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 load that interferes with query execution. Last, but not least, traditional indexes on tables cover all rows equally, even if some rows are needed often and some never.

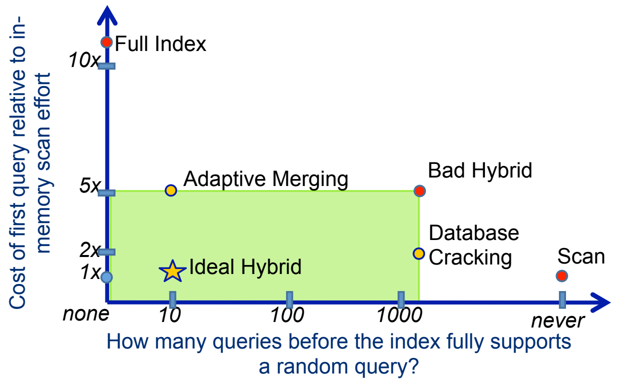


Figure 1: Adaptive Indexing Research Space.

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

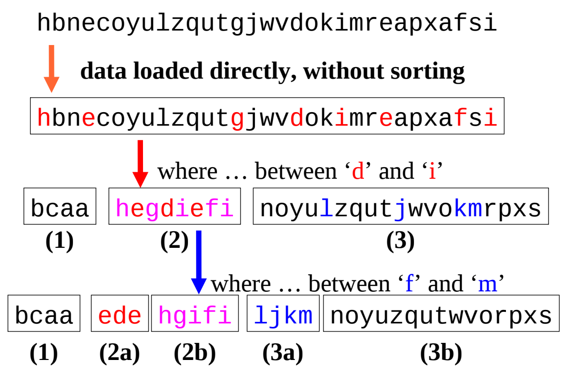

hbnecoyulzqutgjwvdokimreapxafsi
data loaded directly, without sorting
hbnecoyulzqutgjwvdokimreapxafsi
where … between 'd' and 'i'
bcaa
hegdiefi
noyulzqutjwvokmrpxs
(1)
(2)
(3)
where … between 'f' and 'm'
bcaa
ede
hgifi
ljkm
noyuzqutwvorpxs
(1)
(2a)
(2b)
(3a)
(3b)

Figure 2: Database cracking.

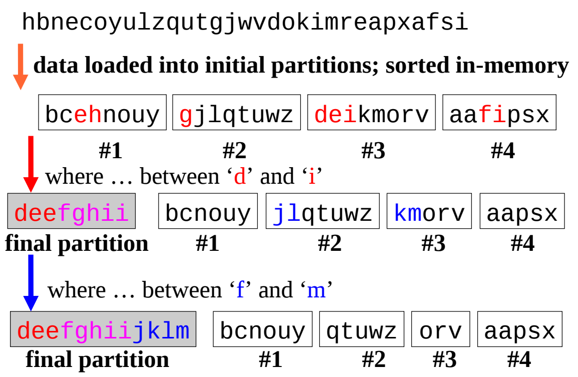


Figure 3: Adaptive merging.

to an optimized index structure. At the same time, we also recognize the risk of developing sub-optimal techniques, such as the one labeled "Bad Hybrid" in the figure.

This paper provides the first detailed comparison between these two techniques through an in-memory implementation in MonetDB. We study the various trends and tradeoffs that occur and we propose and compare a number of new hybrid adaptive indexing approaches. These hybrids are intended to meet both the database cracking design goal of minimizing initial per query overhead and also the adaptive merging design goal of exploiting the concept of runs and merges to converge quickly. The net effect is that the new algorithms offer a light-weight adaptation that converges efficiently to a refined index. As even the first query incurs zero overhead over a full scan approach, this work essentially opens the door to tuning-free systems that adapt automatically to workload changes.

The rest of the paper is organized as follows. Section 2 gives background and analyzes previous approaches to adaptive indexing. Section 3 then presents the various hybrid algorithms and Section 4 provides a detailed experimental analysis. We have implemented all algorithms in MonetDB (`http://monetdb.cwi.nl/`), and our experiments compare the benefits of these hybrid techniques. Finally, Section 5 concludes the paper.

2. BACKGROUND AND PRIOR WORK

Prior Approaches. Most previous approaches to runtime index tuning [1, 2, 3, 8] are non-adaptive, meaning that index tuning and query processing operations are distinct from each other. These approaches first monitor the running workload and then decide which indexes to create or drop based on the observations. Both index tuning and index creation costs impact the database workload. Once a decision is made, it affects all key ranges in an index. The recognition that some data items are more heavily queried than others has led to the concept of partial indexes [14, 15]. A generalization is the concept of materialized views.

"Soft indexes" anticipates adaptive indexing [13]. Like monitor-and-tune approaches, soft indexes continually collects statistics for recommended indexes and then periodically and automatically solves the index selection problem. Like adaptive indexing, recommended indexes are generated (or dropped) as a part of query processing. Unlike adaptive indexing, however, neither index recommendation nor creation is incremental; explicit statistics are kept and each recommended index is created and optimized to completion, although the command might be deferred.

In general, adaptive indexing and approaches that monitor queries then build indexes are mutually compatible. Policies established by the observe-and-tune techniques could provide information about the benefit and importance of different indexes, and adaptive indexing mechanisms could then create and refine the recommended index structures while minimizing additional workload.

Database Cracking. Database cracking combines features of automatic index selection and partial indexes. It reorganizes data within the query operators, integrating the re-organization effort into query execution. When a column is queried by a predicate for the first time, a new cracker index is initialized. As the column is used in the predicates of further queries, the cracker index is refined by range partitioning until sequentially searching a partition is faster than binary searching in the AVL tree guiding a search to the appropriate partition.

Keys in a cracker index are partitioned into disjoint key ranges, but left unsorted within each partition. Each range query analyzes the cracker index, scans key ranges that fall entirely within the query range, and uses the two end points of the query range to further partition the appropriate two key ranges. Thus, in most cases, each partitioning step creates two new sub-partitions using logic similar to partitioning in quicksort [9]. A range is partitioned into 3 sub-partitions if both end points fall into the same key range. This happens in the first partitioning step in a cracker index (because there is only one key range encompassing all key values) but is unlikely thereafter [10].

The example in Figure 2 shows data being loaded directly, without sorting, into an unsorted array. As a side-effect of answering a first query on the range "d – i", the array is split into three partitions: (1) keys before 'd'; (2) keys that fall between 'd' and 'i'; and (3) keys after 'i'. Then a new query for range "f – m" is processed. The values in partition (1) can be ignored, but partitions (2) and (3) are further cracked on keys 'f' and 'm', respectively. Subsequent queries continue to partition these key ranges until the structures have been optimized for the current workload.

Updates and their efficient integration into the data structure are covered in [11]. Multi-column indexes to support selections, tuple reconstructions and general complex queries are covered in [12]. In addition, [12] supports partial materialization and adaptive space management via partial cracking. Finally, recent work [7] has suggested several optimizations for database cracking.

Adaptive Merging. While database cracking functions as an incremental quicksort, with each query resulting in at most one or two partitioning steps, adaptive merging functions as an incremental merge sort, with one merge step applied to all key ranges in a query's result. Under adaptive merging, the first query to use a given column in a predicate produces sorted runs and each subsequent query upon that same column applies to at most one additional merge step. Each merge step only affects those key ranges that are relevant to actual queries, leaving records in all other key ranges in their initial places. This merge logic takes place as a side effect of query execution.

In Figure 3, for instance, the first query (with range boundaries 'd' and 'i') triggers the creation of four sorted runs, loading the data into equally-sized partitions and sorting each in memory, then retrieves relevant values (via index lookup because the runs are sorted) and merges them out of the runs and into a "final" partition. Similarly, results from a second query on range "f – m" are merged out of the runs and into the final partition. Subsequent queries continue to merge results from the runs until the "final" partition has been fully optimized for the current workload.

3. HYBRID ALGORITHMS

One concern about database cracking is that at most two new partition boundaries per query means that the technique requires thousands of queries to converge on an index for the focus range. One concern about adaptive merging is that the technique requires the first query to pay a significant cost for generating initial runs. The difference in reorganization performance, i.e., the number of queries required to have a key range fully optimized, is due to (1) merging with a high fan-in rather than partitioning with a low fan-out of two or three and to (2) merging a query's entire key range rather than only dividing the two partitions with the query's boundary keys. The difference in the cost of the first query is primarily due to the cost of sorting the initial runs.

Our goal in creating hybrid algorithms is to "merge" the best qualities of adaptive merging and database cracking. In particular, we strive to maintain the lightweight footprint of cracking, which imposes a minimal overhead on queries, and at the same time quickly achieve query performance comparable to fully sorted arrays or indexes as adaptive merging manages to achieve.

