Database Cracking

Stratos Idreos
CWI Amsterdam
The Netherlands
Stratos.Idreos@cwi.nl

Martin L. Kersten
CWI Amsterdam
The Netherlands
Martin.Kersten@cwi.nl

Stefan Manegold
CWI Amsterdam
The Netherlands
Stefan.Manegold@cwi.nl

ABSTRACT

Database indices provide a non-discriminative navigational infrastructure to localize tuples of interest. Their maintenance cost is taken during database updates. In this paper, we study the complementary approach, addressing index maintenance as part of query processing using continuous physical reorganization, i.e., *cracking* the database into manageable pieces. The motivation is that by automatically organizing data the way users request it, we can achieve fast access and the much desired self-organized behavior.

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

1. INTRODUCTION

Nowadays, the challenge for database architecture design is not in achieving ultra high performance but to design systems that are *simple* and *flexible*. A database system should be able to handle *huge* sets of data and *self-organize* according to the environment, e.g., the workload, available resources, etc. A nice discussion on such issues can be found in [6]. In addition, the trend towards distributed environments to speed up computation calls for new architecture designs. The same holds for multi-core CPU architectures that are starting to dominate the market and open new possibilities and challenges for data management. Some notable departures from the usual paths in database architecture design include [2, 3, 8, 14].

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

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

The idea of physically reorganizing the database based on incoming queries has first been proposed in [9]. The contributions of this paper are the following. We present the first mature cracking architecture (a complete cracking software stack) in the context of column oriented databases. We report on our implementation of cracking on top of MonetDB/SQL, a column oriented database system, showing that cracking is easy to implement and may lead to further system simplification. We present the cracking algorithms that physically reorganize the datastore and the new cracking operators to enable cracking in MonetDB. Using SQL micro-benchmarks, we assess the efficiency and effectiveness of the system at the operator level. Additionally, we perform experiments that use the complete software stack, demonstrating that cracker-aware query optimizers can successfully generate query plans that deploy our new cracking operators and thus exploit the benefits of database cracking. Furthermore, we evaluate our current implementation and discuss some promising results. We clearly demonstrate that the resulting system can self-organize according to query

workload and that this leads to significant improvement to data access response times. This behavior is visible even when the user focus is randomly shifting to different parts of the data.

Finally, we sketch a research landscape with a large number of challenges and opportunities that cracking brings into database design. We argue that cracking can be explored in multiple areas of database research to bring the property of self-organization, e.g., distributed and parallel databases, P2P databases etc.

The remainder of this paper is organized as follows. In Section 2, we briefly recap the MonetDB system. Section 3 discusses database cracking in more detail and introduces the cracking architecture. Then, Section 4 motivates cracking against traditional index based strategies. In Section 5, we present the algorithms to perform physical reorganization while Section 6 describes some cracking operators and their impact on the query plan. In Section 7, we provide an evaluation of our current implementation. Then, Section 8 discusses open research and opportunities. Finally, in Section 9, we discuss related work, and Section 10 concludes the paper.

2. THE MONETDB SYSTEM

In this section, we will briefly describe the MonetDB system to introduce the necessary background for the rest of our presentation.

MonetDB[1] differs from the mainstream systems in its reliance on a decomposed storage scheme, a simple (closed) binary relational algebra, and hooks to easily extend the relational engine. In MonetDB, every n-ary relational table is represented by a group of binary relations, called *BAT*s [5]. A BAT represents a mapping from an `oid`-key to a single attribute `attr`. Its tuples are stored physically adjacent to speed up its traversal, i.e., there are no holes in the data structure. The `oid`s represent the identity of the original n-ary tuples, linking their attribute values across the BATs that store an n-ary table. For base tables, they form a dense ascending sequence enabling highly efficient positional lookups. SQL statements are translated by the compiler into a query execution plan composed of a sequence of simple binary relational algebra operations. In MonetDB, each relational operator materializes the result as a temporary BAT or a view over an existing BAT. For example, assume the following query:

select R.c from R where $5 \leq$ R.a ≤ 10 and $9 \leq$ R.b ≤ 20

This query is translated into the following (partial) plan:

```
Ra1 := algebra.select(Ra, 5, 10);
Rb1 := algebra.select(Rb, 9, 20);
Ra2 := algebra.OIDintersect(Ra1, Rb1);
Rc1 := algebra.fetch(Rc, Ra2);
```

This plan uses three binary relational algebra operations:

- `algebra.select(b,low,high)` searches all (`oid,attr`) pairs in `b` that satisfy the predicate `low` $\leq$ `attr` $\leq$ `high`.
- `algebra.OIDintersect(r,s)` returns all (`oid,attr`) pairs from `r` where `r.oid` is found in `s.oid`, implementing the conjunction of the selection predicate.
- `algebra.fetch(r,s)` returns all (`oid,attr`) pairs that reside in `r` at the positions specified by `s.oid`, performing the projection.

The relational operations are grouped into modules, e.g. `algebra`, `aggr` and `bat`. The modules represent a logical grouping and they provide a name space to differentiate similar operations.

The actual implementation consists of several tens of relational primitive implementations, but they are ignored here for brevity. They do not introduce extra complexity either. Each relational operator is a function implemented in C and registered in the database kernel using its extension mechanism.

We will now proceed with the rest of our cracking description to see the cracking architecture and how cracking can be realized into a real query plan. In the rest of the paper, we will use the simple example of the MonetDB query plan introduced in this section, to demonstrate how cracking affects it and what modifications had to be done to correctly and efficiently plug-in cracking.

3. CRACKING

In this section, we will introduce the cracking architecture. We will see the necessary data structures and a simple example. Let us now describe how cracking takes place in a column oriented database. It is as follows:

- The *first time* a range query is posed on an attribute A, a cracker database makes a copy of column A. This copy is called the *cracker column* of A, denoted as A_{CRK}.
- A_{CRK} is continuously physically reorganized based on queries that need to touch attribute A.

DEFINITION. Cracking *based on a query q on an attribute A is the act of* physically reorganizing A_{CRK} *in such a way that the values of A that satisfy q are stored in a contiguous space.*

Creating a copy of the column and cracking on it is useful, as it leaves the original column intact, allowing fast reconstruction of records by exploiting the insertion order. In a column oriented database system, every attribute in a relation is represented as a column. Thus, when a query needs to see multiple attributes of the same relation, then tuples in multiple columns have to be combined to produce the result. As shown in [4, 10], this can be done efficiently (by avoiding random access) if all tuples with the same *id* (i.e., attribute values that belong to the same row) are placed in the same position in each column and this position is reflected by the value of *id*. In the cracker columns the original position (insertions sequence) of the tuples is spoiled by physical reorganization. Consequently, we will use the cracker columns for fast value selections while we will use the original columns for efficient projections, exploiting positional lookups based on the tuples' *id*s.

