Adaptive Segmentation for Scientific Databases

Milena Ivanova, Martin L. Kersten, Niels Nes

CWI, Amsterdam, The Netherlands

{milena,mk,niels}@cwi.nl

***Abstract*— In this paper we explore database segmentation in the context of a column-store DBMS targeted at a scientific database. We present a novel hardware- and scheme-oblivious segmentation algorithm, which learns and adapts to the workload immediately. The approach taken is to capitalize on (intermediate) query results, such that future queries benefit from a more appropriate data layout. The algorithm is implemented as an extension of a complete DBMS and evaluated against a real-life workload. It demonstrates significant performance gains without DBA assistance.**

I. Introduction

Emerging column-store database systems [1], [2], [3] call for a revisit of the predominant segmentation techniques to cope with resource limitations, e.g., disk IO, network, memory, and CPU. As they deal with single columns only, their solution space is less complex and enables a better outlook for a self-organizing data segmentation scheme.

In this paper we present the design and evaluation of an adaptive data segmentation technique for the MonetDB system. The solution is based on extending its tactical optimizer to modify query plans by injecting calls to a segment manager to exploit intermediate results. The adaptive segmentation aims for a data layout beneficial for the future queries based on the recent past and the database state. It is closely related to the crackers approach [4], which exploits value-based organization per column to aggressively create new segments each and every query. In contrast, our approach makes segmentation decisions based on heuristic schemes aiming to balance the sizes of disk-based data segments.

Our approach aligns with and extends the recently proposed on-line tuning techniques [5], [6], [7], which focus on an adaptive solution for the index selection problem by monitoring the query load and changing the auxiliary access structures (indices and materialized views) on-the-fly. The adaptive segmentation reorganizes the data itself and partially shares the overhead with the query execution. It is evaluated against a real-life query load from the SkyServer project [8]. Self-organizing segmentation is urgent for the domain of scientific databases, where unpredictable workloads against high volume data come with usually scarce (human) resources for data management.

The remainder is organized as follows: Section II introduces the MonetDB system and Section III describes the main idea of the adaptive segmentation. Two heuristic adaptation schemes are presented in Section IV. The evaluation of the algorithms is given in section V and Section VI summarizes.

II. Background

MonetDB [1] is an open-source column-store DBMS. The central storage component is a binary association table (BAT), i.e., a 2-column data structure. BATs can be defined over the built-in data types and its elements 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 query plans are described in the MonetDB Assemble Language (MAL), which provides a rich set of relational operators over BATs. Their execution paradigm is based on materialization of all intermediate results. The BAT storage enables a different physical organization (sort order, segmentation, etc.) to be chosen for each column in a SQL table. In particular, if an ordered column is segmented on value, a compact meta-index on the segments speeds up selections while having very small storage needs.

Our quest to produce a self-managing system that reorganizes data transparently to the user leads to the choice of the tactical optimization layer of the MonetDB software stack, where global resource decisions are made and MAL programs are transformed to cope with specific cases [9]. We merely have to identify candidate BATs and inject calls to a *segment optimizer*, which transforms operations against a segmented BAT into an instruction sequence over the segments of the BAT relevant for the query. The optimizer uses heuristic schemes to decide about splitting a BAT and injects calls to a reorganizing module. This allows segment reorganization to be performed in an on-line manner during the regular work of the system.

III. Adaptive Segmentation

In the context of our SQL front-end the head type of a BAT is always OID and the tail type is derived from an SQL column. The OID is often kept implicit, i.e. derived from the position of the element in the table and an administrative offset. The result BATs are sorted on the head, which benefits many relational algebra algorithms.

The BAT representation severely limits the segmentation alternatives to the tail, head, a combination of both, or derived from another BAT. Range segmentation on the head allows to keep implicit OID column, and hence it is space efficient and advantageous for many expensive relational operations, such as hash joins on very large BATs.

Range segmentation on the tail is a value-based partitioning scheme, which means that query execution can avoid touching non overlapping segments. However, it destroys head-ordering

978-1-4244-1837-4/08/$25.00 © 2008 IEEE

and requires OID materialization. This increases the storage needs and slows down the individual element access.

Since the n-ary relations are broken into a series of BATs, all should be split in the same way to realise a traditional horizontal partitioning. Derived segments of a BAT B contain elements with the same OIDs as the correspondent segments of A. The derived segmentation is beneficial for tuple reconstructing joins and in a distributed setting, where keeping all corresponding segments at a single site avoids network costs. In this work we focus on the value-based segmentation.

To illustrate the work of the segment optimizer, we use the following code snippet with a range selection over a BAT R.

