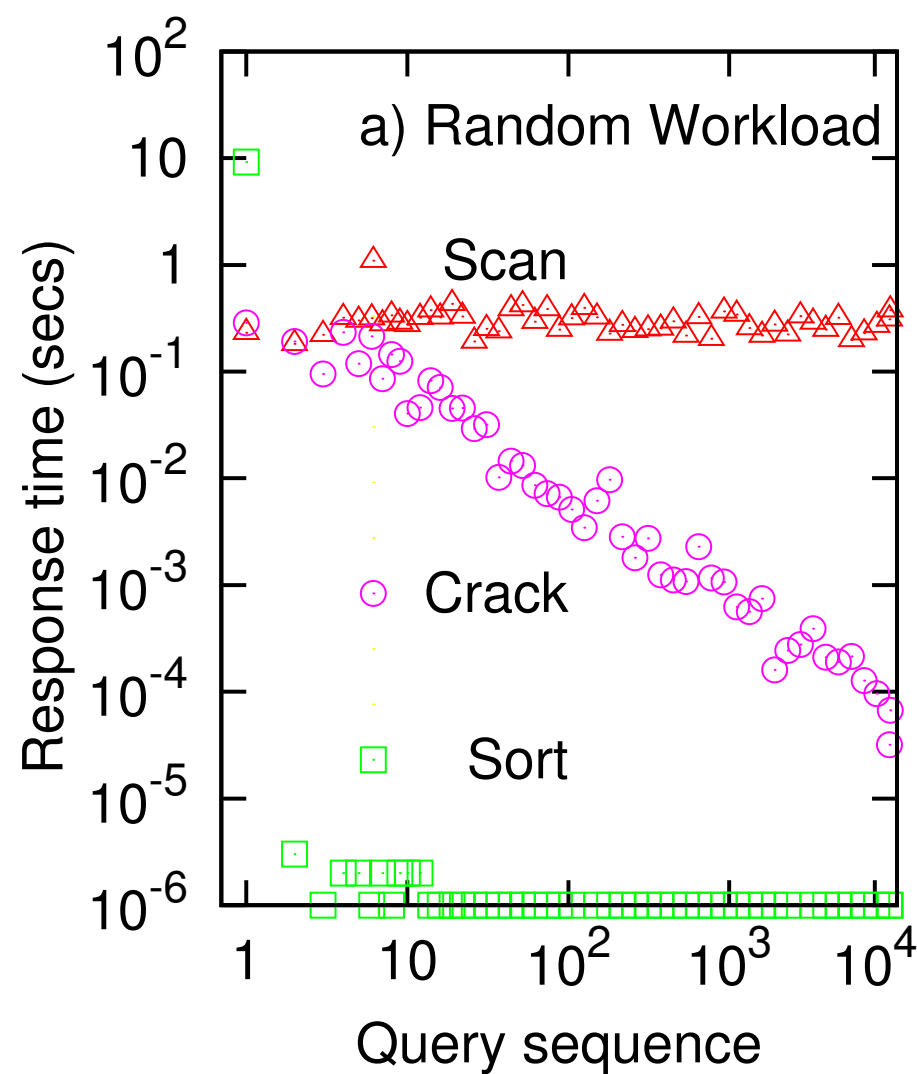
column size 100M  
selectivity 10 tuples



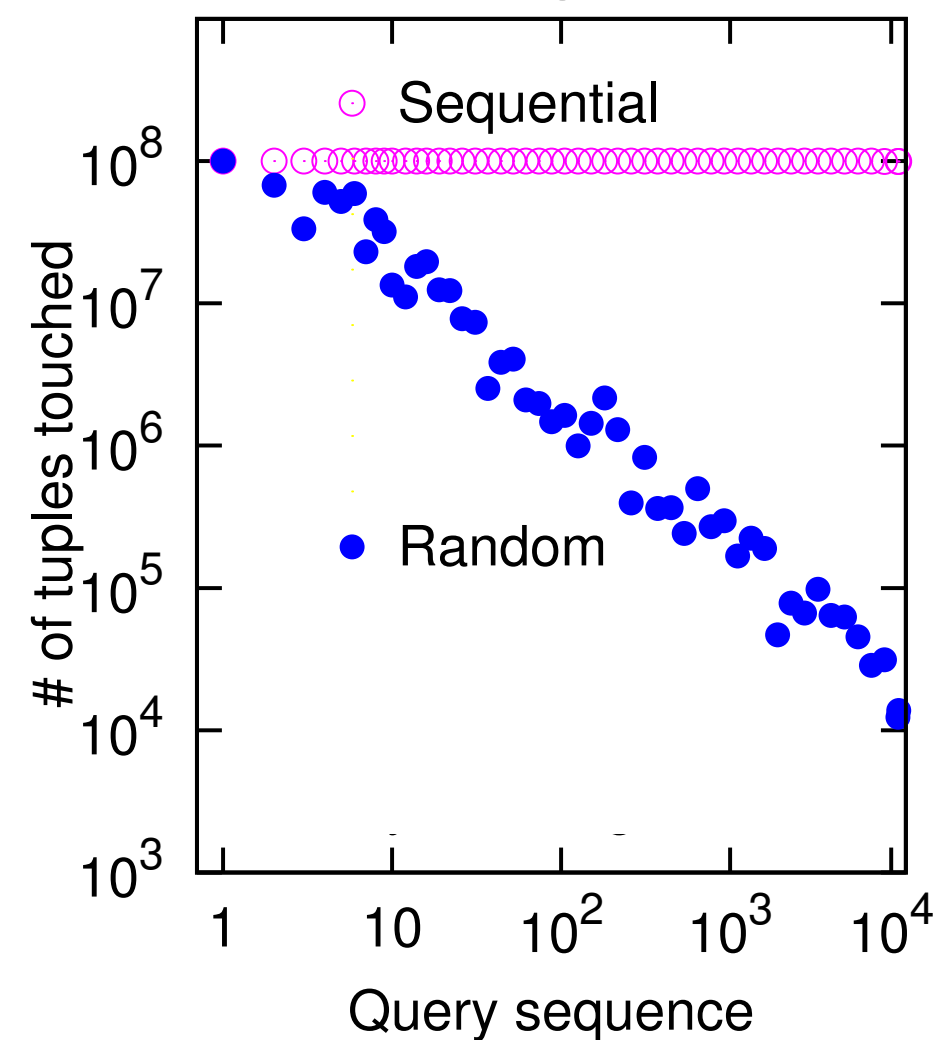
performance degrades to scan

# Query patterns

column size 100M  
selectivity 10 tuples



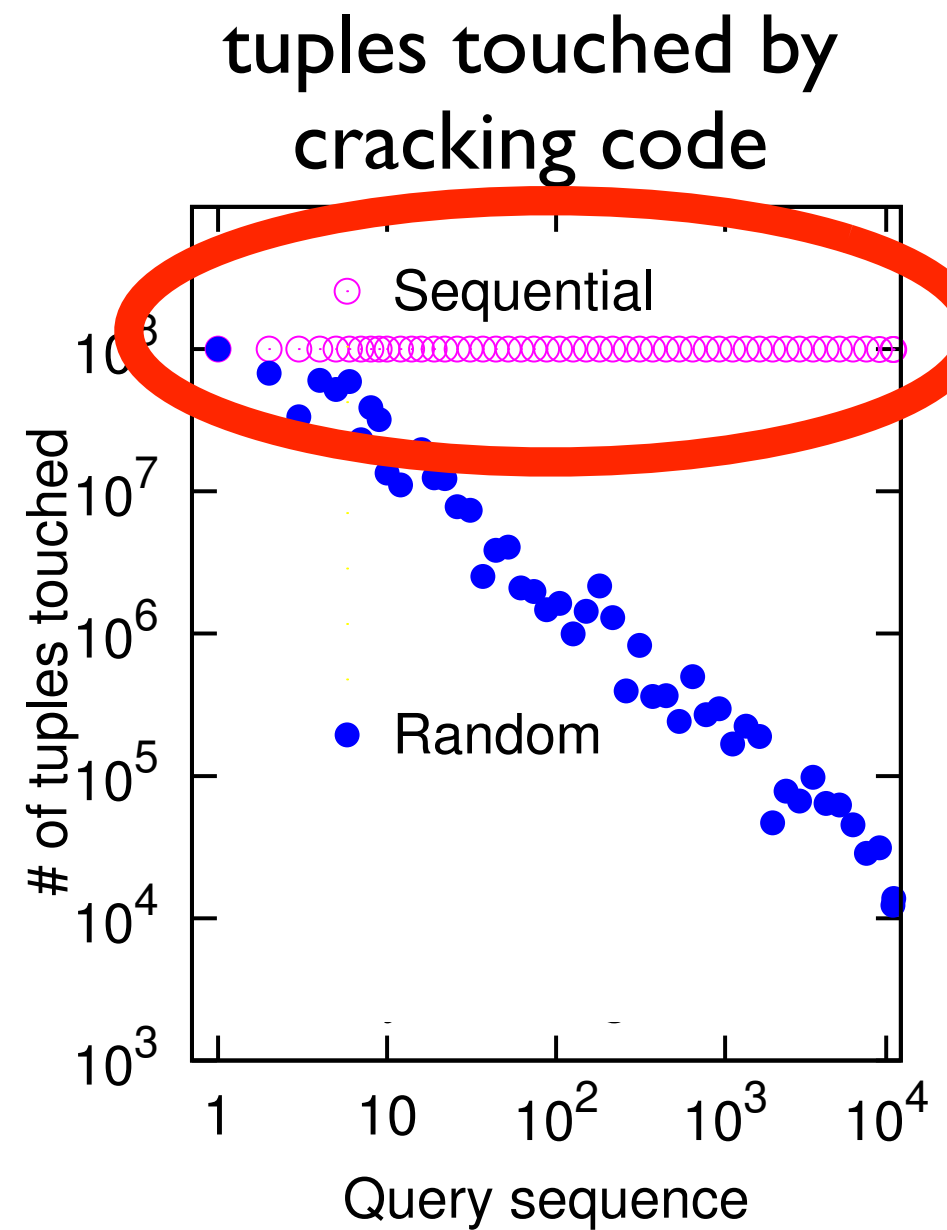
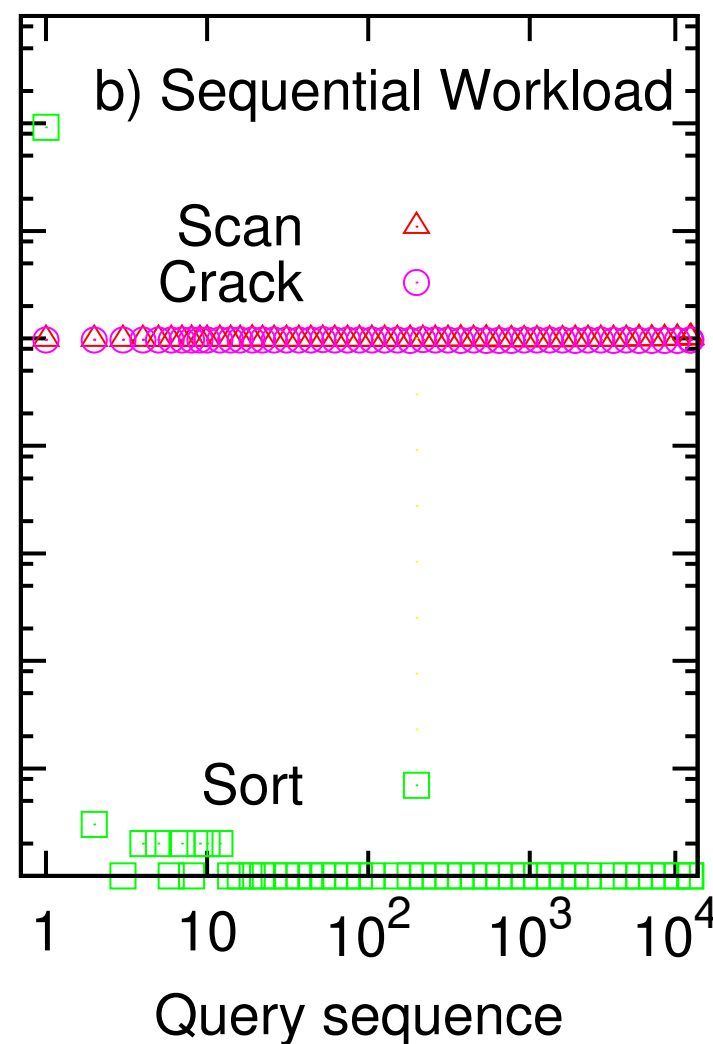
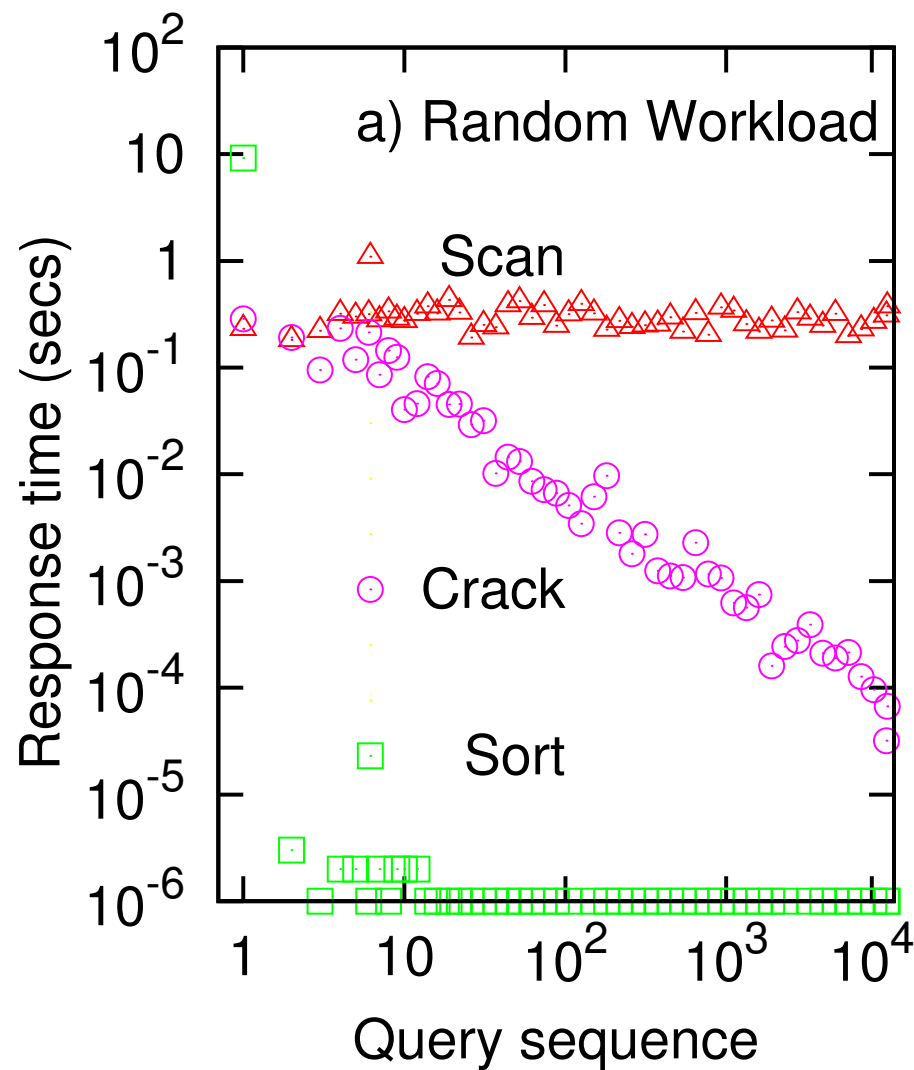
tuples touched by  
cracking code



performance degrades to scan

# Query patterns

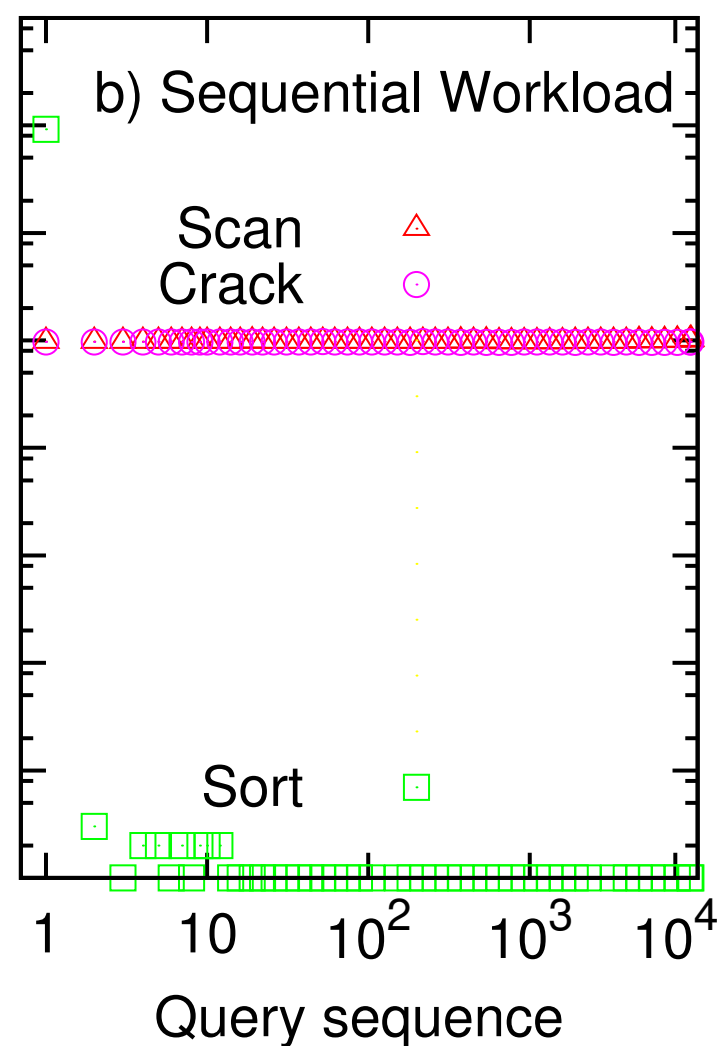
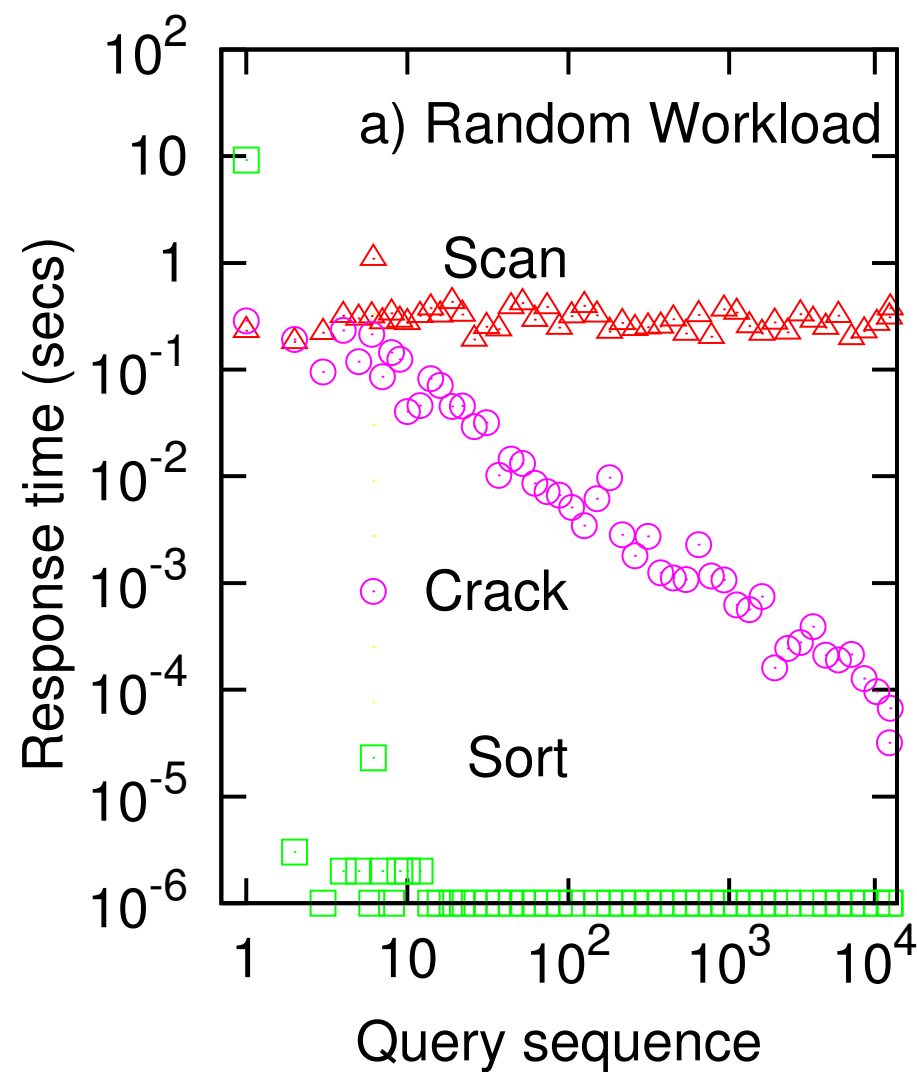
column size 100M  
selectivity 10 tuples



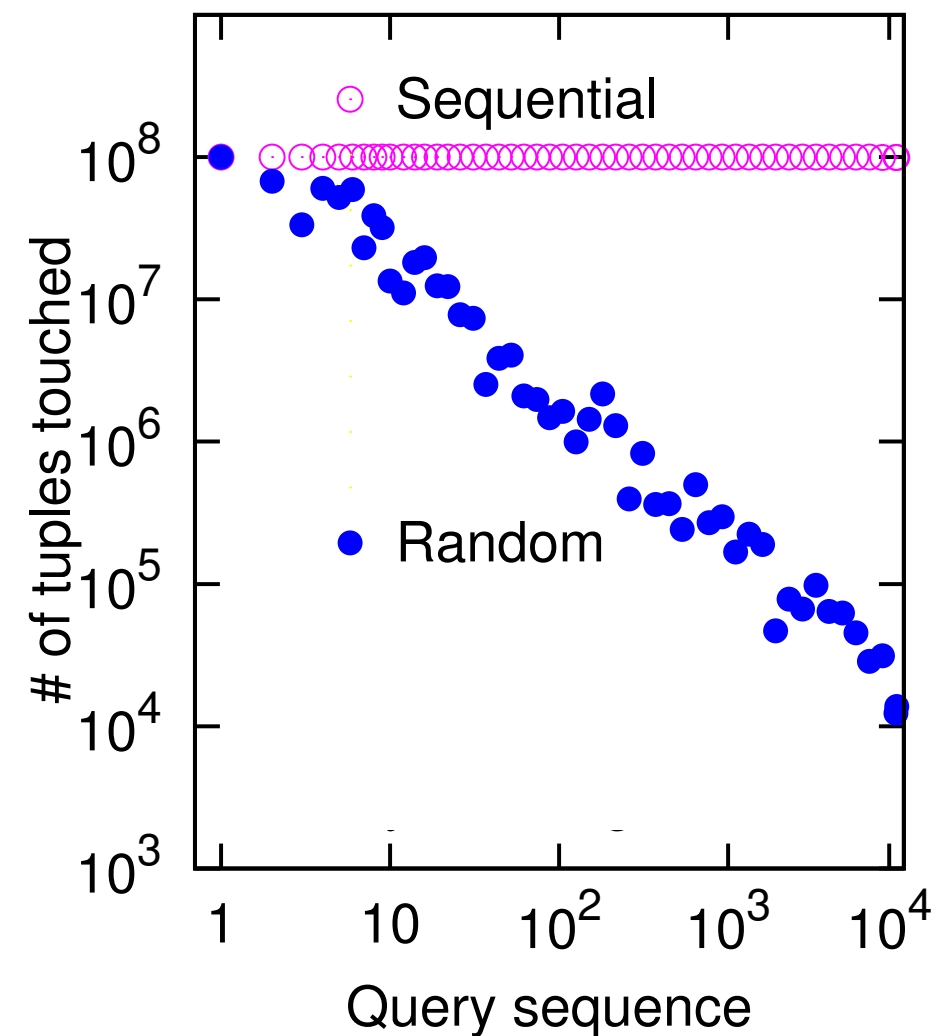
performance degrades to scan

# Query patterns

column size 100M  
selectivity 10 tuples



tuples touched by  
cracking code



performance degrades to scan

# Stochastic Cracking

## **Problem:**

Blind adaptation to queries

## **Solution:**

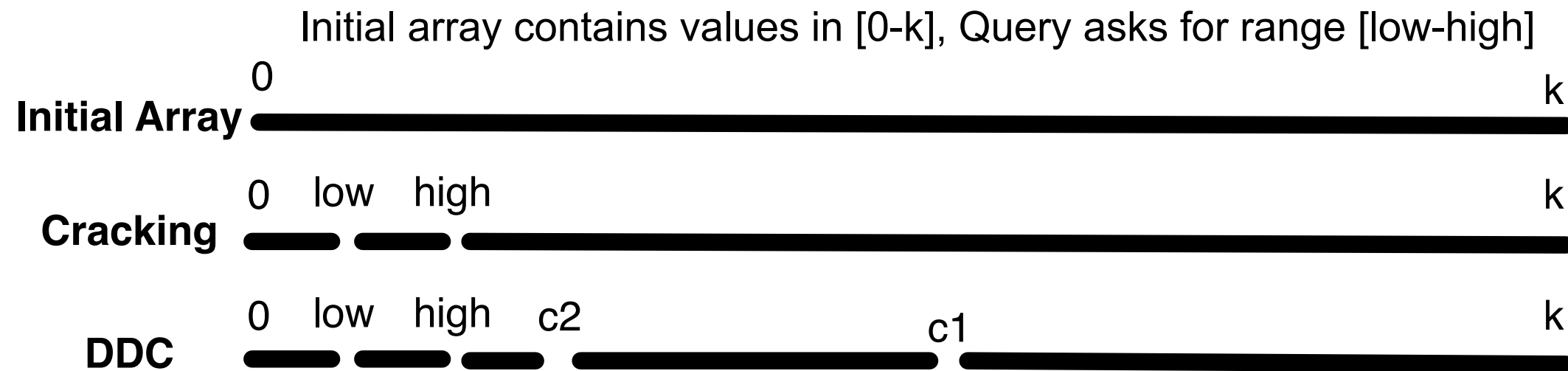
Query driven and data driven adaptation

# Stochastic Cracking

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



# Stochastic Cracking

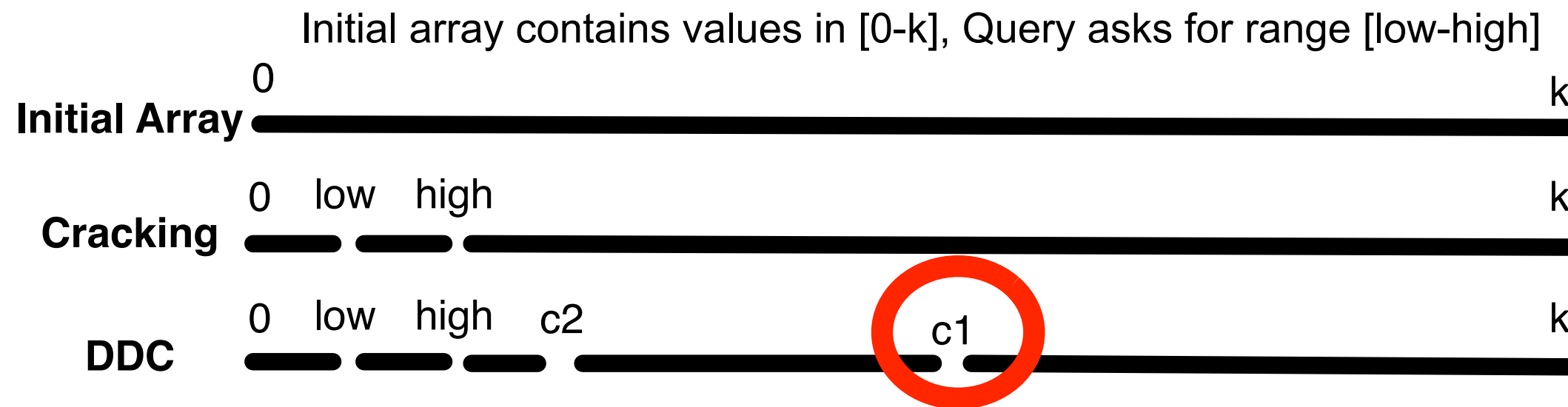


## Data Driven, Center (DDC):

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



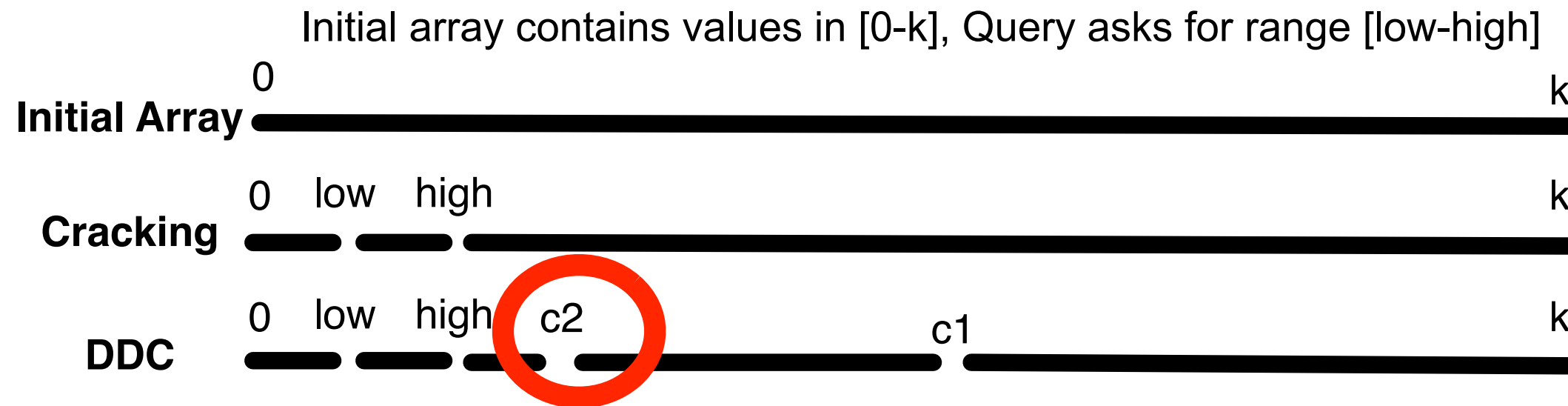
# Stochastic Cracking



## Data Driven, Center (DDC):

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

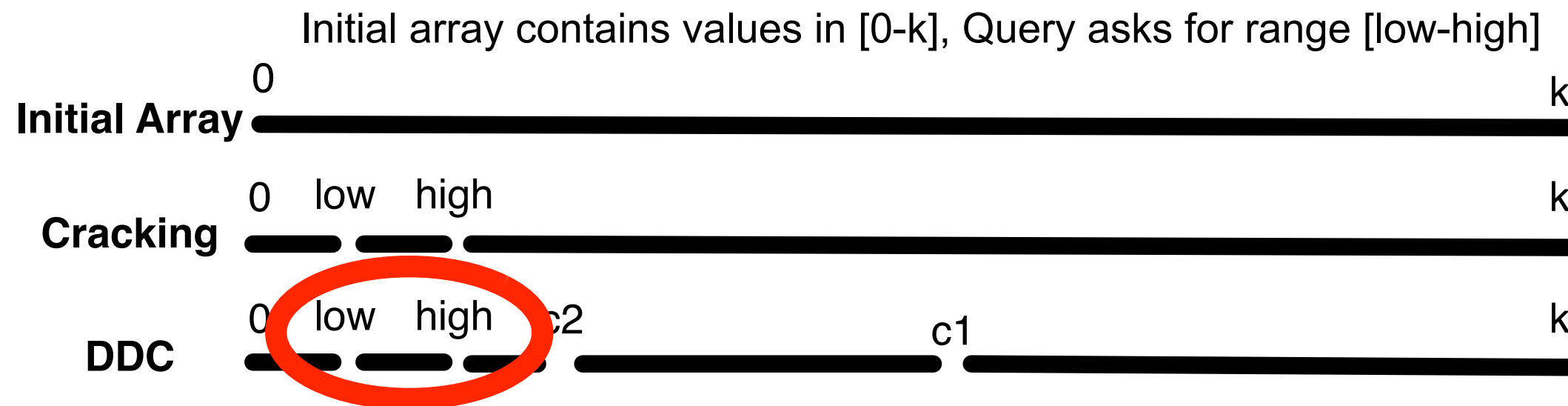
# Stochastic Cracking



## Data Driven, Center (DDC):

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

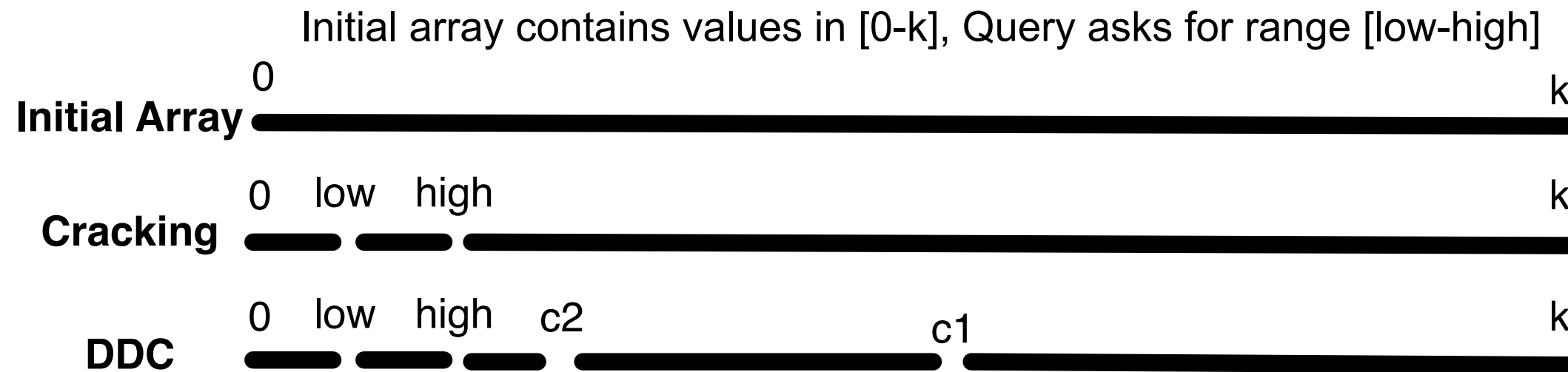
# Stochastic Cracking



## 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):

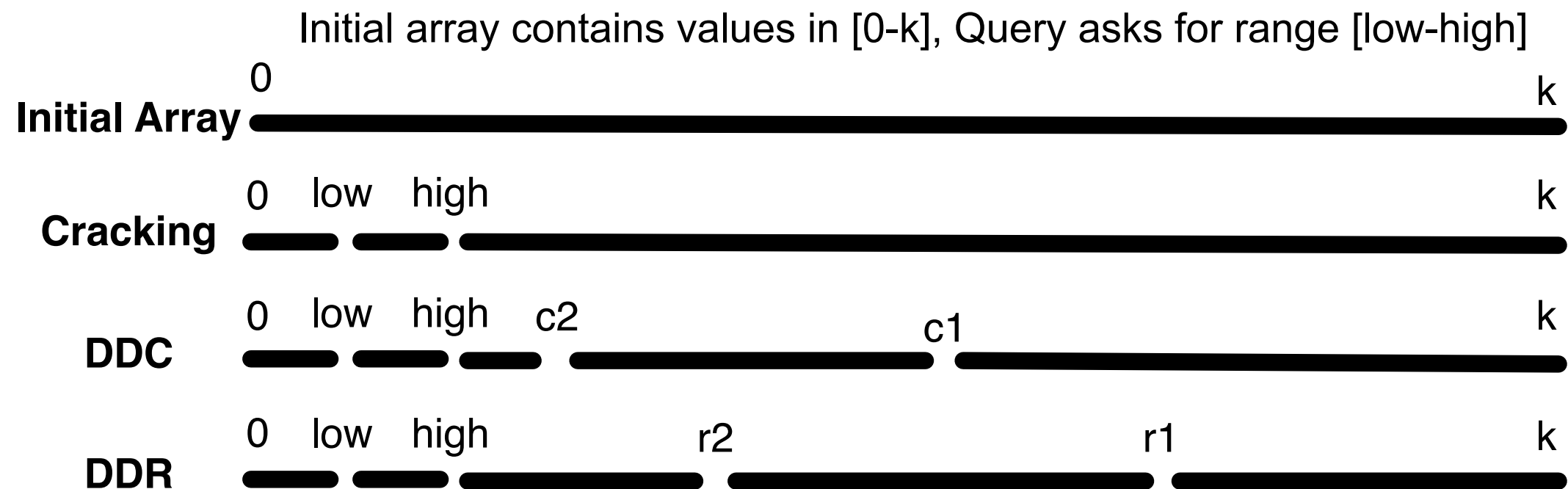
1. Recursively crack a piece in exactly half until in L2 cache.
2. Then crack for the query bounds.

