Object-Relational Mapping Strategies revised – A comparison of Row- and Column-oriented Database Systems

Martin Lorenz, and Johannes Albrecht

***Abstract*—**Relational databases do not provide a structural equivalent to class inheritance, a key feature in object-oriented programming languages. Research has proposed different strategies to map inheritance structures to flat relational tables. Each strategy results in distinctive characteristics regarding memory consumption and query performance. Industry and research have come up with best practices and application guidelines to chose the optimal strategy for a given application context and database workload respectively. However, these guidelines have manifested from experiences with row-oriented database systems.

In this paper, we investigate the impact of column-oriented data lay- out on the characteristics of different mapping strategies. We introduce existing mapping strategies and discuss their suitability in the context of real-world database workloads, which have been reported by related literature. Following this discussion, we propose an adaptation of an existing mapping strategy. This adaptation, promises optimal performance based on the identified workload characteristics. Our proposal is evaluated by conducting an experiment, which compares row- vs. column- oriented data layout regarding memory consumption and query performance. The experiment has two findings. First, it shows that characteristics of mapping strategies behave different for row- and column-oriented data layout. Second, we see that the adaptation, proposed in this paper is superior to other strategies both in terms of memory consumption and query performance. Based on these findings, we reconsider existing ap- plication guidelines and best practices in the context of column-oriented databases.

***Keywords*—**Object-relational mapping, relational databases, class inheritance.

I. Introduction

THE notion of objects as a means to structure code reaches back to the 1960s when Dahl and Nygaard invented Simula-67 [1]. Object- oriented programming as a new programming paradigm was introduced by Alan Kay with the Smalltalk programming language in the 1970s [2]. Since then, object-orientation has evolved into the dominant programming paradigm for applications in various domains. Especially enterprise applications with their inherent aim to capture properties, behavior, and processes of real world companies benefit from object-oriented programming features. Concepts such as encapsulation, aggregation, and inheritance provide system architects with the means to design domain models, which reflect structures and relations of the real world. Based on such domain models, developers are able to communicate and discuss business logic with domain experts to verify conceptual and logical correctness of the system's functionality. A recurring problem software developers face in the context of class inheritance is the definition of appropriate mapping strategies if data, defined by the application's domain model, needs to be persisted to a relational database. Relational algebra does not define a concept equivalent to object-oriented class inheritance. To overcome this limitation, different map- ping strategies have been proposed. There exist three main strategies, which we describe in Section 4. Publications from academia and industry describe advantages and disadvantages of these strategies [6,7]. They boil down to a trade-off between memory consumption and query performance. With the uprising NoSQL trend, object databases have been proposed as possible solutions to the object-relational mapping problem. Unfortunately, they do not provide performance and flexibility demanded by complex enterprise applications. In the relational database arena, Plattner et al. have proposed a column-oriented database system as primary persistence for enterprise applications [10]. Although the biggest value proposition of such system is speeding up analytical and mixed query workloads, we show how to leverage this paradigm beyond its original design goals. Based on theoretical considerations on column-oriented data layout, we propose a single table strategy in combination with horizontal partitioning that not only allows to reduce the absolute memory footprint of persisted inheritance hierarchies compared to row-oriented systems, it also reduces the relative difference in memory consumption when comparing it with the other strategies. We evaluate our approach in a series of experiments using data from real systems of a large enterprise software vendor, which show that contrary to existing literature, the single table approach in a column-oriented, horizontally partitioned data store does not consume excessively more memory then alternative strategies. In some situations it even outperforms the other strategies. Based on that observation, memory consumption can be eliminated as a primary disadvantage for single table mapping strategies in the context of column-oriented databases. Furthermore, we show that the combination of a single table strategy and a column-oriented data layout outperforms any other strategy when benchmarked against the query workload of real world

Martin Lorenz is with the Hasso Plattner Institute, Potsdam, 14482, GERMANY.

Johannes Albrecht, is with the Hasso Plattner Institute, Potsdam 14482, GERMANY.

enterprise systems.

