Self-organizing Tuple Reconstruction in Column-stores

Stratos Idreos
CWI Amsterdam
The Netherlands
idreos@cwi.nl

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

Stefan Manegold
CWI Amsterdam
The Netherlands
manegold@cwi.nl

ABSTRACT

Column-stores gained popularity as a promising physical design alternative. Each attribute of a relation is physically stored as a separate column allowing queries to load only the required attributes. The overhead incurred is on-the-fly tuple reconstruction for multi-attribute queries. Each tuple reconstruction is a join of two columns based on tuple IDs, making it a significant cost component. The ultimate physical design is to have multiple presorted copies of each base table such that tuples are already appropriately organized in multiple different orders across the various columns. This requires the ability to predict the workload, idle time to prepare, and infrequent updates.

In this paper, we propose a novel design, *partial sideways cracking*, that minimizes the tuple reconstruction cost in a self-organizing way. It achieves performance similar to using presorted data, but without requiring the heavy initial presorting step itself. Instead, it handles dynamic, unpredictable workloads with no idle time and frequent updates. Auxiliary dynamic data structures, called *cracker maps*, provide a direct mapping between pairs of attributes used together in queries for tuple reconstruction. A map is continuously physically reorganized as an integral part of query evaluation, providing faster and reduced data access for future queries. To enable flexible and self-organizing behavior in storage-limited environments, maps are materialized only partially as demanded by the workload. Each map is a collection of separate chunks that are individually reorganized, dropped or recreated as needed. We implemented partial sideways cracking in an open-source column-store. A detailed experimental analysis demonstrates that it brings significant performance benefits for multi-attribute queries.

Categories and Subject Descriptors: H.2 [DATABASE MANAGEMENT]: Physical Design - Systems

General Terms: Algorithms, Performance, Design

Keywords: Database Cracking, Self-organization

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.
Copyright 2009 ACM 978-1-60558-551-2/09/06 ...$5.00.

1. INTRODUCTION

A prime feature of column-stores is to provide improved performance over row-stores in the case that workloads require only a few attributes of wide tables at a time. Each relation R is physically stored as a set of columns; one column for each attribute of R. This way, a query needs to load only the required attributes from each relevant relation.

This happens at the expense of requiring explicit (partial) tuple reconstruction in case multiple attributes are required. Each tuple reconstruction is a join between two columns based on tuple IDs/positions and becomes a significant cost component in column-stores especially for multi-attribute queries [2, 6, 10]. Whenever possible, position-based join-matching and sequential data access are exploited. For each relation R_i in a query plan q, a column-store needs to perform at least $N_i - 1$ tuple reconstruction operations for R_i within q, given that N_i attributes of R_i participate in q.

Column-stores perform tuple reconstruction in two ways [2]. With *early* tuple reconstruction, the required attributes are glued together as early as possible, i.e., while the columns are loaded, leveraging N-ary processing to evaluate the query.

On the other hand, *late* tuple reconstruction exploits the column-store architecture to its maximum. During query processing, "reconstruction" merely refers to getting the attribute values of qualifying tuples from their base columns as late as possible, i.e., only once an attribute is required in the query plan. This approach allows the query engine to exploit CPU- and cache-optimized vector-like operator implementations throughout the whole query evaluation. N-ary tuples are formed only once the final result is delivered.

Like most modern column-stores [12, 4, 15], we focus on late reconstruction. Comparing early and late reconstruction, the educative analysis in [2] observes that the latter incurs the overhead of reconstructing a column more than once, in case it occurs more than once in a query. Furthermore, exploiting sequential access patterns during reconstruction is not always possible since many operators (joins, group by, order by etc.) are not *tuple order-preserving*.

The ultimate access pattern is to have multiple copies for each relation R, such that each copy is presorted on an other attribute in R. All tuple reconstructions of R attributes initiated by a restriction on an attribute A can be performed using the copy that is sorted on A. This way, the tuple reconstruction does not only exploit sequential access, but also benefits from focused access to only a small consecutive area in the base column (as defined by the restriction on A) rather than scattered access to the whole column. However, such a direction requires the ability to *predict* the workload

and the luxury of *idle time* to prepare the physical design. In addition, up to date there is no efficient way to maintain multiple sorted copies under updates in a column-store; thus it requires read-only or infrequently updated environments.

In this paper, we propose a self-organizing direction that achieves performance similar to using presorted data, but comes without the hefty initial price tag of presorting itself. Instead, it handles dynamic unpredictable workloads with frequent updates and with no need for idle time. Our approach exploits database cracking [7, 8, 9] that sets a promising direction towards continuous self-organization of data storage based on selections in incoming queries.

We introduce a novel design, *partial sideways* cracking, that provides a self-organizing behavior for both selections and tuple reconstructions. It gracefully handles any kind of complex multi-attribute query. It uses auxiliary self-organizing data structures to materialize mappings between pairs of attributes used together in queries for tuple reconstruction. Based on the workload, these *cracker maps* are continuously kept aligned by being physically reorganized, while processing queries, allowing the DBMS to handle tuple reconstruction using cache-friendly access patterns.

To enhance performance and adaptability, in particular in environments with storage restrictions, cracker maps are implemented as dynamic collections of separate chunks. This enables flexible storage management by adaptively maintaining only those chunks of a map that are required to process incoming queries. Chunks adapt individually to the query workload. Each chunk of a map is separately reorganized, dropped if extra storage space is needed, or recreated (entirely or in parts) if necessary.

We implemented partial sideways cracking on top of an open-source column-oriented DBMS, MonetDB[1] [15]. The paper presents an extensive experimental analysis using both synthetic workloads and the TPC-H benchmark. It clearly shows that partial sideways cracking brings a self-organizing behavior and significant benefits even in the presence of random workloads, storage restrictions and updates.

The remainder of this paper is organized as follows. Section 2 provides the necessary background. Then, to enhance readability and fully cover the research space, we present partial sideways cracking in two steps. Focusing on the tuple reconstruction problem and neglecting storage restrictions at first, Section 3 introduces the basic sideways cracking technique using fully materialized maps, accompanied with an extensive experimental analysis. Then, Section 4 extends the basic approach with flexible storage management using partial maps. Detailed experiments demonstrate the significant benefits over the initial full materialization approach. Then, Section 5 presents the benefits of sideways cracking with the TPC-H benchmark. Related work is discussed in Section 6, and Section 7 concludes the paper.

2. BACKGROUND

This section briefly presents the experimentation platform, MonetDB (v 5.4), and the basics of database cracking.

2.1 A Column-oriented DBMS

MonetDB is a full fledged column-store using late tuple reconstruction. Every relational table is represented as a collection of *Binary Association Tables* (*BATs*). Each BAT is a set of two columns. For a relation R of k attributes, there exist k BATs, each BAT storing the respective attribute as (`key`,`attr`) pairs. The system-generated `key` identifies the relational tuple that attribute value `attr` belongs to, i.e., all attribute values of a single tuple are assigned the same `key`. Key values form a dense ascending sequence representing the *position* of an attribute value in the column. Thus, for base BATs, the key column typically is a virtual non-materialized column. For each relational tuple t of R, all attributes of t are stored in the *same* position in their respective column representations. The position is determined by the insertion order of the tuples. This tuple-order *alignment* across all base columns allows the column-oriented system to perform tuple reconstructions efficiently in the presence of tuple order-preserving operators. Basically, the task boils down to a simple merge-like sequential scan over two columns, resulting in low data access costs through all levels of modern hierarchical memory systems. Let us go through some of the basic operators of MonetDB's two-column physical algebra.

Operator `select(A,v1,v2)` searches all (`key`,`attr`) pairs in base column A for attribute values between v_1 and v_2. For each qualifying attribute value, the `key` value (position) is included in the result. Since selections are mostly performed on base columns, the underlying implementation preserves the `key`-order also in the intermediate results.

Operator `join(j1,j2)` performs a join between `attr1` of `j1` and `attr2` of `j2`. The result contains the qualifying `(key1,key2)` pairs. In general, this operator can maintain the tuple order only for the outer join input.

Similarly, `groupby` and `orderby` operators cannot maintain tuple order for any of their inputs.

Operator `reconstruct(A,r)` returns all (`key`,`attr`) pairs of base column A at the positions specified by r. If r is the result of a tuple order-preserving operator, then iterating over r, it uses cache-friendly in-order positional lookups into A. Otherwise, it requires expensive random access patterns.

2.2 Selection-based Cracking

Let us now briefly recap *selection cracking* as introduced in [7]. The first time an attribute A is required by a query, a copy of column A is created, called the *cracker column* C_A of A. Each selection operator on A triggers a range-based physical reorganization of C_A based on the selection of the current query. Each cracker column, has a *cracker index* (AVL-tree) to maintain partitioning information. Future queries benefit from the physically clustered data and do not need to access the whole column. Cracking continues with *every* query. Thus, the system continuously refines its "knowledge" about how values are spread in C_A. Physical reorganization happens on C_A while the original column is left as is, i.e., tuples are ordered according to their insertion sequence. This order is exploited for tuple reconstruction.