Data Structures. Before presenting our hybrid algorithms, we describe the underlying data structures used in the implementation. Each logical column in our model is represented by multiple pairs of arrays containing row identifiers and key values (as opposed to a single array of pairs). Two data structures organize these pairs of arrays. All tuples are initially assigned to arbitrary unsorted "*initial partitions*". As a side-effect of query processing, tuples are then moved into "*final partitions*" representing merged ranges of key values. Once all data is consumed from an initial partition P, then P is dropped. These are like adaptive merging's run and merge partitions except that we do not necessarily sort the key values, plus the whole architecture has been redesigned for column-stores.

Each initial partition uses a table of contents to keep track of the key ranges it contains. Finally, a single master table of contents — the adaptive index itself — keeps track of the content of both the initial and final partitions. Both tables of contents are updated as key value ranges are moved from the initial to the final partitions. The data structures and physical organization used is that of partial sideways cracking [12]. The final partitions respect the architecture of [12] such as we can reuse the techniques of partial sideways cracking for complex queries, updates and partial materialization.

Select Operator. As with original database cracking, the hybrids presented here result in a new select operator each. In our MonetDB setting, the input for a select operator is a single column and a filtering predicate while the output is a set of rowIDs. In the case of adaptive indexing, all techniques collect all qualifying tuples for a given predicate in a contiguous area via data reorganization. Thus, we can return a *view* of the result of a select operator over the adaptive index, as opposed to materializing the complete result as in plain MonetDB.

Complex Queries. The qualifying rowIDs can be used by subsequent operators in a query plan for further processing. Our hybrids maintain the same interfaces and architecture as with sideways cracking [12] that enable complex queries for adaptive indexing in a column-store. The main idea is that the query plans use a new set of operators that include steps for adaptive tuple reconstruction to avoid random access caused by the reorganization steps of adaptive indexing. Thus, we focus on the underlying improvements and data structures of the select operators in this paper while all techniques, query plans and optimizations for other operators can be found in [12].

Adapting Adaptive Merging. For the sake of an apples-to-apples comparison we have also adapted adaptive merging's design in the column-store environment, i.e., using fixed-width dense arrays. This resulted in several optimizations inspired by our experience in MonetDB and DB cracking but at the same time we also had to leave out several design choices originally described for adaptive merging as they would be inefficient in a column-store setting. In a separate line of work we are studying our techniques in a disk based setting using b-tree structures and slotted pages.

3.1 Strategies for organizing partitions

Our hybrid algorithms follow the same general strategy as our implementation of adaptive merging while trying to mimic cracking-like physical reorganization steps that result in crack columns in the sideways cracking form. The first query of each column splits the column's data into initial partitions that each fit in memory (or even in the CPU cache).[1] As queries are processed, qualifying key values are then moved into the final partitions. Tables of contents and the adaptive index are updated to reflect which key ranges have been moved into the final partitions so that subsequent queries know which parts of the requested key ranges to retrieve from final partitions and which from initial partitions.

Our hybrid algorithms differ from original adaptive indexing and from each other in *how and when* they incrementally sort the tuples in the initial and final partitions. We consider three different ways of physically reordering tuples in a partition; (a) sorting, (b) cracking and (c) radix clustering.

Sorting. Fully sorting initial partitions upon creation comes at a high up-front investment. The first query has to pay a significant cost for sorting all initial partitions (runs). Fully sorting a final partition is typically less expensive, as the amount of data to be sorted at a single time is limited to the query's result. The gain of exploiting sort is fast convergence to the optimal state. Adaptive merging uses sorting for both the initial and the final partitions.

Cracking. Database cracking comes at a minimal investment, performing only at most two partitioning steps in order to isolate the requested key range for a given query. Subsequent queries exploit past partitioning and need to "crack" progressively smaller and smaller pieces to refine the ordering. Contrary to sorting, the overhead is slow convergence to optimal performance. Unless we get an exact hit, some amount of reorganization effort has to be incurred. In our hybrids, if the query's result is contained in the final partitions, then the overhead is small as only a single or at most two partitions need to be cracked. If the query requires new values from the initial partitions though, then potentially every initial partition needs to be cracked causing a significant overhead.

For our hybrids we have redesigned the cracking algorithms such that the first query in a hybrid that cracks the initial partitions is able to perform the cracking and the creation of the initial partitions in a single monolithic step as opposed to a copy step first and then a crack step in original cracking.

Radix-clustering. As a third alternative, we consider a lightweight single-pass "best effort" (in the sense that we do not require equally sized clusters) radix-like range-clustering into 2^k clusters as

[1]In our design for in-memory column-store processing we found that using partitions of size L1/2 was optimal.

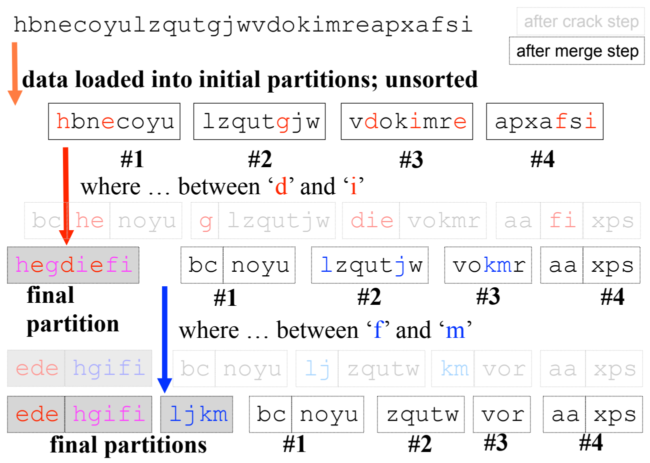


Figure 4: Hybrid Crack Crack (HCC).

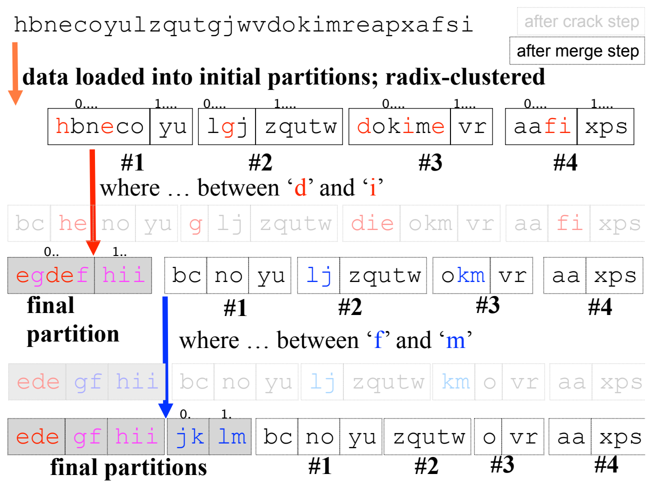


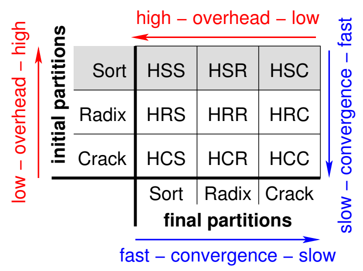
Figure 5: Hybrid Radix Radix (HRR).

follows. Given the smallest ($\underline{v}$) and largest ($\overline{v}$) value in the partition, we assume an order-preserving injective function $f : [\underline{v}, \overline{v}] \to \mathbb{N}_0$, with $f(\underline{v}) = 0$, that assigns each value $v \in [\underline{v}, \overline{v}]$ a numeric code $c \in \mathbb{N}_0$. For instance, we use $f(v) = v - \underline{v}$ for $[\underline{v}, \overline{v}] \subseteq \mathbb{Z}$, and $f(v) = A(v) - A(\underline{v})$ for characters $\underline{v}, v, \overline{v} \in \{$'A', …, 'Z', 'a', …, 'z'$\}$, where $A()$ yields the character's ASCII code. Examples are given in Figure 15 in Appendix D.

With this, we perform a single radix-sort step on $c = f(v)$, using the k most significant bits of $\overline{c} = f(\overline{v})$, i.e., the result cluster of value v is determined by those k bits of its code c that match the positions of the k most significant bits of largest code $\overline{c}$. Investing in an extra initial scan over the partition to count the actual bucket sizes, we are able to create in one single pass a continuous range-clustered partition. With a table of contents that keeps track of the cluster boundaries, the result is identical to that of a sequence of cracking operations that cover all $2^k - 1$ cluster boundaries.