The remainder of this paper is structured as follows. Section 2 presents related work in this field of research. Section 3 outlines conceptual differences between row- and column-oriented data stores with respect to their abilities of efficient memory usage. Section 4 introduces existing strategies to map object-oriented class inheritance to relational databases concepts. Section 5 elaborates on query workload and data characteristics of enterprise applications. Section 6 describes experiments we conducted to validate our considerations. Section 7 analyzes the experimental results, followed by a conclusion and outlook in Section 8.

II. Related Work

The problem of mapping object hierarchies to relational data models is only one topic of a set of conceptual and technical issues between object-orientation and relational algebra, often referred to as object-relational impedance mismatch. There exist a number of works, which provide comprehensive considerations about the different issues and class inheritance in particular [13,14], and [15]. Possible approaches to overcome these deficiencies include the definition of mapping strategies and the description of best practices [6,7]. Publications, which provide solid evaluations of inheritance hierarchy characteristics and their impact on the performance of mapping strategies could not be found. It is mostly best practices and implementation guidelines from vendors of object-relational mapping frame- works that carry subtle information about observations and learnings from existing projects that provide hints to answer the question, which mapping strategy performs best for a given inheritance hierarchy. Consequently, we do not know of any work that has considered such evaluation in the context of either row- or column-oriented databases. The topic of enterprise data and database workload characteristics has been covered by works of Krüger et al. [8,9]. They analyzed existing enterprise systems of large-scale companies to provide a solid foundation for their work on in-memory technology. We will use their findings as basis for our experiments as well.

III. Row- vs. Column-oriented Data Layout

In this section, we introduce the concept of column-oriented database technology. Sections 3.1 through 3.3 give a brief introduction to column-orientation, dictionary encoding, and partitioning. A more comprehensive description of that feature combination and its beneficial interrelation can be found in [11].

A. Column-Orientation

Databases map the logical two-dimensional table structure into one-dimensional physical computer memory using either row- or column-oriented storage. Figure 1 depicts the two physical layouts. Column-oriented database systems store consecutive values of a database field, whereas row-oriented systems store the fields of a database tuple consecutive in memory. Column-stores' main area of application is analytical applications, because they allow fast scans on columns, but they perform rather poorly on transactional workloads, e.g. inserts, updates, single selects with complete tuple reconstruction.

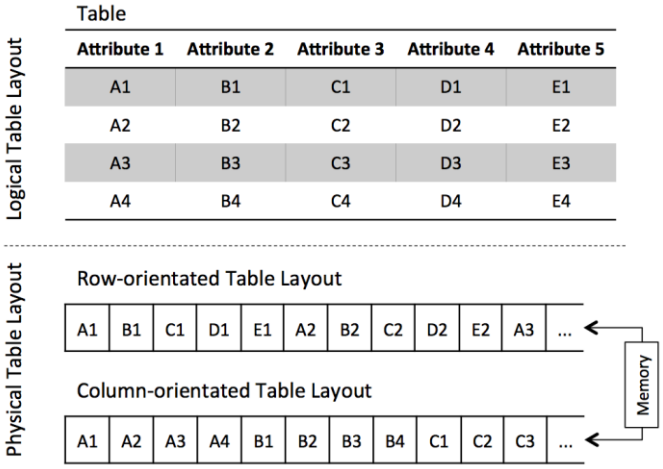

Fig. 1 Column- vs. Row-oriented Table Layout

B. Dictionary Encoding

Dictionary encoding is a lightweight compression scheme, used to reduce the memory footprint of a column. The basic idea is to use a dictionary to map each distinct value of a column to a shorter, memory efficient value. Figure 2 shows an example for a dictionary-encoded column.

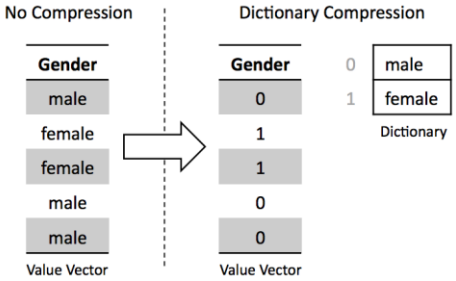

Fig. 2 Example of Dictionary Compression

The most efficient way to encode the values of the dictionary is to use bit compressed integer values, because it uses only the number of bits needed to represent the distinct values of the column. The number of bits needed can be calculated by the following formula,

$$b = \lceil \log_2(n) \rceil \quad (1)$$

where b is the number of bits needed and n the number of distinct values in the column. The example in Figure 2 shows a column with two distinct values, so we need $\lceil \log_2(2) \rceil = 1$ bit to encode these two values.