The cracker column is being continuously split into more and more pieces as queries arrive. Thus, we need a way to be able to quickly localize a piece of interest in the cracker

[1]For a complete description, see the documentation of MonetDB Version 5 at `http://monetdb.cwi.nl/`.

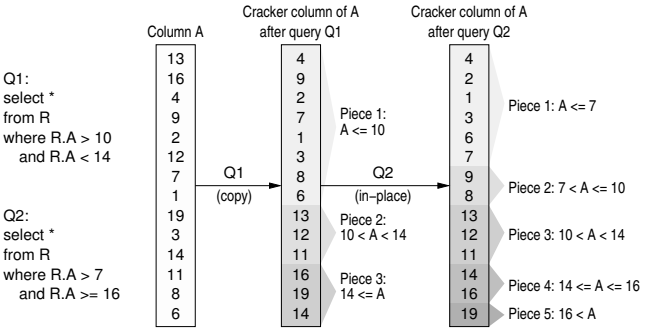


Figure 1: Cracking a column

column. For this purpose, we introduce for each cracker column c a *cracker index*, that maintains information on how values are distributed in c. In our current implementation, the cracker index is an AVL-tree. Each node in the tree holds information for one value v, i.e., it stores a position p referring to the cracker column such that all values that are before position p are smaller than v and all values that are after p are greater. Whether v is left or right inclusive is also maintained in a separate field in the node.

This information can be used to speed up subsequent queries significantly, i.e., queries that request ranges that are an exact match on values known by the index can be answered at the cost of searching the index, only. Even if there is no exact match, the index significantly restricts the values of the column that a query has to analyze.

Consider the example in Figure 1. First, query Q1 triggers the creation of cracker column A_{CRK}, i.e., a copy of column A where the tuples are clustered in three pieces, reflecting the ranges defined by the predicate. The result of Q1 is then retrieved at no extra cost as a *view* on Piece 2 that does not require additional copying of the respective data. We also refer to such views as *column slices*. Later, query Q2 benefits from the information in the cracker index, requiring an in-place refinement of Pieces 1 & 3, only, splitting each in two new pieces, but leaving Piece 2 untouched. The result of Q2 is again a zero-cost column slice, covering the contiguous Pieces 2–4. A third query requesting $A > 16$ would then even exactly match the existing Piece 5.

The observation is that by "learning" what a single query can "teach" us, we can speed up multiple queries in the future that request similar, overlapping or even disjoint data of the same attribute. Some of the questions that immediately arise are the following.

1. How does this compare to sorting, i.e., why not sort upfront?
2. How expensive is it to physically reorganize (part of) the column each time and how we do that?
3. How does cracking fit in the query plan of a modern DBMS?

We will try to address these issues in detail and motivate our choices at each step. Question 1 will be discussed in Section 4, Question 2 in Section 5 and finally Question 3 in Section 6. At this point, note that cracking is an open research topic and our goal is to carefully identify the research space, the opportunities and the pitfalls. The goal of the paper is not to present solid solutions for each issue that appears but to argue that the cracking direction deserves our attention and research since: (a) it is possible to design and implement a cracking database system, (b) clear benefits appear from preliminary results, and (c) multiple exciting opportunities set a promising future research agenda for cracking.

3.1 Cracking a row oriented database

In this work, cracking is described in the context of column oriented databases. However, we envision that cracking can potentially be a useful strategy for row-oriented databases too. For example, the scheme described in this section is applicable in a row oriented database as well with minor modifications. A row oriented database needs to create a cracker column for the attribute that is to be cracked. This could be a binary relation storing reference-value (r, v) pairs where for each value v, reference r points to the location where the actual record that v belongs to is stored. Such an architecture could also be the basis for a hybrid DSM/NSM system with the cracker column providing a fast entry point to interesting data in the query plan and then the original records of the row oriented system will give the rest of the data without any need for combining multiple columns to reconstruct records.

4. CRACKING VS. SORTING & INDICES

One of the main questions is how cracking compares to a sort-based strategy and to traditional indices. In this section, we will try to motivate cracking as an alternative strategy targeted to specific environment conditions that cannot be handled efficiently by a sort-based or a traditional index-based strategy.

Consider sorting first since this may be a more natural question. A cracking strategy eventually enforces order in a cracker column, i.e., the pieces of a cracker column are ordered. Every value in a cracker piece p is larger than every value in all pieces p_i such that p_i is before p in the cracker column. Similarly every value in p is smaller than every value in all pieces p_j such that p_j is after p in the cracker column. So let us discuss how this cracking scheme compares to a sorting strategy, i.e., sort the data upfront and then perform very fast binary search operations.

Assume an environment where it is known *upfront which data is interesting* for the users/queries, i.e., which single (combination of) attribute(s) is primarily requested, and hence should determine the physical order of tuples. Assume also that there is the luxury of *time and resources* to create this physical order *before* any query arrives and that there are no updates or the time difference between any update and an incoming query is sufficient for maintaining this physical order (or maintaining the appropriate indices that provide an order, e.g., B-trees). If all these are true, then sorting is a superior strategy.

Cracking is not challenging any strategy in such conditions. Instead, it mainly targets environments where:

- there is not any knowledge about which part of the data is interesting, i.e., which attributes and ranges (and with what selectivities) will be requested.
- there is not enough time to restore or maintain the physical order after an update.

Algorithm 1 CrackInTwo(c,posL,posH,med,inc)
Physically reorganize the piece of column c between $posL$ and $posH$ such that all values lower than med are in a contiguous space. inc indicates whether med is inclusive or not, e.g., if $inc = false$ then θ_1 is "<" and θ_2 is ">="

```
 1: x1 = point at position posL
 2: x2 = point at position posH
 3: while position(x1) < position(x2) do
 4:   if value(x1) θ1 med then
 5:     x1 = point at next position
 6:   else
 7:     while value(x2) θ2 med &&
        position(x2) > position(x1) do
 8:       x2 = point at previous position
 9:     end while
10:     exchange(x1,x2)
11:     x1 = point at next position
12:     x2 = point at previous position
13:   end if
14: end while
```

Additionally, cracking allows to maintain independent individual physical orders for each attribute. In the experiments section, cracking is compared against the sort approach to demonstrate these issues. We will clearly show that cracking is a lightweight operation compared to sorting and that a cracking strategy needs no upfront knowledge to achieve fast data access.

