Dynamic Partitioning for Enterprise Applications

Martin Grund, Jens Krueger, Juergen Mueller, Alexander Zeier, Hasso Plattner
Hasso Plattner Institute
14482 Potsdam, Germany

Abstract—Today’s enterprise applications face a severe change in how they process data. They evolved from simple data entry applications to complex systems where the focus is to make the right decision based on latest transactional data instead of pre-processed outdated business intelligence data from secondary systems. This paradigm shift results in more analytical queries executed on the transactional database system. In addition we see a change in how the persistence layer is seen and that more and more effort is spent on developing database engines that are designed to address special needs and requirements, e.g. for Web 2.0 applications.

In this paper we discuss that it is crucial for enterprise applications to share application semantics with the database to increase performance. Furthermore we show how this principle can be applied to dynamically partition application data. In contrast to static partitioning we propose to adapt the life-cycle semantics of the application and move data from active to different passive partitions, by leveraging the merge process of our database prototype. The goal is to reduce the amount of data that is touched during query execution to increase the performance of the main workload by the same factor. We present how our HYRISE prototype is implemented and how it can be extended to fully support this use case.

I. INTRODUCTION

In the last years the definition of enterprise applications evolved. Inspired from Web 2.0 and the recent trend in business intelligence the requirements for enterprise applications changed. When we are now talking about transactional processing it no longer covers only the streaming of new data into the system, but building of an analytical context to fulfill a certain task. For modern enterprise applications it is crucial to support the decision making process with up-to-date analysis on transactional data [10]. E.g. during creation of a new opportunity it is important for the sales clerk to understand what the history of the customer is.

To second the assumption that more and more read queries are executed in enterprise applications, we analyzed workload data of an ERP system. Our analysis of a real SAP customer ERP system shows that already today about 84% of all queries are read queries, which is close to the number of read queries executed in an OLAP system. Figure 1 shows the query type distribution for the analyzed system, once for OLTP and for OLAP. Compared to the traditional TPC-C benchmark this is 63% more read queries as the TPC-C defines that 53% of the queries are read queries.

Due to the increased complexity of the queries (more aggregations and groupings) the amount of data that needs to be processed increases as well. To compensate this development we propose to vertically and horizontally partition the data based on the application defined life-cycle of the data records. In addition, research in the area of compressed column databases [1], [11] has shown that compression can be used to better utilize the available bandwidth and achieve higher throughput. However, using only compression neglects the fact that we can achieve even better performance if only a relevant subset of the data has to be scanned sequentially. Even though only a fraction of the active data is used for day-to-day processing we want to keep the passive data available so that we can leverage it for analytics.

Active and passive data have different requirements: while active data is often modified and single rows are retrieved to be displayed and processed by the application, passive data can be further compressed, vertically decomposed and sorted to achieve best performance for analytical queries. But since both data is used in one process it is required to keep both in one system but still reflect their requirements.

Our core contribution is a concept to use life-cycle data from the business application to partition the transactional data. In contrast to current systems our system does not expect that one record will always belong to one partition, but will be moved from one silo to another once it moves to a different state in the life-cycle of the object.

Moving data dynamically to different partitions was not possible until now, but due to our experience in compressed main memory database engines, we explore to leverage the available infrastructure to re-compress new data into a main partition to implement a dynamic partitioning system [11], [6]. As a result we can severely decrease the amount of data processed by the queries in average using a life-cycle aware setup.

The remainder of this paper is structured as follows: In Section II we will explain why the life-cycle of a business object can be used to dynamically horizontally partition the data set in an enterprise application. Section III describes
The current architecture of our prototype HYRISE [6] and which extensions must be made, Section IV explains the implications of dynamic partitioning and how it affects the current implementation. We finish the paper with a comparison of related work in Section V and a conclusion in Section VI.

II. OBJECT LIFE-CYCLE

Every enterprise software bases its implementation on the business processes executed to fulfill a certain goal. To be able to determine which step to execute next in the business process status variables are introduced. Based on those status variables it is possible for the enterprise application to compute which action can be executed next and even which action is allowed to be executed. Undefined behavior with state transitions which are not defined in the original business process cannot be executed in the application and lead to process exceptions.

However, on the database side this relationship is neither known nor leveraged. In addition to semantic properties which can be relevant for transactional purposes the status fields open an interesting possibility for partitioning.