C. Partitioning

As the name implies, partitioning allows to partition data in a table. A table can be partitioned, vertically, by columns, or horizontally, by rows. When we talk about partitioning, we talk about horizontal partitioning. When inserting new records into a partitioned table, an algorithm or an explicit partitioning rule decide what partition a record is being inserted into.

Partitioning can be used to distribute data of a table in a cluster or it can be used to group data of a certain type together.

IV. Inheritance Mapping Strategies

There exist mainly three strategies, which we introduce in this section. Figure 3 depicts an overview of the three strategies.

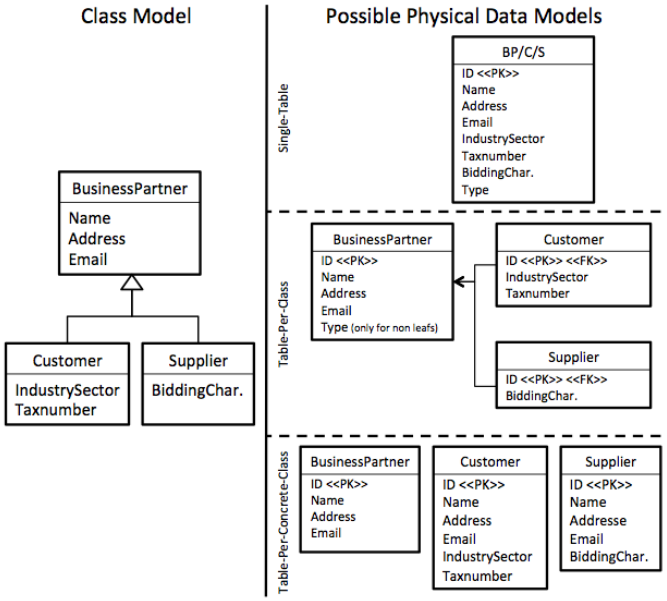

Fig. 3 Overview of Mapping Strategies

A. Single-Table Inheritance

The Single-Table Inheritance strategy (ST) is also referred to as one table per hierarchy inheritance or one inheritance tree one table. The strategy uses one database table to store all the information for an entire inheritance hierarchy. Therefore the table contains a column for every attribute of every class in the hierarchy. To deter- mine the class, a record belongs to, the table introduces a discriminator column. This column stores an identifier that allows mapping a record to a certain class type. If an instance of a class gets stored in the table, only the columns for the attributes of the class and all superclasses will be filled. All columns, which do not belong to the class or its superclasses will be filled with NULL-values.

Fowler [6] defines the following advantages for the ST approach. There is only one table to worry about. There are no joins in retrieving data. Any refactoring, which pushes fields up or down the hierarchy does not require to change the database schema. On the downside, not all fields might be relevant for a record, which may confuse people, who work with the table directly. Columns used only by subclasses lead to wasted space in the database. The table may end up to be to large, requiring many indexes and frequent locking. Furthermore, there is only a single namespace for fields, so you have to be sure you do not need the same name for different fields. Ambler [7] adds, that it is simple to add new classes, because you just need to add new columns to the table. He also states that this strategy is very suitable for fast data access in particular for ad hoc reporting. On the negative side, Ambler mentions that coupling with the class hierarchy increases, because all classes are directly coupled to the same table. A change in a class may affect the table, which then affects the other classes in the table. Indicating the type becomes complex, when significant overlap between types exists and the level of cohesion suffers, because several concepts are stored in one table.