# Stochastic Cracking

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



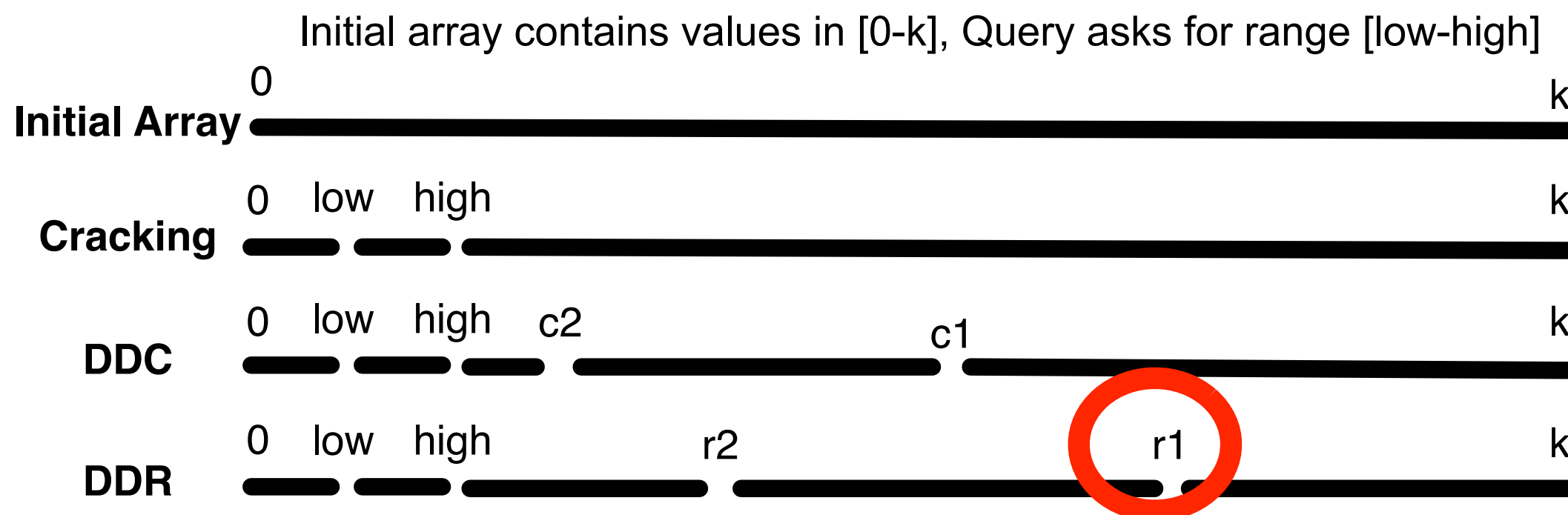
# Stochastic Cracking



## Data Driven, Random (DDR):

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

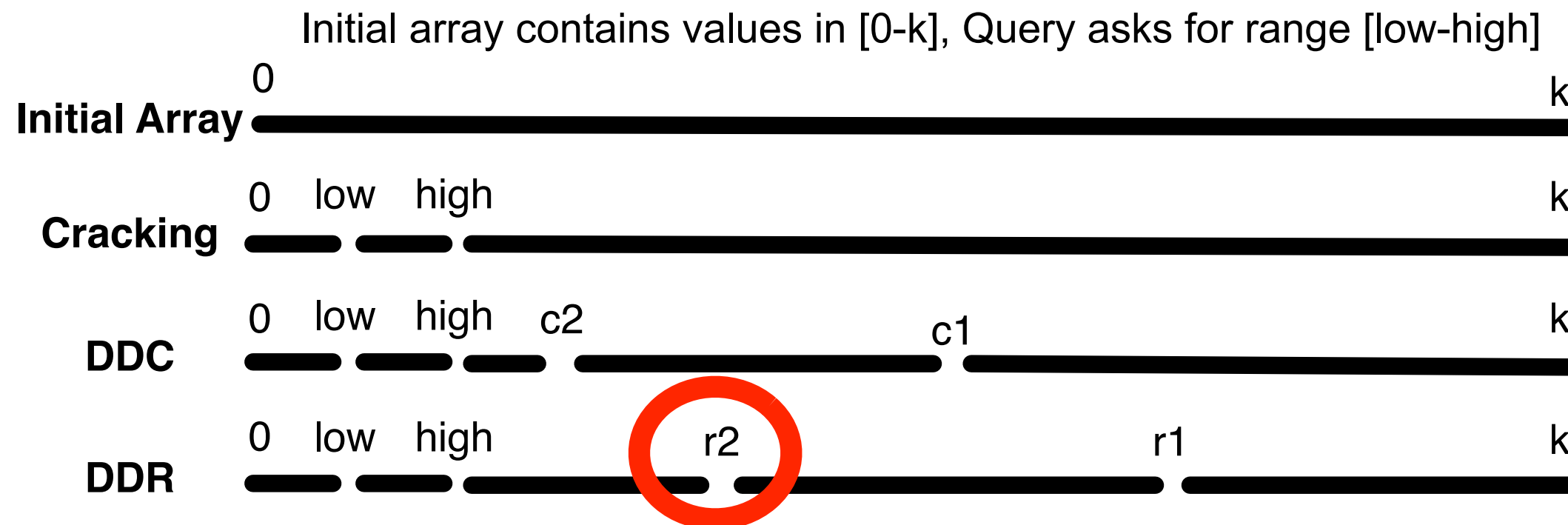
# Stochastic Cracking



## Data Driven, Random (DDR):

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

# Stochastic Cracking

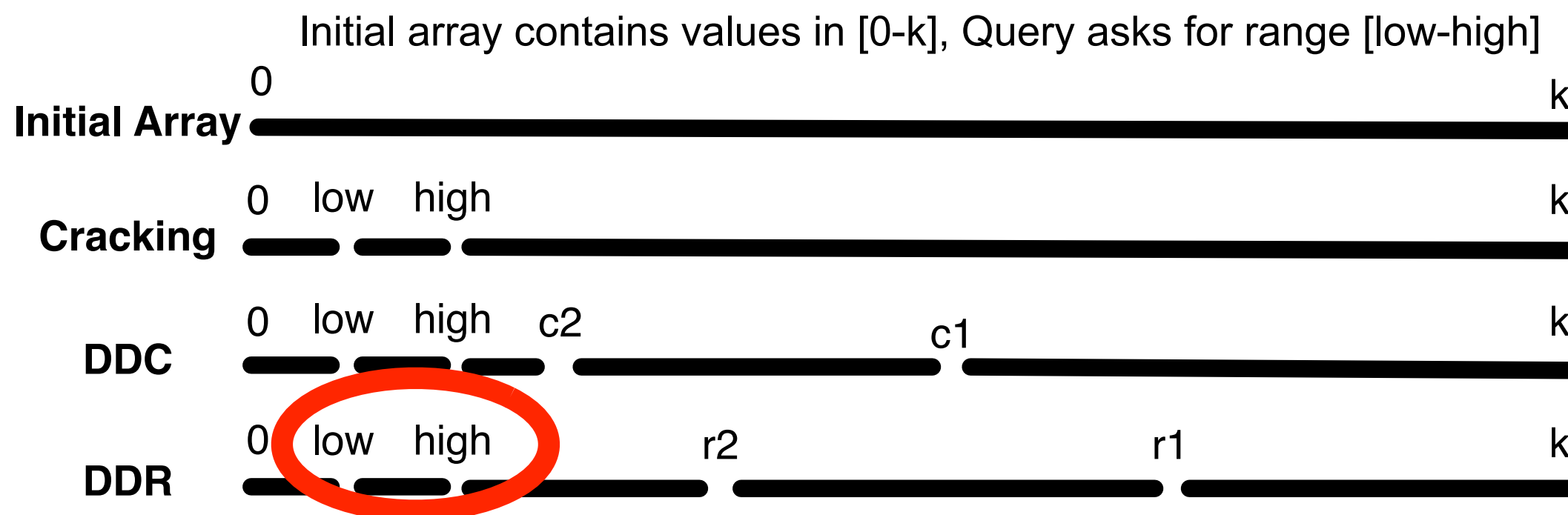


## Data Driven, Random (DDR):

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



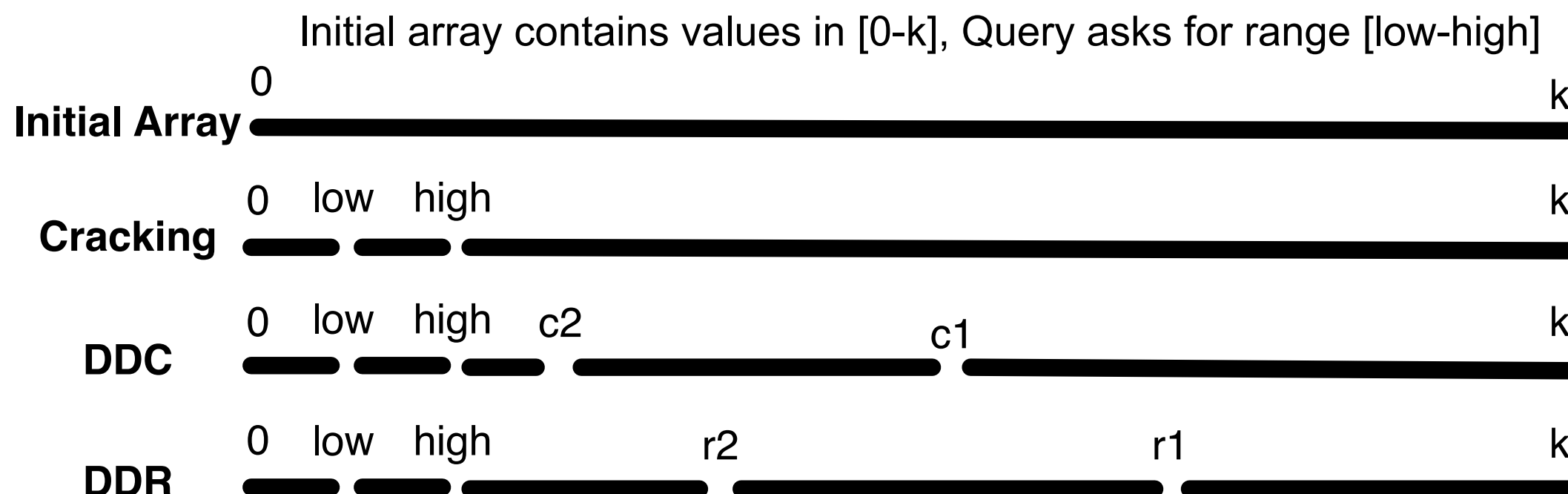
# Stochastic Cracking



## Data Driven, Random (DDR):

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

# Stochastic Cracking

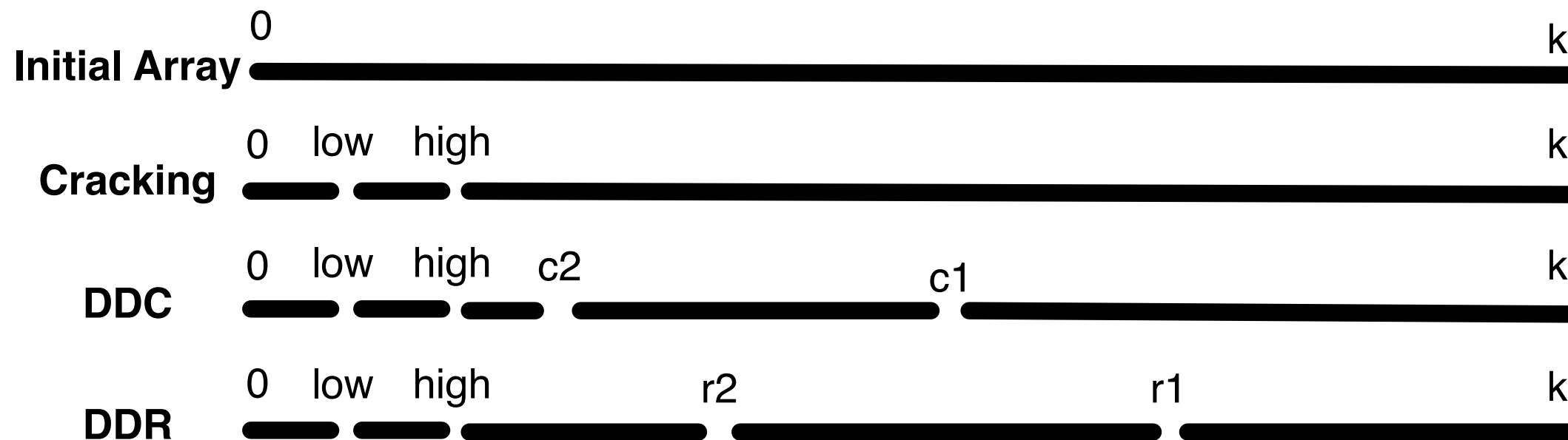


## Data Driven, Random (DDR):

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

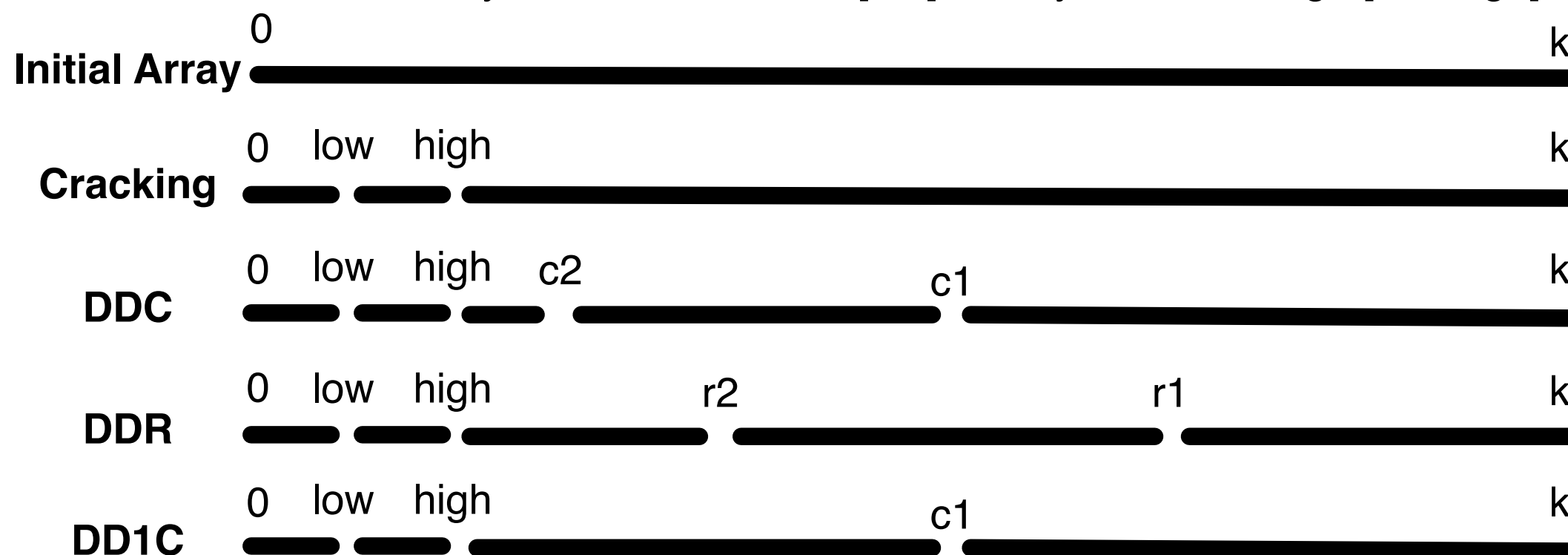
# Stochastic Cracking

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



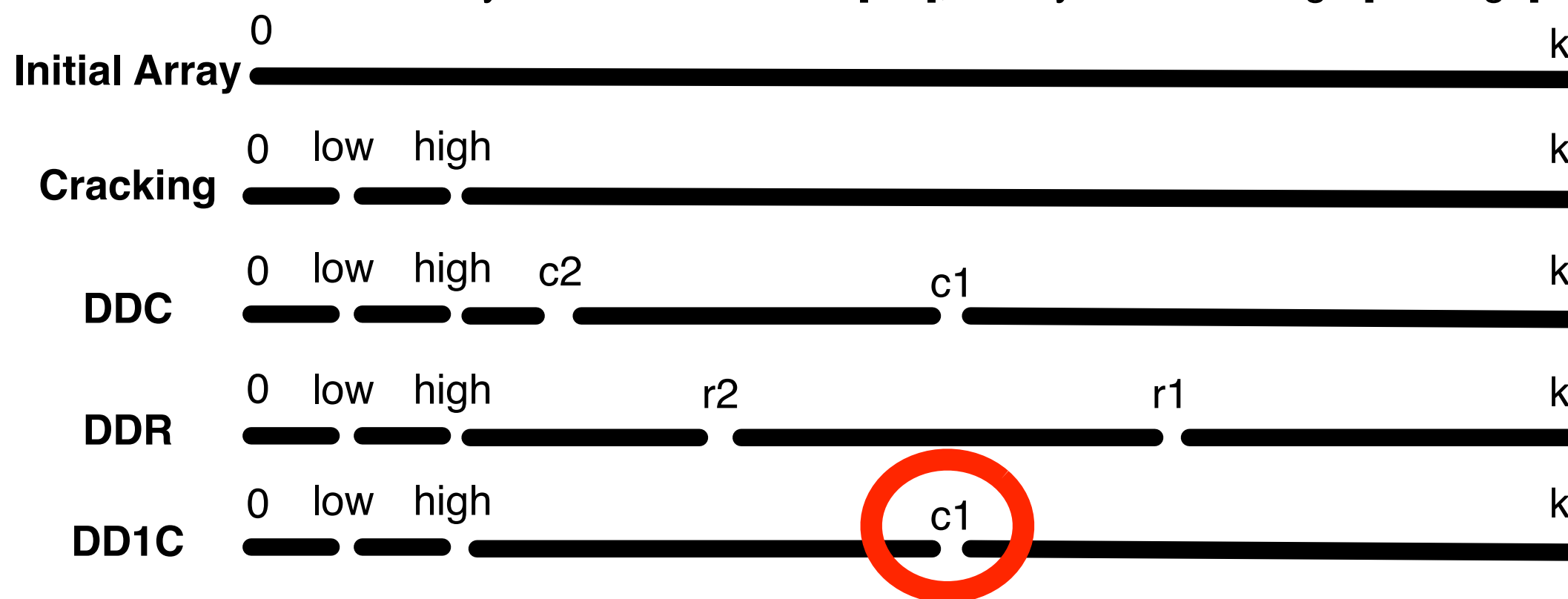
# Stochastic Cracking

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



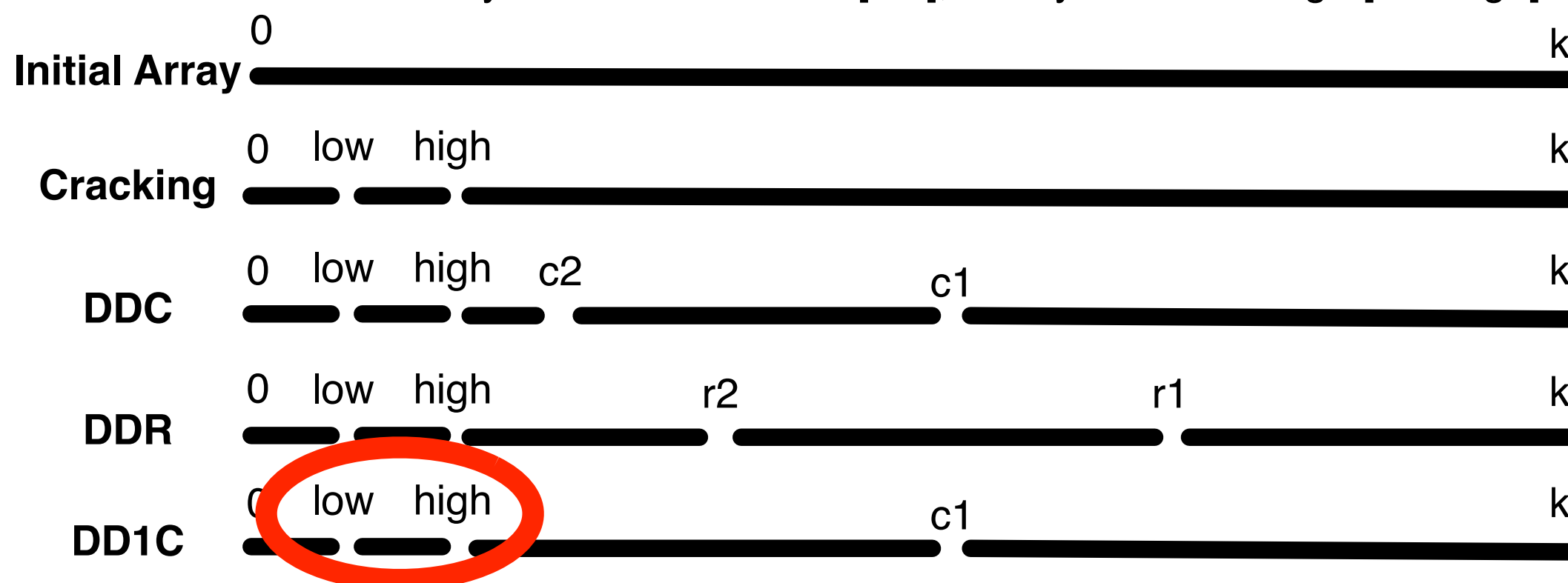
# Stochastic Cracking

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



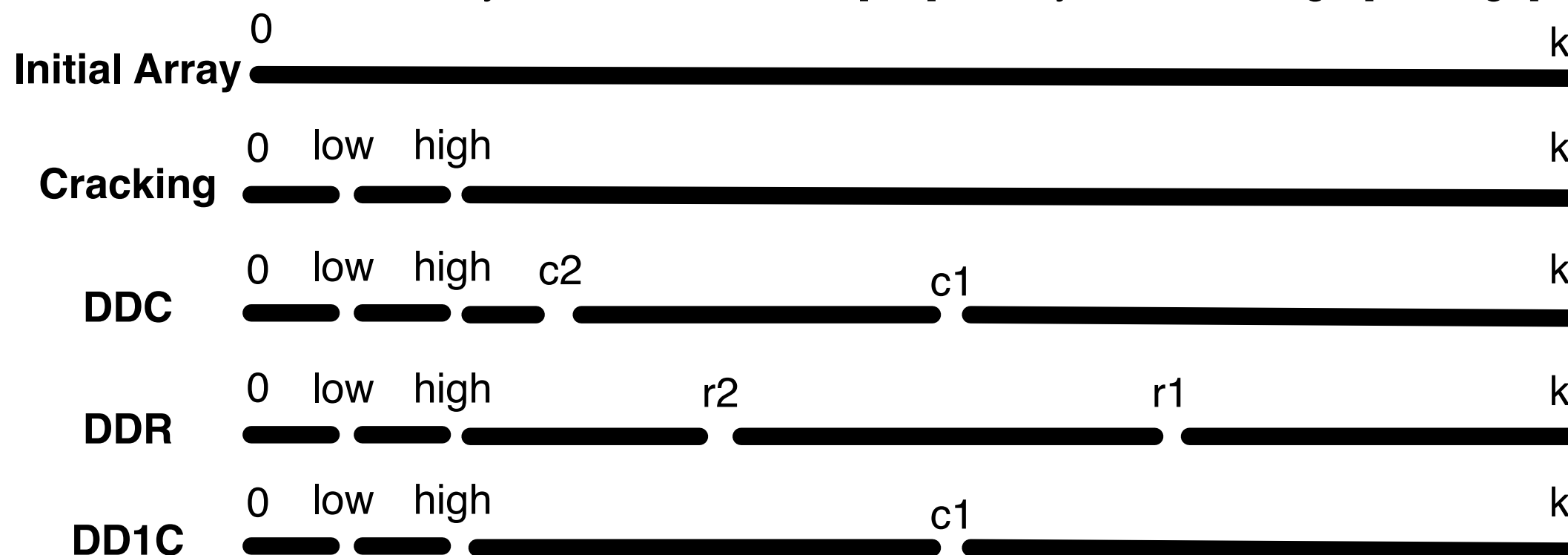
# Stochastic Cracking

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



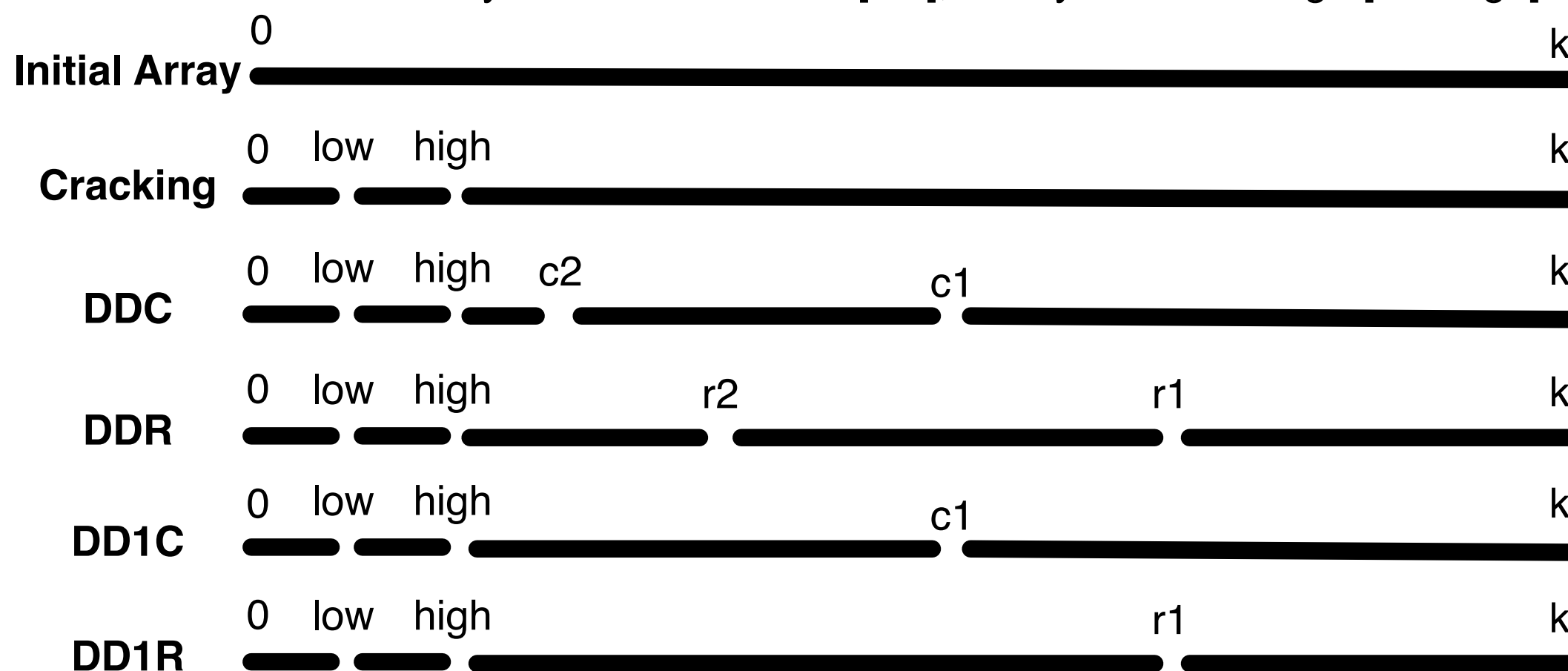
# Stochastic Cracking

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



# Stochastic Cracking

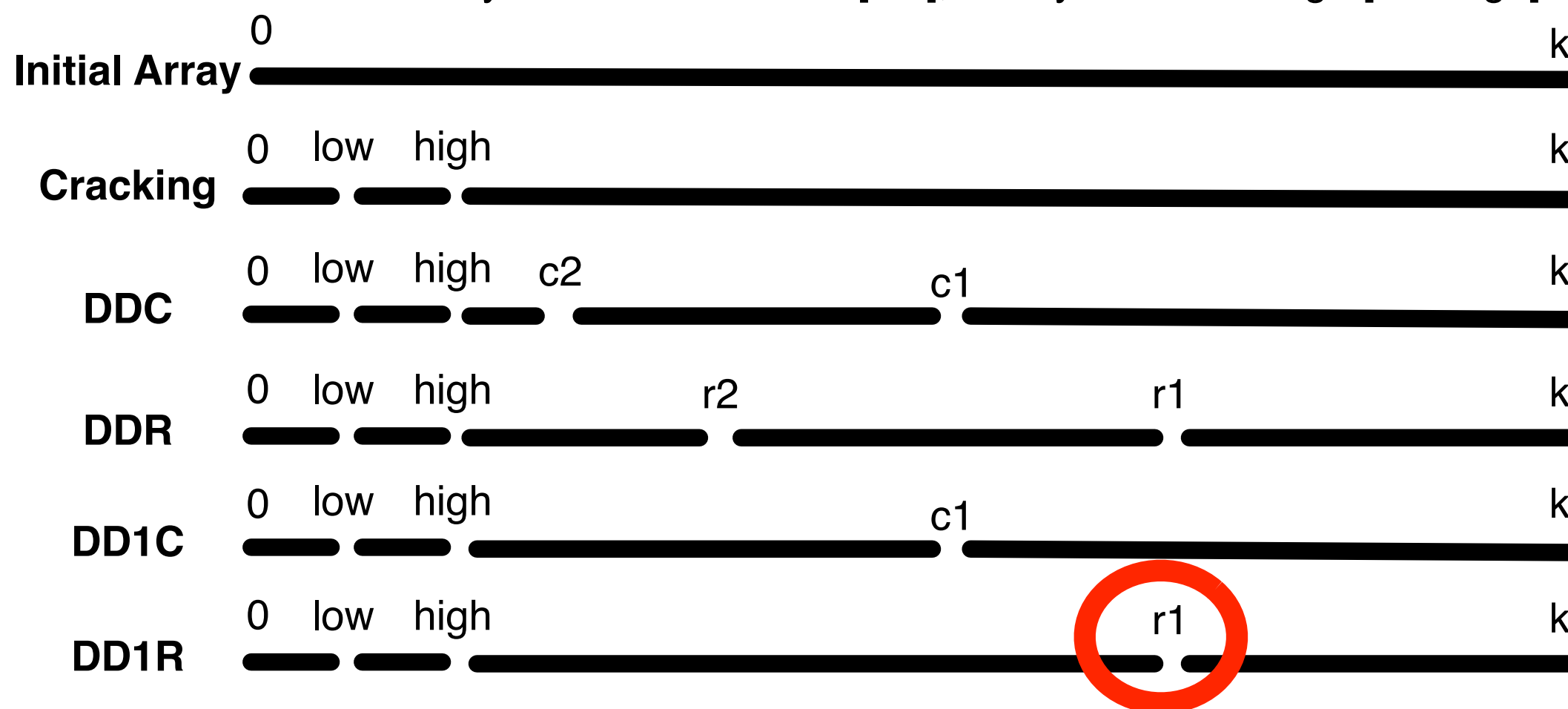
Initial array contains values in  $[0-k]$ , Query asks for range  $[low-high]$





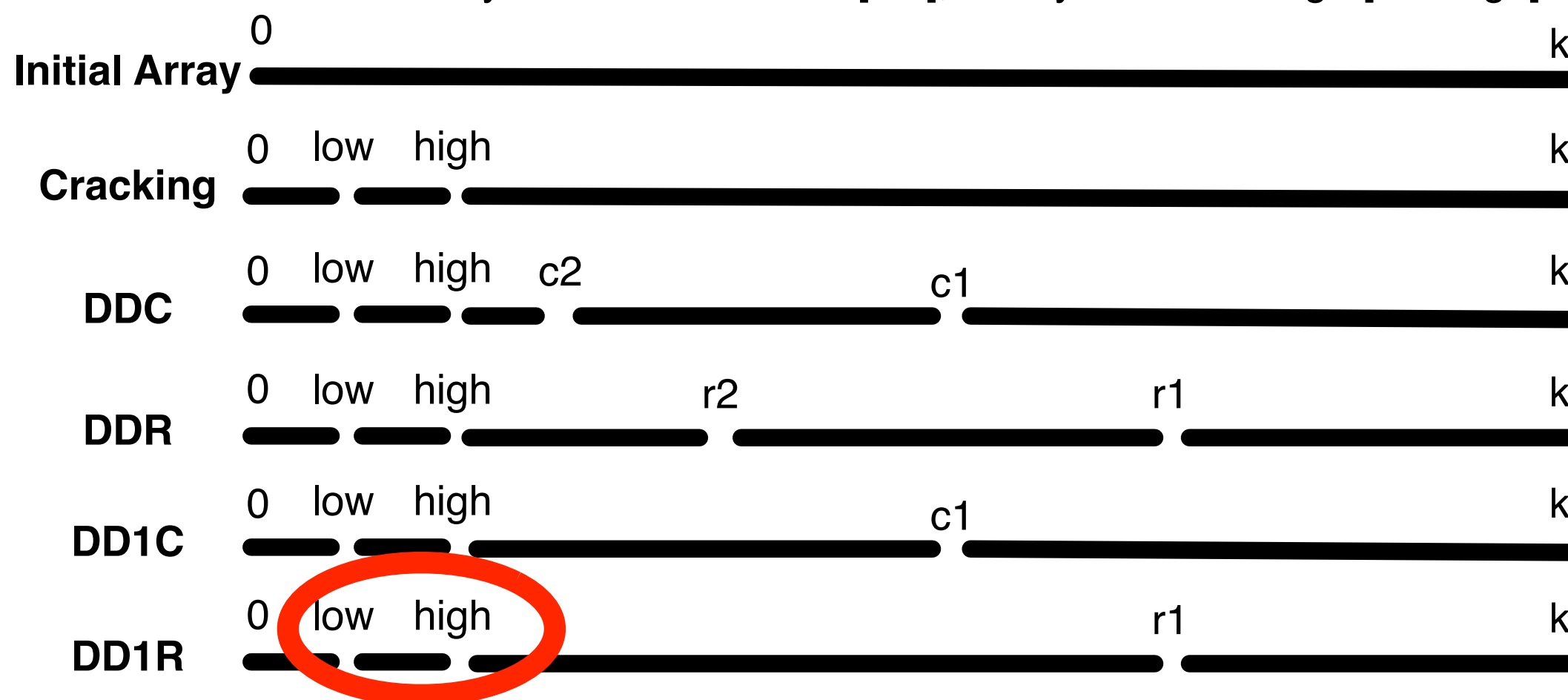
# Stochastic Cracking

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



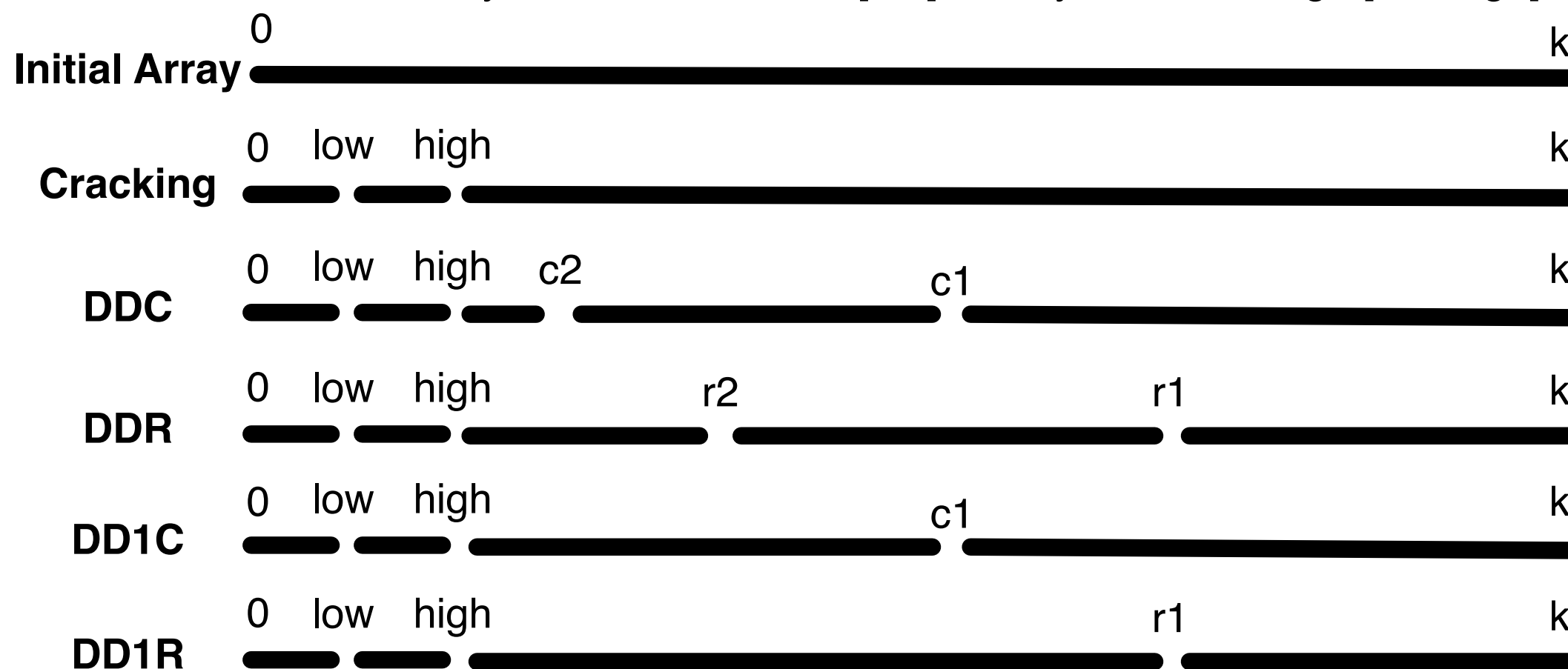
# Stochastic Cracking

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

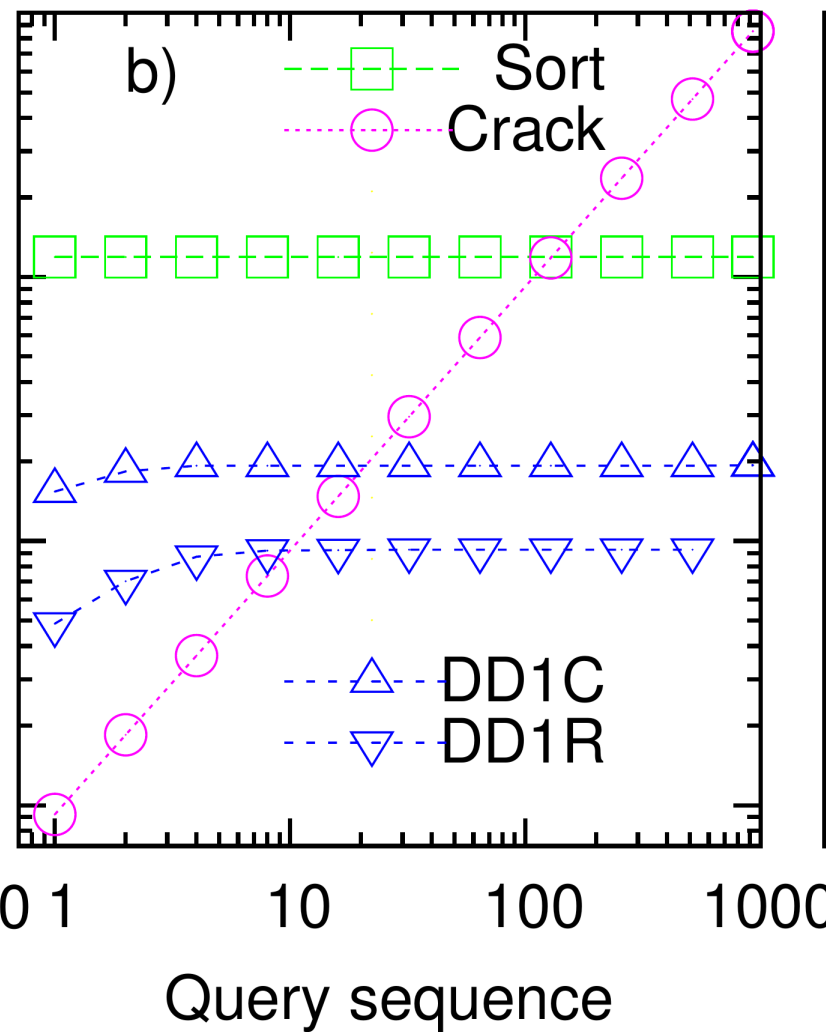
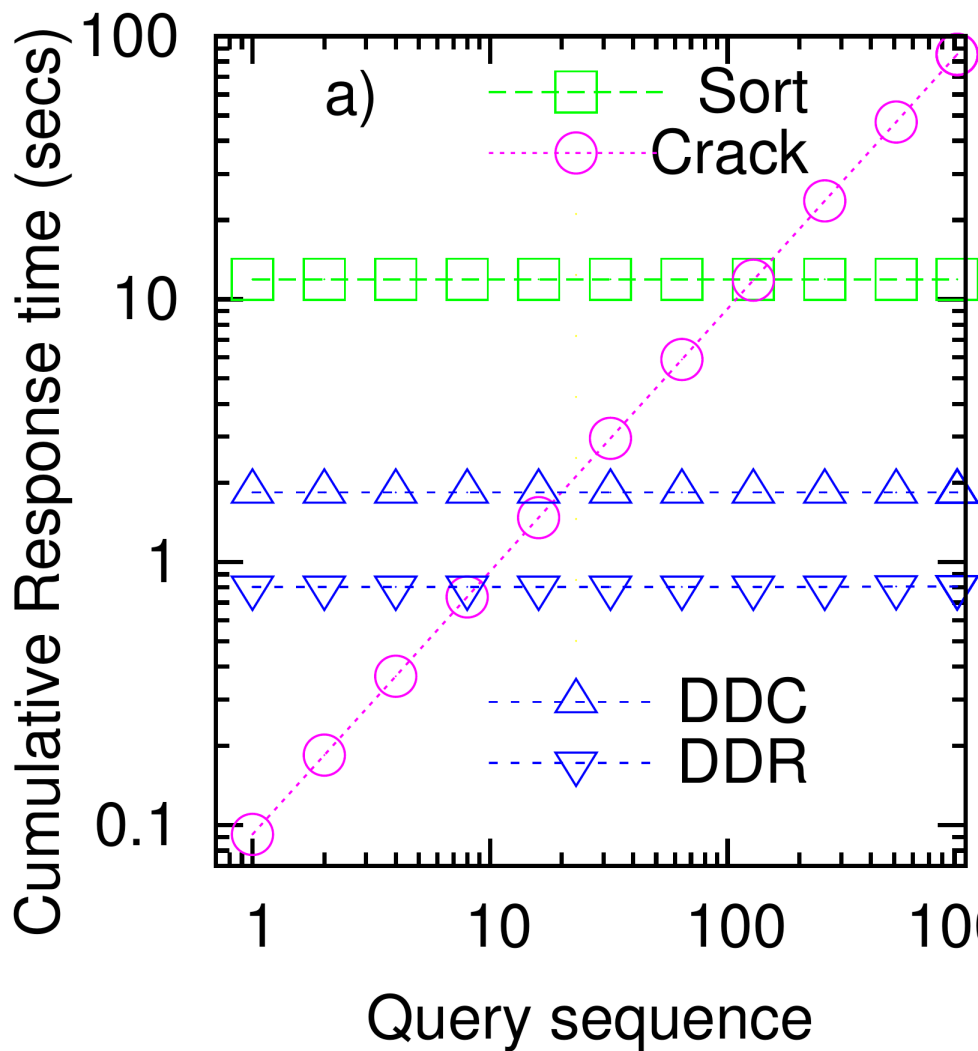


# Stochastic Cracking

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



## 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

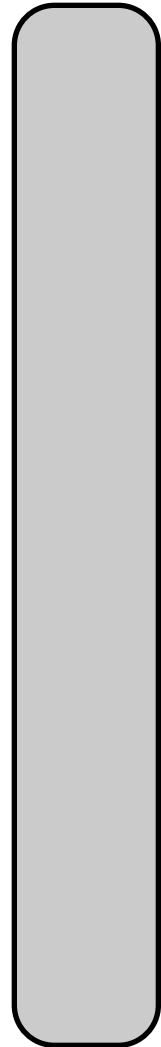
**EDBT'10, SMDB'10, Goetz Graefe and Harumi Kuno**

