Self-organizing Strategies for a Column-store Database

Milena Ivanova
CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands
milena@cwi.nl

Martin L. Kersten
CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands
mk@cwi.nl

Niels Nes
CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands
niels@cwi.nl

ABSTRACT

Column-store database systems open new vistas for improved maintenance through self-organization. Individual columns are the focal point, which simplify balancing conflicting requirements. This work presents two workload-driven self-organizing techniques in a column-store, i.e. *adaptive segmentation* and *adaptive replication*. Adaptive segmentation splits a column into non-overlapping segments based on the actual query load. Likewise, adaptive replication creates segment replicas. The strategies can support different application requirements by trading off the reorganization overhead for storage cost. Both techniques can significantly improve system performance as demonstrated in an evaluation of different scenarios.

1. INTRODUCTION

Self-organization is a desirable property for a database system with changing workloads and scarce resources for administration and maintenance. A reorganization of data structures and indices is ideally performed on-line and with minimal overhead on the ongoing work.

In a row-store system reorganization typically affects the indices [12, 13, 5] and materialized views. Both are expensive to construct and their contribution strongly depends on the stability of the workload characteristics and database volatility. Since index maintenance itself is a costly operation, those that do not improve the workload performance are discarded. The challenge is to balance these often contradicting requirements, i.e. to determine which columns of an n-ary table should be indexed.

In contrast, in a column-store system the individual storage of columns opens different opportunities for indexing. There are fewer degrees of freedom. A column is an independent structure which can be indexed or not, without direct effect on the access characteristics of other columns. This opportunity does not exist in the row-stores where a clustered index on one column directly affects the access characteristics on all other columns in a table. Secondary indices counter this behavior as partial replicas of tables.

The materialized views boil down to management of replicated portions of a column using a partial index to speed up localization. Their selection still depends on an (online) assessment of the workload. Some systems produce the replicas as side-effect of their query execution paradigm at no cost, i.e. all relational operations produce a materialized result. In this case the key decision becomes what partial result to retain.

In a self-organizing setting, decisions of this kind should be made without human interference. Ideally the query load is continuously analyzed and reorganization integrated with the query execution scheme. The main contribution of this paper is an exploration of reorganization opportunities in the context of a self-organizing column-store database. We address primarily read-only workloads, for which column-stores have shown to provide substantial performance gains [4]. We focus on scans over base tables and for each query decide if the result influences our segmentation and replication scheme. We consider two dimensions: segmentation with or without data replication. Adaptive segmentation reorganizes data in-place. This strategy initially incurs a relatively high overhead because large pieces of a column are touched and copied for reorganization purposes. Adaptive replication retains copies of query results to improve future access. It incurs a larger storage requirement than adaptive segmentation, but has a smaller performance overhead. All pieces of no interest to queries are left untouched. Partial replication should, however, control the number of replicas for individual data items.

Existing column-stores typically use hardwired segmentation criteria. For example, in C-Store [15] a column is represented as a sequence of 64 KB blocks. Access methods provide one block at a time to the operations in the query plan tree. Since column-stores have to reconstruct logical tuples from individual columns, it is common to base the physical organization on positional order which accelerates this tuple reconstruction process. This organization also means that operations at leaf nodes of the query execution plan, often selection predicates, require access to the entire column stored on disk.

As an alternative we consider *value-based organization* of columns into a collection of segments, each of which covers

EDBT'08, March 25–30, 2008, Nantes, France.

a contiguous range of attribute values. Such an organization allows a sparse index of segments to be maintained and used by the query optimizer to pre-select and access only segments overlapping with the selection predicates on the column. Similarly to block-oriented operators, the query execution proceeds as iteration over segments. However, it does not require a fixed, predetermined size of segments. Instead, the best segmentation is the one that best reflects splitting of the attribute domain into sub-ranges by the query workload. Unfortunately, the workload often cannot be predicted and/or is changing with time, which means that the ideal segmentation is hard to predict and achieve. However, the system can use the workload and gradually create segments for column pieces accessed by queries.

The storage aspects of value-based column organization are also beneficial. A sparse index of segments requires limited storage while at the same time providing for segment access optimizations. In comparison, to speed up selections by indexing a positional-based organization a dense index is needed, which, depending on value distribution, may get close to a full replica of the column, doubling the storage requirements. Value-based organization can also be combined with column-oriented compression algorithms [1, 18].

The suggested value-based column organization has its pitfalls. Since the positional correspondence of values in multiple columns is not kept, operators that rely on it, e.g., tuple reconstruction, may become somewhat slower. In a proper self-organizing system a trade-off between the advantages of the value-based organization and the extra costs incurred for other operators can be used as a criterion during the physical design phase of a column-store database. This paper provides a starting point and outlook on this important research challenge.

The remainder of the paper is organized as follows: Section 2 provides an introduction to our experimentation platform. Section 3 presents the main design issues of self-organization. Sections 4 and 5 detail the segmentation and replication techniques, respectively. An experimental evaluation of the trade-offs of the techniques is given in Section 6. We round off with a short comparison against related work and an outlook on further research in this area.

2. MONETDB ARCHITECTURE

In this section we introduce the contours of MonetDB [10], a widely used open-source column-store DBMS. It provides a frame of reference for our self-organizing techniques and a platform for real-life experiments.

The central storage component in MonetDB is a binary association table (BAT), i.e. a 2-column data structure. BATs can be defined over any of the built-in set of data types. The elements comprising a BAT are physically stored in a contiguous area. There are no holes, deleted elements, or auxiliary data in this storage structure, which means that a BAT can be conveniently split at any point.

The MonetDB engine interprets query plans described in the MonetDB Assembly Language (MAL), which provides a rich set of relational operators over BATs. In addition, it provides a minimal set of building blocks, such as functional abstractions, guarded blocks, and exception handling, to obtain a computationally complete language. The execution paradigm is based on materialization of all intermediate results.

```
function user.s1_0(A0:dbl,A1:dbl):void;
    X1:bat[:oid,:dbl]:= sql.bind("sys","P","ra",0);
    X16:bat[:oid,:dbl]:= sql.bind("sys","P","ra",1);
    X19:bat[:oid,:dbl]:= sql.bind("sys","P","ra",2);
    X23:bat[:oid,:oid]:= sql.bind_dbat("sys","P",1);
    X30:bat[:oid,:lng]:= sql.bind("sys","P","objid",0);
    X32:bat[:oid,:lng]:= sql.bind("sys","P","objid",1);
    X34:bat[:oid,:lng]:= sql.bind("sys","P","objid",2);
    X14 := algebra.uselect(X1,A0,A1,true,true);
    X17 := algebra.uselect(X16,A0,A1,true,true);
    X18 := algebra.kunion(X14,X17);
    X20 := algebra.kdifference(X18,X19);
    X21 := algebra.uselect(X19,A0,A1,true,true);
    X22 := algebra.kunion(X20,X21);
    X24 := bat.reverse(X23);
    X25 := algebra.kdifference(X22,X24);
    X26 := calc.oid(0@0);
    X28 := algebra.markT(X25,X26);
    X29 := bat.reverse(X28);
    X33 := algebra.kunion(X30,X32);
    X35 := algebra.kdifference(X33,X34);
    X36 := algebra.kunion(X35,X34);
    X37 := algebra.join(X29,X36);
    X38 := sql.resultSet(1,1,X37);
    sql.rsColumn(X38,"sys.P","objid","bigint",64,0,X37);
    sql.exportResult(X38,"");
end s1_0;
```

Figure 1: SELECT OBJID FROM P WHERE RA BETWEEN 205.1 AND 205.12

The compilation stack consists of three components: SQL-MAL code generator, a tactical optimizer, and the run time engine. The SQL compiler for MonetDB maps the relational tables into collections of BATs, whose head column is an OID. The query is compiled into MAL using common heuristic optimization rules aimed at data volume reduction. The tactical optimizer is a MAL to MAL transformation system. It is used to refine the plans using information about resource availability, distribution, and general symbolic evaluation rules. Furthermore, each relational operator has an embedded optimizer to choose the best algorithm for the operands involved. This technique, called operational optimization, has proved pivotal in the performance and maintenance of MonetDB system.