3.2 Hybrid algorithm design

We can apply each of the above techniques on both the initial and the final partitions, and combine them arbitrarily. In this way, we can create $3*3 = 9$ potential hybrid algorithms. We refer to them as *Hybrid Sort Sort* (*HSS*), *Hybrid Sort Radix* (*HSR*), *Hybrid Sort Crack* (*HSC*), *Hybrid Radix Sort* (*HRS*), *Hybrid Radix Radix* (*HRR*), *Hybrid Radix Crack* (*HRC*), *Hybrid Crack Sort* (*HCS*), *Hybrid Crack Radix* (*HCR*), *Hybrid Crack Crack* (*HCC*). We note that *HSS*, i.e., using sorting for both partition types, represents the column-store implementation of the original adaptive merging algorithm (see example in Figure 3). Given that we aim at avoiding the high investment of sorting all initial partitions, we do not consider the *HS** variants any further, focusing on the remaining 6 variants *HR** and *HC**.


high – overhead – low
low – overhead – high
initial partitions
slow – convergence – fast
final partitions
fast – convergence – slow

Sort	HSS	HSR	HSC
Radix	HRS	HRR	HRC
Crack	HCS	HCR	HCC
	Sort	Radix	Crack

Figure 4 depicts an example for Hybrid Crack Crack (HCC). With the first query, the data is loaded into four initial partitions that hold disjoint row ID ranges. (For ease of presentation, we omit the row IDs in Figures 2 – 5.) Then, each initial partition is cracked on the given key range "d – i", and the qualifying key values are moved into the gray-shaded final partition that also forms the result of the first query. The second query's key range "f – m" partly overlaps with the first query's key range. Hence, the final partition holding keys from "d – i" is cracked on 'f' to isolate the overlapping range "f – i". Then, all initial partitions are cracked on 'm' to isolate keys from "j – m" and move them into a new value partition. The result of the second query is then available as ranges "f – i" and "j – m" in the respective final partitions. Subsequent queries are processed analogously to the second query.

Likewise, Figure 5 depicts an example for Hybrid Radix Radix (HRR). With the first query, the data is loaded into four initial partitions, and radix-clustered on the $k = 1$ most significant bits of the codes given in Figure 15 (a) in Appendix D. Then, the clusters that hold the requested key range boundaries ('d' & 'i') are cracked on these, and the qualifying key values are moved into the gray-shaded final partition. To support future look-ups, the newly created final partition is radix-clustered on the $k = 1$ most significant bits of the codes given in Figure 15 (b). The second query's range "f – m" completely covers the "1.." cluster ("h – i") and partly overlaps with the "0.." cluster ("d – g") of the final partition. Hence, the former is completely included in the result, and the latter is cracked on 'f' to isolate range "f – g". Then, all initial partitions are cracked on 'm' to isolate keys from "j – m" and move them into a new final partition. The latter is radix-clustered on the $k = 1$ most significant bits of the codes given in Figure 15 (c). The result of the second query is then available as ranges "f – g", "h – i", "j – k", and "l – m" in the respective final partitions. Subsequent queries are again processed analogously to the second query.

Compared to HCC and HRR, variations Hybrid Crack Radix (HCR) and Hybrid Radix Crack (HRC) swap the treatment of final partitions during the merge step, i.e., HCR uses cracking for initial partitions, but radix-cluster for final partitions, while HRC use radix-cluster for initial partitions, but cracking for final partitions.

Variations Hybrid Crack Sort (HCS) and Hybrid Radix Sort (HRS) invest in sorting each final partition on creation, just as original adaptive merging (respectively Hybrid Sort Sort) does; cf. Figure 3.

Parameters Affecting Performance. An adaptive indexing technique is characterized (a) by how lightweight adaptation it achieves, i.e., the cost of the first few queries representing a workload change [5] and (b) by how fast in terms of time and queries needed it converges to the performance of a perfect index [5]. Several parameters may affect performance for adaptive indexing, i.e., query selectivity, data skew, updates, concurrent queries, disk based processing etc. For example, a hybrid variation that uses sorting will have the edge in an environment with concurrent queries or with limited memory as less queries will require physical reorganization. On the other hand, such a hybrid will suffer in an environment with excess updates. In the rest of the paper, we will study our algorithms with a focus on workload related parameters leaving further analysis to future work. In the appendix section there is more discussion on issues related to updates and concurrency control.

4. EXPERIMENTAL ANALYSIS

In this section, we continue with a detailed experimental evaluation. We implemented adaptive merging and all hybrids in MonetDB, fully integrating the implementation within the cracking module of MonetDB. All algorithms share code and data structures as much as possible, ensuring that all common steps are executed with identical code. As with original database cracking, adaptive merging and the hybrids resulted in new select operators that perform, register and maintain any necessary physical reorganization actions over the data on-the-fly.[2]

Experimental set-up. We use a 2.4 GHz Intel Core2 Quad CPU equipped with 32 KB L1 cache per core, two 4 MB L2 caches (each shared by 2 cores), and 8 GB RAM. The machine runs Fedora 12.

Each experiment measures the time needed to process a sequence of queries following a specific workload pattern over a given data set. Queries are of the following form.

```
select A from R where A > low and A < high
```

We purposely keep queries simple so we can isolate adaptive behavior using metrics such as (a) the response time for each individual query, (b) the cumulative time to answer the complete query sequence and (c) how fast performance reaches the optimal level, i.e., retrieval performance similar to a fully optimized index. Behavior for more complex queries follows the same patterns as shown for original cracking in [12]. Thus here, these simple queries translate to simple plans that essentially contain a single select operator which allows us to fully compare the various alternative adaptive indexing select operators without any other "noise".

In addition to comparing the adaptive indexing methods against each other, we also compare them to two classic approaches: the simple scan of a table without indexes, and a lookup using a full index. The first is typical of a system that has not invested in index optimization for a given workload. There is neither overhead nor performance improvement. The second embodies the case where a full investment is made. In our implementation over a column-store, we use a fully sorted array as an index. The very first query on a column A will first create a copy, A_s, and then perform a complete sort of A_s. All subsequent queries can employ fast binary search on A_s to locate A values.

Analysis. In our first set of experiments we use a column of 3×10^8 tuples with randomly distributed integer values in $[0, 10^8)$. We fire 10^3 queries, where each query asks for a randomly located range with 10% selectivity. This widely spread access pattern necessarily requires many queries to build-up index information. In Appendix A, we show how more focused workloads result in much faster optimization of the relevant key ranges in adaptive indexing.

Figure 7 shows, for all algorithms, the elapsed times of individual queries within the execution sequence. All axes are logarithmic so that the costs of initial queries can be easily compared.

In principle, all strategies other than scans eventually result in fully sorted storage. They differ in *how* and *when* the required $n \log n$ comparisons are performed. For a detailed discussion of adaptation *phases*, we refer readers to [5]. Here we focus upon the trade-off between two measures: overhead over a simple scan for the first queries vs. fast convergence to a complete index.

[2]The complete code base is part of the latest release of MonetDB available via http://monetdb.cwi.nl/.

Scan and Sort. Figure 7(a) shows how the scan and full sort approach behave. Scan has a stable performance as it always needs to do the same job, i.e., scan the entire column. Only the first query is slightly slower as it brings all data from disk. With sort on the other hand, we invest in fully sorting the column during the first query which allows us to exploit binary search for each subsequent query. This represents a major burden for the first query, which happens to trigger the sort, but all queries after that are very fast.

Original Cracking. Database cracking in its original form has a much more smooth and adaptive behavior as seen in Figure 7(a). The first query is less than two times slower than a scan while progressively with each incoming query performance becomes better and better. Already with the second query, cracking matches the scan performance and after the third it exceeds it. By the time the 10th query in the query sequence arrives, cracking is one order of magnitude faster than a scan and keeps improving. The major benefit of cracking is that it can adapt online without disrupting performance. With full index creation like in the sort approach, the index preparation investment is such a big cost that it is only viable if it happens a priori. In turn, this can only happen if we assume enough idle time and sufficient workload knowledge.

Adaptive Merging. As shown in Figure 7(a), cracking achieves a smoother adaptation behavior than adaptive merging, but adaptive merging converges much more quickly. The first 6 queries of adaptive merging are all more expensive than scans, with the first query nearly 5 times slower than a scan. For cracking, only the first query is more expensive than a scan, and even that is only twice as slow. On the other hand, cracking still has not reached optimal performance by the end of the workload, whereas adaptive merging answers the 7th query using only the final partition, matching the performance of a full index. Appendix A gives further insights for this behavior using cumulative metrics. In addition, observe that the optimal performance of adaptive merging is slightly better than that of a full index. This is due to better access patterns caused by the underlying data organization; as mentioned in Section 3 adaptive merging and the hybrids store data as collections of small columns as opposed to a single column in plain MonetDB.