*Incremental sort via external merge sort steps*

# Adaptive Merging

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

*Incremental sort via external merge sort steps*

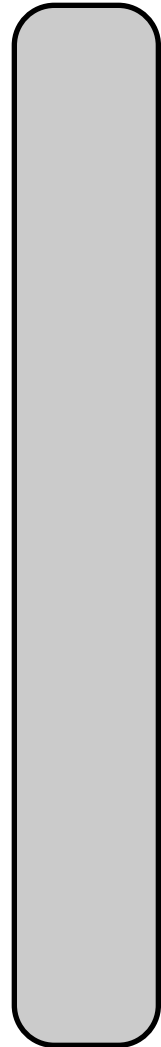


# Adaptive Merging

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

*Incremental sort via external merge sort steps*

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





# Adaptive Merging

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

*Incremental sort via external merge sort steps*

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



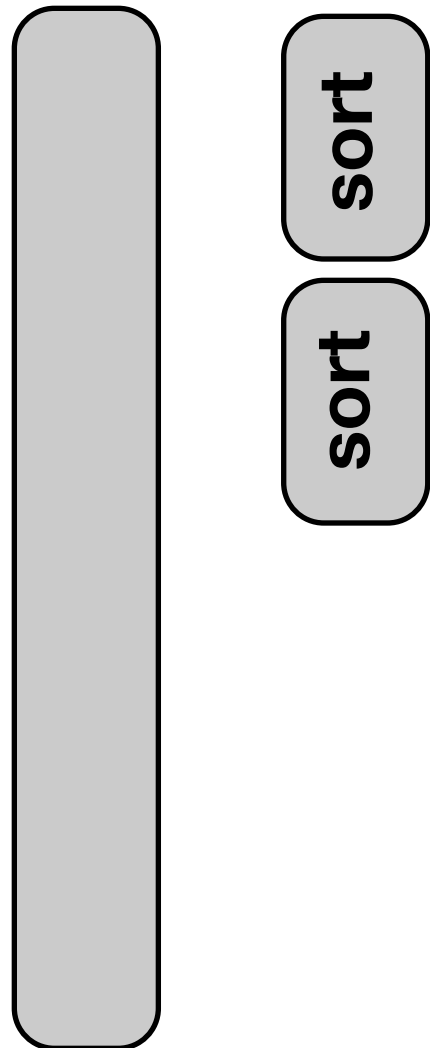
sort

# Adaptive Merging

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

*Incremental sort via external merge sort steps*

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

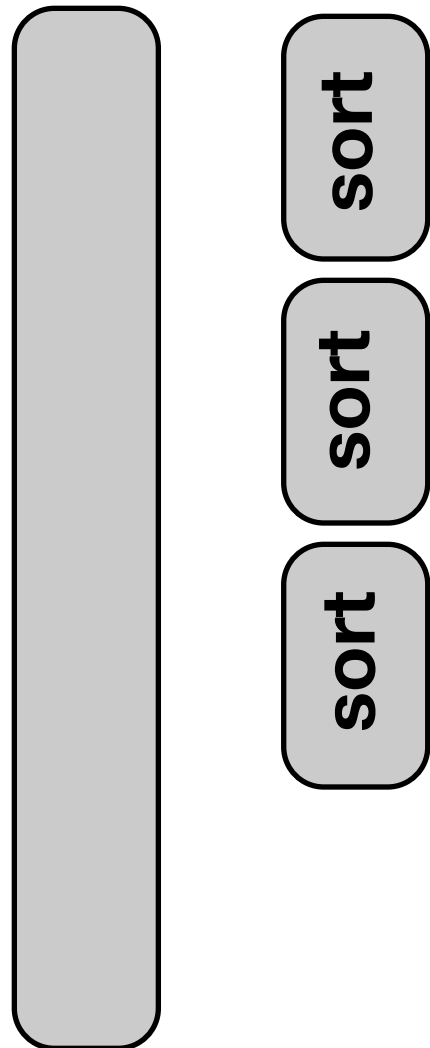


# Adaptive Merging

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

*Incremental sort via external merge sort steps*

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

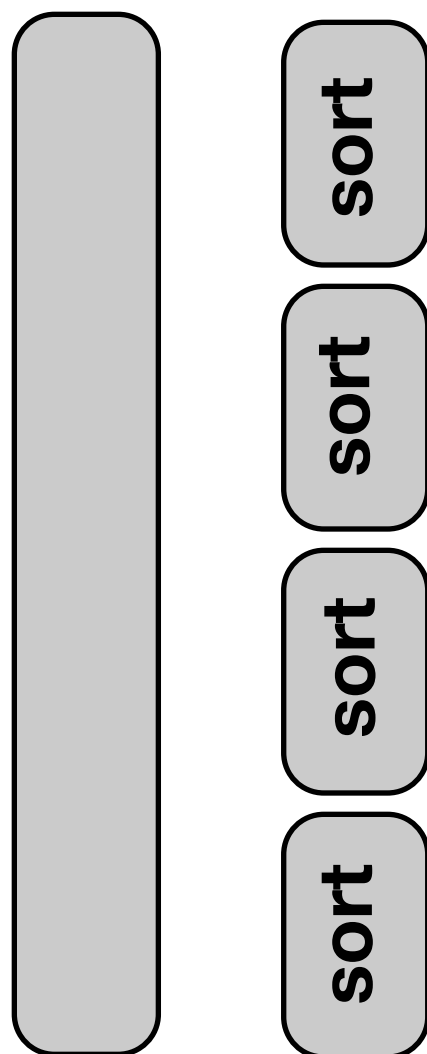


# Adaptive Merging

**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)
```



sort

sort

sort

sort

binary  
search

# Adaptive Merging

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

*Incremental sort via external merge sort steps*

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



sort

binary  
search

sort

binary  
search

sort

sort

# Adaptive Merging

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

*Incremental sort via external merge sort steps*

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



sort

binary  
search

sort

binary  
search

sort

binary  
search

sort

# Adaptive Merging

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

*Incremental sort via external merge sort steps*

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



sort

binary  
search

sort

binary  
search

sort

binary  
search

sort

binary  
search

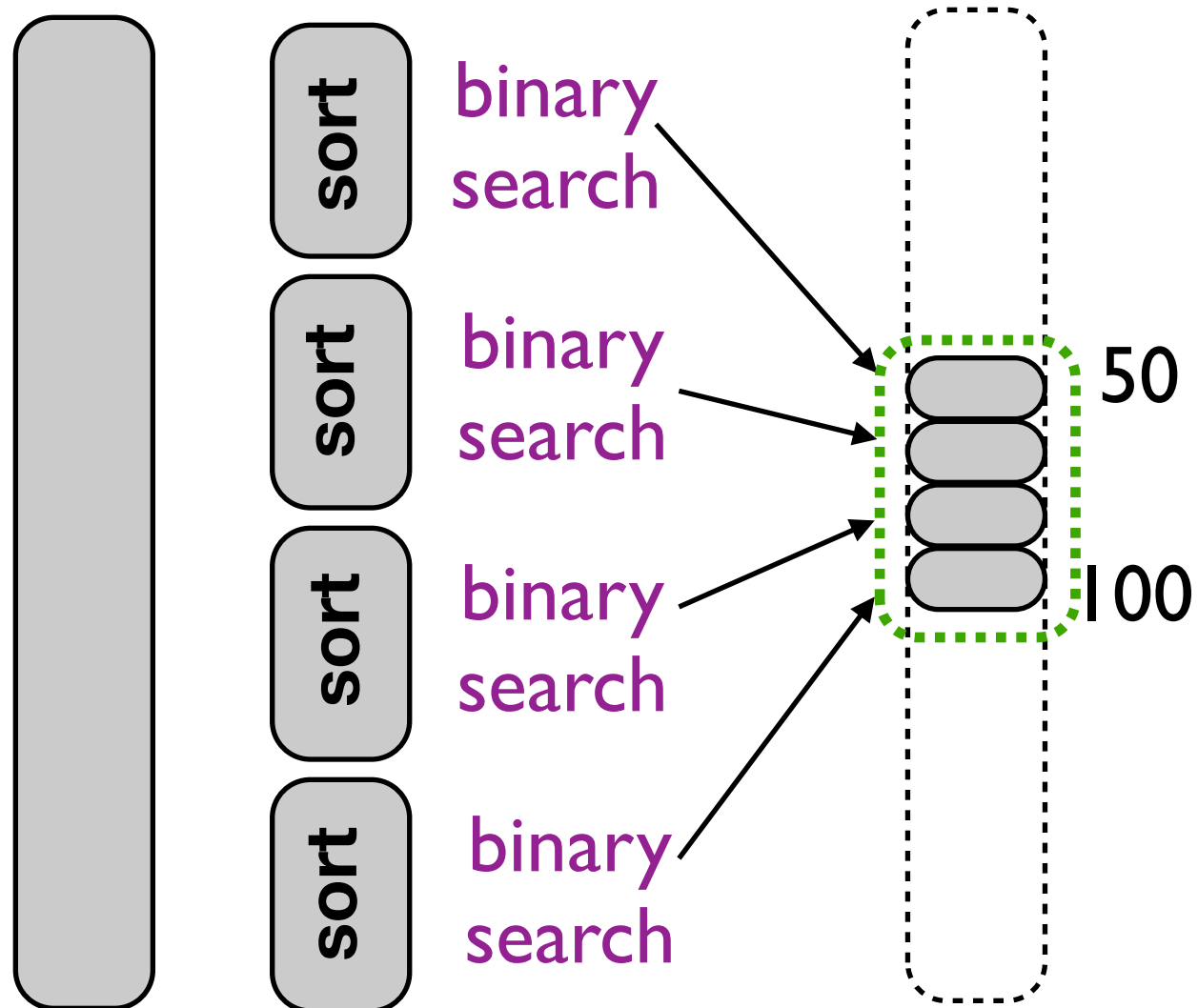


# Adaptive Merging

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

*Incremental sort via external merge sort steps*

select(A,50,100)

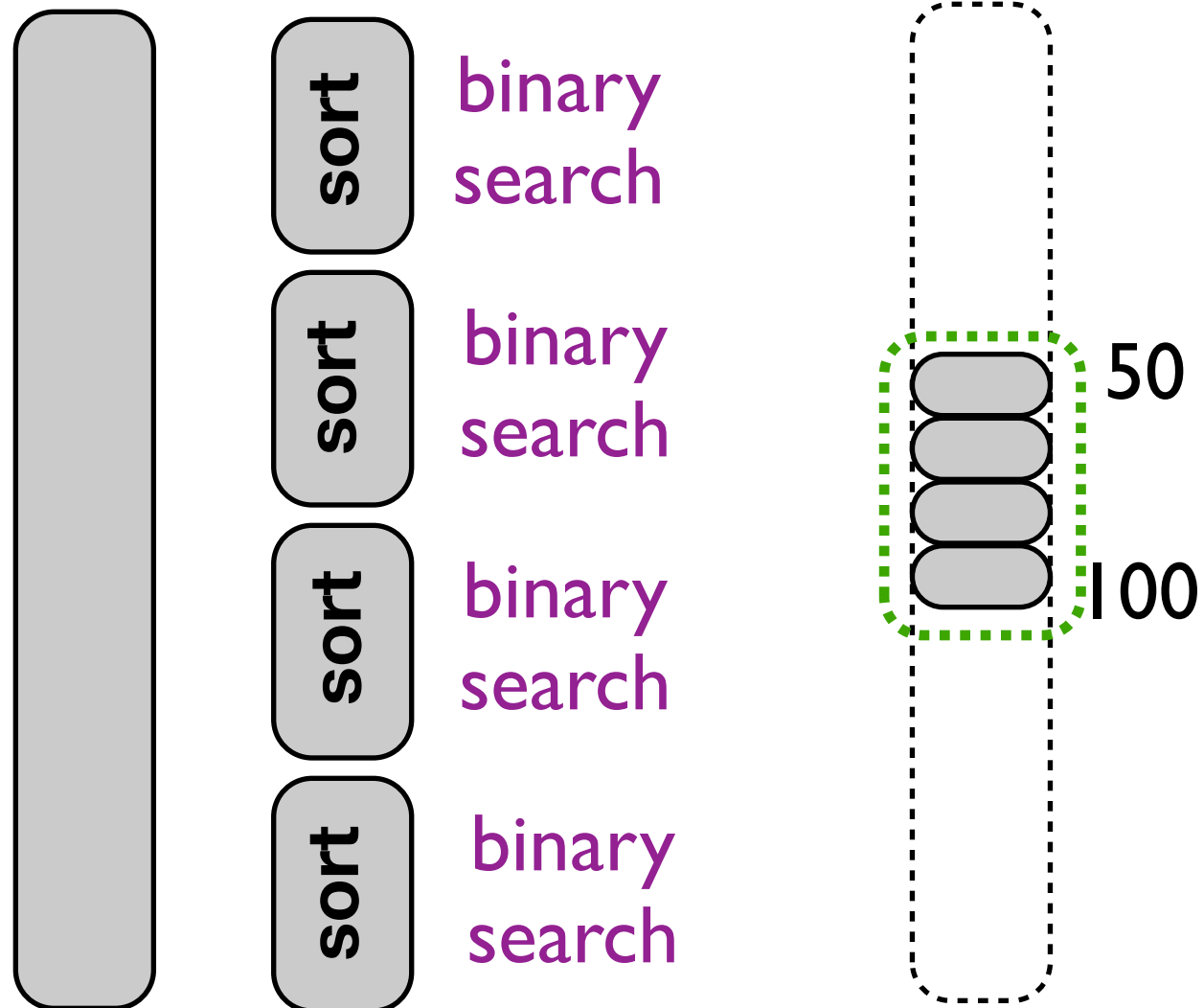


# Adaptive Merging

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

*Incremental sort via external merge sort steps*

select(A,50,100)

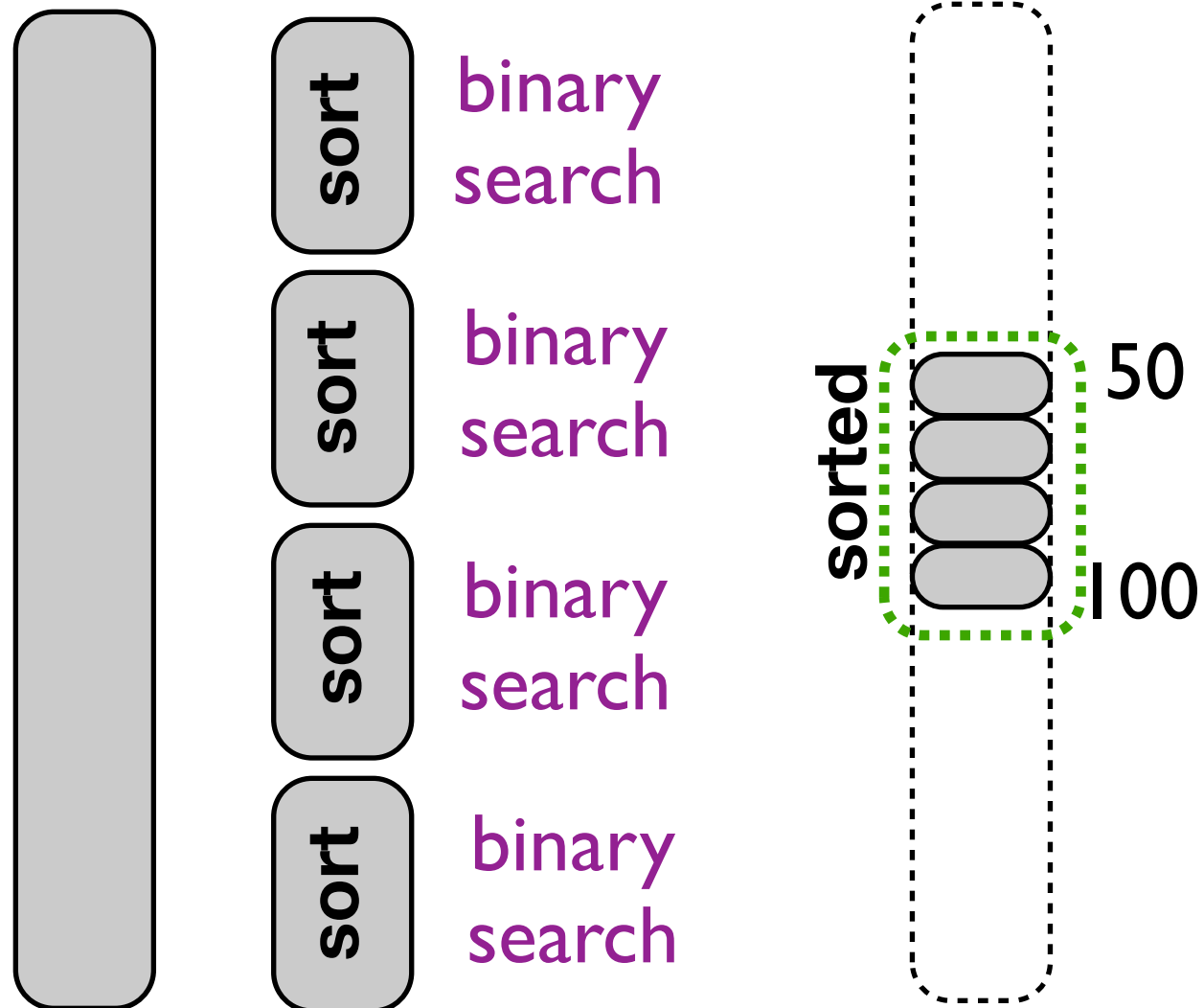


# Adaptive Merging

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

*Incremental sort via external merge sort steps*

select(A,50,100)

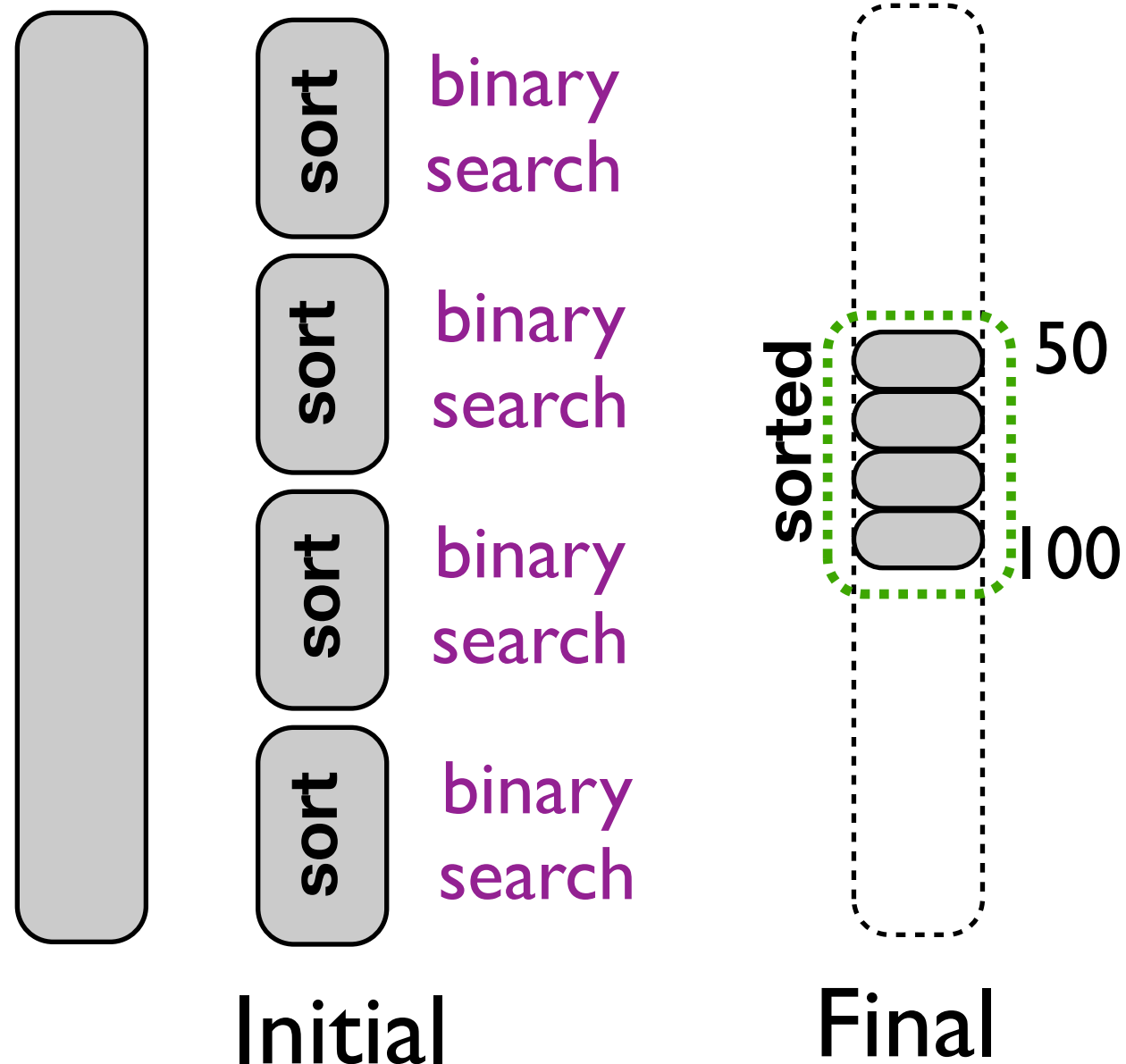


# Adaptive Merging

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

*Incremental sort via external merge sort steps*

select(A,50,100)

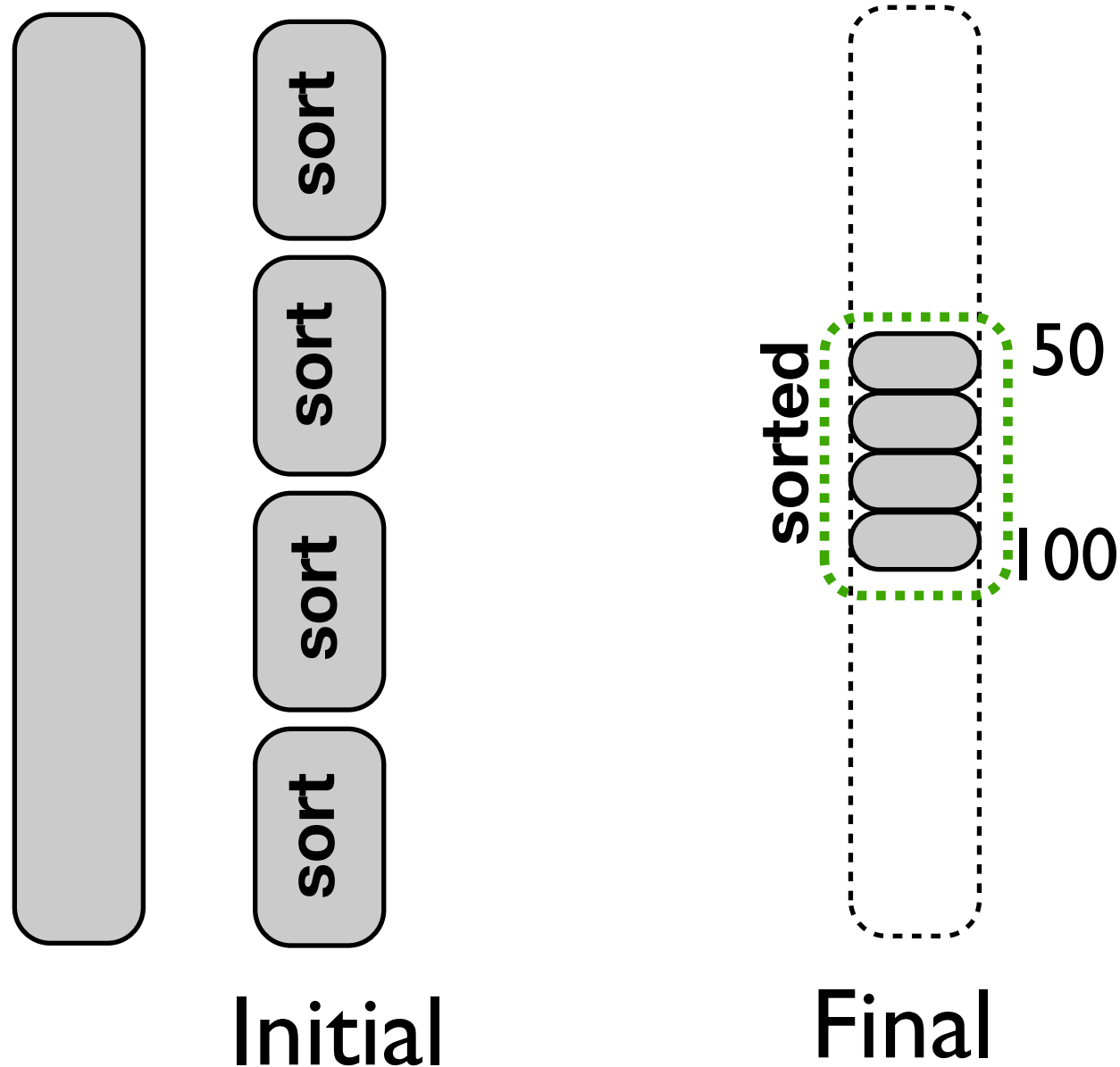


# Adaptive Merging

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

*Incremental sort via external merge sort steps*

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

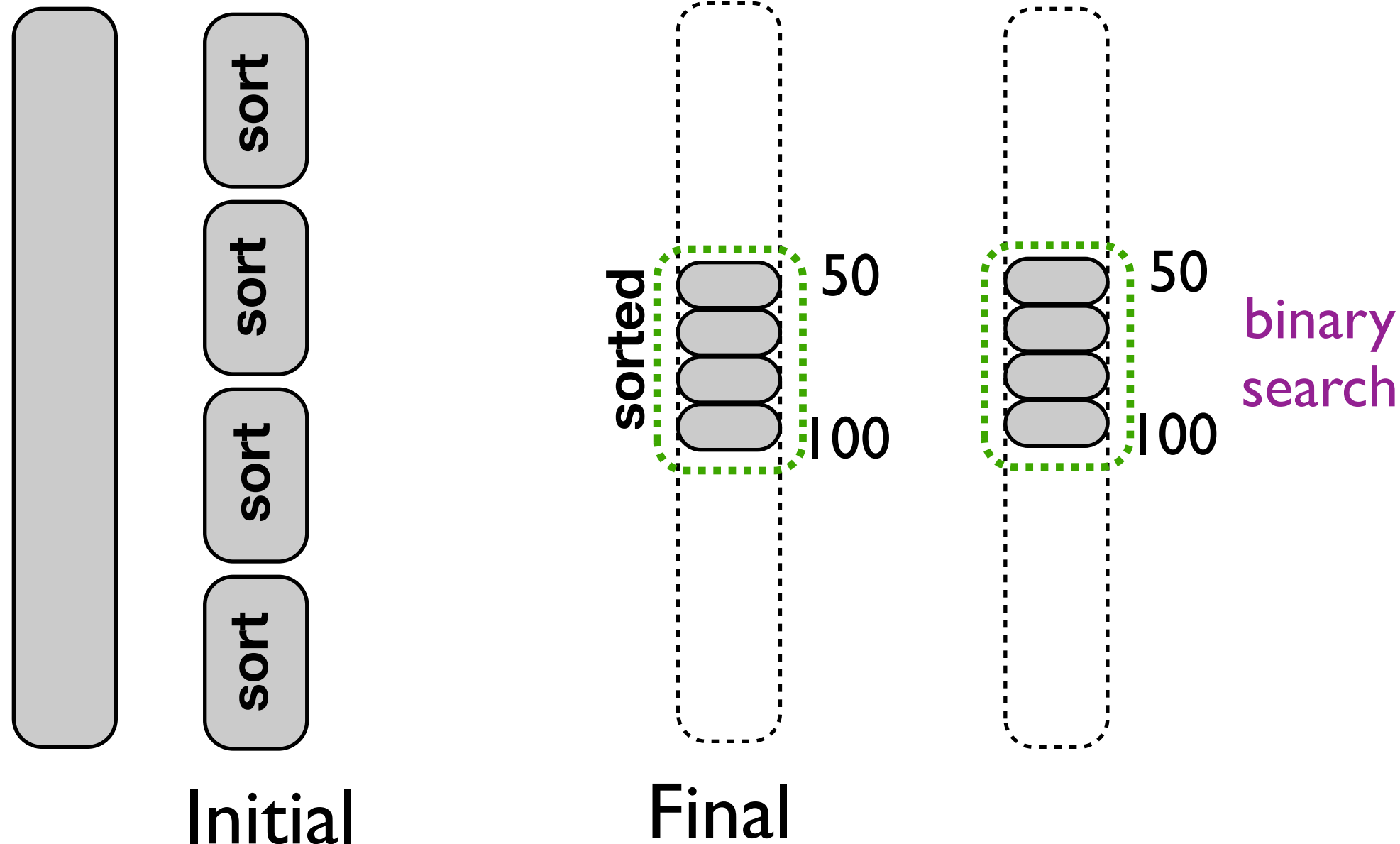


# Adaptive Merging

**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

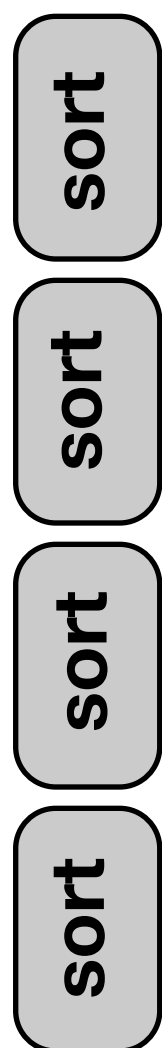
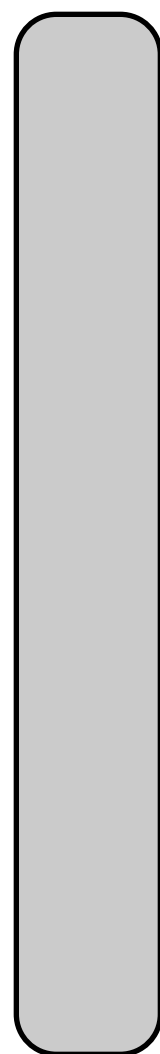
**EDBT'10, SMDB'10**, Goetz Graefe and Harumi Kuno

*Incremental sort via external merge sort steps*

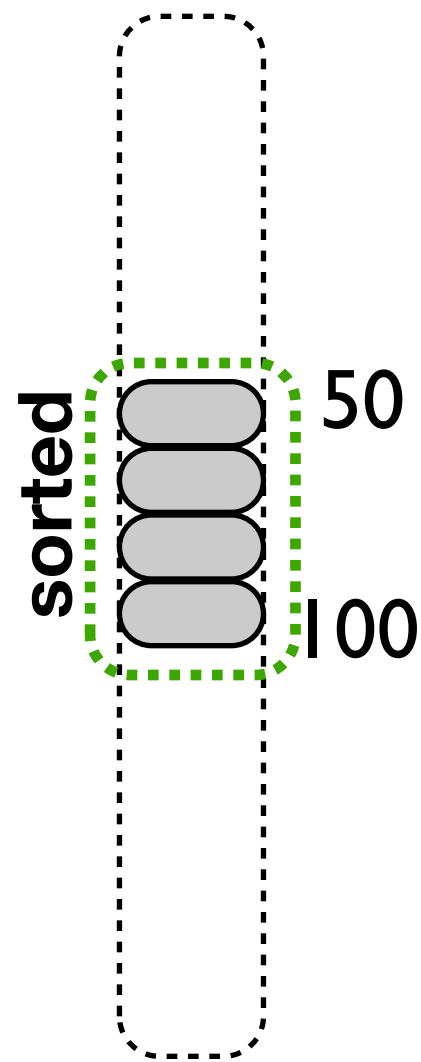
select(A,50,100)

select(A,55,70)

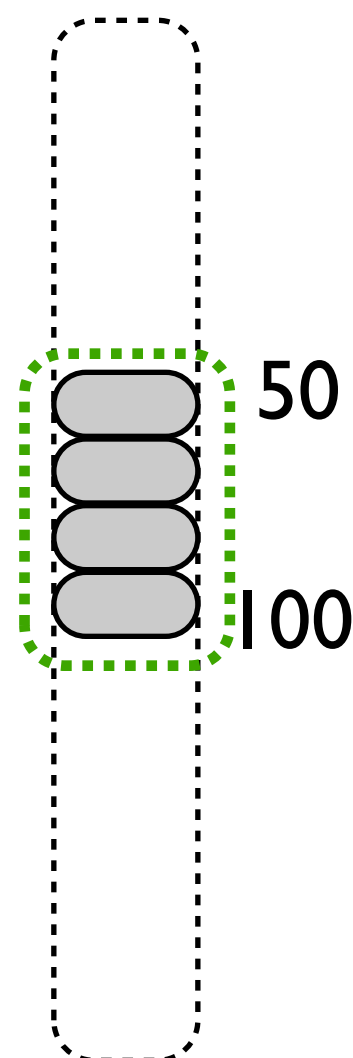
select(A,150,170)



Initial



Final



# Adaptive Merging

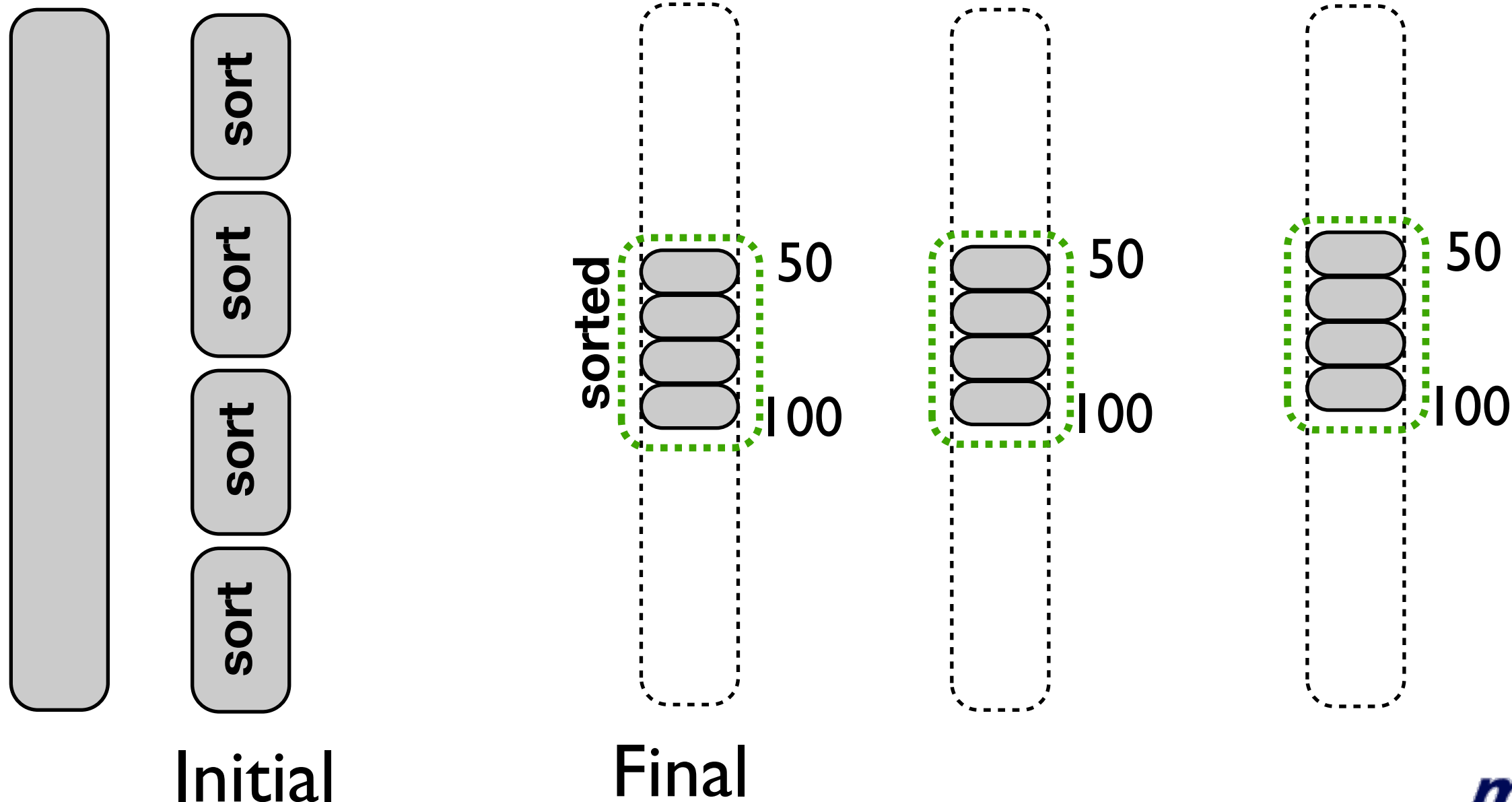
**EDBT'10, SMDB'10**, Goetz Graefe and Harumi Kuno

*Incremental sort via external merge sort steps*

`select(A,50,100)`

`select(A,55,70)`

`select(A,150,170)`





# Adaptive Merging

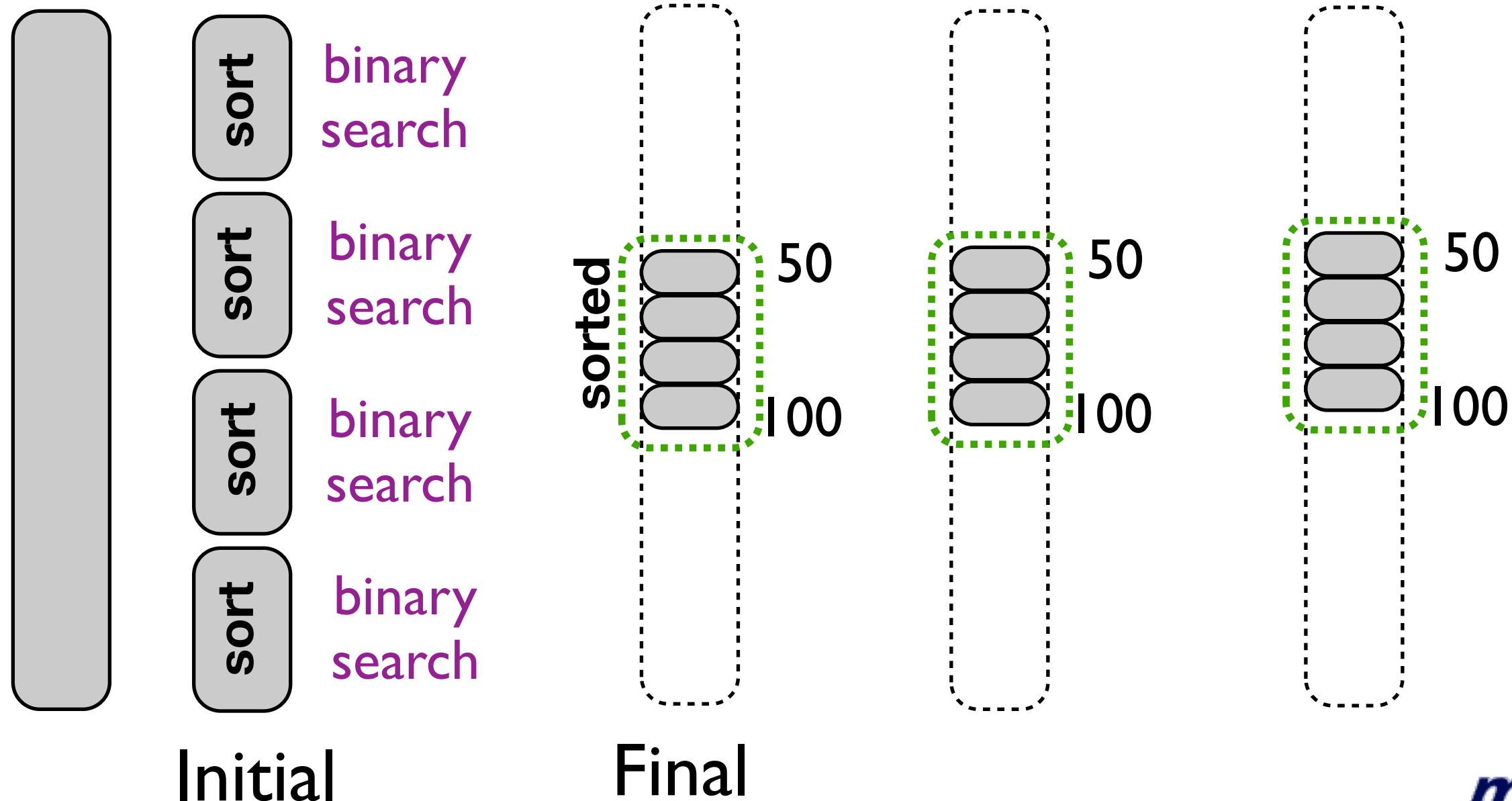
**EDBT'10, SMDB'10**, Goetz Graefe and Harumi Kuno

*Incremental sort via external merge sort steps*

select(A,50,100)

select(A,55,70)

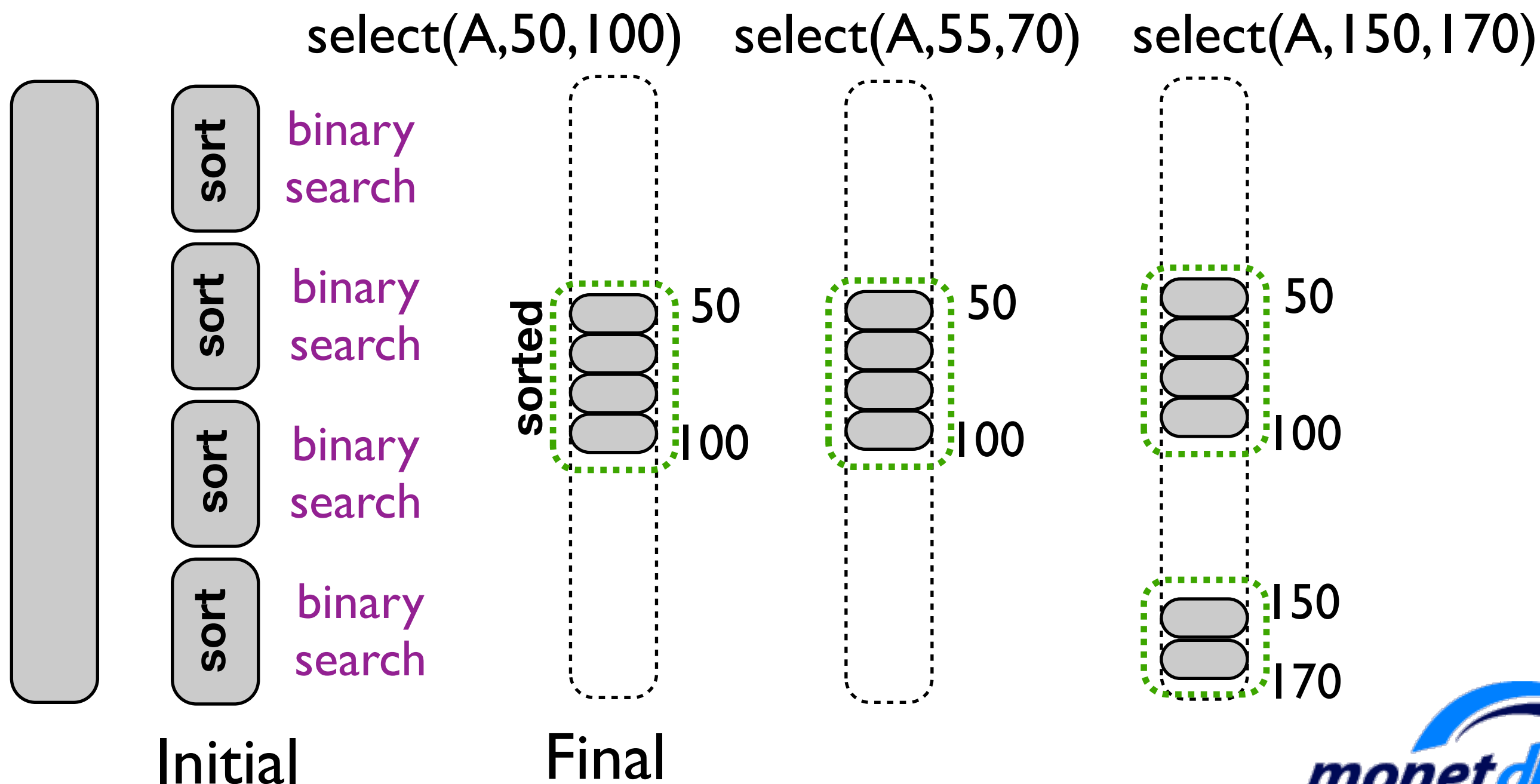
select(A,150,170)



# Adaptive Merging

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

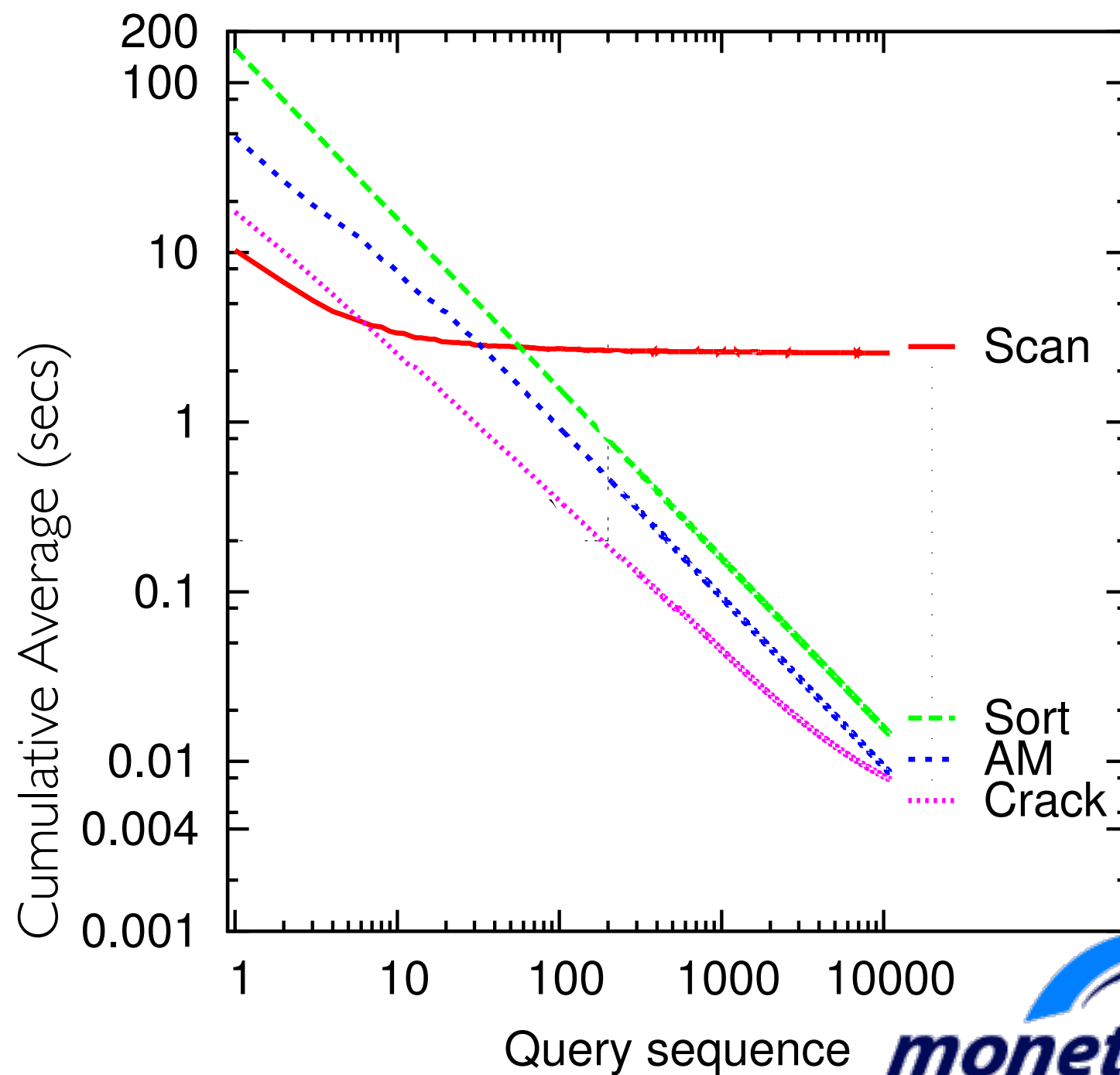
*Incremental sort via external merge sort steps*



# Performance Analysis

## set-up

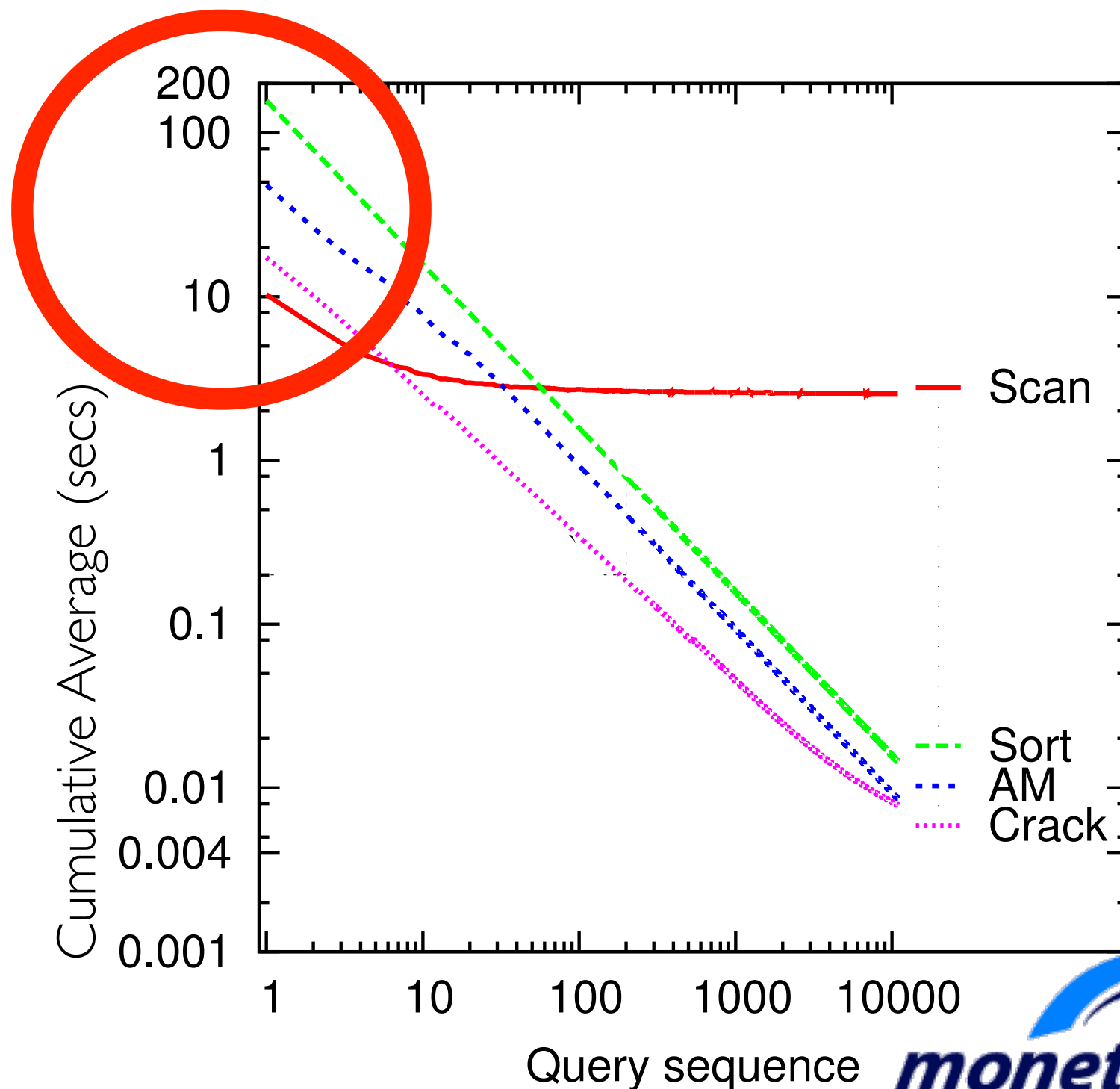
10K random selections  
selectivity 10%  
random value ranges  
in a 30 million integer column



# Performance Analysis

## set-up

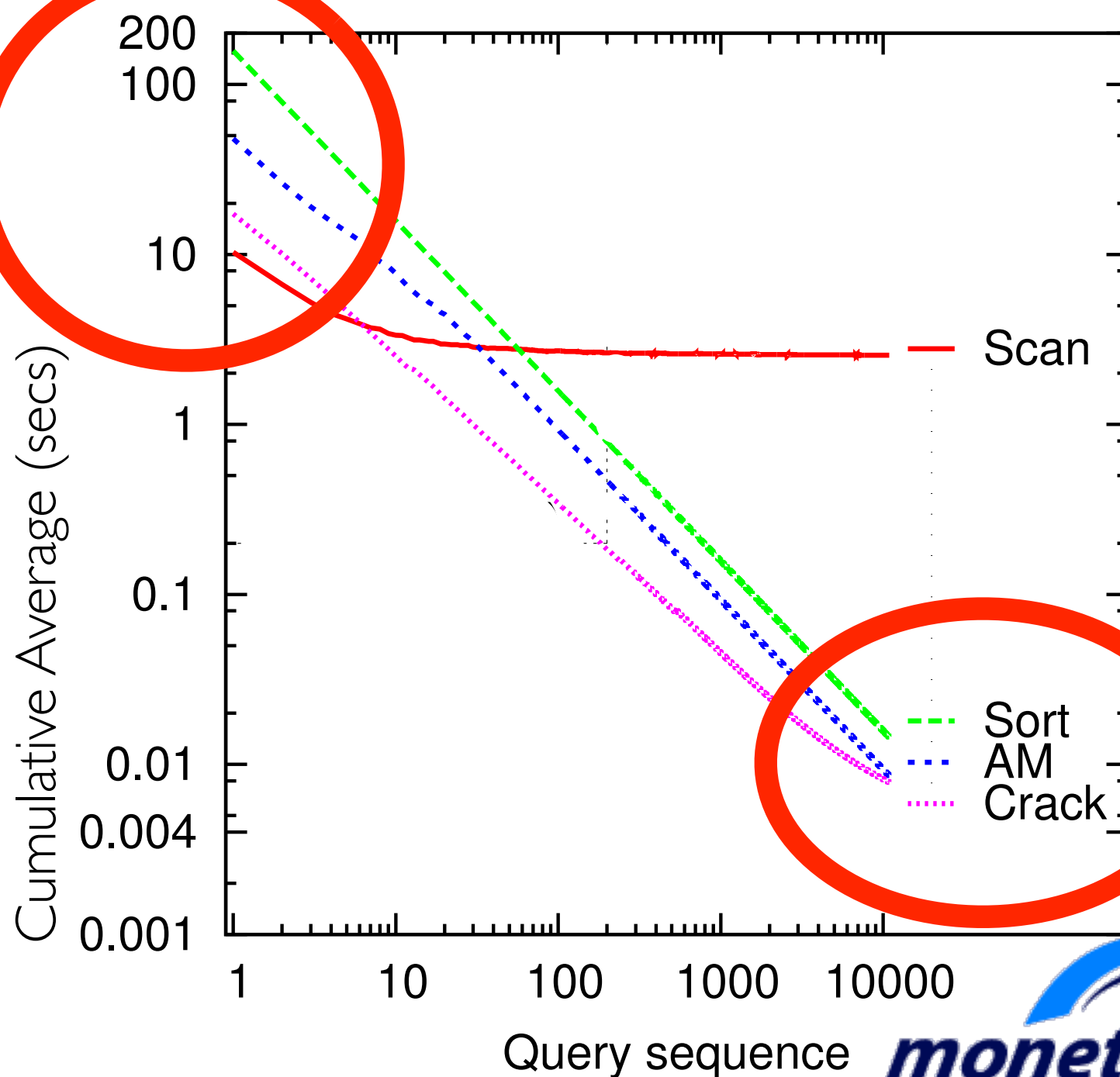
10K random selections  
selectivity 10%  
random value ranges  
in a 30 million integer column



# Performance Analysis

## set-up

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

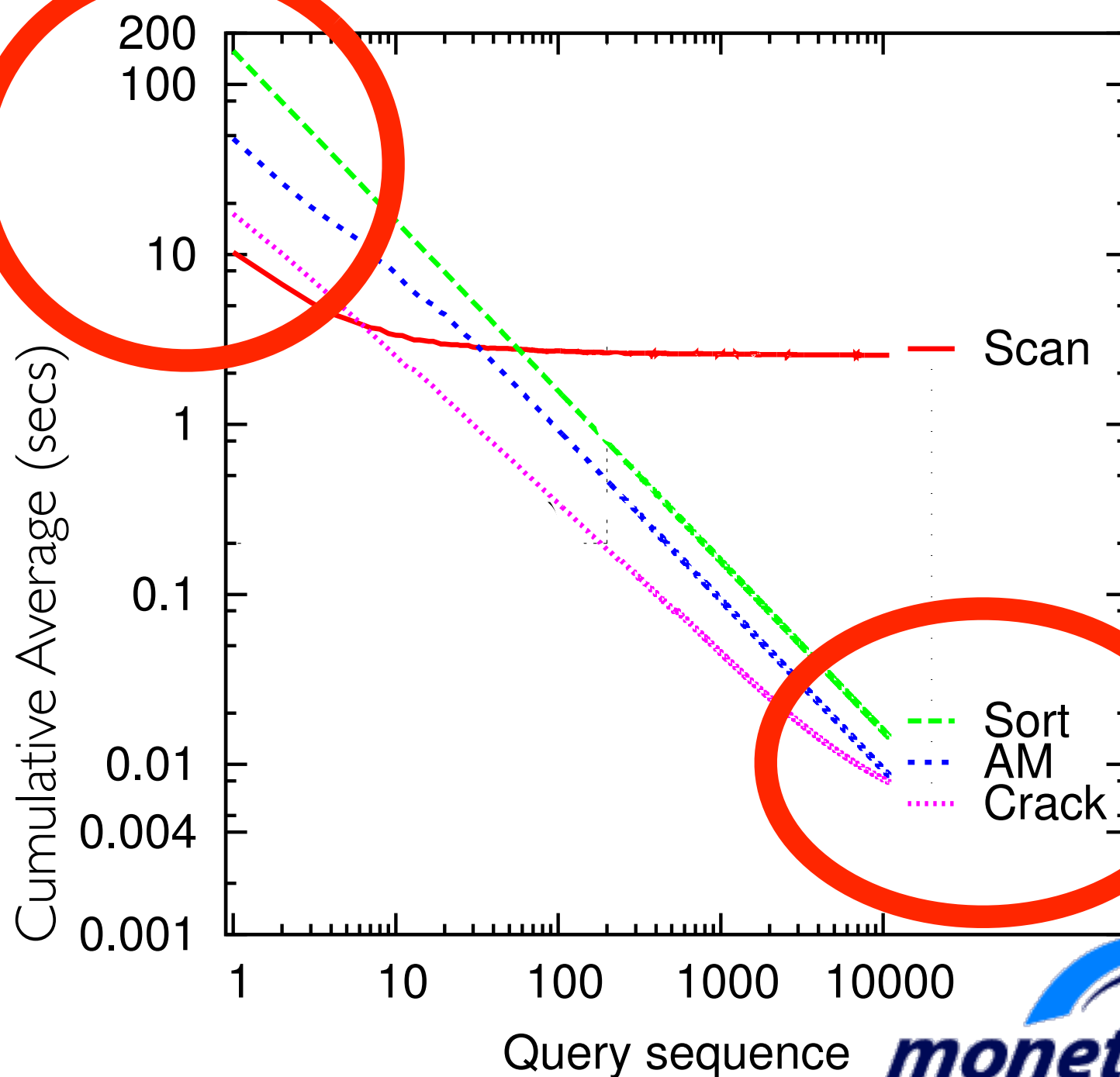


# Performance Analysis

## set-up

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

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



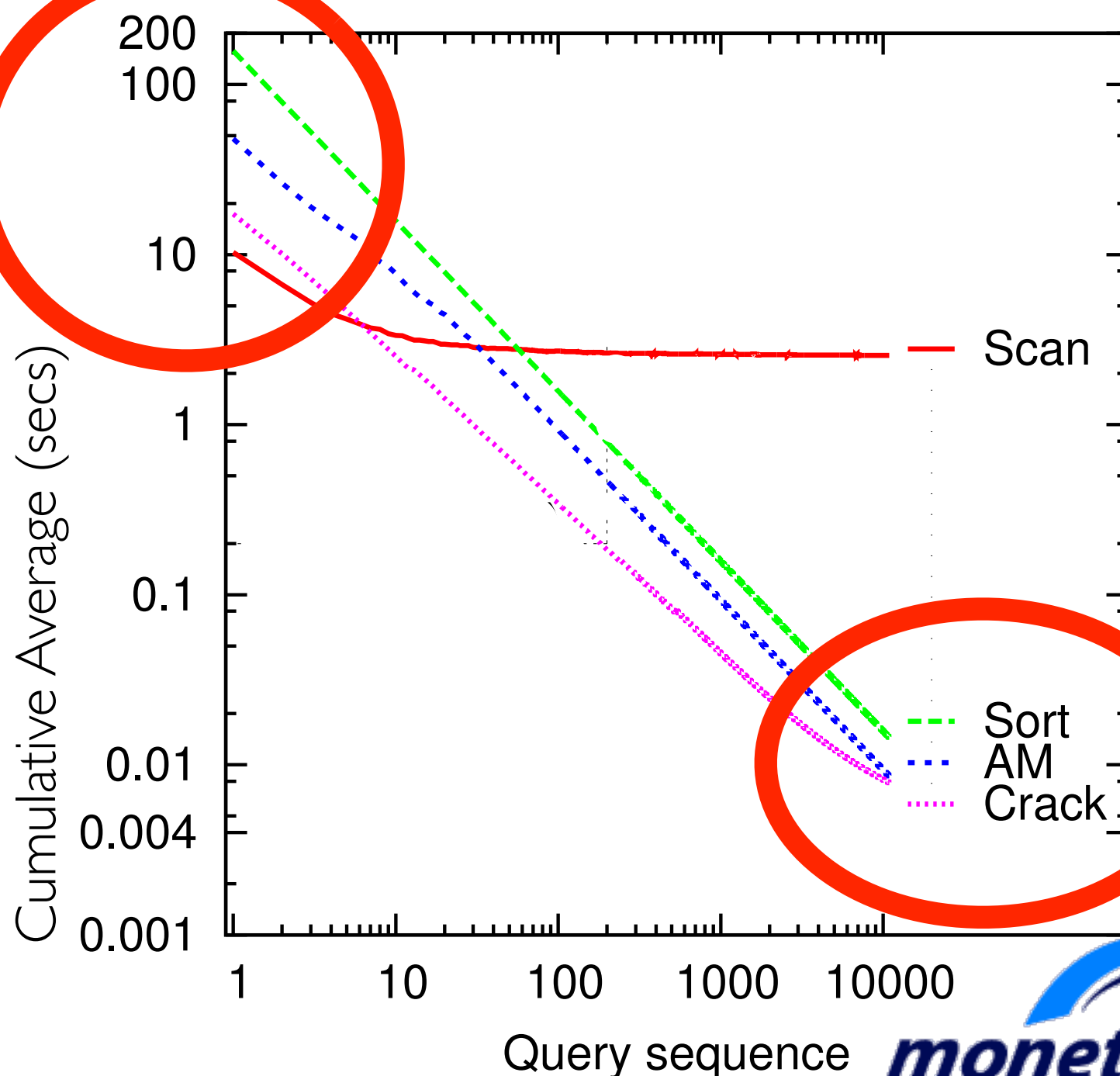
# Performance Analysis

## 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

**Adaptive merging and Cracking are extremes**

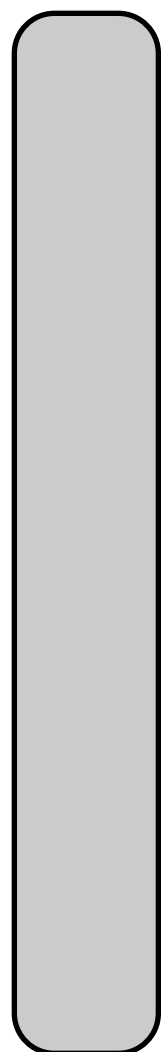
**What is there in between?**

# Crack-Crack

*vary initialization and incremental steps taken*

# Crack-Crack

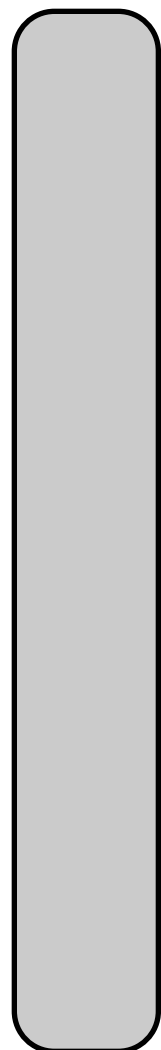
*vary initialization and incremental steps taken*



# Crack-Crack

*vary initialization and incremental steps taken*

