

ICDE 2018

Adaptive Adaptive Indexing

Felix Martin Schuhknecht

Jens Dittrich

Laurent Linden

Big Data Analytics Group

bigdata.uni-saarland.de

Saarland University

2007

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].

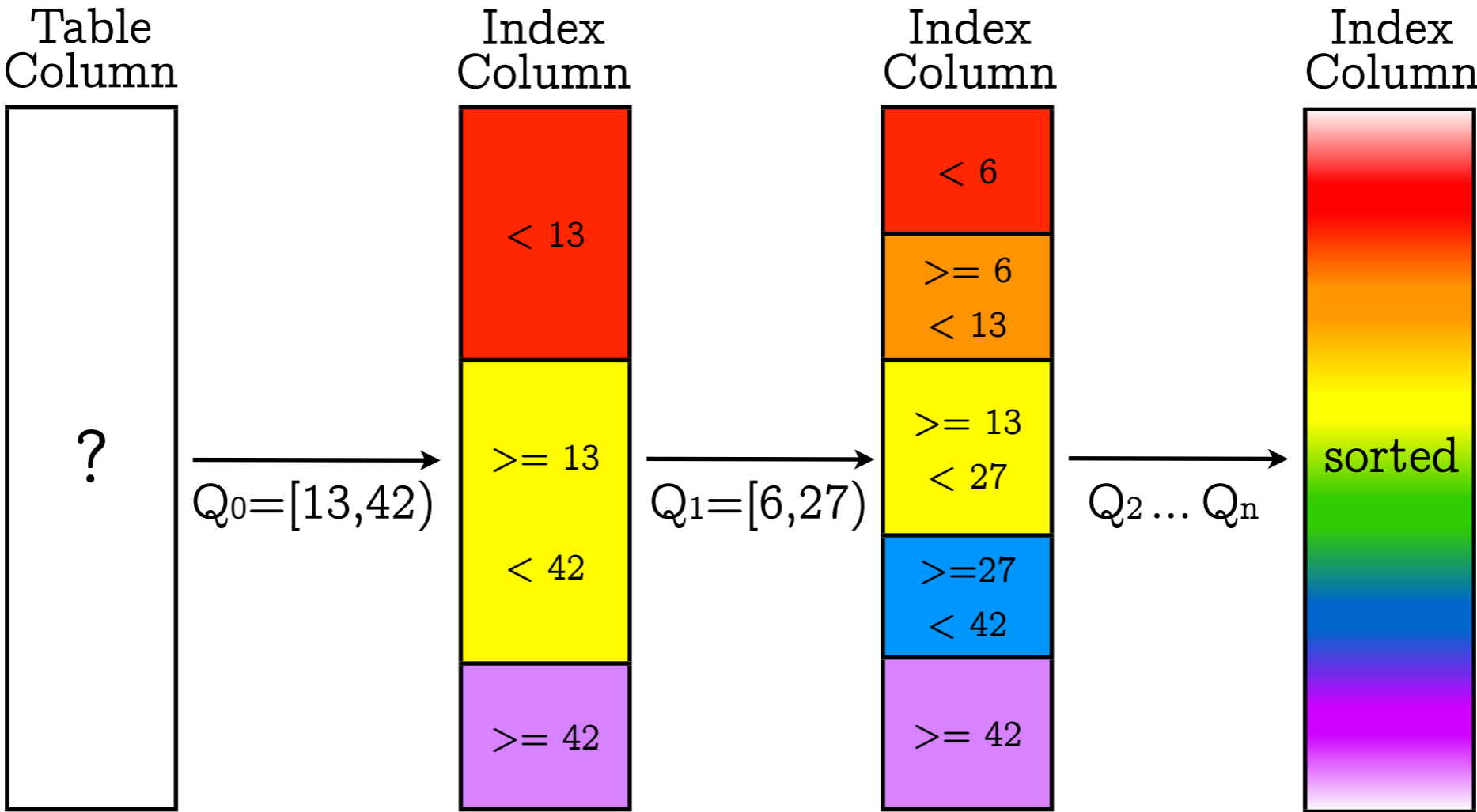
This article is published under a Creative Commons License Agreement (<http://creativecommons.org/licenses/by/2.5/>). You may copy, distribute, display, and perform the work, make derivative works and make commercial use of the work, but you must attribute the work to the author and CIDR 2007.

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

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

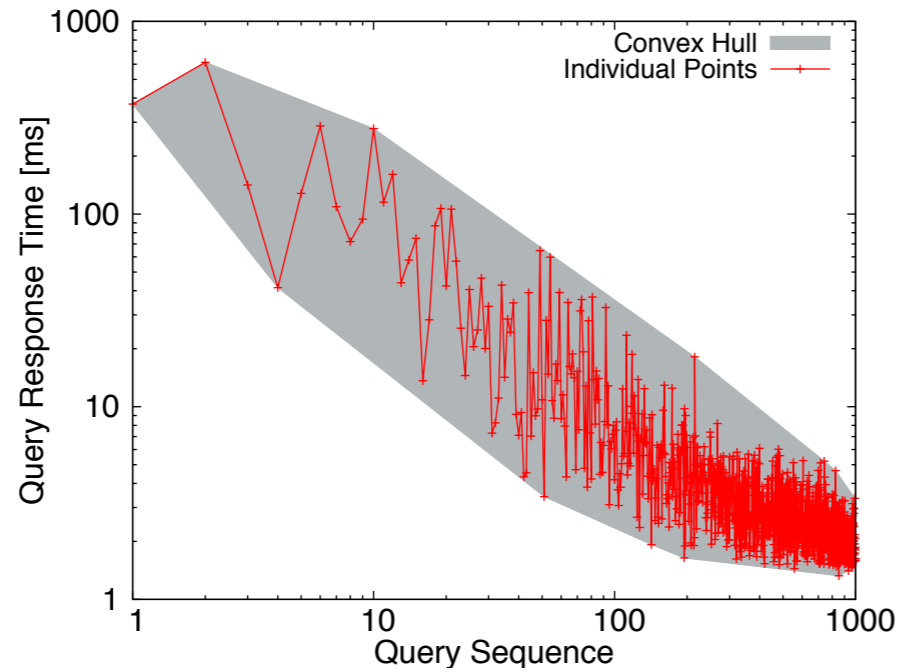
The idea of physically reorganizing the database based on incoming queries has first been proposed in [10]. The contributions of this paper are the following. We present the first mature cracking architecture (a complete cracking software stack) in the context of column oriented databases. We report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

Database Cracking / Standard Cracking

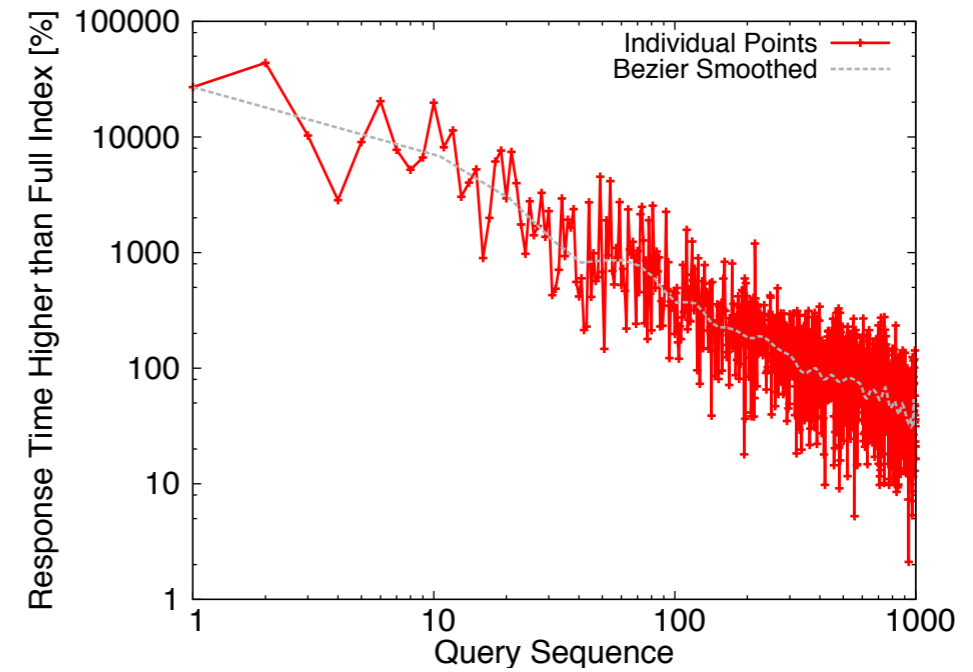


Problems?

