A Plan for OLAP

Bernhard Jaecksch
SAP AG
Dietmar-Hopp-Allee 16
Walldorf, Germany
b.jaecksch@sap.com

Wolfgang Lehner
SAP AG
Dietmar-Hopp-Allee 16
Walldorf, Germany
wolfgang.lehner@sap.com

Franz Faerber
SAP AG
Dietmar-Hopp-Allee 16
Walldorf, Germany
franz.faerber@sap.com

ABSTRACT

So far, data warehousing has often been discussed in the light of complex OLAP queries and as reporting facility for operative data. We argue that business planning as a means to generate plan data is an equally important cornerstone of a data warehouse system, and we propose it to be a first-class citizen within an OLAP engine. We introduce an abstract model describing relevant aspects of the planning process in general and the requirements it poses to a planning engine. Furthermore, we show that business planning lends itself well to parallelization and benefits from a column-store much like traditional OLAP does. We then develop a physical model specifically targeted at a highly parallel column-store, and with our implementation, we show nearly linear scaling behavior.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Parallel databases*

1. INTRODUCTION

One of the main purposes of a data warehouse clearly is to provide insight into historical data, i.e. operational data that has been accumulated up to the current point in time. Yet, there is another cornerstone to data warehousing commonly used in many businesses that, to our knowledge, has not attracted too much attention in the research community so far. Planning is a vital instrument for companies to draw a detailed picture of their expected operational business and to use the plan as a means for documentation and comparison of their success. There exist various models describing multidimensional data structures for data warehousing as well as a number of operators working on these data structures that provide the means for querying the data on every possible level [3, 4]. We argue for the need to extend the model and the set of operators and therefore make business planning a first-class citizen in the field of data warehousing. The basic idea of business planning is to iteratively create new data to describe the expected outcome of forthcoming periods. Often, the newly created data is based on historical data and is then modified and transformed to model future business operations. Planning is an iterative process between, for example, sales managers at different levels in an organization. There is a manager who plans a budget on division level, and there are regional managers who use the given budget for planning in their specific region. This top-down process can then run bottom-up and down again, where the final plan is shaped during an iterative refinement. Having multiple persons working on a plan also shows that planning often is a collaborative process.

EDBT 2010, March 22–26, 2010, Lausanne, Switzerland.

1.1 Query-Modify-Publish paradigm

By comparing a typical OLAP session with a planning session, two contrasting paradigms can be found. OLAP follows the classical Query-Response model. During an OLAP session, the user poses queries to drill down or through data, move to different levels in the aggregation hierarchy, and add or remove filter criteria. In contrast, we identified a new ***Query-Modify-Publish*** paradigm for planning applications. In a typical planning session, users select source data on specific levels of aggregation very much like in a usual OLAP session. However, they then modify and transform this data in a number of steps until it matches their expectations, and finally, they publish it into the system again, making it visible to others. So, in addition to querying, which the new QMP paradigm shares with the traditional QR approach, it goes far beyond it when existing data is modified and new data is created.

1.2 Architectural considerations

There are already a number of existing commercial business planning systems, which all share more or less the same architecture having a traditional relational database system as foundation. Between the user application and the database, there is a planning engine that interfaces with the database via SQL and separates the planning logic from the database. The application itself connects to the planning engine either via a proprietary API or possibly via an MDX-like language. In contrast, we propose a clearly structured layered approach that is oriented at the previously identified QMP paradigm and has the advantage of special support for the specific requirements of planning. The general requirements of planning are subsumed in an abstract way on the first model layer called the General-Planning Model (GPM). For the following two layers below, we suggest a classical divide into an operational model side and a storage model. First, with the operational model, a broad set of planning functions