The primary goal of state-based partitioning is to differentiate data based on a business decision instead of a criterion that is solely based on the cardinality of a certain value in the database (e.g. the different warehouses in TPC-C.) In this assumption most of the business processes in an enterprise system work with active data. In a business process active data is all data belonging to those states that are not final.

Figure 2 shows an example for the definition of the “Lead” Business Object with it’s dependent states and actions. A “Lead” is a potential new customer for a company that might be interested in certain products or services. Such a proprietary model can be transformed into a simple state transition diagram, where each place maps into state and the transition defines which action can move a tuple to a different partition. The goal is to combine data from multiple objects so that all decisions regarding the partition membership are unambiguous and the graph is machine interpretable. An example for this transformation is Figure 3. The different status fields are now aggregated and mapped into partitions and give a better interpretable understanding where which data belongs.

![Fig. 2. Status Model with Events for the Lead Business Object](image)

We define the active data states are open, qualified, and handed over, while the passive data states are rejected and accepted. The optimal partitioning strategy based on active and passive data is trivial and would be to split the data into two partitions containing either active or passive data data.

Based on what we see on the application side almost all modules will work with data from only one partition. The sales clerk will work on a list of last-weeks leads, which are open. As soon as a lead is rejected or accepted, the probability that it will be considered in the day-to-day transactional business is very low and thus they can be moved to a different partition. However, analytical queries will most likely touch past data which is located on a different partition. Moving data is what makes our approach dynamic and unique. We don’t see that such an approach can be handled in other current system implementations.

From our analysis of enterprise application software systems we see up to 7 to 10 years of data stored in the transactional system. The reasons are many-fold: On the one hand customers are scared to archive old data because it has to be transformed to a different format. Due to this changed format it is no longer accessible by the original application. Furthermore if the archiving process is not executed regularly it has to scan all data and creates a reasonable slowdown of the system. On the other hand legal reasons require companies to keep track of certain records for a predefined time span. Even though this data might never be touched again (even by read queries) it is considered for indexes and queries. In contrast with a lifecycle based approach all modified data is checked regularly if it has to be moved and thus reduces the amount of data need to be scanned by the applications. Furthermore even though “old” data is no longer accessed it still remains accessible. This allows that unused vertical and horizontal partitions can be moved further down the memory hierarchy.

Using this dynamic partitioning approach it is possible to heavily reduce the amount of data that need to be processed by day-to-day applications. Furthermore this architecture has a very important property — even though partitioning incurs a small overhead during query execution if a query has to touch multiple partitions the performance will not degrade severely, but rather improve in average.

We also leverage the fact that, from the business perspective, we know that only a small set of tables belong together and
thus the solution space for horizontal partitioning in enterprise applications is limited and concise.

III. ARCHITECTURE

The origin for our prototype architecture is a main memory database system called HYRISE which is designed to support vertical partitioning of any relation into disjoint partitions [6]. The goal of this architecture is optimal query execution for mixed workloads based on transactional and analytical queries. As a result, in HYRISE, relational tables are created as a composition of multiple horizontal and vertical fragments. The meta-data for each table is stored along with the other meta information of the table in the catalog of the database. Until query execution this concept is totally transparent to the database consumer.

In addition to materialized partitioning HYRISE allows to perform virtual composition based on position lists. To avoid early materialization for intermediate results, only meta-data is stacked and position lists can be combined. Now during query execution the combination of position lists is used to materialize the final result. Figure 4 shows an example for a table that is stacked twice. In addition it is possible to extend this concept using direct position resolving. Here, all position lists are combined before the query is executed to avoid random memory accesses depending on the size of the position lists.

In addition to those concepts HYRISE supports both compressed and uncompressed storage schemes. To move data from the uncompressed delta buffer to the compressed main storage HYRISE implements a merge process that will persistently merge the two stores. During query runtime the results from the main and delta buffer are combined as well so that at any time all data is visible. This behavior as explained in [11] is somewhat similar to comparable approaches in Vectorwise [7].

HYRISE’s architecture is the foundation that can be leveraged to support dynamic horizontal partitioning since most of the required components (e.g. horizontal partitioning, merging) are already in place.

A. Life Cycle Storage

In addition to the meta-data stored for the relational data, the partition information has to be stored. As for the vertical partitioning used in HYRISE the meta-data for the horizontal partitions will be transparent to the consumer in the first place, but allows exploitation in the second place — like when writing customized query plans. For each relation the information is stored and a simple predicate mapping is kept to identify each partition. How this predicate mapping could be used during query execution is explained in Section IV-A.