The operator `crackers.select(A,v1,v2)` replaces the original select operator. First, it creates C_A if it does not exist. It searches the index of C_A for the area where v_1 and v_2 fall. If the bounds do not exist, i.e., no query used them in the past, then C_A is physically reorganized to cluster all qualifying tuples into a contiguous area.

The result is again a set of keys/positions. However, due to physical reorganization, cracker columns are no longer aligned with base columns and consequently the selection results are no longer ordered according to the tuple inser-

[1]Partial sideways cracking is part of the latest release of MonetDB, available via `http://monetdb.cwi.nl/`

tion sequence. For queries with multiple selections, we need to perform the intersection of individual selection results on unordered (key,*value*) sets. Without either sequential access or positional lookups, performance worsens significantly for tuple reconstruction. Thus, for conjunctive queries, [7] uses `crackers.select` only for the first selection, introducing the `crackers.rel_select` for all subsequent selections. It performs the tasks of `select` and `reconstruct` in one go.

The terminology "database cracking" reflects the fact that the database is conceptually *cracked* into pieces. It aims at unpredictable/dynamic environments. An extensive discussion on the pros and cons of cracking against traditional indices can be found in [7, 8]. In [8], cracking has been shown to also maintain its properties under high-volume updates.

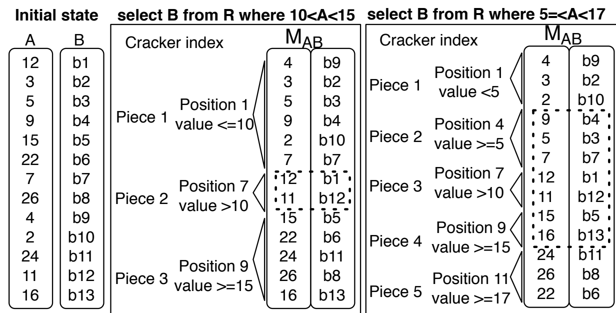


Figure 1: A simple example

3. SIDEWAYS CRACKING

In this section, we introduce the basic *sideways cracking* technique using fully materialized maps. To motivate and illustrate the effect of our choices, we build up the architecture starting with simple examples before continuing with more flexible and complex ones. The section closes with an experimental analysis of sideways cracking with full maps against the selection cracking and non-cracking approaches. The addition of adaptive storage management through partial sideways cracking is discussed and evaluated in Section 4. Section 5 shows the benefits on the TPC-H benchmark.

3.1 Basic Definitions

We define a *cracker map* M_{AB} as a two-column table over two attributes A and B of a relation R. Values of A are stored in the left column, while values of B are stored in the right column, called *head* and *tail*, respectively. Values of A and B in the same position of M_{AB} belong to the same relational tuple. All maps that have been created using A as head are collected in the *map set* S_A of $R.A$. Maps are created *on demand*, only. For example, when a query q needs access to attribute B based on a restriction on attribute A and M_{AB} does not exist, then q will create it by performing a scan over base columns A and B. For each cracker map M_{AB}, there is a *cracker index* (AVL-tree) that maintains information about how A values are distributed over M_{AB}.

Once a map M_{AB} is created by a query q, it is used to evaluate q and it stays alive to speed up data access in future queries that need to access B based on A. Each such query triggers physical reorganization (cracking) of M_{AB} based on the restriction applied to A. Reorganization happens in such a way, that all tuples with values of A that qualify the restriction, are in a *contiguous* area in M_{AB}. We use the two algorithms of [7] to physically reorganize maps by splitting a piece of a map into two or three new pieces.

We introduce `sideways.select(A,v1,v2,B)` as a new selection operator that returns tuples of attribute B of relation R based on a predicate on attribute A of R as follows:

(1) If there is no cracker map M_{AB}, then create one.
(2) Search the index of M_{AB} to find the contiguous area w of the pieces related to the restriction σ on A.
    If σ does not match existing piece boundaries,
    (3) Physically reorganize w to move false hits out of the contiguous area of qualifying tuples.
    (4) Update the cracker index of M_{AB} accordingly.
(5) Return a non-materialized view of the tail of w.

Example. Assume a relation $R(A, B)$ shown in Figure 1. The first query requests values of B where a restriction on A holds. The system creates map M_{AB} and cracks it into three pieces based on the selection predicate. Via cracking the qualifying B values are already clustered together *aligned* with the qualifying A values. Thus, no explicit join-like operation is needed for tuple reconstruction; the tail column of the middle piece forms the query's result. Then, a similar second query arrives. From the index, we derive that (1) the entire middle piece belongs to the result, and hence, (2) only Pieces 1 and 3 must be analyzed and further cracked. As more queries are being processed, the system "learns" — purely based on incoming queries — more about how data is clustered, and hence, can reduce data access.

3.2 Multi-projection Queries

Let us now discuss queries with multiple tuple reconstruction operations. We start with queries over a single selection and multiple projections. Then, Section 3.3 discusses queries with multiple selections.

The Problem: Non-aligned Cracker Maps. A single-selection query q that projects n attributes requires n maps, one for each attribute to be projected. These maps M_{Ax} belong to the same map set S_A. However, a naive use of the maps can lead to incorrect query results. Consider the example depicted in the upper part of Figure 2. The first query triggers the creation of M_{AB} and it physically reorganizes it based on $A<3$. Similarly, the second query triggers the creation of M_{AC} and cracks it according to $A<5$. Then, the third query needs both M_{AB} and M_{AC}. Refining both maps according to $A<4$ of the third query creates correct consecutive results for each map individually. However, since these maps had previously been cracked independently using different restrictions on A, the result tuples are not positionally aligned anymore, prohibiting efficient positional result tuple reconstruction. Maintaining the tuple identity explicitly by adding a `key` column to the maps is not an efficient solution, either. It increases the storage requirements and allows only expensive join-like tuple reconstruction requiring random access due to non-aligned maps.

The Solution: Adaptive Alignment. To overcome this problem, we extend the `sideways.select` operator with an *alignment step* that adaptively and on demand restores the alignment of all maps used in a query plan. The basic idea is to apply *all* physical reorganizations, due to selections on an attribute A, in the *same order* to all maps in S_A. Due to the deterministic behavior of the cracking algorithms [7], this approach ensures alignment of the respective maps.

Obviously, in an unpredictable environment with no idle system time, we want to invest in this extra work only if it

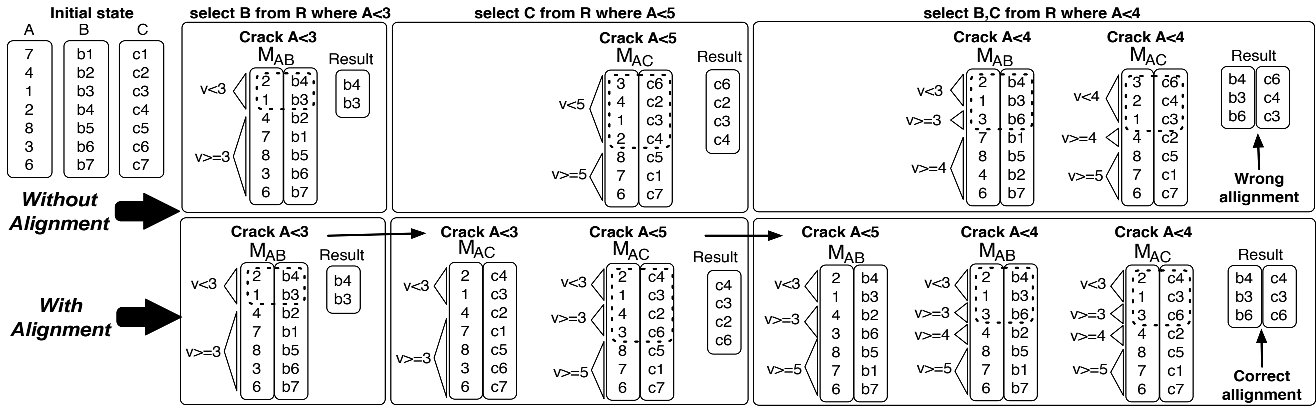


Figure 2: Multiple tuple reconstructions in multi-projection queries

pays-back, i.e., only once a map is required. In fact, performing alignment on-line is not an option. On-line alignment would mean that *every* time we crack a map, we also forward this cracking to the rest of the maps in its set. This is prohibitive for several reasons. First, in order to be able to align all maps in one go we need to actually materialize and maintain all possible maps of a set, even the ones that the actual workload does not require. Most importantly every query would have to touch all maps of a set, i.e., *all* attributes of the given relation. This immediately overshadows the benefit of using a column-store in touching only the relevant attributes every time. The overhead of having adaptive alignment is that each map M_{Ax} in a set S_A needs to materialize the head attribute A so that M_{Ax} can be cracked independently. We will remove this restriction with partial sideways cracking in the next section.

