Database cracking: towards auto-tuning database kernels

Ydraios, E.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 4

Updates*

4.1 Introduction

Until now, we studied database cracking for the static scenario only, i.e., without updates. A new database architecture should also handle high-volume updates to be considered as a viable alternative. For database cracking, update support is of significant importance. The whole idea of cracking is based on learning from previous queries and continuously physically reorganizing the data. An update, though, changes the physical organization of a column as well. Thus, updates invalidate past cracking knowledge given that this knowledge naturally depends on the physical state. Efficiently maintaining the cracking knowledge is the key to make database cracking viable under updates.

Here, we study updates in detail and we show that the nice performance properties of a cracked database can be maintained in a dynamic environment where random updates interleave with random queries.

4.1.1 Contributions

The contributions of this chapter are the following. We present a series of algorithms to support insertions, deletions and updates in a cracking DBMS. We show that our algorithms manage to maintain the advantage of cracking in terms of fast data access. In addition, our algorithms do not hamper the ability of a

*The material in this chapter has been the basis for the ACM SIGMOD07 paper “Updating a Cracked Database” (Idreos et al., 2007b).
cracking DBMS to self-organize, i.e., the system can adapt to query workload with the same efficiency as before and still with no external administration.

The proposed algorithms follow the “cracking philosophy”, i.e., with lightweight actions we continuously try to match the current workload in order to have immediate benefits. This way, an update becomes relevant only once a query actually needs the new data. Thus, in cracking, incoming updates are simply marked as pending actions and we update the “cracking” data structures only once queries have to see the updated data. The whole architecture follows the design principles we set in the previous chapter; everything happens on-the-fly while processing queries, i.e., updating the cracker columns becomes part of the cracker operators.

The proposed algorithms range from the complete case, where we apply all pending actions in one step, to solutions that update only what is really necessary for the current query; the rest is left for the future when users will become interested in this part of the data.

We implemented and evaluated our algorithms on top of the selection cracking architecture. A detailed experimental evaluation demonstrates that updates can indeed be handled efficiently in a cracking DBMS. A direct comparison with an AVL-tree based scheme highlights the savings obtained with the cracking philosophy. Our study is based on two performance metrics to characterize system behavior. We observe the total time needed for a query and update sequence, and our second metric is the per query response time. The query response time is crucial for predictability, i.e., ideally we would like similar queries to have a similar response time. We show that it is possible to sacrifice little from the performance in terms of total query sequence cost and to keep the per query response time in a predictable range for all queries.

4.1.2 Outline

The rest of the chapter is organized as follows. In Section 4.2, we discuss how we fitted the update process into the cracking architecture by extending the select operator. Section 4.3 presents a series of algorithms to support insertions in a cracked database. Then, in Section 4.4, we present algorithms to handle deletions, while in Section 4.5 we show how updates are processed. In Section 4.6, we present a detailed experimental evaluation. Finally, Section 4.7 concludes the chapter.
4.2 When to Update: Self-organizing Updates

There are two main issues to consider regarding updates: (a) \textit{when} and (b) \textit{how} updates are applied. Here, we discuss the first issue, postponing the latter to Section 4.3.

\textbf{What to Update} Before we begin our analysis we should mention that updating the base columns is not affected by cracking. Cracking physically changes cracker columns which are copies of the respective base columns. Hence, we assume that an update has already been applied to the base column (i.e., appended) before it has to be applied to the respective cracker column and cracker index. In the remainder of this chapter, we focus on updating the cracking data structures only.

\subsection*{4.2.1 Updating On Demand}

If we had enough idle time, we could off-line apply all updates or if we had a priori workload knowledge and enough idle time, we could apply only the most beneficial updates. As we have already discussed in previous chapters though, the cracking vision is to self-organize and adapt to the query workload in dynamic environments where we do not have the luxury of a priori workload knowledge or idle time. And all this should happen without any external human administration leading to a completely autonomous system. Our goal is to maintain these properties also in the presence of updates.

One of the key points of the cracking architecture towards this direction is that physical reorganization happens with every query \( q \) and while processing \( q \). Each query causes only data relevant for its result to be physically reorganized. Thus, in order to maintain the self-organizing behavior in the presence of updates, the architecture proposed here is in line with the cracking philosophy, i.e., always do \textit{just enough}. Make small investments with every query to gradually adapt to the current workload patterns while these patterns are still active, without disturbing query processing and without introducing any delays.

An update can be seen as a request that becomes relevant only the first time that a query actually needs to analyze the updated data. In cracking, updating the database becomes \textit{part of query execution} in the same way as physical reorganization entered the critical path of query processing.

Let us proceed with the details of our architecture. The cracker columns and indices are \textit{not} immediately updated as updates arrive. Instead, updates are kept in two separate columns for each attribute: the \textit{pending insertions}
CHAPTER 4. UPDATES

column and the pending deletions column. When an insert request arrives, the new tuples are simply appended to the relevant pending insertions column. Similarly, the tuples to be deleted are appended in the pending deletions column of the referred attribute. Finally, an update query is simply translated into a deletion and an insertion. Thus, all update operations can be executed very fast, since they result in simple append operations to the pending-update columns.

When a query requests data from an attribute, the relevant cracking data structures are updated only if necessary. The updates are applied as part of the crack operators while processing the query. This is a natural extension of the cracking architecture. Recall from previous chapter that the crack select operator makes sure that all values that qualify the given predicate are in a contiguous space in the cracker column. The update-aware select operator does exactly the same thing. To achieve this, it has to physically plug in any relevant updates to the result area.