Figure 1 illustrates a cached non-optimized query plan derived from the statement SELECT OBJID FROM P WHERE RA BETWEEN 205.1 AND 205.12. After optimization the complete plan involves about 80 MAL operations, including resource management tasks, such as releasing BATs with intermediate results from the storage pool. The target for our adaptation algorithms is to modify this plan before it is taken into execution.

Even though MonetDB supports arbitrarily large BATs, the current implementation is optimized for operations having their operands in main memory. The system relies on the operating system to perform virtual memory I/O operations efficiently, which hinders performance as soon as BAT sizes reach the memory limits.

Although the memory capacity of modern computers continuously increases, the application data sizes also grow. Such high volume data are typical in present-day scientific applications, such as the MonetDB/SkyServer project [8]. For example, a single column of a 4-byte real type of the largest table there already takes 1 GB storage space. Although the MonetDB engine touches only the columns relevant to the query, the size of these columns together with the materialized intermediates can quickly exhaust memory resources and incur multiple I/O operations. Consequently, it is desirable to find a column organization that allows to further limit the I/O operations only to the relevant pieces of the columns.

3. SOLUTION SPACE

A solution for self-organization must include answers to a several design issues. This section discusses the most important of them, namely, the level of the software stack that fits best for integration of self-organization; the schemes to determine whether or not reorganization should be undertaken, and when to perform the reorganization itself.

3.1 Integration in the Software Stack

The main design issue of a self-organizing system is where in the software stack to make data segmentation decisions. Those are commonly part either of the database code, e.g., as hardwired page sizes, or of the logical scheme, e.g. partitioning conditions attached to individual table definitions.

Our quest to produce a self-managing system rules out the second option. Reorganization should have no impact on the front-ends, i.e. the user is unaware of any such decision, nor able to control it directly.

Embedding segmentation decisions in the context of the data structures and the relational operators would be a route taken in a Volcano-style engine [6]. In MonetDB each and every operator is highly tuned towards producing a materialized intermediate result using a main-memory frame of reference. Implementing data segmentation on this level requires a complete re-design of the engine, which would affect thousands of algorithms.

Furthermore, in an ideal world the best segmentation is the one that best anticipates and supports the query workload. Since the future is hard to predict, the recent past is considered a good predictor. A hardwired solution based on a priory fixed sized blocks are not considered a viable solution in the volatile application environments.

Therefore, self-organization should aim at the tactical optimization layer of the MonetDB software stack where global resource decisions are made and MAL programs can be transformed to cope with specific cases. We merely have to identify candidate BATs and inject calls to a *segment optimizer*, which transforms operations against a segmented BAT into a segment-aware instruction sequence against individual segments of the BAT relevant to the query. Two principle replacement strategies are possible and the choice is based on the number of segments, preferably known at query optimization time. For a small number of segments, an instance of the instruction is added for each segment relevant to the query. For a large number of segments an iterator approach is applied. In both cases an additional set of instructions might be needed to construct the result from the partial results if a blocking operator follows in the plan.

The optimizer also collects information about CPU and memory usage patterns of key operators against a BAT and decides to split it into pieces, or glue segments together. The choice to integrate segmentation on the tactical optimizer level is crucial to allow on-line reorganization during the regular work of the system.

The segment optimizer uses an in-memory segment meta-index that allows for easy detection of the segmented tables in the query plans. The catalog describes various segment properties that can be used during query optimization without touching the data. For example, the information about segment sizes is used to estimate the memory footprint of the plan and for efficient memory allocation.

The segmentation decision at the optimizer level may still require some segment aware operations. For example, it is relatively easy to design a SUM() over a segmented BAT, while sorting over a segmented column effectively requires a major re-partitioning. Both algorithms are much faster than their centralized version.

To illustrate the code produced by the segment optimizer, we use the first selection operator from the example in Figure 1. The corresponding snippet from the execution plan is:

```
X1:bat[:oid,:dbl] := sql.bind("sys","P","ra",0);
X14 := algebra.select(X1,A0,A1);
```

Here the X1 variable is bound to a BAT named ra, and the execution plan parameters A0 and A1 are bound to the predicate constants 205.1 and 205.12. The select operator evaluates the range predicate RA BETWEEN 205.1 AND 205.12. For a non segmented RA BAT with positional organization the operational optimizer chooses an implementation based on a full scan.

When the RA BAT is split into value-ranged segments, the segment optimizer transforms the snippet above into a sequence with an iterator over the segments as follows:

```
Y1:bat[:oid,:dbl] := bpm.take("sys_P_ra");
Y2 := bpm.new(:oid,:dbl);
barrier rseg:=bpm.newIterator(Y1,A0,A1);
   T1:=algebra.select(rseg,A0,A1);
   bpm.addSegment(Y2,T1);
redo rseg:= bpm.hasMoreElements(Y1,A0,A1);
exit rseg;
```

The Y1 variable is bound to the segmented on value ranges column RA, while the Y2 variable is bound to a new segmented BAT to hold selection results. The iterator is predicate enhanced and uses the segment meta-index to return only those segments that overlap with the selected range of [A0,A1]. The selection over those segments is registered in the meta-index as a part of the result.

3.2 Segmentation Models

While a static segmentation is based on assumptions about a stable and predictable query workload, self-organizing segmentation adjusts to the workload continuously. Given a

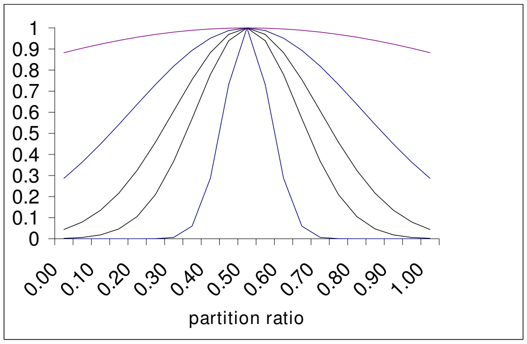


Figure 2: Gaussian Dice

selection predicate on a column, the system needs a policy (also called *segmentation model*) to determine whether the selection should be used to create a segment or not. Intuitively, the policies should avoid creating too many small segments for two reasons: to keep the overhead of segment maintenance minimal, and to avoid segment iteration overhead. On the other hand, keeping large segments incurs unnecessary disk reads and high memory footprint during query execution. Hence, we are looking for models that aim not only to split the column in query-beneficial places, but also to find a balance with respect to segment sizes. Next, we consider two such policies: Gaussian Dice (GD) and Adaptive Page Model (APM).

3.2.1 The Gaussian Dice

This policy exploits randomized behavior to guide segmentation decisions. A naive scheme is to flip a coin with each query to decide on continuous breaking up the column into multiple pieces. To avoid creation of very small segments, we use a 'learning' random generator that reflects the changing segment status as time progresses. The intuition behind this is to give preference to operations that break a segment into approximately equally sized pieces and reduce the impact of point queries on the segments structure.