The same arguments stand against any indexed-based strategy or any strategy that tries to prepare by cleverly clustering together interesting data. For such a strategy, workload knowledge is necessary to identify the interesting data. Additionally, a time and a resource investment has to be done upfront to prepare the data. Thus, similarly to our previous discussion, a strategy that knows the query patterns and can afford to properly prepare and maintain the data is superior to cracking. But if these circumstances are not true, then cracking is a promising alternative.

5. CRACKING ALGORITHMS

In this section, we will present the algorithms that perform the physical reorganization of a column.

Physical reorganization or cracking is an operation that operates on an entire column or on a column slice. Two basic cracking operations are considered, called *two-piece* and *three-piece* cracking, respectively. They both have the effect that they physically reorganize a column of an attribute A given a range predicate such as all values of A that satisfy the predicate are in a contiguous space. The former splits the given column into two new pieces using single-sided predicates $A\ \theta\ med$ while the latter uses double-sided predicates $low\ \theta_1\ A\ \theta_2\ high$, where low, $high$ and med are values in the value range of A and θ, θ_1 and θ_2 are bound conditions. Three-piece cracking is semantically equivalent to two subsequent two-piece crackings ($low\ \theta_1\ A$ & $A\ \theta_2\ high$). However, being a single-pass algorithm itself, it provides a faster alternative for double-sided predicates.

The algorithms for cracking are formally described in Algorithms 1 and 2. Both algorithms touch as little data as possible. The core idea is that, while going through the tuples of a column using 2 or 3 pointers to read, cases where two tuples can be exchanged are carefully identified. The cracking algorithms are cache conscious in the sense that they always try to exploit tuples that are recently read if these tuples must be touched again in the future.

Algorithm 2 CrackInThree(c,posL,posH,low,high,incL,incH)
Physically reorganize the piece of column c between $posL$ and $posH$ such that all values in the range low $high$ are in a contiguous space. $incL$ and $incH$ indicate whether low and $high$ respectively are inclusive or not, e.g., if $incL = false$ and $incH = false$ then θ_1 is ">=", θ_2 is ">" and θ_3 is "<"

```
 1: x1 = point at position posL
 2: x2 = point at position posH
 3: while value(x2) θ1 high &&
    position(x2) > position(x1) do
 4:   x2 = point at previous position
 5: end while
 6: x3 = x2
 7: while value(x3) θ2 low &&
    position(x3) > position(x1) do
 8:   if value(x3) θ1 high then
 9:     exchange(x2,x3)
10:     x2 = point at previous position
11:   end if
12:   x3 = point at previous position
13: end while
14: while position(x1) <= position(x3) do
15:   if value(x1) θ3 low then
16:     x1 = point at next position
17:   else
18:     exchange(x1,x3)
19:     while value(x3) θ2 low &&
        position(x3)>position(x1) do
20:       if value(x3) θ1 high then
21:         exchange(x2,x3)
22:         x2 = point at previous position
23:       end if
24:       x3 = point at previous position
25:     end while
26:   end if
27: end while
```

Multiple algorithms were created before ending up with these simple ones. Algorithms that tried to invest in cleverly detecting situations that required fewer exchange operations or allowed early retirement, turned out to be more expensive due to more complex code (e.g., more branches).

We also experimented with "stable" cracking algorithms, i.e., maintain the insertion order of tuples with values in the same range (i.e., tuples that belong in the same piece of the cracker column). The algorithms became more complex and two times slower. The fast stable algorithms required a buffer space to temporarily store values subject to move, which is an extra memory overhead. The goal was to exploit this order for faster reconstructions of records, but this is not yet verified. The experiments reported use non-stable cracking.

The two algorithms presented in this section are sufficient to cover the needs of a column oriented cracking DBMS in terms of physical reorganization. Obviously, the second algorithm that performs three-piece cracking is significantly more expensive than the two-piece cracking algorithm. It is a more complex algorithm with more pointers,

more branches etc. However, notice that in practice two-piece cracking is the algorithm that is actually used more often. This is the case even when queries use double-sided predicates. Three-piece cracking is used only when all the tuples with values in the range requested by a select operator fall into the *same piece* of the cracker column. This is more likely to happen for the first queries that physically reorganize a column. While the column is split in multiple pieces the chances of requesting a result that falls into only one piece decrease. In general, the range requested will span over multiple contiguous pieces of a cracker column and then only the first and the last piece must be physically reorganized using two-piece cracking. For example, recall the example in Figure 1. Query Q1 uses a double-sided predicate and since it is the first query that cracks the column it uses three-piece cracking. However, the second query Q2 that also uses a double-sided predicate does not have to use three-piece cracking. The interesting tuples for Q2 span over Pieces 1, 2 and 3. Cracking does not need to analyze Piece 2 since it is known that all tuples there qualify for the result. Thus, only Piece 1 is physically reorganized with two-piece cracking to create two new pieces; one that qualifies for the result and one that does not. Likewise for Piece 3.

6. THE CRACKING QUERY PLANS

Let us proceed on how we fit the cracking technique in the query plan generator. Our description is based on MonetDB, where the relational algebra is extended with a small collection of cracker-aware operations. The relational query plan can easily be transformed into one that uses cracker-based algorithms. Eventually, we envision that many operators will be able to benefit from cracking by using information in the cracker index and column.

6.1 The `crackers.select` operator

The first step is to replace the `algebra.select` operator. Recall that the motivation is that cracking happens *while processing queries* and based on queries. The select operator is typically the main operator that provides access on data and typically feeds the rest of the operators in a query plan. Thus, it is a natural step to choose to experiment with this operator first.

As already discussed, operators in MonetDB materialize their result. Here is how a simple select operation works: it receives the column storing a specific attribute, scans it, and creates a new column, containing only tuples with values that satisfy the selection predicate. In our case, in order to explore cracking the select operation will be responsible for the physical reorganization part too. Thus, this new operation should work as follows:

1. search in the cracker index to determine which piece(s) of the cracker column should be touched/physically reorganized
2. physically reorganize the cracker column
3. update the cracker index
4. return a slice of the cracker column as the result (at zero cost).

We extended the algebra with the `crackers.select` operator that performs the above steps. Although this may at first seem as if we added additional overhead in the select operation, this is not the case. The fact that the number of tuples to be touched/analyzed is significantly restricted, along with a carefully crafted implementation of the physical reorganization step, leads to an operation orders of magnitude faster than `algebra.select` in MonetDB that needs to scan all tuples in the column for each query. The initial `crackers.select` call on an attribute is typically 30% slower than `algebra.select` since it has to reorganize large parts of the column. However, *all* subsequent calls will be faster. As more queries arrive, a select operation on the same attribute becomes cheaper since the cracker index learns more and allows us to touch/analyze smaller pieces of the column (see Section 7 for a detailed experimental analysis).