4.2.2 Update-aware Select Operator

We extended the crack select operator such as it always begins with a check to find out whether there are pending updates for the involved attribute, e.g., values that qualify to be in the result. In this way, only if there are pending update values within the requested value range, then one of our update algorithms runs. Then, the data structures are in a “safe” state so that the cracking select operator can continue normally without losing any information or presenting us information that has been deleted in the past.

To efficiently search in the pending insertions and deletions columns for relevant values, we first sort these columns and then use binary search. We do that because this order will be used by our update algorithms in the future (see next section). All algorithms maintain the order of the pending insertions and deletions columns, so sorting is not necessary in the future, unless new updates arrive.

The exact steps of the operator are as follows:

1. search the pending insertions column to find qualifying tuples that should be included in the result
2. search the pending deletions column to find qualifying tuples that should be removed from the result
3. if at least one of the previous results is not empty, then run an update algorithm
4.3. INSERTIONS

(4) search the cracker index to find which pieces contain the query boundaries

(5) physically reorganize these pieces (at most 2)

(6) return the result.

Steps 1, 2 and 3 are our extension to support updates, while Steps 4, 5 and 6 are the original cracker select operator steps as described in Chapter 3. When the select operator proceeds with Step 4, any pending insertions that should be part of the result have been placed in the cracker column and removed from the pending insertions column. Likewise, any pending deletions that should not appear in the result have been removed form the cracker column and the pending deletions column. Thus, the pending columns continuously shrink when queries consume updates. They grow again with incoming new updates.

Updates are received by the cracker data structures only upon commit, outside the transaction boundaries. By then, they have also been applied to the attribute columns, which means that the pending cracker column updates (and cracker index) can always be thrown away without loss of information. Thus, in the same way that cracking can be seen as dynamically building an index based on query workload, the update-aware cracking architecture proposed can be seen as dynamically updating the index based on query workload.

4.3 Insertions

Let us proceed our discussion on how to update the cracking data structures. For ease of presentation, we first present algorithms to handle insertions. Deletions are discussed in Section 4.4 and updates in Section 4.5. We discuss the general issues first, e.g., what is our goal, which data structures do we have to update, how etc. Then, a series of cracker update algorithms are presented in detail.

4.3.1 General Discussion

As discussed in Section 3.2, there are two basic structures to consider for updates in a cracking DBMS, (a) the cracker column and (b) the cracker index. A cracker index $I$ maintains information about the various pieces of a cracker column $C$. Thus, if we insert a new tuple in any position of $C$, we have to update the information of $I$ appropriately. We discuss two approaches in detail: one that makes no effort to maintain the index, and a second that always tries to have a valid (cracker-column,cracker-index) pair for a given attribute.
CHAPTER 4. UPDATES

Pending Insertions Column

To comply with the “cracking philosophy”, all algorithms start to update the cracker data structures once a query requests values from the pending insertions column. Hence, looking up the requested value ranges in the pending insertions column must be efficient. To ensure this, we sort the pending insertions column once the first query arrives after a sequence of updates, and then exploit binary search. Our merging algorithms keep the pending insertions column sorted. This approach is efficient as the pending insertions column is usually rather small compared to the complete cracker column, and thus, can be kept and managed in memory. We leave further analysis of alternative techniques — e.g., applying cracking with “instant updates” on the pending insertions column — for future research.

Discarding the Cracker Index

Let us begin with a naive algorithm, i.e., the forget algorithm (FO). The idea is as follows. When a query requests a value range such that one or more tuples are contained in the pending insertions column, then FO will (a) completely delete (forget) the cracker index and (b) simply append all pending insertions to the cracker column. This is a simple and very fast operation. Since the cracker index is now gone, the cracker column is again valid. From there on, the cracker index is rebuilt from scratch as future queries arrive. The query that triggered FO performs the first cracking operation and goes through all the tuples of the cracker column. The effect is that a number of queries suffer a higher cost, compared to the performance before FO ran, since they will physically reorganize large parts of the cracker column again.

Cracker Index Maintenance

With algorithm FO we delete information (our cracker index) that allows us to answer queries very fast. This has the effect that a number of queries must pay a cost so that we can reach the same levels of performance we had before the update.

Ideally, we would like to handle the appropriate insertions for a given query without losing any information from the cracker index. Then, we could continue answering queries fast without having a number of queries after an update with a higher cost. This is desirable not only because of speed, but also to be able to guarantee a certain level of predictability in terms of response time, i.e., we would like the system to have similar performance for similar queries. This
4.3. INSERTIONS

Figure 4.1: An example of a lossless insertion for a query that requests $5 < A < 50$

calls for a *merge-like* strategy that “inserts” any new tuple into the correct position of a cracker column and correctly updates (if necessary) its cracker index accordingly.

A simple example of such a “lossless” insertion is shown in Figure 4.1. The left-hand part of the figure depicts a cracker column, the relevant information kept in its cracker index, and the pending insertions column. For simplicity, a single pending insert with value 17 is considered. Assume now a query that requests $5 < A < 50$, thus the pending insert qualifies and should be part of the result. In the right-hand part of the figure, we see the effect of merging value 17 into the cracker column. The tuple has been placed in the second cracker piece, since, according to the cracker index, this piece holds all tuples with value $v$, where $12 < v \leq 41$. Notice, that the cracker index has changed, too. Information about Pieces 3, 4 and 5 has been updated, increasing the respective starting positions by 1.

Trying to device an algorithm to achieve this behavior, triggers the problem of *moving* tuples in different positions of a cracker column. Obviously, large shifts are too costly and should be avoided. In our example, we moved down by one position all tuples after the insertion point. This is not a viable solution
in large databases. In the rest of this section, we discuss how this merging step can be made very fast by exploiting the cracker index.

