KEY-VALUE STORAGE ENGINES IN RELATIONAL DATABASES

STEFAN HEMMER

A thesis submitted in partial fulfillment for the degree of
Master of Science

in the
Fachbereich Informatik
AG Datenbanken und Informationssysteme

December 2016

ABSTRACT

This thesis examines interpreted and generated implementations of in-memory tuple representations and their impact on query processing. In recent years, the focus of Database research has switched from mostly I/O-focused optimization efforts to the integration of computational resources. Interpretation-based strategies, introduced to provide flexibility, create an overhead in terms of CPU time and memory consumption. Code generation approaches can remove these cost and concurrently offer an additional level of optimization through compilers.

The tuple preparation of Relational Database Management System utilizing embedded *key-value stores* as storage engines, exhibits interpretational overhead that can be removed with code generation approaches. In an effort to classify different approaches, this thesis examined the structural differences of interpreted and pre-compiled in-memory tuple representations. The impact of marshalling and unmarshalling data on the total query processing effort was analyzed. In a series of experiments it was shown that pre-compiled tuple preparation methods offer a higher throughput than interpreted approaches. Further it was shown that in a Online Transactional Processing benchmark pre-compiled tuple preparation has a positive impact on query processing.

ZUSAMMENFASSUNG

Diese Master-Thesis untersucht interpretierte und generierte Implementierungen von *in-memory* Tupel-Repräsentationen und ihren Einfluss auf die Anfrage-Verarbeitung. Der aktuelle Fokus in der Datenbank-Forschung hat sich von I/O-Optimierungen hin zur Integration von Rechenkapazität verschoben. Interpretations-abhängige Strategien zur Erhöhung der Flexibilität verursachen einen Mehraufwand an CPU-Zeit und einen erhöhten Speicherbedarf. Durch Code-Generierungs-Ansätze können diese Kosten reduziert werden. Außerdem eröffnen sie zusätzliche Möglichkeiten für Compiler-Optimierungen.

In relationalen Datenbank-Management-Systemen, welche eingebettete *Key-Value Stores* als Speichereinheit verwenden, weist die Vorbereitung von Tupeln einen solchen interpretationsabhängigen Mehraufwand auf, der durch Code-Generierungs-Ansätze beseitigt werden kann. Um die Leistung der unterschiedlichen Ansätze zu untersuchen, wurden in dieser Thesis die strukturellen Unterschiede von interpretierter und generierter Tupel-Vorbereitung betrachtet. Desweiteren wurde der Einfluss der Tupel-Vorbereitung auf die Leistung eines Mehrzweckdatenbanksystems analysiert. Es konnte festgestellt werden, dass mit vorkompilierten Ansätzen der Tupel-Vorbereitung eine höhere Verarbeitungsmenge erzielt werden kann als mit interpretations-basierten. In einem Online Transactional Processing Benchmark wurde schließlich bestätigt, dass vorkompilierte Tupel-Vorbereitung einen positiven Einfluss auf die Anfrageverarbeitung hat.

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth [11]

ACKNOWLEDGEMENTS

I would like to take the opportunity and express my gratitude to all the people who supported me during my Master Thesis.

First of all, I would like to thank my direct advisor M.Sc. Caetano Sauer, who offered me advice and guidance, whenever I needed it. It was great working with him in a motivational and friendly atmosphere.

Prof. Dr. Theo Härder I wish to thank for the great opportunity to work on highly relevant and interesting topics in his department, not only during this thesis, but also in the years before in my position as research assistant. Thank you for your inspiring advice and discussions.

I would also like to thank Prof. Dr. Stefan Deßloch for his kindness and supportiveness while being a mentor of my master studies at TU Kaiserslautern.

Finally, I would like to thank my family and my friends for their patience, understanding and encouragement.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

DB	Database
DBMS	Database Management System
RDBMS	Relational Database Management System
API	Application Programming Interface
OS	Operating System
QEP	Query Evaluation Plan
IMDB	In-Memory Database
JIT	Just-In-Time
VM	Virtual Memory
JVM	Java Virtual Machine
SIMD	Single Instruction, Multiple Data
OLTP	Online Transactional Processing
OLAP	Online Analytical Processing
STL	Standard Template Library
POD	Plain Old Data Structure
LLC	Last Level Cache
dTLB	Data Translation Lookaside Buffer
RAM	Random Access Memory
TPS	Transactions Per Second
tpmC	Transactions-per-Minute-C

1

INTRODUCTION

1.1 MOTIVATION

In terms of non-functional requirements, Database Management System (DBMS) development was traditionally driven by two important factors: performance and data independence. The former represents the rate at which a DBMS can satisfy the demand for storage and retrieval. The latter describes the degree of immunity of applications or users to changes in physical storage structures (physical data independence) and conceptual schemata (logical independence).

A driving factor in terms of performance of data intensive applications like DBMSs is the underlying hardware. Thus, the architecture of general-purpose DBMSs developed two to three decades ago was strongly influenced by the high latency of data access operations. Accordingly, the communication between CPU and storage devices was pinpointed as the bottleneck of DBMS operations. Until today, modern general-purpose RDBMS include a standard set of features that address these initial challenges.

To reduce a storage overhead, data is organized in slots on pages of virtual memory. A managed buffer pool controls which pages reside in memory. Data items are accessed via structures like heap files or indices. A locking module enables concurrency and recovery is facilitated by logging modules.

Data independence is introduced on the data model level through the relational model [2]. It organizes data as sets of tuples in two-dimensional tables, called relations. Each tuple consists of a set of fields. Schemata, defined as sets of domains, specify the data types of the fields of a tuple. This abstraction allows the description of data by its structure, without specifying physical representational attributes. The relational model makes it possible to provide declarative methods for specifying and accessing data.

Typically, Relational Database Management Systems (RDBMSs) provide a query processing engine that is capable of interpreting a declarative language like *SQL*. Query op-

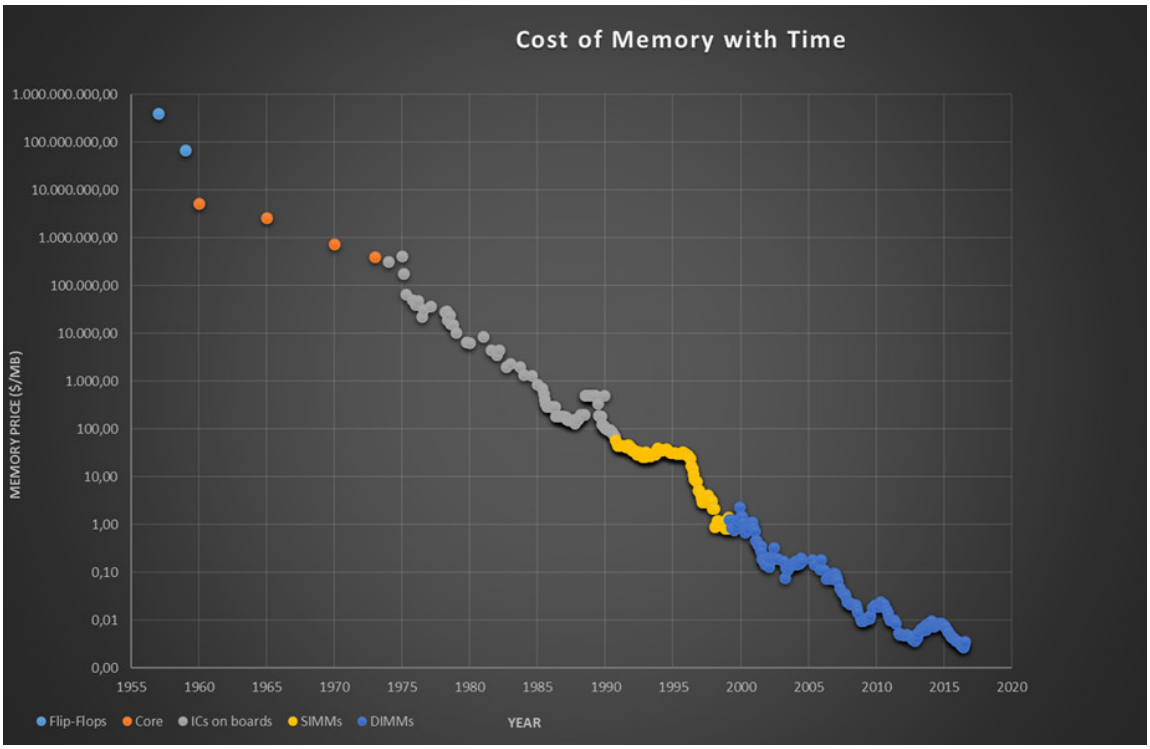


Figure 1: Development of memory costs [15]

timizers compile requests into a logical algebra which can be evaluated and optimized. Subsequently, they translate logical algebra to physical algebra operators that specify the control flow of a query. These expressions are then evaluated to produce the requested result. Most general-purpose DBMSs use an interpretation-based scheme [7] to realize this functionality.

The evolution of hardware demands adaptations to this traditional software architecture. Especially the developments in volatile memory enable interesting variations. As depicted in Figure 1, the price per MB of Random Access Memory (RAM) has decreased dramatically over the last sixty years. The access time to volatile memory is one order of magnitude smaller than the access time to disk storage which allows for faster response times and higher transaction throughput rates of DBMSs.

Consequently, the development of In-Memory Databases (IMDBs) becomes feasible. In contrast to a conventional Database (DB) system, an IMDB stores the data completely in main memory, which provides the necessary performance to realize a set of real-time applications. However, their design also has considerable implications to the architecture of DBMSs [6]. Modules like concurrency control, commit and query processing, re-

covery and data representation need to be optimized for better performance and the byte-addressable, random access style of extracting data.

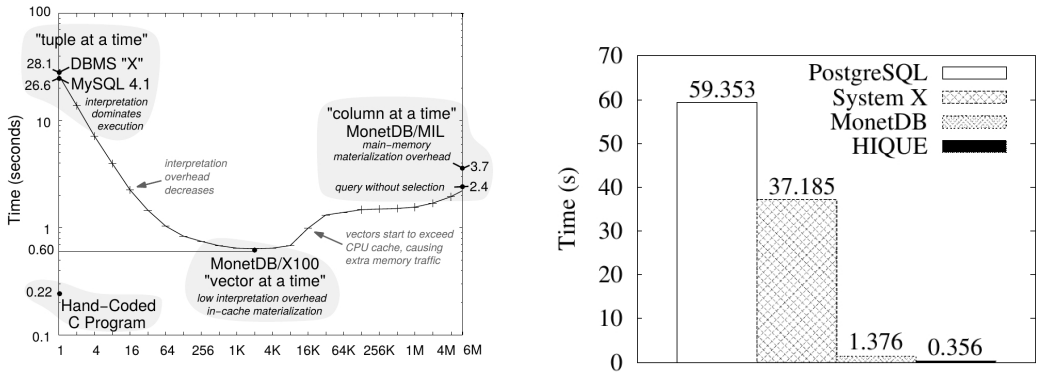
In the context of IMDBs, and to some extent also in modern general-purpose DBMSs, the bottleneck of query processing switches from I/O operations to in-memory processing. In the past, the architecture of DBMSs and their query optimizers would focus on reducing the amount of I/O operations, since they introduced the greatest performance penalties. With large portions of a DB's data sets fitting into main memory, the optimization for these kinds of costs becomes less significant. To increase performance, future cost models of DBMSs must include CPU and memory access costs. Optimization in query processing, in turn, must aim to reduce CPU cycles and memory accesses.

The interpretation-based approach to query operator execution [7] was identified as a source of query processing overhead in several recent publications [16, 17, 22]. Traditionally, RDBMSs are implemented without knowledge of queries they process, therefore query operators are written as generic as possible. While the generality of an interpretation-based model provides the desired data independence, it also introduces an interpretational overhead at runtime.

In general, the performance of interpretation-based approaches suffers from the use of frequent function calls and abstract implementations. Frequent function calls cause code to create additional instructions and branches, resulting in conditions and jumps that cause disruptions of the instruction sequence. Branch-free code facilitates pipelining and superscalar execution in modern CPUs. The use of abstraction also introduces overhead through additional pointer resolution. Virtual calls or calls through function pointers are more expensive than function calls themselves. Even further, they degrade the branch prediction capability of modern CPUs.

DB research introduces two concepts to measure and negate the impact of interpretation-based approaches in query processing: vectorized methods [14, 22] and code-generation strategies [12, 16, 17]. Figure 2 illustrates the difference between computational power of hand-coded, vectorized, code-generation and interpretation-based approaches of query processing based on the performance in the TPC-H Query 1.