Motivation for Hybrids. We view cracking and adaptive merging as the extremes of a spectrum of adaptive indexing, in that cracking is a very *lazy* adaptive indexing approach while adaptive merging a very *eager* one. With the hybrids proposed in this paper, we investigate the space in between. The ideal hybrid would achieve the smooth adaptation and low overhead of database cracking with the fast convergence of adaptive merging. Figure 6 breaks down the first query costs of adaptive merging to help motivate our hybrid choices. It is clear that sorting the initial runs accounts for the major part of the cost while merging and initial partition construction (copy, rowID materialization, etc.) are much more lightweight. It takes 40 seconds to do the sorting step alone, whereas cracking completes processing of the first query including both data copying and reorganization in only 15 seconds (Fig. 7(a)).

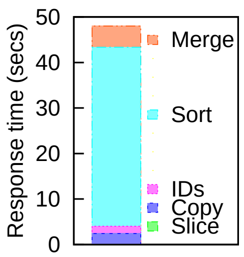


Figure 6: Cost breakdown of first query.

The Hybrids. Figures 7(b) and (c) show that each hybrid occupies a different spot in the space between adaptive merging and cracking depending on how eager its adaptation steps are. Figures 7(b) and (c) use exactly the same x-axis and y-axis as Figure 7(a) so one can compare any hybrid directly with the basic adaptive indexing approaches as well as with scan and full sort.

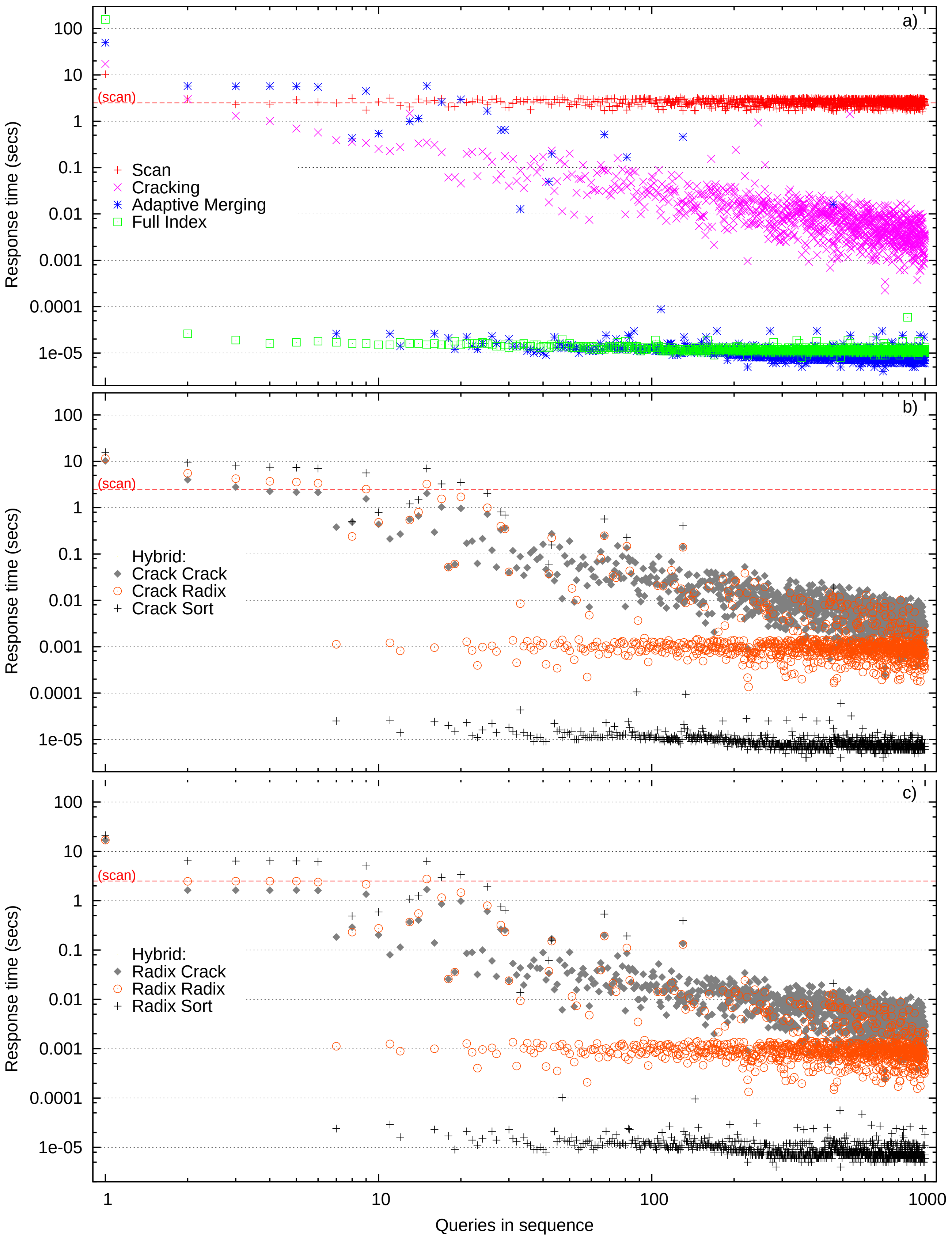


Figure 7: Performance in a Dynamic Workload.

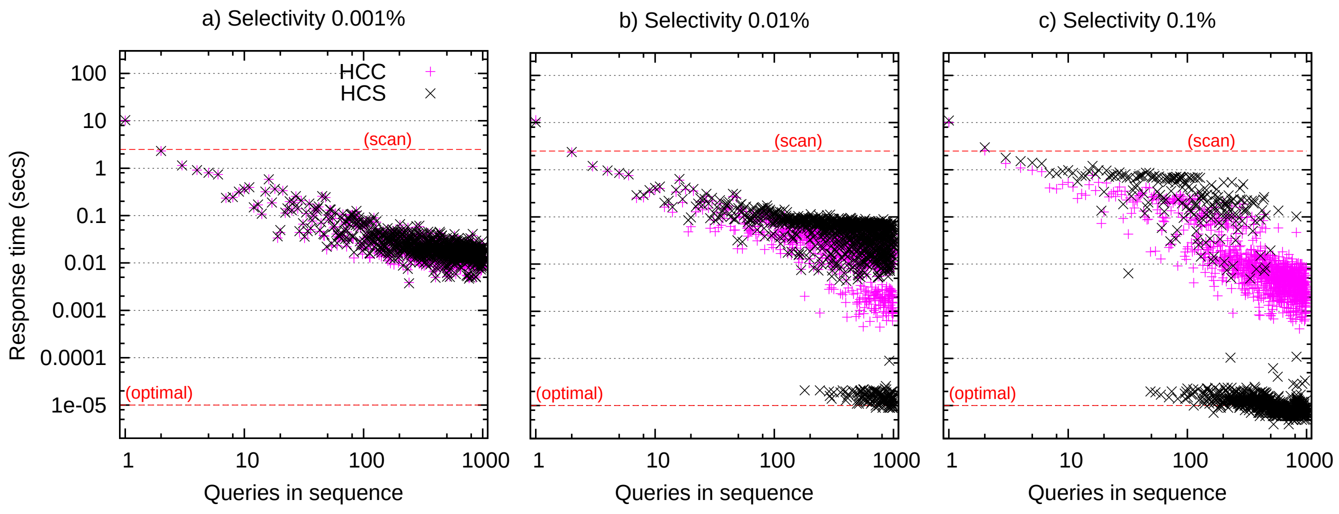


Figure 8: Effect of Selectivity.

Cracking Initial Partitions. The first set of hybrids use cracking on the initial partitions, followed by, depending on the algorithm, either cracking, radix clustering or sorting of the final partition. Figure 7(b) shows their performance.

Lowest Investment. The Crack Crack variation improves heavily over plain cracking by significantly reducing the first query cost to a level equivalent to the cost of a scan; the overhead of adaptive indexing vanishes! The main source of this improvement is that contrary to original cracking, the hybrid operates on batches of the column at a time and it uses the new cracking algorithm that creates and cracks initial partitions in one go. In addition, compared to a scan an adaptive indexing select operator does not need to materialize the resulting rowIDs; it simply returns a view over the adaptive index where after the data reorganization, qualifying tuples appear in a contiguous area. After the first query, the Crack Crack hybrid maintains the smooth behavior of cracking, but it does not achieve the fast convergence of adaptive merging.