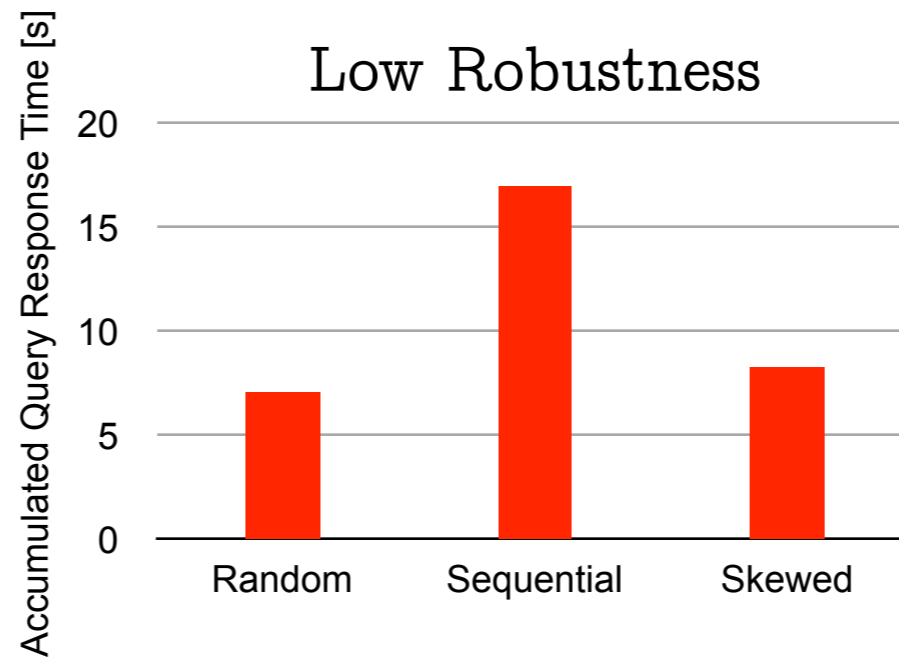
High Variance



Low Convergence Speed



Low Robustness



Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores^{*}

Felix Halim^{*} Stratos Idreos[†] Panagiotis Karras[‡] Roland H. C. Yap^{*}
^{*}National University of Singapore (halim, ryap)@comp.nus.edu.sg
[†]CWI, Amsterdam idreos@cwi.nl
[‡]Rutgers University karras@business.rutgers.edu

ABSTRACT

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

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

1. INTRODUCTION

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

^{*}Work supported by Singapore's M

Permission to make digital or hard personal or classroom use is granted not made or distributed for profit or bear this notice and the full citation to the journal, to post on servers or to request permission and/or a fee. Articles from this journal are published in the Proceedings of the VLDB Endowment Copyright 2012 VLDB Endowment

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 index should provide easy access to for future queries; the remainder of the data remains non-indexed until a query expresses inter-

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

Stratos Idreos[†] Stefan Manegold[‡] Harumi Kuno[§] Goetz Graefe[¶]
[†]CWI, Amsterdam (stratos.idreos, stefan.manegold)@cwi.nl
[‡]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 focused 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 therefore 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-refined for a particular workload component.

Based on an examination of database cracking and adaptive merging, which are two techniques for adaptive indexing, we seek 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 database system. Our experiments compare their behavior against database cracking and adaptive merging, as well as against both traditional full index lookup and scan of unordered data. We show that the new hybrids significantly improve over past methods while at least two of the hybrids come very close to the “ideal performance” 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 three weaknesses. First, the interval between monitoring and index creation can exceed 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. Data access during the monitoring interval neither benefits from nor aids index creation efforts, 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 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 journal are published in the Proceedings of the VLDB Endowment, Vol. 4, No. 9, August 29th – September 3rd 2011, Seattle, Washington. Copyright 2011 VLDB Endowment 2150-8097/11/09... \$ 10.00.

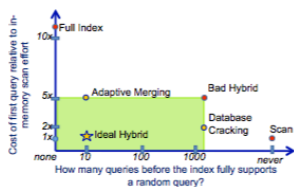


Figure 1: Adaptive Indexing Research Space. Last, but not least, traditional indexes on tables cover all rows equally, even if some rows are needed often and some never.

Our goal is to enable incremental, efficient adaptive indexing, i.e., index creation and optimization as side effects of query execution, with the implicit benefit that only tables, columns, and key ranges truly queried are optimized. As proposed in [5], we use two measures to characterize how quickly and efficiently a technique adapts index structures to a dynamic workload. These are: (1) the initialization cost incurred by the first query and (2) the number of queries that must be processed before a random query benefits from the index structure without incurring any overhead. We focus particularly on the first query because it captures the worst-case costs and benefits of adaptive indexing; if that portion of data is never queried again, then any overhead above and beyond the cost of a scan is wasted effort.

Recent work has proposed two distinct approaches: database cracking [10, 11, 12] and adaptive merging [6, 7]. The more often a key range is queried, the more its representation is optimized. Columns that are not queried are not indexed, and key ranges that are not queried are not optimized. Overhead for incremental index creation is minimal, and disappears when a range has been fully-optimized. In order to evaluate database cracking and adaptive merging, we have implemented both approaches in a modern column-store database system, and find the strengths and weaknesses of the two approaches complementary.

As shown in Figure 1, adaptive merging has a relatively high initialization cost but converges rapidly, while database cracking enjoys a low initialization cost but converges relatively slowly. The green box in Figure 1 thus defines the research space for 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, marked with a star in the figure, that incurs a low initialization cost yet also converges quickly

Database Cracking

Stratos Idreos[†] Martin L. Kersten[‡]
[†]CWI Amsterdam (stratos.idreos@cwi.nl)
[‡]CWI Amsterdam (Martin.Kersten@cwi.nl)

CT

ides provide a non-discriminative navigational to localize tuples of interest. Their maintenance is taken during database updates. In this paper the complementary approach, addressing in-ance as part of query processing using continu-ree organization, i.e., cracking the database into pieces. The motivation is that by automatically data the way users request it, we can achieve fast he much desired self-organized behavior. at the first mature cracking architecture and re-implementation of cracking in the context of a relational system. It led to a minor enhancement al algebra kernel, such that cracking could be without incurring too much processing over-erance, we illustrate the ripple effect of dynamic on the query plans derived by the SQL opti-experiences and results obtained are indicative of reduction in system complexity. We show that system is able to self-organize based on incom- with clear performance benefits. This behavior an when the user focus is randomly shifting to

design sign should nize acable re-found in ments designs. that are abilities depar-design

agement derivative these the

Self-organizing Tuple Reconstruction in Column-stores

Stratos Idreos[†] Martin L. Kersten[‡] Stefan Manegold[‡]
[†]CWI Amsterdam (idreos@cwi.nl)
[‡]CWI Amsterdam (mk@cwi.nl, manegold@cwi.nl)

ABSTRACT

Column-stores gained popularity as a promising physical design alternative. Each attribute of a relation is physically stored as a separate column allowing queries to load only the required attributes. 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 component. The ultimate physical design is to have multiple presorted 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, idle time to prepare, and infrequent updates.

In this paper, we propose a novel design, *partial sideways cracking*, that minimizes the tuple reconstruction cost in a self-organizing way. It achieves performance similar to using presorted data, but without requiring the heavy initial presorting 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 continuously physically reorganized as an integral part of query evaluation, providing faster and reduced data access for 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 chunks that are individually reorganized, dropped or recreated as needed. We implemented partial sideways 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 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 journal are published in the Proceedings of the VLDB Endowment Copyright 2012 VLDB Endowment

Self-selecting, self-tuning, incrementally optimized indexes

Goetz Graefe[†] Harumi Kuno[‡]
[†]Hewlett-Packard Laboratories (goetz.graefe)@hp.com
[‡]Hewlett-Packard Laboratories (harumi.kuno)@hp.com

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

H.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 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 journal are published in the Proceedings of the VLDB Endowment, Vol. 4, No. 9, August 29th – September 3rd 2011, Seattle, Washington. Copyright 2011 VLDB Endowment 2150-8097/11/09... \$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 relevant 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]. Whenever possible, position-based join-matching and sequential data access are exploited. For each relation R_i in a query plan q , a column-store needs to perform at least $N_i - 1$ tuple reconstruction operations for R_i within q , given that N_i attributes of R_i participate in q .

Column-stores perform tuple reconstruction in two ways [2]. With *early* tuple reconstruction, the required attributes are glued together as early as possible, i.e., while the columns are loaded, leveraging N -ary processing to evaluate the query. On the other hand, *late* tuple reconstruction exploits the column-store architecture to its maximum. During query processing, “reconstruction” merely refers to getting the attribute values of qualifying tuples from their base columns as late as possible, i.e., only once an attribute is required in 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 -ary tuples are formed only once the final result is delivered.

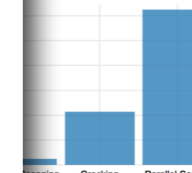
Like most modern column-stores [2, 4, 15], we focus on late reconstruction. Comparing early and late reconstruction, the educative analysis in [2] observes that the latter incurs the overhead of reconstructing a column more than once, in case it occurs more than once in a query. Furthermore, exploiting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

The ultimate access pattern is to have multiple copies for each relation R_i , such that each copy is presorted on an other attribute in R_i . All tuple reconstructions of R_i attributes initiated by a restriction on an attribute A_i can be performed using the copy that is sorted on A_i . This way, the tuple

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 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.



