Revisiting the Theory and Practice of Database Cracking

Fatemeh Zardbani  Aarhus University  Peyman Afshani  Aarhus University  Panagiotis Karras  Aarhus University

ABSTRACT

Database cracking (DBC) provides an adaptive data storage environment that meets the needs of modern applications in business and science, reorganizing data on demand and adapting indexes on the fly, automatically, and collaboratively to query processing. Despite intensive research on cracking and other adaptive indexing variants, their theoretical side has scarcely been investigated. Yet, quite surprisingly, as we show, an antecedent of database cracking in a pure, no-frills form had been developed in the theory community 24 years ahead of its time by the name of deferred data structuring (DDS). While lacking system implementations, DDS corresponds to what we would call, by the terminology used in the database community, materialization-based data-driven center cracking for point lookup queries, as well as a stochastic variant thereof. Further, DDS has gone beyond regular cracking proposals by suggesting a policy that reorganizes index ranges along the median of a sample set, i.e., a mediocre element.

In this paper, we reanalyze state-of-the-art database cracking algorithms with the benefit of hindsight provided by deferred data structuring, and propose new alternatives that use a mediocre element as cracking pivot instead of a random or a median one. In a thorough experimental study, we determine that a logarithmic or linear sample size yields best performance on a standard benchmark across the board of cracking algorithms.

1 INTRODUCTION

Database Cracking (DBC) [1–3] addresses the needs of dynamic environments where workload knowledge and idle time are scarce, queries follow an exploratory path, and new data arrive continuously [1]: as a form of adaptive indexing [6], it paves the way to self-organizing database management systems, eschewing the need for human administration in physical database design. Cracking builds and refines index data structures for a column-oriented database incrementally, in response to queries and arriving data, without a need for human intervention; its core operation, applied within the select operator, reorganizes a column into pieces [5], handles updates [4] and invites security features [8]. A stochastic alternative [1] improves performance by refraining from blindly following queries; it also creates random cracks on its own, and thereby avoids the deterioration of performance that skewed workloads may cause.

Surprisingly, while database cracking has been studied over the last decade, an antecedent thereof had been investigated from a theory perspective two decades in advance by the name of Deferred Data Structuring (DDS) [7]. Specifically, DDS suggested that, instead of processing a data set in advance, we may instead process it while responding to queries. Traditionally, to answer the query "is integer x in list ℓ?", we would scan ℓ in O(n). If the number of queries is high, it pays off to sort O(n log n) in a pre-processing step and then perform binary search in O(log n) for each query. By DDS, we create a data structure that represents the list while processing queries, achieving better performance in all cases. While DDS was proposed for lookup queries on a static data set only, we will argue that its main logic uncannily resembles that of database cracking.

In this paper, we observe the resemblance between DBC and DDS and bring both concepts under the same roof. We conduct a thorough theoretical study of state-of-the-art DBC algorithms under the light of DDS methods. We implement existing DDS and DBC proposals and propose new intermediary, mediocre-based solutions, that inherit both the theoretical elegance of DDS and the practical applicability of DBC.

2 RELATED WORK

We discuss related works in two fields: database cracking and deferred data structuring.

2.1 Database Cracking

Database Cracking [2, 3] reorganizes and indexes columns in an adaptive manner triggered by user queries, within the SELECT operator. In the general case, a query requests all values within a range, [low, high]; when responding to that query, the a cracking system finds one or more pieces of the current index where the requested data resides, reorganizes (i.e., cracks) the column so as to bring the result values between query bounds low and high in a contiguous space, and updates the index accordingly. Figure 1a provides an example.

![Database cracking illustration [1]](image1)

(a) Reorganizing a column.  (b) Cracking algorithms at work.

Figure 1: Database cracking illustration [1].

Stochastic Cracking [1] maintains performance when faced with pathological workloads; in addition to cracking using query bounds as pivots, it creates additional cracks while traversing the index towards query bounds. Figure 1b indicates how several stochastic cracking algorithms work. We discuss the six main alternatives: DDC, DDR, DD1C, DD1R, MDD1R, and PMDD1R.

The Data-Driven Center (DDC) algorithm divides (i.e., cracks) each value range it encounters while traversing the index along the middle (i.e., median) of its value domain recursively, as in an ideal case of pivot selection by quicksort. This recursive splitting process terminates when pieces become smaller than a size threshold. Thereafter, DDC cracks on query bounds as usual. Algorithm 1 illustrates the process. On the other hand, the Data-Driven Random (DDR) algorithm avoids the median-finding overhead: it uses a random element instead of the median as pivot when cracking each value range in Line 6 of Algorithm 1.