To achieve adaptive alignment, we introduce a *cracker tape* T_A for each set S_A, which logs (in order of their occurrence) all selections on attribute A that trigger cracking of any map in S_A. Each map M_{Ax} is equipped with a *cursor* pointing to the entry in T_A that represents the last crack on M_{Ax}. Given a tape T_A, a map M_{Ax} is aligned (synchronized) by successively forwarding its cursor towards the end of T_A and incrementally cracking M_{Ax} according to all selections it passes on its way. All maps whose cursors point to the same position in T_A, are physically aligned.

To ensure that alignment is performed on demand only, we integrate it into query processing. When a query q needs a map M, then and only then, q aligns M. We further extend the `sideways.select(A,v1,v2,B)` operator with three new steps that maintain and use the cracker tapes as follows:

(1) **If there is no T_A, then create an empty one.**
(2) If there is no cracker map M_{AB}, then create one.
(3) **Align M_{AB} using T_A.**
(4) Search the index of M_{AB} to find the contiguous area w of the pieces related to the restriction σ on A.
If σ does not match existing piece boundaries,
  (5) Physically reorganize w to move false hits out of the contiguous area of qualifying tuples.
  (6) Update the cracker index of M_{AB} accordingly.
  (7) **Append predicate $v_1 < A < v_2$ to T_A.**
(8) Return a non-materialized view of the tail of w.

For a query with one selection and k projections, the query plan contains k `sideways.select` operators, one for each projection attribute. For example, assume a query that selects on A and projects B and C. Then, one `sideways.select` operator will operate over M_{AB} and another over M_{AC}. With the maps aligned and holding the projection attributes in the tails, the result is readily available. The bottom part of Figure 2 demonstrates how queries are evaluated using aligned maps yielding the correctly aligned result.

Sideways cracking performs tuple reconstruction by efficiently maintaining aligned maps via cracking instead of using (random-access) position-based joins.

The alignment step follows the self-organizing nature of a cracking DBMS. Aligning a map M becomes less expensive the more queries use M, as incremental cracking successively reduces the size of pieces and hence the data that needs to be accessed. Moreover, the more frequently M is used, the fewer alignment steps are required per query to bring it up-to-date. Unused maps do not produce any processing costs.

3.3 Multi-selection Queries

The final step is to generalize sideways cracking for queries that select over multiple attributes. One approach is to create *wider* maps that include multiple attributes in different orderings. However, the many combinations, orderings and predicates in queries lead to huge storage and maintenance requirements. Furthermore, wider maps are not compatible with the cracker algorithms of [7] and the update techniques of [8]. Instead, we propose a solution that exploits aligned two-column maps in a way that enables efficient cache-friendly operations.

The Problem: Non-aligned Map Sets. Let us first consider conjunctive queries, e.g., the query of Figure 3. A query plan could use maps M_{AD}, M_{BD} and M_{CD}. These maps belong to *different* sets S_A, S_B and S_C, respectively. However, the alignment techniques presented before apply only to multiple maps within the *same* set. Keeping maps of different sets aligned is not possible at all, as each attribute requires/determines its own individual order for its maps. Thus, using the above map sets for the example query inherently yields non-aligned individual selection results requiring expensive operations for subsequent tuple reconstructions.

The Solution: Use a Single Aligned Set. The challenge for multi-selections is to find a solution that uses maps of only *one single* set, and thus can exploit their alignment. We postpone the discussion about how to choose this one set till later in this section. To sketch our approach using the query of Figure 3 as example, we arbitrarily assume that set S_A is chosen, i.e., we use maps M_{AB}, M_{AC} and M_{AD}.

Each map is first aligned to the most recent crack opera-

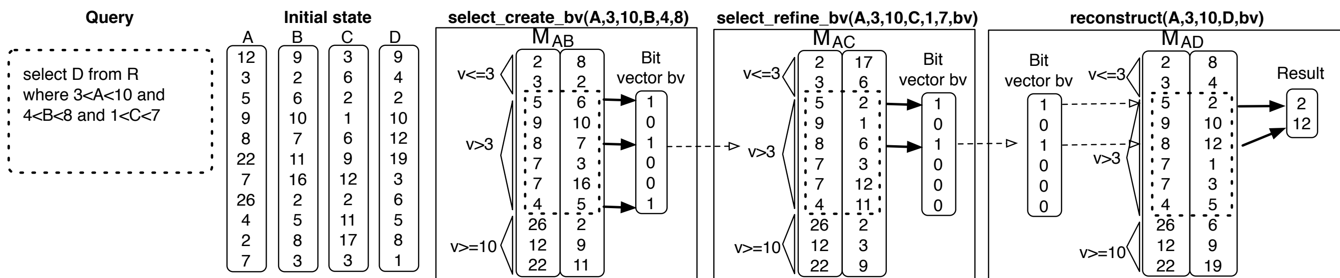


Figure 3: Multiple tuple reconstructions in multi-selection queries

tion on A and only then it is cracked given the current predicate on A. Given the conjunctive predicate, we know that we just created contiguous areas w_B, w_C and w_D aligned across the involved maps that contain all result candidates. These areas are aligned since all maps were *first* aligned and *then* cracked based on the *same* predicate. Thus, all areas have also the same size k. To filter out the "false candidates" that fulfill the predicate on A, but not all other predicates, we use *bit vector* processing (X100 [4] and the study of [2] also exploit bit-vectors for filtering multiple predicates). Using a single bit vector of size k, if a tuple fulfills the predicate on B, the respective bit is set, otherwise cleared. Successively iterating over the aligned result areas in the remaining maps (w_C in our example), the bits of tuples that do not fulfill the respective predicate are cleared. Finally, the bit vector indicates which w_D tuples form the result. An example in Figure 3 illustrates the details using the following three new operators.

`sideways.select_create_bv(A,v1,v2,B,v3,v4)`

(1-7) Equal to `sideways.select` in Section 3.2.
(8) Create and return bit vector bv **for** w **with** $v_3 < B < v_4$.

`sideways.select_refine_bv(A,v1,v2,B,v3,v4,bv)`

(1-7) Equal to `sideways.select` in Section 3.2.
(8) Refine bit vector bv **with** $v_3 < B < v_4$ **and return** bv.

`sideways.reconstruct(A,v1,v2,B,bv)`

(1-7) Equal to `sideways.select` in Section 3.2.
(8) Create and return a result that contains the tail value of all tuples from w **in** M_{AB} **whose bit is set in** bv.

Given the alignment of the maps and the bit vector, only positional lookups and sequential access patterns are involved. In addition, by clustering and aligning relevant data via cracking, the system needs to analyze only a small portion of the involved columns (equal to the size of the bit vector) for selections and tuple reconstructions.

Map Set Choice: Self-organizing Histograms. The remaining issue is to determine the appropriate map set. Our approach is based on the core of the "cracking philosophy", i.e., in an unpredictable environment with no idle system time, always perform the minimum investment. Do just enough operations to boost the current query. Do not invest in the future unless the benefit is known, or there is the luxury of time and resources to do so. In this way, for a query q, a set S_A is chosen such that the restriction on A is the most *selective* in q, yielding a minimal bit vector size in order to load and analyze less data in this query plan.

The most selective restriction can be found using the cracker indices, for they maintain knowledge about how values are spread over a map. The size of the various pieces gives the *exact* number of tuples in a given value range. Effectively, we can view a cracker index as a self-organizing histogram. In order to estimate the result size of a selection over an attribute A, any available map in S_A can be used. In case of alternatives, the *most aligned* map is chosen by looking at the distance of its cursor to the last position of T_A. The bigger this distance, the less aligned a map is. A more aligned/cracked map can lead to a more accurate estimation. Using the cracker index of the chosen map M_{Ax}, we locate the contiguous area w that contains the result tuples. In case the predicate on A matches with the boundaries of existing pieces in M_{Ax}, the result size is equal to the size $|w|$ of w. Otherwise, we assume that w consists of n pieces $W_1, \ldots, W_n$, and derive $|w| = \sum_{i=1}^{n} |W_i|$ and $|w'| = \sum_{i=2}^{n-1} |W_i|$ as upper and lower bounds respectively. We can further tighten these bounds by estimating the qualifying tuples in W_1 and W_n, e.g., using interpolation.

Disjunctive Queries. Disjunctive queries are handled in a symmetrical way. This time the first selection creates a bit vector with size *equal* to the size of the map and not to the size of the cracked area w (as with conjunctions). The rest of the selections need to analyze the areas *outside* w for any *unmarked* tuples that might qualify and refine the bit vector accordingly. The choice of the map set is again symmetric; we choose a set based on the *least* selective attribute. In this way, the areas that need to be analyzed outside the cracked area are as small as possible.