Figure 2a compares the time to execute TPC-H Query 1 for varying vector sizes in the vectorized approach to that of hand-coded versions and MySQL, a general-purpose DBMS. The vectorized approaches of MonetDB/MIL [14] and MonetDB/X100 [22] reduce the interpretational overhead by processing a vector of tuples in one operator call. Thereby,



(a) Interpreted, hand-coded and vectorized approaches [22] (Horizontal axis represents vector sizes in tuples)

(b) Interpreted, vectorized and compiled approaches [12]

Figure 2: TPC-H Query 1 performance of different query operator approaches

additional costs of a function call and abstract implementation are only payed once per set of tuples, instead of once per single tuple. Depending on the size of the tuple vector, the approaches increase the performance by up to two orders of a magnitude. However, it is also observable that hand-coded techniques still outperform even vectorized systems.

Figure 2b compares the execution time of TPC-H Query 1 in two general-purpose DBMSs and the vectorized engine of MonetDB to that of *Hique* [12]. Hique realizes query processing by using a code generation and compilation strategy, which removes interpretational overhead by resolving the indirection at compile time. As Figure 2b shows, Hique decreases the execution time of TPC-H Query 1 by almost one order of a magnitude in comparison to the vectorized approach. Interpretation-based approaches like PostgreSQL and "System X" are outperformed by two orders of a magnitude. Thereby, the idea of compilation strategies being superior to other strategies is reinforced.

Typically, general-purpose DBMSs distinguish between two levels of data representation: logical and physical type structures. For instance, an RDBMS provides constructors on the logical representation level for tuples, relations, sets, lists, arrays, pointers etc.. A physical data representation layer contains types like files, records, record identifiers and large byte arrays. On a system engineering level, this division is reflected in the separation into different architectural layers: the query processing engine and storage engine. Figure 3 shows the five-layer DBMS architecture model [10] and the attribution of the five layers to a generalization into query processing and storage engine. Storage engines control propagation for the physical application of values to disk and offer physical ac-

cess paths. The query processing layer optimizes and executes queries requested in a declarative language by offering logical access paths and views to data.

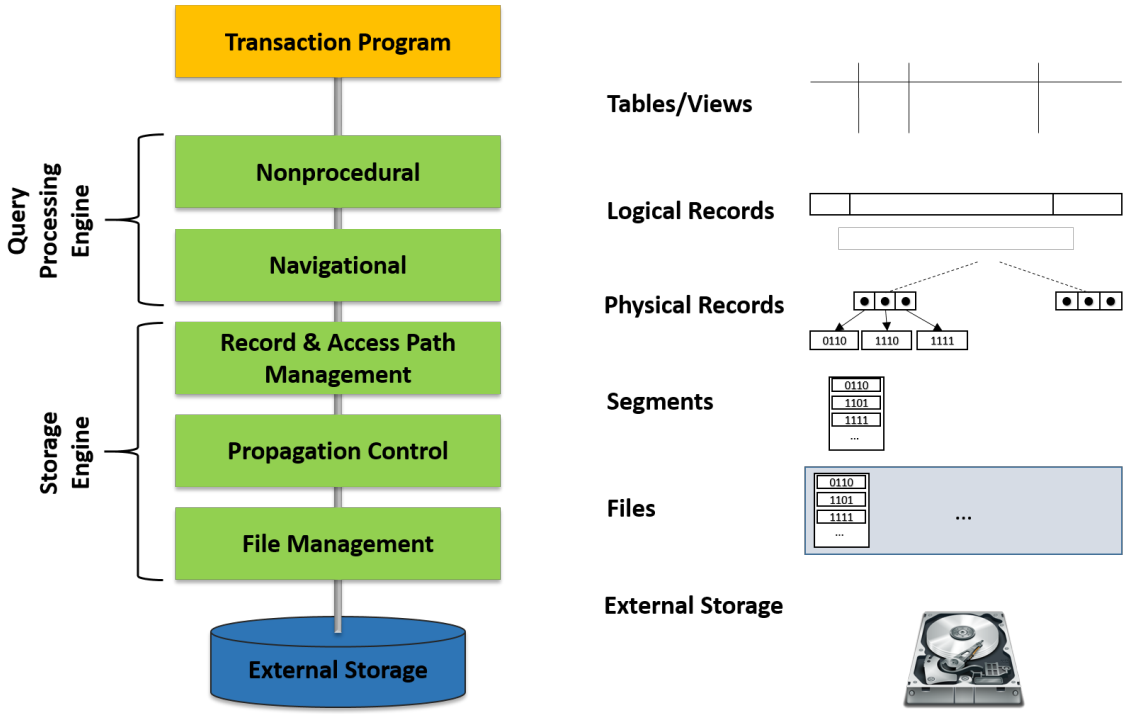


Figure 3: Five-layer DBMS architecture

DBMSs often utilize embedded *key-value stores* to realize a storage engine. The distinguishing feature of a key-value store is its simplicity. Typically, key-value stores offer a simple Application Programming Interface (API), depicted in Listing 1. In a key-value store, data is organized based on the value of a unique key, often a string of characters. Each key is associated with exactly one value, treated as schema-less data. Treating data as a single opaque collection of various fields offers great flexibility and scalability and allows easy and fast lookups.

Listing 1: Key-value Store API

```
void put(string key, byte[] data);
byte[] get(string key);
void remove(string key);
```

RDBMSs, however, operate on the relational model. Data in the relational model is represented as highly structured entities with relationships between them. In contrast to the restricted lookups based on a single value of the data item in key-value stores, the rela-

tional model allows lookups on single fields of an entity. Exposing this data structure to query optimizers allows the application of macro and micro optimizations.

When used as a storage engine, the key-value stores' data model implies that it is oblivious to a DBMS's data model. Hence, interfaces between query processing engine and key-value store must prepare tuples first before they can be processed by either. In one direction, the physical representation, for example a byte array of a key-value store, must be prepared for the usage as a tuple in the relational model, called marshalling. To this end, an RDBMS scans the byte array. For each defined field of a tuple, it extracts the data from its byte representation to an in-memory representation. Conversely, the in-memory representation must be prepared for its opaque storage, called unmarshalling. The RDBMS must iterate through all fields of a row and extract their byte representation. It must concatenate these values in a byte array and submit it to the embedded key-value store for storage.

Additionally to being implemented without knowledge of the types of queries, DBMSs are also implemented without knowledge of the structure of the data they store. Hence, DBMSs utilize an interpretation-based approach to preparing tuples as well. Their main advantage is that they offer a high degree of flexibility in terms of tuple structure at runtime.

In contrast, a compiled approach requires that the fields of a tuple are specified at compile time. Thus, a compiled approach is often regarded as a disadvantage in a DBMS. Schema evolution has the potential to either invalidate a compiled approach to tuple preparation or to necessitate, sometimes costly, re-compilations. Compiled approaches do, however, offer better runtime performance. Similar to generated query operators offered by Hique, generated tuples preparation methods have the advantage interpretational overhead. Additionally, they can offer another layer of optimizations provided by compilers. Especially for the tuple preparation process, where tuples are prepared by iterating through fields, loop/recursive optimizations are promising. Furthermore, the costs of re-compilations can be amortized over the time span between two schema evolutions. Consequently, the tuple preparation interface seems to be a good candidate for pre-compiled approaches. Pre-compilation offers static type safety, i.e. type safety at compile time. Besides the obvious reduction of runtime type errors, static type checking can also be a performance optimization. When a compiler can prove that the source code contains only correct type relations, it can produce binaries without dynamic safety checks,

which in turn improves performance. With CPU and memory costs moving into the focus of query optimization, the potential performance improvements of pre-compiled tuple preparation should be investigated.

1.2 CONTRIBUTION

This work is based on the hypothesis that the transformation of tuples between two different data models imposes a significant computational overhead on query processing. In this context, this work examines the impact of interpretation-based and compiled approaches to marshalling and unmarshalling data.

As illustrated for query operators in Figure 2, interpretation-based approaches exhibit properties that impose negative effects on runtime performance. Namely, they introduce an execution overhead and larger memory footprint. Compiled approaches can remove these effects and further allow micro optimizations. As illustrated in Figure 2a, the ideal implementation of query, in terms of performance, is a hand-written program that executes a given query. Similarly, if the content of a row is known at development time, an hand-written implementation of tuple preparation can specify specialized and fast versions of marshalling and unmarshalling functions for each row. However, hand-coded approaches impose a great effort into the development of specialized DBMSs. For that reason, this work proposes template-based programming and compiler optimizations to increase the efficiency of (un)marshalling data, without increasing development efforts. The work postulates that RDBMSs preparing tuples from key-value stores can benefit from such compiled approaches.

The remainder of this work is organized as follows. Chapter 2 presents the background theory to this work. A system model is introduced and the interface affected by compiled or interpretation-based tuple preparation approaches is identified. Recent DB research is introduced and embedded into the context of this work. Further, the general concepts of compiler optimizations and polymorphism are introduced. Chapter 3 illustrates the design of compiled and interpretation-based approaches to tuple preparation. It compares the design and development effort of an interpretation-based approach using an object-oriented paradigm, a pre-compiled approach using standard C++ `structs` and a generation strategy using template-based programming. Chapter 4 presents experiments to examine the impact of tuple preparation in a general-purpose DBMS. Subsequently, it

compares the performance of compiled to interpretation-based tuple preparation in a series of specialized benchmarks. Finally, an experiment, comparing interpretation-based and generated approaches, with a statically implemented TPC-C benchmark on top of a key-value store is analyzed. Chapter 5 compares these experiments and discusses their results.

2

BACKGROUND THEORY

2.1 FIVE-LAYER SYSTEM MODEL

On a system engineering level, data independence is addressed by a reference architecture depicted in Figure 3 [9]. This reference architecture divides a DBMS into five layers: *non-procedural*, *navigational*, *record & access path management*, *propagation control* and *file management* layer. The ideas of structured programming and information hiding simplify the implementation of higher level components and enable the decoupled implementation of each layer. Each layer can be viewed as an abstract machine. A layer *i* utilizes the implementation of layer *i-1* to realize its own functionality.

If implemented strictly, this reference architecture can deteriorate the overall performance of the system. It introduces six formal interfaces that must be crossed before data can be stored on disk or requests can be served. Not only does the pure invocation of methods in lower layers create a performance overhead (parameter checking, error handling etc.), but data also has to be copied and converted from layer to layer to specific formats. To improve runtime performance, a reduction from the five-layer model to an effective two- or three-layer approach was proposed [9].

2.1.1 *Storage Engine*

In the five-layer model, the bottom three layers can be grouped into a storage engine. The layers are record & access path management, propagation control and file management. The record & access path management layer manages the internal records. This layer dereferences record identifiers to logical page level calls to the buffer manager. The propagation control layer provides the corresponding physical pages and blocks a linear address space, where fix/unfix operations determine data sets currently residing in memory. Finally, the file management layer is responsible for the physical application of

bit patterns in the form of files and blocks to external, non-volatile storage devices, often in collaboration with Operating System (OS) file management structures.

Improvements to the five-layer model [9] eliminate the file management layer by increasing the buffer size so that all logical pages can be allocated concurrently in memory. This adaptation was largely made possible by the developments in the volatile memory market, entailing a larger capacity of primary storage available to DBMSs (cp. Figure 1). The increased feasibility of IMDB led to various implementations like VoltDB [19] or SAP Hana [4]. These approaches not only gain a better performance by removing the interface, but also by eliminating the I/O costs of traditional disk-based systems.

When investigating the overhead of query processing engines, DB research justifiably concentrates on implementations on top of such IMDBs. I/O operations are not a bottleneck in these systems and query performance is guaranteed to improve when the computational performance of the system is increased. IMDBs avoid the usage of a buffer manager, as it introduces an additional layer of indirection. For instance, when a general-purpose DBMS accesses a specific tuple, it must compute disk addresses and then consult a mapping data structure to check if a corresponding block is in memory. IMDBs will avoid this overhead by accessing tuples based on their memory address. Accordingly, they abandon the benefits of a buffer pool. These advantages include the management of data sets too big to fit into primary memory, the natural support for write-ahead logging and isolation from cache coherence issues [8].