6.2 The `crackers.rel_select` operator

In the previous paragraphs we described the crackers select function. Let us now see what is the effect of replacing a simple select with a crackers select in the query plan of MonetDB. Assume the query plan in Section 2. This will be replaced with the following plan.

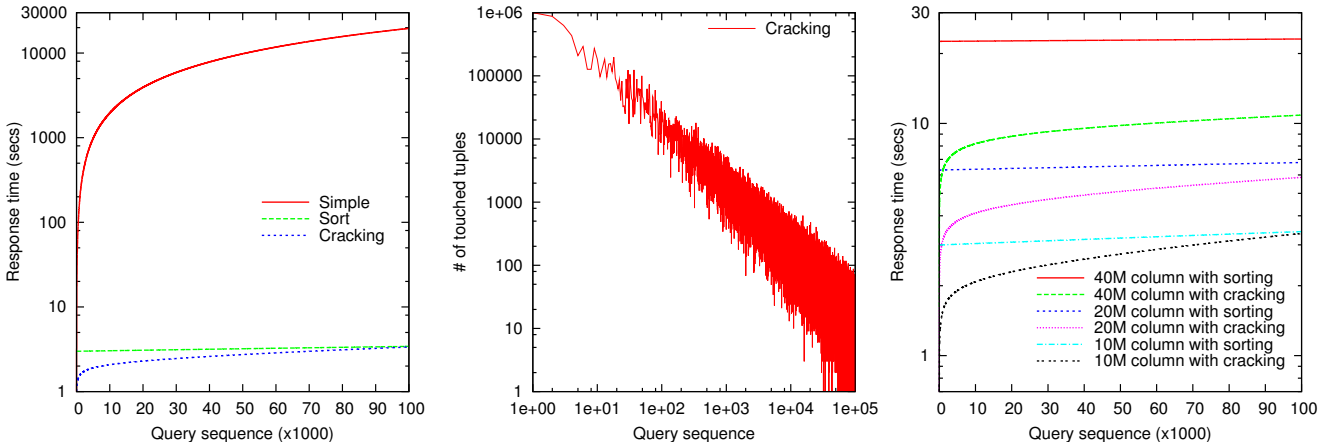
```
Ra1 := crackers.select(Ra, 5, 10);
Rb1 := crackers.select(Rb, 9, 20);
Ra2 := algebra.OIDintersect(Ra1, Rb1);
Rc1 := algebra.fetch(Rc, Ra2);
```

Although the `crackers.select` is much faster, its initial overall effect on the plan turned out to be negligible. The reason is that the `OIDintersect` became more expensive. MonetDB's `algebra.select` produces a result column that is ordered on the insertion sequence of its `oid` values. In this way, a subsequent `OIDintersect` can be executed very fast in MonetDB, by exploiting the `oid`-order of both operands in a merge-like implementation (i.e., it avoids any random memory access patterns). However, the `crackers.select` returns a column that is no longer ordered on `oid`, since it is physically reorganized. This results in a more expensive `OIDintersect` for cracking, requiring a hash-based implementation with inherent random access. For example, experiments with TPC-H query 6 of scale factor 0.1, led to `OIDintersect` being 7 times more expensive.

This side-effect opens a road for detailed studies on both the algebraic operations used and the plan generation scheme of the SQL compiler. Cracker-aware query plans are called for. An example is the new cracking operator `rel_select`. The goal is to completely avoid the `OIDintersect`. The `rel_select` replaces a pair of a `select` and a `OIDintersect`, performing both simultaneously. It takes an intermediate column c_1 and a base column c_2 as arguments. Due to cracking, c_1 is no longer ordered on `oid`. However, being an un-cracked base column, c_2 has a dense sequence of `oid` values, stored in ascending order. Thus, iterating over c_1, `rel_select` exploits very fast positional lookup into c_2 to find the matching tuple (c_1.`oid` = c_2.`oid`), and subsequently checks the selection predicate on c_2.`attr`. The plan transformation was handled readily by the optimizer infrastructure, leading to the following plan for our example.

```
Ra1 := crackers.select(Ra, 5, 10);
Rb1 := crackers.rel_select(Rb, 9, 20, Ra1);
Rc1 := algebra.fetch(Rc, Rb1);
```

This minimal set of just the two cracker operators introduced in this section was sufficient to significantly improve



(a) Selecting on a 10 million column (b) Portion touched by crack select (c) Cracking larger columns

Figure 2: Comparing the crackers select against the simple and the sort strategy

the performance of SQL queries. We explore a much larger set of algebraic cracking operators to exploit the information available in the cracker columns and the cracker indices.

7. EXPERIMENTATION

In this section, we peek into the experimental analysis of our current implementation. The development of cracking was done in MonetDB 5.0alpha and its SQL 2.12 release. All experiments were done on a 2.4 GHz AMD-Athlon 64 with 2 GB memory and a 7200 rpm SATA disk. The experiments are based on a complete implementation. We first discuss a micro-benchmark to assess the individual relational operations. Then, we present results obtained using the complete software stack. Together they provide an outlook on the impact of database cracking.

7.1 Select operator benchmark

From the large collection of micro-experiments we did to arrive at an efficient implementation, we summarize the behavior of the cracker select against traditional approaches. We tested three select operator variants, (a) simple, (b) sort, and (c) crack, against a series of range queries on a given attribute A. Case (a) uses the default MonetDB select operator, i.e., it scans the column and picks the tuples with values that satisfy the predicate. This is done for every query. Case (b) first performs a sort to physically reorganize the column based on the values of the existing tuples. Then, for each incoming query, it picks the tuples with qualifying values using binary search (e.g., the result is always in a contiguous space in the sorted column). Finally, case (c) uses cracking, i.e., it physically reorganizes (part of) the column for each query. The column has 10^7 tuples (distinct integers from 1 to 10^7). Each query is of the form $v_1 < A < v_2$ where v_1 and v_2 are randomly chosen.