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



# Crack-Crack

*vary initialization and incremental steps taken*

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



crack

# Crack-Crack

*vary initialization and incremental steps taken*

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



crack

crack

# Crack-Crack

*vary initialization and incremental steps taken*

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



crack

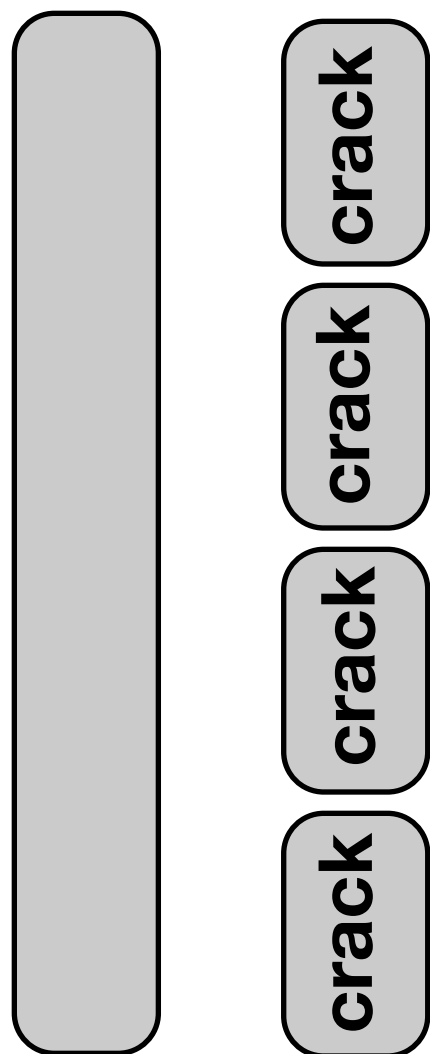
crack

crack

# Crack-Crack

*vary initialization and incremental steps taken*

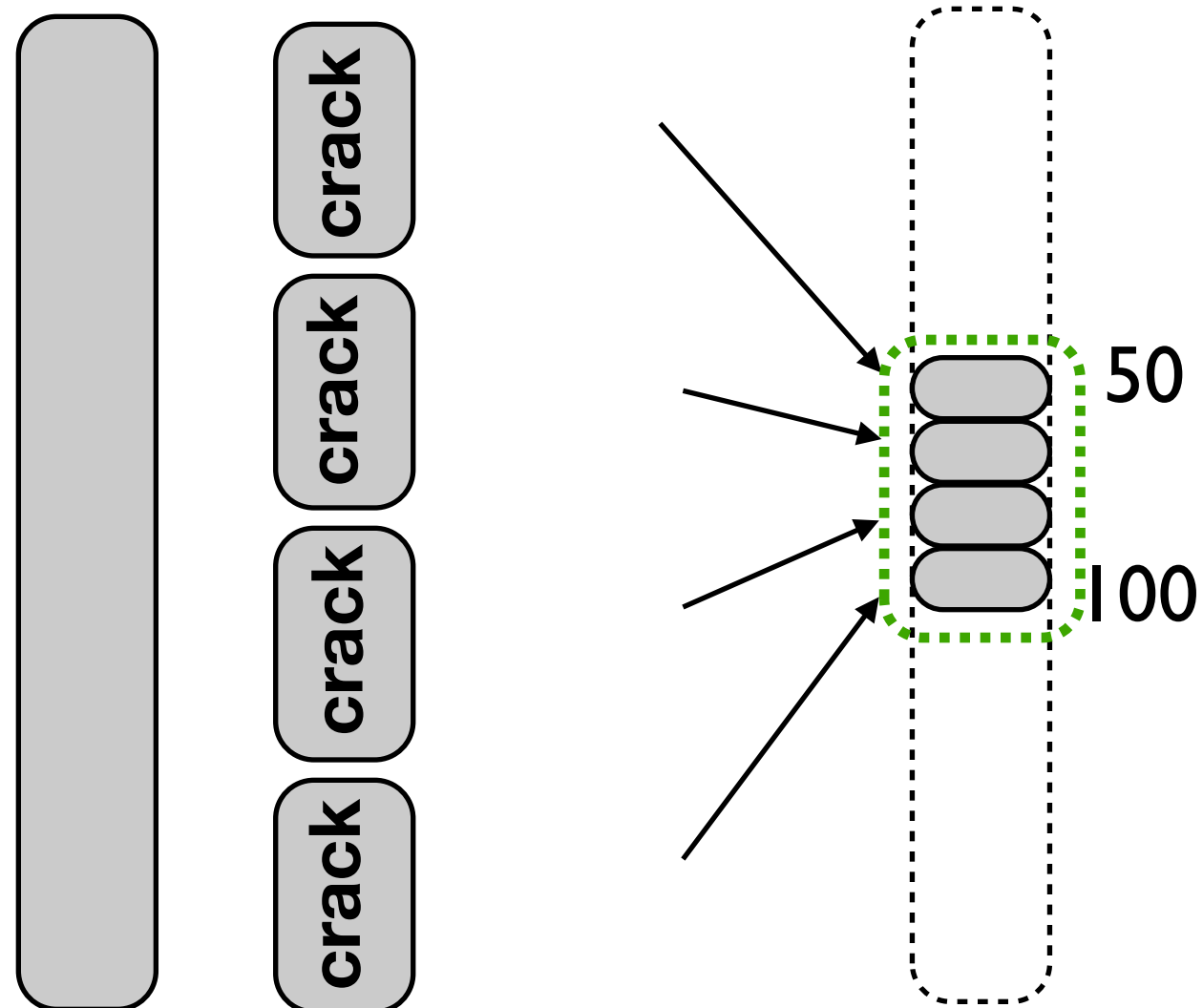
```
select(A,50,100)
```





# Crack-Crack

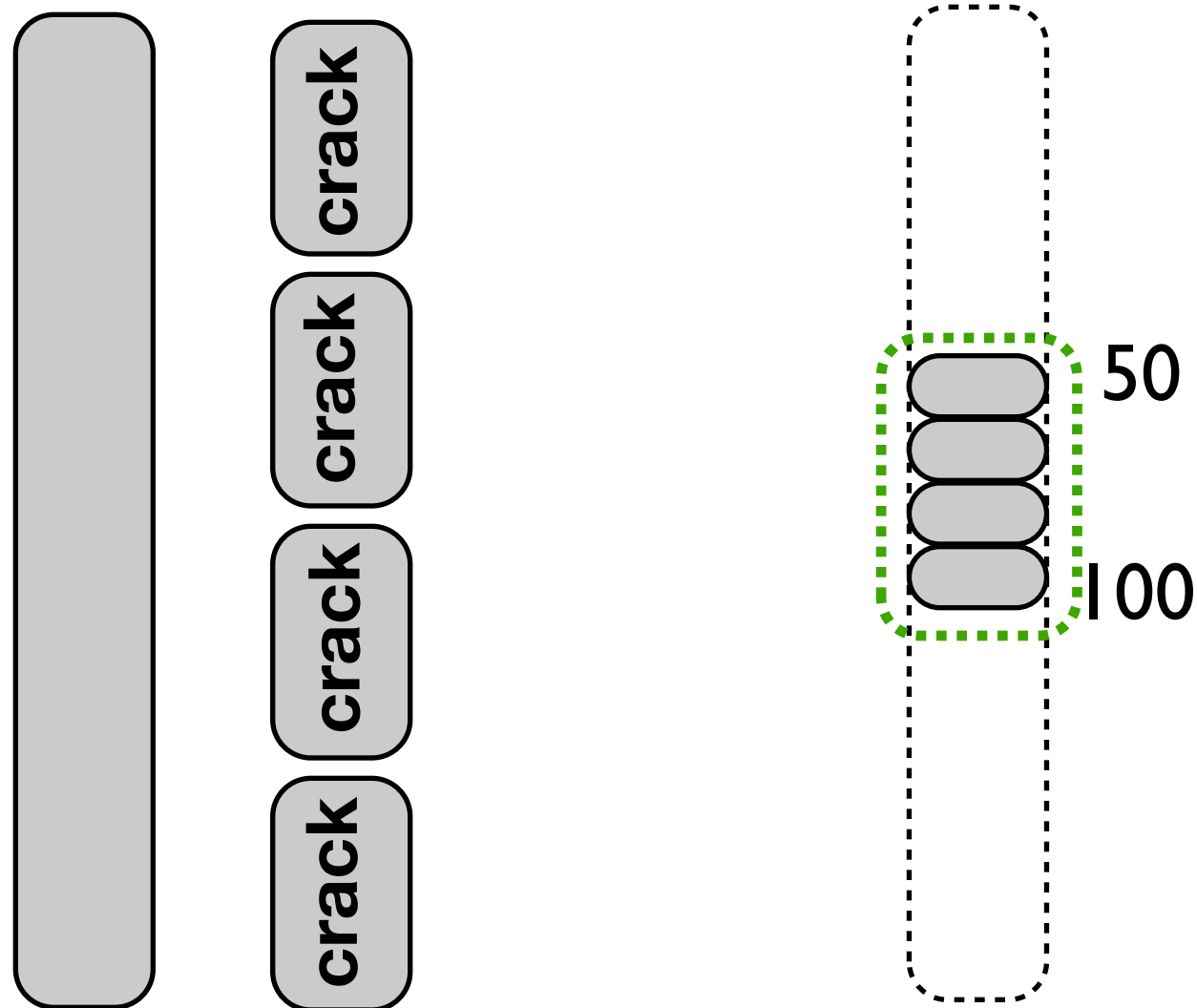
*vary initialization and incremental steps taken*



# Crack-Crack

*vary initialization and incremental steps taken*

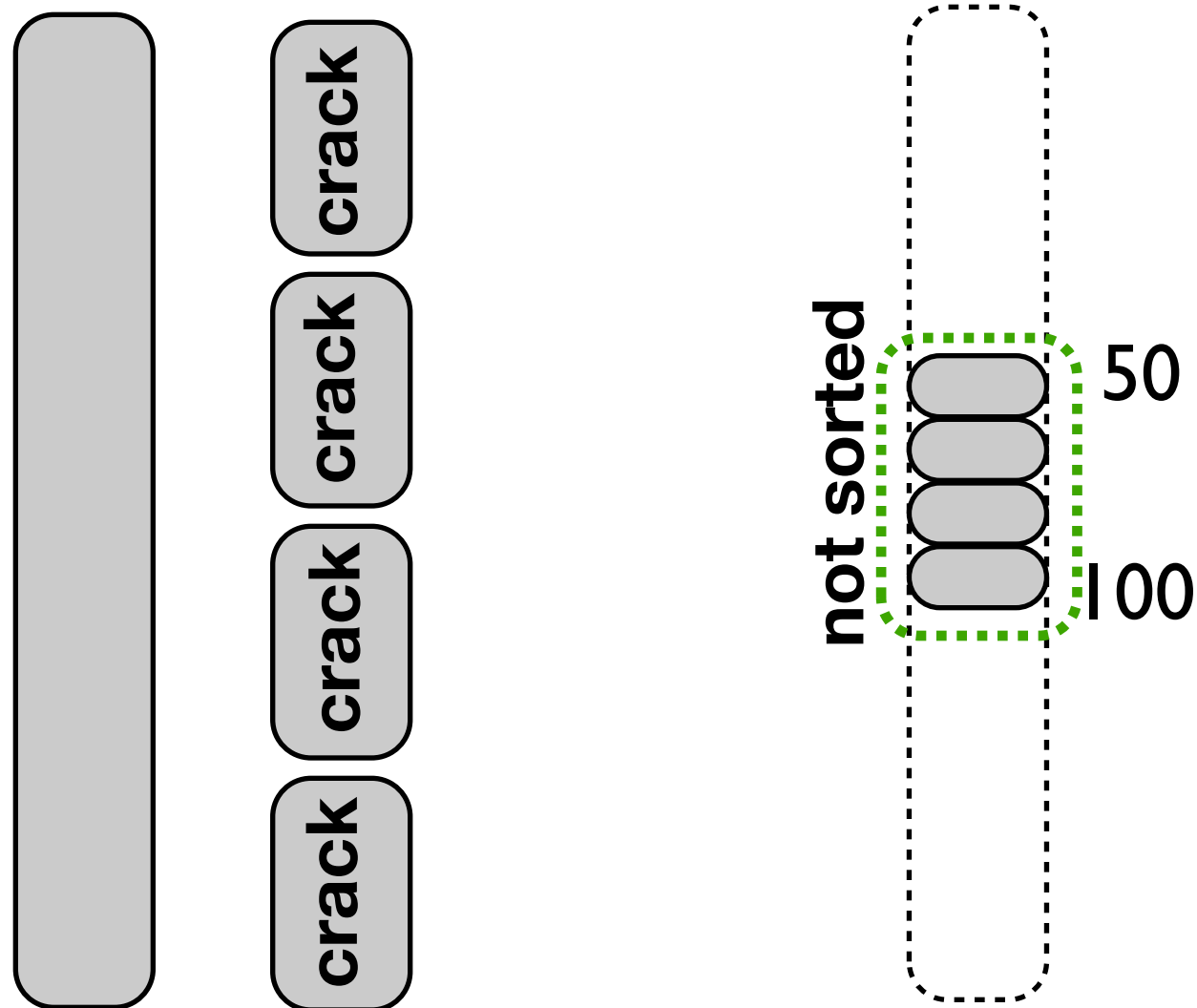
select(A,50,100)



# Crack-Crack

*vary initialization and incremental steps taken*

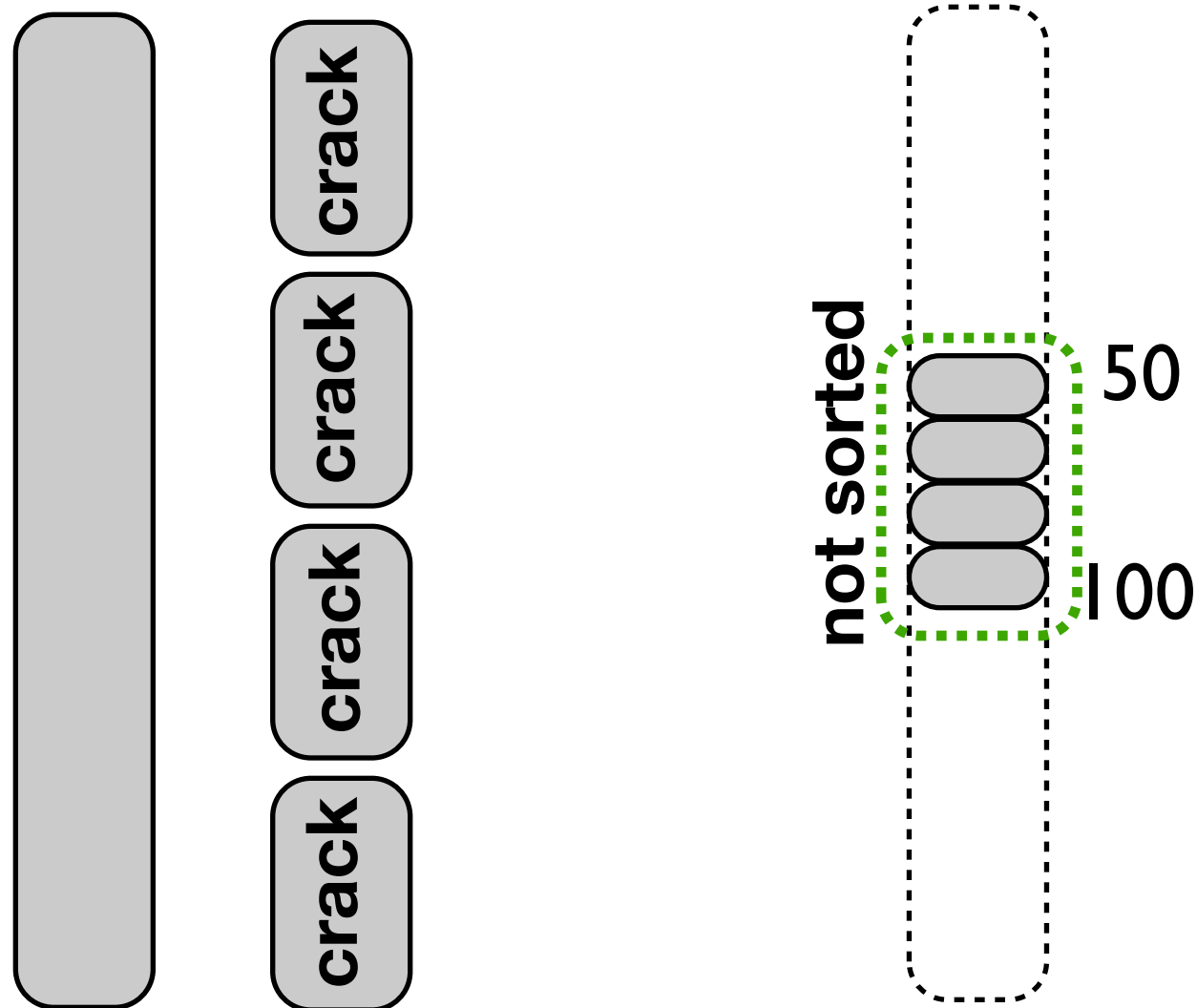
select(A,50,100)



# Crack-Crack

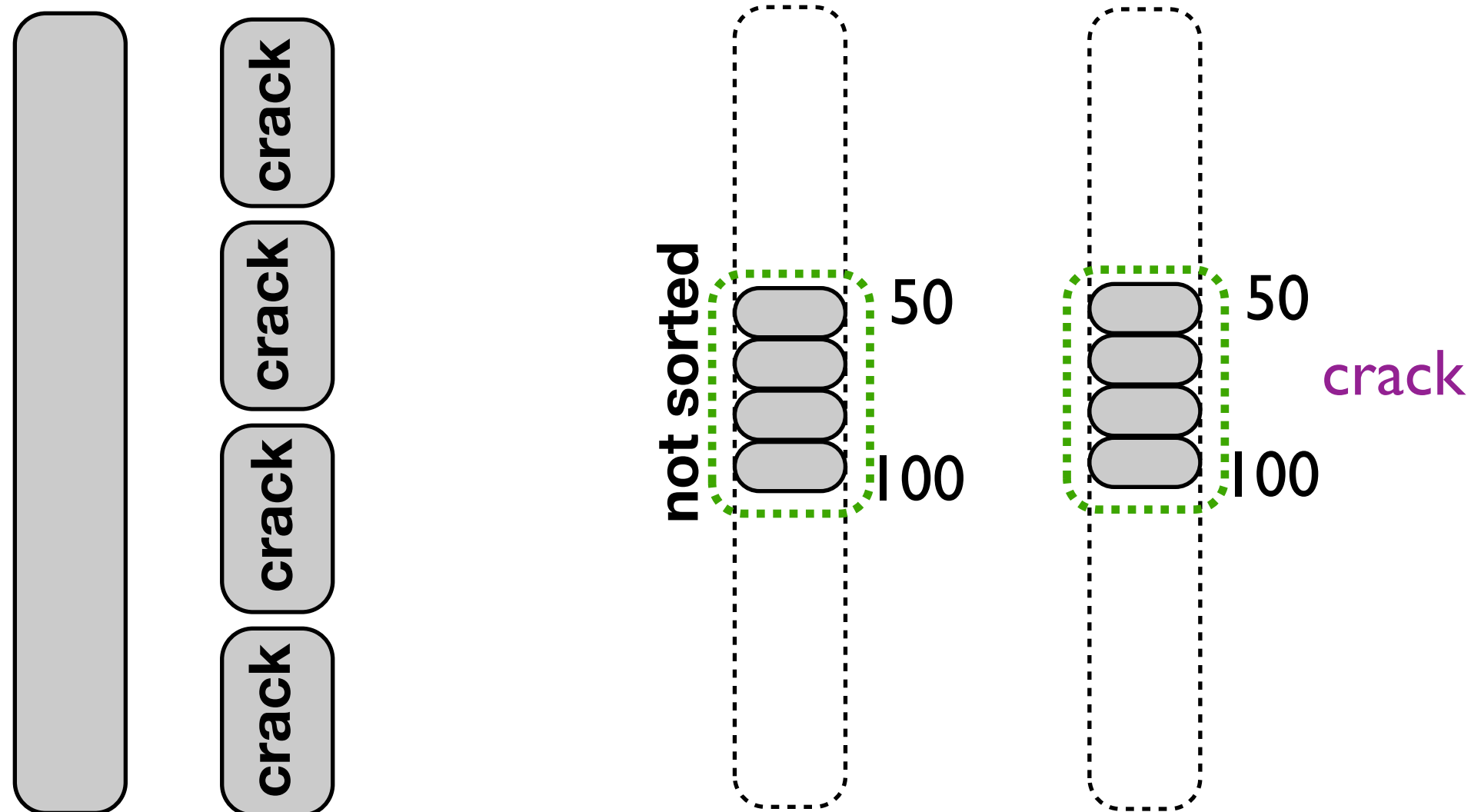
*vary initialization and incremental steps taken*

select(A,50,100)    select(A,55,70)



# Crack-Crack

*vary initialization and incremental steps taken*



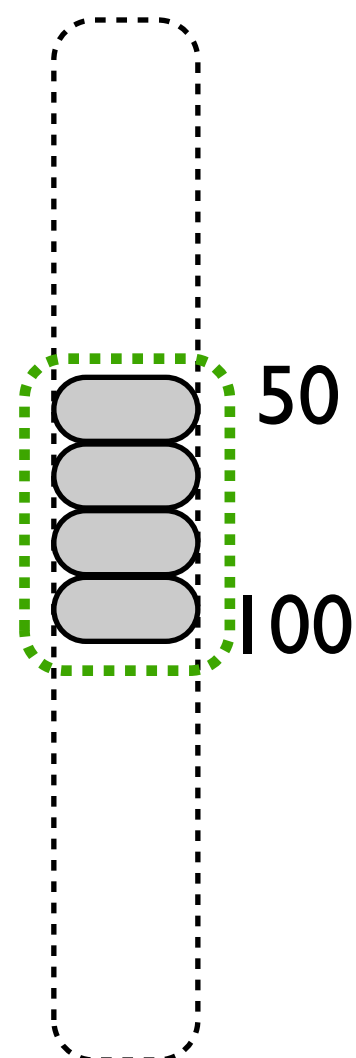
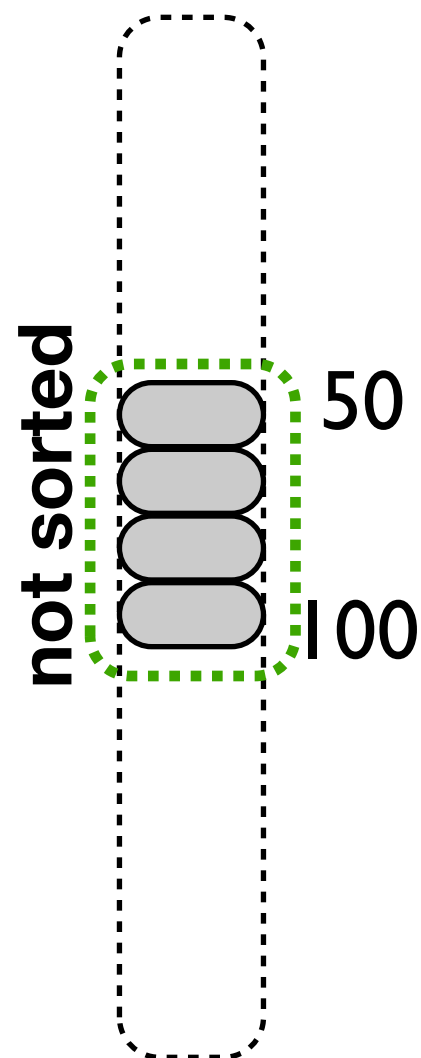
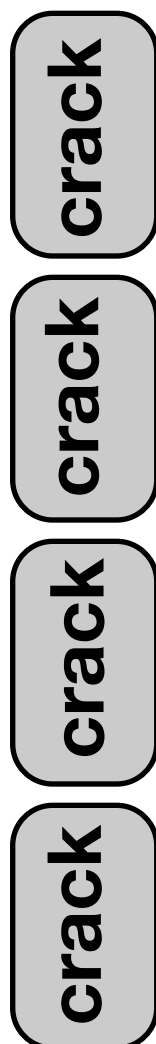
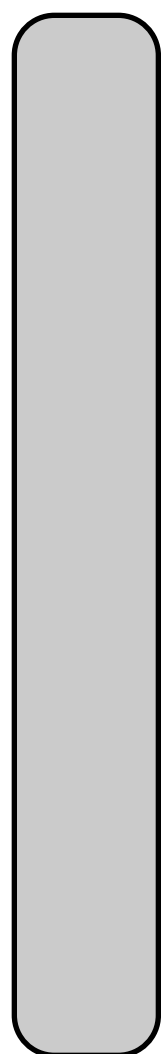
# Crack-Crack

*vary initialization and incremental steps taken*

select(A,50,100)

select(A,55,70)

select(A,150,170)



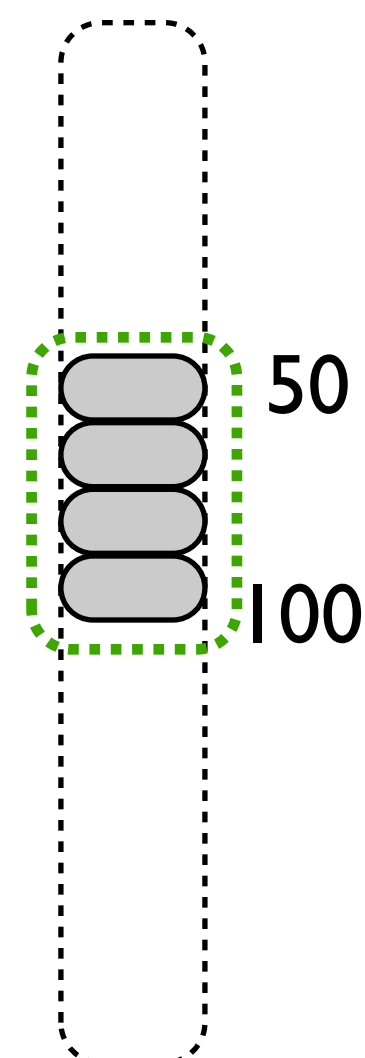
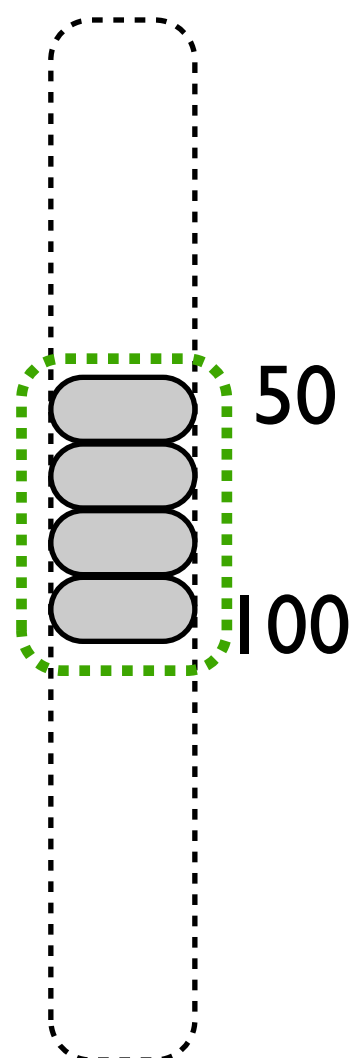
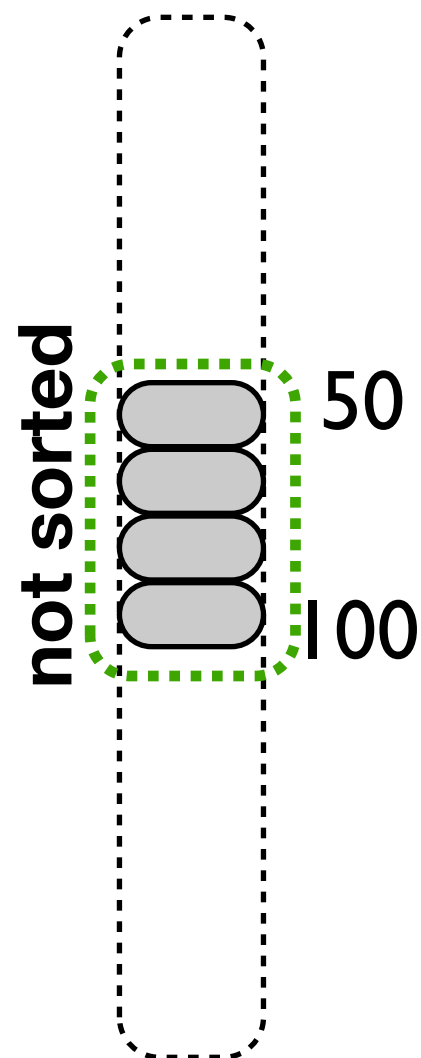
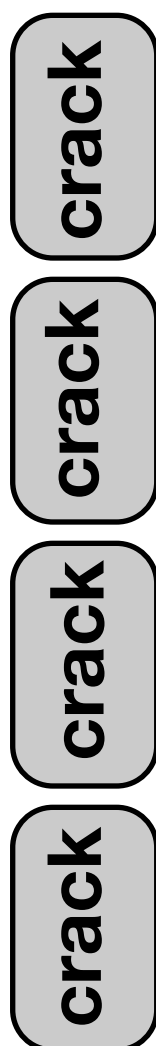
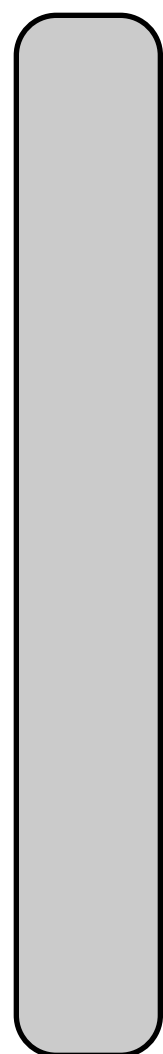
# Crack-Crack

*vary initialization and incremental steps taken*

select(A,50,100)

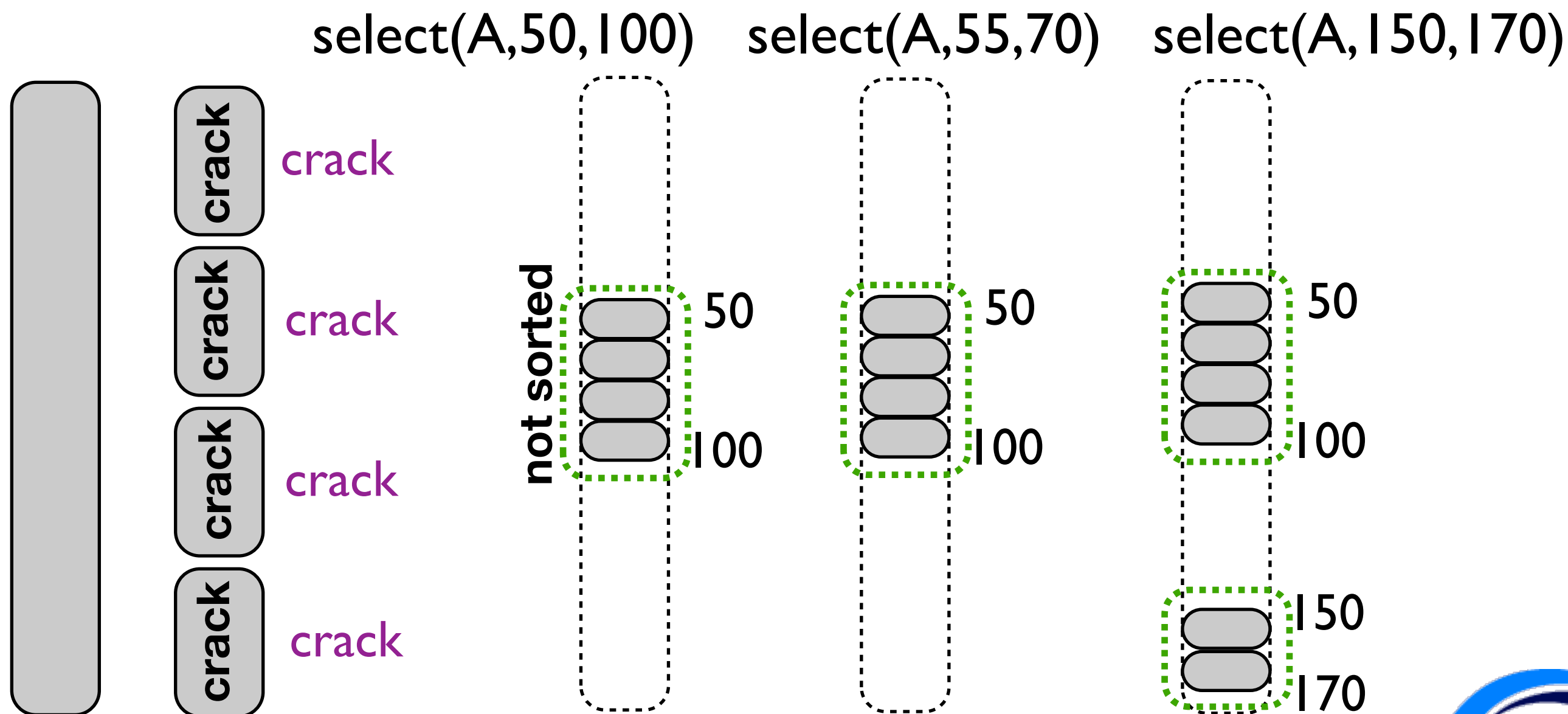
select(A,55,70)

select(A,150,170)



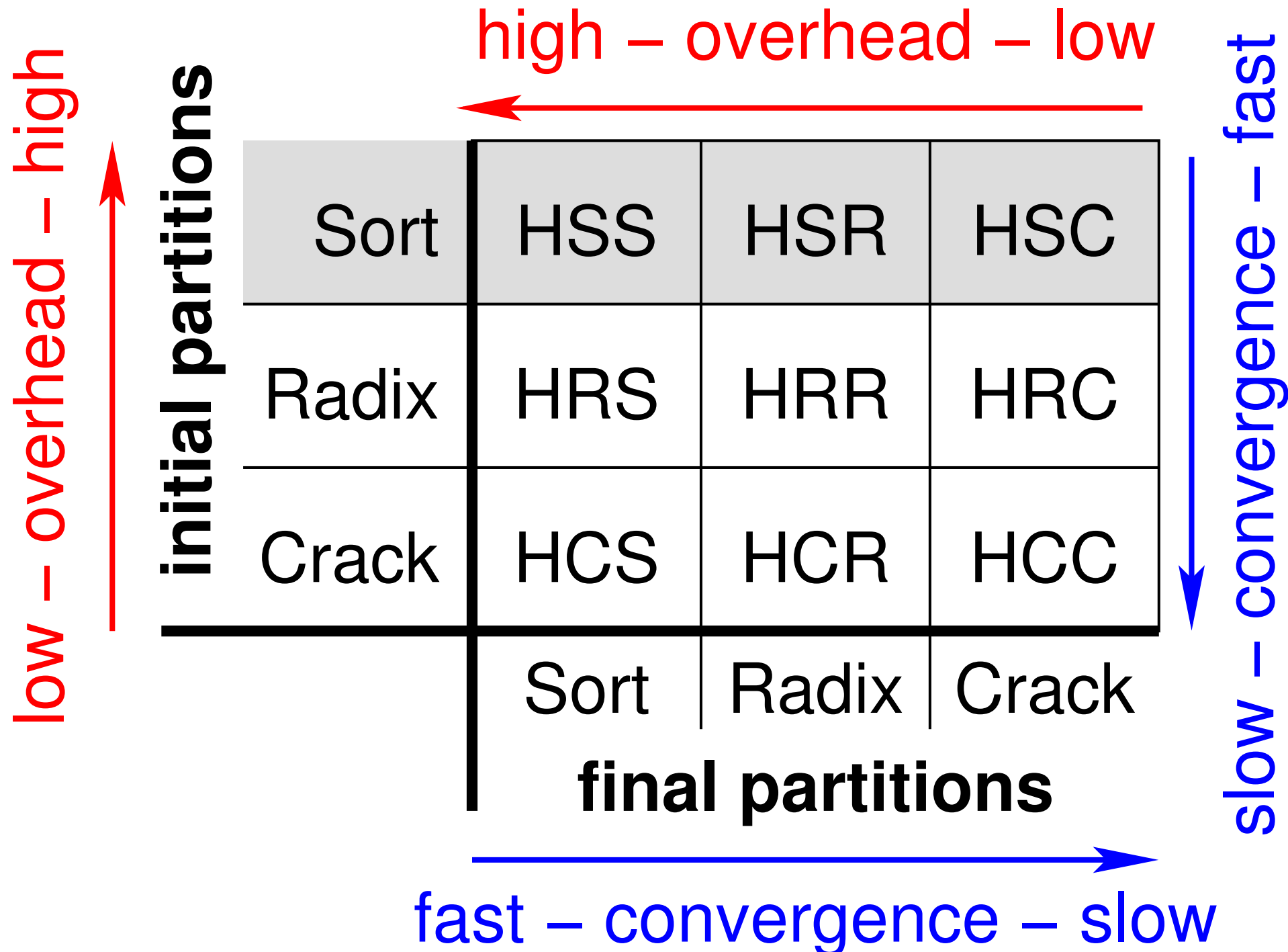
# Crack-Crack

*vary initialization and incremental steps taken*

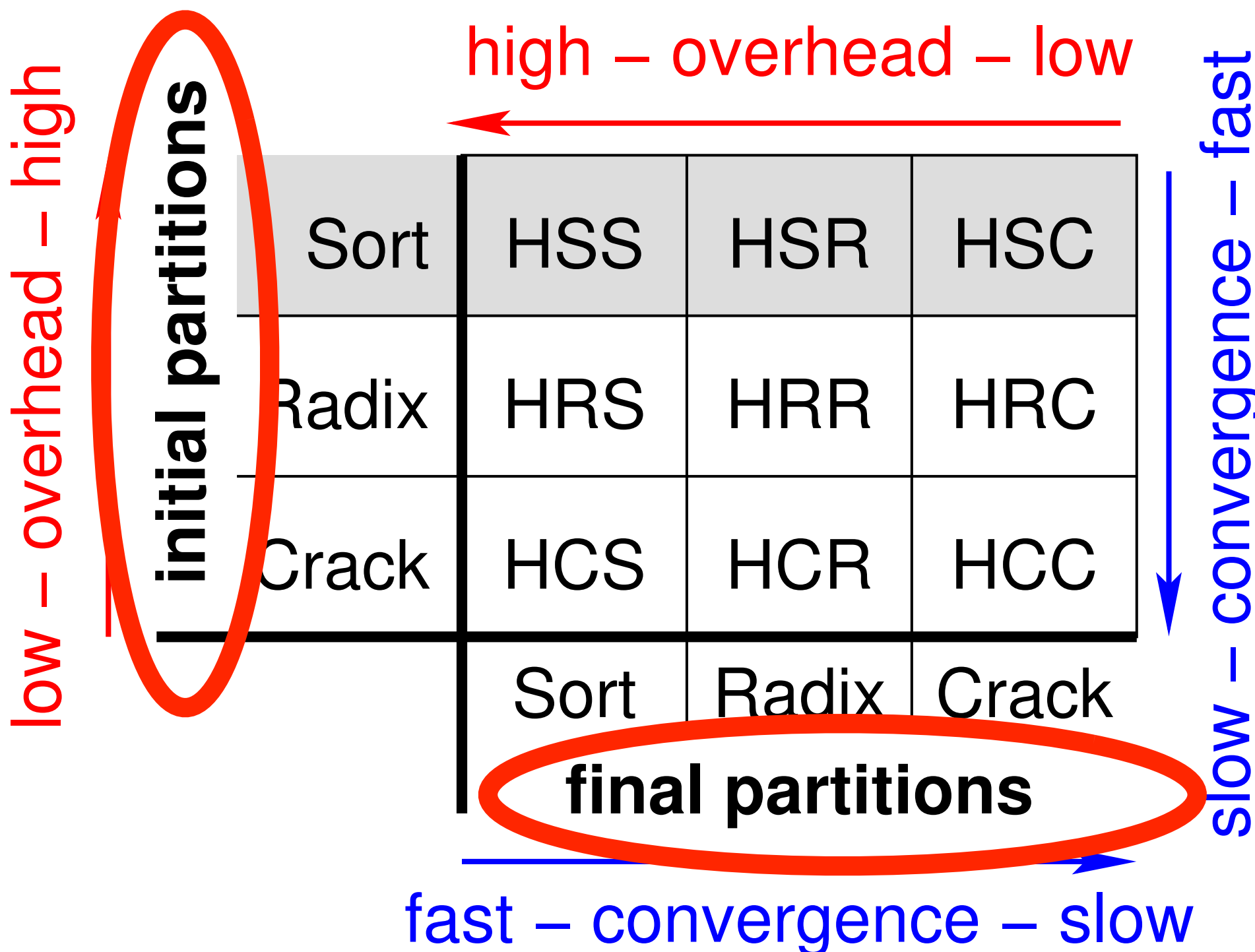




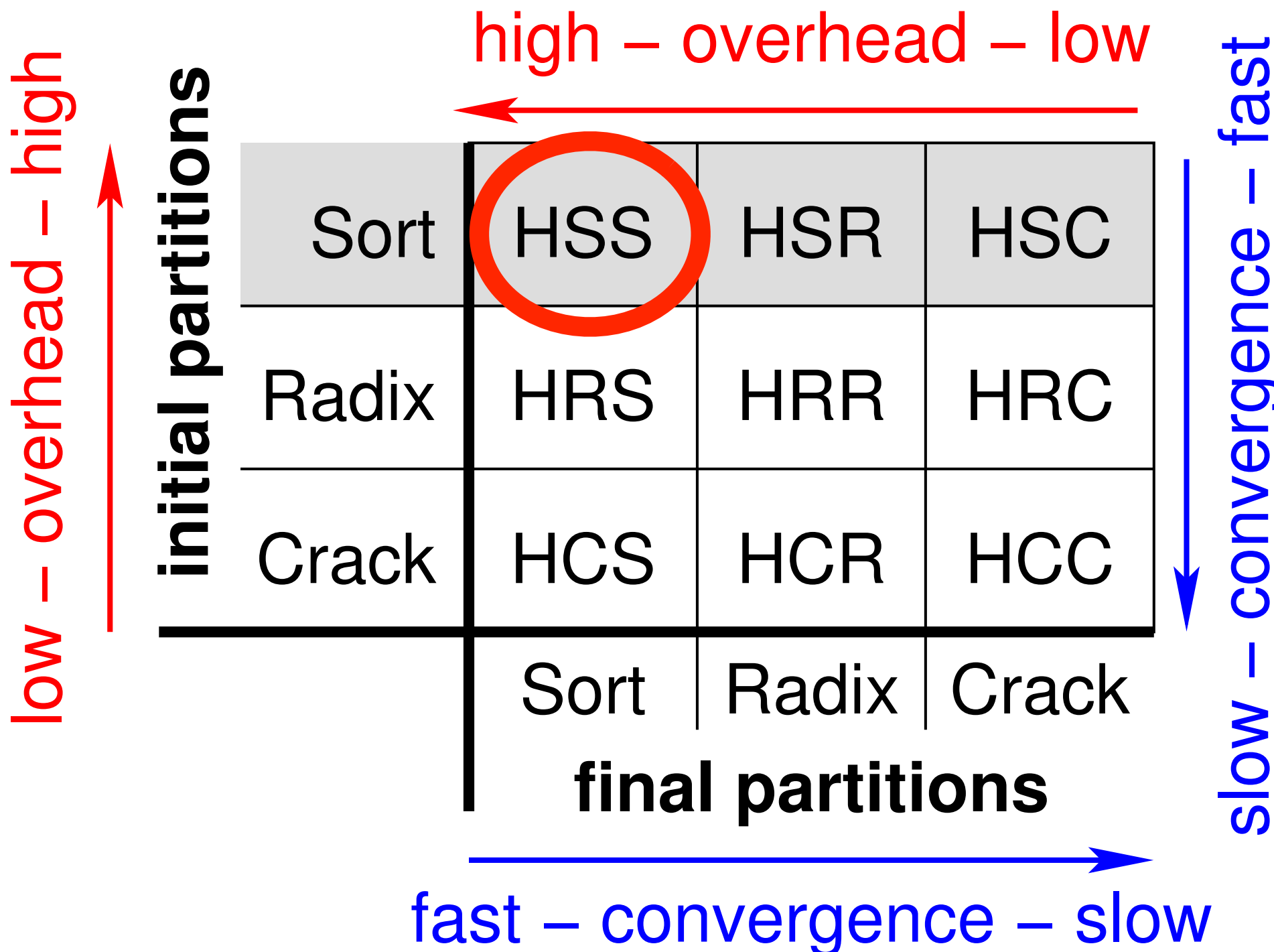
# Adaptive Indexing



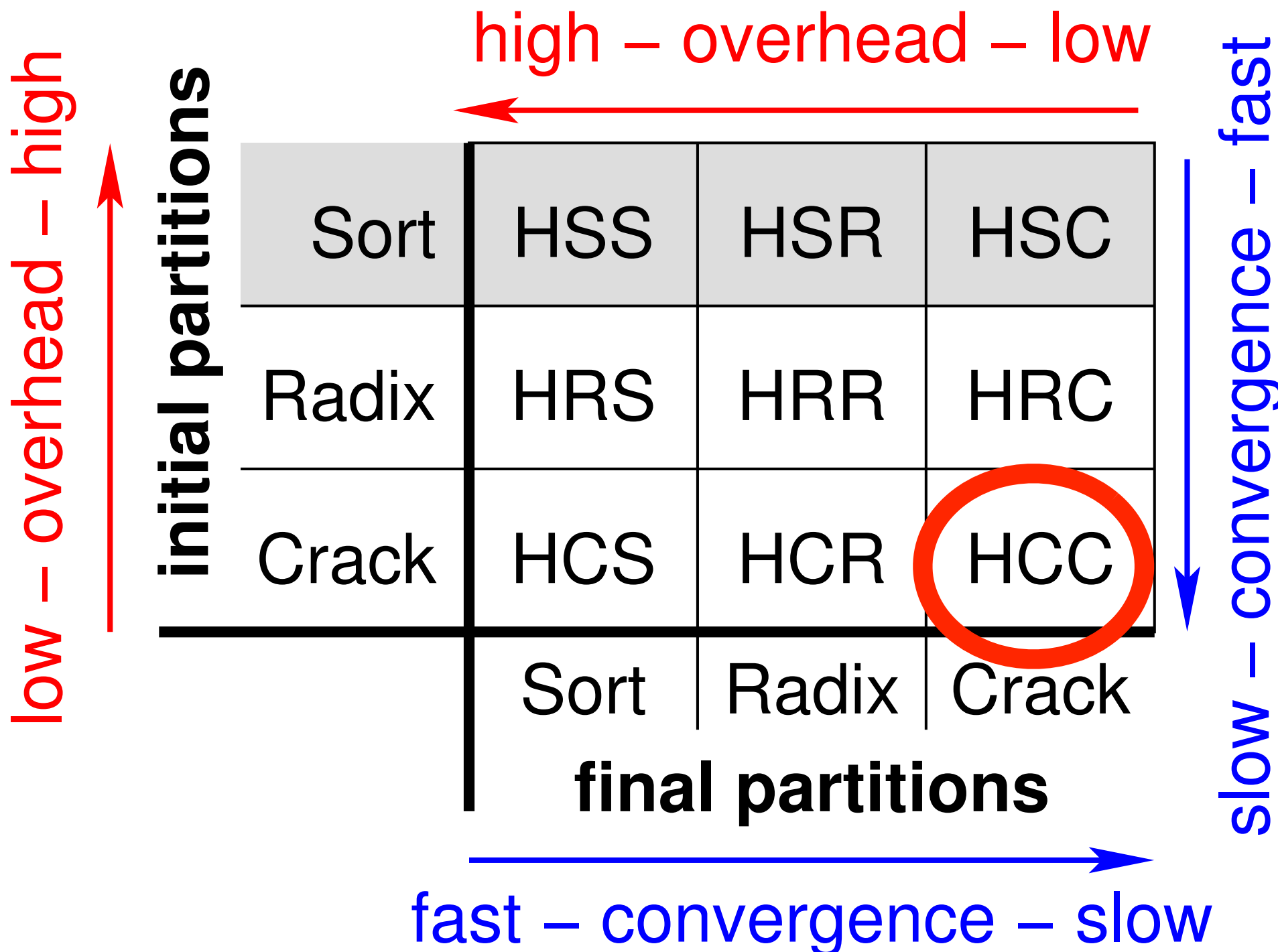
# Adaptive Indexing



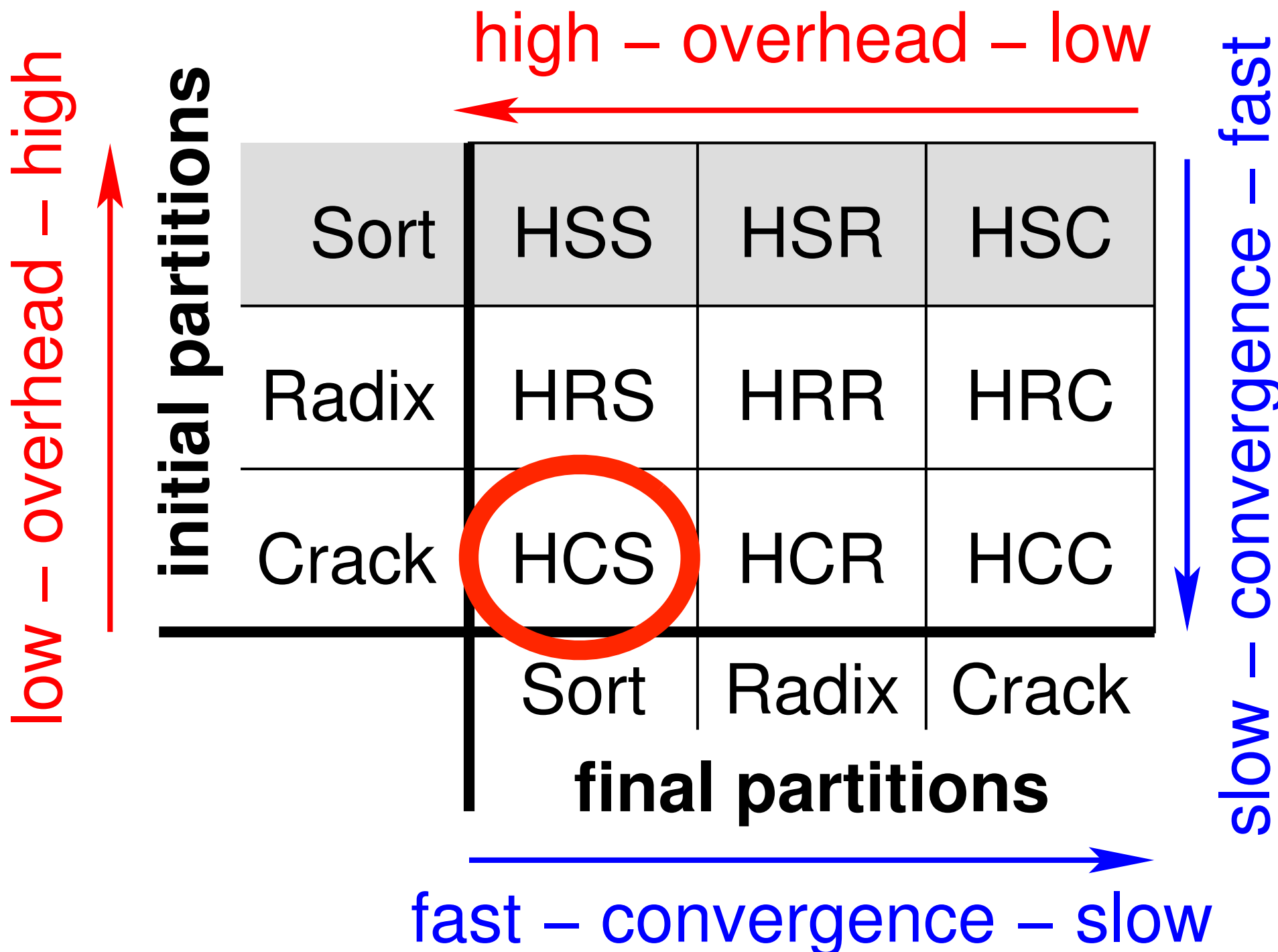
# Adaptive Indexing

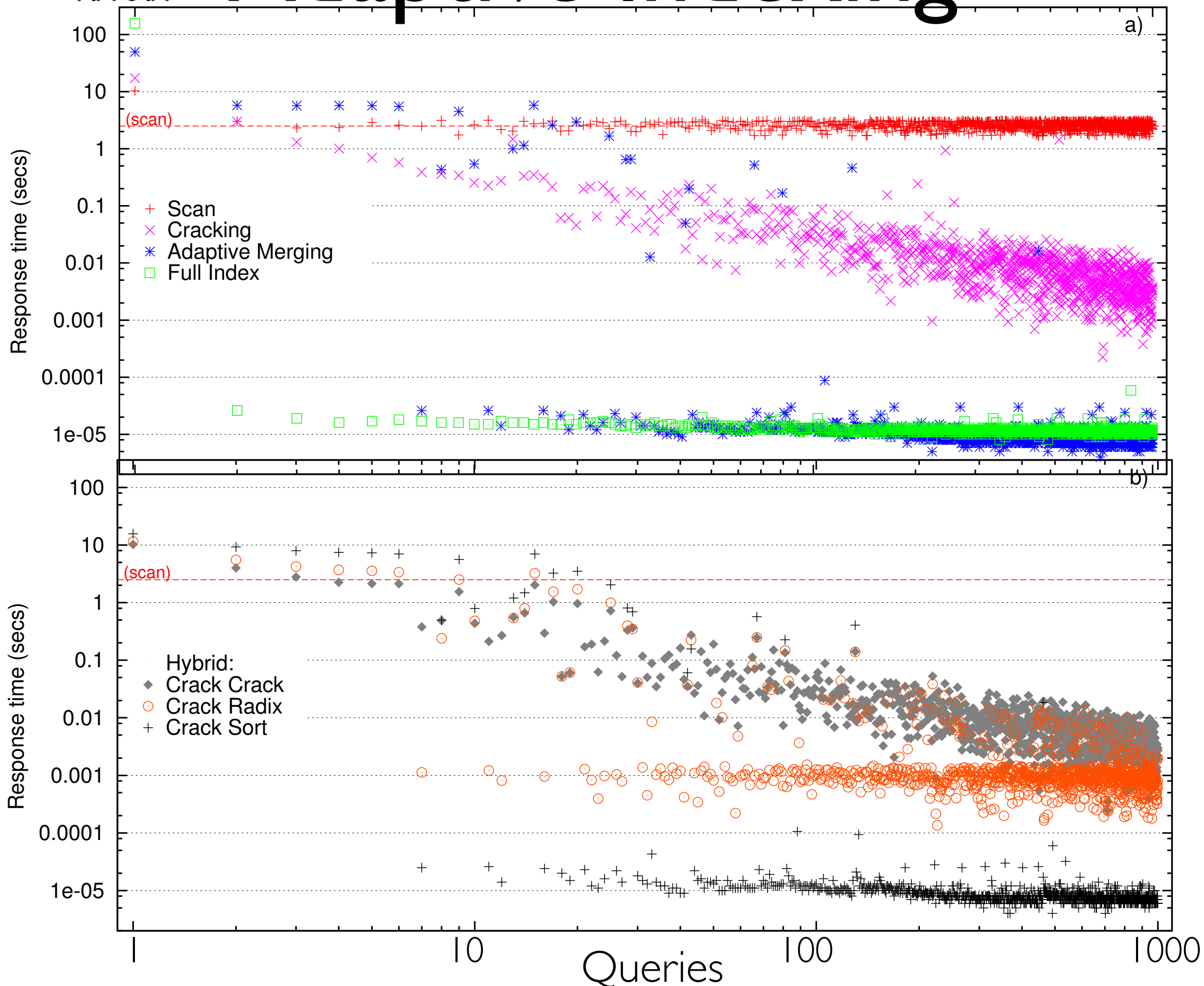


# Adaptive Indexing

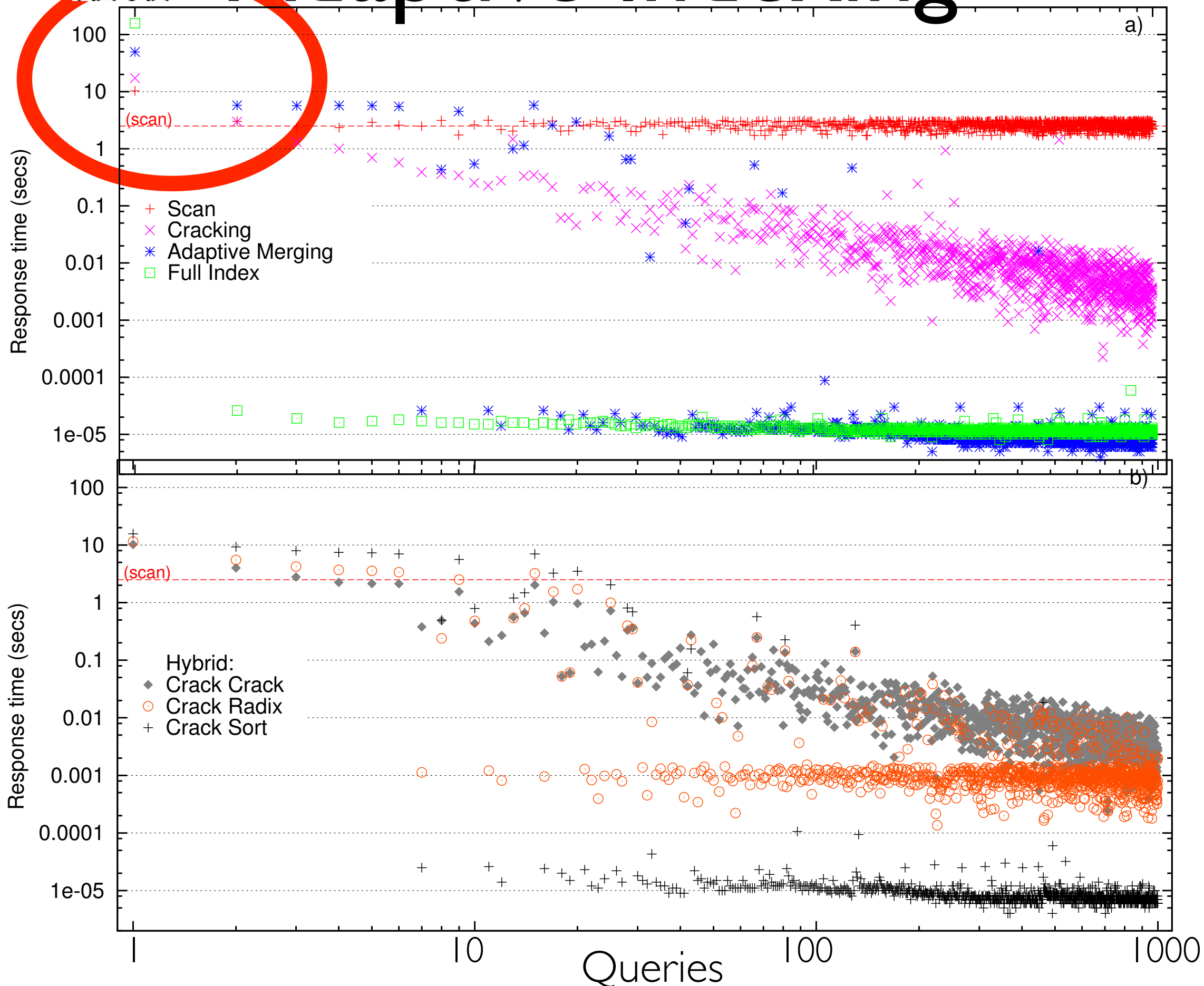


# Adaptive Indexing



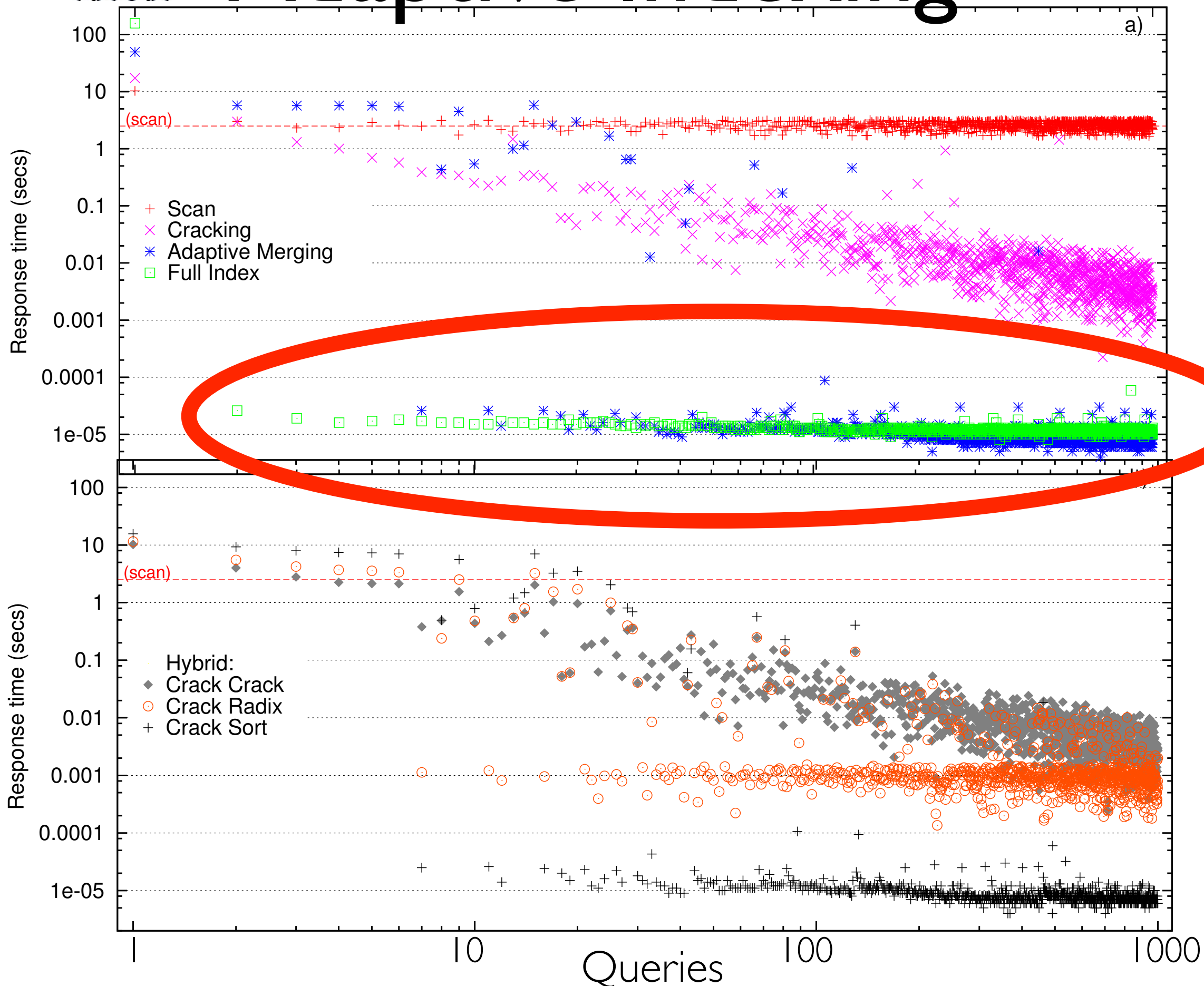


# Adaptive Indexing



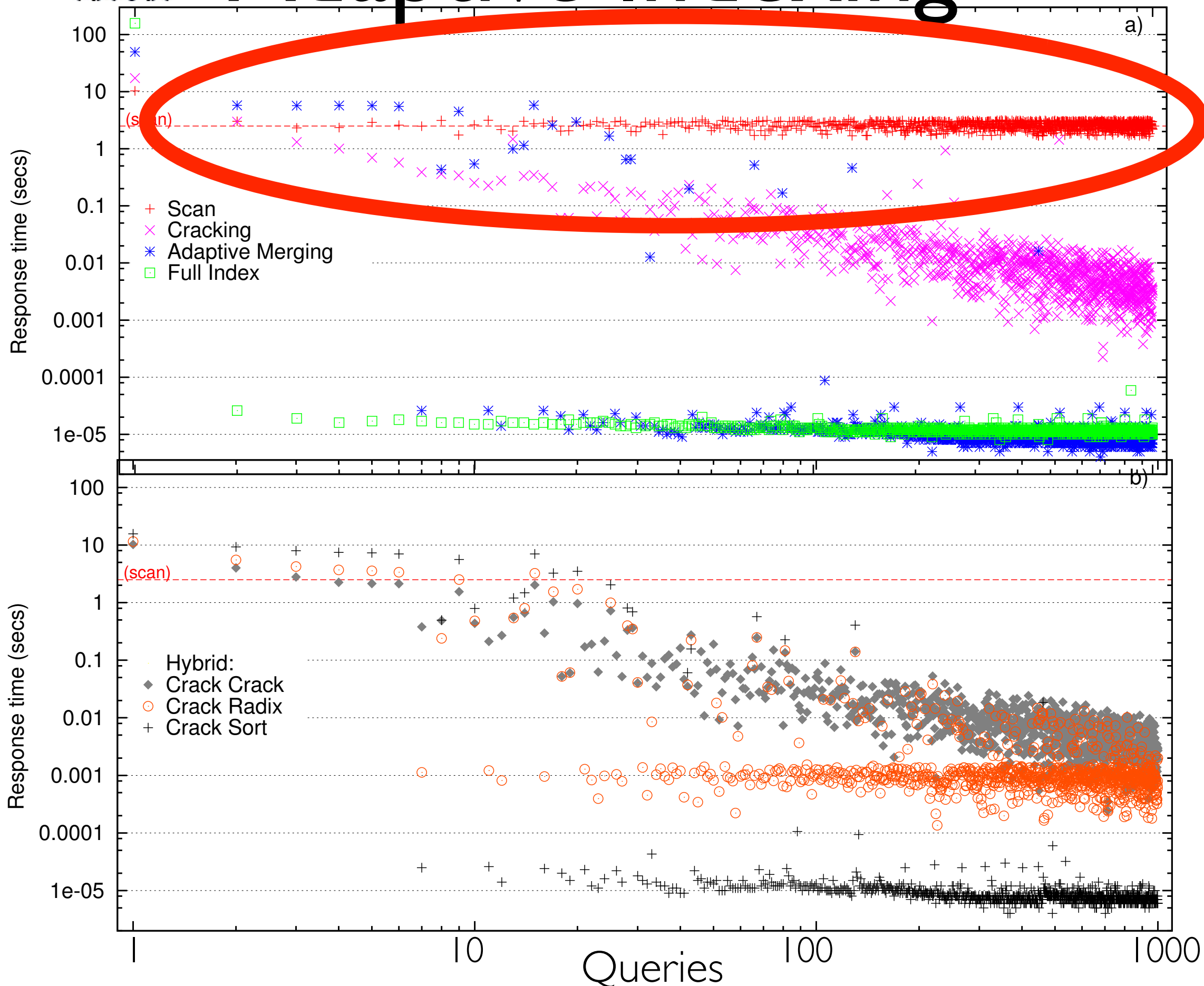


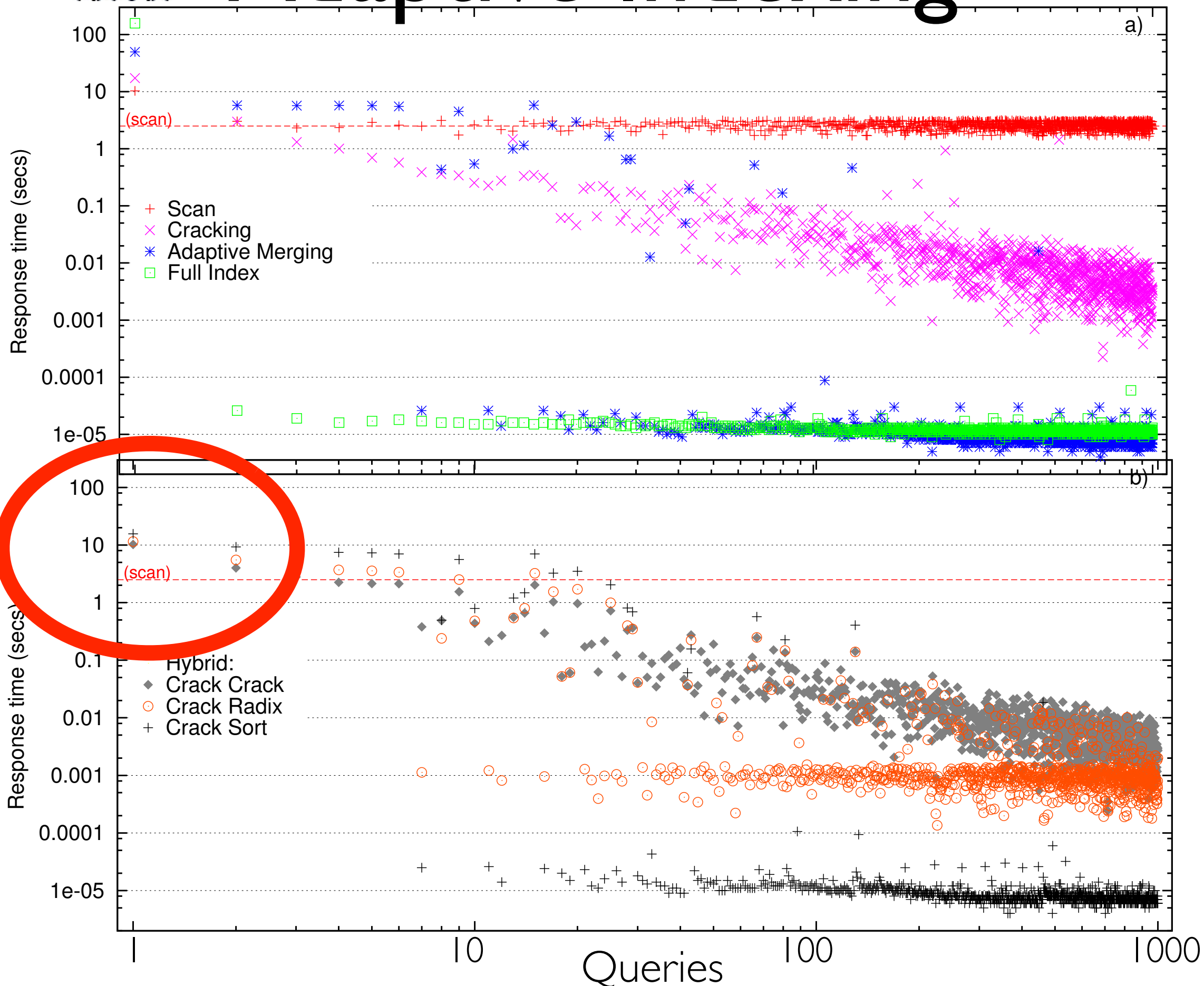
# Adaptive Indexing



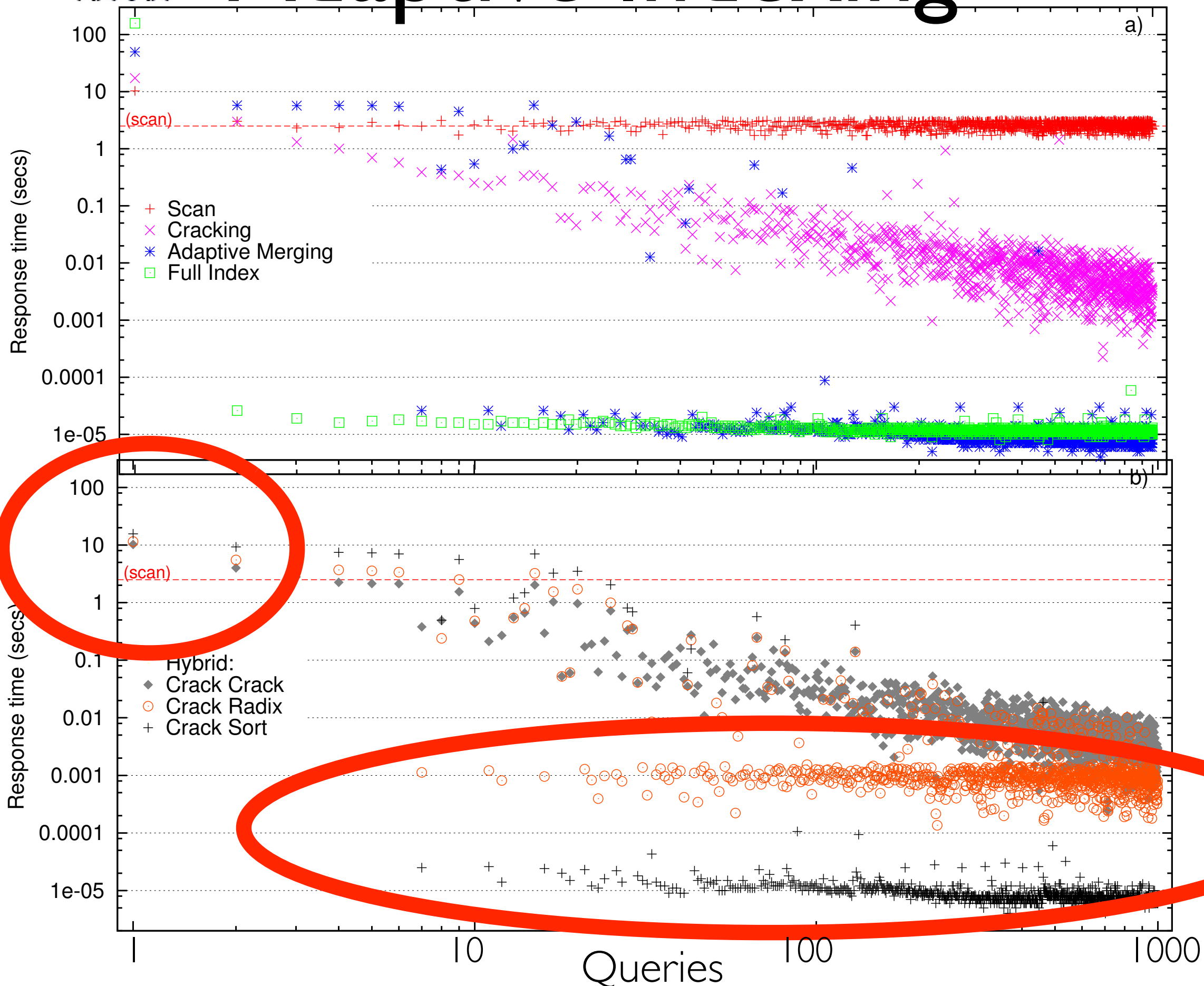


# Adaptive Indexing

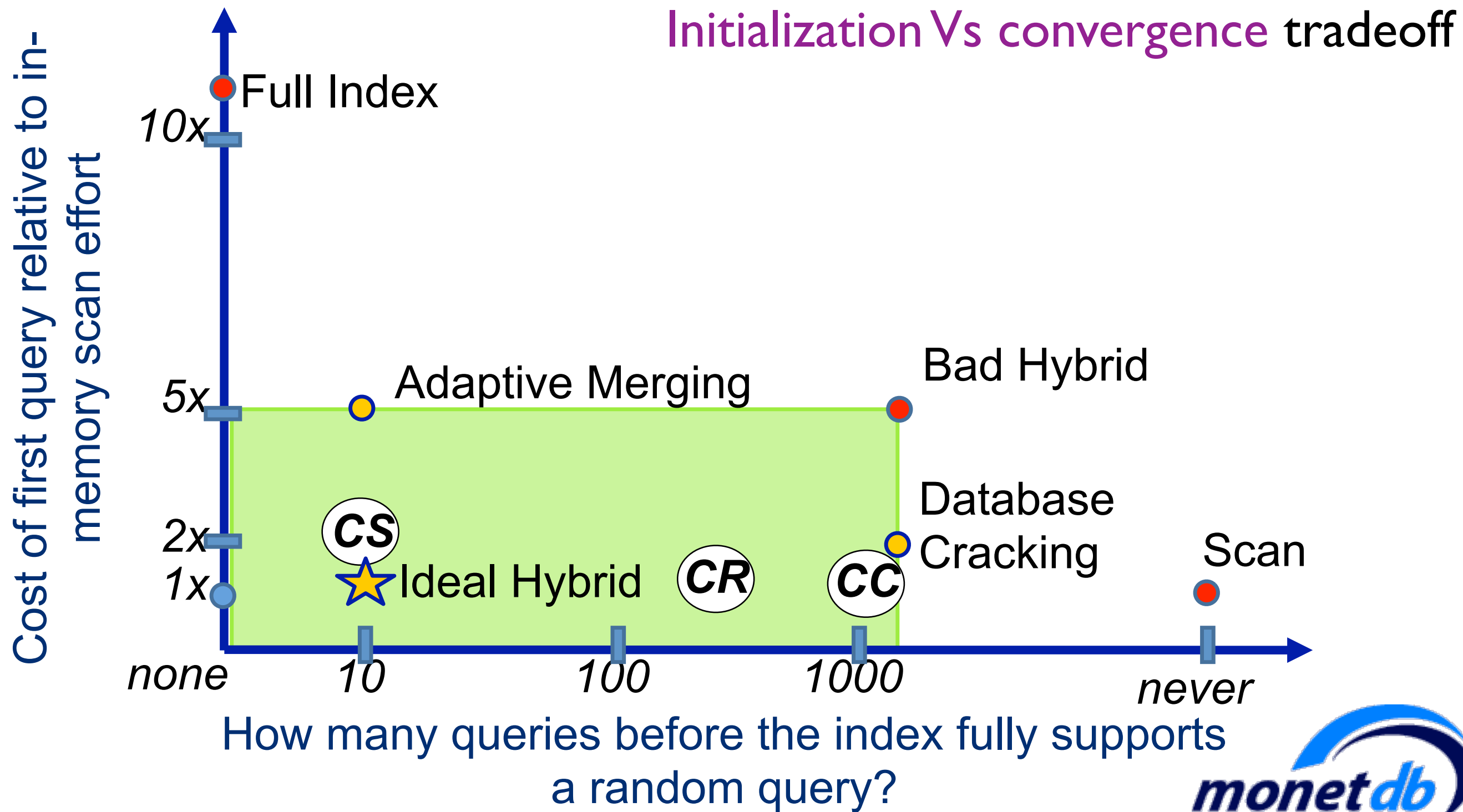




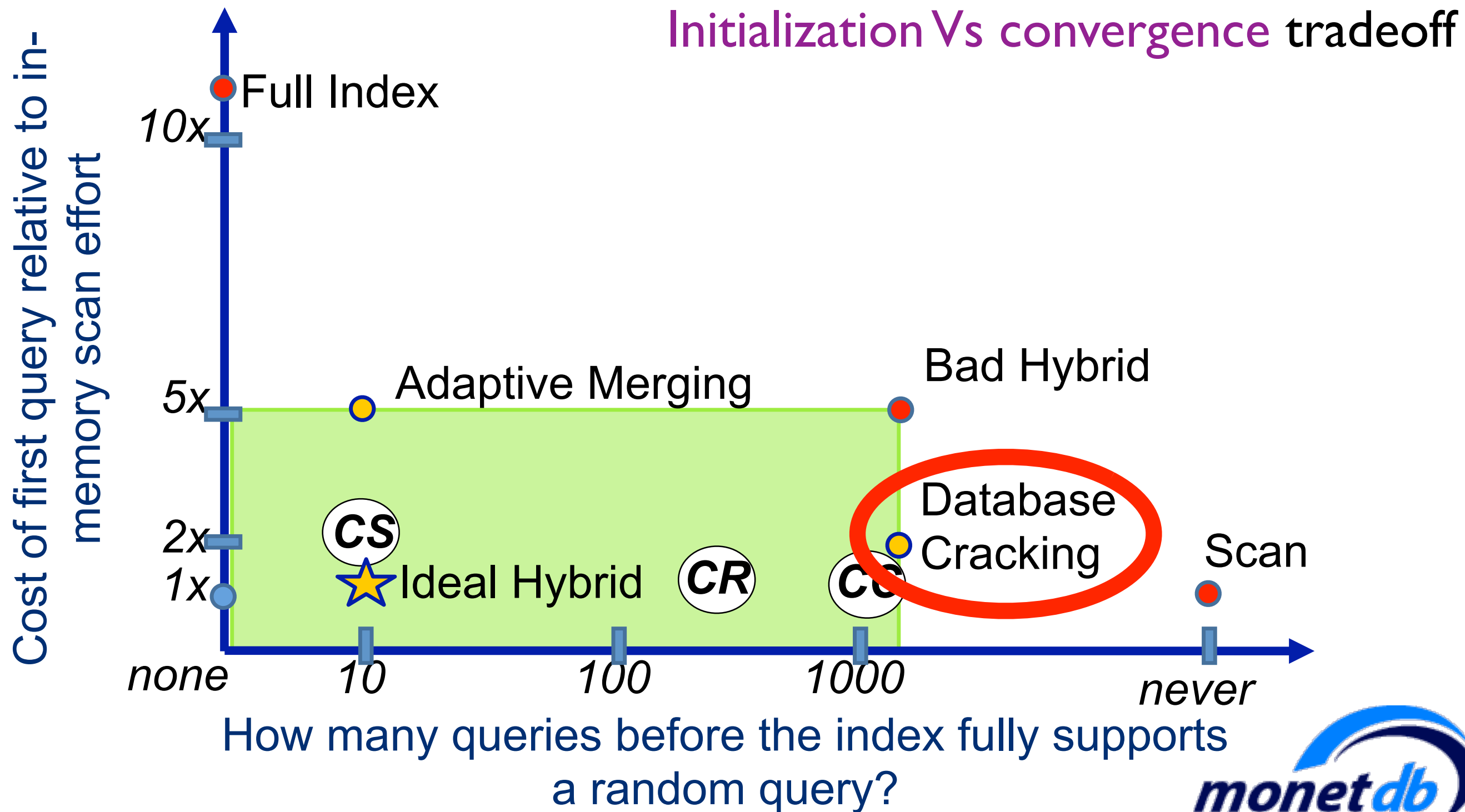
# Adaptive Indexing



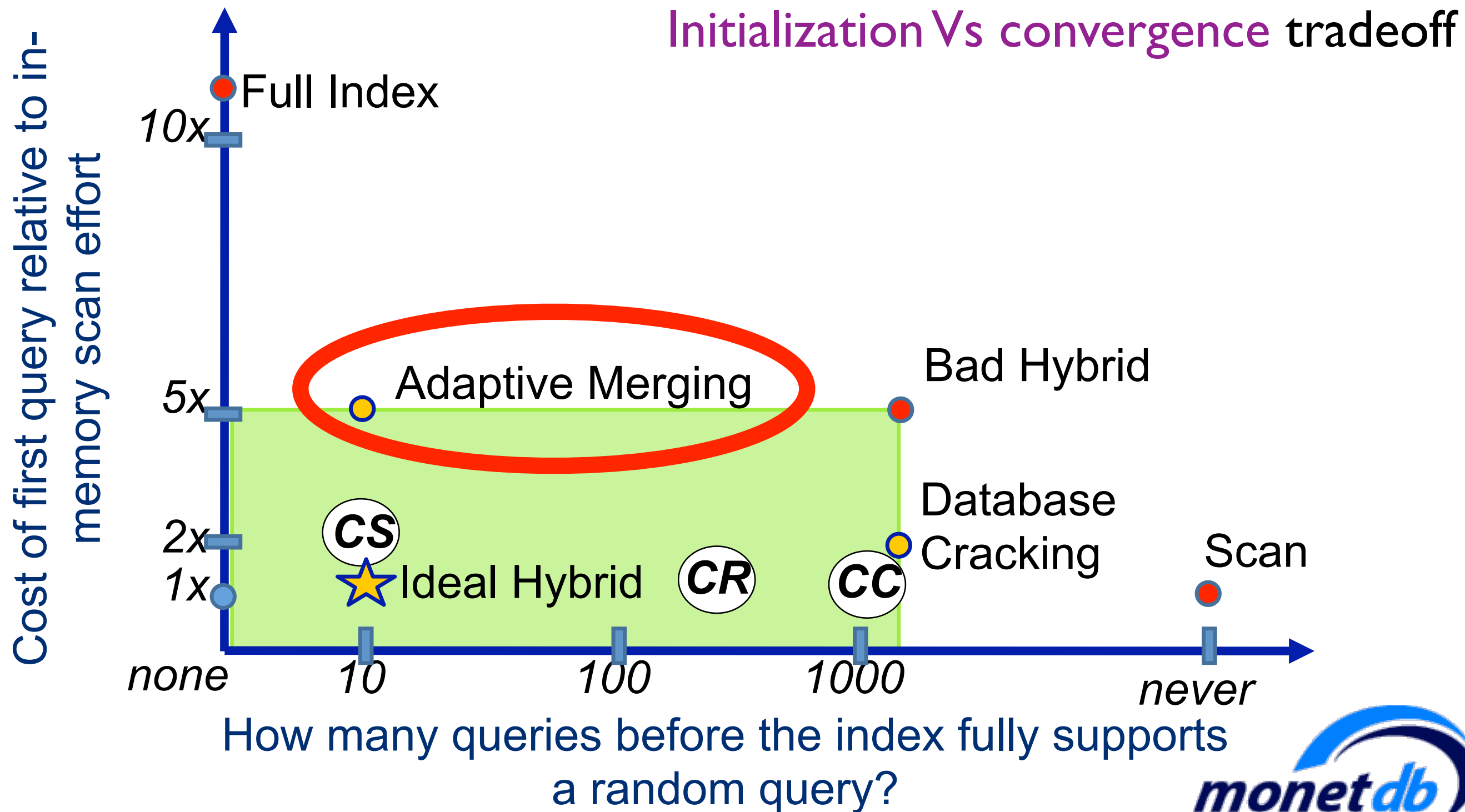
# Adaptive Indexing



# Adaptive Indexing

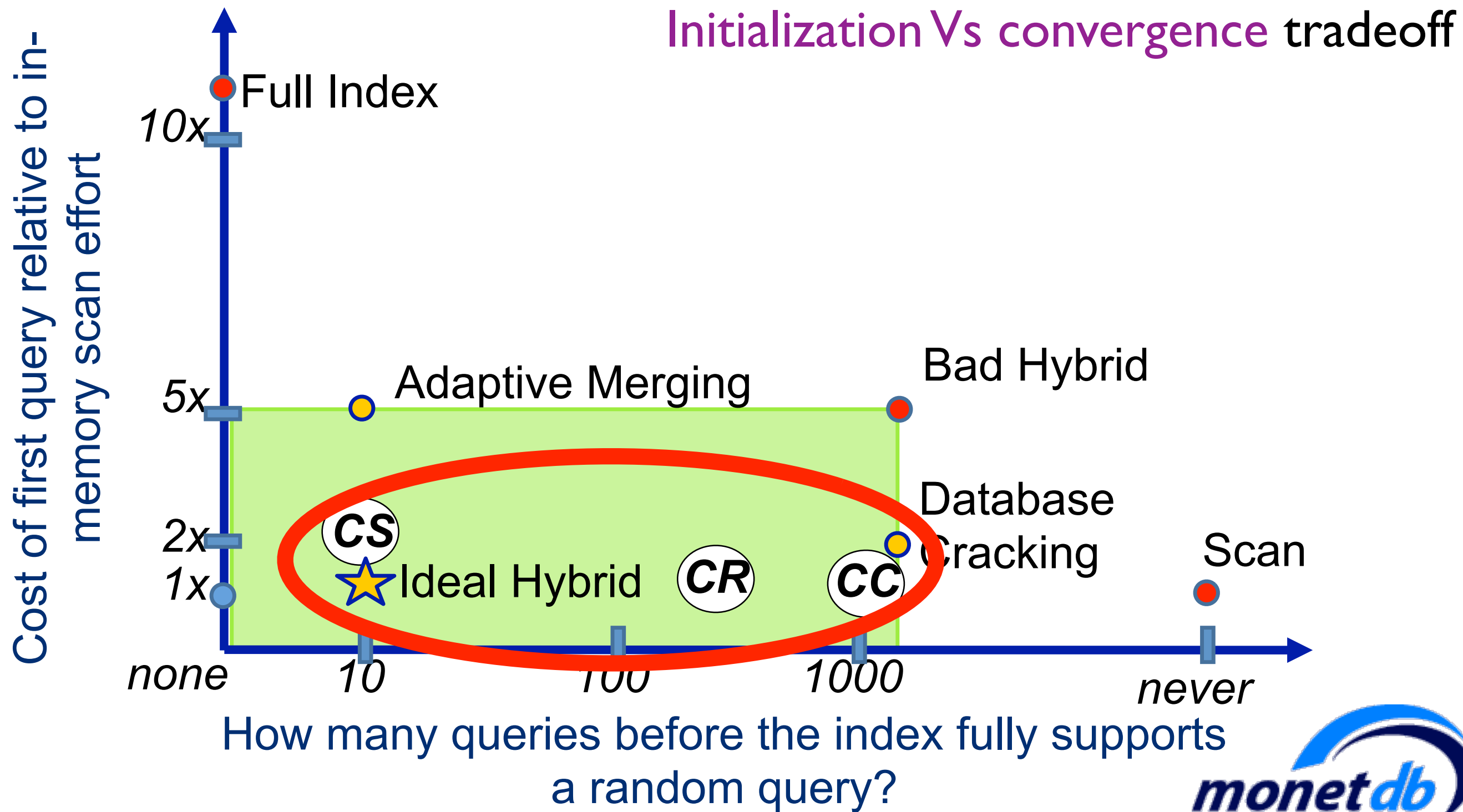


# Adaptive Indexing





# Adaptive Indexing



# Self-organizing Tuple Reconstruction in Column-stores

Stratos Idreos  
CWI Amsterdam  
The Netherlands  
idreos@cwi.nl

Marlin L. Kersten  
CWI Amsterdam  
The Netherlands  
mk@cwi.nl

Stefan Manegold  
CWI Amsterdam  
The Netherlands  
manegold@cwi.nl

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

Felix Haimm<sup>†</sup>  
\*National University of Singapore  
(haimm, ryap)@comp.nus.edu.sg

Panagiotis Karras<sup>‡</sup>  
CWI, Amsterdam  
idreos@cwi.nl

Roland H. C. Yap<sup>†</sup>  
Rutgers University  
karras@business.rutgers.edu

### ABSTRACT

Modern business applications and scientific databases call for increasingly dynamic data storage environments. Such environments are characterised 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 at continuously and automatically adapting 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 traity 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, overlooking the possibility of 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 properties lacking. Our extensive experimental study verifies that stochastic cracking maintains the desired properties of original database cracking while at the same time performs well with diverse realistic workloads.

### 1. INTRODUCTION

Database research has set to reexamine established assumptions in order to meet the new challenges posed by big data, scientific databases, highly dynamic, distributed, and multi-core CPUs.

\*Work supported by Singapore's NUS.

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 the copy is not made or distributed for profit or commercial gain, and that the ACM copyright notice and the title of the publication appear on the first page of any copy. Articles to be reproduced in the proceedings of the VLDB Conference must include the text of this permission notice. Articles from the VLDB Endowment Copyright © 2011 VLDB Endowment. All rights reserved.

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

Stratos Idreos<sup>†</sup> Stefan Manegold<sup>‡</sup>  
CWI, Amsterdam  
(stratos.idreos, stefan.manegold)@cwi.nl

Harumi Kuno<sup>§</sup> Goetz Graefe<sup>§</sup>  
HP Labs, Palo Alto  
(harumi.kuno, goetz.graefe)@hp.com