Costs of Database Operations

of Cracking is fully sorted data, its costs are those of fully sorting the data. With recent (parallel) sorting algorithms [7], however, lessening unattractive. To illustrate this, Figure 1 compares the respective operations on integer values on a 4-Core Sandy Bridge CPU. off-the-shelf (Parallel) MergeSort implements more expensive than a (quasi I/O bound) only three times as expensive as MonetDB's cracking [19]. Even though both Scanning and (y) read and write the same amount of data, the performance difference must, computational costs: Cracking is, unlike Scanning, implemented with the underlying hardware in (be roughly) I/O bound.

In this hypothesis, we make the following contributions: to conduct an in-depth study of the contributing performance of the “classic” Cracking implementation.

on the findings, we develop a number of optimization-based on “standard” techniques like prediction, vector and manually implemented data parallelism using instructions.

develop two different parallel algorithms that exploit thread parallelism to make use of multiple CPU cores.

erously evaluate all developed algorithms on a number of test systems ranging from low-end desktop machines to end servers.

GNU libstdc++ Version 4.8.2

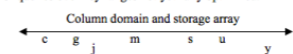


Figure 1. A column store partitioned by database cracking.

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

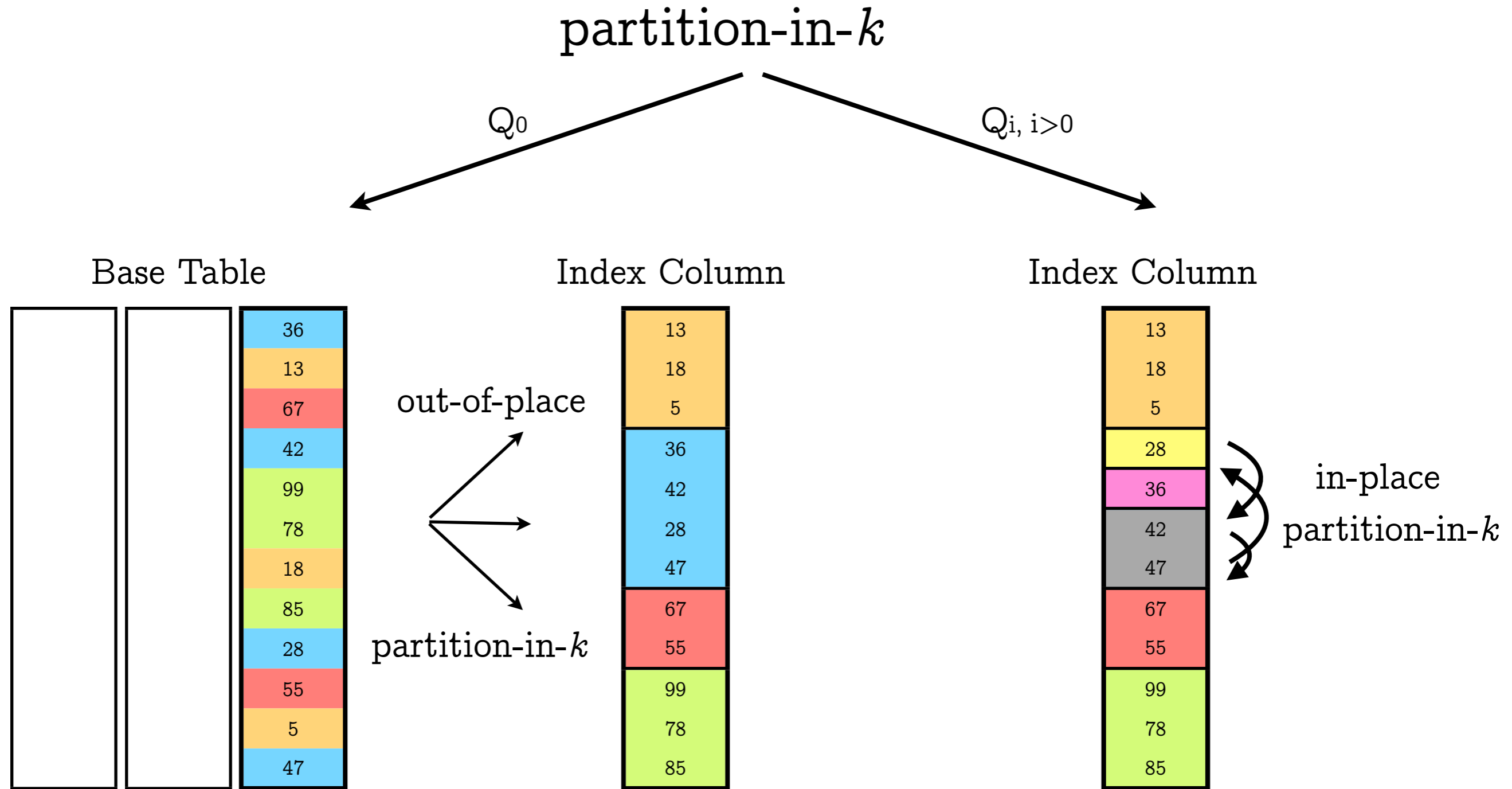
All-in-one?

An Adaptive Adaptive Index?

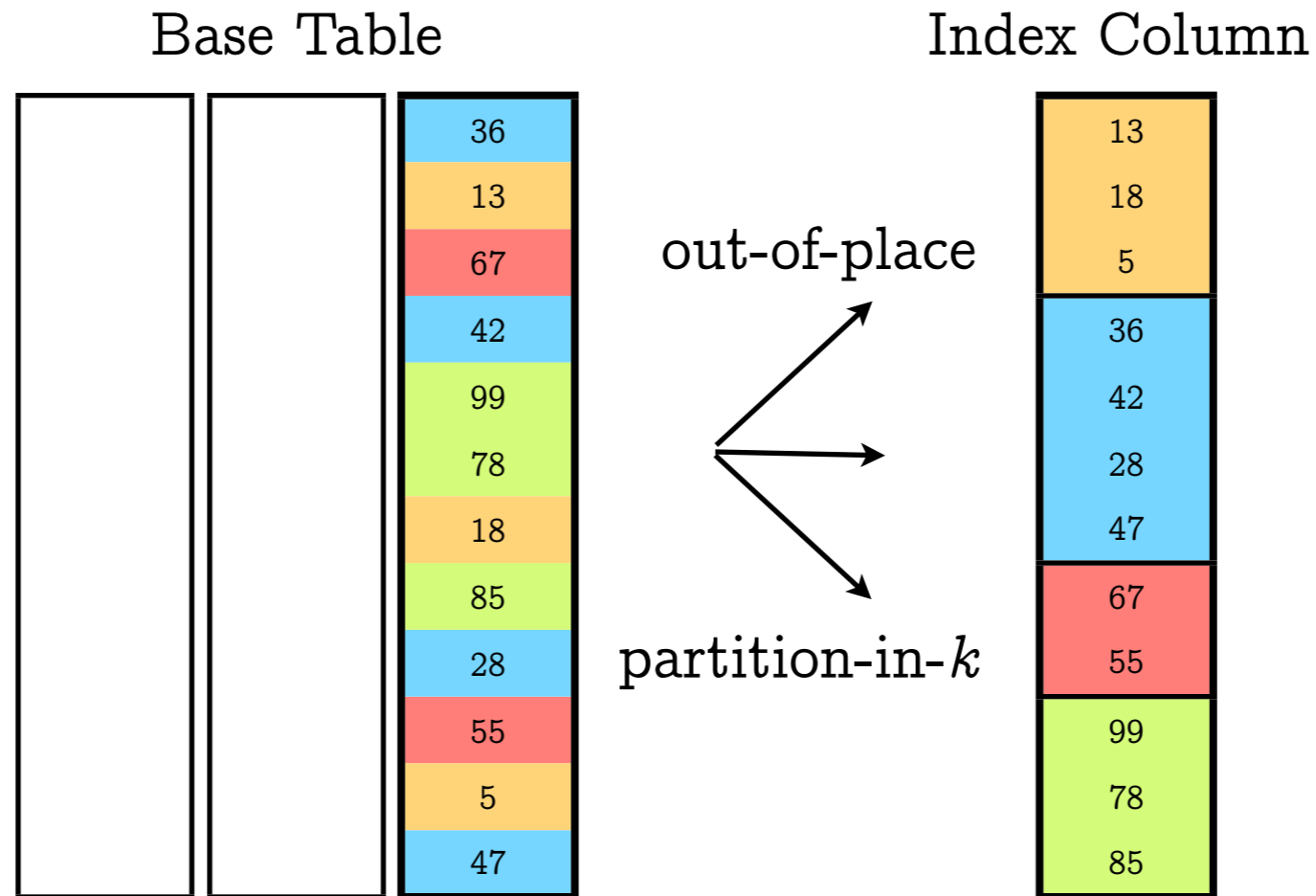
Design rules:

1. Generalize way of refinement
2. Adapt refinement effort
3. Awareness of key distributions

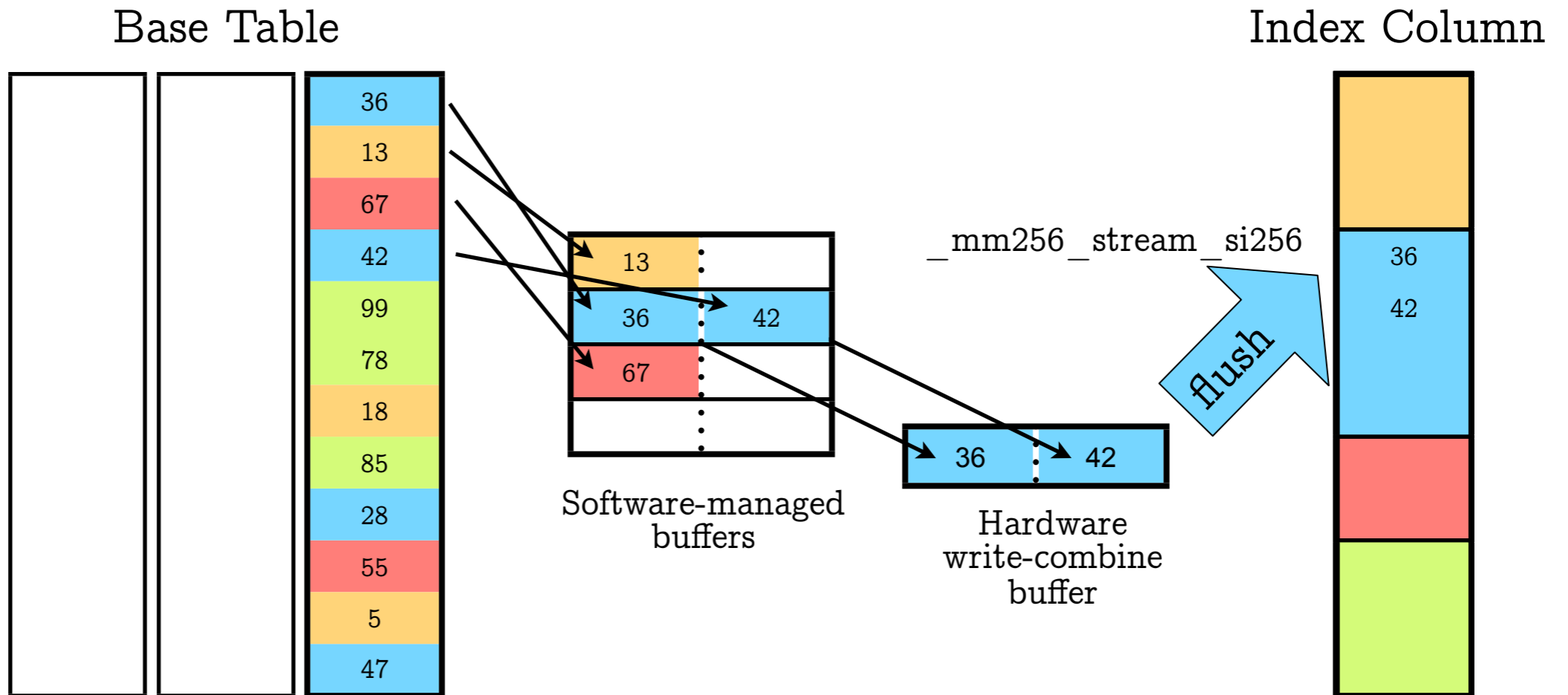
1. Generalize way of refinement:



1. Generalize way of refinement:



1. Generalize way of refinement:

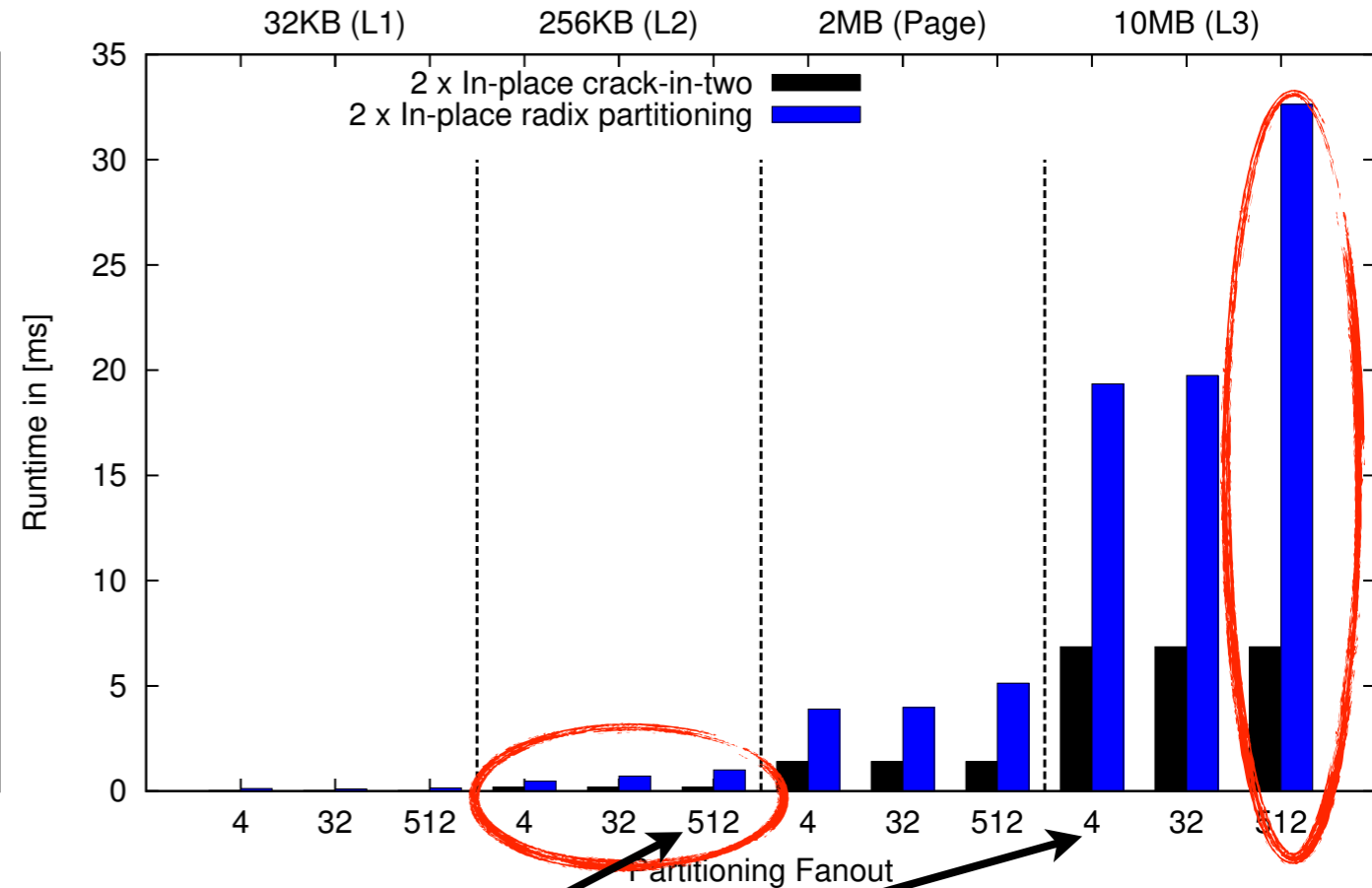
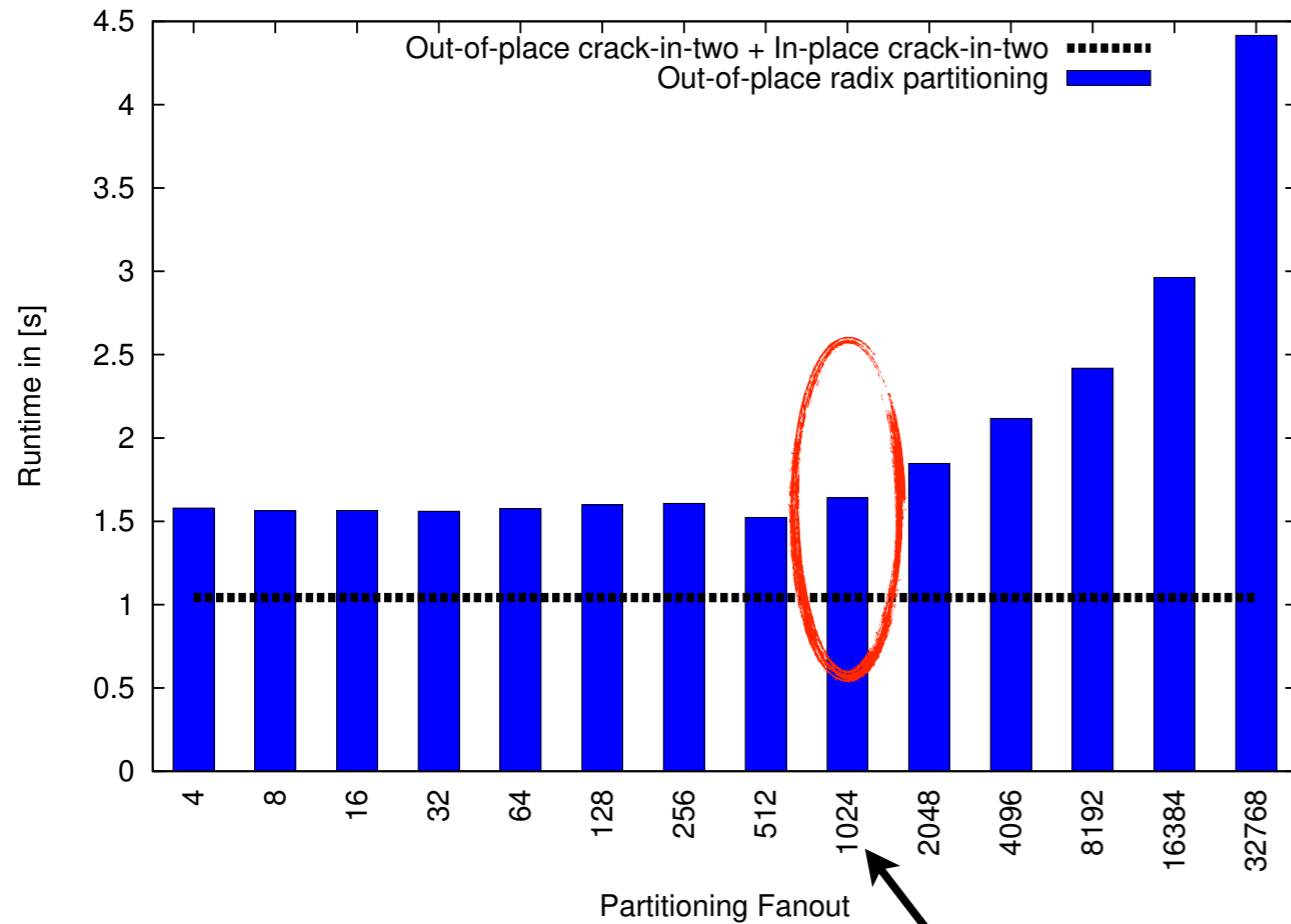