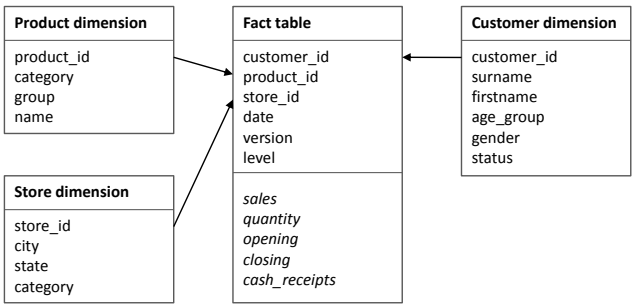

Figure 1: Schema of example cube

should be expressible and it must also incorporate means to formulate typical OLAP queries. Second, as a storage model, we use a column store because of its flexibility. It allows to easily add or remove columns, which is often necessary for intermediate results, and its independent storage of individual columns is often a benefit in OLAP scenarios. Operations often affect a small number of columns instead of entire tuples. Therefore, a column-store needs to touch less data than a row-store does when executing these operations. Additionally, if planning operators work on different independent columns, they lend themselves well to parallelization. However, using a column store mainly affects the implementation on the physical layer and is no exclusive requirement for our planning engine. Third, both layers should be tightly coupled and implemented within the same engine. Since business planning can often be a very data-intensive process, it clearly benefits from moving the logic close to the data, which avoids having to shuffle around large data sets between different engines and system boundaries. Also, the control flow in a planning process is interwoven very tightly with the data flow; therefore, keeping both together in one engine also allows for optimization of both. In the following sections, we will outline specific details of such an implementation within a column-store, namely the SAP Business Warehouse Accelerator (SAP BWA) [2].

The next section emphasizes the particularities of business planning with examples. Section 3 outlines a column-centric logical calculation model that contains specific planning operators making it possible to express a comprehensive set of planning functions. Section 4 introduces an example implementation and concludes with results that show the scalability of our approach.

2. EXAMPLES

We begin with a list of examples and suggest a set of high-level operations that are necessary to cope with the typical challenges that business planning poses to an OLAP engine. Throughout the examples, we refer to the cube with the schema in Figure 1.

Example 1

Most planning sessions start with a copy of existing data. In our example, a planner wants to plan sales of products for different stores for the year 2010. He wants to plan independent of single customers; hence, customer is excluded from the aggregation level. The planner also wants to plan only those products where sales were greater than or equal to 1,000 in the current year. After copying the source data, the planner wants to introduce a new product for the year 2010 based on an existing product and its sales values.

During each session, the planner chooses **multiple levels of aggregation** where planning takes place. That means measures are changed and manipulated that have no direct representation in the underlying cube but represent an aggregated view on the facts. For the above example, this involves aggregating the source data over the customer dimension, filtering all tuples with sales values less than 1,000, and then changing the year dimension from 2009 to 2010. On every level of aggregation, the **generation of new values** can occur. With new values, we refer to combinations of dimension values as new facts. New facts are either created by adding a new unique combination of existing dimension values or, for one or more dimensions, the domain of this dimension is extended with new values. In the former case, such a translation often takes place along the time axis, like in the first part of the example. For the latter case, as in the second part of the example, the planner has to select an existing product and map the name to the new product name that has been added to the product dimension. The planner could also choose to stop selling a product. Therefore, **deletion of facts** is required, too. This, for example, is a requirement that is not present at all in an OLAP environment, and it shows that planning functionality is a superset to standard OLAP functionality.

Example 2

In a next step, a planner wants to plan sales quantities for a number of products for the year 2010, additionally including the newly added product. Instead of planning the quantities for each product individually, which can be cumbersome or impossible if there is a large number of products, the quantities are entered on the aggregated product group level or on state level and should get distributed to each product or each city.

There is a requirement for the **distribution of input values** to more granular levels of aggregation and ultimately down to fact granularity. Distribution or disaggregation is somewhat similar to the drill-down operation in queries, but instead of just selecting values on a finer level, values need to be adjusted according to a distribution function to accumulate the aggregated values entered by the planner.

Example 3

In a last step, the planner wants to calculate opening and closing sales values as well as the amount of cash receipts over a time period of twelve months for all stores. The planner copies the closing value of January to the opening sales value of February. Then, he calculates the closing sales of February based on the sales of the current period multiplied by a fraction of the sales from January. This calculation is then continued for subsequent periods where each value in the current month has a dependency calculated for the previous month.