### ABSTRACT

Adaptive indexing is characterized by the partial creation and refinement of the index as side effects of query execution. Dynamic or shifting workloads may benefit from preliminary index structures (created on the columns and specific key ranges actually queried) — without incurring the cost of full index construction. The costs and benefits of adaptive indexing techniques should thus be compared in terms of initialization costs, the overhead imposed upon queries, and the rate at which the index converges to a state that is fully satisfied for a particular workload component. Based on an examination of database cracking and adaptive merging, which are two techniques for adaptive indexing, we propose a hybrid technique that has a low initialization cost and also converges rapidly. We find the strengths and weaknesses of database cracking and adaptive merging complementary. One has a relatively high initialization cost but converges rapidly. The other has a low initialization cost but converges relatively slowly. We analyze the sources of their respective strengths and explore the space of hybrid techniques. We have designed and implemented a family of hybrid algorithms in the context of a column-store, index system. Our experiments compare their behavior against database cracking and adaptive merging, as well as against the traditional index lookup and scan of unindexed data. We show that the new hybrids significantly improve over past methods while at least two of the hybrids come within the "ideal performance" mark in terms of both overhead per query and convergence to a final state.

### 1. INTRODUCTION

Contemporary index selection tools rely on monitoring database requests and their execution plans, occasionally invoking creation or removal of indexes on tables and views. In the context of dynamic workloads, such tools tend to suffer from the following weaknesses. First, the interval between monitoring and index creation exceeds the specific request patterns that trigger the case there is no benefit to those tools. Second, even if that is not the case, there is no index support during this interval. Data access during the monitoring interval neither benefits from nor incurs the creation costs, and eventual index creation imposes an additional

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 gain, and that the ACM copyright notice and the title of the publication appear on the first page of any copy. Articles to be reproduced in the proceedings of the VLDB Conference must include the text of this permission notice. Articles from this volume were included in present year results at The VLDB Endowment's New York In-Depth Data Base, August 29th - September 3rd 2011, Seattle, Washington. Hybrid adaptive indexing systems, and find its strengths and weaknesses of the two approaches complementary. As shown in Figure 1, adaptive merging has a relatively high initialization cost and converges rapidly, while database cracking has a low initialization cost but converges relatively slowly. The hybrid algorithms in this paper thus represent a step towards adaptive indexing with database cracking and adaptive merging occupying the borders of this space. We recognize the opportunity for an ideal hybrid adaptive indexing technique that sits in the figure, that incurs a low initialization cost yet also converges quickly

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 [1].

**Physical Design.** Good performance in database systems largely relies on proper *partitioning 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 arbitrary 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 be changed by the time we finish tuning; (b) there may be no time to finish tuning properly; and (c) there is no indexing at all.

**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 numerically applied for boosting the performance of the select operator [16], maintenance operators [17], and ad-hoc query optimization [18]. In addition, more recently these ideas have been extended to exploit a partition-tree (the logs [10, 11, 12]).

**Workload Robustness.** Nevertheless, existing cracking schemes have not deeply considered the particular way in which they interact with queries as a hint on how to organize the data. In fact, they have adopted a simple interpretation, in which a select operator is taken to describe a range of the data that a *disruptive* cracker should use to provide easy access to for future queries; the remainder of the data remains non-indexed until a query expresses inter-

## Database Cra

Stratos Idreos  
CWI Amsterdam  
The Netherlands  
idreos@cwi.nl

Marlin L. Kersten  
CWI Amsterdam  
The Netherlands  
mk@cwi.nl

### CT

CT provides a non-discriminative navigational interface to localize tuples of interest. It maintains a cache of tuples from database updates. In this paper, we present a complementary approach, addressing indexing as part of query processing using continuous reorganization, i.e., cracking the database into pieces. The motivation is that by automatically taking the way users request it, we can achieve fast, much desired self-organized behavior. As the first mature cracking architecture and re-implementation of cracking in the context of a relational system, it led to a minor enhancement of the algebra kernel, such that cracking could be implemented without incurring too much processing overhead. We illustrate the ripple effect of dynamic indexing on the query plans derived by the SQL optimizer and results obtained are full-scale of reduction in system complexity. We show that systems are able to self-organize based on incremental, high class performance benefits. This behavior is observed when the user focus is randomly shifting to