Faster Adaptation. The Crack Sort hybrid tries to overcome this issue and uses sorting of the final partitions to speed up adaptation, i.e., it mimics how adaptive merging handles merged values in an eager way. The result shown in Figure 7(b) is that the Crack Sort achieves similar convergence as adaptive merging. Compared with adaptive merging it heavily improves in respect to the initialization costs. For Crack Sort the first query is only marginally slower than a scan while for adaptive merging the first query is 5 times slower than a scan. In other words, Crack Sort achieves the same minimal cost for the first query as with original cracking combined with the fast adaptation of adaptive merging.

Balancing Gains and Investments. Crack Sort sacrifices slightly the ultra fast first query achieved by the Crack Crack and in addition it still is slower than the scan for the first 10 queries while Crack Crack is never slower than a scan. The reason for that is the investment in sorting the final partitions which of course brings the fast adaptation as well. Our next hybrid, the Crack Radix also shown in Figure 7(b) achieves a nice balance between the previous two hybrids. This new hybrid invests in clustering as opposed to sorting the final partitions. Using radix clustering we can achieve range-clustering at a minimal cost. In Figure 7(b) Crack Radix uses $k = 9$, i.e., creates 512 clusters at a time, which we found to be a good balance regarding the investment in clustering and the gain we get. As shown in Figure 7(b), Crack Radix achieves similar performance as Crack Crack towards the beginning of the workload, i.e., a very lightweight first query while the next few queries are all faster than a scan and keep improving. Then similarly to adaptive merging and Crack Sort it achieves a significant boost in performance by Query 10 which is maintained as more queries arrive. Even though the ultimate performance of Crack Radix does not match the performance achieved by adaptive merging or Crack Sort, the behavior is the same and the actual performance is several orders of magnitude faster than original cracking and of course that of a plain scan. And all these improvements come at zero cost since it does not impose any overhead for the first part of the workload sequence. Clustering partitions is more eager than simply cracking but more lazy than fully sorting and thus we achieve a nice balance in performance gains and investments.

Clustering Initial Partitions. The second set of hybrids uses Radix clustering for the initial partitions as opposed to cracking that we used for the first set described above. The net result as shown in Figure 7(c) is that all hybrid variations in this set become a bit more eager during the first query which has to perform the clustering of the partitions. We again use $k = 9$, creating 512 clusters at a time. Thus, the first query of all hybrids in Figure 7(c) is slightly more expensive than that of the hybrids in Figure 7(b) that use cracking for initial partitions. The gain though is that in Figure 7(c) we see a significant improvement for the queries that follow the first, i.e., the first 10 queries that typically have to perform a significant amount of merging and thus in the case of the Radix-* hybrid variations, they can exploit the extra information gains by clustering to more quickly locate qualifying values.

Effect of Selectivity. In our next experiment we demonstrate the effect of selectivity. For ease of presentation we show only the HCC and HCS hybrids against the always scan and full index approach. Scan and full index are naturally immune to selectivity. Figure 8 depicts the results. It uses the same data set and the same y and x axes as in Figure 7. The first observation is that with smaller selectivity it takes more queries to reach optimal performance. This is because the chances of requiring merging actions are higher with smaller selectivity as less data is merged with any given query.

In addition, we observe that with smaller selectivity the difference in convergence is less significant between the "lazy" HCC and the more "active" HCS. At the same time, the lazy algorithms maintain their lightweight initiation advantage. Original cracking and adaptive merging show similar behavior, i.e., cracking resembles HCC behavior and adaptive merging resembles HCS behavior.

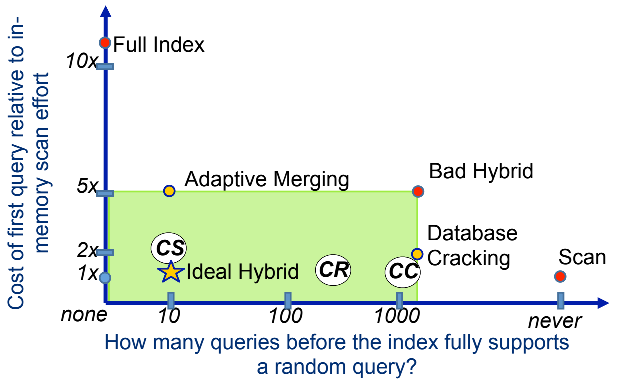


Figure 9: Our hybrids in the adaptive indexing research space.

One optimization to increase convergence speed is to make the algorithms even more active by artificially forcing more active merging steps as we have discussed in [7]. We refrain from exploring this option here as it requires optimization steps beyond the scope of this paper regarding how much extra merging to do and when.

Summary. Figure 9 shows where the new hybrids lie in the research space defined by adaptive indexing. It uses the same layout as Figure 1 in Section 1. For simplicity, we draw only the HC* hybrids. The net result is that we have come very close to the ideal hybrid and heavily improved over both original cracking and adaptive merging. We did so from two angles. HCR has the lightweight footprint of a scan query, and it can still reach optimal performance very quickly while HCS is only two times slower than a scan for the first query but reaches optimal performance as quickly as adaptive merging. In fact, HCS has both the lightweight first query of original cracking and the quick adaptation of adaptive merging. The benefit of HCR is that it provides a more smooth adaptation, never being slower than a scan, as depicted in Figure 7.

Thus, at this point HCS and HCR are both valid choices. HCR is to be used when we want the most lightweight adaptation, while HCS is to be used when we want the fastest adaptation.

Towards the Ideal Hybrid. The next challenge in future work is to approach the ideal hybrid even more by optimizing the current hybrids. The main idea we see as a promising research path is to provide the functionality to dynamically combine steps from different hybrids into a single algorithm. A simple example is a hybrid that for the first part of the workload performs like HCR so that it has a lightweight footprint, but as soon as performance improves over a threshold, such that we can afford some pro-active steps, it switches into HCS-like processing or increases the degree of clustering, in order to more quickly and eagerly optimize the index.

In addition, there are several design choices we can further adopt and adapt from original adaptive merging. For example, initial partitions may shrink in size as we merge more and more values. Furthermore, we can exploit multi-level merging to further reduce costs for reorganization actions in initial partitions. We plan to further investigate such opportunities (see Appendix).

5. CONCLUSIONS

In summary, traditional query processing has relied on index-to-index navigation with index creation as separate workload guided by database administrators or automatic tools. An alternative is query processing based on non-indexed scans, perhaps improved by columnar storage, shared scans, zone filters, etc. Recent work has proposed adaptive indexing, which offers intermediate options: database cracking, which adds little cost to scan-based query processing yet finalizes a cracker index only after thousands of queries, and adaptive merging, which burdens scans with the logic and costs of the phases in external merge sort but quickly transforms a partitioned B-tree with many initial runs into one equivalent to a traditional B-tree index.

Our initial experiments yielded an insight about adaptive merging. Data moves out of initial partitions and into final partitions. Thus, the more times an initial partition has already been searched, the less likely it is to be searched again. A final partition, on the other hand, is searched by every query, either because it contains the results or else because results are moved into it. Therefore, effort expended on refining an initial partition is much less likely to "pay off" than the same effort invested in refining a final partition.

Our new hybrid algorithms exploit this distinction and apply different refinement strategies to initial versus final partitions. They thus combine the advantages of adaptive merging and database cracking, while avoiding their disadvantages: *fast convergence, but hardly any burden is added to a scan.* This appears to enable an entirely new approach to physical database design. The initial database contains no indexes (or only indexes for primary keys and uniqueness constraints). Query processing initially relies on large scans, yet all scans contribute to index optimization in the key ranges of actual interest. Due to the fast index optimization demonstrated in our implementation and our experiments, query processing quickly transitions from relying on scans to exploiting indexes.

Due to the low burden to create and optimize indexes, physical database design using this approach offers the best of both traditional scan-based query processing and traditional index-based query processing, without the need for explicit tuning or for a workload for index creation. Modern semiconductor storage with its very short access latency encourages index-to-index navigation as the principal paradigm of query execution; adaptive indexing with fast convergence and low overhead for index optimization might turn out an excellent complement to modern database hardware.

6. REFERENCES

[1] N. Bruno and S. Chaudhuri. Physical design refinement: the 'merge-reduce' approach. *ACM TODS*, 32(4):28:1–28:41, 2007.

[2] S. Chaudhuri and V. R. Narasayya. Self-tuning database systems: A decade of progress. *VLDB*, pages 3–14, 2007.

[3] S. J. Finkelstein, M. Schkolnick, and P. Tiberio. Physical database design for relational databases. *ACM TODS*, 13(1):91–128, 1988.

[4] G. Graefe. A Survey of B-tree Locking Techniques. *ACM TODS*, 35(2):16:1–16:26, 2010.