2. Adapt refinement effort

Q_0

$Q_i, i > 0$

Input data size

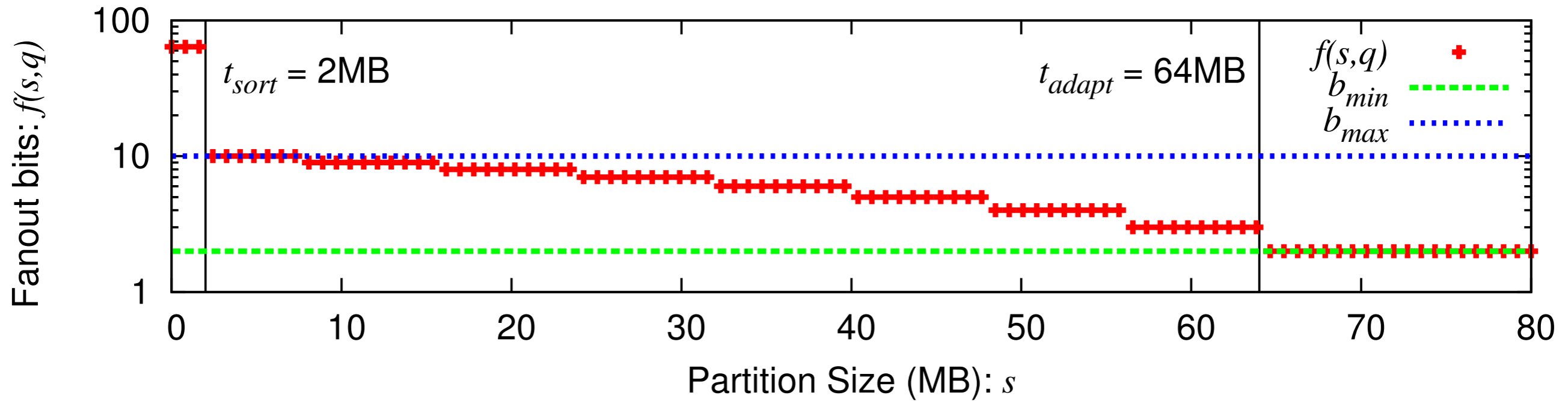


$$f(s, q) = \begin{cases} b_{first} \\ b_{min} \\ b_{min} + \left\lceil (b_{max} - b_{min}) \cdot \left(1 - \frac{s}{t_{adapt}}\right) \right\rceil \\ b_{sort} \end{cases}$$

if $q = 0$
 else if $s > t_{adapt}$
 else if $s > t_{sort}$
 else.