Next to basic planning functions like copying or distributing aggregated values, planning also involves **complex calculations**. Since the process of planning is very dynamic, a generic and extensive way of expressing formulas must

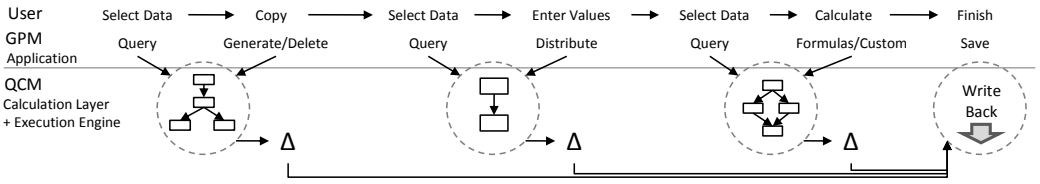

Figure 2: Example planning process with General-Planning Model and Query-Calculation Model

be provided. Finally, most of the aforementioned examples changed measures of the underlying cube or added new facts. **Write-Back** of changes and generated data to the multi-dimensional cube is implicitly required by all previous examples.

All the examples dealt with the manual type of a planning process. Manual planning is determined by interactions with a user, where a session follows the introduced *Query-Modify-Publish* paradigm. It can be compared to a long-running transaction that is finished when the user issues a save or quit command. During the session, the user has the possibility to apply different modifications to the data and to eventually undo the steps taken so far. Another type of a planning process is batch planning. In contrast to the interactive approach, batch planning executes a defined series of plan operations in a given order. Since the order of execution and the steps involved have been parametrized and specified beforehand, there is no need for an undo or user interaction, and it opens the opportunity for optimization.

A number of typical requirements for a planning engine emerge from a planning process, as shown in Figure 2. First, there are typical OLAP query operations for the selection of data required, like slicing, dicing, filtering and aggregation. Furthermore, there are planning-specific requirements, as explained in the previous examples. They include: **generation** or **deletion** of values, **distribution** of aggregated values to more granular facts, calculation of arbitrary complex **formulas** on the planned data and **saving** it to a cube. Of course, this cannot be a complete list, so **custom** functions allow for further extensions. Together with the OLAP query requirements, we subsume these challenges for a planning engine in the following by the term **General Planning Model** (GPM). The GPM forms the top of three model layers that we use to describe the planning process. On the second layer, a platform-agnostic model is positioned that is powerful enough to express any of the required functionalities from the GPM. We term this the ***Q**uery-**C**alculation **M**odel* (QCM) and explain it in detail in the next section. The implemented execution model, not shown in the process, is the one of the SAP BWA execution engine, which will be described briefly in Section 4.

3. QUERY CALCULATION MODEL (QCM)

Before introducing the entities contained in the Query-Calculation Model, we describe the expression of one of the example planning functions in the previous section.

3.1 QCM Example

One of the identified requirements in Section 2 was the ability to distribute values from higher levels of aggregation to lower levels. A planner wants to plan sales quantities for the upcoming period on product group level and distribute the aggregated quantities. As distribution factors, for example, we could use the sales quantity summed up over all previous years. In principle, this involves for each product the calculation of the ratio between its value and the total sum so far, followed by its multiplication with the new aggregated value. Since quantities most likely are integer numbers, rounding errors would be introduced by this naive method. Therefore, special distribution algorithms exist, which deal with rounding errors. One possibility would be to express a distribution algorithm directly within the QCM. Although our QCM has enough expressive power to model the distribution algorithm itself through the use of the *foreach* operator and *column expressions*, it makes sense to provide a special *column function* for it because distribution was identified as one of the key functionalities that is used during planning. Therefore, the example in Figure 3 makes use of a dedicated distribution column function that is part of the model. The column function is executed for each product group only, which means that the *Dist* function is called once on each set of products that belong to a product group. Before executing the distribution function, the input values for each product group are joined to the target data. The result of this operation forms the input of the *foreach* operation that executes the distribution function. With the help of this example, we also introduced a graphical notation for the model displayed in Figure 3. A *column set* is represented by a box whose name is the operator that is applied to the (base) columns. On the bottom of the box, the input columns are specified, and the link to the source columns shows from which *column set* or *column source* they have been obtained. Likewise, the top of the box describes the output set of columns that are visible to any other column

