Synopsis: Hybrid Stores - Partitioning into Rows and Columns

Holger Pirk <holger@airbug.de>

1 Motivation

The real execution time of a database query is not only determined by its theoretical complexity. When run on a real computer other factors like the hardware and the specific implementation are of influence. It has been shown that within the implementation-factors the memory access pattern has the most influence [4]. This is due to the fact that required data is not read byte-wise but block-wise (harddisk blocks, memory pages or cachelines). Even of the blocks some are faster to read than others due to strategical prefetching (mostly based on colocation).

Organizing data in a way that makes it fast to read by an algorithm is a challenging but profitable task. In a database context this is usually done by an experienced administrator (DBA). He needs a lot of knowledge of the internals of the particular DBMS to make qualified decision on different optimization techniques (indices, materialized views, partitioning). He also has to anticipate the usage (OLAP or OLTP) of the database since different usages make different algorithms necessary which in turn show different access patterns.

Tools to support the administrator with these decisions are part of many commercial DBMSes (Microsofts Autoadmin, IBMs Design Advisor). Yet their optimization options are always limited the capabilities of the underlying DBMS. A DBMS with new capabilities is currently developed by the Hasso-Plattner-Institute: It has the ability to store relations as row- as well as column-based [1] – a hybrid DBMS. It also provides the option of vertical partitioning with each partition row- or column-based which is, to our best knowledge, yet unrivaled. This increases the space for potential optimizations but also makes the decisions on them much harder. An automated way to decide on a good layout for the needed data seems valuable.

The potential for optimizations using rowstores vs. columnstores is significant. *Fractured Mirrors* [5] take advantage of this for read-mostly workloads by redundantly storing data in rows and columns. This comes at the cost of an increased memory footprint as well as increased update costs. These are acceptable for read-mostly workloads but become a critical problem with an increasing number of Updates/Inserts.

When focusing on *Operational Reporting* [3] this is exactly what we have: A large number of inserts and analytical queries that have to represent up to the second information. In this scenario of mixed workloads, *Fractured Mirrors* are of little use.

We believe that we can reproduce some of the benefits of *Fractured Mirrors* while reducing the drawbacks by non-redundantly partitioning the schema into row- and column-based partitions. We also believe that we can automate this optimization by analyzing a workload sample (an SQL-trace with weighted queries).

2 Goal

We want to show the potential of non-redundant partitioning in hybrid DBMSes. We want to compare it to the common cases (all row-based and all column-based) as well as with the optimal case of *Fractured Mirrors*.

We will implement an algorithm that, based on a given workload, automatically partitions a schema into row- and column-based partitions. It will estimate the costs of each representation and can thus be used to estimate the benefits of the partitioning.

3 Agenda

Creating a realistic workload is not part of the thesis and we will therefore rely on external sources (HPI, SAP) to provide it.

The thesis will tackle two major problems:

3.1 Finding a good representation for every relation in a schema

We will start out by aproximating the optimal representation for every relation in a given schema by

Calculating the operator-tree for each Query: We will implement a basic SQL-compiler to calculate the relational operator-tree for each Query.

Defining Operator Access Patterns: For each used operator we will have to **manually** define the memory-access-pattern for each representation. We will pick an implementation for every operator, analyze it and describe its access-pattern as a combination (parallel or sequential execution) of basic access patterns like "sequential traversal", "random traversal" or "random access". We can then use the model from [4] to estimate the overall costs of a query.

Calculating the Plan's Access Pattern: We will automatically combine these operators to an access pattern for the plan. Remember that since we only consider single-join-queries, two relations can be involved at max. Since each relation can be stored either row- or column-based we have four access patterns to consider.

Estimating the Plan's Costs: We will evaluate the costs for each representation for each query using the model from [4]. We hope to re-use the implementation of the author.

Finding a good Solution: Finding a good representation for the overall workload is minimizing $\sum_{\text{query} \in \text{workload}} \text{query}_{\text{costs}} \cdot \text{query}_{\text{weight}}$. This is a linear programming problem – methods for solving it exist.

3.2 Proposing a good Partitioning

The second step is to implement an algorithm that automatically proposes a vertical partitioning for a given workload.

Partitioning basically has the advantage of reducing the number of uselessly read bytes when scanning and the disadvantage of an additional lookup for each tuple that has to be accessed from two partitions.

To find a good partitioning for a schema we will have to

Implementing the OBP-Algorithm: The *"Optimal Binary Algorithm"*[2] partitions a given schema based on a workload. It groups attributes that are accessed together into partitions, scaling exponentially with the number of transactions. As a first step, we will implement this algorithm to find a partitioning.

Extending the OBP-Algorithm: OBP does not take the access pattern on attributes into account. We want to extend the algorithm by grouping attributes that are not just accessed together but also accessed in an equal pattern. Reusing our previous findings we expect this to be fairly straight-forward.

3.3 Evaluate the Gains

Finally we want to evaluate the performance gain of our optimization. Based on the progress of the hybrid DBMS-Project we will do this theoretically using our own model or practically using the hybrid DBMS. We are aware that a theoretical evaluation is far less convincing but is still reasonable since the model it is based on [4] is solid.

References

[1] Daniel Abadi, Samuel Madden, and Nabil Hachem. Column-stores vs. row-stores: How different are they really? In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 2008.

[2] W. W. Chu and I. T. Ieong. A transaction-based approach to vertical partitioning for relational database systems. *IEEE Trans. Softw. Eng.*, 19(8):804–812, 1993.

[3] Bill Inmon. Operational and informational reporting. DM Review Magazine, 2000.

[4] Stefan Manegold, Peter Boncz, and Martin L. Kersten. Generic database cost models for hierarchical memory systems. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 191–202. VLDB Endowment, 2002.

[5] Ravishankar Ramamurthy, David J. DeWitt, and Qi Su. A case for fractured mirrors. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 430–441. VLDB Endowment, 2002.