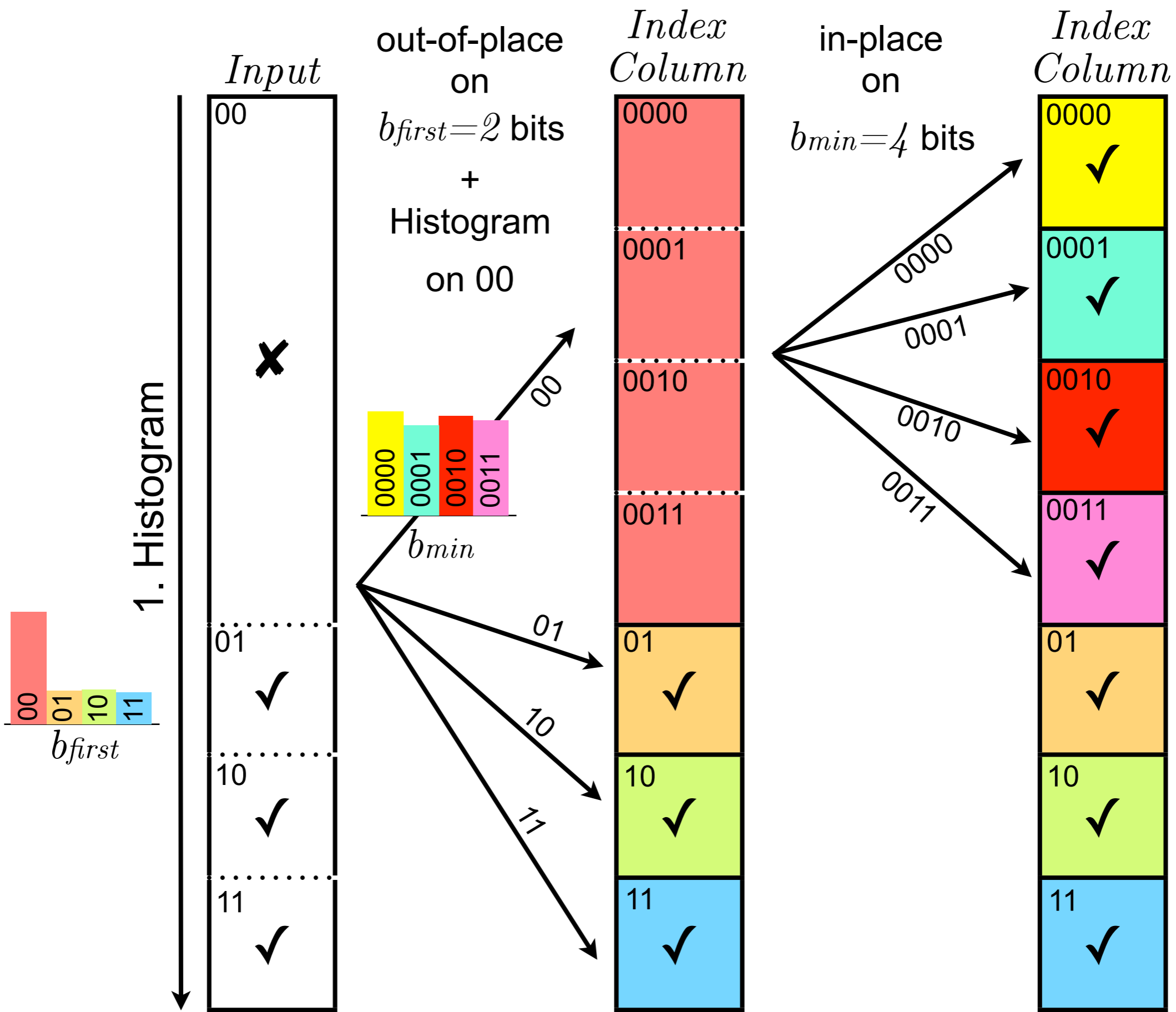
partition size query sequence number

2. Adapt refinement effort

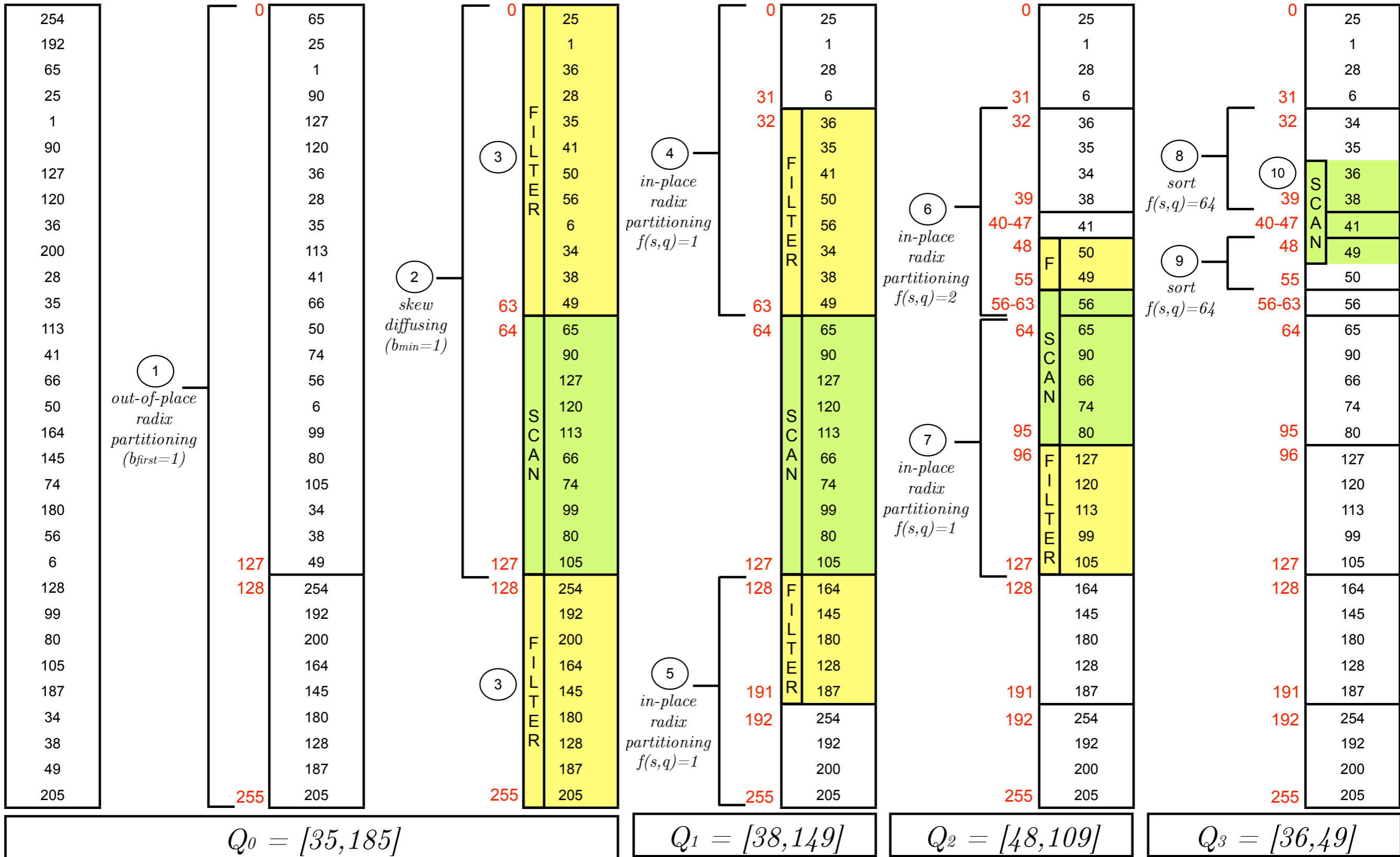


$$f(s, q) = \begin{cases} b_{first} & \text{if } q = 0 \\ b_{min} & \text{else if } s > t_{adapt} \\ b_{min} + \left\lceil (b_{max} - b_{min}) \cdot \left(1 - \frac{s}{t_{adapt}}\right) \right\rceil & \text{else if } s > t_{sort} \\ b_{sort} & \text{else.} \end{cases}$$