Graefe et al. [8] have shown that it is possible to use a buffer pool without a dramatic performance reduction. Through the adaptation of pointer swizzling [21], the authors improve the performance of a database with a buffer pool, achieving a performance close to that of IMDBs when the data set fits into memory. Concurrently, the approach increases the performance in cases where data sets almost fit into primary storage. Despite the possibility of I/O costs overshadowing memory and CPU costs, this approach shows that improvements specified for IMDBs are also applicable to general-purpose systems.

2.1.2 *Query Processing Engine*

In the five-layer model, a query processing engine consists of the non-procedural access and the navigational layer. The non-procedural access layer provides an access path independent, set-oriented API. The API offers access to the logical structures of a DBMS (e.g.

tuples, views, tables) via a declarative language like *SQL* or *XQuery*. It maps set-oriented structures to logical record accesses or scans on logical access paths, while also applying query optimization techniques to improve performance. The navigational layer dereferences logical accesses or scans by supplying their concrete internal record representation. Logical access paths are translated into scans or direct access on structures like B-trees and Hash Tables. The layer can additionally provide sorted record sets e.g. for joins and record-at-a-time modification access.

With a declarative interface it is often possible to evaluate a given query in several different ways with varying performances. Therefore, the non-procedural layer materializes Query Evaluation Plans (QEPs) for each query. The QEP in an RDBMS is a tree structure in which each node of the tree represents a relational operator. This representation allows optimizations on a macro level like plan enumeration strategies, cost modeling and algorithmic improvements.

As the optimization focus moved from I/O costs to CPU and memory costs, modifications to the five-layer model were proposed [9], which included the replacement of the top-most layer. A proposed *access module* incorporates the QEP, by pushing as much of the query preparation as possible from processing to compilation time. This improvement has the potential to eliminate the impact of the non-procedural layer under dynamic conditions.

Implementations can range from interpreters of non-procedural statements, referring to the navigational layer directly, to the full query preparation at compile time. The latter amortizes the additional cost of preparing one ad-hoc query over repetitive query execution. There are also mixed approaches conceivable where only intermediate parts of queries are prepared via access modules. Specific interpreters can then utilize these artifacts, whenever possible. On the one hand, a query prepared at compile time relinquishes the query response time from optimization efforts. On the other hand, the lack of precise statistical data during compile time can lead to suboptimal execution plans. Moreover, all preparation approaches introduce dependencies to the meta-data, valid at the time of compilation which may be invalidated by schema evolution.

Conventional query execution engines interpret a QEP with the *volcano model* [7]. This model's basic assumption is that every physical implementation of a relational-algebraic operator produces and consumes a stream of tuples. The volcano model defines an abstract interface with three functions: *open()* to start an iterator and initialize the internal

state, *next()* to produce another tuple and *close()* to designate the end of the operation and free up resources. Each physical operator of the QEP is implemented with the use of this abstract interface, which enables the combination of arbitrarily complex query evaluation as well as a pipelined execution of all operators on a given tuple. Moreover, it facilitates intra- and inter-operator parallel processing capabilities of DBMSs on multicore platforms.

Recently, there have been several proposals to increase the computational capability of DBMSs, which describe the characteristics of the volcano model as a source of query engine overhead.

Some proposals concentrate on expanding the data granularity utilized by operators within queries. MonetDB [14], for instance, uses an operator-at-a-time scheme to reduce the overhead of per-tuple interpretation. In this model, data is fully materialized inside each operator, which then completely processes its input and invokes the next stage. While this concept reduces the interpretational overhead, it also removes the pipelining benefits of the volcano model. An improvement was made in MonetDB/X100 [22], which applies *vectorized execution* (cp. Figure 2). Instead of materializing a whole table, an already vertically decomposed table is further partitioned horizontally into vectors, on which each operator of the query is evaluated. Experiments showed that data-intensive query performance could be improved by two orders of magnitude compared to traditional iterator model implementations.

Other proposals concentrate on utilizing code generation to improve query processing performance. This allows another layer of optimizations to the query preparation process in form of compiler optimizations. Early adaptations can be found in System R [1], which utilizes a code generator that produces assembly code based on a given query. More recently, Rao et al. [17] reduced runtime overhead by exploiting the Java Reflection API in the context of an IMDB. The proposal dynamically compiles a given query into Java Bytecode, which can be interpreted by the Java Virtual Machine (JVM). Although still relying on the iterator model and limited in its capability, the approach removes virtual function call overheads. Query processing performance was improved on average by factor two, compared to interpreted plan performance.

Hique [12] improves the concept by dynamically instantiating C source code specific to a given query and hardware configuration. This source code is then compiled, dynamically linked and used for query processing. The iterator model is eliminated in

a three-step approach. First data is materialized, applying all predicates and dropping unnecessary fields. Thereafter, the *holistic algorithm instantiation* generates source code for each operator from templates. At last, the single operators are combined to form a single function. Krikellas, Viglas, and Cintra [12] showed that the prototype system outperformed a general-purpose DBMS by two orders of a magnitude and vectorized approaches by a factor of four (cp. Figure 2b).

A similar approach is taken by Neumann [16]. He proposes *HyPer*, a system also avoiding the iterator model. Like Hique, HyPer also compiles a relational algebra tree into low level machine code. HyPer, however, utilizes an LLVM compiler backend [13] to increase the efficiency of on-demand compilation. Even further, it applies a data centric approach to the query compilation process. By reversing the direction of the data control flow, the compiled queries push tuples from producing to consuming operators. Consequently, tuples can be left in CPU caches longer and operators are cheaper to compute. To evaluate the new paradigm, Neumann experimented with HyPer in a hybrid Online Analytical Processing (OLAP) and Online Transactional Processing (OLTP) environment. He could show a significant improvement in compile time comparing a LLVM version to a C++ version as well as a performance boost in query processing. HyPer outperformed a general-purpose DBMS by two orders of a magnitude and vectorized approaches by one order of magnitude.

Finally, a comprehensive study [18] compares the approaches of code generation to vectorized execution in operators. In several case studies, the authors conclude that the best results can be achieved when a combination of both approaches is applied.

2.1.3 *Interface between Query Processing and Storage Engine*

Similar interpretation-related strategies can be observed in the interface between query processing and storage engine. As illustrated in Section 1.1, RDBMSs often rely on key-value stores to implement a storage engine, such that key-value stores provide physical records corresponding to a key to the upper layers, while hiding details of the actual implementation of the storage and propagation.

However, query processing engine and key-value store operate on different data models. For that reason, one major functionality of the navigational layer is the translation from physical records into logical records and vice versa. To that end, the navigational

layer unmarshals byte arrays into their logical, i.e. typed, representation and vice versa marshals logical records into a byte array representation.

The marshalling function of a row receives a byte array. It iterates over the fields of a row and extracts each field's value into an in-memory representation of the tuple. The other way around, the unmarshalling function of a row receives the in-memory representation. It prepares a byte array for the storage in a key-value store by iterating over the fields of a row and extracting the field values from the in-memory tuple representation. It concatenates those values into a byte array and returns this array, such that it can be stored in a key-value store.

The data type of each field within a row determines the implementation of both functionalities. For instance, an unmarshalling function of a field implementation that realizes the `integer` data type must specify a different control flow than the unmarshalling function of a field implementing the `string` data type. While `integer` data could be directly interpreted as bytes, `string` must be prepared in a specific way to allow efficient storage. As the length of the `string` data is variable, marshalling the data generally means extracting the current length of the `string` and placing it before the actual value of the data as a header. This way, when data is later unmarshalled into its logical representation, the unmarshalling function is able to determine how many bytes to extract into the logical `string` representation.

RDBMSs like MySQL utilize an object-oriented paradigm called dynamic polymorphism to realize this functionality. Similar to the volcano model, this is an interpretation-based approach. In this model, rows are represented as a collection of abstract field objects. This dynamic representation of an in-memory tuple offers clear advantages in terms of logical data independence. Through schema evolution, a table, and thus the row of a table, can, at any given point in time, be changed. Dynamic polymorphism facilitates such schema evolution inside an RDBMS. Compared to compiled approaches, it does, however, come with performance penalties.

2.2 STATIC VS. DYNAMIC POLYMORPHISM

Polymorphism is a programming facility in which an abstract object provides a common interface to entities of different types. It represents a way to access a common functionality of a group of entities without knowing the exact implementation. Instead, the inter-

face describes the function in terms of arguments and return values. Inheriting entities can then implement the concrete functionality, adjusted to their specific conditions.

This concept introduces a level of indirection that must be resolved. The resolution can either occur at runtime, called dynamic polymorphism, or at compile time, called static polymorphism. While dynamic polymorphism offers a high degree of flexibility, it has a negative impact on performance. Compared to static polymorphism, it requires additional instructions as well as an increased memory footprint.

Dynamic polymorphism often utilizes a special form of late binding known as the *vtable* (or virtual table). For instance, in C++, the content of the vtable is determined by methods that are declared as `virtual`. For each method declared virtual, the compiler creates an entry in a derived object's vtable pointing to the physical address of the implemented method.

Thus, in terms of performance, the use of dynamic polymorphism introduces a natural execution overhead by adding a layer of indirection to the processing of functions that is resolved at runtime. In a direct function call, the processor can jump directly to a hard-coded address deposited in a program's binary file. In a virtual function call, the address must be loaded from the vtable. The vtable pointer must first be extracted from the current object into a register. The vtable is then traversed to identify the address of the implemented method and afterwards, this function address is copied to a register. Finally, the processor takes the address from the register and jumps to it.

In terms of memory footprint, the dynamic polymorphism approach increases object sizes, since one additional pointer must be stored per object. When an instance of a class object is created, a pointer inside the instance is automatically allocated and assigned. It points to the vtable for that class. On 64-bit systems, the size increase of an additional pointer is 8 bytes per object. While this increase can be negligible, it can also be a substantial overhead for small objects. Concretely, the size increase can mean that less objects fit into the L1-cache of a processor. The number of objects, readily available in this cache structure, directly influences the number of accesses to memory.

To illustrate the performance impact of L1 cache misses, Figure 4 depicts the memory latency of an Intel® Xeon X5670 CPU. Yellow-colored blocks represent block sizes that fit into the L1 cache. Brown-colored blocks correspond to block sizes too big to fit into the cache hierarchy, thus blocks must be fetched from main memory. Depending on the step size, an L1 cache miss can have a substantial impact on the performance of a system.

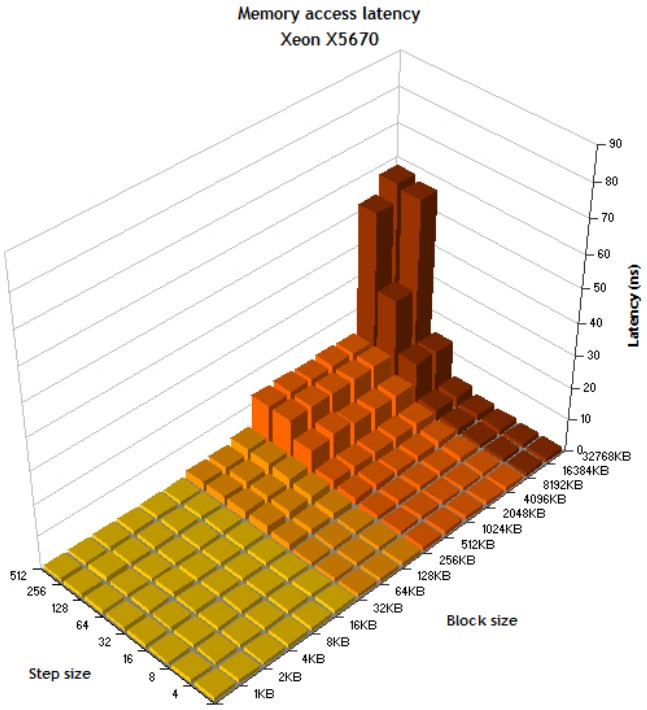


Figure 4: Memory Latency of Intel® Xeon X5670 [20].

Finally, a method declared as `virtual` cannot be inlined. Modern compilers often employ inlining to remove the overhead of calling and returning from a function as well as facilitate opportunities for optimizations (cp. Section 2.3). With dynamic polymorphism, a compiler cannot determine which concrete function is going to be called, thus making it impossible to inline it.

Static polymorphism, on the other hand, is the imitation of polymorphism in programming code. The actual resolution of polymorphism is executed at compile time. Thus, this method postulates that the polymorphic behavior is known at compile time. Consequently, static polymorphism executes faster than dynamic polymorphism. It avoids the runtime overhead and increased memory footprint as no vtable is needed to dispatch a polymorphic function. A compiler analyzes the polymorphic structure and decides which function is going to be called. Instead of automatically creating the vtable and distributing pointers to affected classes, compiler/linker can hard-code the corresponding function's address into a program's binary.