Both DDC and DDR incur an overhead on the first few queries, as they recursively introduce many cracks on the way to query bounds. Two lightweight alternatives, DD1C and DD1R, eschew the recursion, i.e., crack only once at a median or random pivot,
respectively, in addition to cracking at query bounds, by turning the while loop in Line 5 of Algorithm 1 to an if statement.

**ALGORITHM 1: DDC [1]**

Result: Cracks at center of each relevant piece and bounds

1 int DDCrack(C[0-9], v)
2 Find the piece, Piece, that contains value, v;
3 pLow = Piece.firstPosition:
4 pMiddle = Piece.lastPosition:
5 while (pMiddle >= pLow > CRACK-THRESHOLD)
6 pMiddle = (pHigh + pLow)/2:
7 Find crack at pMiddle:
8 if (v <= pMiddle)
9 pLow = pMiddle:
10 else
11 pHigh = pMiddle:
12 position = crack(pLow, pHi, v);
13 return position;
14 /* Main Body: DDC */
15 */ Crack array C on bounds a and b */
16 positionLow = DDCrack(a, b);
17 positionHi = DDCrack(b, c);
18 result = createView(C, positionLow, positionHi);

Still, the hitherto presented algorithms create cracks at each query’s bounds, which may hurt performance without bringing a benefit in the long run. **MDDR**, a variation of **DDR**, dispels the cracking at query bounds as well, and simply materializes query results while creating exactly one random crack per query. Algorithm 2 shows the corresponding pseudocode.

**ALGORITHM 2: MDDR [1]**

Result: Cracks at a random point in one relevant piece and bounds

1 array, split-and-materialize(Piece, a, b)
2 L = Piece.firstPosition:
3 R = Piece.lastPosition:
4 result = new array:
5 X = C[L] rand()[(R - L) + 1];
6 while (L <= R)
7 while (L <= Q and (C[L] < X)
8 if (a <= C[L] and (C[L] < b)
9 result = Add(C[L])
10 L = L + 1;
11 while (L <= R and (C[R] > X)
12 if (a <= C[R] and (C[R] < b)
13 result = Add(C[R])
14 R = R - 1;
15 if (L <= R)
16 swap(C[L], C[R])
17 Add crack on X at position L;
18 return result;
19 /* Main Body: MDDR */
20 */ Crack array C on bound a, b */
21 Find the piece, P1, that contains value, a;
22 Find the piece, P2, that contains value, b;
23 if (P1 == P2)
24 result = split-and-materialize(P1, a, b);
25 else
26 res1 = split-and-materialize(P1, a, b);
27 res2 = split-and-materialize(P2, a, b);
28 return = concat(res1, view1, res2);

In more detail, the algorithm first finds the pieces where the two bounds are. If they are in the same piece, it partitions that piece with respect to a random pivot, while collecting the query results in an array. Should they be in different pieces, it partitions the pieces where each bound belongs and then concatenates the query results, as well as all pieces in between to produce the response to the query. Note that **MDDR** maintains a data-driven character: even though not cracking at query bounds, it introduces random cracks in the pieces where those bounds are.

Even with **MDDR**, the initial queries of a workload need to reorganize almost all the data. **PMDDR** is a progressive instantiation of **MDDR** that takes the incremental nature of cracking one step further. **MDDR** performs a reorganization task on a given piece in smaller units, performed with each query touching that piece. A percentage \( p \) determines how much of the pending reorganisation task is done with each relevant query, while materializing and returning the query result.

### 2.2 Deferred Data Structuring

**Deferred Data Structures [7]** are tree-like structures built in response to queries. Their objective and rationale resembles database cracking, even though they were introduced two decades earlier with a focused theoretical intent and no accompanying system implementation. Consider a list \( f = \{ x_{1}, x_{2}, x_{3}, \ldots, x_{n} \} \) and existence queries therefore. \( q = \{ q_{1}, q_{2}, q_{3}, \ldots, q_{r} \} \). A conventional approach would sort the list and answer each query by binary search. **DDS** performs sorting through query answering; it answers each query in \( O(n) \) time, and partition the list as well while doing so. Algorithm 3 illustrates **DDS**. By the terms of Section 2.1, **Algorithm 3** corresponds to a **DDC** variant of database cracking specialized on point lookup queries, without a size threshold: it cracks recursively on the median, like **DDC** does, and reports the existence or absence of the lookup query value; were there a size threshold, it would correspond to an **MDDC** variant, yet without such a threshold **MDDC** degenerates to **DDC**.