B. Table-per-Class Inheritance

The Table-per-Class Inheritance strategy (TPC) is also referred to as class table inheritance or one inheritance path one table. The strategy uses one database table for each class in the inheritance hierarchy. Each table contains columns for each attribute defined for the specific class and a primary key, which is shared between this class and its superclasses. All values for inherited attributes will be stored in the tables for the superclasses. Thus, the information for one class instance gets distributed between the table for the class itself and the tables for all of its superclasses.

Fowler [6] defines the TPC approach as the simplest relationship between tables and classes. From his point of view, the strengths of this strategy are that all columns in a table are relevant, which prevents wasted space, and that the relationship between the domain model and the database schema is straightforward, which makes it easy to understand. The weaknesses of the strategy are the need to perform joins to load an object. Refactoring of fields up or down the hierarchy causes database changes. Supertype tables may become a bottleneck, because they have to be accessed frequently and a high degree of normalization may make it hard to understand for ad hoc queries. In addition to Fowler's best practices, Ambler [7] states that the TPC approach allows adding subclasses very easy, because you just need to add a new table. On the downside, he adds that this strategy needs many tables in the database, one for every class (plus tables to maintain relationships).

C. Table-per-concrete-Class Inheritance

The Table-per-concrete-Class Inheritance strategy (TPCC) is also referred to as concrete class table inheritance or one class one table inheritance. It uses one database table for each class in the inheritance hierarchy. Each table contains columns for the attributes of a specific class in the hierarchy and also all attributes from the classes it inherits from.

Fowler [6] enumerates the following advantages for the TPCC approach. Each table is self-contained and has no irrelevant fields. Such table can be used by other applications that are not using objects. There are no joins to do when reading data from the tables. Each table is only accessed when that class is accessed, which spreads the load over multiple tables. On the downside, pushing fields up or down the hierarchy requires altering the table definitions. Changes to fields in superclasses trigger changes to each table that has this field, because superclass fields are duplicated across the tables. A search on a superclass forces to check all tables of the corresponding subclasses, which leads to multiple database accesses or weird joins. Ambler [7] adds that ad hoc reporting is easy, because all data needed about a single class is found in a single table. A critical point of this strategy is when an

object changes its role. In that case it is necessary to copy the data into the appropriate table. Similarly, it becomes difficult to support multiple roles and still maintain data integrity.

D.Adaptation of the Single-Table Approach

We propose a fourth strategy (STP) for evaluation in this paper. The ST strategy obviously provides high query performance, especially when it comes to transitive queries, which execute not only on a single class, but on all of the classes, derived from that class. Its flaws lie in its potential waste of memory and large number of rows, since all fields of all classes are stored in a single table. That means attributes will be empty for fields, which are not defined for the particular class. Figure 4 depicts an example table for Single-Table strategy, based on the domain model shown in Figure 3.

Our proposition is to apply horizontal partitioning to the ST strategy to physically divide the table into different partitions. The partitioning scheme comes natural from the different class types, defined within the inheritance hierarchy. That means, the partitioning algorithm groups all records from the same class type in the same partition. The result of that grouping is that every partition only contains columns that are either completely filled or completely empty.

	BusinessPartner Attributes			Customer Attributes		Supplier Attributes	
ID	Name	Address	Email	IndustrySector	Taxnumber	BiddingCharacteristc	Type
1	Name1	Address1	name1@gmail.com	NULL	NULL	NULL	BP
2	Name2	Address2	name2@gmail.com	NULL	NULL	NULL	BP
3	Name3	Address3	name3@gmail.com	Healthcare	T4563727	NULL	C
4	Name4	Address4	name4@gmail.com	NULL	NULL	NULL	BP
5	Name5	Address5	name5@gmail.com	NULL	NULL	LOW Priority	S
6	Name6	Address6	name6@gmail.com	NULL	NULL	LOW Priority	S
7	Name3	Address3	name7@gmail.com	Automotive	T9234621	NULL	C
8	Name7	Address7	name8@gmail.com	Public Sector	T4536341	NULL	C
9	Name8	Address8	name9@gmail.com	NULL	NULL	NULL	BP

Fig. 4 Example for ST Strategy

If we apply the proposed adaptations, the physical data schema from Figure 4 changes into the schema shown in Figure 5. Logically, we still have a perfectly fine ST strategy. A column-store is now able to selectively drop columns for partitions where the column is completely empty, allowing to reduce the overall memory footprint of that table significantly. At the same time, queries that target only a certain class or certain branch of the inheritance hierarchy do not have to scan the whole table.