2.3 COMPILER OPTIMIZATIONS

The primary goal of modern compilers is to transform high-level programming constructs into low-level machine code. Moreover, they add a level of optimizations to improve the efficiency of programs in two dimensions: both in time (i.e. runtime performance) and in space (i.e. memory footprint). Overall, compiler optimizations aim to decrease the number of instructions necessary to compute an output.

Firstly, as intraprocedural analysis and optimization yields better results than interprocedural, the use of function *inlining* can improve performance. Compilers attempt to embed a called function's content into the calling code, instead of executing an actual function call. Thereby, instructions in the called and the calling function can be eliminated. Eliminated instructions comprise instructions preparing stack and registers for the use of a function (i.e. function prologue) and instructions restoring stack and registers at the end of a function (i.e. function epilogue). The primary benefits of inlining originate from other techniques of optimizations. After its application to a function, interprocedural optimization techniques can be applied to a larger function body, further improving performance. However, as inlining duplicates a function body, it deteriorates instruction space usage. Specifically, its excessive use can also hurt overall system performance when inlined code consumes too much of the instruction cache. Therefore, the technique should generally only be applied to small and frequently called functions.

Hardware-specific optimization techniques target memory access reduction and efficient execution pipeline usage. With instruction pipelining, modern CPUs often capitalize on instruction-level parallelism. Instructions are broken down into a sequence of steps, such that different steps can be executed in parallel. Compiler optimization techniques, like instruction scheduling, aim to keep an execution pipeline filled with independent instructions, maximizing instruction-level parallelism.

On the memory access level, compilers attempt to increase the spatial and temporal locality of reference of memory objects. The compiler tries to group accesses to data and code together, so that they can be efficiently distributed to registers and caches. Strong locality of reference amplifies the efficiency of the memory hierarchy, through caching, prefetching and branch prediction.

Compilers can also employ optimization techniques that target loops to avoid loop-invariant code or to recognize induction variables. They can partially or completely un-

roll loops, when the number of iterations is known at compile time. This removes instructions addressing jumps inside the loop and count de-/increments. Similar techniques can be applied in recursive variants, where recursive functions can be inlined.

3

IN-MEMORY TUPLE IMPLEMENTATION DETAILS

A tuple in the relational model is a similar concept to that of a row in a table. The equivalent to an attribute in the relational model is the field of a row. To implement and execute operations in the relational model, an RDBMS needs to implement field and row structures.

The following chapter will illustrate interpreted and compiled versions of in-memory tuples, specifically in context of an embedded key-value store. While all methods of field and row implementations are affected by whether they are interpreted or compiled, this work will concentrate on the (un)marshalling aspects of such an implementation. Nevertheless, the effects discussed can be translated to access methods as well.

3.1 INTERPRETED REPRESENTATION

An operator processing a row of a table initializes an interpreted in-memory tuple by instantiating a row and assigning abstract field objects. Listing 2 demonstrates the concept by an example of a statically implemented update transaction for the NEW_ORDER table of the TPC-C benchmark.

Initially, a catalog is consulted to determine which fields to initialize. After the row is instantiated, the underlying key-value store is accessed via a `kvs_handle` object to retrieve a byte array, which is then passed to a marshalling method of the `Row` object. The method extracts the byte values for each of the values from the array and organizes them into logical `Field` object instances. Operations on `Field` objects are executed by calling virtual getter/setter methods (cp. Listing 4). Finally, when the logical row needs to be stored in the key-value store, the `Row` class offers a `unmarshall()` method that returns the logical representation as a byte array. The byte array can then be submitted to the key-value store via the `kvs_handle`.

Listing 3 illustrates an excerpt of a dynamic implementation of a row. An essential part of the `Row` implementation is the `fields` member variable. For each column defined in

Listing 2: Static `neworder` Update Implementation

```
void update_neworder_trx(KeyValueStore kvs_handle, uint field_pos, int value,
    string key){
  CATALOG_ENTRY new_order_CE = catalog.find("TPCC_NEWORDER");
  Row no_row= new Row();
  for(auto field : new_order_CE.fields){
    switch(field.type){
      ...
      case INTEGER: no_row.add_field(new Field_int());break;
    }
  }
  no_row.marshall(kvs_handle.get(key));
  no_row.fields.at(field_pos)  /*<- Vector-style access */
                 ->set_int_val(value);
  kvs_handle.put(key_buffer, no_row.unmarshall());
}
```

a table, this array contains an abstract `Field` object. As illustrated in Listing 2, methods processing a logical row can utilize this variable as a handle to access individual field values.

Listing 3: Dynamic Row Implementation

```
class Row {
  std::vector<Field*> fields;
  void add_field(Field* field){ ... }
  void remove_field(uint idx){ ... }
  void unmarshall_row(StreamType& dest){
          dest.resize(this->get_size());
          std::vector<char>::iterator it = dest.begin();
          for(uint i =0; i< fields.size(); i++){
            if (!((*fields[i])->is_null()))
              (*fields[i])->unmarshall(it);
          }
  }
  void marshall_row(StreamType& src){
          for(uint i=0; i< fields.size(); i++){
              (*fields[i])->marshall(src.data());
          }
  }
}
```

Typically, a dynamic row implementation specifies some variation of the illustrated `marshall_row()` and `unmarshall_row()` methods. If the field is not `null`, the method calls the respective (un)marshalling methods on each of the abstract `Field` objects. The

`Row` class's methods manage pointers into the byte arrays, such that (un)marshalling implementations of fields can extract values without searching in the byte array.

The `fields` variable's type is defined as an STL class template `std::vector`, a sequence container representing arrays that can change in size. Other implementations may differ, but the concept of a flexible container remains an essential part of an RDBMS implementation. It allows removing or adding fields to a row at runtime, which is a central element of schema evolution.

Listing 4: Dynamic Field Implementation

```
class Field{
  public:
    virtual void marshall(char*& in)=0;
    virtual void unmarshall(std::vector<char>::iterator&)=0;
    virtual size_t get_size()=0;
    virtual set_int_val(int val)=0;
    ...
}
class Field_int : public Field {
  int value;
  Field_int() {}
  Field_int(int val): value(val) {}
  void marshall(char*& in);
  void unmarshall(StreamType::iterator& out);
  void set_int_val(int val){ value=val;}
  size_t get_size(){return sizeof(int);}
};
class Field_varchar : public Field {
  std::string value;
  Field_varchar() {}
  Field_varchar(std::string val): value(val) {}
  ...
  size_t get_size(){return sizeof(size_t)+value.length();}
};
```

Listing 4 shows excerpts of an example implementation of an interpreted in-memory field. An abstract `Field` class defines the interface methods of a field. Concrete implementations of a field inherit from this abstract class. Typically, each such concrete implementation is bound to a data type. As the actual implementation of a field's functionality differs for each data type, common methods of the `Field` object are declared `virtual`.

The `get_size()` method returns the size that a field's value consumes in its representation in a byte array. For example, a `Field_int` implementation returns the byte size of an integer. In contrast, for a `Field_varchar` implementation the size must first be computed by retrieving the variable length of the `string` variable. Besides, it adds size to the return

value to be able to store the length with the string. This concept of different control flows for the same functionality manifests the polymorphism approach discussed in Section 2.2. Similary, the implementation of marshalling and unmarshalling methods differs depending on the data type a field implementation represents. Listing 5 demonstrates the code for an (un)marshalling implementation on an integer-typed field and varchar-typed field. In general, an RDBMS needs to specify an implementation of an abstract field for every data type of the relational model that requires specialized in-memory or on-disk storage. More precisely, not all data types offered in the *non-procedural layer* necessarily require a specific field implementation.

Listing 5: Dynamic (Un)Marshalling Implementation

```
1  void Field_int::marshall(char*& res){
       char tmp[4]; invert_endianess(tmp, res, 4);
       int* val = reinterpret_cast<int*>(tmp);
       value=*val;
       res+=sizeof(int);
6  }
   void Field_int::unmarshall(std::vector<char>::iterator& out){
       const char* ptr = reinterpret_cast<const char*>(&value);
       convert_endianess(out, ptr, sizeof(int));
   }
11 ...
   void Field_varchar::marshall(char*& res){
     size_t* len_ptr= reinterpret_cast<size_t*>(res); res += sizeof(size_t);
     if (*len_ptr == 0u){
         value= std::string(); return;
16   }
     value= std::string(res,*len_ptr);
     res+=*len_ptr;
   }
   void Field_varchar::unmarshall(std::vector<char>::iterator& out){
21   size_t len= value.length();
     const char* ptr = reinterpret_cast<const char*>(&len);
     std::copy(ptr,ptr+sizeof(size_t),out);
     out+=sizeof(size_t
     std::copy(value.data(),value.data() + len, out);
26   out+=len;
     }
```

Other possible implementations represent polymorphic fields with UNIONS on data types and SWITCH/CASE statements inside the (un)marshalling functions (cp. Listing 12 in Appendix A). Although these implementations are not strictly considered as dynamic polymorphism, they simulate the same behavior. Like vtables, SWITCH/CASE statements represent a sequence of jumps to addresses stored in a look-up table. To realize

SWITCH/CASE structures, field implementations would explicitly have to store their logical data type, e.g. SQL types in an RDBMS. This again increases the memory footprint of an in-memory field representation. On top of that, the use of UNIONS increases a field's size, as union member variables occupy a memory area equal to the size of the biggest member of the union.

3.2 COMPILED REPRESENTATION

A compiled implementation of an in-memory tuple representation can be realized in two ways: `struct` implementation or static polymorphism. With `struct` implementations, a developer or a tool specifies a static tuple as a struct or class and within (un)marshalling and access methods. With static polymorphism, the developer specifies the structure of the tuple and lets a C++ compiler generate access functions and (un)marshalling functionality.

This section will introduce both concepts. In both cases code generation is possible. However, static polymorphism is better integrated with the programming language. Another common characteristic of these approaches is that the data structures representing a tuple are fixed at compile time. This is why schema evolution at runtime cannot be implemented without recompilations with either approach.

3.2.1 `struct` *Implementation*

Listing 6 illustrates excerpts of a data structure realizing a tuple of the `customer` table of the TPC-C benchmark. The depicted struct serves as a container for a number of heterogeneous values. For each field of a tuple a member variable in the body of the corresponding struct has to be specified.

Functionalities of a row, like (un)marshalling, are implemented in methods of the tuple. To illustrate the concept, Listing 7 extends Listing 6 with a `struct` implementation of an unmarshalling function for a tuple of the TPC-C `customer` table.

The individual components of this method specification are easy to assemble. Each variable's data type determines the control flow of marshalling or unmarshalling the variable. Hence, the specification of methods can be assembled from the specifications for the data type of each individual variable. However, in this model, the actual value

Listing 6: `struct` Tuple Implementation

```
struct st_tpcc_static_customer_tuple{
    int c_w_id;
    int c_d_id;
    int c_id;
    double c_discount;
    ...
    std::array<char,2> c_middle;
    std::string c_data;
}
```

Listing 7: `struct` Unmarshalling Implementation

```
struct st_tpcc_static_customer_tuple{
  ...
  void unmarshall(StreamType& res){
          res.resize(this->size());
          StreamType::iterator out=res.begin();
          const char* ptr = reinterpret_cast<const char*>(&c_w_id);
          convert_endianess(out, ptr, sizeof(int));
          ...
          len= c_data.length();
          ptr = reinterpret_cast<const char*>(&len);
          std::copy(ptr,ptr+sizeof(size_t),out);
          out+=sizeof(size_t);
          std::copy(c_data.data(),c_data.data() + len, out);
          out+=len;
  }
}
```

of a field can only be obtained by addressing the named member variable of a `struct`. Thus, methods accessing a row's field values must be either hand-written by developers or generated by specialized tools.

3.2.2 *Static Polymorphism Implementation*

Languages like C++ offer the concept of template metaprogramming to realize static polymorphism. This technique allows developers to specify data structures, compile time constants and functions such that a compiler can generate source code from them, i.e. compile time generation. The STL is a software library that heavily uses template metaprogramming. It offers components for containers, algorithms, functions and iterators.

A container structure, the STL offers, is the `std::tuple` implementation, which is a class template that allows the storage of heterogeneous elements. Similar to the `struct` concept, this class template offers the ability to pack a fixed-size collection of possibly different objects together inside a single object. Listing 8 shows excerpts of an exemplary type definition of an STL tuple that represents a logical tuple of the `orderline` table of the TPC-C benchmark.