3.4 Complex Queries

Until now we studied multi-selections/projections queries. The rest of the operators are not affected by the physical reorganization step of cracking as no other operator, other than tuple reconstruction, depends on tuple insertion order. Thus, joins, aggregations, groupings etc. are all performed efficiently using the original column-store operators (e.g., see our experimental analysis). Potentially, many operators can exploit the clustering information in the maps, e.g., a `max` can consider only the last piece of a map or a `join` can be performed in a partitioned like way exploiting disjoint ranges in the input maps. We leave such directions for future work consideration as they go beyond the scope of this paper.

3.5 Updates

Update algorithms for a cracking DBMS have been proposed and analyzed in detail in [8]. An update is not applied immediately. Instead, it remains as a *pending* update and it is applied only when a query needs the relevant data assisting the self-organizing behavior. This way, updates are applied *while* processing queries and affect only those tuples relevant to the query at hand. For each cracker column,

there exist a pending insertions and a pending deletions column. An update is merely translated into a deletion and an insertion. Updates are applied/merged in a cracker column without destroying the knowledge of its cracker index which offers continual reduced data access after an update.

Sideways cracking is compatible with [8] as follows. Each map M_{AB} has a pending insertions table holding (A,B) pairs. Insertions are handled independently and on demand for each map using the Ripple algorithm [8]. The extension is that the first time an insertion is applied on a map of set S_A, it is also logged in tape T_A so that the rest of the S_A maps can apply the insertions in the correct order during alignment. For deletions we only need one pending deletions column for each set S_A as we only need (A,`key`) pairs to identify a deletion. Since maps do not contain the tuple `keys`, as cracker columns do, we maintain a map $M_{A\texttt{key}}$ for each set S_A. This map, when aligned and combined with the pending deletions column, gives the positions of the relevant deletes for the current query in the currently aligned maps. The Ripple algorithm [8] is used to move deletes out of the result area of the maps used in a plan.

3.6 Experimental Analysis

In this section, we present a detailed experimental analysis. We compare our implementation of selection and sideways cracking on top of MonetDB, against the latest non-cracking version of MonetDB and against MonetDB on presorted data. We use a 2.4 GHz AMD Athlon 64 processor equipped with 2 GB RAM. The operating system is Fedora Core 8 (Linux 2.6.23). Unless mentioned otherwise, all experiments use a relational table of 9 attributes (A_1 to A_9), each containing 10^7 randomly distributed integers in $[1, 10^7]$.

Exp1: Varying Tuple Reconstructions. The first experiment demonstrates the behavior in query plans with one selection, but with multiple tuple reconstructions:

(q_1) select $max(A2)$, $max(A3)$ ... from R where $v_1 < A1 < v_2$

We test with queries with 2 to 8 attributes in the select clause. For each case we run 100 queries requesting random ranges of 20% of the tuples. Figure 4(a) shows the results for the 100*th* query (full query sequence behavior is shown in next experiments). The structures in both cracking approaches have been reorganized by the previous 99 queries.

For all systems, increasing the number of tuple reconstructions increases the overall cost while presorted MonetDB and sideways cracking significantly outperform the others.

Presorted			Sid. Cracking			Sel. Cracking			MonetDB		
Tot	*TR*	*Sel*	*Tot*	*TR*	*Sel*	*Tot*	*TR*	*Sel*	*Tot*	*TR*	*Sel*
43	0.2	0.02	47	0.4	0.5	771	725	0.3	483	211	229

The above table breakes down the cost (in milli secs) for the case of 8 tuple reconstructions. It shows the contribution of tuple reconstruction (*TR*) and selection (*Sel*) to the total cost (*Tot*). For presorted data, the table is already sorted on the selection attribute. Naturally, selections happen very fast (using binary search). Tuple reconstructions are also extremely fast since the projection attributes are already aligned with the selection result, given that only tuple order-preserving operators are involved in these queries (we show more complex examples later on). Sideways cracking achieves similar performance to presorted data by continuously aligning and physically clustering relevant data together both for selections and for tuple reconstructions.

On the contrary, selection cracking improves over MonetDB significantly on selections but suffers from tuple reconstruction costs. With MonetDB, the `select` operator is order-preserving, hence, tuple reconstruction is performed using in-order positional `key`-lookups into the projection attribute's base column. The resulting sequential access pattern is very cache-friendly ensuring that each page or cache-line is loaded at most once. On the contrary, with selection cracking, the result of `crackers.select` is no longer aligned with the base columns due to physical reorganization. Consequently, the tuple reconstruction is performed using *randomly ordered* positional `key`-lookups into the base column. Lacking both spatial and temporal locality, the random access pattern causes significantly more cache-/page-misses, making tuple reconstruction more expensive.

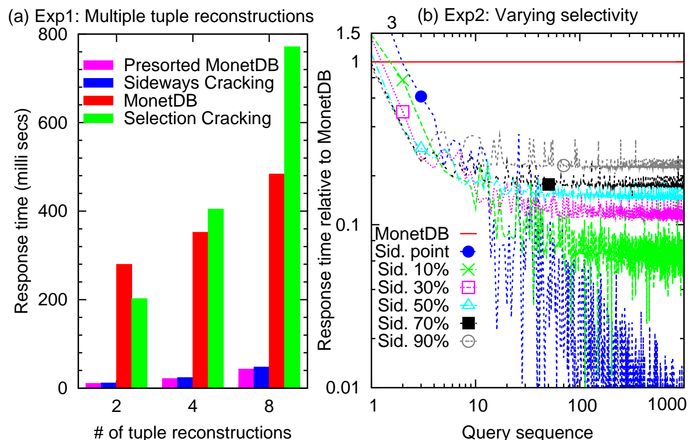
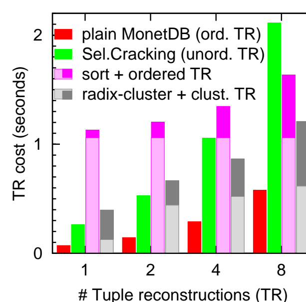


Figure 4: Improving tuple reconstruction

Exp2: Varying Selectivity. We repeat the previous experiment for 2 tuple reconstructions, but this time we vary selectivity factors from point queries up to 90% selectivity. We run 10^3 queries selecting randomly located ranges/points. Figure 4(b) shows the response time relative to the performance of non-cracking MonetDB (per selectivity factor). Sideways cracking significantly outperforms MonetDB on all selectivity ranges. In general, the first query is slightly slower for sideways cracking since the appropriate maps are created. After only a few queries, the maps are physically reorganized to an extent that significantly fewer non-qualifying tuples have to be accessed, allowing sideways cracking to quickly outperform MonetDB. As queries become less selective, sideways cracking outperforms MonetDB sooner (in terms of processed queries). With less selective queries, more tuples have to be reconstructed producing higher costs for the non-cracking column-store. For the same reason, with more selective queries, the relative benefit of cracking is smaller for the initial queries as the (smaller) benefits in tuple reconstruction are being partially shadowed by the higher cracking costs of the first queries.



Exp3: Reordering. A natural direction to improve tuple reconstruction with selection cracking is to reorder unordered intermediate results. For the graph on the left, we use both sorting and a cache-friendly radix-clustering algorithm [10] that restricts random access during reconstruction to the cache. This achieves similar

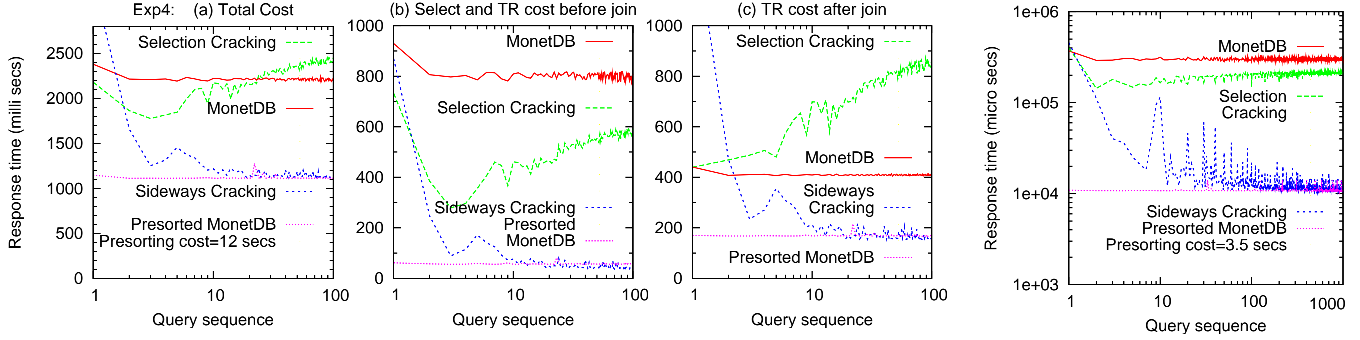


Figure 5: Join queries with multiple selections and tuple reconstructions (TR) **Figure 6: Skewed workload**

reconstruction performance as purely sequential access at a lower investment than sorting. We see that the investment in clustering (sorting) pays off with 4 (8) or more projections. In this way, reordering intermediate results pays off when multiple projections share a single intermediate result. However, it is not beneficial with only a few projections or with multiple selections, where individual intermediate results prohibit the sharing of reordering investments. Also, as seen in Figure 4 presorted MonetDB and sideways cracking significantly outperform plain MonetDB even with just a few tuple reconstructions by having columns already aligned.