	BusinessPartner Attributes			Customer Attributes		Supplier Attributes		
ID	Name	Address	Email	IndustrySector	Taxnumber	BiddingCharacteristc	Type	
1	Name1	Address1	name1@gmail.com	NULL	NULL	NULL	BP	P1
2	Name2	Address2	name2@gmail.com	NULL	NULL	NULL	BP	
4	Name4	Address4	name4@gmail.com	NULL	NULL	NULL	BP	
9	Name8	Address8	name9@gmail.com	NULL	NULL	NULL	BP	
3	Name3	Address3	name3@gmail.com	Healthcare	T4563727	NULL	C	P2
7	Name3	Address3	name7@gmail.com	Automotive	T9234621	NULL	C	
8	Name7	Address7	name8@gmail.com	Public Sector	T4536341	NULL	C	
5	Name5	Address5	name5@gmail.com	NULL	NULL	LOW Priority	S	P3
6	Name6	Address6	name6@gmail.com	NULL	NULL	LOW Priority	S	

Fig. 5 Adapted ST Strategy

It suffice to scan the partitions that contain records for that particular class or its transitive children.

V. Enterprise Application Characteristics

The experiment suite that we use is designed to analyze mapping strategies for arbitrary inheritance hierarchies and query workloads with respect to memory consumption and query performance. Different software system domains have distinguishable characteristics in terms of structure of the inheritance hierarchies and query workload on these hierarchies. In order to derive meaningful conclusions from the results of the experiments it is necessary to understand the workload characteristics of the particular software system domain.

A. Enterprise Workload Characteristics

From a query workload point of view, it is interesting what types of queries are executed on the inheritance hierarchy. Krüger et al. have analyzed enterprise systems with respect to query workloads and data characteristics. On a high level, they distinguish between read and write queries, where write queries consist of insert, update, and delete and read queries of range selects, table scans, and lookups. For a detailed description of the definition of the particular query type, we refer to the cited paper [9]. Our experiment framework can easily be extended to support other types of queries, but we restrict our implementation to the ones found by Krüger et al. in their system analysis.

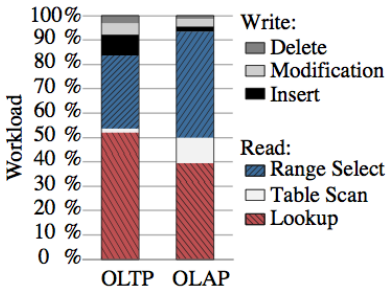

Fig. 6 Query Distribution [9]

Since we are analyzing inheritance hierarchies, we need to distinguish between transitive and non-transitive queries. The ”is-a” relationship that constitutes an inheritance hierarchy is transitive. If we define a query for a class, which has derived child classes, we need to define whether or not the execution of that query should operate only on the class itself or also on all of its child classes. Translated into database terms it means that we have to decide whether or not to query all tables of classes, which correspond to classes derived from the class the query was defined for. Figure 6 depicts the result of their analyzes. The result of the analyzes shows a clear dominance of read queries even in transactional systems, which is a valuable information when discussing the results of our experiments.

VI. Experiments

In this section we describe our experiment setup and the procedure of gathering the results in these experiments.

A. Experiment Setup

The following two sections explain our experiment setup in terms of the hard- ware specification, the database management systems (DBMS) we used and the hierarchies we tested.

Hardware and DBMS specifications For our experiments we used a VM with SUSE Linux Enterprise Server for SAP Applications 11.2, 8x Intel Xeon CPU X5670 @ 2.93GHz and 64 GB memory. On this VM we conducted experiments on three different DBMS, whereof two were row-oriented and one column-oriented. As row-oriented DBMS we used MySQL 5.6.14 with the My- ISAM Storage Engine and PostgreSQL 9.3.0. For the experiments on the site of column-oriented DBMS we utilized SAP HANA 1.50.00.376738.