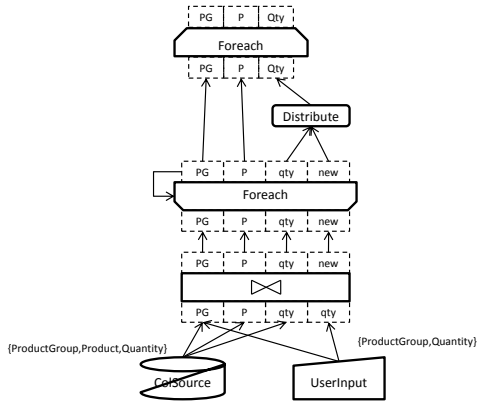

Figure 3: QCM for distributing values using the Distribute column function

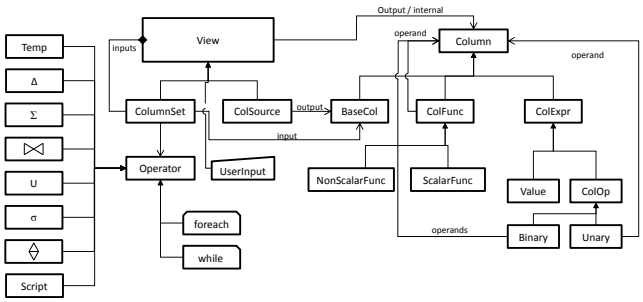

Figure 4: Schema of the Query-Calculation Model (QCM)

set. Column expressions are drawn as expression graphs on top of columns. An empty expression means that the input column is routed right through to the output column. Otherwise, the output column is the result of the column expression. Multiple expressions can refer to the same input column but produce different output columns, therefore allowing easy duplication of columns. Furthermore, there is the notion of an internal column that references one or more input columns but is not available as output of the column set. It provides a means to express internal state variables during the calculation of expressions.

3.2 QCM

The central idea of the QCM is to have column-based views that can be combined with each other through the use of operators to form more complex views. In principle, a view in the QCM is very similar to a database view. However, the use of columnar data structures and its focus on column operations differentiates it from a database view and the typical relational model. It allows the QCM to benefit from the advantages that a column-store has over row-store in OLAP query scenarios [1], and the planning functions themselves profit from it, too. Taking the previous distribution example, multiple column functions can be executed in parallel on a column set, thus distributing to multiple facts in one pass. Also, the easy addition of new columns or the deletion of unused columns makes the column-centric QCM views much more flexible. For a QCM view, there is the distinction between a *ColumnSet* and a *ColumnSource*. A *ColumnSource* represents either base columns from the underlying data at the most granular aggregation level, i.e., dimensions and keyfigures, or a set of input values that are entered by a user during a planning session. A *ColumnSet* can be an arbitrary set of columns at every possible aggregation level containing base columns as well as calculated columns. This conforms well with the previously mentioned requirement that planning takes place on various levels of aggregation. Stacking different views onto each other forms a directed acyclic graph with the leaf nodes always being source columns or inputs, and the links between the views symbolize the flow of data starting at the leaf nodes. So, the key feature of the model is its ability to express all types of OLAP queries and additionally allow sophisticated calculations to change column values or add new columns on each view.

3.2.1 Column set operators

There are a number of basic operations like *Grouping*, *Union*, and *Join* that work on complete column sets and are basically the same as their namesakes in a relational database. However, we extend the standard set of operators with two control-flow operators, *Foreach* and *While*, both working on column sets, too. The *Foreach* operator has the semantics to perform calculations for each combination of column values. Additionally, it also allows the definition of a context. The use of a context partitions the data in a *ColumnSet* and performs the *Foreach* loop for every partition separately. Thus, the context also serves as a hint for parallelization to the underlying execution engine. As described in Section 2, historical data is often modified to generate new facts. For these scenarios, the *Foreach* operator is well suited. Nevertheless, there are cases where entirely new data has to be generated that cannot be obtained by modifying existing data. In that case, the *While* operator steps in and allows iterative calculations and the application of column functions that produce new values. The operator starts with an initial state and proceeds until a condition is matched.