The ingredients of GD are the ratio between the sizes of the segment produced P and the segment considered for splitting S, and the ratio between the size of S towards the total column size. The basis of the model is a Gaussian probability distribution G with $\mu = 0.5$ and $\sigma = Size_S/TotSize$. The function $O(x) = G(x)/G(0.5)$ is used as a decision function for all segment ratios. The shape of the probability for several values of σ is shown in Figure 2. The choice of σ parameter gives preference to selections splitting relatively large segments.

Whenever a segmentation decision is made, we randomly draw a number $r \in [0..1)$ and check if $r < O(Size_P/Size_S)$. In this way selections that split a segment in a ratio $x = 0.5$ have higher probability to be used for reorganization than operations extracting small pieces.

3.2.2 Adaptive Pagination Model

The second policy is an adaptive pagination model where the decision about reorganization is taken deterministically using estimates of the segment sizes. Aiming to achieve a balance of segment sizes, we introduce a pair of bounds: a lower bound M_{min} is used to guard the system against fragmentation into too small pieces, while an upper bound $M_{max}, M_{min} < M_{max}$ specifies how many extra reads the system is ready to pay for point queries. APM decides about splitting a segment S based on the following rules:

1. if $Size_S < M_{min}$, the segment is left intact.
2. if $Size_S > M_{min}$ the sizes of sub-segments created by the selection are checked. If all of them have estimated size above M_{min}, the segment is reorganized using the materialized result of the selection.
3. if the selection creates small pieces with size under the M_{min} bound, it is considered inappropriate to be used for splitting. However, there is a chance that the segment might be queried again in the near future. To speed up subsequent queries and to better control the memory footprint the segment is reorganized if $Size_S > M_{max}$. Since the query bounds would cause creation of some small piece, the splitting point in this case is chosen among the query bounds or an approximation of the mean value in the segment.

A characteristic of the APM model is that sizes of segments touched by queries converge relatively fast to the interval $M_{min} <= Size_S <= M_{max}$. By adjusting the parameters M_{min} and M_{max} we can tune the policy to be more or less aggressive in issuing column reorganizations.

3.3 Self-organizing Techniques

When the segment optimizer decides that it is beneficial to split a segment S, the third design issue is to decide when to conduct the split. The major alternatives are:

- Post-processing. The optimizer only marks the segment for splitting, but the reorganization is performed after the query is executed. This allows for finding an 'ideal point' for the split, for example producing equi-depth sub-segments balancing memory resources. Furthermore, several suggested splits can be combined in one batch and performed at once choosing optimal splitting points. This alternative resembles existing off-line reorganization in many systems. However, the potential delay may cause subsequent queries on the same segment to miss potential benefits. The total overhead can also be substantial, since the reorganization requires all marked segments to be loaded again in memory and scanned.
- Eager materialization. The optimizer keeps the selected sub-segment and eagerly reorganizes the original one. Thus, the creation of the selected sub-segment is piggy-backed on the query execution and incurs no extra costs. However, the sub-segments outside the selection scope have to also be materialized, which incurs the main reorganization overhead. This overhead substantially increases the processing time of initial queries when the entire column or its large parts are being reorganized. It becomes faster as more and more queries are processed. Another issue for consideration is that the selection bounds may not be the optimal

Algorithm 1 Adaptive segmentation

```
for all segments $S$ overlapping with query range $[QL, QH]$ do
    if segmentation model decides split of $S$ then
        scan $S$ and materialize its sub-segments
        replace $S$ with its sub-segments
```

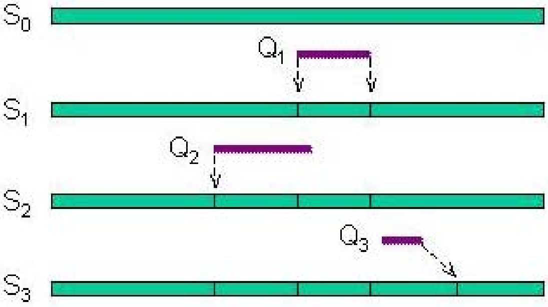


Figure 3: Segmentation example

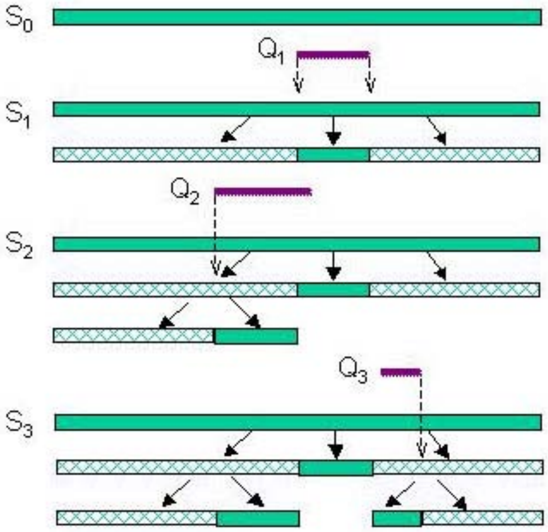


Figure 4: Replication example

split points from fragmentation and memory resource point of view. We will call this alternative *adaptive segmentation*.

- Lazy materialization. The materialized results of a selection are kept and registered in the segment meta-index, but the remaining pieces are reorganized upon need when subsequent queries touch them. Thus, partial replica segments are created which incurs bigger storage requirements. The strategy also affects the query execution when a best suited replica set needs to be chosen for each query. The main advantage is that almost entire reorganization is piggy-backed on the query execution resulting in minimal total overhead and minimal disturbance on the query load. We will call this alternative *adaptive replication*.

In this work we consider the last two self-organization techniques interleaved with the query execution. The segment optimizer implements them by injecting calls to a corresponding reorganizing module after select operations over segmented tables. In the following sections we describe those strategies in detail.

4. ADAPTIVE SEGMENTATION

With this technique a column is represented as a sequence of adjacent non-overlapping segments. Initially, the column is stored in a single segment which is gradually reorganized into a list of segments as selection queries arrive. Each request creates an opportunity for splitting some segments. Whether this opportunity is exploited by the system is decided using the segmentation model, GD or APM. If the model decides to split a segment, it is *replaced* by its two or three sub-segments.

The main steps of adaptive segmentation are shown in Algorithm 1. Figure 3 illustrates the process using the APM model for an example load of three queries. In the initial state S_0, the column is represented by a single segment. Query Q_1 causes its reorganization into three segments (rule 2). Next, Q_2 issues a split of the first sub-segment, but not of the second where the selection is too small (rule 2 is not fulfilled). Note, that query Q_2 does not need to scan the last segment which does not overlap with its range, i.e. it immediately benefits from the reorganization triggered by the first query. Finally, query Q_3 with small selectivity causes a split at the mean value of the last segment (rule 3).

5. ADAPTIVE REPLICATION

In adaptive replication segments are organized in a hierarchical structure, a *replica tree*. A segment S is a child of a segment P if the range of values in P is a super-set of the range of values in S. The root of the tree is a segment containing the entire domain of values of the attribute stored in the column.