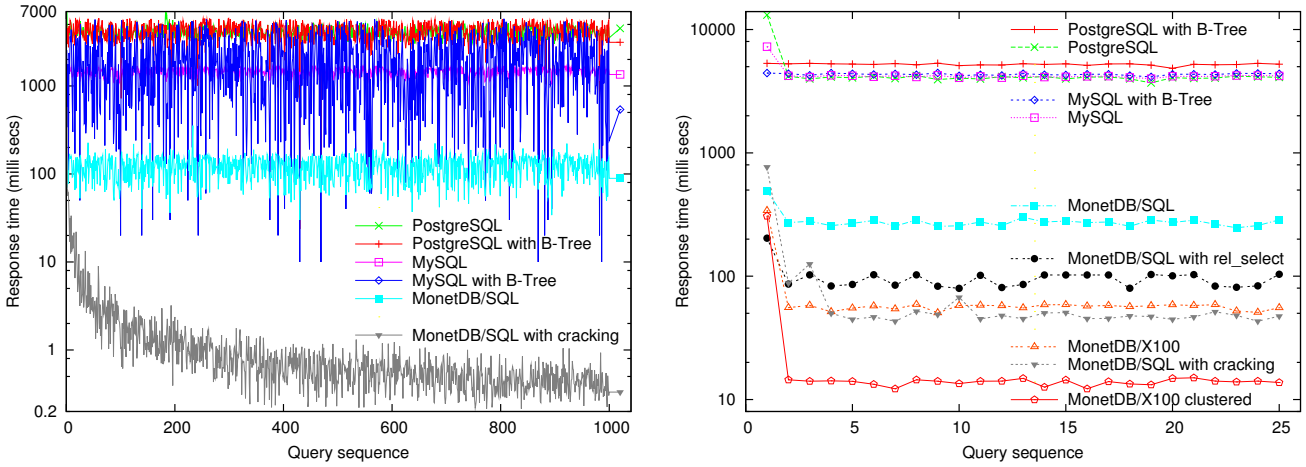
Figure 2(a) shows the results of this experiment. On the x-axis queries are ranked in execution order. The y-axis represents the cumulative time for each strategy, i.e., each point (x, y) represents the sum of the cost y for the first x queries. Evidently, cracking and sorting beat the simple scan approach in the long run. The simple scan grows linearly with the number of queries. After sorting the data, the cost of selecting any range in the column is in a few micro seconds. The overhead is the sorting phase. The sort loads the first query with an extra cost of 3 seconds, while a simple scan-select needs 0.27 seconds, and the first cracking operation costs 0.38 seconds. For cracking, only the first query is slightly more expensive compared to simple select. All subsequent queries benefit from previous ones and are faster since they do not analyze every column value.

The cost of cracking highly depends on the size of the piece that is being physically reorganized. This is the reason why as more queries come cracking becomes cheaper. This is visible in Figure 2(a) by observing the pace with which the cracking curve grows. Initially, the curve grows faster and then as more queries arrive, smaller pieces are cracked and the curve grows with a smaller pace. Note, that in the experiment of Figure 2(a) cracking has a cost ranging from 10 to 100 micro seconds. Our data show that this variation is due to the size that is cracked each time. Figure 2(b) shows the portion of the column that is analyzed each time by the crackers select operator. The more queries arrive, the less data need to be touched/physically reorganized.

A critical point when comparing cracking with sort is to determine the break-even point, i.e., when their cumulative costs become even. For a 10 M values table, this is at around 10^5 queries in our current implementation (see Figure 2(a)). Then, the investment of sorting starts to pay off. The overhead of penalizing the first queries however remains.

Figure 2(c) shows how sort and cracking scale on larger columns. As the size increases, cracking becomes more advantageous. For example, observe the points after 10^5 queries to see what cracking gains. This phenomenon can be explained considering the algorithmic complexity. The first cracking operation has a complexity of $O(N)$, where N is the size of the column, while sorting costs $O(N \log N)$.

The results shown in Figure 2 are subject to significant improvements by a more "intelligent" maintenance strategy. The break-even point can be shifted further into the future. For example, in Figure 2(b) we see that already after the first 8-10 queries cracking touches an order of magnitude less data and becomes significantly faster (10 times faster than simple select). In the experiments so far, the cracker



(a) A simple count(*) range query

(b) TPC-H query 6

Figure 4: Effect of cracking in some simple queries

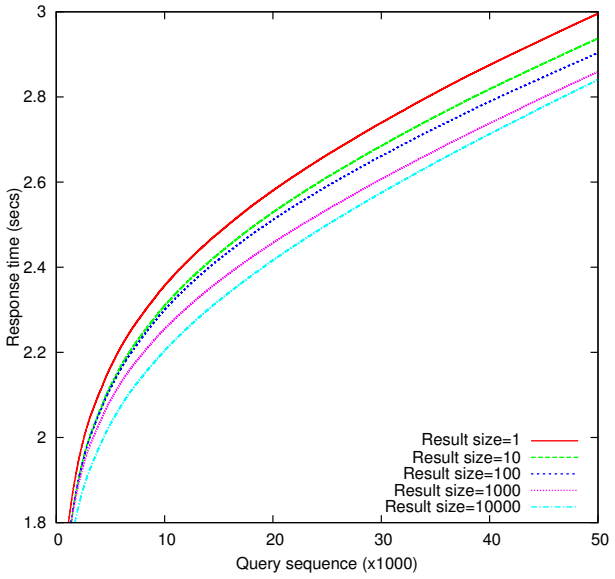


Figure 3: Effect of selectivity in the learning process

index is always updated and some portion is physically reorganized. Cut-off strategies prove effective in this area, i.e., we can disable index updates once the differential cost of subsequent selects drops below a given threshold. Each time the index is updated we pay the cost of this update and future searches in the index become more expensive. We are exploring strategies where there is the option to update the index or not. Experiments where index updating is manually disabled after a number of queries proved this concept by speeding up future crackers select calls. Here a cost model is needed to make a proper decision based on current workload, size of the column etc.

Figure 3 shows the effect of selectivity on cracking. As before, on a column of 10^7 tuples a series of range queries is fired. This time we vary the selectivity by requesting such ranges that the result size is always S tuples. The area where a query range falls into the value space is still random. The experiment is repeated for S=10^0,10^1,10^2,10^3 and 10^4.

In Figure 3, we show the cumulative cost for each run. The main result seen is that *the lower the selectivity the less effort* is needed for cracking to reach optimal performance. This is explained as follows. With higher selectivities cracking creates small pieces in a cracker column to be the result of the current query but leave large pieces (those outside the result set for the current query) to be analyzed by future queries. In this way, future queries have a high probability to physically reorganize large pieces. On the contrary, when selecting large pieces the cracker column is partitioned more quickly (with less queries) in more even pieces. However, from Figure 3 it is clear that this pattern can be observed but it is not one that dramatically affects performance for cracking. In addition, the fact that ranges requested are random clearly indicates that cracking successfully brings the property of self-organization independently of the selectivities used. In all cases, results obtained outperform those of a sort based or a scan based strategy in a similar way as observed in Figure 2.