Hierarchies We conducted experiments on three different inheritance hierarchies. The first hierarchy is a very deep but narrow hierarchy with a depth of five and each class has exactly one child (in the following referenced as deep hierarchy). The second hierarchy is deep but also wide. This hierarchy also has an inheritance depth of five but each class has two children (in the following referenced as tree hierarchy). The third hierarchy is a flat hierarchy with an inheritance depth of one and the root class has five children (in the following referenced as flat hierarchy).
For the size and query performance measurements, the classes of the hierarchies were mapped to the databases according to the strategies ST (partitioned and non-partitioned), TPC and TPCC. In order to measure the impact of the in- stance count per class in the hierarchy on the size and query performance of the database tables, we measured the mapping of the hierarchies with an instance count of 100 to 100'000 per class.

B. Result Collection

In the following two sections we describe how we collected the data for our database size and query performance results.

Database Size To retrieve the space consumption impacts of the strategies, we measured the database schema size for each of the three DBMS, each strategy and different instance counts per class in the hierarchy. Since our main goal is to determine how well the STP strategy performs compared to ST but especially to TPC and TPCC.

Query Performance For the measuring the query performance, we defined delete, update, insert, range select, non-key attribute select (table scan) and table lookup queries to cover the operations needed in a typical OLTP and OLAP workload as shown in figure 6. The queries were defined for each class in the inheritance hierarchy and also we differentiated between transitive and non- transitive queries. Each query was executed ten times and the average runtime of these executions were calculated.

VII. Result Analysis

In the following sections we show and interpret our measurement results in terms of database space consumption and query performance for the different inheritance mapping strategies. Due to our comprehensive analyses, we gathered a lot of data of which we only can present a digest in this paper.

A. Database Size

In this section we analyze the results of our experiments regarding space consumption and query performance for the different inheritance mapping strategies and databases. Our experiments showed, that regarding space consumption for the different strategies, the column-oriented HANA strongly varies from a row- oriented database like MySQL and PostgreSQL.

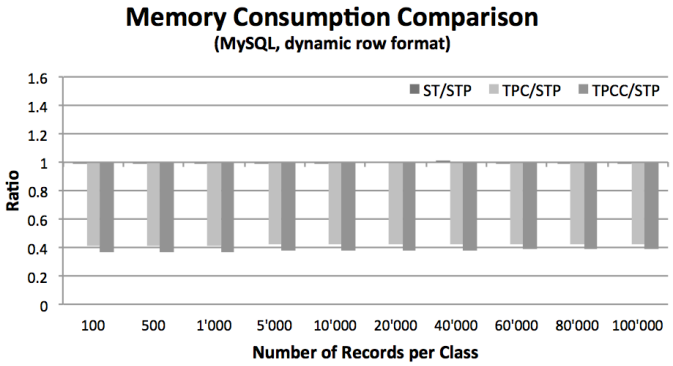

Fig. 7 Comparison of memory consumption of STP vs. ST, TPC, and TPCC in row-store

While with MySQL the space consumption ratio between the different strategies stays constant regardless of the instance count per class in the inheritance hierarchy, the STP approach in HANA performs better compared to the other strategies, the more instances get created. For our experiment with the tree hierarchy, the STP approach performed better than the ST and the TPC approach for an instance count of over 80'000. Also the space consumption of the STP approach gets close to the TPCC approach, which is often described as the best strategy in terms of space consumption in the literature. Figure 7 and Figure 8 show the results for MySQL as a representative for row- oriented DBMS, SAP HANA for column-oriented DBMS and the tree hierarchy.