design  
era  
y-  
hibid  
nd  
ac  
re-  
red  
in  
a  
own  
down  
built  
change  
TU

present  
relative  
about the

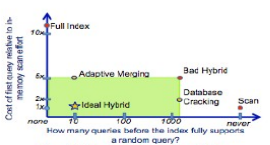


Figure 1: Adaptive Indexing Research Space. This graph illustrates the trade-off between initialization cost and convergence speed for different indexing techniques. The x-axis represents the number of queries required for the index to be fully satisfied for a random query, with markers at 100, 1000, and 10000. The y-axis represents the cost of full index construction relative to a full index, also with markers at 100, 1000, and 10000. The 'Ideal Hybrid' curve shows the lowest cost and fastest convergence. 'Database Cracking' shows intermediate performance, while 'Adaptive Merging' has the highest cost and slowest convergence.

### ABSTRACT

Column-stores gained popularity as a promising physical design alternative. Each attribute of a relation is physically stored as a separate column of wide tables at a time. Each required attribute. The overhead incurred is on-the-fly tuple reconstruction for multi-attribute queries. Each tuple reconstruction is a join of two columns based on tuple IDs, making it a significant cost element. The ultimate physical design is to have multiple pre-computed copies of each base table such that tuples are already appropriately organized in multiple different orders across the various columns. This requires the ability to predict the workload, life time to prepare, and infrequent updates.

In this paper, we propose a novel design, *partial adaptive cracking*, that minimizes the tuple reconstruction cost in a self-organizing way. It achieves performance similar to using pre-sorted data, but without requiring the heavy initial pre-sorting step itself. Instead, it handles dynamic, unpredictable workloads with no idle time and frequent updates. Auxiliary dynamic data structures, called *cracker maps*, provide a direct mapping between pairs of attributes used together in queries for tuple reconstruction. A map is continually physically reorganized as an integral part of query evaluation, providing faster and reduced data access over future queries. To enable flexible and self-organizing behavior in storage-limited environments, maps are materialized only partially as demanded by the workload. Each map is a collection of separate chains that are individually re-organized, dropped or recreated as needed. We implemented partial adaptive cracking in an open-source column-store. A detailed experimental analysis demonstrates that it brings significant performance benefits for multi-attribute queries.

**Categories and Subject Descriptors:** H.2 [DATABASE MANAGEMENT]: Physical Design - Systems  
**General Terms:** Algorithms, Performance, Design  
**Keywords:** Database Cracking, Self-organization

Permission to make digital or hard copies of all or part of this work for

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

Goetz Graefe  
Hewlett-Packard Laboratories  
1501 Page Mill Road  
Palo Alto, CA 94304

Harumi Kuno  
Hewlett-Packard Laboratories  
1501 Page Mill Road  
Palo Alto, CA 94304

### Abstract

In a relational data warehouse with many tables, the number of possible and promising indexes exceeds human comprehension and requires automatic index tuning. While monitoring and reactive index tuning have been proposed, adaptive indexing focuses on adapting the physical database layout for and by actual queries.

"Database cracking" is one such technique. Only if and when a column is used in query predicates, an index for the column is created; and only if and when a key range is queried, the index is optimized for this key range. The effect is akin to a sort that is adaptive and incremental. This sort is, however, very inefficient, particularly when applied on block-access devices. In contrast, traditional index creation sorts data with an efficient merge sort optimized for block-access devices, but it is neither adaptive nor incremental.

We propose *adaptive merging*, an adaptive, incremental, and efficient technique for index creation. Index optimization focuses on key ranges used in actual queries. The resulting index adapts more quickly to new data and to new query patterns than database cracking. Sort efficiency is comparable to that of traditional B-tree creation. Nonetheless, the new technique promises better query performance than database cracking, both in memory and on block-access storage.

### Categories and subject descriptors

E.2 Data storage representations — arrays, sorted trees.

### Keywords

Database index, adaptive, automatic, query execution.