7.2 Full query evaluation

A novel query processing technique calls for an evaluation in the context of a fully functional system. In this section, we provide an outlook on the evaluation of cracking using SQL queries processed by MonetDB/SQL. Figure 4(a) shows the results for the following query.

$$\text{select count(*) from R where R.a} > v_1 \text{ and R.a} < v_2$$

The same experiment is ran with PostgreSQL and MySQL both with and without using a B-tree on the attribute. To avoid seeing the DSM/NSM effect, a a single column table is used populated with 10^7 randomly created values between 0 and 9999. We fire a series of a thousand random queries. Cracking quickly learns and significantly reduces the cost of MonetDB (eventually more than two orders of magnitude) while the rest of the systems maintain a quite stable performance. Since queries are random, selectivities are random. This example clearly demonstrates the ability of cracking to

adapt in an environment where there is no upfront knowledge of what queries will be fired. For example, observe that MySQL with a B-tree also reaches high performance for some queries. These are queries with high selectivity where the B-tree becomes useful. However, in order to maintain such performance levels, a traditional system needs upfront knowledge and a stable query workload.

Figure 4(b) shows the results for TPC-H query 6. Again, cracking significantly improves the performance of MonetDB. The graph flattens quickly due to the limited variation in the values of the lineitem shipdate attribute (which is the one that is being cracked). All systems have an extra cost for the first query since this includes the cost of fetching the data. Since in this query the `rel_select` operator is used, we include a run with the new `rel_select` operator, but without cracking, to clearly show the positive cracking effect. Detailed analysis of the trace shows room for further improvement by exploiting the cracking information in other operators as well.

We also include results obtained with the MonetDB/X100 prototype [15, 16]. Its architecture is aimed at pure pipelined query performance and currently lacks a SQL front-end. With cracking enabled, MonetDB/SQL performs slightly better than MonetDB/X100 on unclustered data. It can be beaten using a pre-clustered, compressed table and a hand-compiled query plan, ignoring administrative and transaction overhead. Moreover, the cost of the initial clustering is ignored. This way, it shows the base-line performance achievable in an ideal case. Since the techniques in both source lines are orthogonal, we expect that applying cracking in MonetDB/X100 will have a significant effect as well.

8. A RESEARCH LANDSCAPE

The study underway explores the consequences of cracking in all corners of a database system architecture. It calls for a reconsideration of both its algebraic core and its impact on plan generation. In this section, we give a snippet of the topics under investigation, open research issues and challenging opportunities for cracking in a modern DBMS.

8.1 Optimizing for cracked query plans

There are some clear optimization opportunities in a cracked query plan. For example, the cracked query plans for TPC-H illustrate many opportunities for query optimization. With the introduction of the `crackers.rel_select` operator only one attribute/column for each distinct relation involved in a query, is physically reorganized. Thus, a *choice* needs to be taken to decide which attribute to crack each time. This is a choice that can affect performance significantly. Possible criteria may be the existence or not of a cracker index and a cracker column, e.g., if there is a choice between two attributes A_1 and A_2, where A_1 has already been cracked in the past while A_2 has not, then it could be of benefit to crack A_1. This is because if A_2 is cracked, then a copy of the A_2 column has to be created to be the cracker column and the cracking algorithm has to go through all tuples of A_2 since this would be the first cracking operation on this attribute. However, it is not clear whether this direction will always be the right choice to make. For example, if the query is not selective enough in A_1 and highly selective in A_2 then a bigger intermediate result is created that leads to more expensive operations further in the query plan. This could be avoided if we would choose to crack A_2 and pay an extra cost while cracking. Even if the overall cost is the same, an investment was made for the future to speed up queries referring to A_2. Spreading the "negative" effect of cracking on the first query over multiple users is another possibility.

However, there are even more parameters that one might want to consider, e.g., storage space limitation. In this case, our effort would be to minimize the number of attributes cracked so that columns do not have to be replicated.

Notice, though, that a cracker index and a cracker column is information that is completely disposable, i.e., they can be erased at any time without any overhead or persistency issues. This observation means that a potential self-organizing strategy could be possible where all cracked columns that exist in a database must not exceed a given maximum storage size S. Then, the available cracked columns at each time depend on query workload, i.e., attributes that are used often will be cracked to speedup future queries. S may vary over time, allowing the system to adapt to both query workload and resource restrictions.

Similarly with the above discussion, there is an optimization opportunity regarding the *order* of `crackers.rel_select` operators (that operate on the same relation) in a query plan. It is clear that we would like to have lower in the query plan the `crackers.rel_select` operator that creates the smaller intermediate result so that we can speed up the subsequent operators.

8.2 A histogram for free

Up to this point we have "used" the cracking data structures only in the context of the select operation. In the following two subsections we will see two examples of how a cracker index and a cracker column can be used for other operations too.

The cracker index contains information on actual value ranges of a given attribute. This is useful information that can be potentially used in more cases than the select operator. For example, the cracker index could play the role of a "histogram" and allow us to take decisions that will speed up query processing. With a cracker index it is known for a given range *how many* tuples in a column are in that range. This could be a valuable approximate information, e.g., in the case where the given range is not an exact match with what exists in the cracker index.

Traditionally histograms are maintained separately, which leads to an extra storage and operation cost. In addition, they are not always up to date with the current status in the database. On the contrary, with cracking the histogram-like information comes for free since no extra storage or operations are needed to maintain it. An interesting observation, is that here the histogram is a self-organized data structure as well; it creates and maintains information only for parts of the data that are interesting for the users. This is a powerful property for a structure that can potentially affect many aspects of query processing performance.

8.3 Cracker joins

As we discussed in the previous subsection, the cracking data structures can be useful for a number of operations. For example consider the join operator. The join operator has traditionally been the most expensive and challenging operator in database systems. Nowadays, a modern database system uses a large number of join algorithms to provide

efficient computation for various different cases. The cracking data structures can be used to make the join problem simpler.

For every attribute cracked, there exists a cracker column where tuples are clustered in various value ranges and these clusters are partially sorted/clustered. In this way, it is straightforward to device a sort merge-like algorithm that operates directly on the cracker columns. The advantage is that in this case there is no need to *invest* any time and resources in *preparing* for the join, e.g., sort the data or create a hash table. Instead, the join can directly start.

We envision that this can lead to significant speedup but also it could potentially simplify the code base of a modern DBMS by eliminating the need for all the different join implementations. In addition, in a multi-core or distributed environment, the natural properties of cracker columns will allow us to identify disjoint sets of the two join operands that can be computed at a different CPU or site.