Listing 8: STL tuple definition of a `orderline` tuple

```
using TPCC_ORDERLINE_TUPLE= std::tuple<int, //0: OL_W_ID
        int, //1: OL_D_ID
        int, //2: OL_O_ID
        int, //3: OL_NUMBER
        int, //4: OL_I_ID
        double, //5: OL_DELIVERY_D(ate)
        double, //6: OL_AMOUNT
        int, //7: OL_SUPPLY_W_ID
        double, //8: OL_QUANTITY
        std::array<char,24>> //9: OL_DIST_INFO
;
```

Instead of accessing data members via their name, like in a `struct`, the elements of a tuple are accessed per their order in the tuple. Still, the selection of a data member within such a tuple is done at template-instantiation level, i.e. at compile time. The STL offers helper function templates `std::get<I>(tpl)` that return references to the object located at compile time constant index `I` of a tuple instance `tpl`.

As access to data elements of STL tuples must be specified at compile time, methods accessing these elements must be as well. While a implementation similar to the `struct` implementation (cp. Listing 7) is still a possibility, the class template definition allows the usage of generated approaches. A template-oriented generation approach is especially advantageous for methods that must iterate through all elements of a tuple, e.g. the (un)marshalling methods of RDBMSs discussed here.

For illustrational purposes, Listing 9 and Listing 11 show excerpts of an generated, recursive approach to an unmarshalling function for a STL tuple. The `unmarshaller()` function template serves as the interface to the control flow of the unmarshalling process. This function template by itself does not specify a function. Without a call in the implementation, no code is generated from this specification. To instantiate the template function, a calling method needs to provide a template argument. The compiler will then

Listing 9: Recursive Unmarshalling Functions for a STL Tuple

```
template <class T>
inline void unmarshaller(const T& obj, StreamType::iterator& res) {
    unmarshal_helper<T>::apply(obj,res);
}
template <class... T>
struct unmarshal_helper<std::tuple<T...>> {
      static void apply(const std::tuple<T...>& obj, StreamType::iterator& res) {
          unmarshal_tuple(obj, res, int_<sizeof...(T)-1>());
      }
}
template <class T, size_t pos>
inline void unmarshal_tuple(const T& obj, StreamType::iterator& res, int_<pos>) {
        constexpr size_t idx = std::tuple_size<T>::value-pos-1;
        unmarshaller(std::get<idx>(obj), res);
        unmarshal_tuple(obj, res, int_<pos-1>);
}
template <class T>
inline void unmarshal_tuple(const T& obj, StreamType::iterator& res, int_<0>) {
        constexpr size_t idx = std::tuple_size<T>::value-1;
        unmarshaller(std::get<idx>(obj), res);
}
```

generate an actual function. Template arguments can either be explicitly defined in the call or C++ will deduce the template argument from the function argument.

When the compiler discovers a call to the `unmarshaller()` method, it will generate the `unmarshaller()` and `unmarshal_tuple()` methods as well as an `unmarshal_helper` class if it doesn't already exist. It generates an `unmarshaller()` function for each distinct tuple definition. It also creates one `unmarshal_tuple()` function per tuple and index position in the tuple, i.e. for a tuple with degree 2 it generates two `unmarshal_tuple()` functions (cp. Listing 10). Finally, it creates methods from the `apply()` method template, for each call to `apply()` with a distinct template parameter. These static `apply()` methods contain the actual unmarshalling functionality for each data type.

Listing 10 shows an abstract illustration of template parameter replacement for a small tuple of two fields. Template arguments that replace the template parameters in Listing 9 are highlighted in red. It should be noted that the gcc compiler does not generate the high-level C++ code shown. It rather directly translates equivalent functionality into assembly code.

Listing 10: Abstraction of Compiler Output for Template-Based Unmarshalling Functionality.

```
inline void unmarshaller(const std::tuple<int,double>& obj, StreamType::iterator&
    res
    unmarshal_helper<std::tuple<int,double>>::apply(obj,res);
}
struct unmarshal_helper{
      static inline void apply(const std::tuple<int,double>& obj, StreamType::
          iterator& res) {
          unmarshal_tuple(obj, res, int_<2-1>());
      }
}
inline void unmarshal_tuple(const  std::tuple<int,double>& obj, StreamType::
    iterator& res, int_<1>) {
        constexpr size_t idx = 1-1; // 2-1 Generated at compile time
        unmarshaller(std::get<0>(obj), res);
        unmarshal_tuple(obj, res, int_<0>());
}
inline void unmarshal_tuple(const  std::tuple<int,double>& obj, StreamType::
    iterator& res, int_<0>) {
        constexpr size_t idx = 2-1;
        unmarshaller(std::get<1>(obj), res);
}
struct unmarshal_helper {
      static inline void apply(const  std::tuple<int,double>& obj, StreamType::
          iterator& res) {
          rawr::unmarshal_tuple(obj, res, int_<2-1>());
      }
}
```

A generic version of the `unmarshal_helper` and the corresponding `apply()` can be found in line 8 of Listing 11. Like the specializations of the virtual (un)marshal methods of the `Field` class in the dynamic polymorphism approach (cp. Listing 5), they specify the process of casting and copying the data of a field value into a byte array and maintaining the cursor/pointer into the byte array. However, instead of specifying a type for the input parameter `obj`, it provides a template, such that the `apply()` methods needed can be generated by the compiler. When the unmarshalling specification of a data type differs from the generic approach, a new `unmarshal_helper` class must be defined. For instance, beginning in line 1, an `unmarshal_helper` is implemented as a specialization of the unmarshalling process of a string field. Through function overloading, the compiler recognizes specializations at compile time and replaces calls to the generic `apply()` with a call to the specified one.

Listing 11: Recursive Unmarshalling Functions for a STL Tuple cont'd

```
template <class tuple_type> struct unmarshal_helper<std::string> {
      static inline void apply(const std::string& obj, StreamType::iterator& res)
          {
3         unmarshaller(obj.length(), res); //Store string length first
          std::copy(obj.data(),obj.data()+obj.length(),res);
          res+=obj.length();
      }
}
8 template <class T>
  struct unmarshal_helper {
      static inline void apply(const T& obj, StreamType::iterator& res) {
          const char* ptr = reinterpret_cast<const char*>(&obj);
          std::copy(ptr,ptr+sizeof(T),res);
13        res+=sizeof(T);
      }
}
```

At runtime, the calls to `apply()` methods will be resolved to their generated implementations. Through function overloading the `unmarshal_helper` class will provide a method to unmarshal a field for each type specified in a tuple.

As a hint to the compiler, all functions are declared INLINE. As sizes of the tuples are known at compile time and thus the recursion depth as well, the compiler can then perform additional optimizations like recursion inlining.

Marshalling functions or functions determining the size of a byte array representation of a row can be implemented in a similar fashion. This static polymorphism approach

allows the specification of dynamic-polymorphism style functions, but resolves the ambiguity at compile time.

4

EXPERIMENTS AND RESULTS

The underlying hypothesis of this work states that pre-compiled tuple preparation can offer substantial performance improvements in a DBMS. To verify this hypothesis, the following chapter presents several experiments. Initially, Section 4.1 characterizes the execution environment of the experiments. Section 4.2 analyzes the overall performance impact on query execution imposed by tuple preparation. An analysis of the CPU profile of MariaDB [5], a popular RDBMS, is performed. Methods and functions relevant to tuple preparation are identified and their effort is classified in the context of query execution performance. Section 4.3 evaluates the approaches presented in Chapter 3 in a series of specialized benchmarks. Finally, Section 4.4 compares the dynamic to the static polymorphism approach in context of statically defined TPC-C benchmark transactions.

4.1 ENVIRONMENT

4.1.1 *Key-value Store*

Experiments with an embedded key-value store were executed on top of *Zero*, a fork of *ShoreMT*. *ShoreMT*, implemented at Carnegie Mellon University and later at EPFL, focused on improving the scalability of storage operations in a multi-processor environment. The latest fork of *Zero*, developed at HP Labs and TU Kaiserslautern, supports Foster B-tree data structures with ACID semantics, orthogonal key-value locking, an improved Lock Manager design and techniques that enable instant restart and recovery.

4.1.2 *System Specifications*

All experiments were carried out on a server with dual Intel Xeon X5670 processors (cp. Table 1 for specifications). The system was equipped with 96 GB of 1333 MHz DDR3 memory. Two Samsung SSD 840 with 265GB were used to store log and database files.

INTEL X5670 PROCESSOR SPECIFICATION	
CPU Frequency	2933MHz
Number of Cores	6 (2 logical cores per phyiscal)
L1 cache	6x 32KB instruction cache 6x32KB data cache
L2 cache	6x 256KB
L3 cache	12MB

Table 1: Specification of Intel® Xeon X5670

The installed OS was Ubuntu 14.04 LTS with Linux kernel version 3.13. C++ implementations were compiled with gcc 4.9.4. Profiling data was collected with the Linux profiler *perf* at a sampling rate of 2000 Hz.

4.1.3 *TPC-C Benchmark*

Benchmarking experiments were executed based on the TPC-C specifications [3]. TPC-C is an OLTP benchmark that simulates a specified number of terminal operators connected to a database of a wholesale business. It defines tables for warehouses, districts, customers, old and new orders, items, stocks and order lines. The size of each of the tables depends on a scaling factor. It also specifies five available transaction types: submitting a new order, submitting a payment, delivery, checking an order status and testing and increasing a stock level.

4.2 IMPACT OF TUPLE PREPARATION

A part of the premise of this work is that tuple preparation has a substantial impact on the performance of a DBMS. In order to have that impact, the effort of tuple preparation must be a significant portion of the total effort of query processing. To analyze the impact of tuple preparation in a general-purpose DBMS, a qualitative performance analysis of MariaDB was performed. MariaDB is a community-developed fork of MySQL. It pro-

vides a pluggable storage engine architecture which allows the utilization of key-value stores as storage engines. It realizes the query processing capabilities of an RDBMS and provides an abstract `handlerton` interface that defines the API of a storage engine. By specifying methods that create tables and add, delete, update or retrieve rows, key-value stores can be plugged into the MariaDB system. To normalize experiment data of tuple preparation, a storage engine plugin for Zero was developed.

Like most general-purpose DBMSs, MariaDB realizes tuple preparation with the interpreted approach. An abstract field object specifies methods for (un)marshalling and accessing a field of a row. Per connection and table, the system initializes a `TABLE` object that contains an array of abstract field objects. This array represents a logical row in MariaDB.

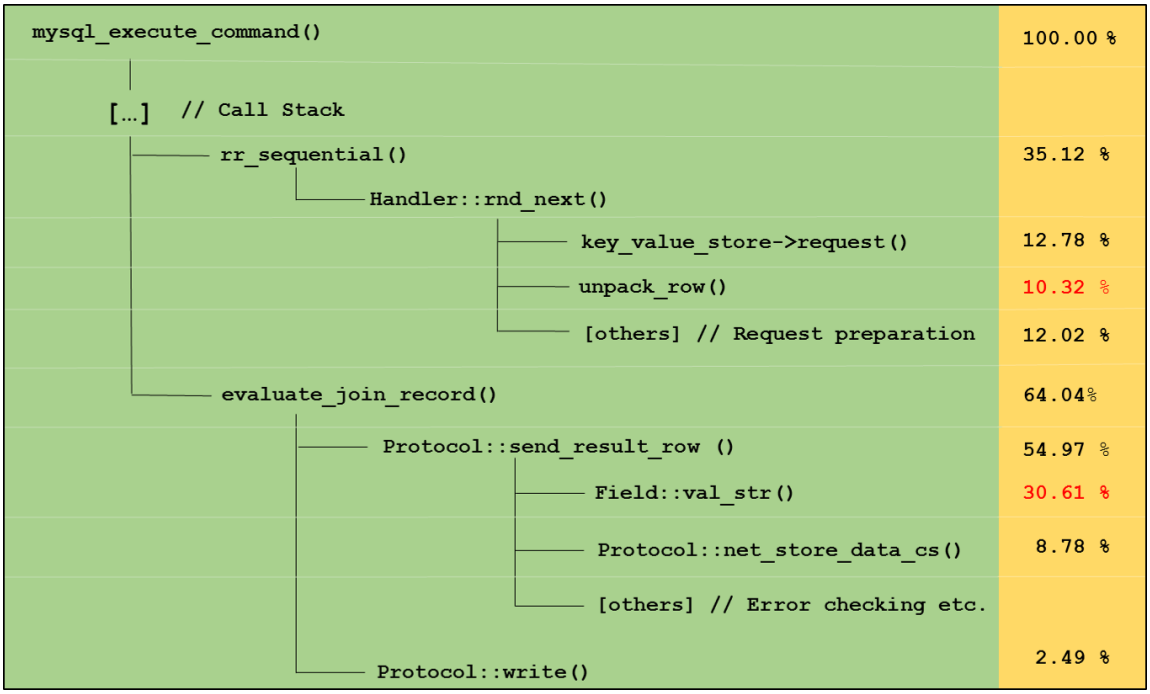


Figure 5: Simplified CPU profile of a TPC-C customer table scan on MariaDB

To illustrate the performance impact of marshalling tuples from a key-value store, Figure 5 shows a simplified excerpt of a CPU profile of a table scan on a TPC-C customer table of scaling factor 4, i.e. a table with 120000 customer rows. Query execution in MariaDB begins with the `mysql_execute_command()` method. For better readibility, the call stack in Figure 6 was reduced to children of this method. Query parsing, analysis and optimization methods are not illustrated, because their share of the total effort of query processing for a simple query like a table scan is minimal.