[5] G. Graefe, S. Idreos, H. Kuno, and S. Manegold. Benchmarking adaptive indexing. *TPCTC*, pages 169–184, 2010.

[6] G. Graefe and H. Kuno. Adaptive indexing for relational keys. *SMDB*, pages 69–74, 2010.

[7] G. Graefe and H. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. *EDBT*, pages 371–381, 2010.

[8] T. Härder. Selecting an optimal set of secondary indices. *Lecture Notes in Computer Science*, 44:146–160, 1976.

[9] C. A. R. Hoare. Algorithm 64: Quicksort. *Comm. ACM*, 4(7):321, 1961.

[10] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. *CIDR*, pages 68–78, 2007.

[11] S. Idreos, M. L. Kersten, and S. Manegold. Updating a cracked database. *SIGMOD*, pages 413–424, 2007.

[12] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column stores. *SIGMOD*, pages 297–308, 2009.

[13] M. Lühring, K.-U. Sattler, K. Schmidt, and E. Schallehn. Autonomous management of soft indexes. *SMDB*, pages 450–458, 2007.

[14] A. N. S. Praveen Seshadri. Generalized partial indexes. *ICDE*, pages 420–427, 1995.

[15] M. Stonebraker. The case for partial indexes. *SIGMOD Record*, 18(4):4–11, 1989.

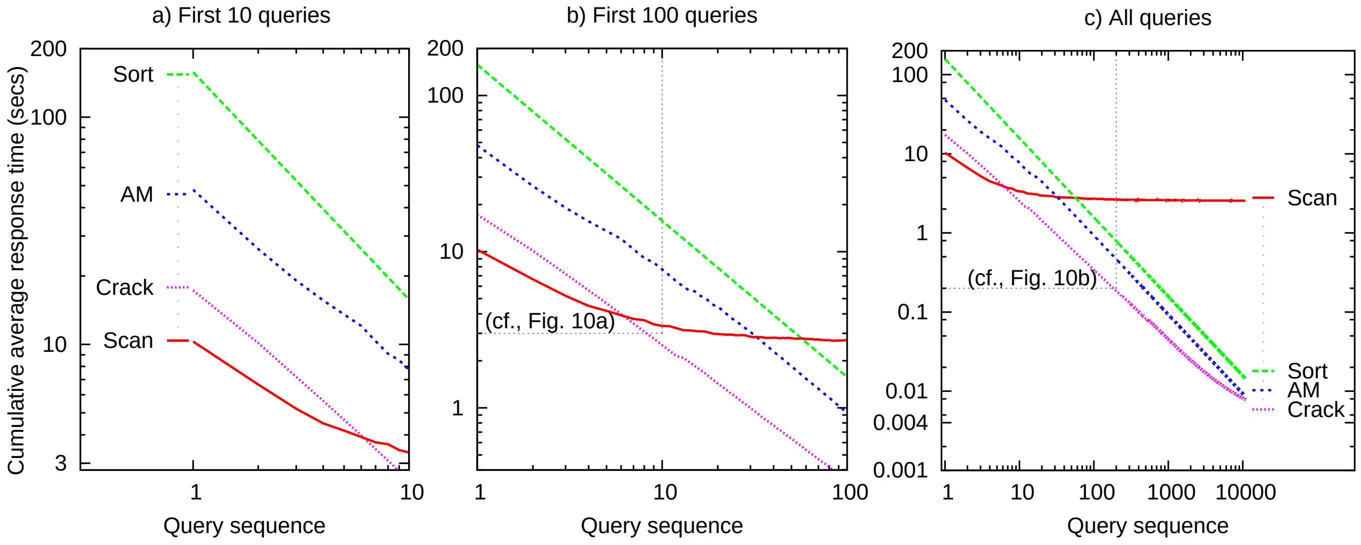


Figure 10: Adaptive Indexing Vs Off-line Indexing and Scans.

APPENDIX

Here, we provide more discussion on several topics to complement the discussion in the main paper. We provide additional experimental analysis, e.g., on focused workloads. We provide discussion on more complex query scenarios, e.g., updates and multi-column indexes, as well as a more extensive future work discussion.

A. ADDITIONAL EXPERIMENTS

A.1 Cumulative Average Costs

In the main paper, we demonstrated our results in terms of the performance achieved by each single query independently. Another interesting factor is what happens if we consider the workload as a whole, i.e., the cumulative costs to run a sequence of queries. Here we choose the cumulative average metric to demonstrate these effects as the workload develops for a sequence of 10^4 queries, i.e., $\bar{T}(i) = \sum_{j=1}^{i} t(q_j)/i$ for $i \in \{1,\ldots,10^4\}$.

Figure 10 depicts the cumulative average response time for original adaptive indexing against traditional approaches. Part c) shows the whole range of 10^4 queries, while parts b) and a) magnify the results for the first 200 and first 10 queries, respectively.

The very first query for each algorithm represents the main difference in terms of investments needed with a new workload or a workload shift. Then we see that as the workload develops each algorithm behaves differently. For example, the new information compared to the main paper, is that cracking needs 7-8 queries to improve over total scan costs, while adaptive merging needs 30 queries and sorting needs 60 queries. On the other hand, we also see the faster pace with which adaptive merging improves over cracking while sorting improves even faster. As we have discussed in the main paper as well, this is due to the more complete index structures that adaptive merging can enjoy over cracking. The more complete data structures though come with an initialization cost for both adaptive merging and sorting.

This graph helps to motivate the hybrids. It is evident that the initial investment of adaptive merging and full sorting needs a good amount of effort until it can be amortized by future gains. In other words, even if we ignore the disruption for the first query, we still need in this example 10^4 queries until we can amortize the costs and start benefiting over plain cracking. On the other hand, we have demonstrated clearly that adaptive merging reaches the ultimate per query performance very fast while cracking needs a few thousands of queries and when adaptive merging is at the absolute optimal performance cracking is still several orders of magnitude away.

The hybrid algorithms presented in this paper significantly improved over both algorithms by balancing the design choices. Figure 11 shows how the hybrids behave under the same metric (cumulative average) for the first part of the workload. This helps us see past the first query and see how the behavior evolves especially for the initial queries as it is evident that these are the queries that incur the major disruption in performance if indexes are to be built online due to lack of workload knowledge and idle time.

Figure 11(a) shows how Hybrid Crack Crack achieves performance similar to a scan for the first query and then it follows the behavior of plain cracking, representing a zero overhead adaptive indexing approach. Hybrid Crack Sort improves over plain cracking for the first query and then it follows a behavior which lies between plain cracking and adaptive merging. It improves with a similar pace as adaptive merging but due to the sorting steps it poses an overhead for the first couple of queries compared to cracking.

Similar discussions and observations apply for the rest of the hybrids in Figures 11(b) and (c). As we observed in the main paper, Hybrid Crack Radix provides a quite balanced behavior and as we see here (Figure 11(c)), this behavior is evident with the cumulative average metric as well. Crack Radix achieved both a low overhead for the first query and also a low overhead for the first batch of queries. This is because, contrary to Crack Sort and Adaptive Merging, the Crack Radix uses lightweight clustering operations after the first query to reorganize final partitions.

Thus, similarly to our summary in the experiments of the main part of the paper, the Crack Radix hybrid represents our most lightweight hybrid that can still adapt very fast while the Crack Sort represents a reasonably lightweight hybrid that can adapt extremely fast. Both hybrids are significantly better than their ancestors, database cracking and adaptive merging.