8.4 More cracking operators

Until now we discussed open research directions where the the cracked query plans can be optimized or the already existing cracking data structures can be exploited. The future research agenda for cracking also includes the design of new operators that exploit the basic concepts of cracking.

The core principles of a cracking operation is that it physically reorganizes data such that the result tuples of an operator are cluster together. This happens while processing queries and it is based on queries which gives the property of self-organization. The challenge is to investigate if such an approach is possible to design and implement in more operators, other than the select operator. For example, is it possible to apply this strategy for joins or aggregate operations? This includes the creation of new algorithms that apply physical reorganization, investigating the impact on existing query plans and possible new operators or modifications necessary to materialize the cracking benefits. If this is achieved, it will bring substantial improvements on overall database performance since joins and aggregate operations are amongst the most demanding operations.

An important challenge in this area is to efficiently support multiple cracking operators on the same attribute in a query plan, e.g., is it possible to apply a cracking select operations and a cracking join operation on the same attribute for the same query?

8.5 Concurrency issues

Cracking means physical reorganization which naturally creates various concurrency considerations. For example, such a situation can occur when a query is using a cracker column to feed a next operator in its query plan while another query wants to start a cracking select on the same attribute, and thus on the same cracker column. One can think of various situations where concurrency issues arise. Supporting concurrent access on cracker columns is an open area with multiple opportunities.

For example, a first simple solution would be to restrict access at the operator level, i.e., do not allow a query q_1 to physically reorganize a cracker column c as long as c is needed to be read by another operator in another query q_2. In this case, q_1 might wait until q_2 releases c or fall back to the traditional non-cracking select operator on the original columns. More exciting solutions can be created by carefully identifying disjoint parts of the cracker column where queries can safely operate (i.e., read and/or physically reorganize) in parallel, e.g., the lock table can be defined in terms of the cracker index.

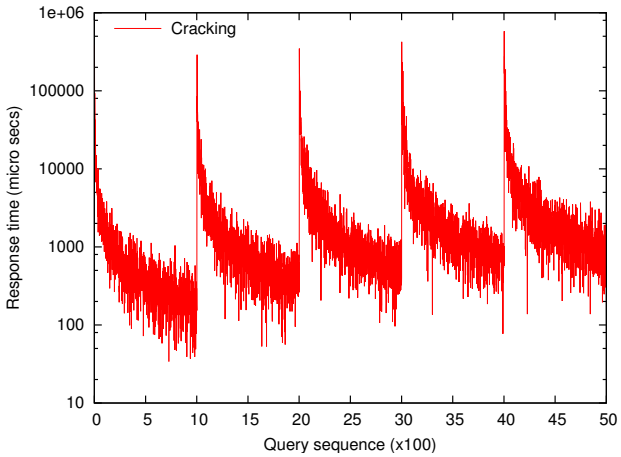


Figure 5: Behavior of cracking after batch insertions

8.6 Updates

Another important issue is that of updates. In practice, tables do not appear as fully instantiated before being queried; they grow over time. For the original columns, that maintain the insertion order, an update is translated in appending any new tuples. The critical issue is how the cracker columns are updated.

In MonetDB, we experiment with an architecture to defer updates using delta tables with pending insertions, deletions and simple updates. Propagation to the original column happens upon transaction commit. One approach for cracking is to keep the modifications separate until their table size surpasses a threshold. Pending update tables are merged into the query plan by the cracker optimizer to compensate for an out-of-date original column and cracker index.

In insert-only situations, all new tuples can be naïvely appended at the end of a cracker column and "forget" the cracker index. Rebuilding the index happens soon by queries interested in the specific attribute. As the cracker column is already partially sorted, future cracking operations become faster. Figure 5 shows an example of what happens under batch inserts. 10^3 queries are fired against a 10 million column. Then we perform a batch insertion of another 10 million tuples, which are appended to the cracker column. The cracker index is erased and we continue for another 10^3 queries, and so on. The figure clearly shows that cracking quickly reaches high performance after each insertion.

8.7 Cracking out of memory

The experiments presented in this paper are all in an in-memory environment. This is a reasonable choice to begin with to avoid seeing any swapping overheads. One of the next steps for cracking is to gracefully handle out of memory data, i.e., how to crack a column when the column does not fit in memory? Our initial experiments indicate that cracking remains a significantly more lightweight operation compared to sorting, or scanning a column. However, it is clear that there are more opportunities in this area.

One of the options is to logically split the column into multiple partitions in advance. Each partition fits in memory, for example the first partition contains the first x tuples, the second one contains the next x tuples and so on. Thus, partitioning is a zero cost operation since it only includes marking each partition with its start and end position in the cracker column. Then, each partition is cracked *separately*, namely there exist a separate cracker column cracker index pair for each partition. When processing queries a select operator will search all cracker indices for a given attribute and will physically reorganize each partition separately only if it is needed. Results can be merged within the query plan while the various partitions can be eventually merged or further split into smaller ones.

This is a very flexible approach that leaves the core cracking routines untouched and requires mainly some scheduling at a higher level.

8.8 When to crack

The cracking architecture has been described with a basic assumption/step, i.e., that physical reorganization happens for each incoming query. However, it is clear from our experimentation and experience that when cracking for very large query sequences, we may face such conditions where it may be of benefit not to further partition a cracker column. Administration and maintenance of the cracker column-cracker index pair becomes obviously more expensive when the number of pieces is increased.

There is a large number of opportunities here to explore in order to further optimize data access. For example, we envision that there is a cut-off point in terms of the minimum size for a piece in a cracker column, e.g., a disk page or a cache line leading to a more cache conscious architecture. Furthermore, following the self-organizing roots of cracking one may invest research on designing algorithms where pieces in a cracker column are not only partitioned when queries arrive but also are merged, to reduce maintenance and access cost on the cracker index. Here there is a need for the proper cost models to take the various decisions on-line while processing queries.

8.9 Apriori cracking

We already discussed that cracking targets environments where there is not enough time to prepare data (sort/cluster or create the proper index) and where there is not any workload knowledge. Notice though that there could be situations where there is indeed not enough idle time for a traditional index-based strategy but since cracking is a very lightweight operation there might be enough time to perform a number of cracking operations. One of the crucial observations from our experimentation is that already the second query after a cracking operation shows significant improvement in response time due to restricted data access. In this way, one may run a number of fake queries on an attribute as long as the system is idle so that future real queries can benefit from an already existing and partitioned cracker column.