Exp4: Join Queries. We proceed with join queries that both select and project over multiple attributes. Two tables of 7 attributes and the following query are used.

(q_2) select $max(R1)$,$max(R2)$,$max(S1)$,$max(S2)$ from R,S
where $v_1 < R3 < v_2$ and $v_3 < R4 < v_4$ and $v_5 < R5 < v_6$
and $k_1 < S3 < k_2$ and $k_3 < S4 < k_4$ and $k_5 < S5 < k_6$
and $R7 = S7$

We run 10^2 randomly created queries, with fixed selectivity factors of 50%, 30% and 20% for the conjunctions of each table. Figure 5(a) shows the results. All systems evaluate the queries starting from the most selective predicate. Sideways cracking and presorted MonetDB achieve similar performance and significantly outperform the other approaches. Presorting of course has a high preparation cost (12 secs). Figure 5(b) shows separately the selections and tuple reconstruction costs before the join. For a fair comparison both MonetDB and MonetDB on presorted data use the faster `rel_select` operator of selection cracking for the tuple reconstructions prior to the join. Figure 5(c) also shows separately the tuple reconstruction costs after the join.

For both cost components, presorted data and sideways cracking significantly improve the column-store performance by providing both very fast selections due to value ranges knowledge, but also very fast tuple reconstructions due to more efficient access patterns. Qualifying data is already aligned and clustered in smaller areas, i.e., equal to the result size of the most selective predicate. On the contrary, MonetDB and selection cracking have to reconstruct the required attributes from the full base columns.

Selection cracking loses its advantage in selections due to random access patterns in tuple reconstruction, even before the join. Also, the order of tuple-`keys` in cracker columns becomes more distorted, as more queries contribute to cracking, resulting in further increasing reconstruction costs.

For all systems, tuple order in the intermediate result of the inner input is lost after the join. Thus, all systems per-

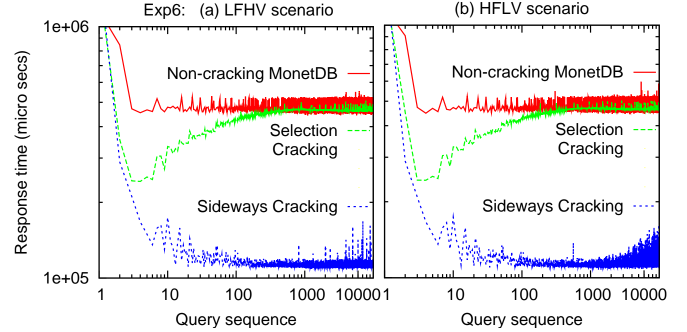


Figure 7: Effect of updates

form tuple reconstruction for this table using random access patterns. However, plain MonetDB and selection cracking need to prompt the base columns as the tuples to be reconstructed for each attribute A are scattered through the whole base column of A. On the other hand, MonetDB on presorted data and sideways cracking have the qualifying data clustered in a smaller area in each column and thus can improve significantly by loading and analyzing less data.

Naturally, more selections or more tuple reconstructions (either before or after the join), further increase the benefits of presorted data and sideways cracking.

Exp5: Skewed Workload. Sideways cracking gracefully adapts to the workload and exploits any opportunity to improve performance. To illustrate this powerful property, assume a 3 attribute table and the following query type.

(q_3) select $max(B)$,$max(C)$ from R where $v_1 < A < v_2$

We choose v_1 and v_2 such that 9/10 queries request a random range from the first half of the attribute value domain, while only 1/10 queries request a random range from the rest of the domain. All queries request 20% of the tuples. Figure 6 shows that sideways cracking achieves high performance similar to presorted data by always using cache-friendly patterns to reconstruct tuples. Skew affects the "learning" rate of sideways cracking, making it reach the best performance quickly for the hot-set. Since most of the queries focus on a restricted area of the maps, cracking can analyze this area faster (in terms of query requests) and break it down to smaller pieces (which are faster to process). With queries outside the hot-set, we have to analyze a larger area (though not the whole column). This is why we see the peaks roughly every 10 queries. However, as more queries

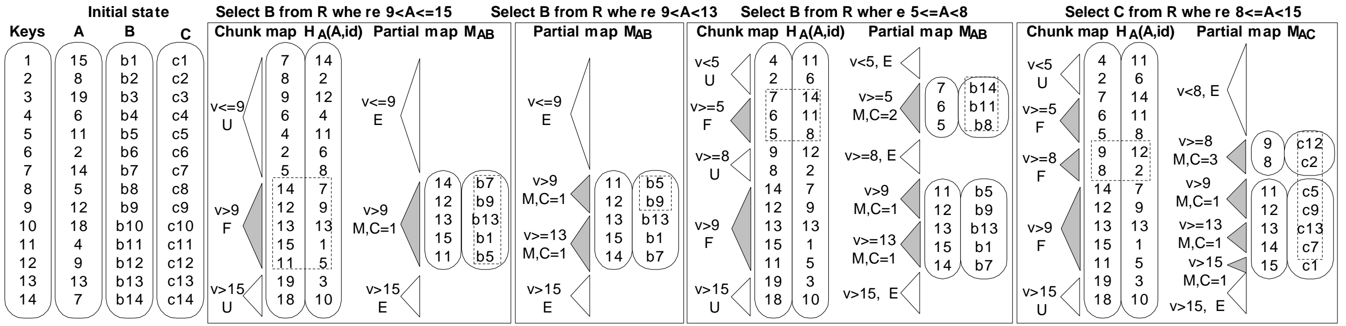


Figure 8: Using partial maps (U=Unfetched, F=Fetched, E=Empty, M=Materialized, C=ChunkID)

touch the non-hot area, in a self-organizing way, sideways cracking improves performance also for the non-hot set.

Exp6: Updates. Two scenarios are considered for updates, (a) the high frequency low volume scenario (HFLV); every 10 queries we get 10 random updates and (b) the low frequency high volume scenario (LFHV); every 10^3 queries we get 10^3 random updates. Random q_3 queries are used. Figure 7 shows that sideways cracking maintains high performance and a self-organizing behavior through the whole sequence of queries and updates demonstrating similar performance as in [8]. We do not run on presorted data, here, since to the best of our knowledge there is no efficient way to maintain multiple sorted copies under frequent updates in column-stores [6]. This is an open research problem. Obviously, resorting all copies with every update is prohibitive.

4. PARTIAL SIDEWAYS CRACKING

The previous section demonstrated that sideways cracking enables a column-store to efficiently handle multi-attribute queries. It achieves similar performance to presorted data but without the heavy initial cost and the restrictions on updates and workload prediction. So far, we assumed that no storage restrictions apply. As any other indexing or caching mechanism, sideways cracking imposes a storage overhead. This section addresses this issue via partial sideways cracking. An extensive experimental analysis shows that it significantly improves performance under storage restrictions and enables efficient workload adaptation by partial alignment.

4.1 Partial Maps

The motivation for partial maps comes from a divide and conquer approach. The main concepts are the following. (1) Maps are only *partially* materialized driven by the workload. (2) A map consists of *several chunks*. (3) Each chunk is a *separate* two-column table and (4) contains a given value *range* of the head attribute of this map. (5) Each chunk is treated *independently*, i.e., it is cracked separately and it has its own tape. Figure 8 illustrates a simplified example.

Basic Definitions. A map set S_A of an attribute A consists of (a) a collection of partial maps and (b) a *chunk map* H_A. H_A contains A values along with the respective tuple `key`. Its role is to provide the partial maps of S_A with any missing chunks when necessary. Each partial and chunk map has an AVL-tree based index to maintain partitioning information. Different maps in the same set do not necessarily hold chunks for the same value ranges. A partial map is created when a query needs it for the first time. The chunk map for a set S, is created along with the creation of the first chunk of the first partial map in S.

An area w of a chunk map is defined as *fetched* if at least one partial map has fetched all tuples of w to create a new chunk. Otherwise, w is called *unfetched*. Similarly, an area c of a partial map is defined as *materialized* if this map has created a chunk for c. Otherwise, c is called *empty*. Figure 8 shows some simple examples.

For each fetched area w, the index of a chunk map maintains (i) a list of references to w, i.e., the IDs of the partial maps that currently hold a chunk created by fetching w, and (ii) a tape where all the cracks that happen on the chunks created by w are logged. If all these chunks are dropped (discussed below under "Storage Management"), then w is marked again as unfetched and its tape is removed.