We introduce two types of segments: materialized and virtual. The materialized segments contain real data. Virtual segments are used to support the replica tree by completing the ranges of materialized segments. Consider a selection query $[QL, QH]$ that splits a segment S with range $R = [SL, SH]$ into two sub-ranges $R1 = [SL, QL - 1]$ and $R2 = [QL, SH]$ and assume for simplicity an integer domain and inequality $SL < QL < SH < QH$. The values in the range $R2$ are materialized as a result of the selection operator, and thus a sub-segment with this range is created at no extra cost. The range $R1$ is a complement of the selected range $R2$ to the original segment range R. A virtual segment $S1$ is created with a range $R1$ and its size is estimated, but no data is copied. Both new segments $S1$ and $S2$ are attached to the replica tree as children of segment S. If a later query hits the virtual segment $S1$, the system may choose to materialize it. In that case the segments $S1$ and $S2$ would form a full replica of the segment S which allows dropping the original segment S to reduce the storage requirements.

The growing of the replica tree for the example workload is illustrated in Figure 4. Virtual segments are depicted with cross-filling. The result of selection $Q1$ is kept as a

Algorithm 2 Adaptive replication

```
procedure AdaptReplication($ql, qh$)
    $cv \leftarrow getCover(ql, qh, root)$
    for all $s \in cv$ do
        $M \leftarrow analyseRepl(ql, qh, s)$
        scanMat(s,M)
        check4Drop(s)
end procedure
```

Algorithm 3 Find minimal covering set of a query

```
$cover$ global variable for covering set
$cur$ current position in cover
function getCover($ql, qh, s$)
    $start \leftarrow cur$
    if $s.ancnumber = 0$ then                    ▷ Recursion bottom
        if $s.virtual$ then
            return 0
        else
            $cover[cur] \leftarrow s$
            $cur \leftarrow cur + 1$
            return 1
    else                                         ▷ Recursion on ancestors
        for all $p \in s.ancestors$ do
            if $qh >= p.low \wedge ql <= p.hgh$ then
                if $(getCover(ql, qh, p) = 0)$ then
                    $cur \leftarrow start$                 ▷ Backtrack
                    if $s.virtual$ then
                        return 0
                    else
                        $cover[cur] \leftarrow s$
                        $cur \leftarrow cur + 1$
                        return 1
        return 1
end function
```

replica segment. Together with two complementing virtual segments they cover the entire domain range. Note, that both queries Q_2 and Q_3 overlap with virtual segments and need to scan the entire column in contrast with adaptive segmentation.

The main steps in adaptive replication are shown in Algorithm 2. First, given the selection bounds $[QL, QH]$ the system determines a set of materialized segments to be used for query answering. Each segment in the set is analyzed for potential replication using the segmentation model. If the system decides to create a replica, it is added to a materialization list, M. Then a single scan of the covering segment is used to materialize the replicas in the list and the query results. Finally, the algorithm checks if the segment can be dropped from the replica tree in case its children fully replicate it. Next, we will describe those steps in more detail.

Since data is partially replicated, query processing should avoid generating the same results more than once. To exploit advantages of partial replicas, the system should determine an optimal replica set to answer a query. Since virtual segments do not contain data, the optimizer finds the minimal set of materialized segments covering the selection query. More formally, the minimal covering set $S = \bigcup S_i$ for a query $[QL, QH]$ is defined as follows:

Algorithm 4 Analyze possible replicas. Return list M

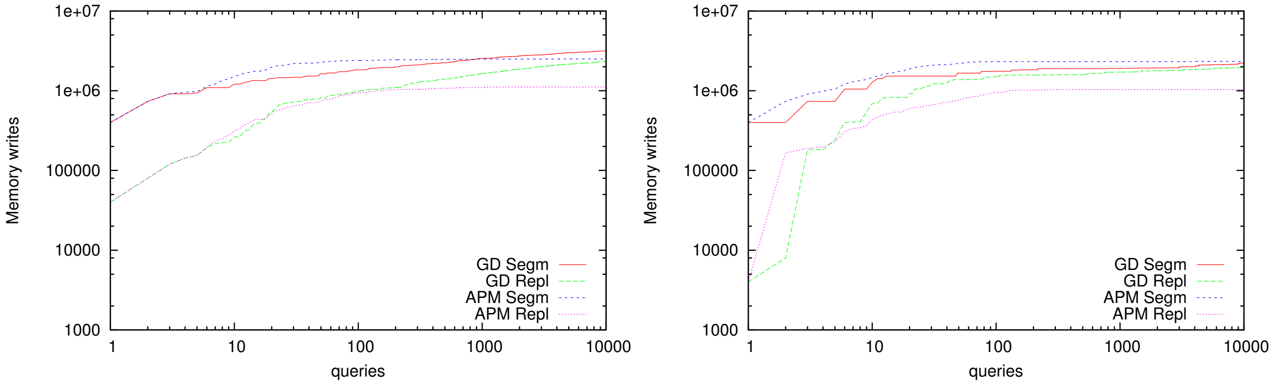
```
function analyzeRepl($ql, qh, s$)
    if $s.ancn = 0$ then                         ▷ Recursion bottom
        $c \leftarrow segModel(ql, qh, s)$
        switch $c$ do
         case 0 :                    ▷ query entirely covers s or
                                  ▷ small subsegments in small s
            if $s.virtual$ then
                $M \leftarrow s$        ▷ s is materialized without split
         case 1 :                    ▷ query covers lower part of s
            $m \leftarrow newSegment(s, s.low, qh,' mat')$
            $v \leftarrow newSegment(s, qh + 1, s.hgh,' vir')$
         case 2 :                    ▷ query covers upper part of s
            $v \leftarrow newSegment(s, s.low, ql - 1,' vir')$
            $m \leftarrow newSegment(s, ql, s.hgh,' mat')$
         case 3 :                    ▷ query entirely inside s
            $v1 \leftarrow newSegment(s, s.low, ql - 1,' vir')$
            $m \leftarrow newSegment(s, ql, qh,' mat')$
            $v2 \leftarrow newSegment(s, qh + 1, s.hgh,' vir')$
         case 4 : ▷ some subsegment is small but s is large
            if $s.low < ql \wedge s.hgh > qh$ then
                                     ▷ split on one query border
                if $qh - s.low < s.hgh - ql$ then
                    $m \leftarrow newSegment(s, s.low, qh,' mat')$
                    $v \leftarrow newSegment(s, qh + 1, s.hgh,' vir')$
                else
                    $v \leftarrow newSegment(s, s.low, ql - 1,' vir')$
                    $m \leftarrow newSegment(s, ql, s.hgh,' mat')$
        end switch
        $M \leftarrow m$
    else                         ▷ Analyze ancestors for replication
        $M \leftarrow \emptyset$
        for all $p \in s.ancestors$ do
            if $qh >= p.low \wedge ql <= p.hgh$ then
                $M \leftarrow M \cup analyzeRepl(ql, qh, p)$
    return M
end function
```

1. All segments S_i are materialized.
2. The selection range $[QL, QH]$ is included in the union of segment ranges $[SL_i, SH_i]$.
3. No segment can be replaced by its materialized children without violating rule 2.
4. No segment can be dropped from the set without violating rule 2.

The steps for finding the covering set are presented in Algorithm 3. An important assumption while finding the covering set is that all segments are equally accessible in memory.

For each segment in the covering set the algorithm analyzes the overlap between the query range and the segment bounds to decide about replica creation using the segmentation model, GD or APM, and their parameters. Algorithm 4 sketches the rules of this analysis. Following the main idea to create replicas only for those parts in which queries have expressed interest, the algorithm adds replicas of selection ranges (cases 1, 2, and 3) or the smallest super-set of the selection (cases 0 and 4).