### 1 Introduction

In a relational data warehouse with a hundred tables and a thousand columns, billions of indexes are possible, in particular if partial indexes, indexes on computed columns,

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 gain, and that the ACM copyright notice and the title of the publication appear on the first page of any copy. Articles to be reproduced in the proceedings of the VLDB Conference must include the text of this permission notice. Copyright © 2010 ACM 978-1-60558-943-9/10/0001...\$10.00.

### 1. INTRODUCTION

A prime feature of column-stores is to provide improved performance over row-stores in the case that workloads require only a few attributes of wide tables at a time. Each relation  $R$  is physically stored as a set of columns; one column for each attribute of  $R$ . This way, a query needs to load only the required attributes from each relation. This happens at the expense of requiring explicit (partial) tuple reconstruction in case multiple attributes are required. Each tuple reconstruction is a join between two columns based on tuple IDs/positions and becomes a significant cost component in column-stores especially for multi-attribute queries [2, 6, 10]. Wherever possible, position-based join-making and tuple reconstruction are exploited: For each relation  $R$ , in a query plan  $q$ , a column-store needs to perform at least  $N_c - 1$  tuple reconstruction operations for  $R$ , within  $q$ , given that  $N_c$  attributes of  $R$  participate in  $q$ .

Column-stores perform tuple reconstruction in two ways [2]. With early tuple reconstruction, the required attributes are joined together as early as possible, i.e., while the columns are loaded; leaving  $N_c - 2$  processing to evaluate the query. On the other hand, late tuple reconstruction exploits the column-store architect to the maximum. During query processing, "reconstruction" merely refers to getting the attribute value by vector by etc. as not from  $R$  but from the query plan. This approach allows the query engine to exploit CPU- and cache-optimized vector-like operator implementations throughout the whole query evaluation.  $N_c - 1$  tuples are formed only once the final result is delivered.

Like most modern column-stores [2, 4, 15], we focus on late reconstruction. Comparing early and late reconstruction, the overhead analysis in [2] observes that the latter incurs the overhead of reconstructing a column more than once, in case it occurs more than once in a query. Furthermore, exploiting sequential access patterns during reconstruction is not always possible since many operators (joins, group by, order by etc.) are not fully order preserving.

The ultimate access pattern is to have multiple copies of the query plan. This approach allows the query engine to exploit CPU- and cache-optimized vector-like operator implementations throughout the whole query evaluation.  $N_c - 1$  tuples are formed only once the final result is delivered.

Like most modern column-stores [2, 4, 15], we focus on late reconstruction. Comparing early and late reconstruction, the overhead analysis in [2] observes that the latter incurs the overhead of reconstructing a column more than once, in case it occurs more than once in a query. Furthermore, exploiting sequential access patterns during reconstruction is not always possible since many operators (joins, group by, order by etc.) are not fully order preserving.

The ultimate access pattern is to have multiple copies of the query plan. This approach allows the query engine to exploit CPU- and cache-optimized vector-like operator implementations throughout the whole query evaluation.  $N_c - 1$  tuples are formed only once the final result is delivered.

## Poor Man's Sort!

Stratos Idreos  
Harvard University  
idreos@seas.harvard.edu

Permission to make digital or hard copies of all or part of this work for

design  
era  
y-  
hibid  
nd  
ac  
re-  
red  
in  
a  
own  
down  
built  
change  
TU

present  
relative  
about the

Costs of Database Operations  
If Cracking is fully sorted data, its costs are those of fully sorting the data. With recent (parallel) sorting algorithms [7], however, basing indexing. To illustrate this, Figure 1 compares the respective operations on 64-bit integers on a 4-Core Sandy Bridge CPU, off-the-shelf (Presto!) Merge-sort implements an algorithm more expensive than a (quasi) I/O bound, only three times as expensive as MergeSort's of the "class". Cracking implementation (by read and write the same amount of data, but costs. The performance difference must, operational costs: Cracking is, unlike Scan- however, implemented with the underlying hardware in the (roughly) I/O bound.

This hypothesis, we make the following contributions: indicate an in-depth study of the contributing performance of the "class". Cracking implementation (by read and write the same amount of data, but costs. The performance difference must, operational costs: Cracking is, unlike Scan- however, implemented with the underlying hardware in the (roughly) I/O bound.

on the findings, we develop a number of optimization on "standard" techniques like prediction, version and manually implemented data parallelism using instructions.