Creating Chunks. New chunks for a map M_{Ax} are created on demand, i.e., each time a query q needs tuples from an empty area c of M_{Ax}. The area c corresponds to an area w of H_A. We distinguish two cases depending on whether w is fetched or not. Firstly, if w is unfetched, then currently no other map in S_A holds any chunks created from w. In this case, depending on the value range that q requires, we either make a new chunk using all tuples of w or crack w in smaller areas to materialize only the relevant area (see examples in Figure 8). Secondly, in case w is already marked as fetched, it must not be cracked further, as this might lead to incorrect alignment as described in Section 3.2. For example, if multiple maps are used by a single query q that requires chunks created from an area w, then these chunks will not be aligned if created by differently cracked instances of w. Hence, a new chunk is created using all tuples in w. To actually create a new chunk for a map M_{AB}, we use the `keys` stored in w to get the B values from B's base column.

Storage Management. A partial map is an *auxiliary* data structure, i.e., without loss of primary information, any chunk of any map can be dropped at any time, if storage space is needed, e.g., for new chunks. In the current implementation, chunks are dropped based on how often queries access them. After a chunk is dropped, it can be recreated at any time, as a whole or only in parts, if the query workload requires it. This is a completely self-organizing behavior. Assuming there is no idle time in between, no available storage, and no way to predict the future workload, this approach assures that the maximum available storage space is exploited, and that the system always keeps the chunks that are really necessary for the workload hot-set.

Before creating a new chunk, the system checks if there is sufficient storage available. If not, enough chunks are

dropped to make room for the new one. Dropping a chunk c involves operations to update the corresponding cracker index I. To assist the learning behavior lazy deletion is used, i.e., all nodes of I that refer to c are not removed but merely marked as deleted and hence can be reused when c (or parts of it) is recreated in the future.

Dropping the Head Column. The storage overhead is further reduced by dropping the head column of actively used chunks, at the expense of loosing the ability of further cracking these chunks. We consider two opportunities.

First, we drop the head column of chunks that have been cracked to an extend that each piece fits into the CPU cache. In case a future query requires further cracking of such pieces, it is cheap to sort a piece within the CPU cache. This action is then logged in the tape to ensure future alignment with the corresponding chunks of other maps.

Second, we drop the head column of chunks that have not been cracked recently as queries use their pieces "as is". Once we need to crack such a chunk c in the future, we only need to recover and align the head as follows. If a chunk c' (of the same area as c) in an other map still holds the head and is less or equally aligned to the state that c was when it lost its head, then the head is recovered from c'. Otherwise, the head is taken from the chunk map. The first case is cheaper as less effort is needed to align the head.

Chunk-wise Processing. As seen in Section 3.3, each sideways cracking operator O first collects (using proper cracking and alignment) in a contiguous area w of the used map M all qualifying tuples based on the head attribute of M. Then, it runs the specific operator O over the area w, e.g., create or refine a bit vector based on a conjunctive predicate, perform a projection, etc. With partial sideways cracking we have the opportunity to improve access patterns even more by allowing chunk-wise processing. Each operator O handles one chunk c of a map at a time, i.e., load c, create c if the corresponding area is empty, crack or align c if necessary and finally run O over c.

Partial Alignment. Partial maps allow significant optimizations during alignment. The key observation is that we do not always need to perform *full* alignment, i.e., align a chunk c up to the last entry of its tape. If c is not going to be cracked, it only needs to be aligned with respect to the corresponding chunks of the other maps of the same map set used in this query, i.e., up to the *maximum* cursor of these chunks. We call this *partial* alignment. When performing any operator over a map, only the *boundary* chunks might need to be cracked, i.e., the first chunk where the lower bound falls and the last chunk where the upper bound falls; *all* other chunks in between benefit from partial alignment.

Even for boundary chunks, partial sideways cracking can in many cases avoid full alignment as follows. Assume a chunk c as a candidate for cracking based on a bound b. First, we perform partial alignment on c and *monitor* the alignment bounds. If b matches one of the past cracks, then cracking and thus full alignment of c is not necessary. Otherwise, full alignment starts. However, even then, if b is found on the way, then alignment stops and c is not cracked.

Updates. The Ripple algorithm of [8] is already designed to update only the parts (value ranges) necessary for the running query. Thus, the update strategy and performance in partial maps remains the same as in Section 3. Chunk maps are treated in the same way, i.e., a chunk map H_x has its own pending updates structures and areas on H_x are updated only on demand. Thus, before making a new chunk from an unfetched area w, w is updated if necessary. Naturally, updates applied in a chunk map are also removed from the pending updates of all partial maps in this set.

4.2 Experimental Analysis

We proceed with a detailed assessment of our partial sideways cracking implementation on top of MonetDB. Using the same platform as in Section 3.6, we show that partial maps bring a significant improvement and a self-organizing behavior under storage restrictions and during alignment.

For storage management with full maps, we use the same approach as for partial maps, i.e., existing maps are only dropped if there is not sufficient storage for newly requested maps. We always drop the least frequently accessed map(s).

For the experiments throughout this section we use a relation with 11 attributes containing 10^6 tuples and 5 multi-attribute queries (Q_i, $i \in \{1, \ldots, 5\}$) of the following form.

(Q_i) select C_i from R where $v_1 < A < v_2$ and $v_3 < B_i < v_4$

All queries use the same A attribute but different B_i and C_i attributes, i.e., each query requires two different maps. A fully materialized map needs 10^6 tuples. All queries select random ranges of S tuples. We run 10^3 queries in batches of 100 per type, i.e., first 100 Q_1 queries, then 100 Q_2 queries, and so on, while enforcing a storage threshold of T tuples.

Handling Storage Restrictions. We use $S = 10^4$ and three different storage restrictions: (a) no limit (in practice, all 10 maps used by the 5 queries fit within $T = 10^7$); (b) $T = 6.5{*}10^6$, i.e., slightly more than required to keep 6 full maps concurrently; (c) $T = 2{*}10^6$, i.e., only 2 full maps can co-exist (the minimum to run one query using full maps).

Figures 9(a), (b) and (c) show the per query cost for each case separately. In all three plots, full maps show the same pattern. Once every 100 queries, very high peaks (i.e., per query costs) severely disturb the otherwise good performance. These peaks relate to the workload changes between query batches. The first 5 peaks reflect the costs of initially creating the cracker maps for each batch, plus aligning them with the cracks of the preceding batch. Requiring no alignment, the first peak is smaller than the next 4. As of query 500, the batch cycle is repeated. With unlimited storage, all created maps are still available for reuse, requiring only alignment but no recreation with peaks 5–10 in Figure 9(a). With limited storage, the first maps had to be dropped to make room for later batches, requiring complete recreation of the maps once the cycle restarts. Figure 9(d) shows full maps allocating storage in blocks of two full maps per batch.

In contrast, partial maps do not penalize single queries, but distribute the map creation and alignment costs evenly across all queries, using chuck-wise granularity to more accurately adapt to changing workloads. Due to slightly increased costs for managing individual chunks instead of monolithic maps, partial maps do not quite reach the minimal per query cost of full maps. However, this investment results in a much smoother and more predictable performance behavior due to more flexible storage management (cf., Fig. 9(d)). Additionally, partial maps reduce the overall costs of query sequences (cf., Fig. 11 & 12 and discussion below).

Adaptation. Partial maps can fully exploit the workload characteristics to improve performance. To demonstrate this, we re-run the basic experiment with two variations: (a) we keep the uniform workload, but increase the selectivity using $S = 10^3$; (b) we keep $S = 10^4$, but use a skewed

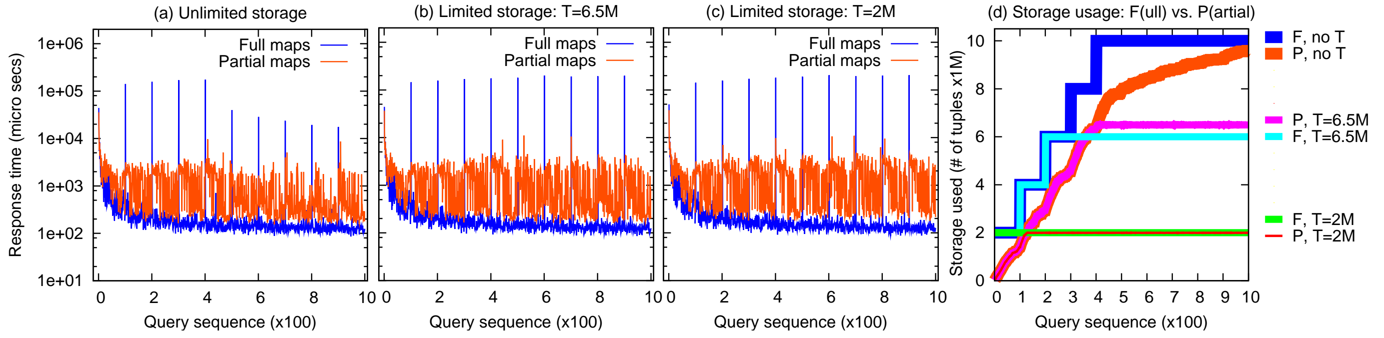


Figure 9: Efficient handling of storage restrictions with partial maps (S=10K)