The following three elements are a core feature of QCM to express the iterative planning process as described by the GPM with undo steps and temporary modifications that are saved at the end of a session. To temporarily materialize the result of a view, the *temp* column set operator can be used as root node in the QCM graph. Optionally, the *delta* operator calculates for each measure of its input the difference to its value in the source cube. Temporary results containing deltas combined with the actual source cube represent the current planning results during a user session. By deleting these intermediate results, the application can easily undo the effect of a planning step during the planning process. An explicit *save* call aggregates all temporary deltas and merges them into the source cube. The *Script* operator allows custom scripting within a calculated column set and serves as an extension point for calculations that are beyond the scope of the model. Finally, there are filter and sort operators. Depending on their position in the data-flow graph, they serve as input filters and output filters that allow controlling the data that is consumed by a column set operation and selecting the data to be passed to subsequent views after the calculations. The sorting, as the name implies, can be used to apply orderings to the set of columns.

3.2.2 Column functions and expressions

So far, column sets and operators that work on column sets have been discussed. Through the innovative concept of column functions and column expressions, the model introduces very powerful calculation capabilities. As can be seen in Figure 4, the notion of a column has several different meanings. Each input column of a column set is called a *BaseColumn*, regardless of whether it comes from a column source or is a calculated column that is the output of another column set. Within each column set, *ColumnFunctions* can be applied to columns returning another column or a scalar value. Column functions are comparable to set functions in the Multidimensional Expression language MDX. Within this analogy, the input set of a column function is the set of values from the input column. Typical scalar column functions are *SUM*, *AVG*, *MIN*, *MAX* or *COUNT*, whereas non-scalar functions are *Filter*, *Range*, *Sort* or *Distribute*. What makes this con-

cept truly useful is the possibility to nest column functions such that one function is applied to the result of another function. Together with the fact that column functions are evaluated for every iteration of the *Foreach* and *While* operators, they provide a powerful means to calculate complex expressions.

Aside from base columns and column functions, columns can be modeled as the result of a column expression. A column expression uses arithmetic and Boolean operators that combine one, two or more columns. There are common arithmetic operators provided like $+, -, *, /$ as well as Boolean operators like $AND, OR, NOT, <, >, \leq, \geq$ etc. Since each column in a column set has the same number of values or is a scalar value, the meaning of an expression like $column_1 * column_2$ is straightforward: every value c_{1i} of $column_1$ is multiplied with value c_{2j} of $column_2$ with $i = j$ and $i, j = 1..size(column_1)$. For a column and a scalar value, every value of the column is multiplied with the scalar. Boolean expressions behave the same way and are evaluated for each pair of values from both columns.

4. EVALUATION

The aim of this section is to verify by experiments that the implementation of the QCM within the SAP BWA engine shows excellent scaling behavior. Furthermore, we show that the proposed planning model benefits from the implementation on top of a column-store in comparison to a traditional row-store.

4.1 Implementation

After having introduced the capabilities of the QCM, we explain the implementation of the QCM on the SAP TREX execution engine with the help of the previously explained *Distribute* function in this section. The TREX execution engine differentiates between plan data and plan operations. Each plan operation can run independent of another and in parallel and starts as soon as all data inputs are available. An execution plan has some plan data inputs and outputs. It is a data-flow graph where each plan operation is connected to another plan operation via its inputs and outputs - very much like the QCM model itself. Typically, the physical representation of a calculation view from the QCM consists of one or more plan operations and plan data. For example, for each of the column set operators, there exists a plan operation. However, since the BWA's main capability is fast querying and aggregation, these functionalities are reused within the execution model. Therefore, instead of a stack of filtering, sorting and grouping operators, alternatively a *search* operator is available that combines these operations into one and leverages the power of the highly parallel BWA engine. In fact, the *search* operator is used whenever a column set should be retrieved from an OLAP column source. It takes as plan data input a query and a reference to the cube that is the target of the search. It then outputs a column set data structure that can be used by all subsequent operators. So-called *InternalTables* are used as columnar data structure within the TREX engine representing *ColumnSets* of the QCM. Another vital part of the QCM is the various column functions and column expressions. Column functions are implemented to work directly on single columns of an *InternalTable*. In our example, the distribution function is one such column function that is implemented on an *InternalTable* column. Column expressions are evaluated for each value combination that exists in the columns specified as source for the iteration, i.e they are evaluated row-wise.