4.3.2 Shuffling a Cracker Column

We make the following observation. Inside each piece of a cracker column, tuples have no specific order. This means that a cracker piece \( p \) can be shifted \( z \) positions down in a cracker column as follows. Assume that \( p \) holds \( k \) tuples. If \( k \leq z \), we obviously cannot do better than moving \( p \) completely, i.e., all \( k \) tuples. However, in case \( k > z \), we can take \( z \) tuples from the beginning of \( p \) and move them to the end of \( p \). This way, we avoid moving all \( k \) tuples of \( p \), but move only \( z \) tuples. We will call this technique shuffling.

**Shuffling From The Top of a Column**

For a more complete description, let us first give an example by merging insertions starting from the beginning of a cracker column. Subsequently, we will show that merging from the end of a column is a superior technique.

In the example of Figure 4.1 (without shuffling), 10 tuples are moved down by one position. With shuffling we need to move only 5 tuples. Let us go through the example of Figure 4.1 again, using hopping this time to see why this happens. First, we move \( v_1 \), the first value of Piece 3 in a temporary space \( temp_1 \). The new value 17 can be placed in the position where \( v_1 \) was. Then, we move \( v_2 \), the first value of Piece 4 in a temporary space \( temp_2 \). \( v_1 \) will be moved then from \( temp_1 \) in the position where \( v_2 \) was. Then, we move \( v_3 \), the first value of Piece 5 at the end of the column. In the position where \( v_3 \) was, we move \( v_2 \) from \( temp_2 \). Of course, the cracker index has to be updated so as Pieces 3, 4 and 5 will have start positions greater by one. In this way, we made only 5 moves instead of 10 that we had when moving all values down one position. The benefits of hopping clearly depend on where in the cracker column the new value belongs to and also on how many pieces exist in the cracker column but in general it is faster than simply shifting all values.

A critical point with shuffling is the order in which we merge the updates. As described in the previous paragraph, it requires two temporary spaces to temporarily hold values subject to move. To be safe we have to allocate for both spaces enough space to hold as many values as many the values we are trying to insert. Clearly this increases the memory requirements of the algorithm. In addition, having to continuously move values back and forth from the temporary spaces, may seriously hamper performance depending on where in the cracker
column the new values belong and how many pieces exist. Furthermore, the following problem might occur. The values that we have in the temporary space, and that we should put in the beggining of the next piece \( p \), might be more than the actual values contained in \( p \). This significantly increases the complexity of correctly updating the cracker index on the starting position of the various pieces since a lot of state has to be kept around. An alternative is to simply choose to delete information about \( p \) from the cracker index.

**Shuffling From The Bottom of a Column**

Our solution to all the above is to change the direction of shuffling, i.e., start merging from the bottom of the cracker column and not from the piece where the first insertion belongs to. This idea makes everything simple. No state at all has to be kept and no temporary space is needed since there is already free space when we move a value.

Let us go through the example again, this time shuffling from the end of the cracker column to see why. We start from the last piece, Piece 5. The new tuple with value 17 does not belong there. To make room for the new tuple further up in the cracker column, the first tuple of Piece 5, \( t_1 \), is moved to the end of the column, freeing its original position \( p_1 \) to be used by another tuple. We continue with Piece 4. The new tuple does not belong here, either, so the first tuple of Piece 4 (position \( p_2 \)), is moved to position \( p_1 \). Position \( p_2 \) has become free, and we proceed with Piece 3. Again the new tuple does not belong here, and we move the first tuple of Piece 3 (position \( p_3 \)) to position \( p_2 \). Moving to Piece 2, we see that value 17 belongs there, so the new tuple is placed in position \( p_3 \) at the end of Piece 2. Finally, the information in the cracker index is updated so that Pieces 3, 4 and 5 have their starting positions increased by one. Thus, only 3 moves were made this time. This advantage becomes even bigger when inserting multiple tuples in one go.

Algorithm 3 contains the details to merge a sorted portion of a pending insertions column into a cracker column. Notice, that the pending insertions column must be sorted for the algorithm to work correctly since insertions are handled one by one starting from the one with the biggest value. In general, the procedure starts from the last piece of the cracker column and moves its way up. In each piece \( p \), the first step is to place at the end of \( p \) any pending insertions that belong there. Then, \textit{remaining} tuples are moved from the beginning of \( p \) to the end of \( p \). The variable \textit{remaining} is initially equal to the number of insertions to be merged and is decreased for each insertion put in place. The process continues as long as there are pending insertion to merge. If the
Algorithm 3 Merge(C,I,posL,posH)
Merge the cracker column C with the pending insertions column I. Use the tuples of I between positions posL and posH in I.

1: remaining = posH - posL + 1
2: ins = point at position posH of I
3: next = point at the last position of C
4: prevPos = the position of the last value in C

5: while remaining > 0 do
6:   node = getPieceThatThisBelongs(value(next))
7:   if node == first piece then
8:     break
9:   write = point one position after next
10:  cur = point remaining – 1 positions after write in C

11:  while remaining > 0 and
12:       (value(ins) > node.value or
13:        (value(ins) == node.value and node.incl == true)) do
14:     move ins at the position of cur
15:     cur = point at previous position
16:     ins = point at previous position
17:     remaining −−

18:  if remaining == 0 then
19:    break
20:  next = point at position node.position in C
21:  tuples = prevPos - node.position
22:  cur = point one position after next

23:  if tuples > remaining then
24:    w = point at the position of write
25:    copy = remaining
26:  else
27:    w = point remaining – tuples positions after write
28:    copy = tuples

29:  for i = 0; i < copy; i + + do
30:    move cur at the position of w
31:    cur = point at previous position
32:    w = point at previous position

33:  if node == first piece and remaining > 0 then
34:  write = point one position after next
35:  for i = 0; i < remaining; i + + do
36:    move cur at the position of w
37:    cur = point at next position
38:    w = point at next position
first piece is reached and there are still pending insertions to merge, then all remaining tuples are placed at the end of the first piece. This procedure is the basis for all our merge-like insertion algorithms.