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

Here X1 is a variable bound to the BAT R, and A0 and A1 variables are bound to constants specifying the selection range. The segment optimizer finds out in the segment meta-index that R is a segmented BAT and transforms the snippet above into the following sequence:

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

The new code includes a predicate enhanced iterator that uses the segment meta-index to return only those R segments that overlap with the selected range $[A0, A1]$. The partial selection results over those segments are registered in the meta-index as pieces of the segmented result BAT Y2.

Having on hand the materialized result of the selection T1, the optimizer calls the adapt module to reorganize the original segment RSEG on-the-spot. The processing time of the current query increases, but the total overhead of re-segmentation is minimized, since it re-uses the materialized selection result.

IV. Segmentation Schemes

The adaptive scheme implemented in MonetDB explores two routes: randomized decision making and adaptive pagination. Both use heuristics to minimize overhead of the reorganization decision algorithm.

A. The Gaussian Dice

The Gaussian dice (GD) scheme uses a 'learning' random generator that reflects the changing segment status as time progresses. The basis is a Gaussian probability distribution G with $\mu = 0.5$ and $\sigma = Size_S/TotSize$, where $Size_S$ is the size of the segment S considered for splitting, and $TotSize$ is the total size of the BAT. This σ value gives preference to operations splitting relatively big segments.

The function $O(x) = G(x)/G(0.5)$ is used as a decision function with x set to the ratio $Size_P/Size_S$, where $Size_P$ is the size of the produced segment P. Whenever a segmentation decision is made, we randomly draw a number $r \in [0..1)$ and check if $r < O(x)$. In this way operations splitting a segment in a ratio close to $x = 0.5$ have higher probability to be used for reorganization than selections extracting small pieces.

B. Adaptive Pagination Model

In an adaptive pagination model(APM) the reorganization decision is taken deterministically using segment size estimates. Aiming to achieve balanced 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. The APM rules are as follows:

1) if $Size_S < M_{min}$, the segment is left intact.
2) if $Size_S > M_{min}$ the segment is reorganized if all of its sub-segments have estimated size above M_{min}.
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 are evidences that the segment might be queried again in the near future. The segment is reorganized if $Size_S > M_{max}$ choosing a splitting point among the query bounds or an approximation of the mean value in the segment.

A characteristic of the APM scheme 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.

V. Evaluation

Experiments are run on a two Dual Core AMD Opteron(tm) Processor 270 2GHz with 8GB memory using a 100GB sample of the SDSS-4 database. This fits on a simple desktop PC, but certainly makes the database disk bound on most queries.

The test queries select ranges over a BAT RA. Three workloads, 200 queries each, were extracted from the SkyServer log: RANDOM picks 1 out of every 300 queries and covers the attribute domain uniformly; SKEW extracts 200 subsequent queries from the log which access two very limited areas of the domain, and CHANGING consists of four pieces of 50 subsequent queries with changing point of access.

Figure 1 shows the average time spent in adaptation vs. selection after the first 200 queries. For all workloads the adaptation overhead for the APM schemes is smaller than for Gaussian Dice. The former one is more conservative in splitting small segments. Similarly, the overhead for APM1-5 is bigger than for APM1-25 due to splitting when a small range is selected out of a segment with size $5MB < Size_S < 25MB$.

Due to the smaller upper bound M_{max} the APM1-5 scheme creates smaller segments than APM1-25 as shows the segment statistics in Table I. Smaller segments give bigger gain from saved scanning as illustrated by the reduced selection times in Figure 1.

Figure 2 shows the cumulative query times for the adaptive schemes compared against non-segmented database. For all workloads the initial overhead for APM reorganization slows