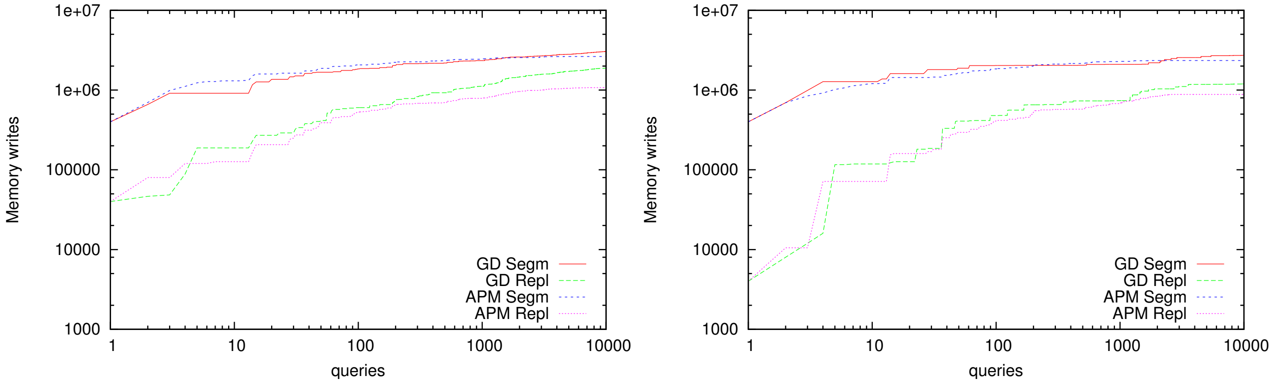
As time passes, the number of replicas grows and the system must take care to not over-utilize storage resources. This is



(a) Selectivity 0.1

(b) Selectivity 0.01

Figure 5: Cumulative memory writes due to segment materialization. Uniform distribution



(a) Selectivity 0.1

(b) Selectivity 0.01

Figure 6: Cumulative memory writes due to segment materialization. Zipf distribution

achieved by continuous checking for possible replica deletions. If all immediate children of a segment are materialized, the segment can safely be dropped from the tree and its children attached directly to its parent. If this is a materialized segment, the effect is a release of storage resources. The steps are described in Algorithm 5. Currently, we do not impose limitations on the replica tree depth, nor on the total storage budget. These issues are a subject of future work. The next section includes an evaluation of the depth and storage parameters of the replica tree.

Algorithm 5 Check and drop replica

```
procedure CHECK4DROP($s$)
    if $s.ancnumber = 0$ then return
    for all $p \in s.ancestors$ do
        CHECK4DROP(p)
    for all $p \in s.ancestors$ do
        if $p.virtual$ then
            return                    ▷ children do not replicate s
    $q \leftarrow s.parent$                   ▷ s can be dropped
    for all $p \in s.ancestors$ do
        $p.parent \leftarrow q$
    replace $s$ in $q.ancestors$ with $s.ancestors$
    free memory allocated for $s$
end procedure
```

6. EXPERIMENTAL EVALUATION

The evaluation includes a thorough simulation of both adaptive techniques complemented with experimentation with adaptive segmentation on a real workload on a 100 GB data set from the SkyServer project [14].

6.1 Simulation

We simulated the core algorithms of MonetDB, its management in a constrained memory buffer setting, and its read/write behavior as data is flushed to secondary store. This architecture conscious simulator has proved to be a good basis for assessing the techniques proposed before the engineering task started.

The results reported here are based on a column with 100 K values taken from a domain of a 1 M different integer values. We simulated both adaptive segmentation and replication using both segmentation models GD and APM. The APM bounds were set to 3 KB and 12 KB, respectively for M_{min} and M_{max}. The simulated workload contained 10 K range selection queries. To investigate strategy behavior with various workloads we chose two selectivity factors (0.1 and 0.01), as well as uniform and skewed (Zipf) distribution of the queries over the attribute domain.

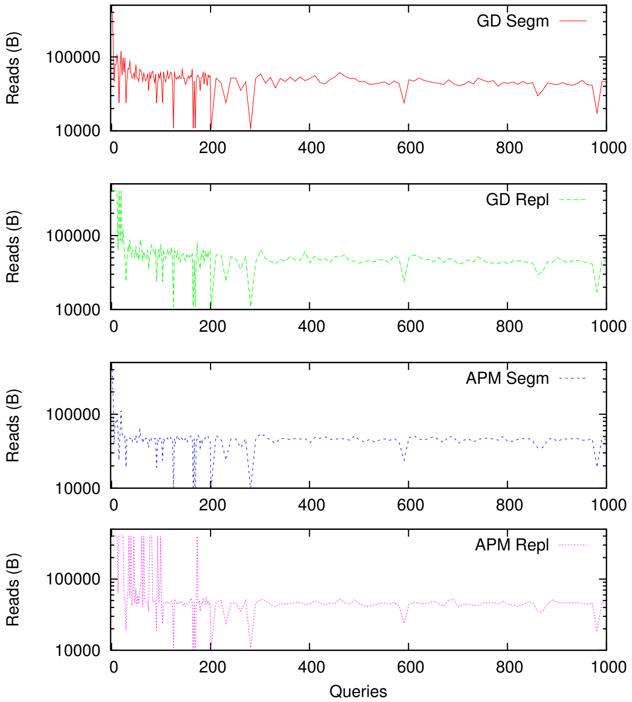


Figure 7: Memory reads for the first 1000 queries. Uniform distribution, selectivity 0.1

6.1.1 *Overhead*

To estimate the reorganization overhead we measured the number of writes due to segment materialization with segments including query results, as shown in Figures 5 and 6 for uniform and zipf distribution, respectively. For all combinations of selectivity and distribution, adaptive replication requires less writes than its counterpart segmentation. This confirms our expectations, since the replication lazily materializes segments after being accessed by some query. For the deterministic APM model, the reduction of writes is stable by a factor of 2.5 (note that graphs use log scale), while for the GD model the difference depends on the workload characteristics.

In general, the APM model stops reorganizing the column after an initial number of queries. With uniform query distribution (Fig.5) this saturation comes after approximately a hundred queries, while with skewed distribution (Fig.6) we still observe reorganization after 3000 queries since some previously untouched areas of the domain are hit for the first time and reorganized. In contrast, the GD model keeps issuing reorganization with decreasing probability, since it does not have strict bounds for segment sizes.

A close look at the first few queries shows that the GD model is less aggressive in replication and thus can be preferable if we want to reduce the overhead on the initial workload.

The selectivity factor in the query load affects the segmentation decisions. While segmentation reorganizes an entire segment independently of the precise selected size, the replication overhead is directly affected by the selectivity factor, as it can be seen for the initial queries. This observation also hints that the replication technique needs a guard against replicating too big pieces.