### 4.3.3 Merge-like Algorithms

Based on the above shuffling technique, we design three merge-like algorithms that differ on the amount of pending insertions they merge per query, ranging from the one extreme of merging all pending insertions in one step, to merging only what is relevant for the current query. All algorithms are based on the same approach regarding when they run: an update algorithm will be triggered only if a select operator is called by a query on the given attribute that requests a value range such that at least one pending insertion tuple should be part of the result. If no query arrives that requests a value contained in the pending insertions, then the insertions will never be merged. The algorithms also differ in the way they make room for the pending insertions in the cracker column. We continue our discussion by describing each individual merge-like strategy in detail.

**MCI**

Our first algorithm is called the *merge completely insertions* (MCI). algorithm. According to MCI, once a query requests any value from the pending insertions column, the pending insertions column is merged completely, i.e., all pending insertions are placed in the cracker column. The disadvantage is that MCI “punishes” a single query with the task to merge all currently pending insertions, i.e., the first query that needs to touch the pending insertions after the new tuples arrived. On the other hand, we are going through the merging process only once so we can save, as we will see in the experiments section, in terms of total execution time.

To run MCI, Algorithm 3 is called for the full size of the pending insertions column. Thus, $posL = 0$ and $posH$ is the size of the pending insertions column. In Figure 4.2(b) we see an example of the result that MCI will have over an attribute that has an initial condition shown in Figure 4.2(a). The query that triggered MCI requests everything between 15 and 50. Since there are such values in the pending insertions, MCI runs. All pending insertions are properly merged, the cracker index has been updated while the pending insertions column is left empty.
Figure 4.2: An example of how MC, MG and MR will work for a select that requests \( v \) such as \( 15 < v < 50 \)
4.3. INSERTIONS

MGI

MCI is approaching the cracking philosophy in the sense that it will run only if the result of the current query should contain values that exist in the pending insertions column. Our second algorithm, the merge gradually insertions, (MGI), algorithm, goes one step further. In MGI, if a query needs to touch $k$ tuples from the pending insertions column, where $k \geq 1$, then it will merge only these $k$ tuples into the cracker column, and not all pending insertions. The remaining pending insertions wait for future queries to consume them. Thus, MGI does not burden a single query to merge all pending insertions.

For MGI, Algorithm 3 runs for only a portion of the pending insertions column that qualifies as query result. When this procedure is finished, MGI needs to shrink the pending insertions column so that it is dense again and can be reused in the future. The exact positions are easy/cheap to find with binary search since the pending insertions column is sorted. This is done by shifting up the values that did not qualify for the given select and are after the consumed insertions. In Figure 4.2(c) we show an example of what the result of MGI will be. We see that only the necessary values for the current query are merged, values 18 and 45 so the cracker column grew only two positions. In addition, MGI has shrunk the pending insertions column by two position since 18 and 45 need no longer to be there. We will show in the experiments section that although MGI outperforms MCI in terms of cost per query, at the same time this comes with a high total cost, i.e., the total time of processing a sequence of queries and insertions becomes very expensive.

MRI

Our third algorithm is called the merge ripple insertions (MRI) algorithm. The basic idea behind MRI is triggered by the following observation about MCI and MGI. In general, there is a number of pieces in the cracker column that we shift down by shuffling until we start merging. These are all the pieces from the end of the column until the piece $p_h$ where the tuple with the highest qualifying value belongs to. These pieces are irrelevant for the current query since they are outside the desired value range. All we want, regarding the current query, is to make enough room for the insertions we must merge. This is exactly why we shift these pieces down.

To merge $k$ values MRI starts directly at the position that is after the last tuple of piece $p_h$. From there, $k$ tuples are moved into a temporary space $temp$. Then, the procedure of Algorithm 3 runs for the qualifying portion of
the pending insertions as in MGI. The only difference is that now the procedure starts merging from piece $p_h$ and not from the last piece of the cracker column. Finally, the tuples in $temp$ are merged into the pending insertions column. Merging these tuples back in the cracker column is left for future queries. Note, that for a query $q$, all tuples in $temp$ have values greater than the pending insertions that had to be merged in the cracker column because of $q$ (since these tuples are taken from after piece $p_h$). This way, the pending insertions column is continuously filled with tuples with increasing values up to a point where we can simply append these tuples at the cracker column without affecting the cracker index (i.e., tuples that belong to the last piece of the cracker column).

Let us go through the example of Figure 4.1 again, using MRI this time. Piece 3 contains the tuple with the highest qualifying value. We have to merge tuple $t$ with value 17. The tuple with value 60 is moved from position 12 in the cracker column to a temporary space. Then the procedure of Algorithm 3 starts from Piece 3. $t$ does not belong in Piece 3 so the tuple with value 56 is moved from position 10 (the first position of Piece 3) to position 12. Then, we continue with Piece 2. $t$ belongs there so it is simply placed in position 10. The cracker index is also updated so that Pieces 3 and 4 have their starting positions increased by one. Finally, the tuple with value 60 is moved from the temporary space to the pending insertions. At this point MRI finishes without having shifted Pieces 4 and 5 as MCI and MGI would have done.

In Section 4.6, a detailed analysis is provided that clearly shows the advantage of MRI by avoiding the unnecessary shifting of non-interesting pieces. Of course, the performance of all algorithms highly depends on the scenario, e.g., how often updates arrive, how many of them and how often queries ask for the values used in the new tuples. We examine various scenarios and show that all merge-like algorithms always outperform the non-cracking and AVL-case.