**ALGORITHM 3: DDS via recursive median finding [7]**

Result: Create a tree structure representation of list \( f \) while responding to queries

1 boolean SEARCH(v:node, q:query)
2 if (v is not labeled)
3 EXPAND(v);
4 if (label(v) == q)
5 return true;
6 if (v is a leaf node)
7 return false;
8 if (q < label(v)
9 return SEARCH(left child(v), q);
10 if (q > label(v)
11 return SEARCH(right child(v), q);
12 void EXPAND(v:node)
13 S = set(s);
14 m = MEDIUM = FIND(S);
15 label(v) = m;
16 if (S == 1)
17 return;
18 S_L = [x | x is in S and x < m]
19 S_R = [x | x is in S and x > m]
20 set(left child(v)) = S_L
21 set(right child(v)) = S_R;
22 /* Main Body */
23 initialize the tree \( T_x \) with the n data keys at the root;
24 Get a query q;
25 Results = SEARCH(root, q);
26 Output the result;
27 goto Line 21;

Further, DDS [7] comes along with a randomized proposal, which replaces the exact median-finding operation with a mediocre function, i.e., the median of small sampled set of values, whose computed rank passes a sanity test, instead of the whole set, as Algorithm 4 shows; this choice improves the cost per query while it still creates a well-balanced tree structure in the long term. Once again, the rationale is reminiscent of what we would call **MDDR** in database cracking terms. In particular, a randomized **DDS** where the size of the sampled set is one element corresponds to an MDDR cracking algorithm specialized on point lookup queries and without a size threshold.

**ALGORITHM 4: Mediocre finding function [7]**

Result: Finds mediocre of set \( T \)

1 int mediocreoflist(T(list of values)
2 \( t = \text{size}(T) \)
3 Pick a random of sample S of size \( 2 \times \lceil t/2 \rceil + 1 \) from T;
4 \( m = \text{MIDIAN} = \text{FIND}(S) \);
5 Compute rank(m) by comparing with each element of \( T = S \);
6 if (rank(m) is not in the range \( (t/2) \times \lceil t/2 \rceil \))
7 then return m;
3 THEORETICAL ANALYSIS

Here, we analyze state-of-the-art database cracking algorithms by the tools of deferred data structuring, assuming a cracking size threshold of zero and point lookup queries. We also propose, study, and build upon an alternative stochastic cracking algorithm, \textsc{DDM}, which uses a mediocre element as cracking pivot, as in Algorithm 4, instead of a random one, as \textsc{DDR} algorithms do, or a median one, as \textsc{DCC} algorithms do.

**Theorem 3.1.** \textsc{DCC}: The number of operations needed to process \( r \) queries on a list of \( n \) points is no more than \( \lambda(n, r) \):

\[
\lambda(n, r) = \begin{cases} 
3\log r + r \log n, & \text{if } r \leq n \\
(3n + r) \log n, & \text{if } r > n
\end{cases}
\]

**Proof.** In case \( r \leq n \), at any level of the tree, at most \( n \) nodes are expanded. For the top \( \log r \) levels, the total cost is less than \( 3\log r \), since all nodes have to be expanded. The creation of a crack includes finding the median, which requires \( 3\text{set(node)} \). The cost of node expansion at level \( i \) of the tree for \( i > \log r \), is \( O(rn/2^i) \), since the expansion of a node at this level costs at most \( 3n/2^i \). Summing over all but the first \( \log r \) levels, we get an \( O(n) \) cost, which is dominated by \( 3n \log r \). Searching for each query costs \( O(\log n) \), since the tree is balanced, hence \( r \log n \) term. When \( r > n \), expansion will complete the tree, with a cost of \( 3n \log n \), while search follows the same principles. \( \square \)

**Theorem 3.2.** \textsc{DDR}: The number of operations needed to process \( r \) queries on a list of \( n \) points is no more than \( \lambda(n, r) \):

\[
\lambda(n, r) = n^2 + rn
\]

and is expected to be no more than \( \lambda(n, r) \) operations:

\[
\lambda(n, r) = \begin{cases} 
1.39n \log r + r \log n, & \text{if } r \leq n \\
(1.39n + r) \log n, & \text{if } r > n
\end{cases}
\]