The CPU time consumption of a scan execution, illustrated in Figure 5, is divided between the `rr_sequential()` and `evaluate_join_record()` methods. The former calls a handler's `rnd_next()` methods to extract the data from the key-value store. The effort of the `rnd_next()` methods is divided again between two categories: key-value store access and tuple preparation. The former includes efforts to prepare a request, initialize data structures and extract a byte array from the key-value store. These efforts added up to 24.8 % of the total scan processing effort. An `unpack()` method prepares tuples for processing in the MariaDB system, by extracting values into a buffer and distributing pointers to `Field` objects. This process accounted for 10.32 % of the total query processing effort.

In a table scan, `evaluate_join_record()` prepares the data for its submission to the requesting application, in this case to a local *mysql* terminal application. Aside from error checking and protocol initialization efforts, the method calls the `Protocol::send_result_row()` function to extract the values of each `Field` object of a result row into a network data stream. `Protocol_text::store()`'s efforts to extract the field values by calling the `val_str()` functions of `Field` objects added up to 30.61 % of the total effort of a table scan. In contrast, 11.27 % of the CPU time was spent on copying the extracted data into a network data stream.

This analysis of a scan operation on a TPC-C customer table showed that the actual marshalling of a tuple from its key-value store representation into MariaDB's in-memory representation of a row took more took little more than 10.32% of the total execution time of the scan query. However, in MariaDB's case the field values are not stored as typed values inside a `Field` object. Rather, MariaDB stores the in-memory representation of a row in one or multiple byte buffers. Whenever a typed value is needed, the field implementations return it on demand by copying and casting from the in-memory representation. Hence, the marshalling process is also executed in the evaluation of records. Adding the effort of the evaluation of field values to the effort of marshalling eventuates in a total percentage of the whole query processing effort to 40.93 %. Although limited in its expressiveness, this analysis showed that improvements to the tuple preparation process can have an impact on the overall query processing performance of an RDBMS.

Figure 6 illustrates the CPU profile of inserting 10000 tuples into the TPC-C customer table. Again, the actual execution of the query starts with the `mysql_execute_command()` method. Hence, query parsing, analysis and optimization efforts are not illustrated here,

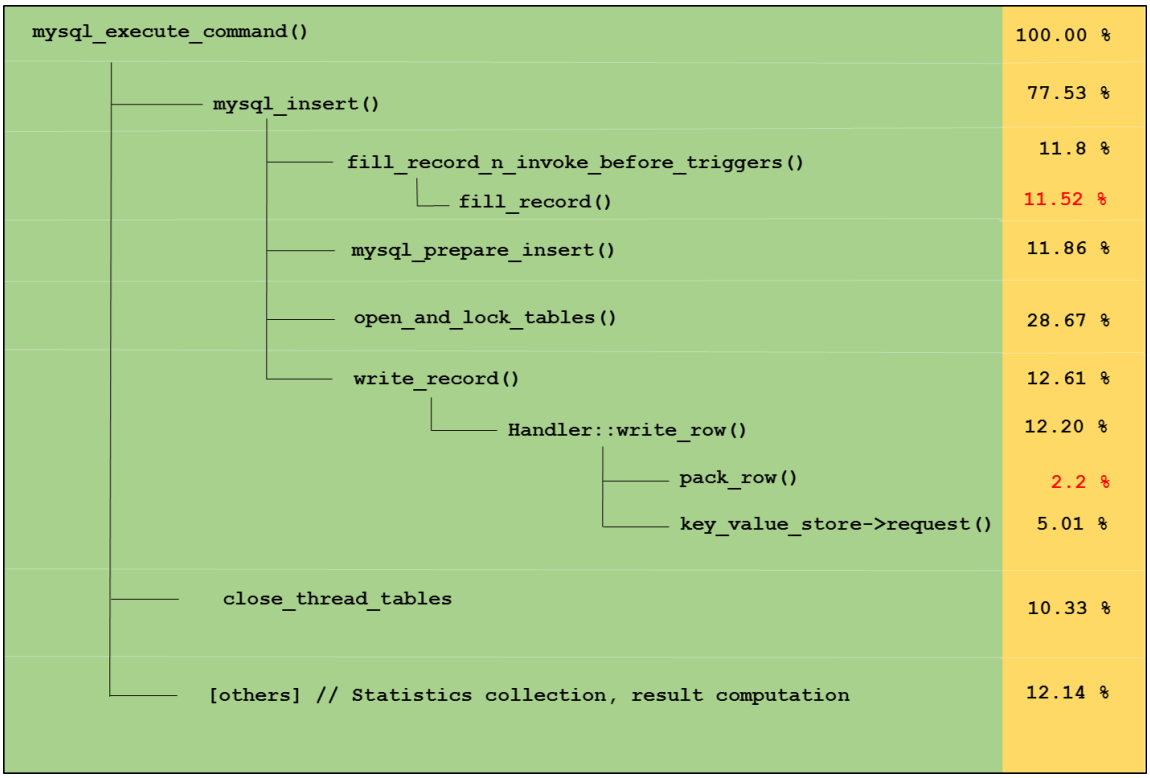


Figure 6: Simplified CPU profile of inserting TPC-C tuples on MariaDB

as they are negligible for simple queries. The execution effort in `mysql_execute_command()` can be distributed between statistics-related, locking, transaction-processing, data insertion and tuple preparation efforts.

`close_thread_tables()` and `open_and_lock_tables()` are responsible for communicating locking and unlocking commands to the storage engine, as well as starting and committing transactions. Together, the effort in this function accounted for 39 % of the CPU time spent for 10K insert statements.

`prepare_insert()` extracts the tuple structures from a catalog and prepares in-memory field structures. This method accounted for 11.86 % of the query execution time. `fill_record()` stores the values entered by a user or application in field structures, by accessing `store()` functions in `Field` objects. `store()` functions cast data to the buffer type and copy the values into byte arrays. Implementations of fixed size types like integers, doubles, chars etc. store field values in one continuous byte array per row. Variable sized field implementations like blobs or strings are stored separately.

`pack_row()` is called whenever a single byte array of a row is required. It accesses the `pack` methods of `Field` objects, which copy data into a buffer they receive as a parameter. In this way, the `pack_row()` method assembles a single byte array, which can be stored

in a key-value store. Packing the values accounts for 2.2 % of the execution time in this experiment. `write_record()` calls key-value store's method to add a byte array. In this particular experiment, 5.01 % of the execution time was spent on executing this function.

As the query optimizer of MariaDB operates on statistical values, several functions collect statistical data throughout the insertion process. Together, the efforts to compute these statistics added up to 12.14 % of the total effort spent inserting 10000 tuples into a TPC-C customer table.

This analysis of 10K insert operations on a TPC-C customer table showed that the unmarshalling of a tuple from MariaDB's in-memory representation to a key-value store representation only took 2.2 % of the effort to insert 10,000 tuples into MariaDB. Again, a factor that has to be considered is the special design of MariaDB's in-memory tuple representation. As illustrated before, tuples in MariaDB are not stored as typed values. Instead, they reside inside a byte buffer, from where MariaDB extracts typed values on-demand. This benefits the unmarshalling process in that the values are already available in byte arrays. Thus, the unmarshalling process of a tuple is reduced to copying the byte values from multiple byte arrays into a single buffer. The actual effort of filling the byte array with values is spent before in methods to fill the records. Thus, to estimate the effort spent on tuple preparation these efforts must be added. In consequence, the total effort spent on tuple preparation by MariaDB for inserting 10K TPC-C customer tuples increases to 13.72 %.

An implementation factor to be considered here is the fact that no bulk insertion methods were available in the handler interface of the key-value store. Hence, tables needed to be opened and closed repeatedly, which introduced an overhead of more then 30 % of the total effort of query processing. In a production environment, this overhead would be reduced significantly, which could potentially increase tuple preparation's share of query processing effort.

Related, but strictly speaking not part of tuple preparation, are the contents of the insert preparation methods of MariaDB (i.e. `mysql_prepare_insert()`). In these methods MariaDB extracts the structure of a tuple from a catalog and instantiates fields objects dynamically. As these methods do not transform data between data models, this work does not classify this part of query execution as tuple preparation. It is, however, noteworthy that a compiled approach to tuple preparation would have a positive impact here as well. While the contents of the catalog would still need to be checked for potential

schema evolution, in a compiled approach the method would not need to iterate over all fields to instantiate a row. It could simply allocate space for its pre-compiled row type for that table.

An interpretation of the collected CPU profile confirms the hypothesis that tuple preparation can have an impact on query processing. All together, the effort of unmarshalling a tuple accounts for about 11.52 % of the effort to insert 10K customer tuples into a TPC-C database. Thus, improvements to tuple preparation will also impact the overall performance of a DBMS.

It is additionally noteworthy that these experiments concentrated on worst-case scenarios of tuple preparation. They comprise a table scan and insertions in a table with a fairly large degree and multiple variable-sized fields. The overhead of tuple preparation increases with a higher degree of a table, as the for each field virtual function call costs must be payed. Thus, the percentage of total effort spent on tuple preparation could be lower when experimenting with tables with a smaller degree. Similarly, the extensive use of variable-sized fields increases the effort of marshalling, as variable-sized fields are more expensive to allocate. For variable-sized fields additional memory space needs to be allocated. Often spatial locality can not be guaranteed when allocating variable-sized fields, which in turn can introduce a substantial performance overhead. Hence, operations on tables with solely fixed-size fields can exhibit a lower percentage of the total effort spent on tuple preparation.

The executed queries were simple. Thus, the impact of query analysis and optimization is also negligible. In more realistic scenarios, these operations can take a substantial share of the total effort spent on query processing, depleting the share of total effort spent on tuple preparation.

Additionally, the experiments were executed with a single active client. In more realistic scenarios, multiple clients work on the same data set, which introduces an overhead, as clients have to wait for access to data. A locking-related overhead could potentially further decrease the tuple preparation's share of query processing effort.

4.3 SPECIALIZED BENCHMARKS

To analyze the performance improvements of compiled tuple preparation approaches with respect to interpreted approaches, a series of experiments was executed outside of the context of a key-value store.

An experiment of unmarshalling and marshalling a tuple of the TPC-C customer table was implemented. This experiment includes implementations of the (un)marshalling functionality for a `struct` implementation, a dynamic polymorphism implementation and a static polymorphism implementation realized with STL tuples. A fixed amount of tuples was created in each data model, filled with random values and stored in a set. Marshalling and unmarshalling processes chose a random tuple from this set to operate on. Each experiment was carried out 100 times. In each iteration of the experiment, 10G tuples were unmarshalled into byte arrays and then marshalled back to a logical tuple.