4.4 Deletions

Deletion operations form the counter-part of insertions and they are handled in the same way, i.e., when a new delete query arrives to delete a tuple $d$ from an attribute $A$, it is simply appended to the pending deletions column of $A$. Only once a query requests tuples of $A$ that are listed in its pending deletions column, $d$ might be removed from the cracker column of $A$ (depending on the delete algorithm used). Our deletion algorithms follow the same strategies as with insertions; for a query $q$, (a) the merge completely deletions (MCD) removes all deletions from the cracker column of $A$, (b) the merge gradually deletions
(MGD) removes only the deletions that are relevant for \(q\) and (c) the *merge ripple deletions* (MRD), similar to MRI, touches only the relevant parts of the cracker column for \(q\) and removes only the pending deletions interfering with \(q\).

Let us now discuss *how* pending deletes are removed from a cracker column \(C\). Assume for simplicity a single tuple \(d\) that is to be removed from \(C\). The cracker index is again used to find the piece \(p\) of \(C\) that contains \(d\). For insertions, we had to make enough space so that the new tuple can be placed in any position in \(p\). For deletions we have to *spot* the position of \(d\) in \(p\) and clear it. When deleting a single tuple, we simply scan the (usually quite small) piece to locate the tuple. In case we need to locate multiple tuples in one piece, we apply a join between the piece and the respective pending deletes, relying on the underlying DBMS’s ability to evaluate the join efficiently e.g., by building a (temporary) hash table on the fly.

We do that by performing a join operation as follows. We create a non materialized view (slice) \(Cs\) of \(p\). We do the same for the portion of \(D\) where deletions for \(p\) lay. Then we have \(Ds\). Both slices are binary tables of type (key-value). In \(Cs\) we replace all values in the tail with increasing numbers starting from 0 (that denote the positions of tuples in \(p\)). Then, we reverse \(Cs\) so that the tail becomes the head and the head the tail. Then we get the positions of deletes in \(p\) by a relational join between \(Cs\) and \(Ds\). The algorithm used for the join is handled by the MonetDB kernel depending on the sizes of the two views.

Once the position of \(d\) is known, it can be seen as a “hole” which we must fill to adhere to the data structure constraints of the underlying DBMS kernel. We simply take a tuple from the end of \(p\) and move it to the position of \(d\), i.e., we use shuffling to *shrink* \(p\). This leads to a hole at the end of \(p\). Consequently, all subsequent pieces of the cracker column need to be *shifted up* using shuffling. Thus, for deletions the merging process starts from the piece where the lowest pending delete belongs to and *moves down* the cracker column. This is the opposite of what happens for insertions, where the procedure *moves up* the cracker column. Conceptually, removing deletions can also be seen as moving holes down until all holes are at the end of the cracker column (or at the end of the interesting area for the current query in the case of MRD), where they can simply be ignored.

In MRD, the procedure stops when it reaches a piece where all tuples are outside the desired range for the current query. Thus, *holes will be left* inside the cracker column *waiting for future queries* to move them further down, *if needed*. In Algorithm 4, we formally describe MRD. Variable *deletions* is initially equal to the number of deletes to be removed and is increased if holes are found inside the result area, left there by a previous MRD run. The algorithm for MCD and
Algorithm 4 RippleDeletions(C,D,posL,posH, low, incL, hgh, incH)
Update C with the tuples between positions posL and posH of D.

1: remaining = posH - posL + 1
2: del = point at first position of D
3: Lnode = getPieceThatThisBelongs(low, incL)
4: stopNode = getPieceThatThisBelongs(hgh, incH)
5: LposDe = 0
6: while true do
7:   Hnode = getNextPiece(Lnode)
8:   delInCurPiece = 0
9:   while remaining > 0 and (value(del) > Lnode.value or
10:      (value(del) == Lnode.value and Lnode.incl == true)) and (value(del) > Hnode.value or
11:      (value(del) == Hnode.value and Hnode.incl == true)) do
12:      del = point at next position
13:      HposCr = Lnode.pos + (deletions – remaining)
14:      HposCr = Hnode.pos
15:      holesInCurPiece = Hnode.holes
16:      if delInCurPiece > 0 then
17:         HposDe = LposDe + delInCurPiece
18:         positions = getPos(b, LposCr, HposCr, u, LposDe, HposDe)
19:         pos = point at first position in positions
20:         posL = point at last position in positions
21:         crk = point at position HposCr in C
22:         while pos <= posL do
23:            if position(posL)! = position(crk) then
24:               copy crk into pos
25:            else
26:               posL = point at previous position
27:               crk = point at previous position
28:               holeSize = deletions – remaining
29:               tuplesInCurPiece = HposCr – LposCr – delInCurPiece
30:               if holeSize > 0 and tuplesInCurPiece > 0 then
31:                  copy tuplesInCurPiece tuples from position (LposCr + 1)
32:               else
33:                  copy holeSize tuples from position
34:                  (LposCr + 1 + (tuplesInCurPiece – holeSize))
35:                  at position (LposCr – (holeSize – 1))
36:               if tuplesInCurPiece == 0 then Lnode.deleted = true
37:               remaining– = delInCurPiece
38:               deletions+ = holesInCurPiece
39:         if Hnode == stopNode then break
40:         LposDe = HposDe
41:         Hnode.holes = 0
42:         Lnode = Hnode
43:         Hnode.pos– = holeSize + delInCurPiece + holesInCurPiece
44:      if hghNode == last piece then C.size– = (deletions – remaining)
45:      else Hnode.holes = deletions – remaining
46:   remaining = remaining - delInCurPiece
47:   deletions+ = holesInCurPiece
4.5 Updates

A simple way to handle updates is to translate them into deletions and insertions, where the deletions need to be applied before the respective insertions in order to guarantee correct semantics.