Our customer analysis shows that in large enterprise applications a huge number of tables exists, however only a few out of them are used to store actual transactional event data, while the other typically carry industry specific or configuration data [10]. Approximately, such an enterprise system has around 70,000 tables, while only 300 to 700 (only 150 tables have more than 10M rows, the overall system size is ≈ 4.8TB) are used for event data. Assuming that those are the tables that will cover most of the read-intensive load, it is only important to cover them from a partitioning perspective. Given this information, we can assume that with partitioning the overall number of tables will not severely increase and the amount of additional meta-data can be handled by the HYRISE.

Figure 5 shows the general setup of HYRISE for our dynamical partitioning approach where all partitions of a single partition specification are located on a single node. The life-cycle meta-data is stored and accessible by the storage manager component. The single partitions of each table are stored individually in HYRISE and are combined as required during query runtime. In this example setup using the Octopus Merge all partitions are read only and only the delta buffer is modifiable.

Based on the previously described architecture it is as well possible to scale this dynamic partitioning approach to multiple nodes. However, under the assumption that active data of one partitioning definition will not be distributed, multi-side write transactions will not occur. This assumption is supported by recent hardware development which allow current available rack systems up to 1TB of main memory, which is enough to cover the active transactional data part of most enterprise systems.

IV. DYNAMIC PARTITIONING

In this section we will describe in detail how we consider implementing the dynamic partitioning approach.

To be able to successfully execute this concept, the database engine must be aware of two different meta models. Interestingly in today’s enterprise applications the life-cycle of objects is already modeled, but only on a higher level and with no connection to the persistence layer. The first model
is the meta description of the business object. Such a model follows common approaches from object-relational schema modeling with one hard requirement: each business object must have exactly one root node. The second meta-model is the description of the life-cycle.

To be able to model the life-cycle of an enterprise application correctly three requirements have to be defined:

1) Given the separation of enterprise data into master-data (describing customer base data, addresses etc.) and event data (transactional documents, etc.), only event data can be partitioned.

2) Object $o_j$ can only be element of exactly one life-cycle $L_j$.

3) Each life-cycle $L_j$ has exactly one root $o_j.parent = \text{nil}$ object. All status changes will be reflected on this root object.

Only using those definitions it is possible to explicitly select one life-cycle model that should be applied for a given business object. The biggest advantage of this partitioning approach is that the status definition used for the business objects closely follows the business process. In contrast to classical sharding approaches the partition specification covers more than relation and joins all participating tables into one partition set. As a result for all tables in the specification the different partitions are known in advance and are coherent.

Figure 6 shows an example with multiple business objects participating in this partitioning definition. In this example the Sales Order is the root object for the partitioning strategy. In case the strategy defines two states “active” and “passive” which correlate to the states of the Sales Order. Furthermore in the business process the dependent business objects Delivery and Invoice are defined. Interestingly the business process has clear semantics when a Sales Order is closed which is when all dependent activities are finished — all deliveries are shipped and all invoices are paid. As soon as this stage is reached the sales order can be closed and will never be modified again. This semantic is defined by the legal requirements of an ERP system.

The broader specification of partitions has the advantage that it follows on the one hand the definition of the business processes and their data structures and the natural encapsulation of application modules, e.g. application objects from Customer Relationship Management do not share data with the Financial Application and thus they can be partitioned independently without interference.

A. Query Execution

When applying horizontal partitioning query execution plays a major role. Two main aspects have to be considered: the first aspect is the latency due to multi-partition data access and secondly the partition predictability.

Due to the requirement that all modifiable partitions of one partitioning definition reside on one node, multi-node write-access can be avoided. Based on this definition and the expected workload we can derive that most of the queries will only touch data from a single partition. Other queries (e.g. analytical queries) that access multiple partitions will read more data to execute the query. As a consequence the longer query runtime hides the latency between the access of multiple partitions, even if they are shared to multiple nodes.

To be able to choose the right partition, the query must contain information that can be interpreted to avoid making unnecessary requests to different partitions. To determine the right partition on the root object is considerably easy since the status predicates can be evaluated. For dependent objects this becomes more complex and it is required to add additional meta-data to choose the right partition.