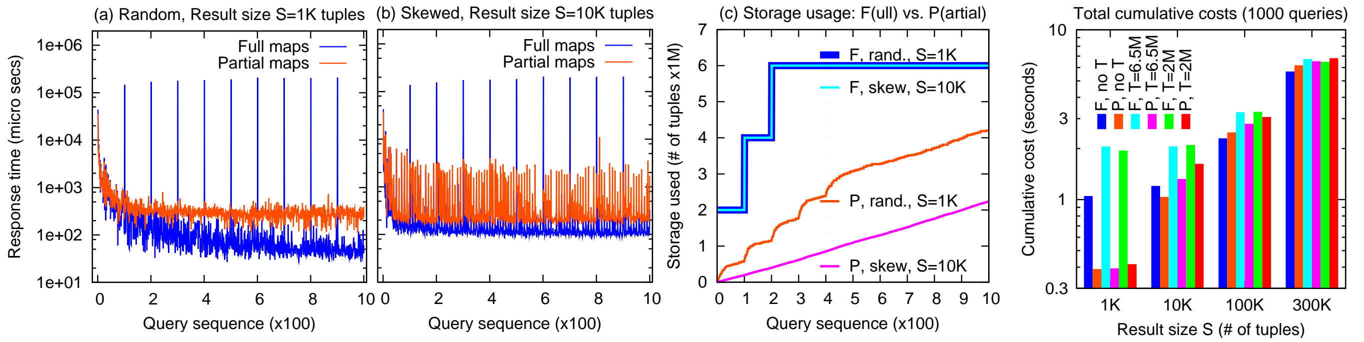
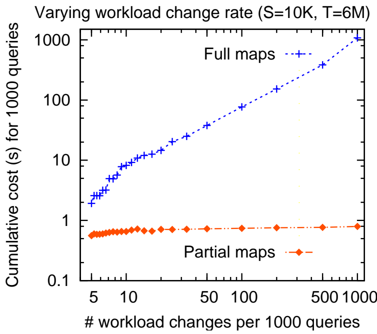


Figure 10: Efficient adaptation to the workload with partial maps (T=6.5M) **Figure 11: No overhead**



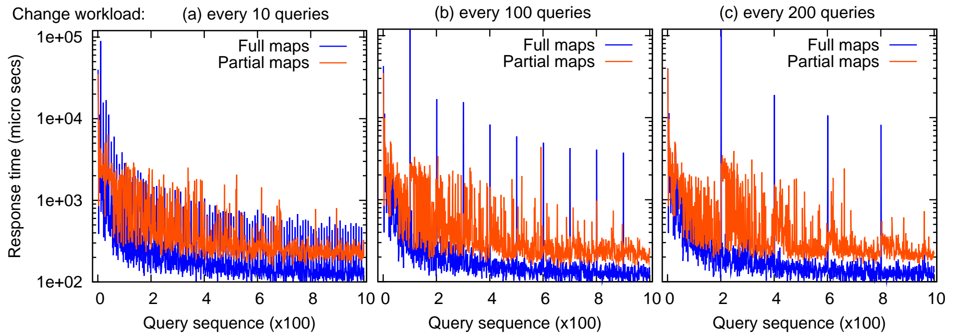


Figure 12: Total costs **Figure 13: Improving alignment with partial maps** (S=10K, T=unlimited)

workload. To simulate the skew, we force 9/10 queries to request random ranges from only 20% of the tuples while the remaining queries request ranges from the rest of the domain. Both runs use a storage threshold of $T = 6.5*10^6$.

Figures 10(a) and (b) depict the results. Compared to the previous experiment, the workload is now focused on specific data parts, either by more selective queries or by skew. In both cases, partial sideways cracking shows a self-organizing and normalized behavior without penalizing single queries as full maps do. Being restricted to handling complete maps (holding mostly unused data), full maps cannot take advantage of the workload characteristics and suffer from lack of storage. Figure 10(c) illustrates that full maps demand more storage and thus quickly hit the threshold. In contrast, partial maps exploit the available storage more efficiently and more effectively by materializing only the required chunks.

No Overhead in Query Sequence Cost. So far, we demonstrated that partial maps provide a more normalized per query performance compared to full maps. In addition, Figure 11 shows that these benefits come for free. It depicts the total cost to process *all* queries in the basic experiment by varying both the selectivity and the storage threshold. With 30% selectivity ($S = 3*10^5$), both approaches have similar total cost while with more selective queries partial maps significantly outperform full maps. This behavior combined with the more normalized per query performance gives a strong advantage to partial maps. The next experiment demonstrates that the advantage of partial maps over full maps increases with more frequent workload changes.

Adapting to Frequently Changing Workloads. In all previous experiments we assume a fixed rate of changing workload, i.e., every 100 queries. Here we study the effect of

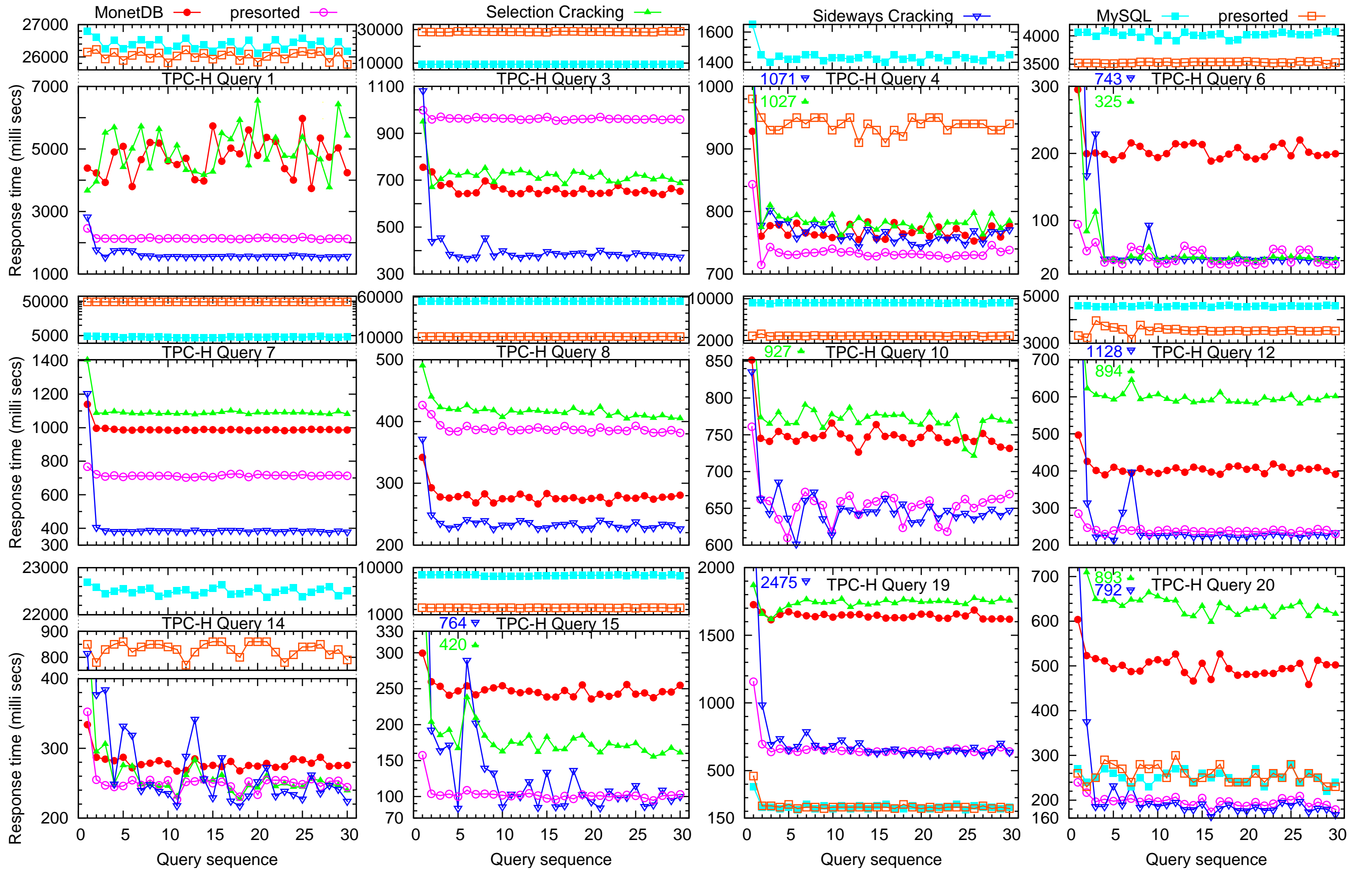


Figure 14: TPC-H results ("presorted" times exclude presorting costs; Q4,8,10: 3 min.; Q1,6,7,12,14,15,19,20: 11 min.; Q3: 14 min.)

varying this parameter. We run the basic experiment with fixed $S = 10^4$ and $T = 6{*}10^6$ but for various different rates of changing the workload. Figure 12 shows the total cost to process all queries for each case. The performance of full maps faces a significant degradation as the workload changes more often, causing maps to be dropped and recreated more frequently. In contrast, due to flexible and adaptive chunk management, partial maps offer a stable high performance that decreases hardly with more frequent workload changes.