However, since our algorithms apply pending deletions and insertions (i.e., merge them into the cracker column) purely based on their attribute values, the correct order of deletions and insertions of the same tuples is not guaranteed by simply considering pending deletions before pending insertions in the update-aware cracker select operator. In fact, problems do not only occur with updates, but also with a mixture of insertions and deletions. Consider the following three cases.

(1) A recently inserted tuple is deleted before the insertion is applied to the cracker column, or after the inserted tuple has been re-added to the pending insertions column by MRI. In either case, the same tuple (identical key and value) will appear in both the pending insertions and the pending deletions column. Once a query requests (the attribute value of) that tuple, it needs to be merged into the cracker column. Applying the pending delete first will not change the cracker column, since the tuple is not yet present there. Then, applying the pending insert, will add the tuple to the cracker column, resulting in an incorrect state. We can simply avoid the problem by ensuring that a to-be-deleted tuple is not appended to the
pending deletions column, if the same tuple is also present in the pending insertions column. Instead, the tuple must then be removed from the pending insertions column. Thus, the deletion effectively (and correctly) cancels the not yet applied insertion.

(2) The same situation occurs if a recently inserted (or updated) tuple gets updated (again) before the insertion (or original update) has been applied. Again, having deletions cancel pending insertions of the same tuple with the same value solved the problem.

(3) A similar situation occurs, when MRI re-adds “zombie” tuples, a pending deletion which has not yet been applied, to the pending insertions column. Here, the removal of the to-be-deleted tuple from the cracker column implicitly applies the pending deletion. Hence, the respective tuple must not be re-added to the pending insertions column, but rather removed from the pending deletions column.

In summary, we can guarantee correct handling of interleaved insertions and deletions as well as updates (translated into deletions and insertions), by ensuring that a tuple is added to the pending insertions (or deletions) only if the same tuples (identical key and value) does not yet exist in the pending deletions (or insertions) column. In case it does already exist there, it needs to be removed from there.

This scheme is enough to efficiently support updates in a cracked database without any loss of the desired cracking properties and speed. Our future work plans include research on unified algorithms that combine the actions of merging pending insertions and removing pending deletions in one step for a given cracker column and query. Such algorithms could potentially lead to even better performance.

4.6 Experimental Analysis

In this section, we demonstrate that our algorithms allow a cracking DBMS to maintain its advantages under updates. This means that queries can be answered faster as time progress and we maintain the property of self-adjustment to query workload. The algorithms are integrated in the MonetDB code base.

All experiments are based on a single column table with $10^7$ tuples (unique integers in [1, $10^7$]) and a series of $10^4$ range queries. The range always spans $10^4$ values around a randomly selected center (other selectivity factors follow). We
4.6. EXPERIMENTAL ANALYSIS

study two update scenarios, (a) low frequency high volume updates (LFHV), and (b) high frequency low volume updates (HFLV). In the first scenario batch updates containing a large number of tuples occur with large intervals, i.e., many queries arrive between updates. In the second scenario, batch updates containing a small number of tuples happen more often, i.e., only a small number of queries have arrived since the previous updates. In all LFHV experiments we use a batch of $10^3$ updates after every $10^3$ queries, while for HFLV we use a batch of 10 updates after every 10 queries. Update values are randomly chosen in $[1, 10^7]$.

All experiments are conducted on a 2.4 GHz AMD Athlon 64 processor equipped with 2 GB RAM and two 250 GB 7200 rpm S-ATA hard disks configured as software-RAID-0. The operating system is Fedora Core 4 (Linux 2.6.16).

4.6.1 Basic Insights

For readability, we start with insertions to obtain a general understanding of the algorithmic behavior. We compare the update-aware cracker select operator against the scan-select operator of MonetDB and against an AVL-tree index created on top of the columns used. To avoid seeing the “noise” from cracking of the first queries we begin the insertions after a thousand queries have been handled. For example, for LFHV the first insertions arrive after a thousand queries and then after every thousand queries, while for the HFLV scenario the first insertions arrive after a thousand queries and then every ten queries. Figure 4.3 shows the results of this experiment for both LFHV and HFLV.

The $x$-axis ranks queries in execution order. The logarithmic $y$-axis represents the cumulative cost, i.e., each point $(x, y)$ represents the sum of the cost $y$ for the first $x$ queries. The figure clearly shows that all update-aware cracker select algorithms are superior to the scan-select approach. The scan-select scales linearly, while cracking quickly adapts and answers queries fast. The AVL-tree index has a high initial cost to build the index, but then queries can be answered fast too. For the HFLV scenario, FO is much more expensive. Since updates occur more frequently, it has to forget the cracker index frequently, restarting from scratch with only little time in between updates to rebuild the cracker index. Especially with MCI and MRI, we have maintained the ability of the cracking DBMS to reduce data access.

Notice, that both the ranges requested and the values inserted are randomly chosen, which demonstrates that all merge-like algorithms maintain the ability of a cracking DBMS to self-organize and adapt to query workload.
In the LFHV scenario (Figure 4.3 (a)) algorithms FO, MCI, MGI and MRI are all orders of magnitude faster than the normal select. For example, if we look at the total cost required for the $10^4$ queries to run (this is the point where the curves cross the right $y$-axis), FO and MGI are two orders of magnitude faster than the normal select while MCI and MRI are almost three orders of magnitude faster.