A.2 Focused Workloads

In the main paper, we studied simple focused workloads where the pattern was that we simply restricted our interest to a specific

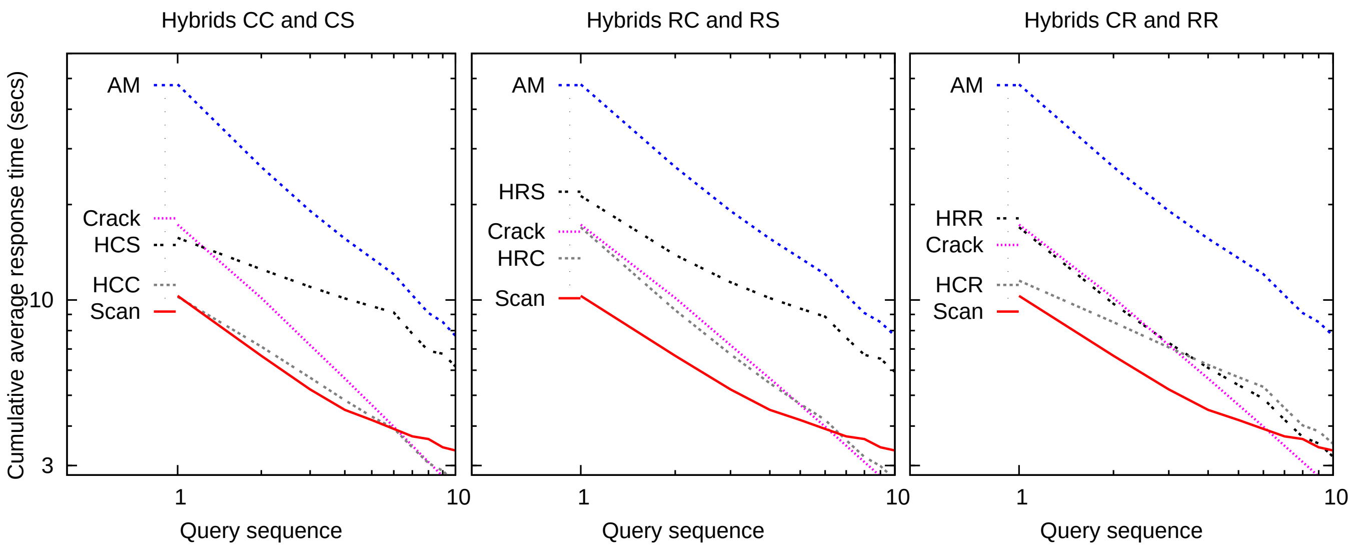


Figure 11: Adaptive Indexing Hybrids vs Cracking and Adaptive Merging (AM).

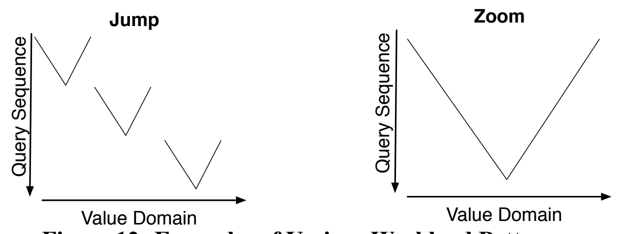


Figure 12: Examples of Various Workload Patterns.

part of the domain, i.e., to *x*% of the possible ranges we could query. Here, we study more complex and general patterns where the focus gradually shifts into target areas based on a specific pattern. These kind of scenarios represent more realistic workloads where the user explores and analyzes the data based on on-the-fly observations.

Jump. In this experiment, we analyze a workload with a jumping focus, i.e., the focus is initially on 20% of the data set, then after 1000 queries it jumps to a different 20%, then to a different 20% and so on. This represent a behavior where the user will eventually study the whole data set but we do not need the complete data set optimized in one go or at all times. Figure 12 shows such a workload; as the query sequence evolves the workload focuses on a small area of the domain and then the focus jumps to a different area.

Zoom. The zoom workload pattern reflects a zooming behavior where progressive understanding during query processing leads to the actual point of interest, i.e., the workload stepwise zooms into shrinking areas of interest. Figure 12 shows such an example. Here, the first 2000 queries are randomly spread over the whole key range, the next 2000 queries focus on only the center 80% of the key range, the next 2000 queries focus on the center 60% and so on, until in the $5th$ step the target area has shrunk to the center 20% of the key range.

Discussion. Figure 13 shows the results. It is meant to provide a high level visualization to provide insights of how adaptive indexing works in a variety of scenarios. For simplicity of presentation, we include results for the Crack Crack hybrid only. We use columns of $4 * 10^8$ tuples and a query selectivity of 10%.

Sort and scan are *insensitive* to the workload and thus they provide the same performance across the range of workloads tested. For example, scan will always scan the same amount of data while sort will always do a complete sort to exploit binary search from there on. Thus, scan has always the same high cost, while sort will always pay a heavy initial cost. Adaptive indexing though provides a very different behavior trying to always exploit the workload patterns in order to improve. This is true for all adaptive indexing techniques with the hybrid having the edge due to the more lightweight adaptation as we discussed in the main paper. We do not provide figures that focus on the initial part of the sequence, comparing initial costs, as this is the same behavior seen before. The goal here is to present an overview of how the adaptive indexing algorithms adapt to the changing workload patterns. For example, notice how for the Zoom workload all adaptive indexing techniques improve faster (in terms of queries processed) to the optimal performance levels compared to a random workload as well as improving even further. The fastest query in the Zoom workload is one order of magnitude faster than the fastest in the Random one. By focusing the workload on specific key ranges, adaptive indexing can improve its knowledge over these key ranges more quickly. Similar observations stand for the Jump workload pattern as well. In this case, the shift from one key range to another is visible by a few high peaks each time representing a workload change. Once the workload focuses again, performance improves very fast and in most cases performance stays way faster than the plain scan approach. Without any external knowledge and completely in an automatic and self-tuning way, performance improves purely based on the workload.

B. UPDATES & MULTI-COLUMN INDEXES

Updates and multi-column queries are two important milestones towards supporting more complex workloads with adaptive indexing. Updates bring an extra complexity because data we actively reorganize needs to also consider any update changes at the same time. Multi-column queries impose challenges regarding the alignment of values across different columns as we keep physically reorganizing those columns.

Both topics have been extensively studied under the cracking paradigm in [11] and [12]. All hybrid algorithms in this paper use final partitions that are crack columns in the format proposed for sideways database cracking in [12] using sets of multiple small arrays to represent a single logical column. We use exactly the

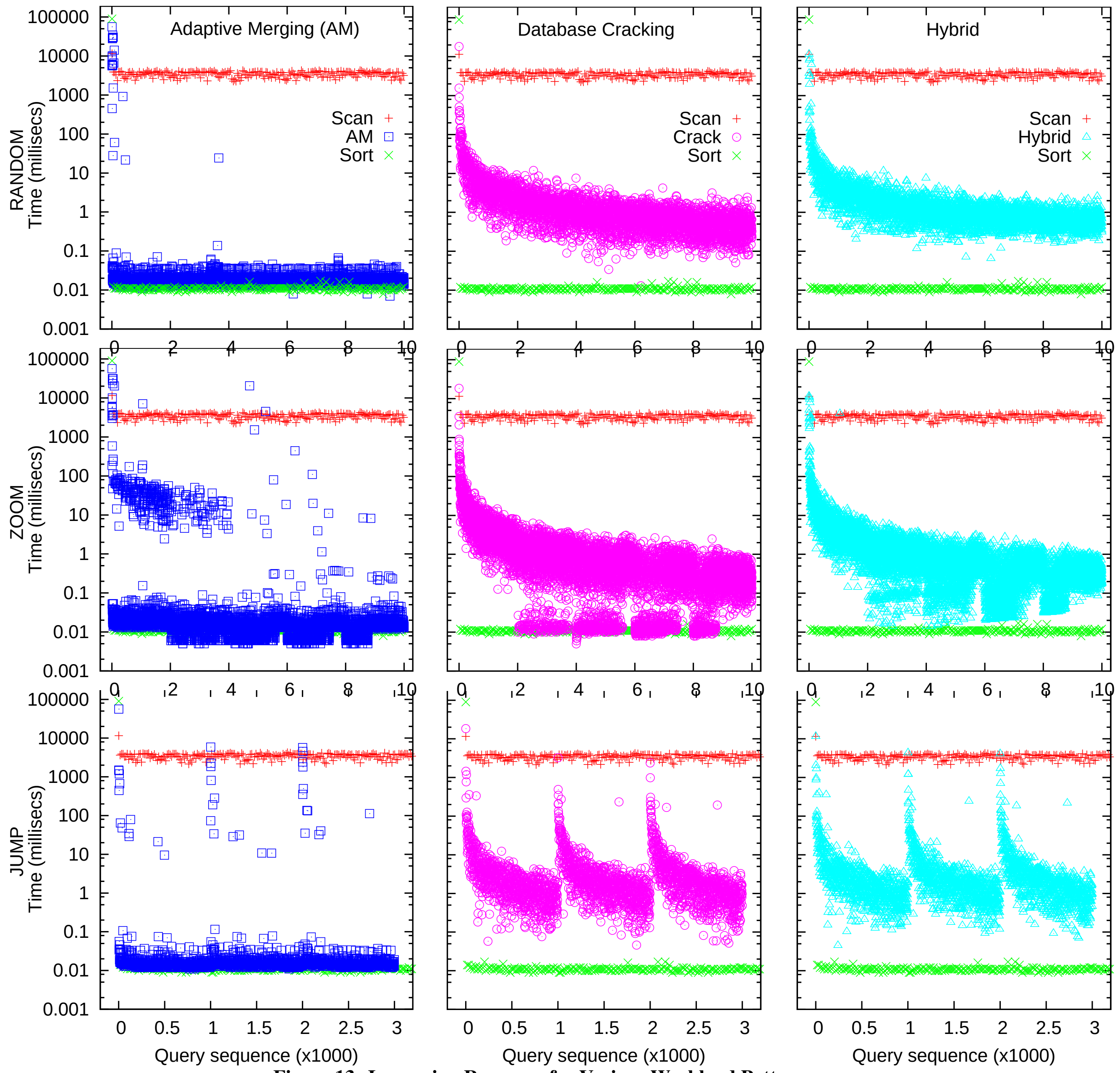


Figure 13: Improving Response for Various Workload Patterns.

same design, data structures and the same code base. This way, all techniques of [11] and [12] transfer directly to our hybrids as well.