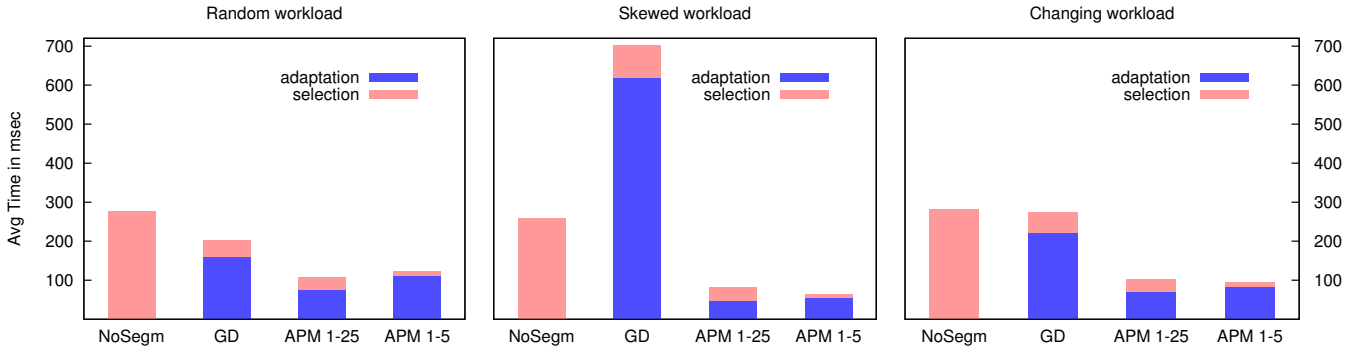
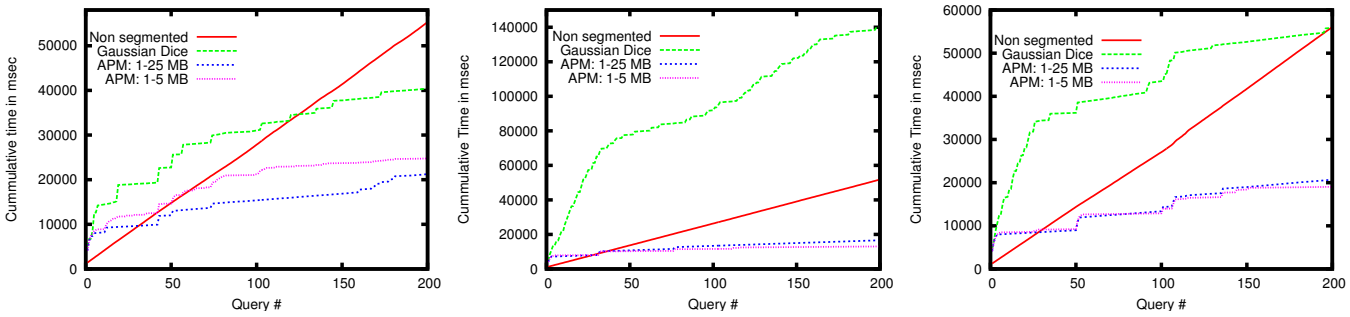


Fig. 1. Times for adaptation and selection



(a) Random workload (b) Skewed workload (c) Changing workload

Fig. 2. Cumulative times

TABLE I

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

down the first queries but provides better system response after a relatively small number of queries. The total overhead of APM schemes for skewed workload (Fig. 2b) is smaller than for random load, since the reorganization affects a very limited area of the domain.

The GD scheme shows bigger overhead than APM with worst performance for skewed load. The skewed queries hardly differ in their selection predicate and chop very small pieces. As a result 80% of the segments contain less than a 1000 tuples, which are expensive to reorganize and in addition incurs gluing of small pieces for the subsequent queries.

The performance for changing workload in Figure 2c illustrates how shifting the point of query interest triggers reorganization of untouched segments. It results in a temporal increase of the overhead after queries 50 and 100, which saturates soon after too.

VI. CONCLUSIONS

Scientific databases require efficient database segmentation for their ad-hoc workloads and high performance exploratory queries. Unfortunately, these environments often set aside limited resources for (human) data management. Self-organizing databases have become a must.

In this paper we have demonstrated how self-organizing database algorithms find their natural embedding in the tactical optimizer layer of MonetDB. The evaluation of sample policies against the SkyServer application shows that the overhead in data administration is negligible and that query performance drastically improves on real-life query streams.

REFERENCES

[1] (2007) MonetDB. [Online]. Available: http://monetdb.cwi.nl/
[2] M. Stonebraker *et al.*, "C-Store: A Column Oriented DBMS," in *Proc. VLDB*, 2005.
[3] R. MacNicol and B. French, "Sybase IQ Multiplex - Designed For Analytics." in *Proc. VLDB*, 2004, pp. 1227–1230.
[4] S. Idreos, M. L. Kersten, and S. Manegold, "Database Cracking," in *Proc. CIDR*, Asilomar, CA, USA, January 2007.
[5] K. Sattler, E. Schallehn, and I. Geist, "Autonomous Query-Driven Index Tuning," in *Proc. IDEAS*, 2004, pp. 439–448.
[6] K. Schnaitter, S. Abiteboul, *et al.*, "COLT: Continuous On-line Tuning," in *Proc. SIGMOD*, 2006, pp. 793–795.
[7] N. Bruno and S. Chaudhuri, "An Online Approach to Physical Design Tuning," in *Proc. ICDE*, 2007, pp. 826–835.
[8] (2007) Sloan Digital Sky Survey / SkyServer. [Online]. Available: http://cas.sdss.org/dr6/en/
[9] M. Ivanova, M. L. Kersten, and N. Nes, "Self-organizing Strategies for a Column-store Database," in *Proc. EDBT*, 2008, to appear.