To discuss in more detail the various issues, we will use Figure 4.4. Figure 4.4 shows the cost per query through the complete LFHV scenario sequence. The scan-select has a stable performance at around 80 milliseconds while the AVL-tree has a high initial cost to build the index, but then query cost is never more than 3.5 milliseconds. When more values are inserted into the index, queries cost slightly more. Again FO behaves poorly. Each insertion incurs a higher cost to recreate the cracker index. After a few queries performance becomes as good as it was before the insertions.

MCI overcomes the problem of FO by merging the new insertions only when
4.6. EXPERIMENTAL ANALYSIS

Figure 4.4: Cost per query (LFHV)
CHAPTER 4. UPDATES

Figure 4.5: Number of pending insertions (LFHV)

requested for the first time. A single query suffers extra cost after each insertion batch. Moreover, MCI performs a lot better than FO in terms of total cost as seen in Figure 4.3, especially for the HFLV scenario. However, even MCI is problematic in terms of cost per query and predictability. The first query interested in one or more pending insertions suffers the cost of merging all of them and gets an exceptional response time. For example, a few queries carry a
response time of ca. 70 milliseconds, while the majority cost no more than one millisecond.

Algorithm MGI solves this issue. All queries have a cost less than 10 milliseconds. MGI achieves to balance the cost per query since it always merges fewer pending insertions than MCI, i.e., it merges only the tuples required for the current query. On the other hand, by not merging all pending insertions, MGI has to merge these tuples in the future when queries become interested. Going through the merging process again and again causes queries to run slower compared to MCI. This is reflected in Figure 4.3, where we see that the total cost of MGI is a lot higher than that of MCI.

MRI improves on MGI because it can avoid the very expensive queries. Unlike MGI it does not penalize the rest of the queries with an overhead. MRI performs the merging process only for the interesting part of the cracker column for each query. In this way, it touches less data than MGI (depending on where in the cracker column the result of the current query lays). Comparing MRI with MCI in Figure 4.4, we see the absence of very expensive queries, while comparing it with MGI, we see that queries are much cheaper. In Figure 4.3, we also see that MRI has a total cost comparable to that of MCI.

In conclusion, MRI performs better than all algorithms since it can keep the total cost low without having to penalize a few queries. Performance in terms of cost per query is similar for the HFLV scenario, too. The difference is that for all algorithms the peaks are much more frequent, but also lower, since they consume fewer insertions each time. We present a relevant graph later in this section.

4.6.2 Effect of the Number of Pending Insertions

To deepen our understanding on the behavior of the merge-like algorithms, we measure in this experiment the number of pending insertions left after each query has been executed. We run the experiment twice, having the requested range of all queries span $10^4$ and $10^6$ values, respectively.

In Figure 4.5, we see the results for the LFHV scenario. For both runs, MCI insertions are consumed very quickly, i.e., only a few queries after the insertions arrived. MGI continuously consumes more and more pending insertions as queries arrive. Finally, MRI keeps a high number of pending insertions since it replaces merged insertions with tuples from the cracker column (unless the pending insertions can be appended). For the run with the lower selectivity we observe for MRI that the size of the pending insertions is decreased multiple times through the query sequence which means that MRI had the chance to
simply append pending insertions to the cracker column.

### 4.6.3 Selectivity effect

Having sketched the major algorithmic differences of the merge-like update algorithms and their superiority compared to the non-cracking case, we discuss here the effect of selectivity. First, the algorithms are triggered only when the result of a query must contain part of the pending insertions. Obviously, the lower the selectivity of the query (i.e., the larger the result size) the higher the probability of this to happen. In addition, MGI and MRI merge only the values that are necessary for the current query so again selectivity can affect their performance.

For this experiment, we fire a series of $10^4$ random range queries that interleave with insertions as before. However, different selectivity factors are used such that the range spans over (a) 1 (point queries), (b) 100, (c) $10^4$ and (d) $10^6$ values.

In Figure 4.6, we show the cumulative cost. Let us first discuss the LFHV scenario. For point queries we see that all algorithms have a quite stable performance. With such a high selectivity, the probability of requesting a tuple from the pending insertions is very low. Thus, most of the queries do not need to touch the pending insertions, leading to a very fast response time for all algorithms. Only MCI has a high step towards the end of the query sequence, caused by a query that needs one tuple from the pending insertions, but since MCI merges all insertions, the cost of this query becomes high. As the selectivity drops, all update algorithms need to operate more often. Thus, we see higher and more frequent steps in MCI. For MGI observe that initially, as the selectivity drops, the total cost is significantly increased. This is because MGI has to go through the update process very often by merging a small number of pending insertions each time. However, when the selectivity becomes even lower, e.g., 1/10 of the column, MGI again performs well since it can consume insertions faster. Initially, with a high selectivity, MRI is faster in total than MCI but with dropping selectivity it loses this advantage due to the merging process being triggered more often. The difference in the total cost when selectivity is very low, is the price to pay for having a more balanced cost per query. MCI loads a number of queries with a high cost which is visible in the steps of the MCI curves. In MRI curves, such high steps do not exist.

For the HFHV scenario, MRI always outperforms MCI. The pending insertions are consumed in small portions very quickly since they occur more often. In this way, MRI avoids doing expensive merge operations for multiple values.
4.6. EXPERIMENTAL ANALYSIS

Figure 4.6: Effect of selectivity in cumulative cost in the LFHV (a,b,c,d) and in the HFLV (e,f,g,h) scenario
Figure 4.7: Effect of selectivity in per query cost
In Figure 4.7, we illustrate the cost per query for a low and a high selectivity for both scenarios. In general, the same pattern as in our first experiment can be observed. MRI maintains its advantage in terms of not penalizing single queries.