Table 1: Average read sizes in KB for 10 K queries

Strategy	U 0.1	U 0.01	Z 0.1	Z 0.01
GD Segm	40.7	31.2	41.8	11.2
GD Repl	41.1	28.5	43.7	11.1
APM Segm	43.6	12.7	46.3	11.3
APM Repl	45.0	13.2	48.5	13.4

6.1.2 *Benefits*

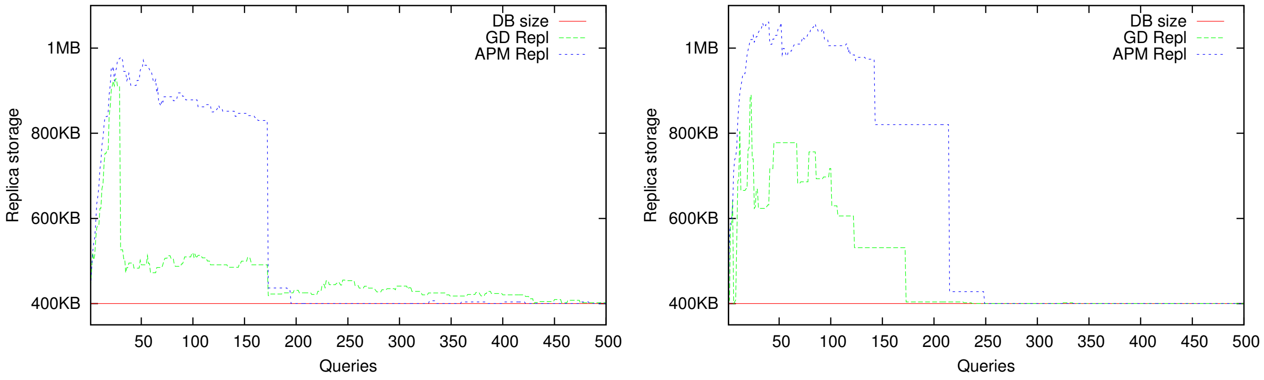
Next, we investigated how different strategies achieve the main goal of segmentation, namely to improve performance by limiting data scans to query-relevant pieces of the column. Figure 7 shows the number of memory reads during the first 1000 queries with uniform distribution. As expected, the number of reads drops very fast with adaptive segmentation for both APM and GD models. Initially, the replication curves, especially the APM replication, show a number of spikes corresponding to a full scan of the column. Those spikes are caused by queries that hit untouched areas of the attribute domain that are not covered yet by smaller partial replicas (recall queries $Q2$ and $Q3$ in the example in Section 5). In all cases the number of reads reduces and stabilizes as query workload progresses.

Table 1 shows the average number of reads per query for the entire experimental sequence of 10 K queries. For the workload with selectivity 0.1 the number of reads converges to the minimal number of 40 KB for all strategies and query distributions, with values for replication slightly above the segmentation values.

The number of reads with the APM model and selectivity 0.01, converges to 11-13 KB and does not reach the minimum determined by the selection size of 4 KB. The reason is that small selections trigger a split only if the segment size $Size_S > M_{max}$, set here to 12 KB. Since entire segments are read the number of reads cannot go under the segment sizes. The GD strategy creates large segments for uniformly distributed queries and selectivity 0.01. Since small selectivity in GD, i.e. a small value of x, is associated with small probability for a split, the reorganization is triggered very rarely and segments remain relatively large.

6.1.3 *Replica Tree*

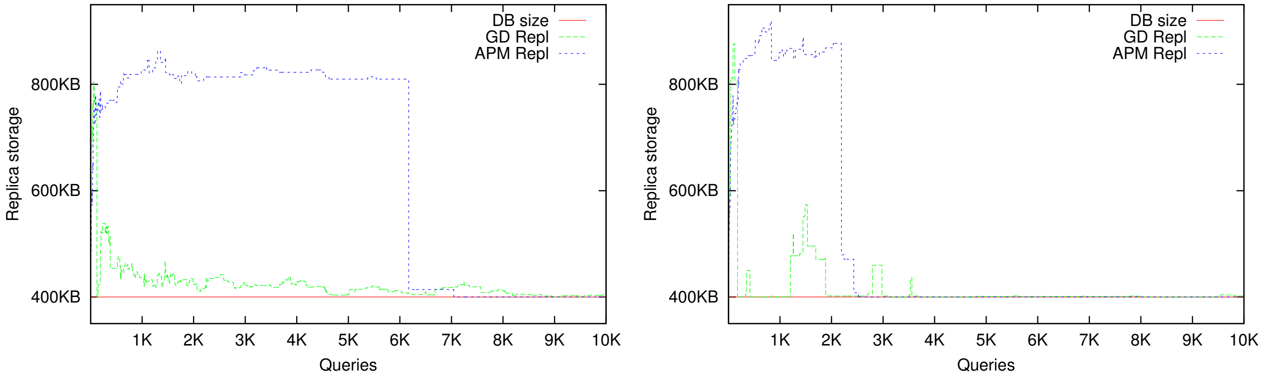
Since adaptive replication requires extra storage, we also used the simulator to analyze the parameters of the replica tree. With a uniformly distributed query load, the replica tree needs extra storage of about 1.5 times the column size, which reduces substantially after the first 250 queries (Fig. 8). The biggest drops in the storage curve correspond to the moments when the initial segment containing the entire column was fully replicated by its materialized children and dropped. The replica tree transforms into a structure very close to the segment list created by the adaptive segmentation with occasional, small size replicas. This process also occurs with skewed workload (Fig. 9) but it takes much longer (between 3 K and 6 K queries) until the skewed queries hit and reorganize all areas of the attribute domain.



(a) Selectivity 0.1

(b) Selectivity 0.01

Figure 8: Replica storage. Uniform distribution



(a) Selectivity 0.1

(b) Selectivity 0.01

Figure 9: Replica storage. Zipf distribution

Another observation is that storage needs always reduce faster with the GD model. The main reason is that the models treat large segments with small sub-segments differently: The APM would split a materialized segment S if $Size_S > M_{max}$ materializing a sub-segment that is a superset of the selection. Thus, APM allocates storage but needs to wait until a new query hits the non materialized subsegments before getting a full replica and releasing the storage taken by S. In the same situation, due to the small selection and, hence, small value of the parameter x, the GD model will decide with high probability to not split, thus not taking any extra storage. Similarly, if the segment S is virtual, the GD decision to not split it causes its materialization at once, thus allowing its parent segment P to be dropped if S was P's only virtual ancestor. APM again materializes a sub-segment of S and waits for a new query to trigger the materialization of complementary sub-segments before it is possible to release the storage taken by P.

6.2 SkyServer Workload

To ground the simulation we also performed experimental runs against a prototyped extension of the MonetDB system. In this section we report on results obtained with the adaptive segmentation algorithm on a real workload from the SkyServer project [14].

The evaluation platform is a PC with two Dual Core AMD Opteron(tm) Processor 270 2 GHz and 8 GB memory. We took a 100 GB sample of the SDSS -4 database. This still fits on a simple desktop PC, but certainly makes the database disk bound on most queries. Even in a column-store DBMS.

We filtered the queries overlapping with the footprint of the 100 GB database from a one-month query log of SkyServer. The column of interest is the right ascension (RA), a real data type, included in most spatial search queries. Three workloads, each of 200 queries, were extracted: RANDOM picks one out of every 300 queries and covers the attribute domain uniformly; SKEW extracts 200 subsequent queries from the log that access two very limited areas of the domain, and CHANGING consists of four pieces of 50 subsequent queries with changing point of access. We used two versions of the APM model with M_{max} set to 5 MB and 25 MB, respectively, and M_{min} set to 1 MB.

We expected adaptive segmentation to show better query times after the initial overhead is amortized with the gains from the faster selection. We also expected the adaptive strategies to suit better to the skewed workload.