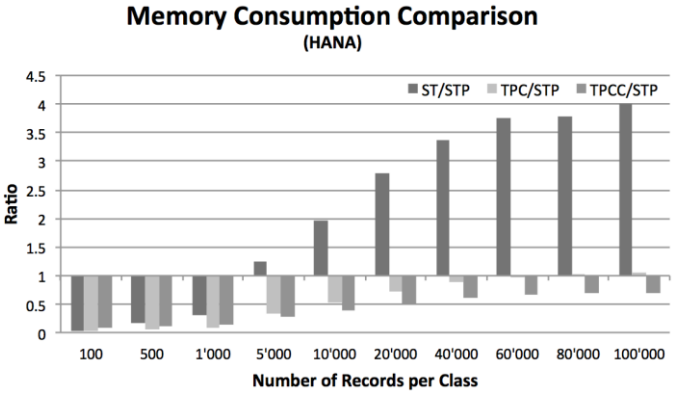

Fig. 8 Comparison of memory consumption of STP vs. ST, TPC, and TPCC in column-store

B. Query Performance

In the following we present the results for MySQL and SAP HANA and the tree inheritance hierarchy.

1) Insert

On MySQL the insert query performance is very fast. Excluding the TPC approach, the strategies performed equally good, independent of the mapping strategy, the number of class instances or the hierarchy depth. Each of the query runtimes averages at one millisecond or less. Depending on the hierarchy depth, the TPC approach needs more than one insert query, because an entity will be stored in multiple tables, if it is not the root class.

For SAP HANA there are differences between the strategies. The TPCC approach performs best, followed by the ST approach and the STP approach. The query performance of the TPC approach depends on the depth of the queried class in the inheritance hierarchy. While executing an insert operation for an in- stance of the root class is equally as fast as the TPC approach, the performance for inserting an element in level five of the hierarchy is between the ST and STP approach.

2) Update

Except for the TPC approach, the update performance for MySQL averages at one millisecond or less. The TPC approach is substantially slower, because before the update query gets executed, there first has to be a query towards a MySQL information schema table, to find out, which table the to be updated attribute belongs to.

This also applies for HANA, except the query to find out the attributes location has to be executed against a table of the SYS schema. In total, the TPC queries get executed faster on HANA. However the STP approach is the slowest for update operations in HANA.

3) Range Selects

Range select queries have to be differentiated between transitive and non-transitive queries. For MySQL and transitive range select queries the STP and TPC strategies perform best. ST performs worse the deeper the level the queried class is in the inheritance structure. TPCC is the worst strategy when it comes to transitive queries, because querying a class requires union operations with all of its subclasses. Therefore the query execution time gets worse the closer the queried class is located towards the root class. Non-transitive range select queries in MySQL perform best when using STP, TPC or the TPCC. ST however gets slower the deeper the queried class is located in the hierarchy.

Transitive range select queries in HANA perform best with the ST approach, followed by the STP approach. TPC loses performance with a rising number of class instances and TPCC shows a similar behavior as in MySQL except the total execution times are much smaller, especially for a high class instance count. Non-transitive range select queries for HANA perform equally fast on average with the ST, STP and TPCC strategy. However, TPC forms an exception. With TPC, the average query execution time increases with the depth level of the class in the hierarchy. This is because of the additional join conditions that are needed to fetch a whole class instance from the tables belonging to its inheritance path.

4) Non-key select and lookup

For transitive non-key select and lookup queries on MySQL the TPC strategy is the best strategy. The query average query execution times of the other strategies is for ST always high and for STP and TPCC it depends very much on the level of class in the hierarchy. Their query execution time gets lower the deeper the class is in the hierarchy. However for non-transitive queries the results are different. ST still shows high execution times, but TPCC has the best performance followed by STP. The performance of the queries for the TPC strategy depends on the level of class in the hierarchy, due to the join conditions mentioned before.

Similar to MySQL, TPCC has the slowest query execution times for transitive non-key select and lookup queries on HANA (see Figure 9), depending on how close the queried class is to the root class.