develop two different parallel algorithms that exploit thread parallelism to make use of multiple CPU cores. We also evaluate all developed algorithms on a number of test cases ranging from low-end desktop machines to high-end servers. GNU libatomic - Version 4.8.2

One approach is to focus on enabling very fast scans, e.g., using shared scans and columnar storage formats, an approach suitable to high-bandwidth high-latency devices such as traditional disk drives and disk arrays. Low-latency database storage such as flash memory will likely re-energize research into index-based query processing.

Another approach is to tune indexes in response to the actual workload. Contemporary index selection tools rely on monitoring database requests and their execution plans, occasionally invoking creation or removal of indexes on tables and views. Such tools tend to suffer from three weaknesses. First, the interval between monitoring and index creation exceeds the duration of a specific request pattern; in which case there is no benefit to those tools. Second, even if that is not the case, there is no index support during this interval, so data access during the interval is wasted with respect to index creation, and eventual index creation imposes an additional load that interferes with query execution on other tables. Last but not least, traditional indexes on tables cover all rows equally, even if some rows are needed often and some never. For example, recent business transactions are more often than those years ago, and extreme price fluctuations are more interesting than stable prices, etc. Even where it is possible to limit an index, e.g., using a partial index, or a materialized view, it is often difficult or impossible to predict the key ranges to focus on.

Database cracking [KM07a, KM07b] has pioneered focused, incremental, automatic optimization of the representation of a data collection — the more often a key range is queried, the more its representation is optimized. This optimization occurs entirely automatically, as a side effect of queries over key ranges not yet fully optimized.

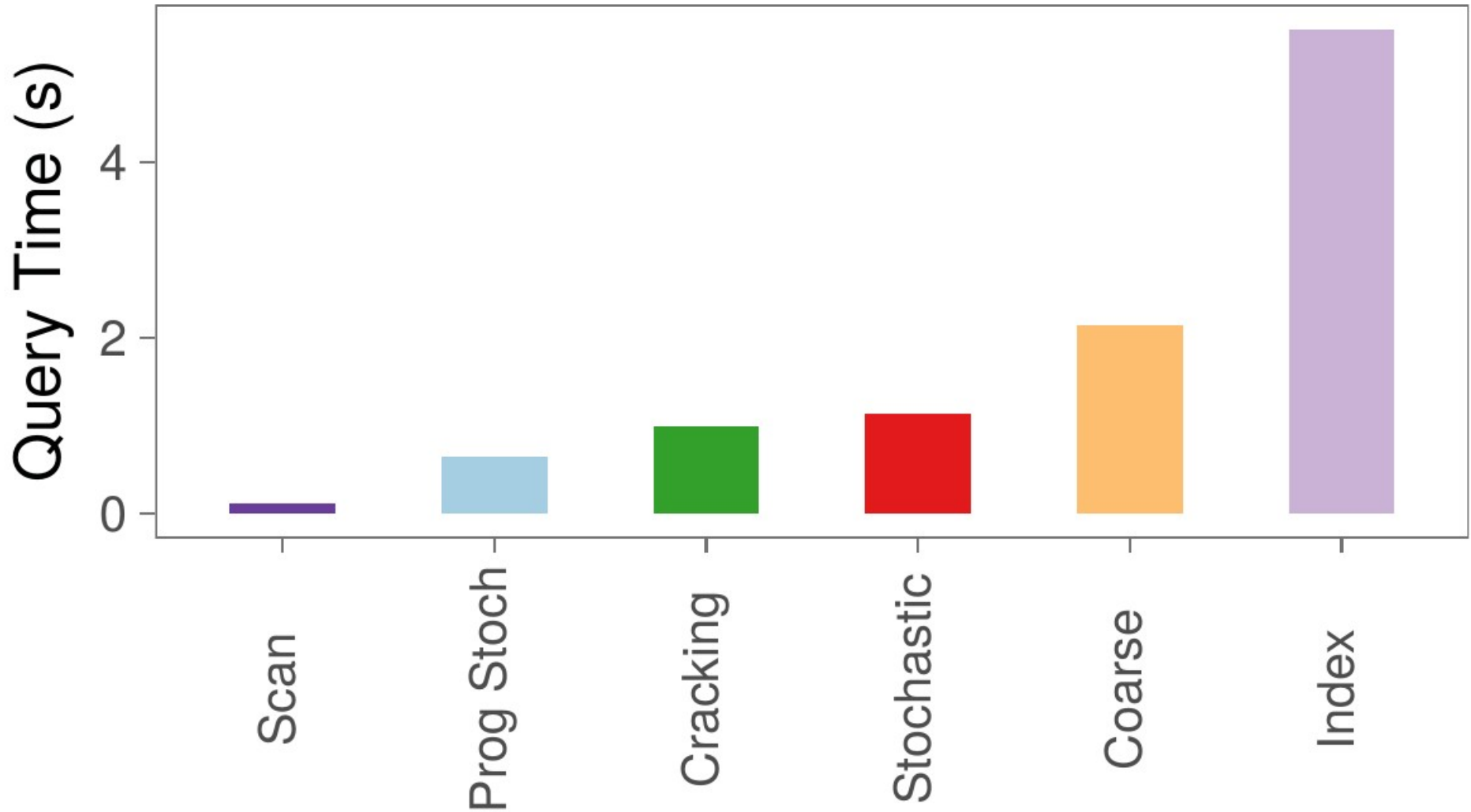
Column domain and storage array  
c g j  
r s u y  
Figure 1: A column store partitioned by database cracking. For example, after the column store illustrated in Figure 1 has been queried with range boundary values c, g, m, s, and u, all key values below c have been assigned to

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 gain, and that the ACM copyright notice and the title of the publication appear on the first page of any copy. Articles to be reproduced in the proceedings of the VLDB Conference must include the text of this permission notice. Copyright © 2010 ACM 978-1-60558-943-9/10/0001...\$10.00.

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 gain, and that the ACM copyright notice and the title of the publication appear on the first page of any copy. Articles to be reproduced in the proceedings of the VLDB Conference must include the text of this permission notice. Copyright © 2010 ACM 978-1-60558-943-9/10/0001...\$10.00.



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



# Progressive Indexing

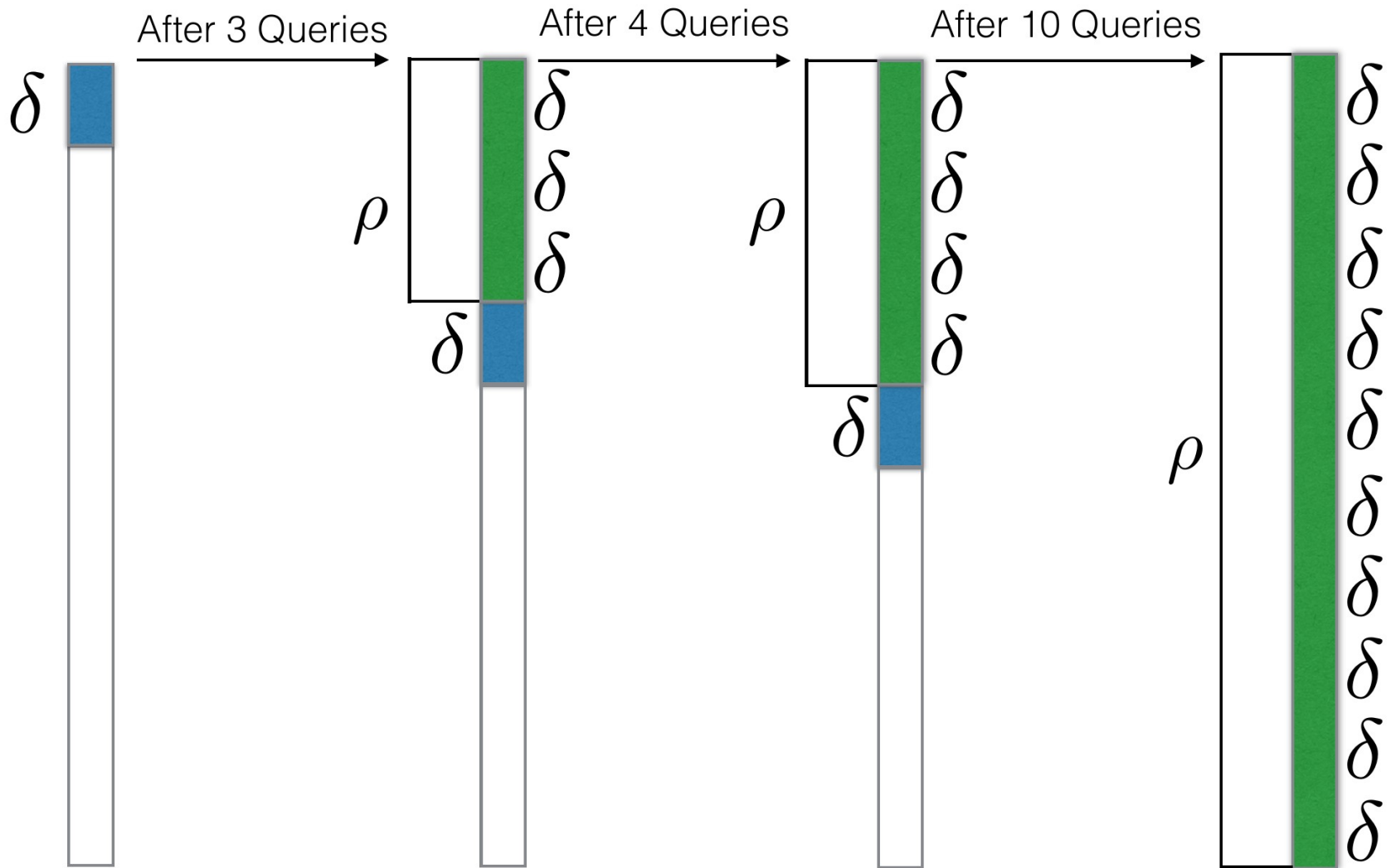
Can we / how to:

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

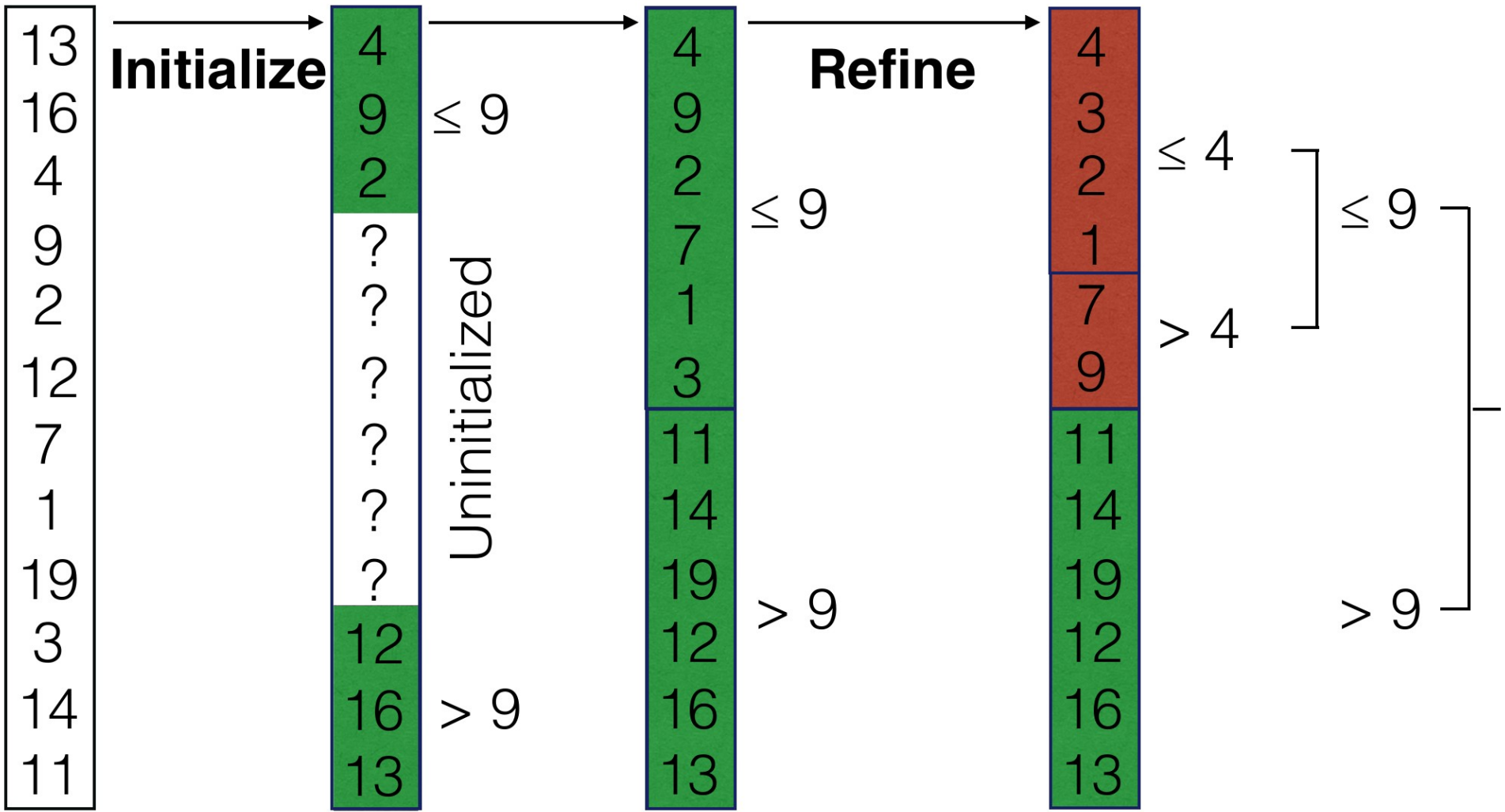
Yet unexplored “dimensions”:

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

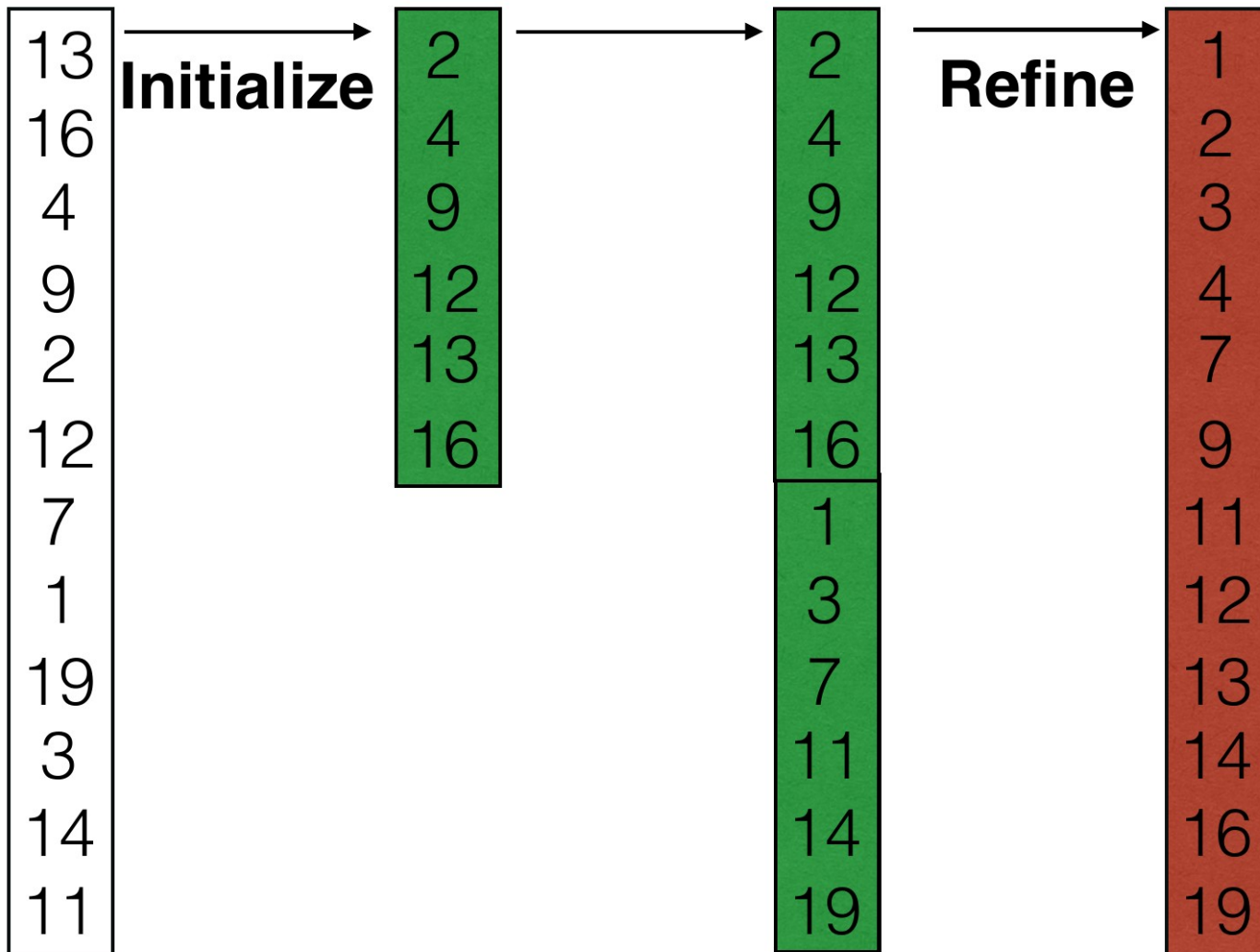
# Progressive Indexing



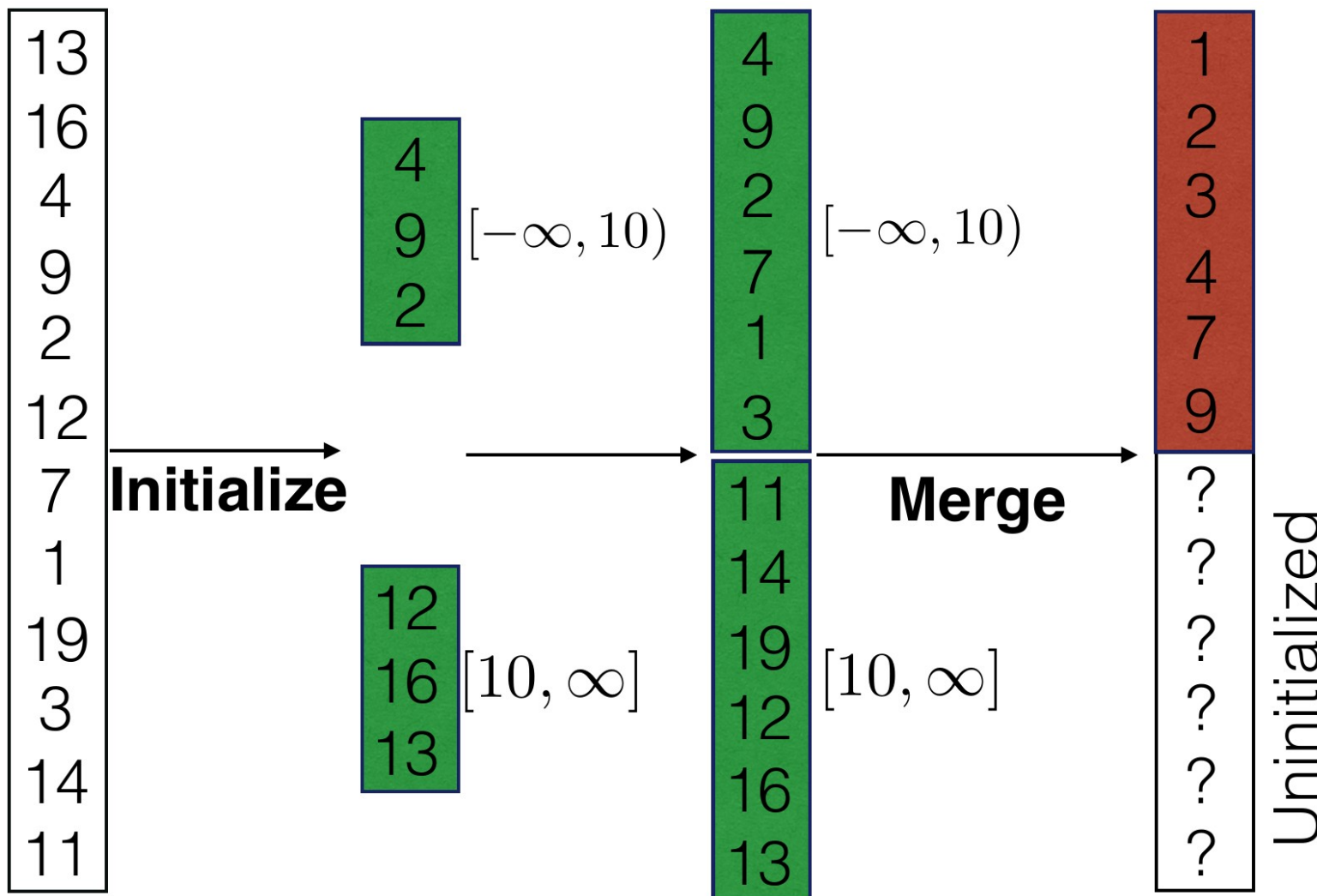
# Progressive Quick-Sort



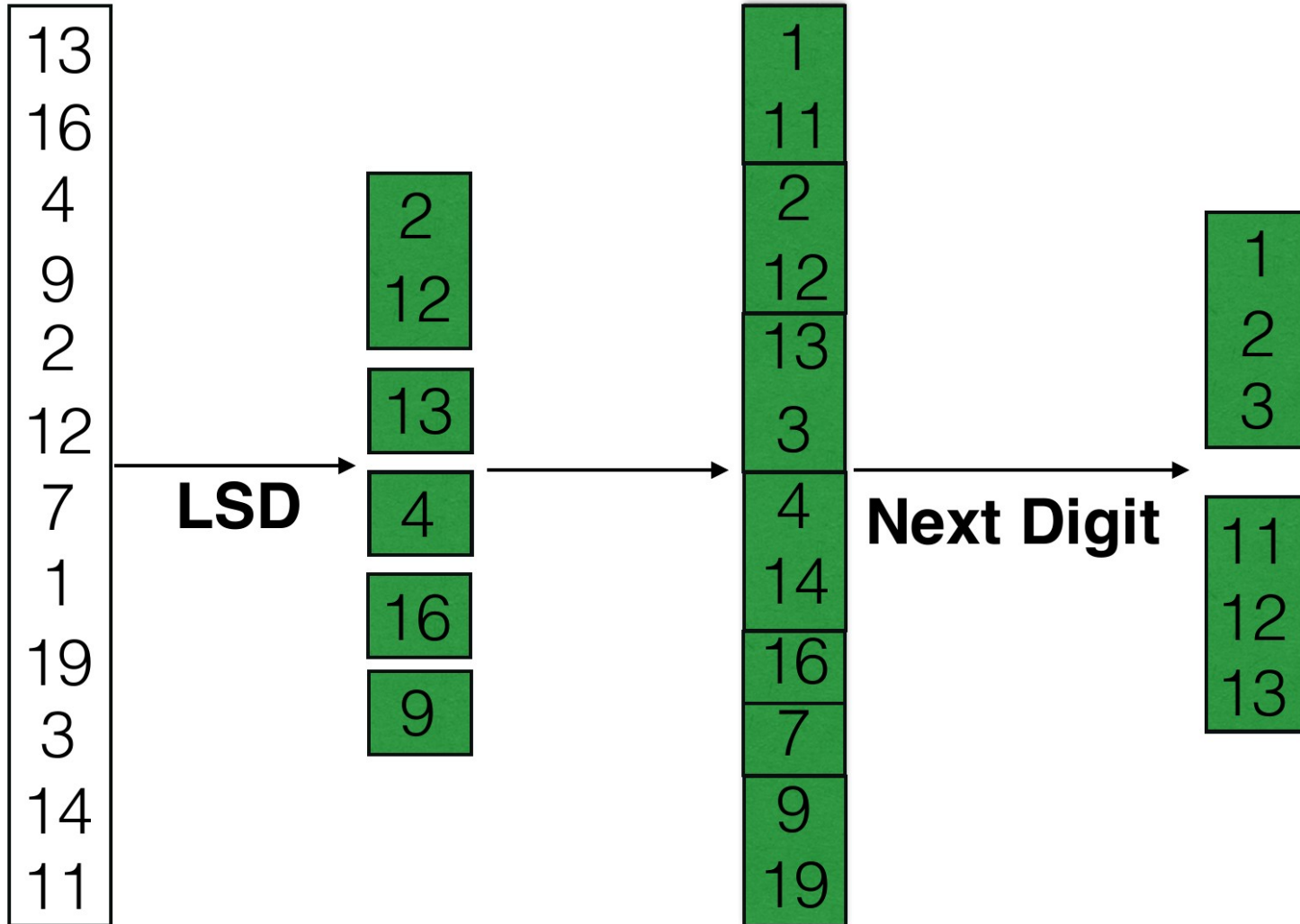
# Progressive Merge-Sort



## Progressive Bucket-Sort



# Progressive Radix-Sort

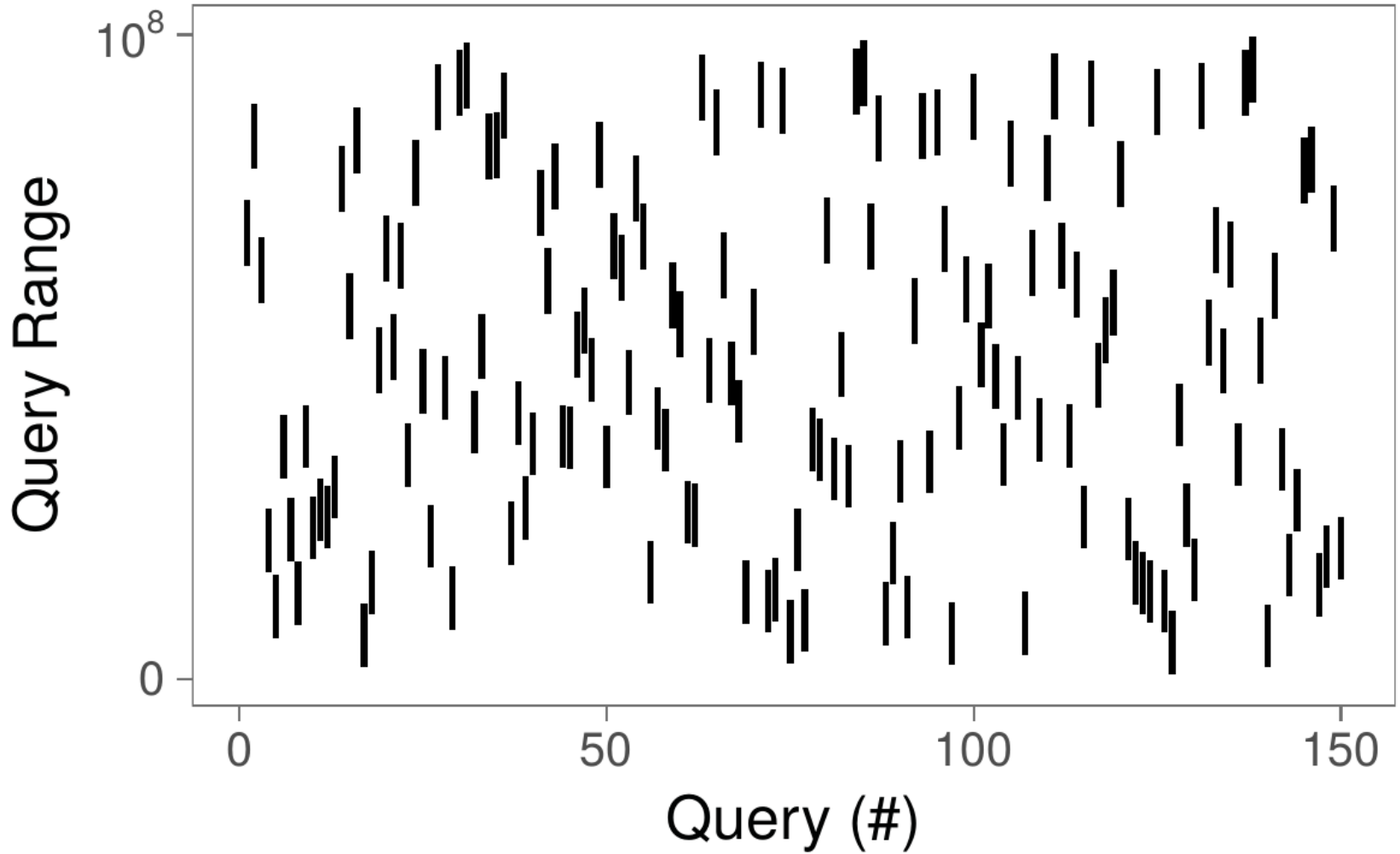


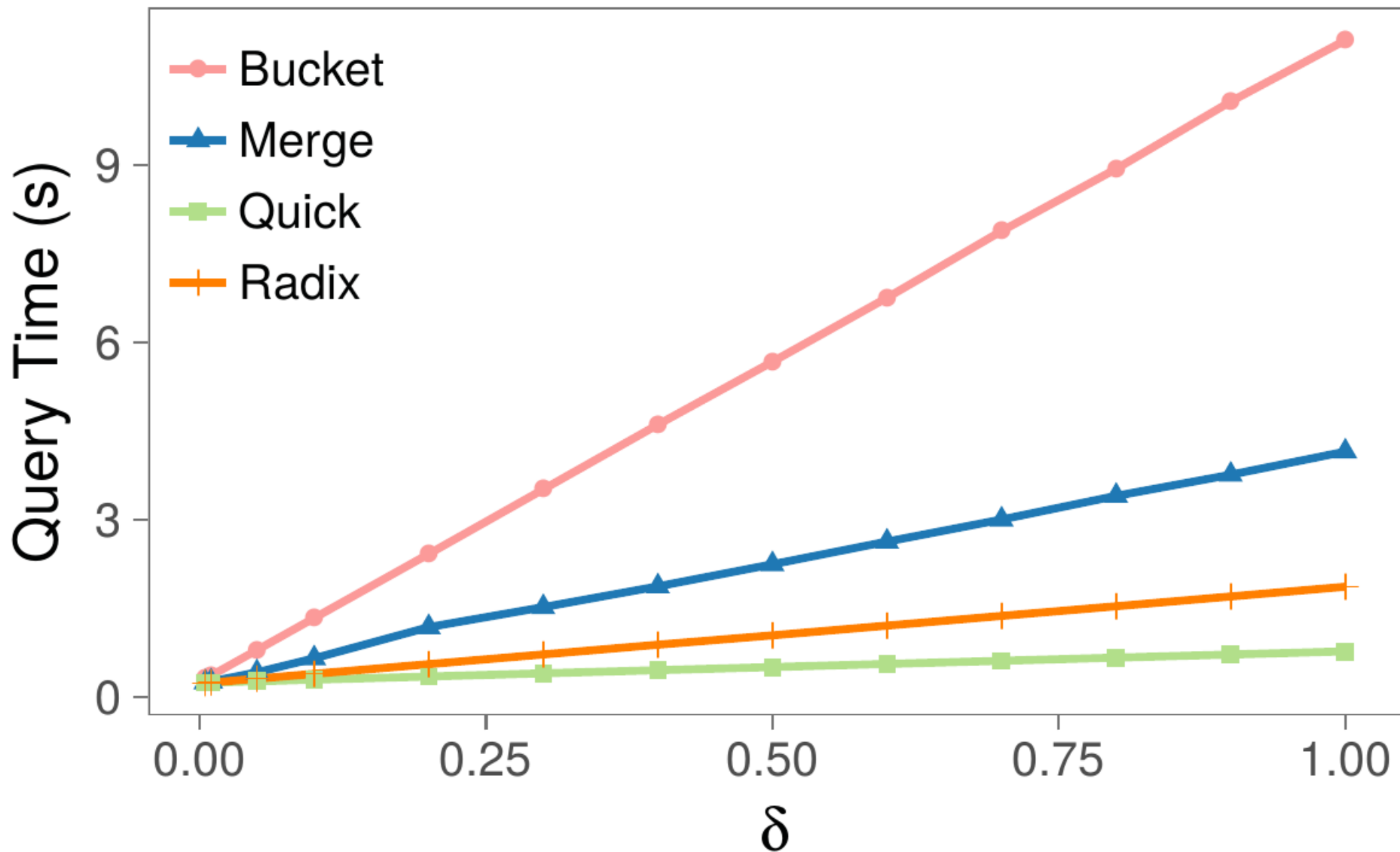
# Experimental Setup

- 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

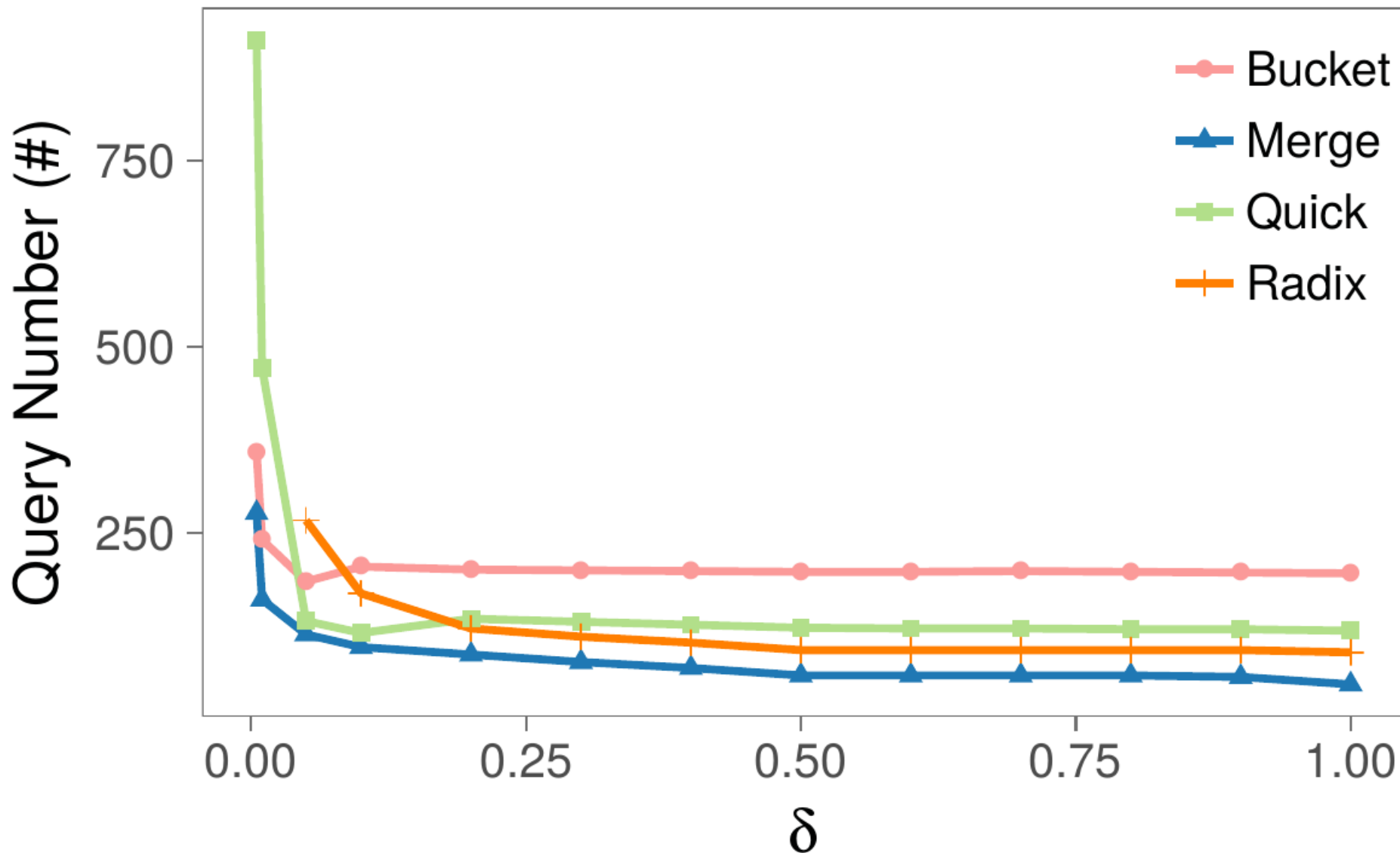


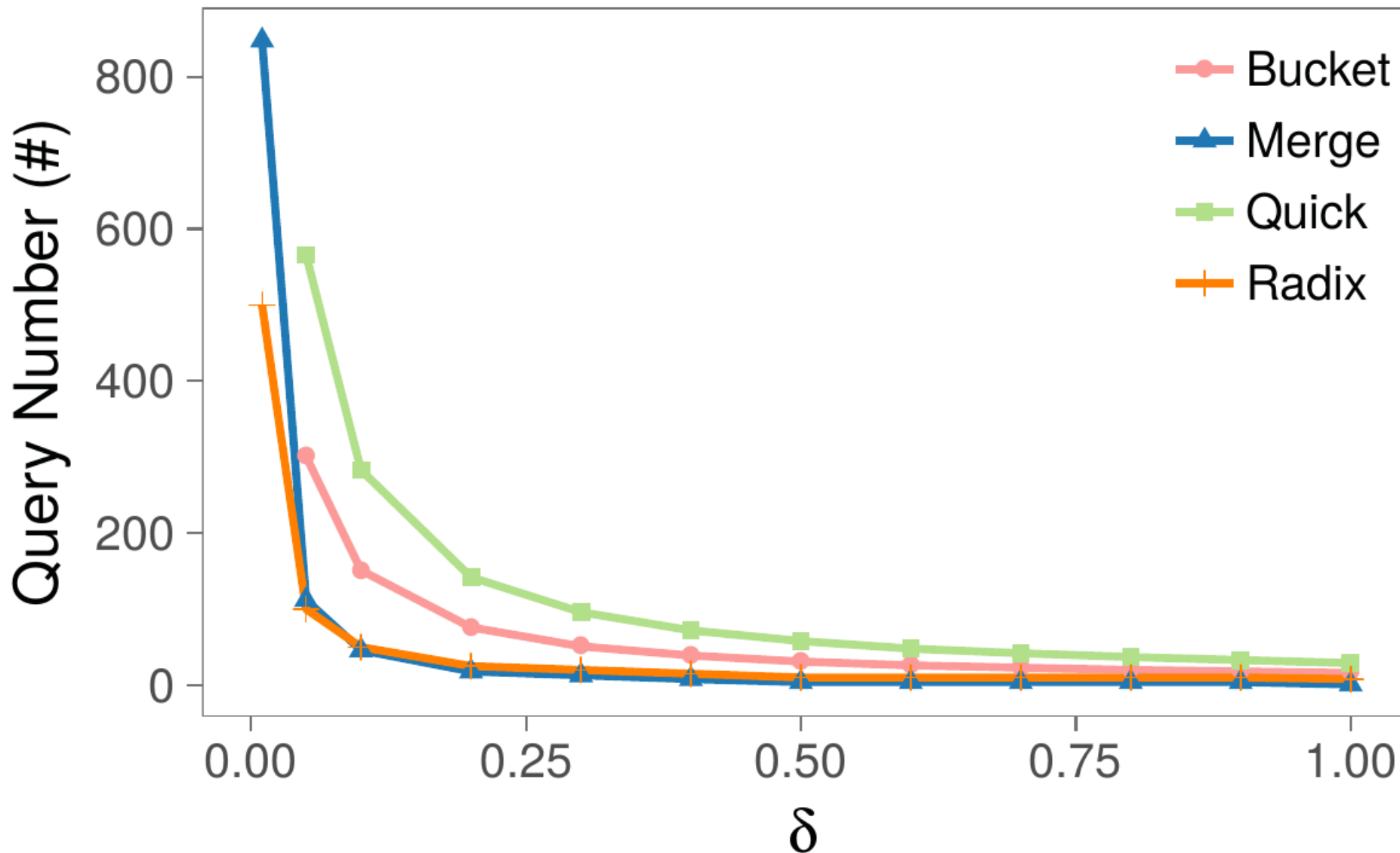
# Random Workload



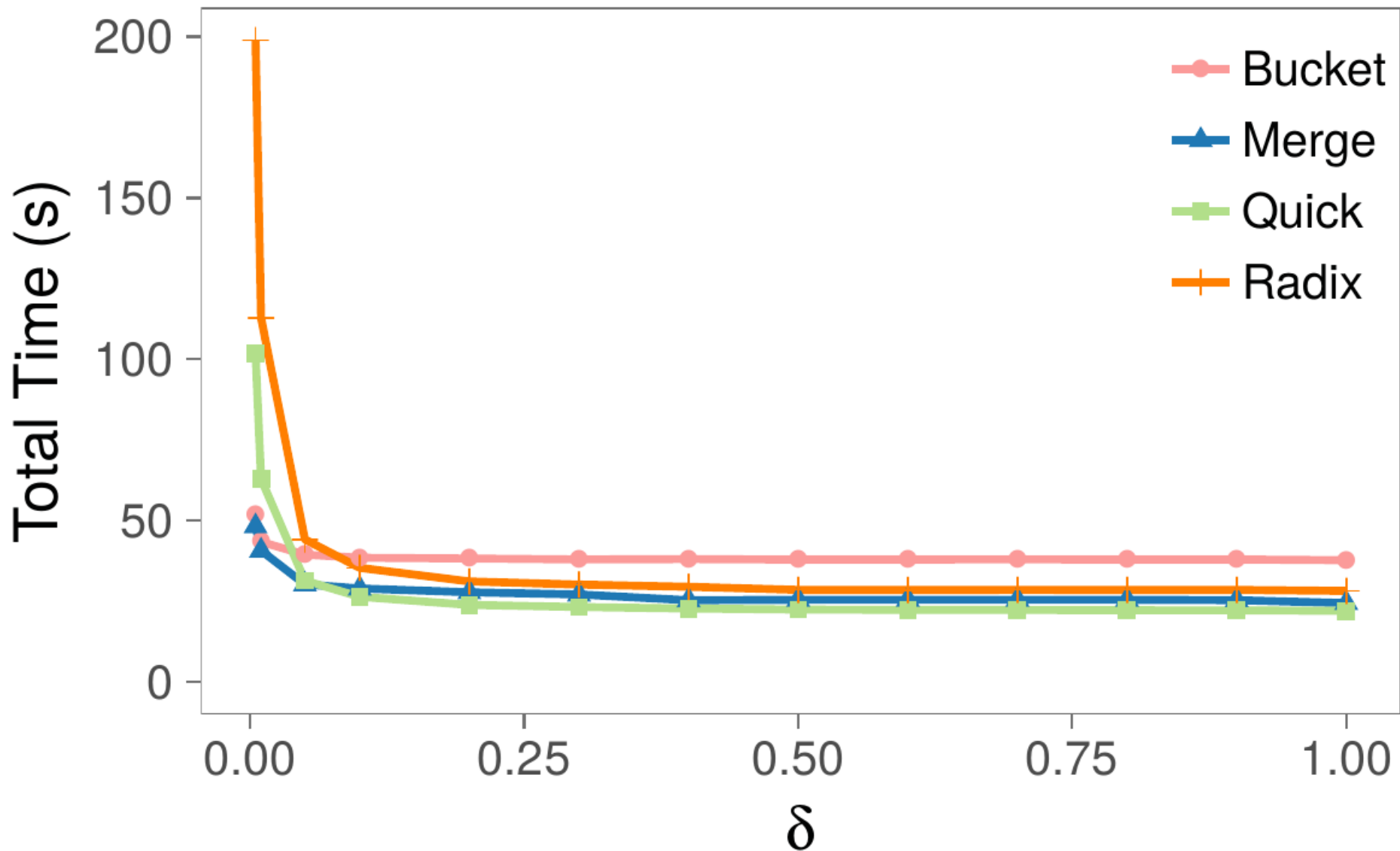
Varying  $\delta$ :  
1<sup>st</sup> Query Cost

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





# Varying $\delta$ : Entire Workload Cost

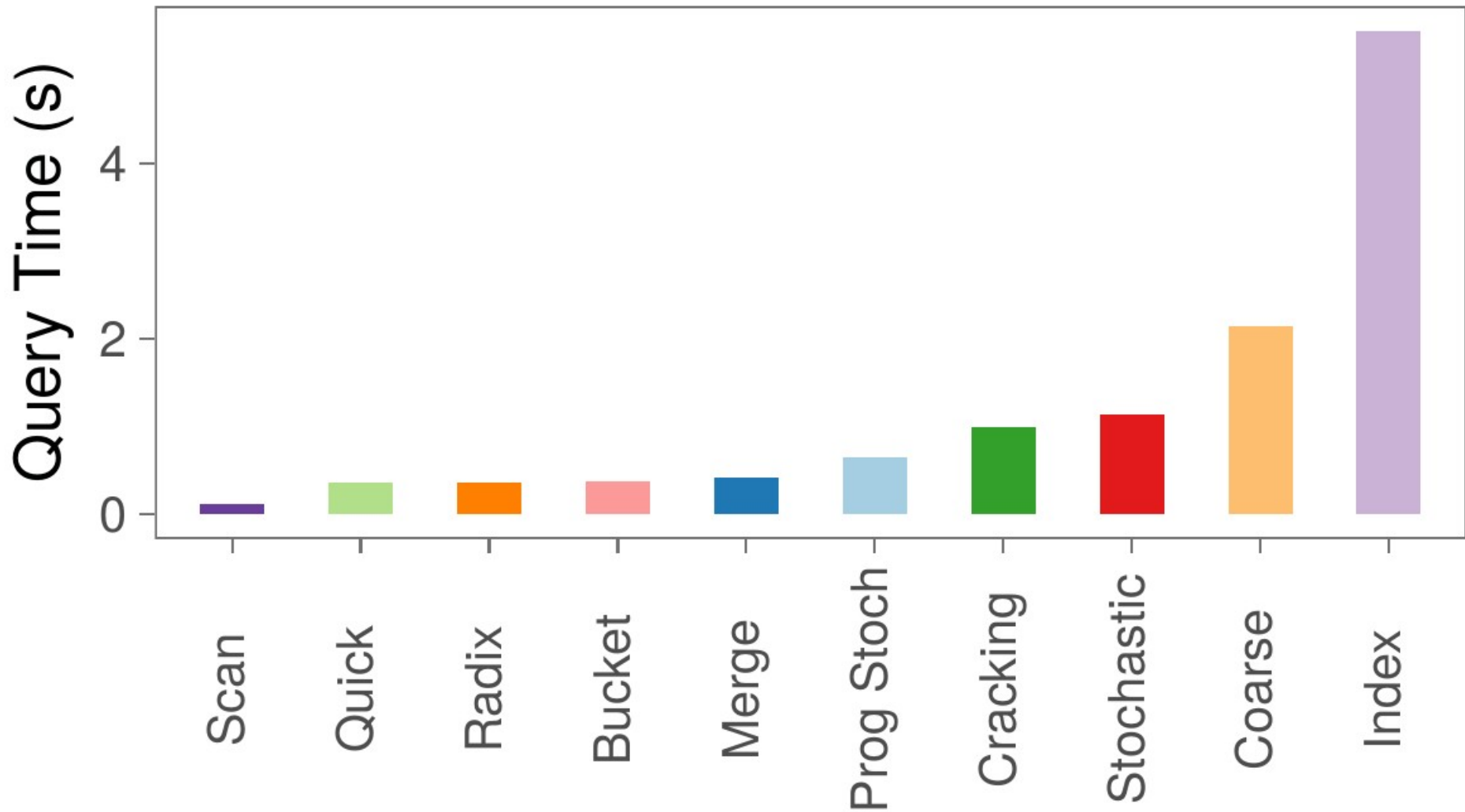


# Chosen $\delta$ :

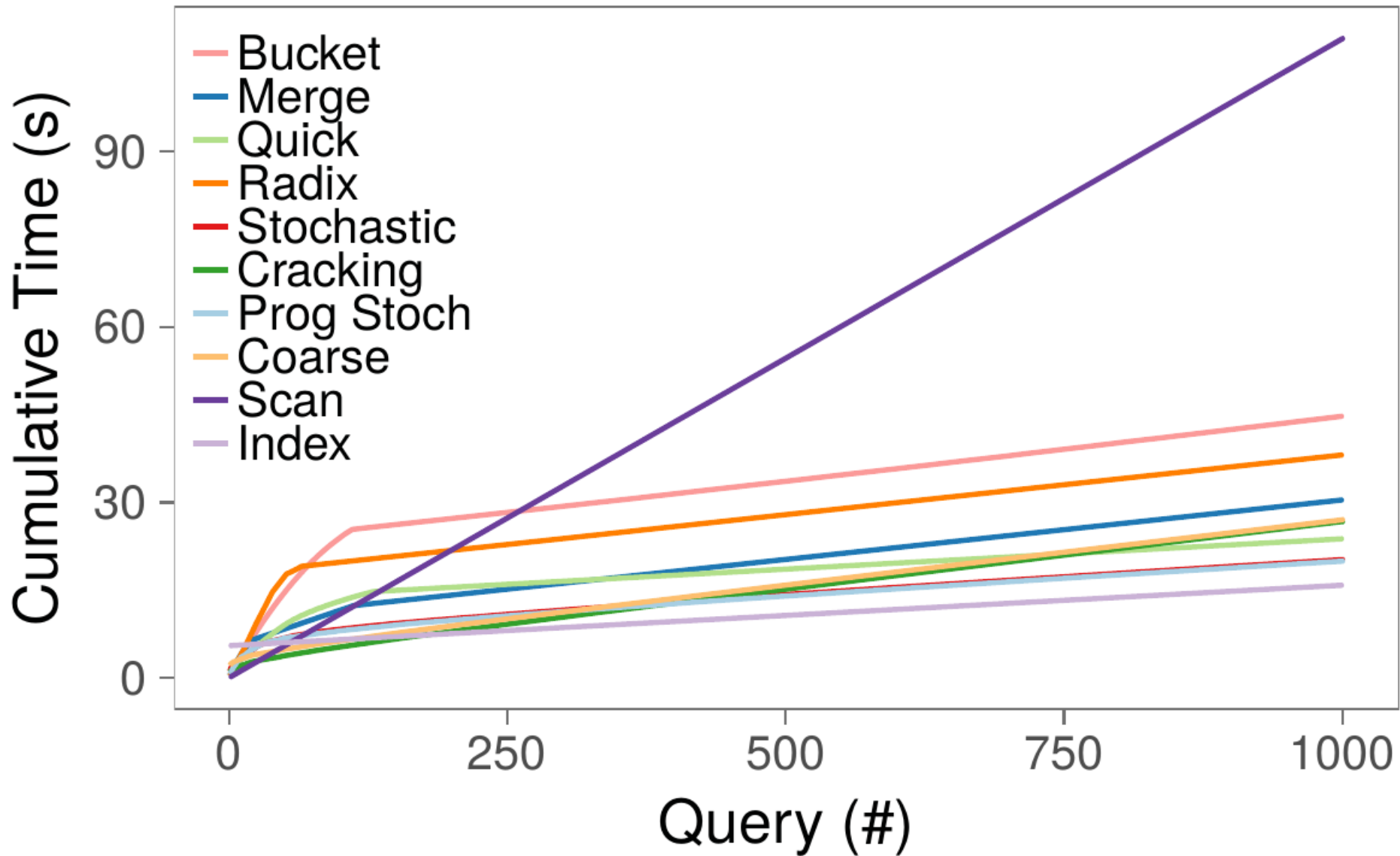
## 1<sup>st</sup> Query $\approx$ 2x Scan

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

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

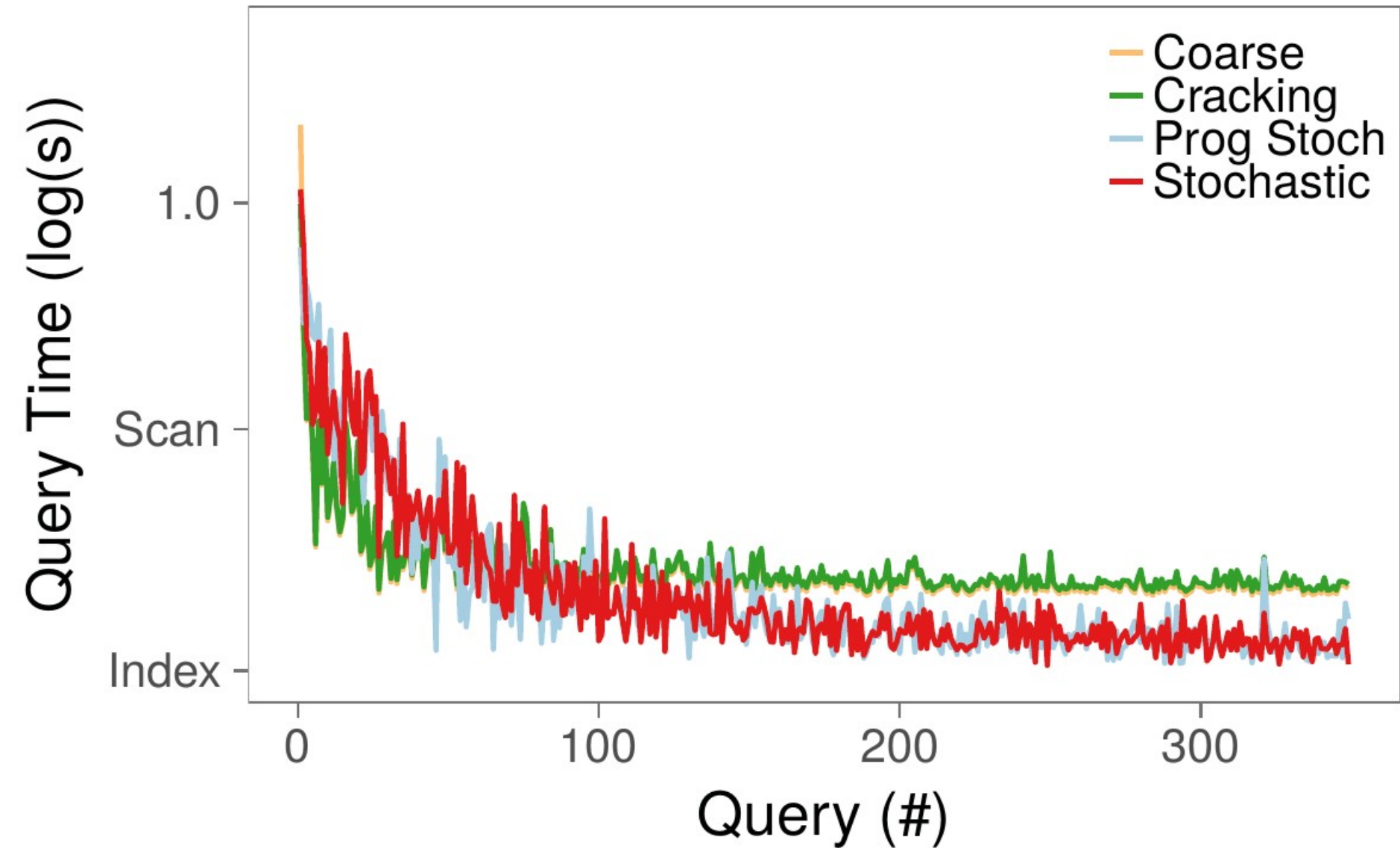


# Comparison: Entire Workload

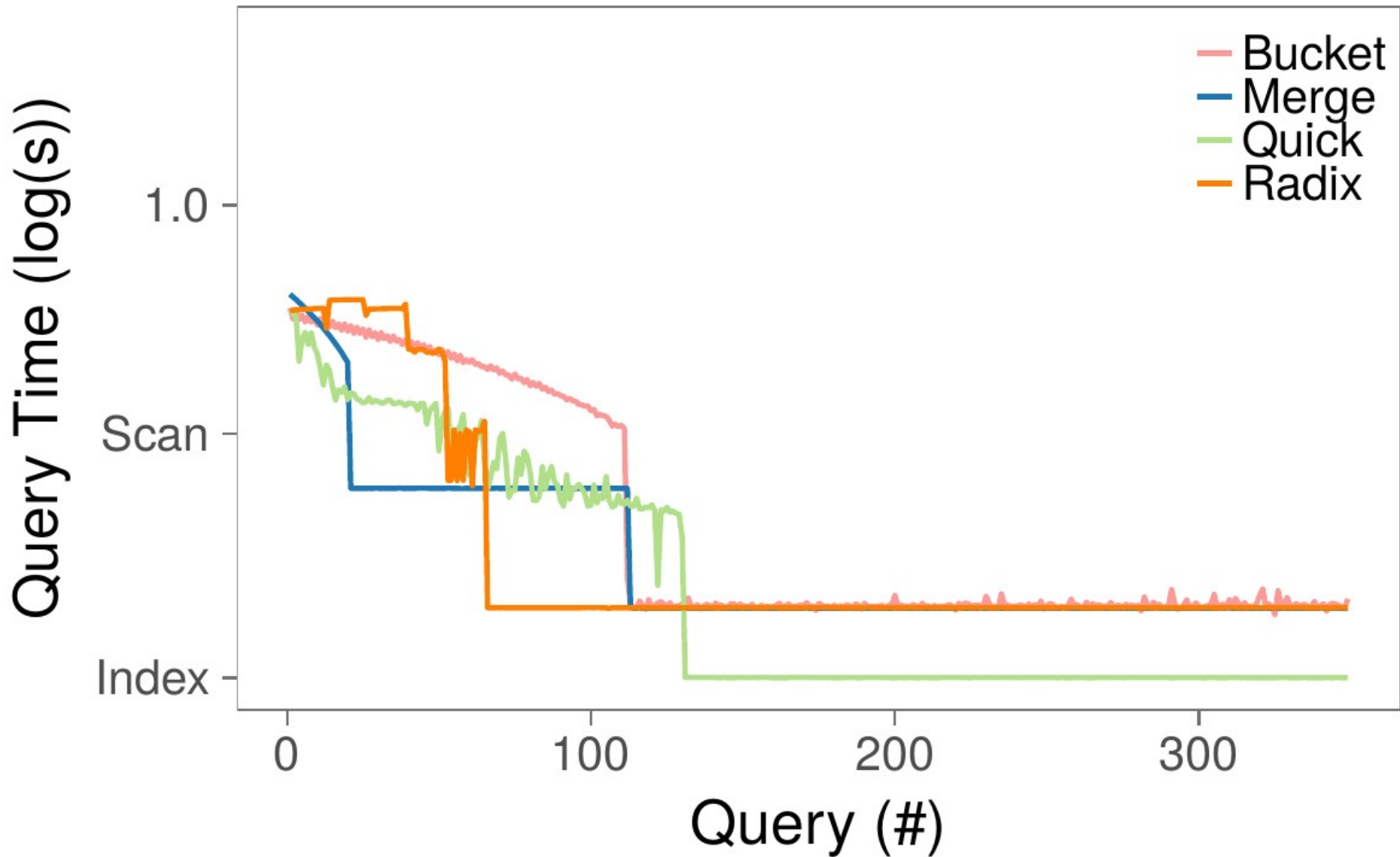




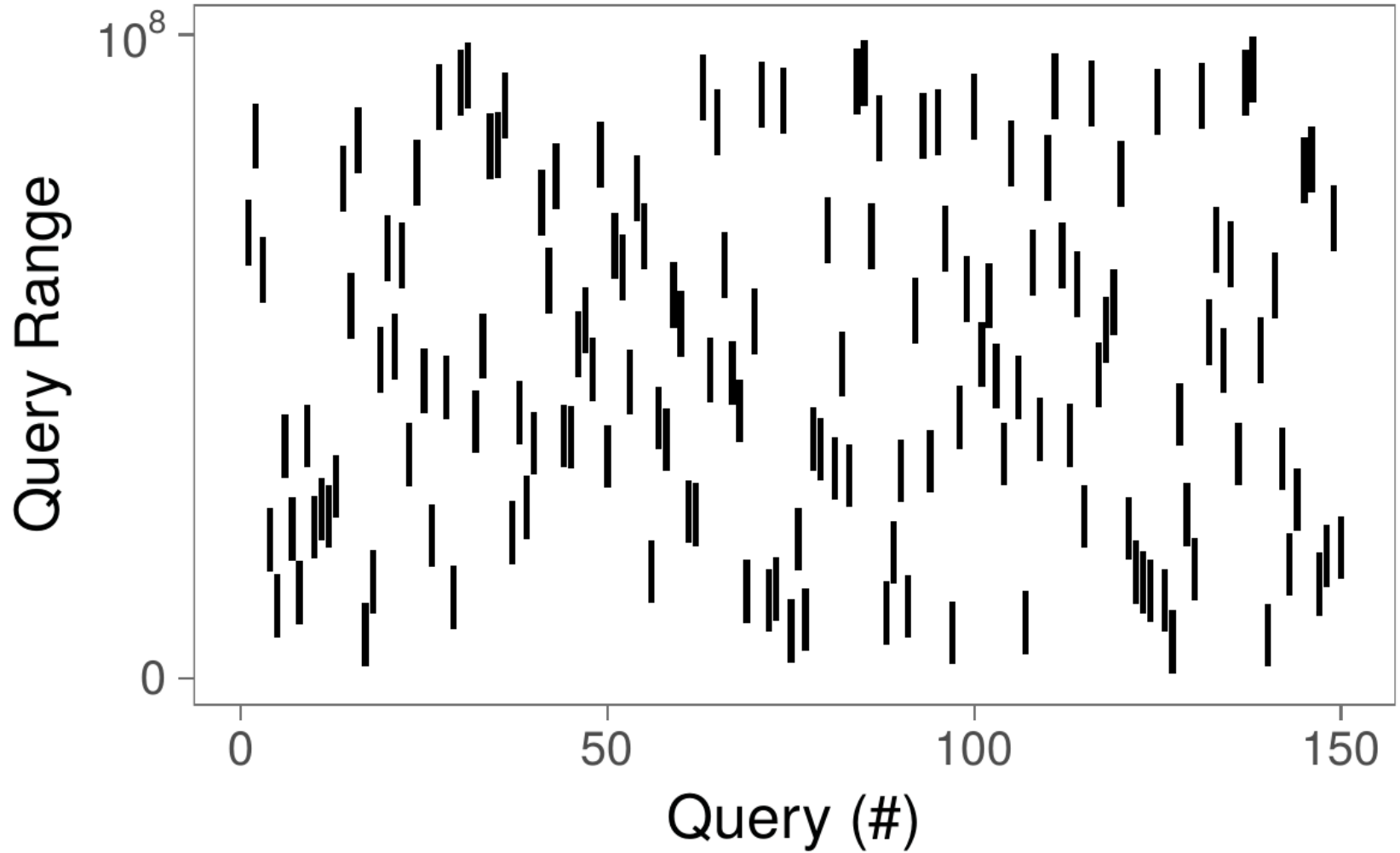
# Comparison: Adaptive Indexing



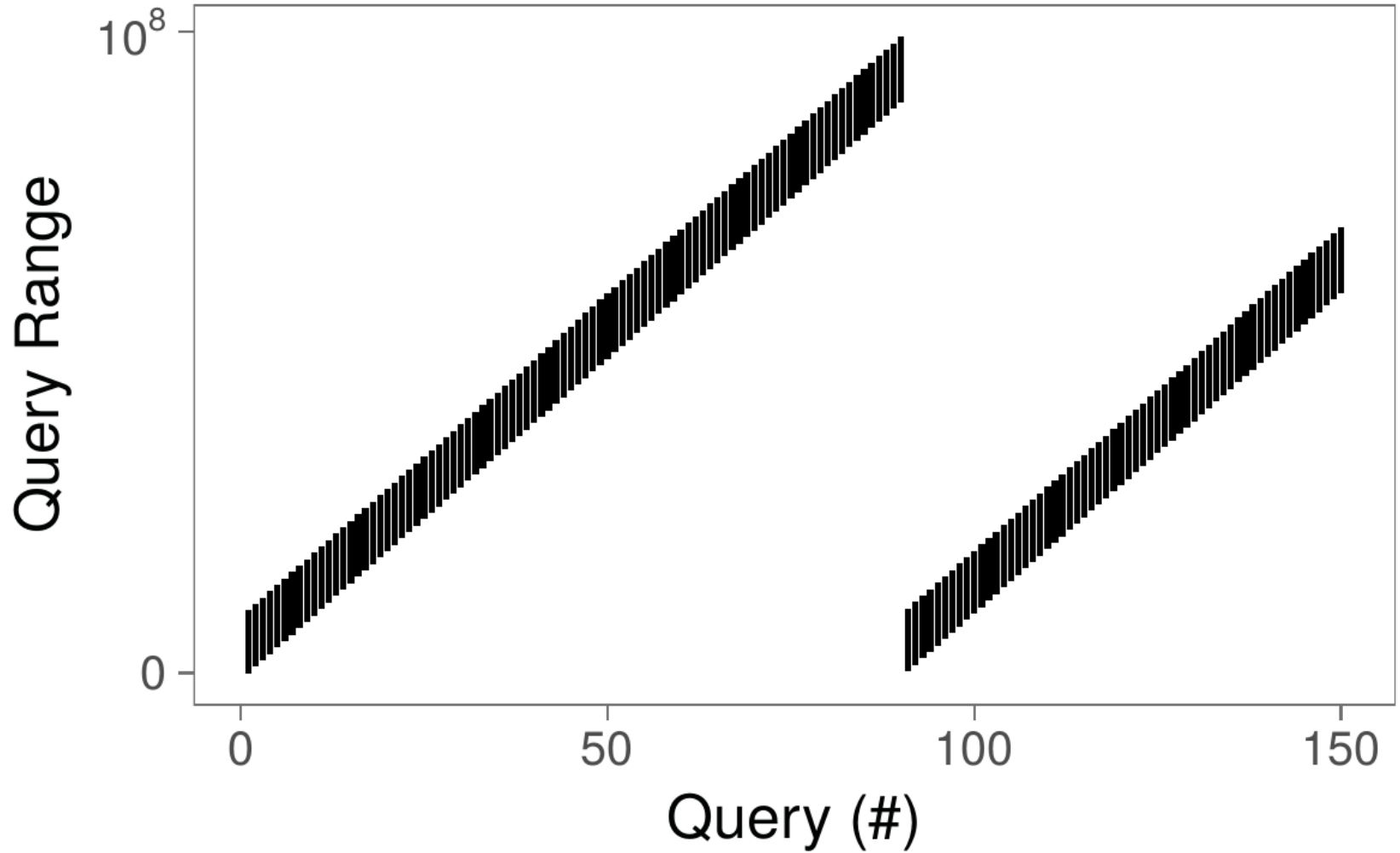
# Comparison: Progressive Indexing



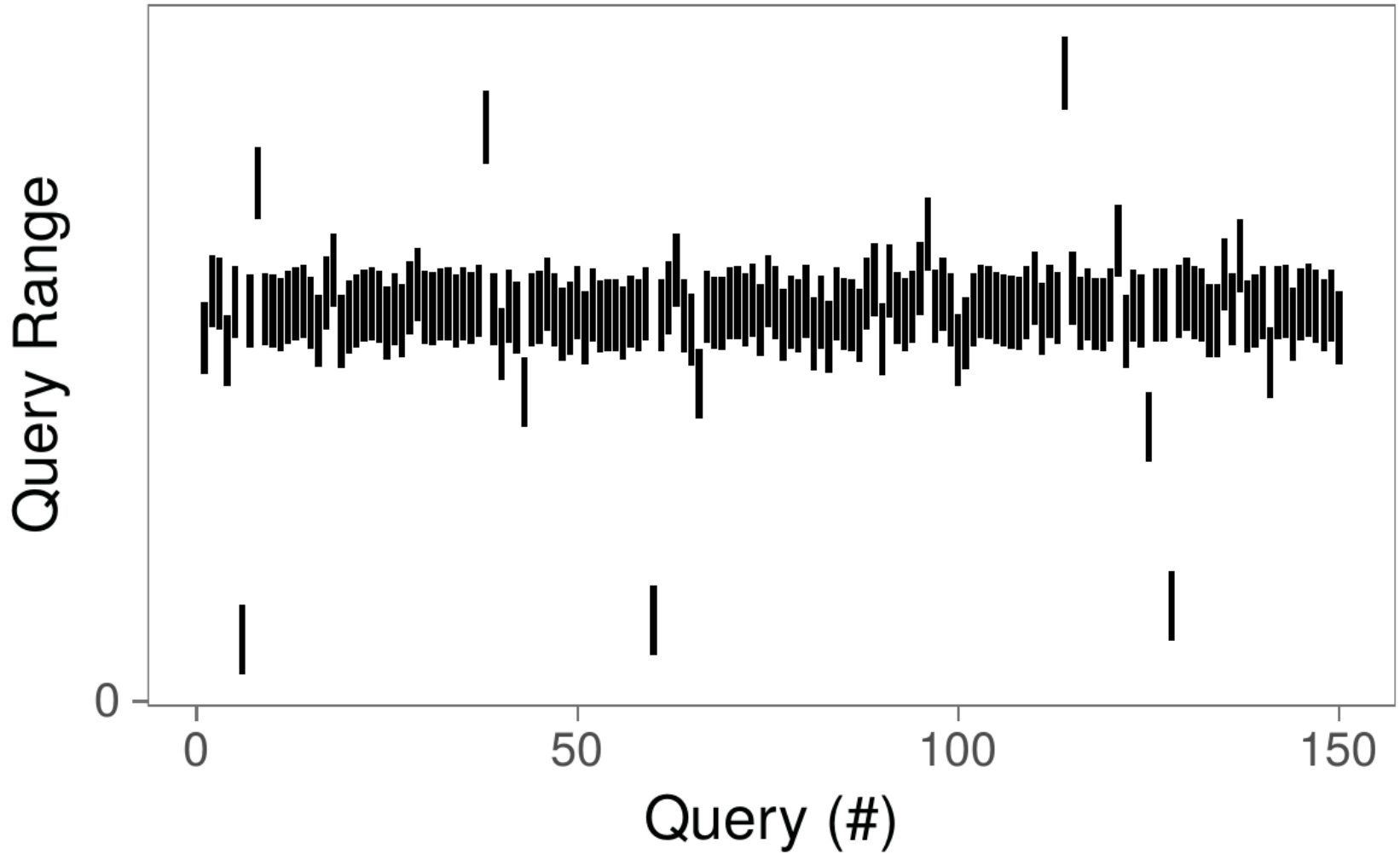
# Random Workload



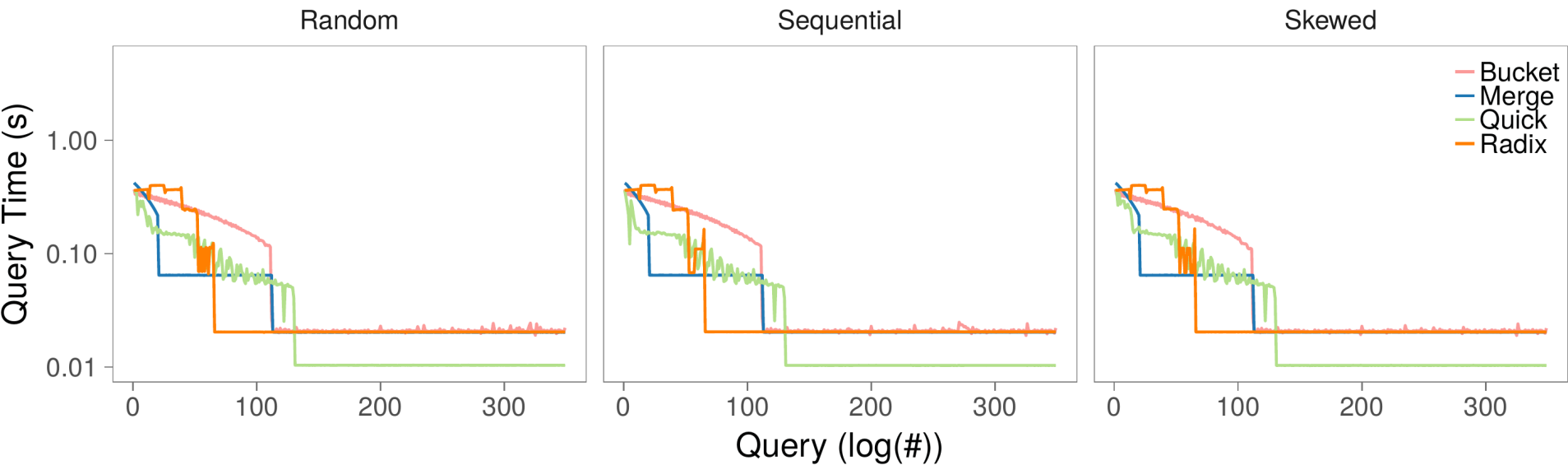
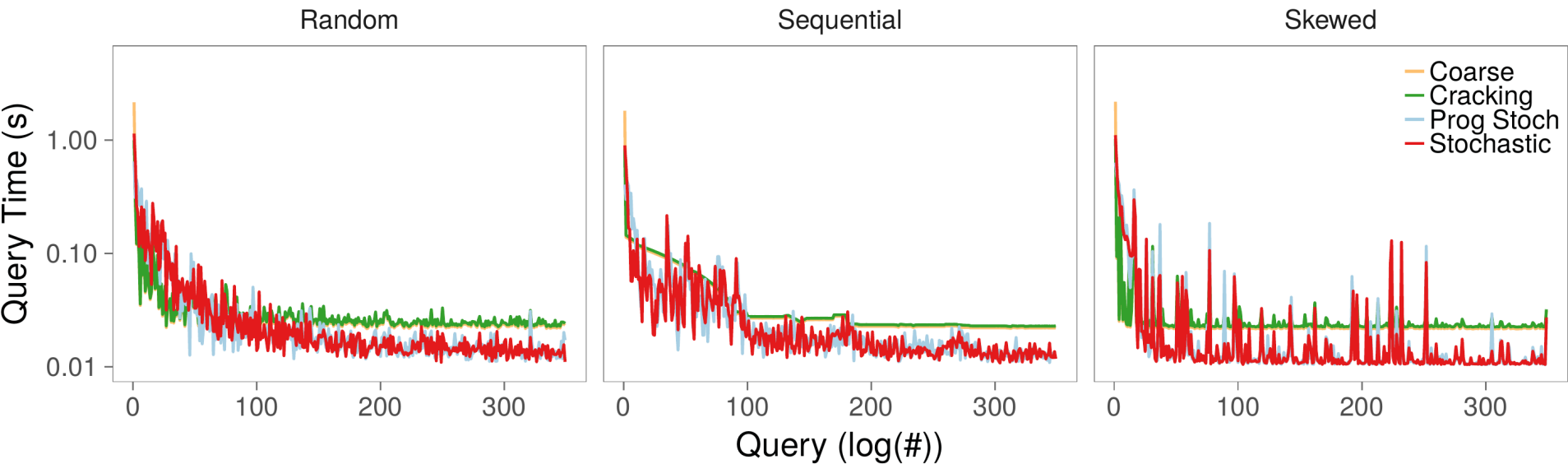
# Sequential Workload



# Skewed Workload



## Different Workloads



## # Queries until Pay-off

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<sup>st</sup> query
- Various basic sorting algorithms
  - Quick-sort
  - Merge-sort
  - Bucket-sort
  - Radix-sort