3. Awareness of key distributions: skew?

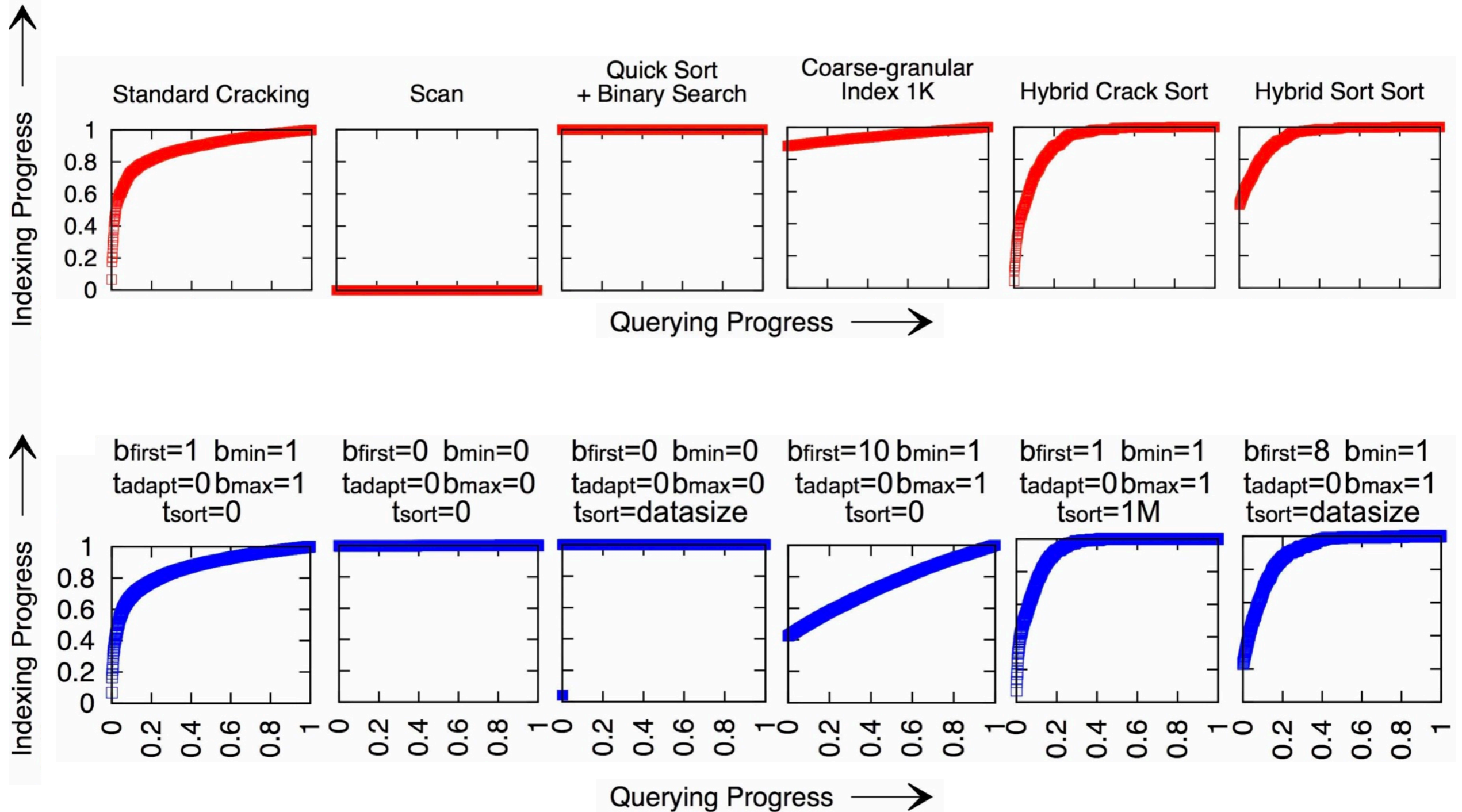


Putting it all together

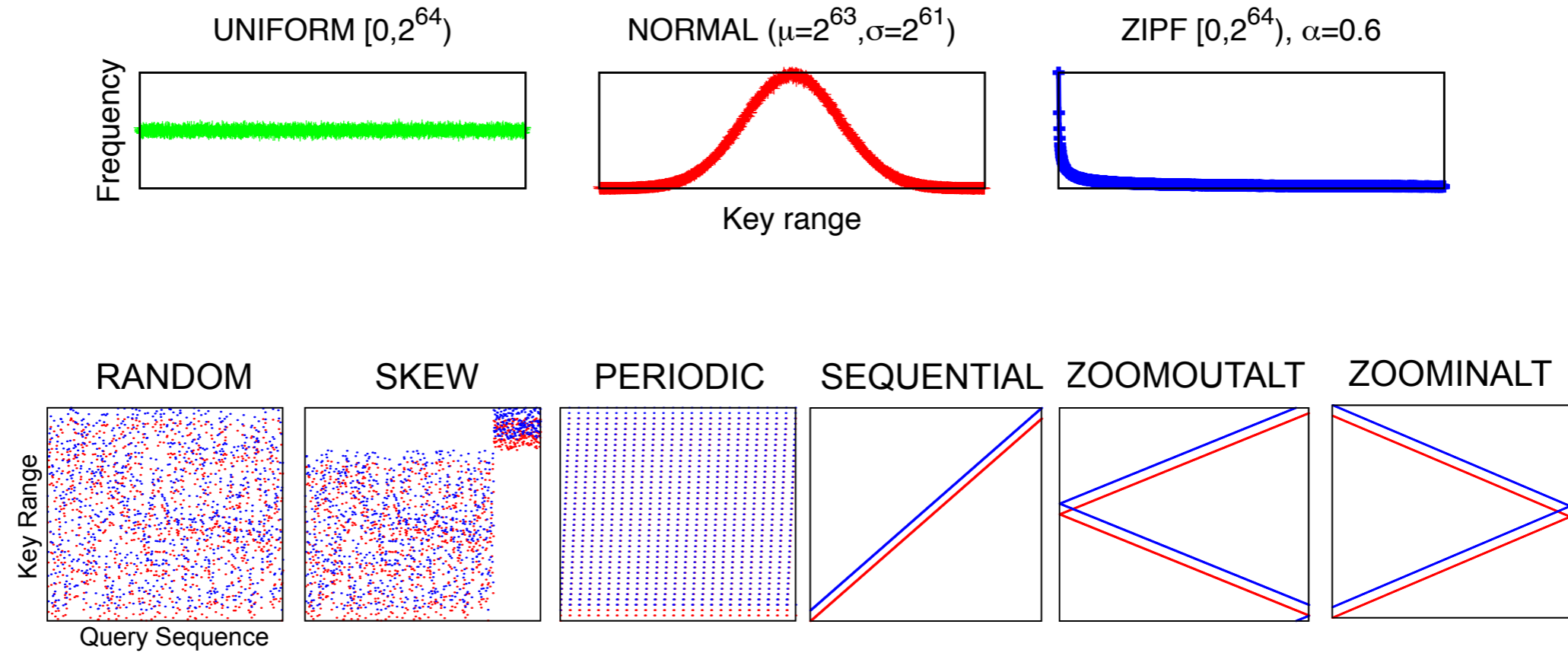


Emulation

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

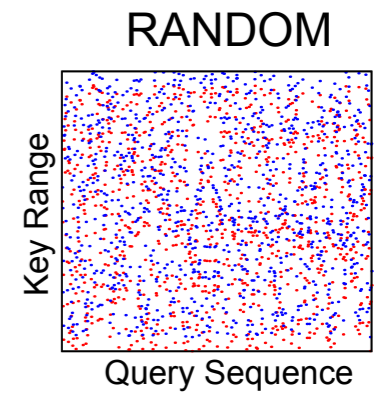
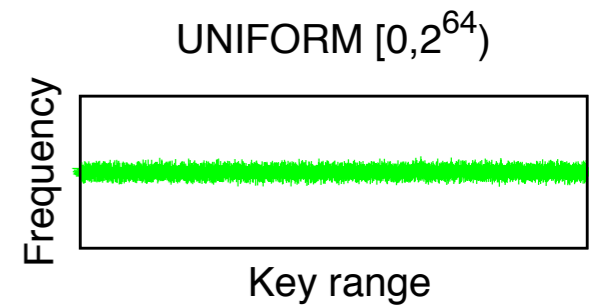
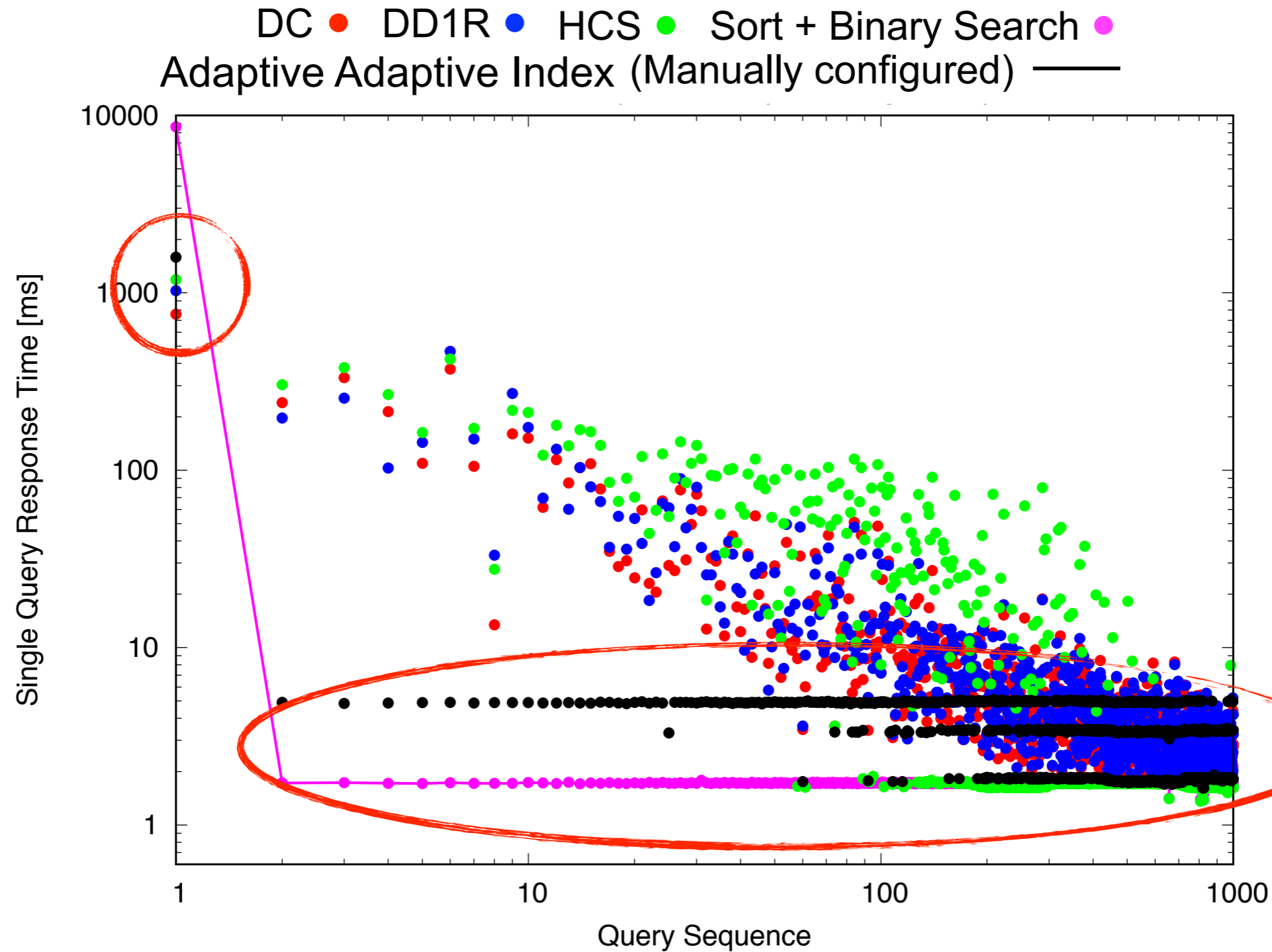


Test Setup



[Felix Halim, Stratos Idreos, Panagiotis Karras, Roland H. C. Yap:
Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores, PVLDB Vol. 5, No. 6]

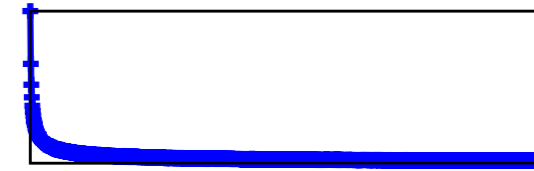
Individual Query Response Times



$b_{first} = 10$
 $b_{min} = 3$
 $b_{max} = 6$
 $t_{adapt} = 64MB$
 $t_{sort} = 256KB$