Figure 10 shows the average time spent in adaptation vs. selection after the first 200 queries. For all workloads the

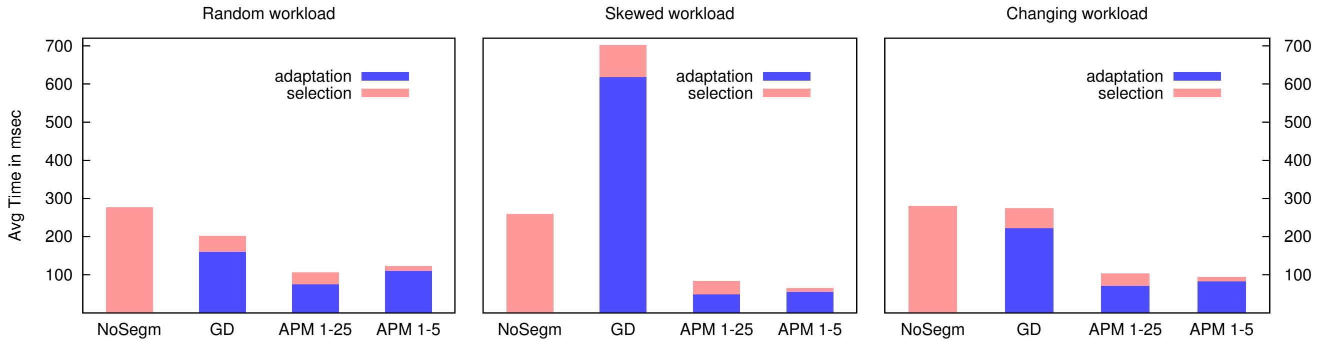


Figure 10: Times for adaptation and selection

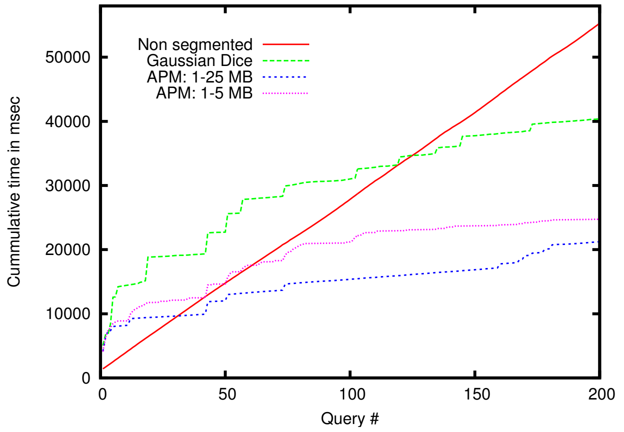


Figure 11: Cumulative time for random workload

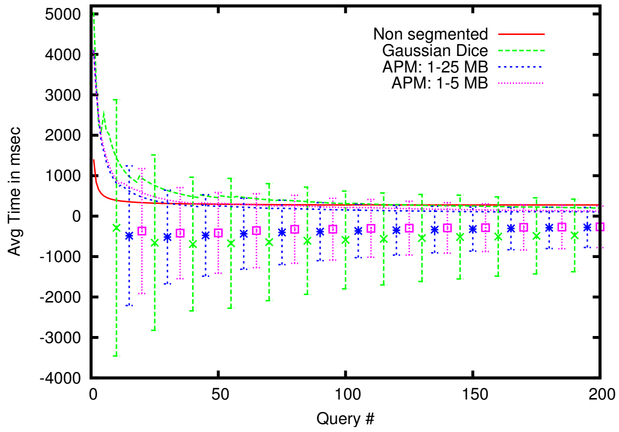


Figure 12: Moving average query time for random workload

Table 2: Segments statistics

Load	Scheme	Segm.#	Avg size	Deviation
Random	GD	31	5.6	7.9
Random	APM 1-25	23	7.6	7.5
Random	APM 1-5	62	2.8	1.3
Skewed	GD	100	1.7	9.9
Skewed	APM 1-25	6	28.9	9.6
Skewed	APM 1-5	10	17.4	14.5

adaptation overhead for the APM schemes is smaller than for Gaussian Dice since the former is more conservative in splitting small segments. Similarly, the overhead for APM1-5 is bigger than for APM1-25 which does not split when a small range is selected out of a segment with $Size_S < 25\,\text{MB}$.

Due to the smaller upper bound the APM1-5 scheme creates smaller segments than APM1-25, as shown by the segment statistics in Table 2. Smaller segments give bigger gain from saved scanning as illustrated by the reduced selection times in Figure 10.

Figures 11 and 12 show a more detailed picture of accumulated and moving average query times, respectively, for the adaptive schemes compared with a non-segmented database on a random workload. The initial overhead for reorganization in adaptive strategies slows down the first queries but provides better system response after a relatively small number of queries, with APM1-25 first amortizing the overhead after 30 queries.

Figures 13 and 14 illustrate the accumulated and moving average query times for skewed workload. The total overhead of APM schemes is smaller than for a random load, since the reorganization affects a very limited area of the domain. However, the GD scheme hits its worst case. In the skewed load queries hardly differ in their selection predicate and chop very small pieces. As a result, 80% of the segments contain less than 1000 tuples, which are expensive to reorganize and, in addition, need gluing of small pieces for the subsequent queries.

The performance for a changing workload in Figures 15 and 16 illustrates how shifting the point of query interest triggers reorganization of untouched segments. It results in a temporary increase of the overhead after queries 50 and 100, which evens out soon after too.

7. RELATED RESEARCH

The major DBMS vendors support segmentation and replication in a static, non self-organizing way that requires explicit human guidance. Workload analyzers are considered