**Proof.** As there is no guarantee that the created tree will be balanced, in the worst case, the random numbers chosen to create cracks yield a completely unbalanced tree. A query may cause the entire tree to be created in \( O(n^2) \). Due to the lack of balance, search can take up to \( n \) operations, producing the \( rn \) term. In the average case, we expect performance similar to quicksort [9], with 1.39 in place of 3 in Theorem 3.1. \( \square \)

**Theorem 3.3.** \textsc{DDM}: The number of operations needed for processing \( r \) queries in a list of \( n \) points is no more than \( \lambda(n, r) \):

\[
\lambda(n, r) = \begin{cases} 
(1 + \alpha)(n \log r + r \log n), & \text{if } r \leq n \\
(1 + \alpha)(n + r) \log n, & \text{if } r > n
\end{cases}
\]

with probability greater than \( 1 - \frac{\log r}{\beta n} \), where \( \alpha \ll 1 \) and \( \beta \) depends on the value of \( \alpha \).

**Proof sketch.** The proof follows from [7]. The height of the tree created only differs from \( \log n \) by a constant, so the search operations are \( r \log n \). Then, the probability of the first sample chosen rendering a median that passes the test is higher than \( 1 - \frac{1}{\text{set(node)}} \), which leads to the conclusion that the total cost of testing for mediocrity is at most \( (1 + \alpha)n \log r \) with probability higher than \( 1 - \frac{\log r}{\beta n} \) and \( \beta \) depends on the value of \( \alpha \). The total cost of finding the medians for the first \( \log r \) levels is \( O(n^2 r^3) \) with probability higher than \( 1 - \frac{\log r}{\beta n} \), from which the complexity for \( r \leq n \) follows. If \( r > n \) the tree will be complete with some extra costs, as in the \textsc{DDR} case. \( \square \)

**Theorem 3.4.** \textsc{DD1C}: The number of operations needed to process \( r \) queries in a list of \( n \) points is no more than \( \lambda(n, r) \):

\[
\lambda(n, r) = \begin{cases} 
14n + r \log n, & \text{if } r \leq n \\
(3n + r) \log n, & \text{if } r > n
\end{cases}
\]

**Proof.** As the first query, \( q_1 \), is made to a list of length \( n \), the median should be found in \( 3n \). Then, one of the two crack pieces is chosen and another crack is made, with regards to the query. The process of partitioning takes at most \( \frac{3}{4} \) comparisons. For the second query, up to two comparisons are made based on the crack created in the first query, to choose a chunk of size at most \( n/2 \), find its median, and crack one of the resulting pieces, yielding \( \frac{3n}{2} + \frac{3}{4} \). Following the same pattern until the \( r \)-th query, we get a cost of:

\[
\sum_{i=1}^{\log r} \frac{3n}{2i-1} + \sum_{i=1}^{\log r} \frac{n}{2i} = 7n \sum_{i=1}^{\log r} \frac{1}{2i} < 14n
\]

while the search component costs \( r \log n \). When the number of queries reaches \( n \), the tree will be complete, and the cost is as in the proof of Theorem 3.1. \( \square \)

**Theorem 3.5.** \textsc{DDIR}: The number of operations needed to process \( r \) queries in a list of \( n \) points is no more than \( \lambda(n, r) \):

\[
\lambda(n, r) = \begin{cases} 
3rn + r, & \text{if } r \leq n \\
n^2 + rn, & \text{if } r > n
\end{cases}
\]

and is expected to be no more than \( \lambda(n, r) \) operations:

\[
\lambda(n, r) = \begin{cases} 
7.56n + r \log n, & \text{if } r \leq n \\
(1.39n + r) \log n, & \text{if } r > n
\end{cases}
\]

**Proof.** In the worst case in terms of random pivot choices, for query \( q_1 \), we pick a random point and partition based on it, in at most \( n \) operations. The pieces may be of size 1 and \( n - 1 \). One of those two is partitioned based on the query in \( n - 1 \) comparisons. The second query will partition a piece of size as large as \( n - 2 \) for the random crack and one as large as \( n - 3 \) for the query bound, and so on, in \( n - 2(i - 1) + \lceil n - (2i - 1) \rceil \) operations for the \( i \)-th query. Going all the way to \( q_r \), we have

\[
\sum_{i=2r-1}^{n} (n - i) = 2rn - 2r^2 + r
\]

operations for expansion and \( rn \) operations for search. Should the number of queries exceed \( n \), the tree is completed, hence \( n^2 \) operations. In the expected case, the tree resembles a balanced tree and results follow Theorem 3.4, with the quicksort complexity factor 1.39\( n \) replacing 3\( n \) in calculations, as in Theorem 3.2. \( \square \)

**Theorem 3.6.** \textsc{DD1M}: The number of operations needed to process \( r \) queries in a list of \( n \) points is no more than \( \lambda(n, r) \):

\[
\lambda(n, r) = \begin{cases} 
(1 + \alpha)n \log r + r \log n, & \text{if } r \leq n \\
(1 + \alpha)(n + r) \log n, & \text{if } r > n
\end{cases}
\]

with probability greater than \( 1 - \frac{\log r}{\beta n} \).