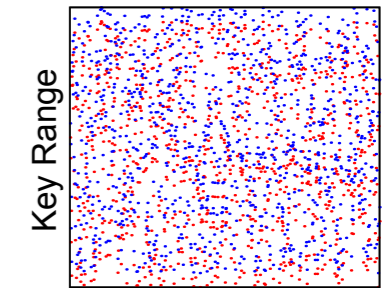
Individual Query Response Times

ZIPF $[0, 2^{64})$, $\alpha=0.6$

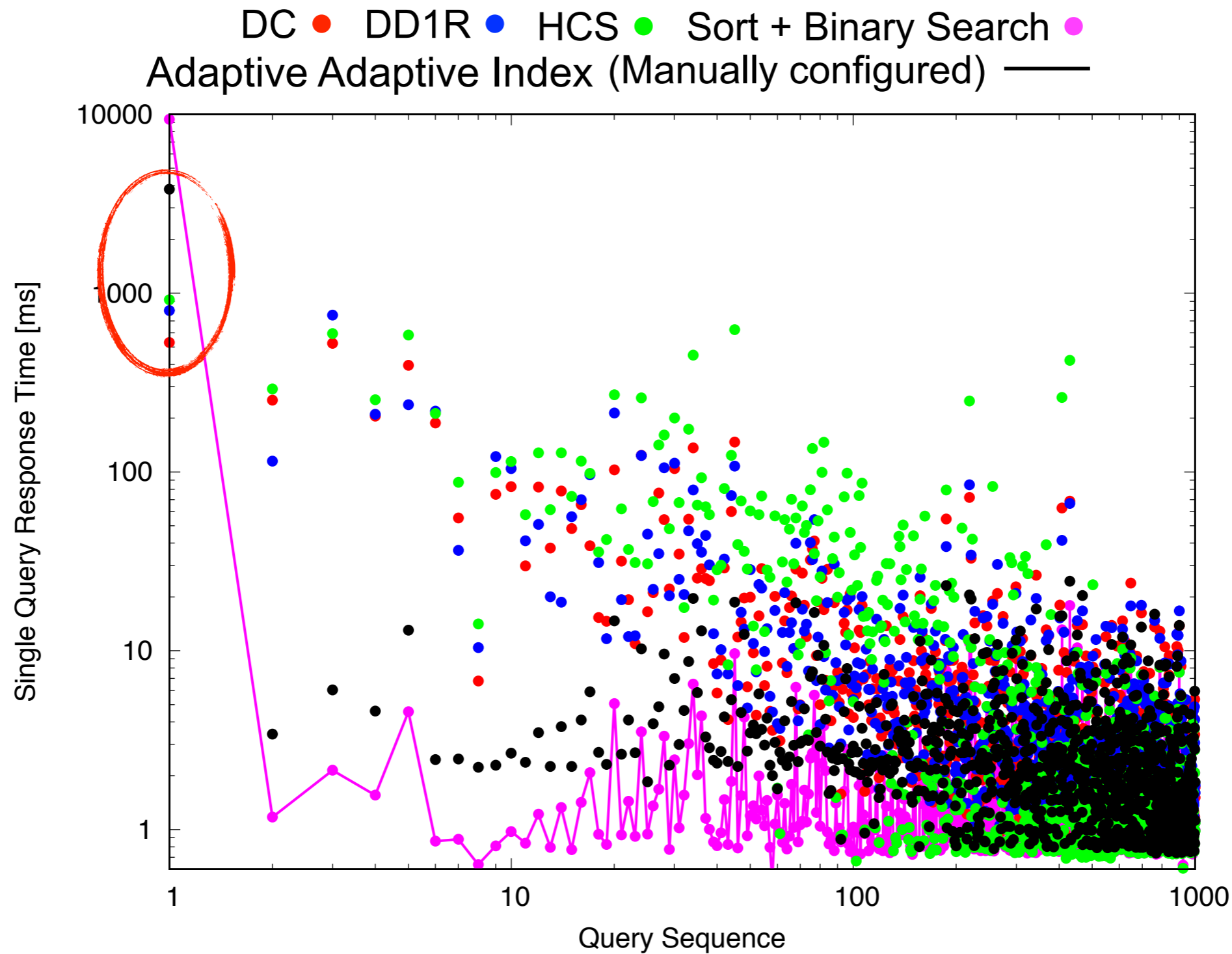


Key range

RANDOM



Query Sequence



$b_{first}=10$

$b_{min}=3$

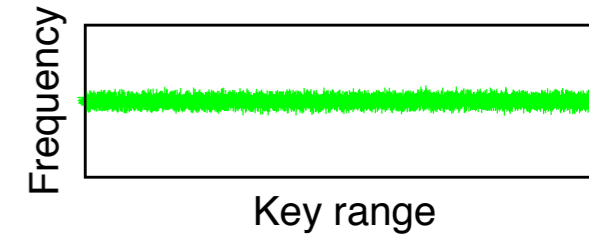
$b_{max}=6$

$t_{adapt}=64MB$

$t_{sort}=256KB$

Accumulated Query Response Times

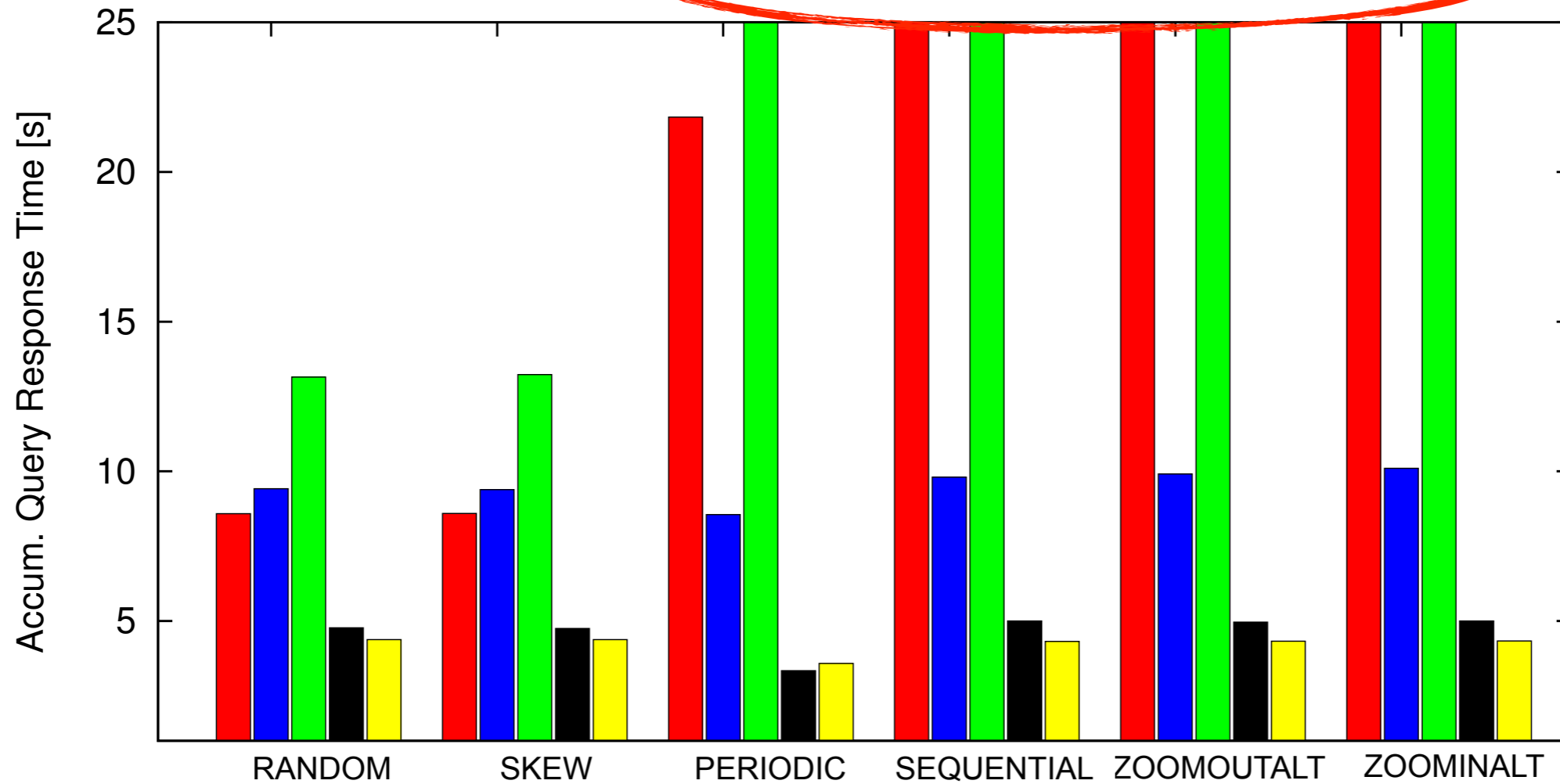
UNIFORM $[0, 2^{64})$



DC █ DD1R █ HCS █

Adaptive Adaptive Index (Manually configured) █

Adaptive Adaptive Index (Simulated annealing configured) █



$b_{first} = 10$

$b_{min} = 3$

$b_{max} = 6$

$t_{adapt} = 64MB$

$t_{sort} = 256KB$