Let us first discuss the LFHV scenario. As we see in Figure 4.7 (a), MCI has a few high spikes when it is necessary to merge. All other queries are very fast. MGI has much lower cost per query while MRI even lower since it avoids touching not interesting parts of the cracker column. For a lower selectivity (Figure 4.7 (b)) the relative picture is the same. MRI and MGI have a few higher peaks than before but still a lot cheaper than MCI. For the HFLV scenario, our observations are quite similar. MRI again outperforms the rest of the algorithms with a very low cost per query. In the HFLV scenario, all algorithms have quite dense peaks. This is reasonable, because by having updates more often, we also have to merge more often, and thus we have fewer tuples to merge each time. In addition, MCI has lower peaks compared to the previous scenario, but still much higher than MRI.

4.6.4 Long Query Sequences

All previous experiments were for a limited query sequence of $10^4$ queries interleaved with updates. Here, we test for sequences of $10^5$ queries. As before, we test with a column of $10^7$ tuples, while the queries request random ranges that span over $10^4$ values. Figure 4.8 shows the results. Compared to our previous experiments, the relative performance is not affected (i.e., MRI maintains its advantages), which demonstrates the algorithmic stability. All algorithms slightly increase their average cost per query until they stabilize after a few thousand queries. However, especially for MRI, the cost is significantly smaller than that of an AVL-tree index or the scan-select operator. The reason for observing this increase, is that with each query the cracker column is physically reorganized and split to more and more pieces. In general, the more pieces in a cracker column, the more expensive a merge operation becomes, because more tuples need to be moved around.

In order to get the very last bit of performance, our future work plans include research in allowing a cracker column/index to automatically decide to stop splitting the cracker column into smaller pieces or decide to merge existing pieces together so that the number of pieces in a cracker column can be a controlled parameter.
Figure 4.8: Effect of long query sequences
4.6. EXPERIMENTAL ANALYSIS

4.6.5 Performance under Deletions

Switching our experiment focus to deletions produces similar results. The relative performance of the algorithms remains the same. For example, on a cracker column of $10^7$ tuples, we fire $10^4$ range queries that request random ranges of size $10^4$ values. We test both the LFHV scenario and the HFLV scenario.

In Figure 4.9, we show the cumulative cost and compare it against the MonetDB scan-select that always scans a column and an AVL-tree index. The AVL-tree uses lazy deletes, i.e., spot the appropriate node and mark it as deleted so that future queries can ignore it. As with insertions, all cracker update algorithms are superior to the AVL-tree index and the scan-select. Figure 4.10 shows the cost per query (for the LFHV case), where we observe the same pattern we saw for insertions with the ripple version, the MRD algorithm, outperforming all others. The same stands for the rest of the experiments we did for deletions to see the effect of selectivity, the effect of the size of the query sequence and so
on. Due to space restrictions we omit these results.

An interesting difference between insertions and deletions, is that the latter requires finding the actual position for a pending deleted tuple. As we described in Section 4.4, this is done with a join operation between the respective parts of the cracker column and the pending deletions column. This is more expensive
4.6. EXPERIMENTAL ANALYSIS

Figure 4.11: Cost to locate deletes for MCD

when the cracker pieces are large. For this reason the pattern shown graphically in Figure 4.11 is relevant. It shows only the queries that do an update for MCD in our previous experiment. We depict the total cost for each query and the cost to locate the deletes removed from the cracker column. Observe that initially, e.g., for the first query that is forced to update, the total cost is mainly due to the cost of locating tuples to be deleted. The rest of the merge process is quite cheap, since with fewer pieces in the cracker column, fewer tuples need to be moved. The next query that starts an update has a much lower total cost. It can locate deletes much faster due to having smaller pieces in the cracker column (around $10^3$ queries have cracked the column in between). For the remaining update queries, the cost to locate deletes is continuously becoming smaller due to the cracker pieces becoming smaller. The total cost remains quite stable, because by having smaller pieces we also need to move more tuples while removing deletes. This pattern exists in the other algorithms, too, e.g., observe MRD in Figure 4.10. After the first thousand queries, when the first update happens, the cost per query is higher compared to that of future queries that handle smaller pieces in the cracker column.
4.6.6 Full Updates Performance

By now it should be clear that updates do not produce any surprises. The same patterns emerge, i.e., the combination of the ripple algorithms is the one that outperforms all others having the lowest and most stable cost per query along with a low total cost. Due to space restrictions (and similarity of results) we show only the cost per query for the merge-like algorithms. As before, the experiments are based on a column of $10^7$ tuples, where we fire $10^4$ range queries that request random ranges of size $10^4$ values. A thousand updates arrive every thousand queries.

The results are shown in Figure 4.12. The only difference is that queries that need to consume both pending insertions and pending deletions cost slightly
more. For example, the combination of the gradual algorithms and the combination of the ripple algorithms never drop below 100 microseconds (as more queries arrive), which was often the case in the previous experiments. However, the relative performance is the same and still significantly lower than that of an AVL-tree or the scan-select, especially for the ripple case.

4.7 Summary

Just-enough and just-in-time are the ingredients in cracked databases. The physical store is extended with an efficient navigational index as a side product of running query sequences. It removes the human from the database index administration loop and relies on self-tuning by adaptation.

In this chapter, we extended the approach towards volatile databases. Several novel algorithms are presented to deal with database updates using the cracking philosophy. The algorithms were added to an existing open-source database kernel and a broad range experimental analysis, including a comparison with a competitive index scheme, clearly demonstrates the viability of cracking in column-stores.

With these promising results the road for many more discoveries of self-* database techniques lies wide open. The following chapters show how we can exploit cracking for efficient tuple reconstruction and arbitrary join processing in a column-store.