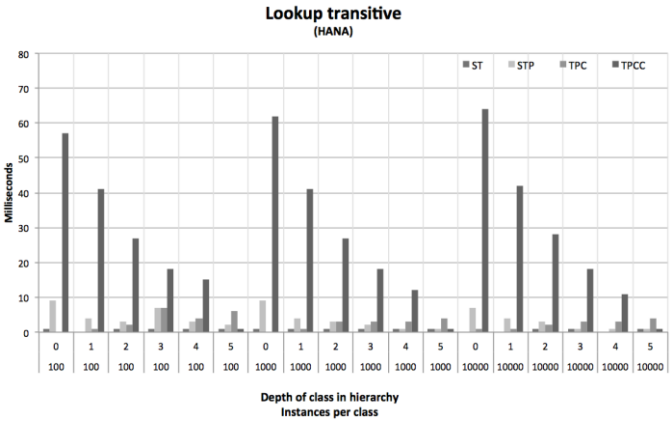

Fig. 9 Lookup transitive query performance for HANA

However the total execution times are smaller for HANA. The fastest strategy is ST. STP and TPC are on par with ST, depending on the inheritance depth of the queried class. STP is a little bit slower when querying classes close to the root class and vice versa for TPC. However, all ST, STP and TPC have single-digit execution times. Non-transitive non-key select and lookup operations on HANA are equally fast in all strategies except TPC, which again performs worse the higher the inheritance depth of a class is.

VIII. Summary and Future Work

In this paper, we investigated the impact of column-oriented data layout on the characteristics of different mapping strategies. We proposed an adaptation of the Single-Table mapping strategy. We evaluated our proposal by conducting a suite of experiments, which compared row- vs. column-oriented data layout regarding memory consumption and query performance. The experimental results revealed that characteristics of mapping strategies behave different for row- and column- oriented data layout. We showed that the adaptation, proposed in this paper, eliminates the disadvantage of increased memory consumption for the Single- Table approach. Furthermore, compared the query performance characteristics of each strategy against query distribution, derived from an analysis of a real world enterprise system.

For future work, we look into defining a cost model for inheritance mapping strategies that allows to compare different mapping strategies for a given inheritance hierarchy structure and a given workload. Furthermore, we plan to

perform a source code analysis of different software systems of a large vendor of enterprise software, to get a deeper understanding of the typical structure of inheritance hierarchies in existing domain models.

REFERENCES

[1] Ole-Johan Dahl, Kristen Nygaard, SIMULA: an ALGOL- based simulation language, Communications of the ACM, v.9 n.9, p.671-678, 1966.
http://dx.doi.org/10.1145/365813.365819

[2] W Daniel H. H. Ingalls, The Smalltalk-76 programming system design and implementation, Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, p.9-16, 1978.

[3] Ralph E. Johnson and Brian Foote. Designing reusable classes. Journal of Object-Oriented Programming, p.22-35, 1988.

[4] J. Daly, A. Brooks, J. Miller, M. Roper and M. Wood, The Effect of Inheritance on the Maintainability of Object-Oriented Software: An Empirical Study, Proc. IEEE Int'l Conf. Software Maintenance, p.20-29, 1995.
http://dx.doi.org/10.1109/ICSM.1995.526524

[5] E Antero Taivalsaari, On the notion of inheritance, ACM Computing Surveys (CSUR), v.28 n.3, p.438-479, 1996.
http://dx.doi.org/10.1145/243439.243441

[6] M. Fowler, Patterns of Enterprise Application Architecture, Boston: Addison-Wesley, 2003 .

[7] C S. Ambler, Agile Database Techniques: Effective Strategies for the Agile Software Developer, Wiley, 2003.

[8] J. Krüger et al., Data Structures for Mixed Workloads in In-Memory Databases, ICCIT, 2010.

[9] J. Krüger et al., Fast Updates on Read-Optimized Databases Using Multi-Core CPUs, PVLDB, Volume 5, No. 1, 2011.

[10] H. Plattner, SanssouciDB: An In-Memory Database for Processing Enterprise Workloads, BTW, 2011.

[11] H. Plattner et al., In-Memory Data Management., Springer, 2010.

[12] Abadi et al., Integrating compression and execution in column-oriented database systems, Proceedings of the 2006 ACM SIGMOD international conference on Management of data, 2006.
http://dx.doi.org/10.1145/1142473.1142548

[13] T. Neward, The Vietnam of Computer Science, The Blog Write, 2006.

[14] W. R. Cook, Ali H. Ibrahim, Integrating programming languages and databases: What is the problem?, Expert Article on ODBMS.ORG, 2005.

[15] W C. Ireland, et al., A classification of object-relational impedance mismatch, Advances in Databases, 2009.