Updates. The main idea is that updates are modeled as pending insertions and pending deletions. Conceptually, they can be thought of as yet another partition needing to be merged. When an operator processes a given area of the crack array, it also checks whether there are any pending updates for this area. If there are, it directly applies the updates on-the-fly. The trick is to do this in a lightweight fashion while at the same time maintaining knowledge about how values are organized, i.e., without invalidating index information. The algorithms of [11] exploit the fact that a crack array is not fully sorted and swap/move/insert values only at the edges of the affected areas in an array. Similarly, deletes leave empty spaces at the edges of cracked areas, which are then filled in with future updates on this or neighboring areas.

For the hybrids with sorted partitions, the costs of merging updates are naturally higher. There are a number of ways to tackle this by further exploiting the ideas in [11], i.e., to adaptively "forget" some structure and allow for temporarily non sorted partitions which can be sorted again with future queries. In fact, similar ideas also apply to the hybrids with cracked partitions; once there are too many partitions in a given value range then the maintenance costs become higher and merging part of those partitions becomes the way out. We leave such optimizations for future analysis.

Multi-column Indexes. The main idea in [12] is that the knowledge gained for one column over a sequence of queries is adaptively passed to other columns when the workload demands it, i.e., when multi-column queries appear. The whole design is geared towards improving access to one column given a filtering action in another one, leading to efficient tuple reconstruction in a column-store setting. The net result is that the multi-column index is essentially built and augmented incrementally and adaptively with new pieces from columns and value ranges as queries arrive.

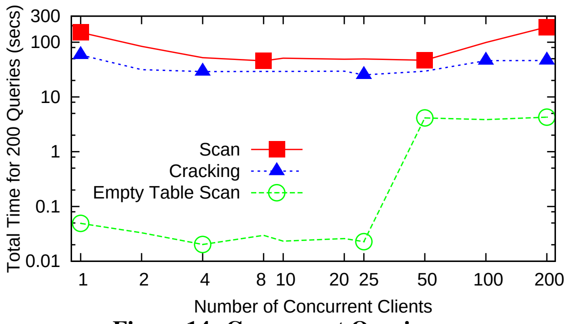


Figure 14: Concurrent Queries.

C. FUTURE WORK

Concurrency control, logging, & recovery. We are currently exploring how to best implement concurrency control, logging, and recovery in an adaptive indexing system.

As we have already shown in [4], distinguishing between database contents and representation has significant implications for concurrency control. Insofar as read queries performing scans or index lookups can invoke side-effect operations that incrementally refine the physical design, "read" queries may seem to introduce significant concurrency control overhead as they refine index structures. However, we observe that the adaptive indexing operations impact only index structures, and never database contents. Furthermore, index refinement is optional and can be done opportunistically or even skipped altogether. These distinctions relax constraints and requirements with regard to concurrency control of adaptive indexing compared to those of traditional explicit index updates and enable new techniques for reducing the performance overhead of concurrency control during structural updates.

Here we give an example of the performance of adaptive indexing under concurrent queries. For this example we use database cracking. We made sure that all critical structures are protected via mutexes, using a very basic approach with per-column granularity. We use a table of 100 million tuples with unique randomly distributed integers and simple range queries of the form `select count(*) from R where v1<A<v2`.

We repeatedly run a sequence of 200 random range queries (10% selectivity), each time increasing the number of concurrent streams. In more detail, we run the serial case where one client runs all 200 queries, one after the other. Then, we use 2 clients that start at the same time and each one fires 100 queries. Then, we repeat the experiment by starting 4 clients at the same time and each one fires 50 queries and so on. The last run uses 200 clients that simultaneously fire 1 query each. To ensure that our results are not dominated by management overhead for handling up to 200 concurrent clients, we first run our query over an empty table, i.e., with pure query execution time being virtually zero.

Figure 14 depicts the results. Database cracking maintains a rather stable performance similar to the scan based approach. Note that with scan, there is no concurrency control mechanisms involved as this is purely a read only workload. Performance improves initially due to temporal locality of accessing data across queries and then as we reach the limits of the hardware with more concurrent clients it increases again. The main point here is that there is no significant degradation in adaptive indexing. If we measure the pure locking cost of the adaptive index, i.e., the cost spent in acquiring and releasing locks it is less than one second. Thus, by using short latching periods and quickly releasing latches as soon as possible, database cracking manages to exploit concurrent queries as opposed to suffering from them. In addition, it is interesting to notice that since cracking gains continuously more and more knowledge about the data, these latching periods become ever shorter which improves performance even more.

Here we demonstrated that even with a very simple locking strategy, database cracking can achieve very good performance in terms of concurrency control. Future work in this area includes the study of techniques that allow locking of *only parts* of the adaptive index where we expect a greater flexibility in terms of how many queries we can run in parallel. In addition, note that due to its more lazy nature plain cracking creates the most concurrency conflicts from all adaptive indexing techniques we presented in this paper. This way, we expect that our more active techniques will have even better performance in this respect.

Traditional disk-based storage. Although in this paper we focus upon an in-memory column-store database, we ultimately strive for efficient operation with any storage media, such as RAM, NV-RAM, flash, and disks. Focusing on a disk-based system would significantly change design trade-offs and we expect the balance between lazy and eager hybrids to be different. For example, effort spent organizing the initial partitions would have a much higher pay-off if I/O costs were incurred when accessing initial partitions.

Page-based Vs. Columnar Storage. Similarly, we plan to study in more detail the side-effects of using a page-based storage scheme. The design and experiments in this paper are based on a column-store system which allows us to have an apples-to-apples comparison of all techniques with a complete implementation inside the original cracking module of MonetDB. However, a number of design issues are expected to be handled more efficiently in a page-based system as opposed to an array-based one. For example, for adaptive merging and the hybrid algorithms, we can probably make the initial part of the query sequence a bit more lightweight by more easily removing data from the initial partitions once they are merged. On the other hand, in our current design, we have exploited the columnar structure to provide better locality and access patterns by adopting several implementation tactics from original cracking. It will be an interesting and useful study to investigate how the various tradeoffs balance also depending on the workload.

Adaptive Indexing in Auto-tuning Tools. Finally, we anticipate that adaptive indexing ability to make incremental investments could find a place in auto-tuning tools that currently analyze workloads a-priori and then create indexes wholesale. Auto-tuning tools could decide which indexes and materialized views to create immediately, which to prohibit (e.g., in order to avoid all space or update costs), and which to create incrementally using adaptive indexing techniques. In addition, the auto-tuning tool could decide for each incremental index how much of an initial investment to make, based both on the objectives of the current workload and on the anticipated relevance of this index to the expected workload.

D. RADIX-CLUSTERING TOOLS

a) ['a', 'z']:

v	$A(v)$	$f(v)$	radix-bits
'a'	97	0	~~000~~ **0** 0000
'b'	98	1	~~000~~ **0** 0001
⋮	⋮	⋮	⋮
'o'	111	14	~~000~~ **0** 1110
'p'	112	15	~~000~~ **0** 1111
'q'	113	16	~~000~~ **1** 0000
'r'	114	17	~~000~~ **1** 0001
⋮	⋮	⋮	⋮
'y'	121	24	~~000~~ **1** 1000
'z'	122	25	~~000~~ **1** 1001

b) ['d', 'i']:

v	$A(v)$	$f(v)$	radix-bits
'd'	100	0	~~00000~~ **0** 00
⋮	⋮	⋮	⋮
'g'	103	3	~~00000~~ **0** 11
'h'	104	4	~~00000~~ **1** 00
'i'	105	5	~~00000~~ **1** 01

c) ['j', 'm']:

v	$A(v)$	$f(v)$	radix-bits
'j'	106	0	~~000000~~ **0** 0
'k'	107	1	~~000000~~ **0** 1
'l'	108	2	~~000000~~ **1** 0
'm'	109	3	~~000000~~ **1** 1

Figure 15: Encoding and radix-bits as used by algorithm HRR in Fig. 5.