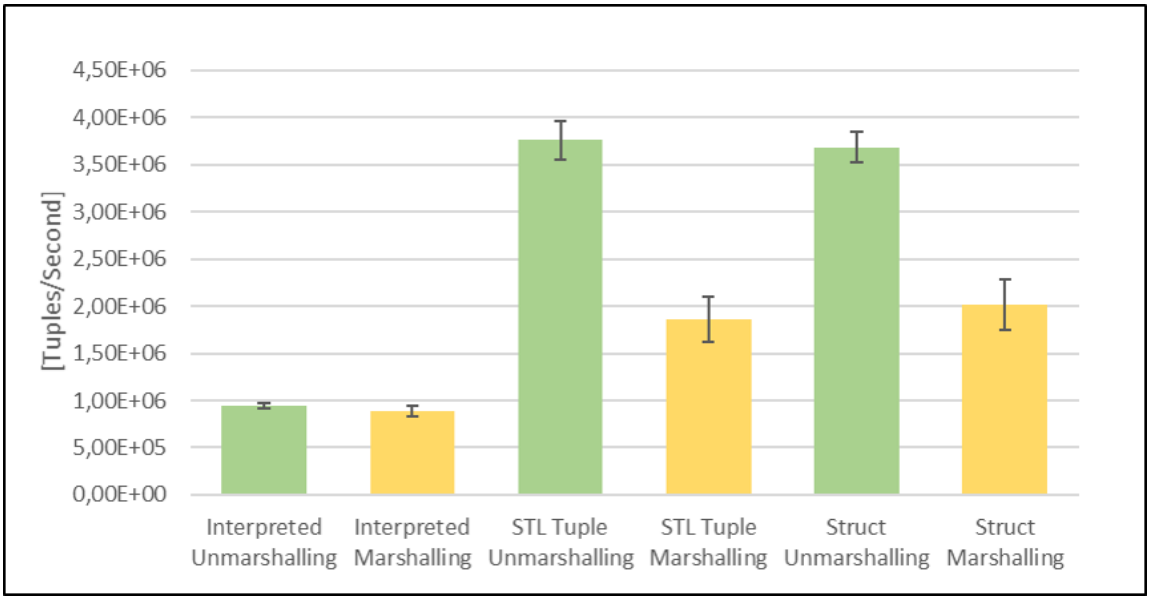


Figure 7: Average throughput (un)marshalling of TPC-C customer tuples. Error bars depict the standard deviation

Figure 7 compares the throughput of unmarshalling and marshalling of TPC-C customer tuples in each approach. The dynamic polymorphism approach was able to achieve an average throughput of 948,509.12 tuples per second for unmarshalling and 893,667.77 tuples per second for marshalling tuples. By comparison, the STL tuple approach could achieve an average throughput of 3,760,065.95 tuples per seconds for unmarshalling and 1,857,358.17 tuples per seconds for marshalling tuples. Thus, the STL tuple approach in-

creased the throughput by factor of 3.96 for unmarshalling tuples by a factor of 2.07 for marshalling tuples compared to the dynamic polymorphism approach. The `struct` approach, could achieve 3,685,723.04 tuples per second unmarshalling throughpu on average and 2,014,599.02 tuples per second marshalling throughput. Hence, STL tuple unmarshalling achieved 92.19 % of the throughput of a `struct` approach respectively 102.02 % marshalling performance.

This exploration on interpretation-based and pre-compiled tuple preparation approaches showed that a significant performance advantage could be gained by the latter. The throughput of marshalling tuples in the interpretation-based approach was doubled by pre-compiled approaches and the throughput of unmarshalling tuples almost quadrupled. The data additionally showed that the STL tuple approach is able to achieve the performance increase of `struct` approaches.

The larger impact on unmarshalling performance by compiled optimizations can be justified with a lack of memory allocations. The effort of marshalling is dominated by allocation efforts, as member variables holding the values of a field need to be allocated anew before they are filled with values from the byte array. As this holds true for each of the three approaches and contains expensive system calls, especially for strings, the compiled approaches were not able to achieve as much of a performance increase as the unmarshalling function. In contrast, unmarshalling functions only allocate memory once in the beginning. They initially extract the size of tuple in its byte representation and adjust the size of the byte array accordingly.

In an effort to investigate these improvements, performance statistics were collected for the generation of 1000 random TPC-C tuples and (un)marshalling of these tuples 10G times. Table 2 illustrates these results. The STL tuple approach and `struct` approach executed the process with almost equally as many instructions, branches and branch misses. In the interpreted approach, the instructions increased by a factor of 1.96 and branches by a factor of 1.81. Branch misses increased by a factor of 2.4 in the interpreted approach. The STL tuple approach introduced 4.85 % more L1 data cache loads than the `struct` approach. In the interpreted approach, this measurement increased by a factor of 2.76. STL tuple and `struct` produced virtually the same amount of L1 Data Prefetches and misses, while the interpreted approach produced 3.95 times more prefetches and 2.68 more prefetch misses.

Table 2: Performance Counter Statistics of (un)marshalling tuples

PERFORMANCE STATISTICS

	Interpreted	STL tuple	`struct`
Instructions	87.732G	44.731G	43.087G
Branches	18.557G	10.209G	10.008G
Branches misses	0.67G	0.27G	0.271G
L1 Data Cache Loads	29.098G	11.072G	10.559G
L1 Data Cache Misses	0.696G	0.399G	0,359G
L1 Data Prefetches	0.557G	0.141G	0.141G
L1 Data Prefetch Misses	0.059G	0.02G	0.022G

In comparison, the interpreted approach to tuple preparation showed a doubling of the number of instructions. As discussed in Section 2.2, the dynamic polymorphism paradigm entails this behaviour. Virtual function calls are indirect function calls. As such when they are executed additional instructions are needed to fetch the function address from memory. Conceptually, further instructions are needed to determine the memory address by loading the vtable. In modern compilers like gcc, the pointer to the vtable is the very first word of an objects binary representation. In this case, the instructions to fetch the vtable pointer are eliminated as the object often already in the CPU's registers. Thus, the cost of virtual function calls becomes equal to the cost of an indirect function call i.e. doubling the number of instructions as can be observed.

The interpreted approach also exhibits double the amount of branches compared to STL tuple or `struct` approaches. As virtual functions can't be inlined, they always create a branch in the code. In a `struct` approach, these branches don't exist as the complete (un)marshalling functionality of a row is contained within a single method. In the STL tuple approaches, the polymorphic functions are inlined by a compiler. Hence, an increase of branches by a factor of 1.81 in the interpreted approach compared to the compiled approaches can be observed. As the use of virtual functions also deteriorates branch predicition capabilities, a slight increase in branch miss rates from 2.64 % respectively 2.7 % in the compiled approaches to 3.61 % is observable.

The increase in L1 data cache loads in the interpreted approach compared to the compiled approaches can be justified by the memory layout of a row in each approach. In the `struct` approaches, primitive data members are layed out in-memory in the order of their declaration. Depending on the STL implementation, primitive data members of an STL tuple are either also layed out in the order of their declaration or in the reverse order. As such both approaches have in common that the data members which implicitly represent fields are allocated closely together in-memory. This improves the locality of reference and reduces the amount of L1 cache loads needed to execute (un)marshalling. In contrast,the in-memory row in the interpreted approach contains an array of pointers to field representations. This structure can cause additional L1 data cache loads, as the fields are not necessarily allocated closely together. The gcc compiler tries to compensate an increase of L1 data cache loads by using prefetch operations. More precisely, L1 prefetch operations in the interpreted approach are increased by a factor of 3.96, a bigger increase than the increase of 2.76 of L1 data cache loads. Also, with a 10.59 % L1 prefetch miss rate, the interpreted approach produces slightly less prefetch misses than the compiled approaches with 14.18 %, respectively 15.06 %. Hence, the architecture of modern CPUs is better utilized by the interpreted approach.

To quantify the impact of compiler optimizations, another experiment was executed with compiler optimizations turned off. To that end, the option `O0` was passed to the gcc compiler which disables the majority of compiler optimizations. Figure 8 illustrates these results. Again, marshalling and unmarshalling was executed 10G times on 1000 random tuples in each in-memory representation of tuples.

The average throughput of unoptimized unmarshalling of the static polymorphism approach was reduced to 119,336.32 tuples per second. The unoptimized marshalling of tuples with static polymorphism achieved an average throughput of 382,512.14 tuples per second. Compared to 179,089.70 tuples per second unmarshalling and 823,144.11 tuples per second marshalling of the `struct` approach, this results in 66.63 % unmarshalling performance and 46.47 % marshalling performance.

This experiment shows that a significant portion of the performance increase of the STL tuple approach is related to compiler optimizations. When inlining is disabled the costs of repeated direct function calls in the STL tuple approach decreases the throughput. Consequently, the approach of using template-based programming no longer can perform as

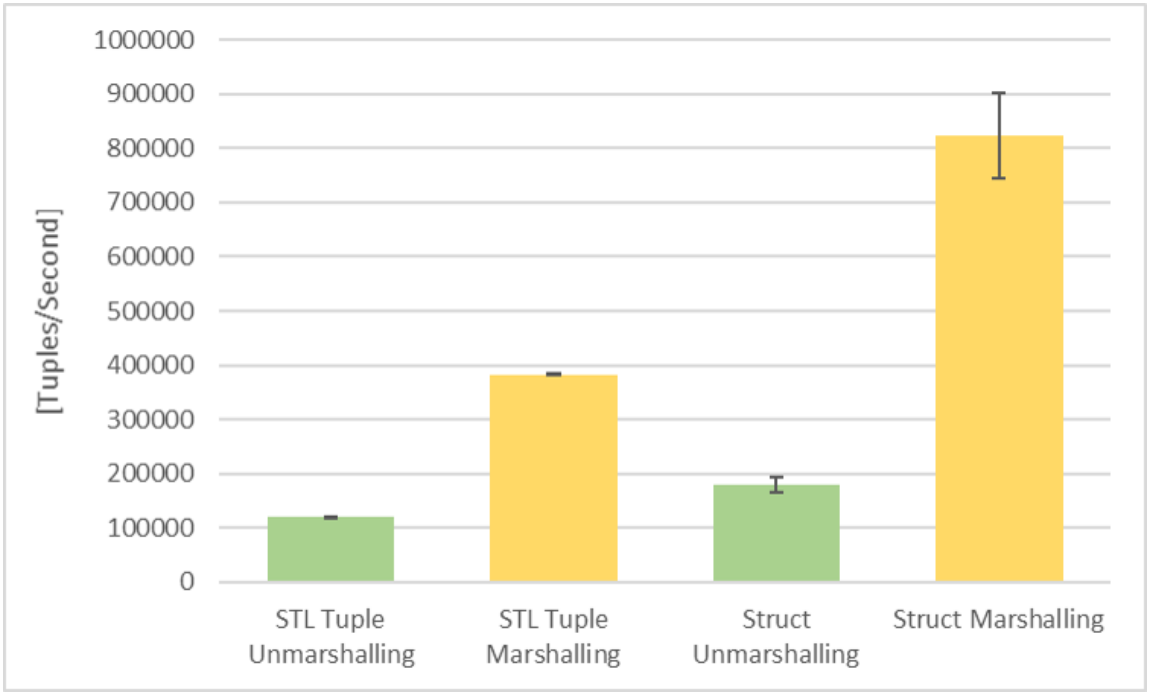


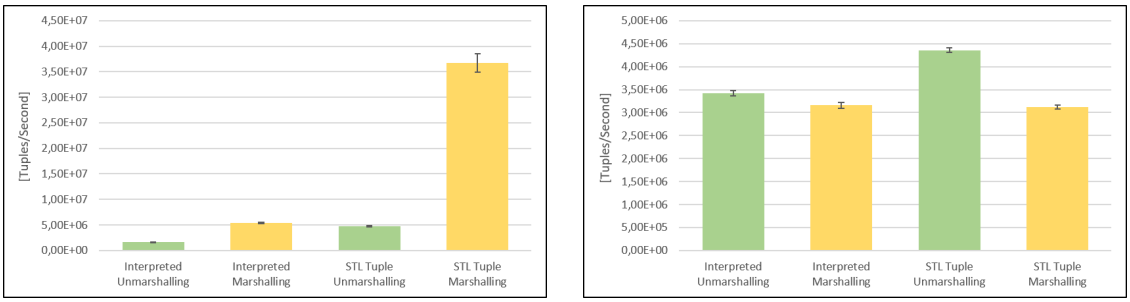
Figure 8: Average throughput of (un)marshalling of TPC-C customer tuples without compiler optimizations. Error bars depict the standard deviation

well as the the strategy of generating structs. Hence, a performance gap between the STL tuple and the `struct` strategy develops.

To quantify the effects of the tuple structure, an experiment was performed that compared the (un)marshalling of tuples with different degrees. To that end, a tuple with 20 `double` fields and a tuple with 4 `string` fields were defined in both, the STL tuple and the interpreted approach. Both tuples were filled with random values, whereby the size of the strings was chosen such that the total size of both tuples was equal (i.e. 40 characters per string). Again, in each of 100 iterations of the experiment, tuples out of a fixed set were randomly chosen and unmarshalled into a byte array and marshalled back into their logical representation 10G times. Figure 9 illustrates the results of this experiment.