Due to the fact that HYRISE uses SQL as a query language, it is currently not possible to return complex hierarchical result sets where the correct partition is identified by the query on the root node of the partitioning set. One solution to this problem uses the meta information of the partitioning scheme, if a record is in partition $A$ than all dependent records must be in $A$ as well. Every returned record has a virtual field that contains information of which partition it origins. This information can then be used for any subsequent call for this use-case. Different query languages can handle this meta-data differently and e.g. encapsulate it in the result object.

As described earlier all modifications are captured in the delta buffer of the partition set of one table. The reconciliation to determine in which partition a single record belongs can be detected during merge time. Depending on the strategy the selected record is then moved directly to the next partition or moved to the correct delta buffer of the new partition. In any case only the delta buffer of the active partition could contain records from multiple states. The delta of all other partitions is always coherent.

We expect the best performance for applications that leverage their own modeling infrastructure and express by hard predicates which data they request. For most of the use-cases it is easy to determine if they request data from an active or passive partition due to the definition of the business process. If it is not possible to determine the partition upfront, the query will be executed normally and results from all partitions are merged.
B. Recognizing State Change

Given a definition for the life-cycle of a single object one important task is to identify if the state for a given record changed. Since all modifications of data will be recorded in the main delta buffer of the active partition of the system for each relation only one location is relevant to lookup the actual value.

When analyzing if the state of a certain object changed, it is possible to distinguish between two different approaches — immediate or deferred state analysis. The immediate state analysis has to perform all checks and validations during the modifying transaction while the deferred analysis will check for state changes at merge time.

Due to the fact that the amount of objects that were changed in one transaction is considerably low it is possible to detect the state change for the object, but defer the moving of the objects content and dependent objects to a later point in time. Due to the fact that the delta buffer structure is leveraged to capture state changes we can use the merge process to identify which records should be moved to which partition. As a result the merge process analyzes state changes from the root object and than spreading out to dependent objects. To store information if objects were moved to different partitions a foreign key list should be stored that keeps track of moved objects. Thus as soon as the merge process processes a dependent object no additional lookup to the root object is required to validate that the state changed but it can be directly applied from the foreign key list.

Following the previous requirement that the final state modification will always happen on the root object we think that the deferred analysis is preferable because all data will be scanned during merge and thus additional reconciliation during the transaction is avoidable.

C. Moving Data Across Partitions

One of the key components in the system is what we call the merge component. In the current implementation of HYRISE, this asynchronous process can be used to merge data from the delta buffer with the data from the current main data store into a new compressed main store. The main features of this process are that it already supports the notion of different horizontal partitions and is already capable of combining those partitions. While traditionally the merge only considers a single delta and main structure we experimented with a logarithmic merge, where multiple horizontal partitions are merged together with the delta buffer depending on their age. This logarithmic implementation is the primary foundation for the dynamic horizontal partitioning.

a) Current Implementation: The implementation as described in [11] focuses on merging data from the delta buffer to the main store. During this merge phase all columns of the complete table are scanned and evaluated in different ways — either to rebuild the dictionary compression or to validate if the tuple is valid or not. Since all tuples are touched at this time, it is easily possible to evaluate if a certain tuple should be considered to be merged into the current partition or moved to a secondary one. In addition to the simple two-way merge, HYRISE supports currently a first version of the Octopus Merge is implemented and used to build a generational merge model, in contrast to the following concepts for the generational merge the actual state of a business object is irrelevant but only the time of insert determines the relationship of a tuple to a single partition.

The following two paragraphs describe the possible enhancement of the merge to support moving of tuples from one partition to second.

b) Octopus Merge: The goal of the Octopus Merge approach is to be as efficient as possible in terms of amount a single tuple is scanned multiple-times. As soon as the merge process starts it reads the life-cycle meta-data for this table and “open” all relevant partitions. For all tuples in the delta buffer the algorithm checks to which partition the tuple belongs and apply the merge process for this tuple. Compared to the original implementation this differs only in the number of tables that participate in this process and is closest to the current implementation.

c) Distributed Recursive Merge: In contrast to the Octopus Merge the Distributed Recursive Merge does not process all partitions at the same time. For all tuples in the delta buffer it identifies the to which partition it belongs and move the record to the delta buffer of the corresponding partition. Depending on the size of the delta buffer of the partition and the system load, the actual recompression can be triggered to regain the best query performance.