Often, there are different physical execution plans possible to express a functionality modeled in QCM. We use the example of the *Distribute* function again to explain an execution plan. As mentioned before, the *Search* operator delivers an *InternalTable* that is then joined with the column that contains the new summed-up quantities on product-group level supplied by the user. The joined column set then serves as input for the *foreach* operator and contains the sales column that is the target of the disaggregation. The output of the *foreach* operator is again an *InternalTable*, which can be stored as temporary result using the physical *Temp* operator. As is signaled by the context argument of the *foreach* operator, we distribute not one value but one value per product group, i.e. data partition. Thus, the disaggregation function can be executed for each set of products independently and in parallel. The data partitioning via the context of the *foreach* operator can also be used to parallelize the calculation from Example 2. This is also demonstrated in the scalability evaluation in Section 4.3. Referring to the distribution example again, there is another option to introduce parallelism. If the value for a product group is spread to a very large set of products, it can be useful to split the set of products into multiple parts. Then, for each part, an intermediate sum can be calculated that determines how much all products from that part contribute to the product group sum. Now, the overall sum is allocated to those partial sums building a set of new partial sums. The distribution of these new intermediate sums to individual products can now be executed in parallel again. This form of hierarchical distribution can also be combined with the previously described parallelization of the distribution of multiple input values.

For the evaluation, we use the sequence of planning steps that was explained in the introductory examples. The planning session contains two copy steps COPY1 and COPY2, the distribution of values DIST and the calculation of a complex formula CALC. For the comparison of row- vs. column-store, we used a dual CPU workstation with 4GB of main memory running Windows XP 64Bit.

4.2 Column-Store vs. Row-Store

With the first experiment, we wanted to evaluate the suitability of an in-memory column-store as a planning engine and compared it to a traditional row-store and an in-memory row-store. The row-store is a well tuned commercial database system and the functions have been implemented as stored procedures using procedural SQL. For both systems, the size of the source dataset is chosen such that it fits completely into the memory of the workstation and has $\approx$ 1.000.000 rows. The size of the selected and manipulated data set has $\approx$ 250.000 rows. Since each planning step during a planning session produces a temporary result, we used temporary tables for the row-store to store the results, in order to minimize the influence of logging and make it comparable with the temporal column sets of the column-store. Furthermore, before the measuring, we loaded the data into memory to minimize disk access. All these actions should ensure that

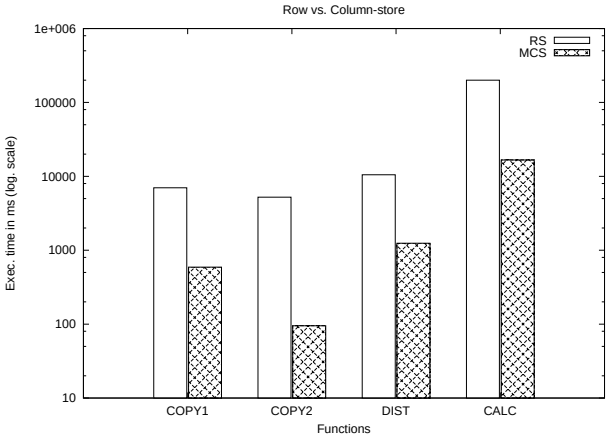

Figure 5: Comparison of Row-Store (RS) and In-Memory Column-Store (MCS)