**Proof.** By Lemma 3 in [7], with high probability we only need to compute a median over a sample once, as the first attempt passes the mediocrity test. For the first \( r \) queries, we compute the medians of nodes that will overall lead to \( O(n^2 r^3) \) (Lemma 5 in [7]). The cost testing for mediocrity at level \( i \), denoted by \( c_i \), is proven to be less than \((1 + \alpha)n \log r \). After cracking at the
We apply each algorithm on the same 160K-query workload and use the 4TB SkyServer with p □

The cost of search remains similar to that of DDM.

Overall, materialization-based algorithms have similar complexities as their default counterparts due to their common bases.

4 EXPERIMENTAL STUDY

We assess the performance of mediocre-based variants database cracking algorithms [1] inspired from our study of deferred data structures [7]. We conduct experiments on a Ubuntu Linux server release 18.04 machine with a 10-core 3.1GHz Intel E5-2687W processor and 377GB of RAM. All methods are implemented in C++ upon the code1 of [1]; our code is also available2 online. We use the 4TB SkyServer3 data and workload [1], derived from an astronomy project mapping the universe. We filter selection predicates from 160K chronologically ordered queries using the right ascension attribute of the Photoobjall table, which contains 500 million tuples. Query patterns are complex, as users tend to focus in a specific area of the sky before moving on.

4.1 Compared Algorithms

We compare the state-of-the-art stochastic cracking algorithms presented in Section 2, namely DDR, DD1R, MDD1R, and PMDD1R with p = 0.1 and CRACK_THRESHOLD = 128 to their median-based counterparts and to counterparts that use a mediocre cracking pivot, i.e., the median of a random sample set of a cracked piece, rather than a median or random one: DDM, DD1M, MDD1M, and PMDD1M, respectively.

The median-based counterparts of DDR, DD1R and MDD1R are DDC, DD1C and MDD1C respectively; we included those three in our study, but they proved to be too expensive. We do not include a median-based counterpart of PMDD1R, since median-finding operations are hard to render progressive. The mediocre-based policy is reduced to the median-based one for m equal to piece size, and to the randomized one for m = 1.

4.2 Results

We apply each algorithm on the same 160K-query workload and measure the total runtime, juxtaposing mediocre-based variants to their randomized and median-based counterparts, where such exist. Figure 2 shows the results when varying the sample set size m from 1 to 41, and as a linear, logarithmic, or exponential function of piece size, with the mediocre-based policy, averaging over 110 runs and foregoing the mediocrity check [7]. The case of single crack without materialization is very similar to that of single crack with materialization in Figure 2b, hence we omit a separate figure for that case.

In the case of a constant sample size, the cost of calculating the median of a small sample set is initially a worthwhile price to pay for the benefits it brings, yet the cost to benefit ratio deteriorates as m grows; binomial fit curves visualize the trends in Figure 2a, 2b, and 2c. However, cases where sample size is a simple function of piece size achieve the best performance in all variants. The logarithmic function is the best performer with recursive and simple crack with and without materialization. In the variant with progressive materialization, a linear function, m = size/1000, performs best.

5 CONCLUSION

We revisited the theory and practice of database cracking, which has been intensively studied in practice, yet scantily examined in theory. We provided the first thorough study of the complexity of the all state-of-the-art stochastic cracking algorithms, drawing from an overlooked 32-year-old study that introduced analogous concepts under the name of deferred data structuring. Inspired from deferred data structuring, we introduced a refined stochastic cracking policy that uses a sample-based mediocre pivot, rather than an arbitrary random or median one, for data-driven cracking. We showed that variants of state-of-the-art stochastic cracking algorithms using the mediocre-based policy have lower complexity than their median-based and randomized counterparts with high probability, and demonstrated experimentally that they stand out in terms of cumulative time efficiency.

REFERENCES


1https://github.com/felix-halim/scrack
2https://gitlab.com/fatemeh.zardbani/adaptive-indexing
3http://cas.sdss.org/