d) Comparison: Both strategies have their advantages and disadvantages. The advantages of the Octopus Merge are if all relevant partitions are co-located on the same node all memory is local. Furthermore this approach does not require to keep additional local delta buffers to each partition. The disadvantages are that many different partitions exist and many different memory locations have to be touched to merge the results. In addition it can yield a negative impact if the delta buffer size is not optimal with regards to the size of the partition a single tuple is merged into. The following extreme example illustrates this problem: Given 10k records are stored in the delta buffer and we have in total 3 partitions, if now 5k records are merged into partition 1 and 4999 are merged into partition 2 than only one record needs to be merged into partition 3. But merging only a single record is clearly unnecessary and can severely harm the performance of the system.

The advantage of the Distributed Recursive Merge is it’s good self-partitioning behavior allowing to split the different partitions over multiple nodes without requiring that each partition is locally available. Thus, moving a tuple from one partition to another can be modeled as a delete from the main delta buffer and than an insert into the new partition, resulting in an insert in the new delta buffer and still leaving the coherency of all partitions intact. Using this approach it is possible that the best point in time to merge can be decided separately for each partition — depending on the amount of tuples, the number of distinct values etc. in the delta buffer.
However, the major disadvantage is that more data has to be touched during re-distribution of the record and that additional memory is required to store the delta buffer for each partition.

D. Research Challenges

Our approach to dynamic partitioning based on life-cycle data is based on the capability of our system to store data compressed and uncompressed and move data between both partitions using a merge process. Our experiments have shown that the merge process revealed good results, but due to the increased complexity of the partitioning strategy we expect a slow-down that needs to be analyzed and further improved

Especially a direct comparison of the different merge strategies that result in an automated decision when to choose which strategy are challenging.

In addition to the persistence layer query execution becomes very important in this setting. Even though our system restricts the availability of active and modifiable partitions to one system the transactional behavior has to be observed. Closely related is the question how predicates of queries can be evaluated to determine the right partitions, e.g. if the application asks for sales orders of the last week, can we deduct that those are all open sales orders and that the first change date of a sales order of the passive partitions is three month ago? Furthermore we see potential to explore how query languages like SQL could be modified to allow partition context operations so that only the minimum number of partitions have to be touched.

Based on the access patterns to the different partitions we see potential to use different storage media besides main memory. Instead of keeping everything available in RAM, it could be possible to “archive” data to Flash, Disk or even Tape, but still keeping the same format and access methods so that it “looks” like the data is still available.

The last challenge is on the application side. Even though the dynamic partitioning scheme closely follows the business process on the application side, it has to be evaluated in how far the applications need to be modified to make them partitioning aware.

V. RELATED WORK

The related work to our dynamic partitioning approach is two-fold: On the one hand there is the classical workload aware approach of partitioning. Based on the result of the workload analysis the tables are horizontally partitioned to improve the throughput, increase parallelization or avoid multi-side transactions [2], [4]. On the other hand it is possible to horizontally partition the tables without prior knowledge of the application based on hashing algorithms [5].

Closest to our approach we see the work of H-Store [8], [9] where a query workload is used to partition data across multiple nodes with the overall goal to reduce overhead in concurrency control for partitioned databases. However, H-Store lacks support for compression and built-in vertical partitioning.

However, this approach does not consider compression and most important vertical partitioning. In addition to close to optimal partitioning of data our system approach would allow fast analytical query execution. An interesting example is SciDB which allows specific decomposition of logical arrays in vertical partitions or chunks [3]. Furthermore we think that we can leverage the research in compressed read-optimized databases[1] to enrich the re-compression process for dynamic partitioning and especially improve the moving of data between partitions [11].

VI. CONCLUSION

In this paper we showed how to extend the existing database prototype HYRISE with dynamic horizontal partitioning based on the life-cycle of application objects. We motivated this approach based on our observations of real enterprise applications, where most of the transactional processing is done on the active part of the data, but with the drawback that “active” cannot be defined based on simple attributes. We explored how the life-cycle of enterprise objects grouped in a partition specification can be used to dynamically partition the data with the key differentiator that a certain tuple can be moved from one partition to another. In addition, we also described how moving the data can be implemented based on HYRISE prototypes compression engine.

We finished the paper by listing the major research challenges ahead and what we propose to address in the next phase of our project, like performance modeling, query languages and application models.

REFERENCES