Alignment Improvements. Let us now demonstrate the benefits of partial maps during alignment. We run the basic experiment for $S = 10^4$. To concentrate purely on the alignment cost we use only two types of queries and assume no storage restrictions. Figure 13 shows results for changing the workload every 10, 100 or 200 queries. As we decrease the rate of changing workloads, the peaks for full maps become less frequent, but higher. These peaks represent the alignment cost. Each time the workload changes, the maps used by the new batch of queries have to be aligned with the cracks of the previous batch; the longer the batch, the more cracks, the higher the alignment costs. Partial maps do not suffer from the alignment cost. Being able to align chunks only partially, and only those required for the current query, partial maps avoid penalizing single queries, bringing a smoother behavior to the whole query sequence. Furthermore, notice that as more queries are processed, partial maps gain more information to continuously increase alignment performance, assisting the self-organizing behavior.

5. TPC-H EXPERIMENTS

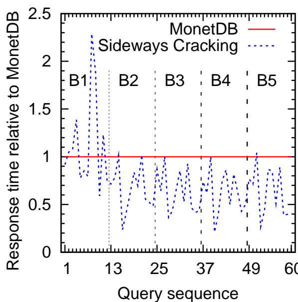
In this section, we evaluate our implementation in real-life scenarios using the TPC-H benchmark [14] (scale factor 1) on the same platform as in the previous experiments. We use the TPC-H queries that have at least one selection on a non-string attribute, i.e., Queries 1, 3, 4, 6, 7, 8, 10, 12, 14, 15, 19, & 20 (cf., [14]). String cracking and base table joins exploiting the already partitioned cracker maps are expected to yield significant improvements also for the remaining queries, but these are directions of future work and complementary to this paper. For each query, we created a sequence of 30 parameter variations using the random query generator of the TPC-H release. For experiments on presorted data, we created copies of all relevant tables such that for each query there is a copy primarily sorted on its selection column and (where applicable) sub-sorted on its group-by and/or order-by column. We use MySQL to show the effects of using presorted data on a row-store.

Figure 14 shows the costs for each query sequence. Sideways cracking achieves similar performance to presorted MonetDB (ignoring the presorting cost). Depending on the query, the presorting cost is 3 to 14 minutes, while as seen in Figure 14 the first sideways cracking query (in each query sequence) is between 0.75 to 3 seconds. In a self-organizing way, sideways cracking continuously improves performance without requiring the heavy initial step of presorting and workload knowledge. For most queries, it outperforms plain MonetDB as of the second run; for Queries 1 & 10, already

Q	$SiCr$	$PrMo$
1	64%	50%
3	44%	-46%
4	4%	6%
6	80%	83%
7	62%	28%
8	20%	-36%
10	12%	9%
12	41%	42%
14	19%	12%
15	62%	60%
19	61%	61%
20	67%	65%

the first run is faster. The table on the left summarizes the benefits of sideways cracking ($SiCr$) and presorted MonetDB ($PrMo$) over plain MonetDB on the tested TPC-H queries (Q). Having both efficient selections and tuple reconstructions, both sideways cracking and presorted MonetDB manage to significantly improve over plain MonetDB especially for queries with multiple tuple reconstructions on large tables, e.g., Queries 1, 6, 7, 15, 19, 20. Queries with multiple non tuple-order-preserving operators (group by, order by, joins) and subsequent tuple reconstructions yield significant gains by restricting tuple reconstructions to small column areas, e.g., Queries 1, 3, 7. Query 19 is an example where a significant amount of tuple reconstructions are needed as it contains a complex disjunctive where clause. The column-store has to reconstruct each attribute multiple times to apply the different predicates whereas the row-store processes the tables tuple-by-tuple. Sideways cracking minimizes significantly this overhead providing a comparable performance to the row-store.

In certain cases (e.g., Queries 3, 7, 8), cracking manages to even outperform presorted MonetDB. The TPC-H data comes already presorted on the keys of the `Order` table. Plain MonetDB (and MySQL) exploit the sorted keys especially during joins (most queries join on `Order` keys). Fully sorting on the selection attribute completely destroys this order, making the presorted case even slower than the original one (both with MonetDB and MySQL). With sideways cracking, though, the initial order is only partially changed, providing more efficient access patterns during joins.



Our final experiment features a mixed workload. We run 5 sequential batches (B1..B5) of 12 different TPC-H queries with varying parameters. The figure on the left shows the performance of sideways cracking relative to MonetDB. Already within the first batch (B1), sideways cracking outperforms MonetDB in many queries. This is because queries can reuse maps and partitioning information created by different queries over the same attributes. The high peak in the first batch comes from Query 12 that uses a map set not used by any other query. Naturally, after the first batch sideways cracking improves even more.

6. RELATED WORK

Self-organization has become an active research area, e.g., [3, 5, 11, 13]. Pioneering literature mainly focuses on predicting the future workload as a basis for an appropriate physical design. This is mainly an off-line task that works well in stable environments or with a delay in dynamic ones. In contrast, cracking instantly reacts to every query, refining the physical data organization accordingly without the need for a workload predictor, or lengthy reorganization delays.

The only other column-store that uses physical reorganization is C-Store [12, 1, 2]; each attribute *sort order* is propagated to the rest of the columns in a relation R, maintaining multiple projections over R. It targets read-only scenarios using the luxury of time to pre-sort the database completely and exhaustively. Cracking targets exactly the opposite environments with continuous and sudden workload shifts and updates [8]. Direct comparison with C-store was not possible as it does not provide a generic SQL interface. However, we believe that our experiments against MonetDB on presorted data give a fair comparison of sideways cracking against a presorted column-store.

A very interesting area is the opportunity to improve performance using compression. This naturally gives a boost in presorted data performance [1], while it is a promising research direction for database cracking too.

7. CONCLUSIONS

In this paper, we introduce partial sideways cracking, a key component in a self-organizing column-store based on physical reorganization in the critical path of query execution. It enables efficient processing of complex multi-attribute queries by minimizing the costs of late tuple reconstruction, achieving performance competitive with using presorted data, but requiring neither an expensive preparation step nor a priori workload knowledge. With its flexible and adaptive chunk-wise architecture it yields significant gains and a clear self-organizing behavior even under random workloads, storage restrictions, and updates.

Database cracking has only scratched the surface of this promising direction for self-organizing DBMSs. The research agenda includes calls for innovations on cracker-joins, compression, aggregation, distribution and partitioning, as well as optimization strategies, e.g., cache-conscious chunk size enforcement in partial sideways cracking. Furthermore, row-store cracking is a fully unexplored and promising area.

8. REFERENCES

[1] D. Abadi et al. Integrating compression and execution in column-oriented database systems. SIGMOD 2006.
[2] D. Abadi et al. Materialization Strategies in a Column-Oriented DBMS. ICDE 2007.
[3] S. Agrawal et al. Database Tuning Advisor for Microsoft SQL Server. VLDB 2004.
[4] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. CIDR 2005.
[5] N. Bruno and S. Chaudhuri. To Tune or not to Tune? A Lightweight Physical Design Alerter. VLDB 2006.
[6] S. Harizopoulos et al. Performance Tradeoffs in Read-Optimized Databases. VLDB 2006.
[7] S. Idreos, M. Kersten, and S. Manegold. Database Cracking. CIDR 2007.
[8] S. Idreos, M. Kersten, and S. Manegold. Updating a Cracked Database. SIGMOD 2007.
[9] M. Kersten and S. Manegold. Cracking the Database Store. CIDR 2005.
[10] S. Manegold et al. Cache-Conscious Radix-Decluster Projections. VLDB 2004.
[11] K. Schnaitter et al. COLT: Continuous On-Line Database Tuning. SIGMOD 2006.
[12] M. Stonebraker et al. C-Store: A Column Oriented DBMS. VLDB 2005.
[13] D. C. Zilio et al. DB2 Design Advisor: Integrated Automatic Physical Database Design. VLDB 2004.
[14] *TPC Benchmark H.* http://www.tpc.org/tpch/.
[15] *MonetDB.* http://monetdb.cwi.nl/.