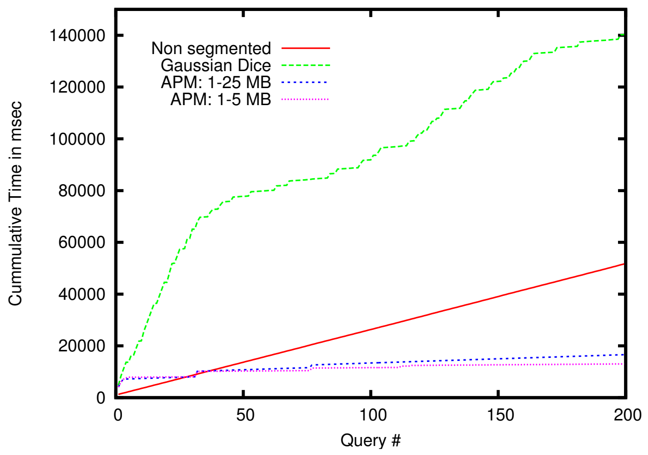


Figure 13: Cumulative time for skewed workload

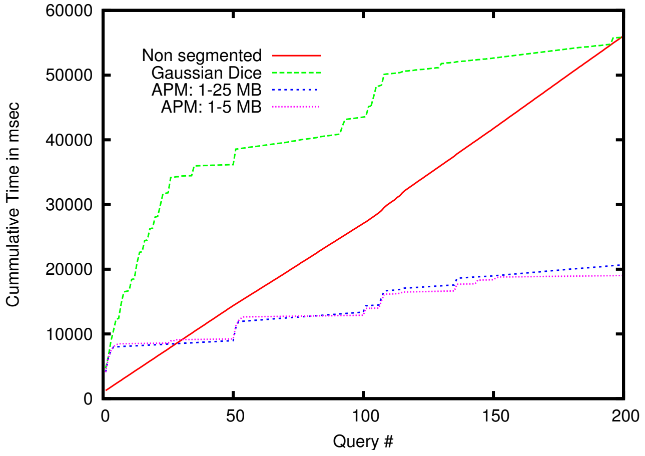


Figure 15: Cumulative time for changing workload

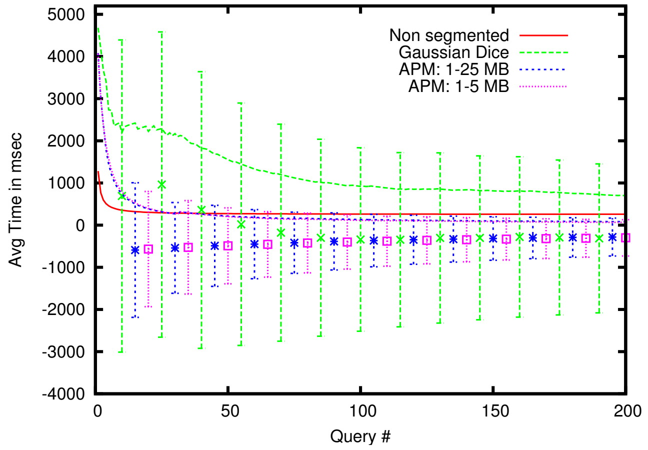


Figure 14: Moving average query time for skewed workload

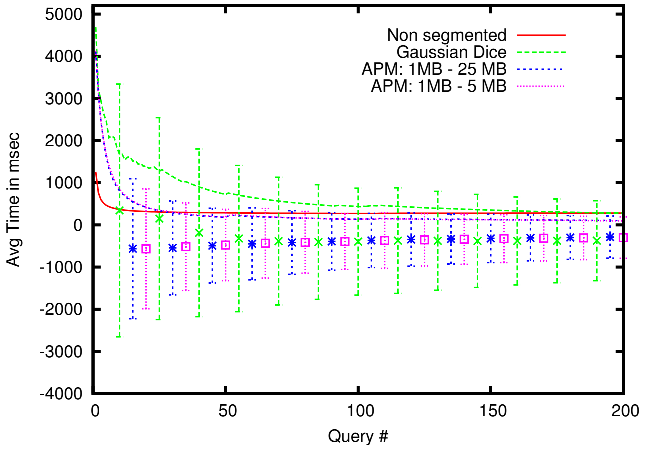


Figure 16: Moving average query time for changing workload

the tools to help in this process [17, 16, 3]. The main principle is to find an optimal configuration of the database for the expected application workload. Since the task is resource intensive, it is performed off-line and relies on human guidance to determine the representative workload and to take the final decisions. AutoPart [11] proposes workload-based database partitioning and illustrates it on the SkyServer database. The algorithm is applied off-line and on the level of the SQL schema. In contrast, adaptive techniques work continuously on the tactical optimizer level completely transparently for the SQL front-end.

On-line tools for physical design tuning have been a hot area of research in the last years [12, 13, 5]. They monitor the query load, keep statistics on the existing and virtual alternatives and change the auxiliary access structures (indices and materialized views). We however investigate reorganization techniques applicable on-the-fly to the main data structures in a column-store.

The related work on column-stores focuses on issues such as materialization of intermediates in the query plan [2], combining compression with query execution [1], etc., and strongly relies on the assumption about positional ordering of columns. Projections (sets of columns) are organized using primary and secondary sorting which keeps positional correspondence of values in multiple columns. Data access is block-oriented and iterators process all blocks, while adaptive techniques presented here exploit a value-based organization per column to limit data access to the segments of interest.

Our approach is in-line with the promising development of database cracking [7], which, however, reorganize a complete in-memory replica of the cracked column. In contrast, the adaptive segmentation reorganizes the column itself in a way suitable for large columns residing on disk and only needs to keep the segment meta-index in memory.

Sybase-IQ is a system with column-store as an auxiliary indexing scheme. It represents the case where a table attribute is complemented with a replica organized as a bit-map. The decision which attribute to handle this way is again a human decision.

SD-SQL Server [9] also proposes dynamic reorganization of the segments of *scalable tables*. The reorganization is, however, issued on the SQL front-end level, which requires special SQL syntax when queries are posted against scalable tables. Adaptive segmentation is supported on the tactical optimizer level and is transparent to the front-ends. Splitting in SD-SQL Server is controlled by the inserted data values and a pre-set segment capacity, while in adaptive segmentation it is controlled by the access frequency in the query workload. Finally, the target applications also differ: SD-SQL Server aims to fulfill the needs of gradually growing tables for mixed-workload applications, while we address data warehouse applications with few large bulk loads and prevailing read-only queries.

8. SUMMARY

Adaptive segmentation and replication in a column-store provide an outlook on significant performance improvements and simplified management. Reorganization decisions can be made an integral part of query execution, i.e. query results are harvested to improve future performance.

Both the careful simulation and the experimentation with a prototype implementation of the adaptive segmentation technique, confirmed our expectations for improved performance due to optimized access to relevant pieces of the columns with value-based segment organization.

Both strategies demonstrate their advantages and shortcomings in different scenarios. The adaptive segmentation uses a minimal amount of storage to maintain a sparse segment index, but the start-up costs are relatively high. The adaptive replication trades off some extra storage to allow further reduction of the overhead. Likewise, the APM segmentation model can be chosen when a good overall long-term reduction of the overhead is preferable, while GD model is useful if it is important to reduce the initial overhead or the extra storage need of adaptive replication.

This research opens a number of interesting issues. Adaptive replication needs further investigation on how to automatically achieve optimal replica configuration in the presence of storage limitations. To achieve complete self-organization, the APM segmentation model needs to automatically determine the values of its controlling parameters. Another direction of work are complementary merging strategies that counter the fragmentation into small segments occurring with GD model for some query workloads. Orthogonal to the above issue is how to exploit the partitioning provided by the segmentation and replication in a distributed column-store system.

Acknowledgments

The authors would like to thank to the SkyServer team for providing the 100 GB data set. Part of this research has been funded by the Dutch BSIK/BRICKS project.

9. REFERENCES

[1] D. J. Abadi, S. R. Madden, and M. C. Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In *Proc. SIGMOD*, 2006.

[2] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. Madden. Materialization Strategies in a Column-Oriented DBMS. In *Proc. ICDE*, pages 466–475, 2007.

[3] S. Agrawal et al. Database Tuning Advisor for Microsoft SQL Server 2005. In *Proc. VLDB*, 2004.

[4] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. CIDR*, Asilomar, CA, USA, 2005.

[5] N. Bruno and S. Chaudhuri. An Online Approach to Physical Design Tuning. In *Proc. ICDE*, pages 826–835, 2007.

[6] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[7] S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. In *Proc. CIDR*, Asilomar, CA, USA, January 2007.

[8] M. Ivanova, N. Nes, R. Goncalves, and M. L. Kersten. MonetDB/SQL Meets SkyServer: the Challenges of a Scientific Database. In *Proc. SSDBM*, Banff, Canada, July 2007.

[9] W. Litwin, S. Sahri, and T. Schwarz. An Overview of a Scalable Distributed Database System SD-SQL Server. In *Proc. BNCOD*, pages 16–35, 2006.

[10] MonetDB, http://monetdb.cwi.nl/, 2007.

[11] S. Papadomanolakis and A. Ailamaki. AutoPart: Automating Schema Design for Large Scientific Databases Using Data Partitioning. In *Proc. SSDBM*, pages 383–392. IEEE Computer Society, 2004.

[12] K. Sattler, I. Geist, and E. Schallehn. QUIET: Continuous Query-driven Index Tuning. In *Proc. VLDB*, 2003.

[13] K. Schnaitter, S. Abiteboul, et al. COLT: Continuous On-line Tuning. In *Proc. SIGMOD*, pages 793–795, 2006.

[14] Sloan Digital Sky Survey / SkyServer, http://cas.sdss.org/, 2007.

[15] M. Stonebraker et al. C-Store: A Column Oriented DBMS. In *Proc. VLDB*, 2005.

[16] G. Valentin et al. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *Proc. ICDE*, 2000.

[17] D. C. Zilio et al. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *Proc. VLDB*, 2004.

[18] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proc. ICDE*, Atlanta, GA, USA, 2006.