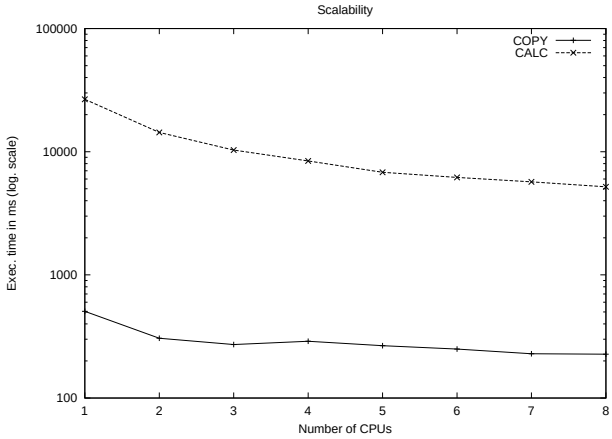

Figure 6: Scalability of the Copy and Calculate function on BWA

we compare the two implementation techniques and data representation paradigms rather than doing a comparison of disk vs. memory. Figure 5 shows the results of the first measurements. Overall, the measurements show for each planning function that the BWA engine is at least 10 times faster than the row-store. Also, on the row-store, the CALC function consumes a considerable amount of time.

4.3 Scalability

For the scalability tests, the experimental setup uses a server blade with 8 CPU cores and 8GB of memory running 64-Bit Linux. To simulate the scaling, we limited the number of execution units in our physical execution plans to the number of cores simulated. To set a baseline, the plan was executed by one execution unit only. Subsequently, we distributed the execution to 2, 3 and up to all 8 CPUs. Figure 6 shows the reduction of execution time compared to the baseline for the COPY and CALC functions. As can be clearly seen, the scaling for both COPY and CALC functions is close to linear and conforms with our expectations. Obviously, optimal scaling can only be achieved if the execution plan can be well parallelized. However, this is most often the case with planning functionality that is expressed with our QCM. For example, partitions of data that follow from the *foreach* operator or independent calculations on different columns are always well suited for parallelization. Since these mechanisms are inherent to the model, it is very suitable for the parallel execution infrastructure of the SAP BWA.

5. CONCLUSIONS

In this paper, we argued for planning functionalities to be treated as first-class citizens in the realm of OLAP modeling and engines. We summarized the main requirements that a planning application poses to an underlying engine and identified a Query-Modify-Publish paradigm that most planning applications follow. To our knowledge, we introduced for the first time a model that is powerful enough to express many of the required planning functions. One key property of the model is that it is column-centric. Hence, it is well-suited to be implemented on a column-store, as was shown in the paper. It benefits from the column centricity for the very same reason as OLAP engines benefit from it. Furthermore, we showed with an example that the model also lends itself well to an implementation on a parallel processing engine, and we showed the scalability of this approach on the SAP Business Warehouse Accelerator, a distributed query and processing engine. There are a number of issues for future research. First, the translation between the QCM to the execution model bears a lot of potential for possible optimization, so an optimizer between these two layers seems an interesting and promising thing to look at. Furthermore, this optimizer can also use parallelization hints within the QCM to create well parallelized plans. We should also mention that some of the planning functions have been implemented in Python for easy prototyping. Implementing them with the BWA's native implementation language (C++) will result in further speedup. Another idea is to omit the step of saving and materializing the intermediate results of single planning steps. Instead, the engine could only save the actual QCM, and whenever the data is requested, calculate the requested results on the fly. By stacking the models from multiple steps on top of each other, again, the stacked model provides potential for optimization.

6. ACKNOWLEDGMENTS

We would like to thank the whole TREX team for their invaluable contributions and their continuous support.

7. REFERENCES

[1] D. J. Abadi, S. R. Madden, and N. 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*, pages 967–980, New York, NY, USA, 2008. ACM.

[2] R. Burns and R. Dorin. The sap netweaver bi accelerator - transforming business intelligence, white paper, winter corporation, 2006, http://www.wintercorp.com/whitepapers/whitepapers.asp.

[3] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *ICDE*, pages 152–159, 1996.

[4] P. Vassiliadis and T. Sellis. A survey on logical models for olap databases. *SIGMOD Record*, 28:64–69, 1999.