8.10 Distributed cracking

Cracking is a natural way to partition data into "interesting" pieces based on query workload. Therefore cracking can be explored in a distributed or parallel setting by distributing pieces of the database to multiple nodes. For example, each node holds one or more pieces. The cracker index can be known by all nodes so that a query can be navigated to the appropriate node that holds the interesting data. One can explore more sophisticated architectures where the cracker index is also distributed to multiple nodes, to reduce maintenance cost (more expensive in a distributed setting mainly due to network traffic creation and delays). In this way, each node has a partial knowledge of the index and a partial knowledge of the data, thus there is a need for the proper distributed protocols/query plans to correctly and completely resolve a query. However, these are typical requirements/research challenges in any distributed setting. The key here is that the way data is distributed is done in a self-organized way based on query workload which we envision that can lead to a distributed system with less network overhead since interesting data for queries will be already together.

Distributed cracking is a wide open research area. It can be explored in the context of distributed or parallel databases. Furthermore, it can be explored in the context of P2P data management architectures where typically current research is focusing on a relaxed notion of the strict ACID database properties to allow for more flexible and fault tolerant architectures. Distributed cracking can exploit these new trends to potentially minimize the traffic creation caused by the self-organizing steps of cracking, i.e., data migration in a distributed environment.

8.11 Beyond the horizon

In this section, a large number of open research problems and potential opportunities for cracking have been identified. The fact that cracking is possible to design and implement in a modern DBMS opens a large area for experimentation. We expect that the core idea of cracking, i.e., react on the user request and perform the self-organizing step while processing queries, can be heavily explored within a DBMS and will lead to interesting results. Furthermore, cracking can naturally be explored in distributed and parallel environments.

9. RELATED RESEARCH

In this section, we briefly discuss related work. The recently proposed system C-Store [14] is related to cracking in many interesting ways. First, C-Store itself is also a brave departure from the usual paths in database architecture. C-Store is a column oriented database system, too. The main architecture novelty of C-Store is that each column/attribute is sorted and this order is propagated to the rest of the columns in the relation to achieve fast record reconstruction. In this way, multiple projections of the same relation can be maintained, up to one for each attribute. To handle the extra storage space required, compression is used, which is also shown to speedup query execution in column oriented databases [1, 16]. Thus, C-Store also physically reorganizes the data store. In many ways, a comparison between a cracking approach with the C-Store one would contain the arguments used in the comparison with the sort strategy. The main characteristic of cracking is its self-organization based on the query workload, doing just enough, touching enough of the data each time, and being able to change the focus to different parts of the data dynamically. These properties are not explored in the C-Store approach. On the other hand, taking the extreme route and

not doing just enough (i.e., sort the data) allows C-Store to exploit compression in a way that is probably hardly justified in a cracker setting. Another important issue is how updates are taken care of. While this is still an open issue for all systems mentioned, it seems that for cracking updates are a simpler problem since (a) there is less data to update, e.g., at most one extra column per attribute, while for C-Store there are multiple copies of a single column participating in different projections, and (b) maintaining a partially sorted column is cheaper than a fully sorted one.

Partial indexing has not received much attention as the prime scheme for organizing navigational access. Partial indices have been proposed to index only those portions of the database that are statistically likely to benefit query processing [13, 12]. They are auxiliary index structures equipped with range filters to ignore elements that should be referenced from the index. Simulations and partial implementations in Postgres have been undertaken. They have demonstrated the potential. The cracking index proposed here goes a step further. It creates a partial index over the storage space dynamically, rather then being told by a DBA. Moreover, it clusters the data, which improves processing significantly. And it has been implemented in its full breadth in an SQL engine.

Semantic query caching has been proposed as a scheme to optimize client-server interaction (in web and mobile settings) by keeping track of queries and their answers [7, 11]. Significant performance improvement can be obtained when a query can be answered through the cached results. Our cracker maps provide a description of the previous queries and the cracker index identifies the corresponding answers. As such, it can act as a server-side semantic query cache. The cracker map maintenance algorithm is expected to work better, because the clustering results are kept in persistent store. Even when we have to trim the cracker index, we can do so with a clear few on the cost incurred to redo a cracking over the pieces joined together.

Partitioned databases is another area where physical reorganization was used to speed up data access. Data is partitioned on multiple disks or cites based on some criteria, e.g., hash-based partitioning, range-based partitioning etc. Similarly with our discussion for indices in general, in partitioned databases we need to decide a partitioning scheme and then prepare the data. So the notion of self-organization while queries are processed does not exist. Partitioning has also been explored in the context of parallel and distributed databases. Furthermore, it is a concept heavily used in the current research on P2P databases, e.g., using Distributed Hash Tables data is split among multiple nodes using hashing. We envision that cracking techniques can be applied in such distributed environments too, since partitioning is a natural property that cracking brings into the database.

10. CONCLUSIONS

In this paper we introduced a database architecture based on *cracking*. We show that cracking is possible and simple to implement and that changes required in the modules of our experimentation platform MonetDB were straightforward to do. We clearly demonstrate that the resulting system can self-organize based on incoming user requests by significantly restricting data access. We show clear performance benefits and we have set a promising future research agenda.

11. REFERENCES

[1] D. J. Abadi, S. R. Madden, and M. C. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, 2006.

[2] A. Ailamaki, D. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB*, 2001.

[3] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, 2000.

[4] P. Boncz and M. Kersten. MIL Primitives For Querying a Fragmented World. *The VLDB Journal*, 8(2):101–119, Mar. 1999.

[5] P. Boncz, A. Wilschut, and M. Kersten. Flattening an Object Algebra to Provide Performance. In *ICDE*, 1998.

[6] S. Chaudhuri and G. Weikum. Rethinking database system architecture: Towards a self-tuning risc-style database system. In *VLDB*, 2000.

[7] P. Godfrey and J. Gryz. Semantic query caching for hetereogeneous databases. In *KRDB*, 1997.

[8] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. Qpipe: A simultaneously pipelined relational query engine. In *SIGMOD*, 2005.

[9] M. Kersten and S. Manegold. Cracking the database store. In *CIDR*, 2005.

[10] S. Manegold, P. A. Boncz, N. Nes, and M. L. Kersten. Cache-conscious radix-decluster projections. In *VLDB*, 2004.

[11] P.Godfrey and J. Gryz. Answering queries by semantic caches. In *DEXA*, 1999.

[12] P. Seshadri and A. N. Swami. Generalized partial indexes. In *ICDE*, 1995.

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

[14] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E.O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column oriented dbms. In *VLDB*, 2005.

[15] M. Zukowski, P. Boncz, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, 2003.

[16] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *ICDE*, 2006.