In the experiment with tuples with a high degree, illustrated in Figure 9a, the interpreted approach achieved a throughput of 5,390,697.81 tuples per second marshalling data on average and 1,609,980.75 tuples per second unmarshalling data. Compared to that, the STL tuple approach reached an average throughput of 36,746,048.36 tuples per second marshalling tuples, an increase by a factor of 6,81. Unmarshalling tuples with a high degree in the STL tuple approach resulted in an average throughput of 4,755,207.91

tuples per second. Compared to the interpreted approach, the STL tuple approach increases the unmarshalling throughput of the high degree tuples by a factor of 2.95.



(a) High degree tuples

(b) Low degree tuples

Figure 9: Average throughput of interpreted and compiled approach for tuples with different degrees. Error bars depict the standard deviation

An illustration of the experiment with low degree tuples can be observed in Figure 9b. Throughput of marshalling low degree tuples in the interpreted approach was 3,157,570.18 on average, while in the STL tuple approach the throughput was 3,122,207.01. Within a margin of error, neither approach performed better than the other with low degree tuples. Comparing the unmarshalling of records, the interpreted approach reached an average throughput of 3,420,530.18, whereas the STL tuple approach attained 4,360,329.76 tuples per second, resulting in a performance difference of 27,47 % in favor of the STL tuple approach.

These two experiments show that the number and types of fields influence the amount of performance benefit gained by the STL tuple approach. The effects of an increased locality of reference and compiler optimizations can be observed in the experiments with high degree tuples. Similar to the performance boost observed in the experiments with TPC-C customer tuples, the unmarshalling performance of the STL tuple approach increased by a factor of three. With a factor of six, a significantly higher performance boost was observable in STL tuple marshalling. This leads to the assumption that the compiler here was able to apply additional optimization techniques which is facilitated by the structure of the tuple. Each field of the tuple has only primitive data types. In a STL tuple these double values are stored in-memory back-to-back. For double types, no specialized treatment of marshalling values is needed. Thus, a perfect version can marshal a byte array by simply copying it into the STL tuple representation with one copy operation. In the interpreted approach, each field object is separated from the next

field object by at least a pointer to itself and the vtable pointer. In addition to the overhead of virtual function calls, the marshalling of data into the interpreted representation can not be optimized in such a way, entailing a bigger performance gap.

In comparison, the low degree tuples consist of string tuples which are more costly to allocate. As strings are of variable size, field values for this type are not stored together with the row object. Instead, in both approaches, the field value representation contains pointers to values. The costs of copying and allocating then dominate the marshalling process and the STL tuple approach can not gain better performance.

Still, unmarshalling of low degree tuples is slightly faster in the STL tuple approach than the interpreted. Here, the only allocation costs are related to managing the byte array. The decreased locality of reference influences the amount of performance benefits of the STL tuple approach. Yet, the lack of vtable lookups and function invocation leads to a performance increase of 27,47 %.

4.4 STATIC TPC-C IMPLEMENTATION

To test the performance improvements in a more realistic RDBMS enviroment, the TPC-C benchmark was implemented in a static framework for the STL tuple approach and the interpreted approach. The transactions were hand-coded which eliminates the overhead of a query optimizer. Experiments were executed on a TPC-C database of scaling factor 4. Transaction requests were submitted by 4 clients. To reduce the impact of I/O costs, the buffer pool size was set such that the whole database would fit into main memory. Each experiment was carried out 42 times. In each iteration of the experiments, a new database was loaded and transactions were executed for 20 minutes. Figure 10 illustrates the performance results.

The TPC-C benchmark aims to measure how many `new order` transactions can be executed, while the system is executing the four other transaction types. To that end, the performance of a TPC-C execution can be examined with Transactions-per-Minute-C (tpmC).This measurement describes how many committed new order transactions can be executed per minute, while 4 other transaction types are also active. Illustrated in Figure 10 is the performance of interpreted and STL tuple preparation approaches in the TPC-C benchmark in tpmC. In this experiment, the measurements showed an increase of successful new order transaction in the STL tuple approach by 8.87 %.

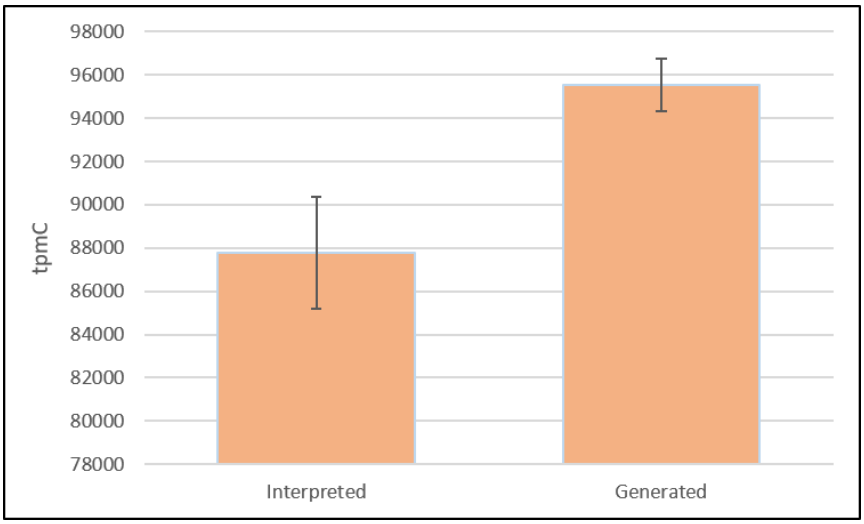


Figure 10: Average TPC-C throughput in tpmC. Error bars depict the standard deviation

Observable performance gains of the STL tuple approach in a more realistic environment were limited to below 10 %. The TPC-C benchmark is a benchmark with relatively short transactions. Hence, locking and transaction processing efforts have a substantial share of the query processing efforts. Consequently, the impact of an improved tuple preparation is small. Nevertheless, it was shown that in OLTP scenarios pre-compiled tuple preparation does have an positive impact on query processing in a RDBMS.

5

CONCLUSION

In order to remove interpretational overhead and allow compiler optimizations, pre-compiled approaches to tuple preparation in a DBMS are desirable. In this work, the impact tuple preparation techniques were evaluated and improvements were investigated. In particular, this work compares interpretation-based approaches to pre-compiled approaches of tuple preparation in an RDBMS.

The predominant method of tuple preparation in general-purpose DBMSs is based on an interpretation-at-runtime strategy. This concept introcudes a level of indirection in the representation of tuples. Runtime flexibility is gained at the cost of performance. The use of virtual function calls increases the amount of instructions necessary to execute tuple preparation. It generates a number of unavoidable branches and hinders modern CPUs in their prediction capability. Additionally, polymorphic structures can exhibit non-optimal memory layouts.

Pre-compiled approaches attempt to remove the costs of interpretation by resolving it at compile time. While this approach decreases the flexibility of an in-memory tuple representation, it also increases tuple preparation performance. The level of indirection in the representation of fields is removed at compile time and consequently, further micro optimization can be applied.

Qualitative explorations in a general-purpose DBMS led to the conclusion that tuple preparation can be a substantial factor in query processing. Especially in certain scenarios, it imposes a computational overhead on query processing, consequently slowing down DBMS performance. However, it can also be concluded that the impact is dependent on a number of factors. Among others, query analysis and locking efforts impact the percentage of query processing spent on tuple preparation. Only in certain configurations are improvements of tuple preparation relevant for the performance of a DBMS. Hence, a cost model for tuple preparation is desirable. A cost model should incorporate the distribution of CPU time during query processing. For instance, further analyses could collect more precise statistics about the amount of CPU time spent in query analysis, locking,

value retrieval and tuple preparation modules. Based on these statistics, query profiles could be defined that could benefit from pre-compiled preparation approaches.

A series of specialized benchmarks showed that the efficiency of tuple preparation can be improved by pre-compiled approaches. Both, marshalling and unmarshalling of tuples, improved significantly in two distinct pre-compiled approaches compared to an interpreted approach. The collected performance statistics reinforced the structural differences between approaches as they showed the sources of overhead. The experiment also demonstrated that a STL tuple approach using template-based programming can perform as fast as a `struct` variant. Although both variations can be generated, the better integration into the programming language makes template-based programming more preferable. The illustrated implementation further has the advantage of seamlessly integrating with the STL.

Additional experiments without compiler optimization showed the dependence of the performance increase on them. Without compiler optimizations the performance gap between the STL tuple approach and the `struct` approach increased substantially in favor of the latter. Hence, optimizations play an essential role in the performance gains of template-based programming. Disabling optimizations also accelerates the compilation process. An in-depth analysis of the correlation between compile time, performance and compiler optimizations could provide a cost model for different configurations. Depending on the frequency of schema evolutions, different levels of optimizations could be chosen.

In a further comparison, it was observable that performance of interpreted and generated tuple preparation is dependent on the structure of a tuple. In experiments with primitive data types and tuples of a high degree, a larger performance boost was noticeable in generated approaches. In contrast, when allocation and copy costs of variable-sized fields dominated the cost of marshalling, no performance advantage was observable. The conclusion here is that the degree of a tuple and the types of the fields directly influences the possible performance increase by pre-compiled approaches. Generally speaking, tuples with a high degree are more likely to benefit from pre-compiled tuple preparation. However, the field types of a tuple are equally important. An in-depth analysis of the correlation between tuple structure and performance boost is required. Future cost models should be extended with a measurement incorporating the structure of a tuple.

Experiments with the TPC-C benchmark showed more realistic measurements. A slight increase in performance was noticeable in the implementation utilizing generated tuple preparation. As TPC-C is a benchmark with relatively short transaction, this behaviour was expected. Under such circumstances, the efforts of locking, value extraction and transaction processing dominate the workload of a DBMS, so that the impact of improved tuple preparation is relatively small.

Holistically, it is possible to conclude that pre-compiled tuple preparation has limited benefits in OLTP applications. Although, a performance increase is certainly noticeable, a bigger throughput boost could be seen in OLAP systems. Potentially, the query profile of OLAP systems could facilitate bigger performance gaps, as these types of systems are more focused on the extraction and evaluation of data. Further investigations with OLAP benchmarks are needed to substantiate this claim.

In conclusion, pre-compiled approaches can substantially improve transformation of data between models. They benefit from a resolution at compile time and thereby decrease runtime overhead. They profit from compiler optimizations and thus will benefit from future compiler improvements, at no charge other than potentially increased compile time.

As tables in an RDBMS can be subject to schema evolutions, recompilations of the generated approach can be necessary. However, the costs of recompiling tuple preparation methods in an RDBMS can be amortized between schema evolutions. Still, it is noteworthy that in certain scenarios the performance improvements can be negligible. Depending on the frequency of schema evolutions and including compilation costs, pre-compiled approaches could result in a net loss of performance.

In the context of DBMSs, certain query profiles can benefit from pre-compiled approaches. As long as data is readily available in-memory and other query processing modules do not overshadow tuple preparation costs, pre-compiled approaches can increase the performance of DBMSs. It is additionally noteworthy that interpreted and generated approaches are not mutually exclusive. There are systems conceivable that utilize a mix of both approaches to realize tuple preparation. These systems could be categorized as a hybrid between OLAP/OLTP. However, the validity of such hybrid systems depends on the development of a tuple preparation cost model.

A

APPENDIX

Listing 12: Dynamic Field Implementation Alternative

```
struct field_value_t
{
    field_desc_t* _pfield_desc;
    bool          _null_flag;
    union s_field_value_t {
      bool         _bit;
      short        _smallint;
      char         _char;
      int          _int;
      ...
      char*        _string;
    }   _value;
    char* _data;

    inline void field_value_t::set_value(const void* data, const uint length)
    {
        switch (field_desc->type()) {
          case SQL_BIT:
          case SQL_SMALLINT:
          case SQL_CHAR:
          case SQL_INT:
          case SQL_FLOAT:
          case SQL_LONG:
                    memcpy(&_value, data, _max_size);
              break;
          case SQL_TIME:
                    memcpy(_value._time, data, MIN(length, _real_size));
              break;
          case SQL_VARCHAR:
                    set_var_string_value((const char*)data, length);
              break;
          case SQL_FIXCHAR:
          case SQL_NUMERIC:
          case SQL_SNUMERIC:
                    memcpy(_value._string, data, _real_size);
              break;
          }
    }

    int           get_int_value() const;
    short         get_smallint_value() const;
    bool          get_bit_value() const;
    char          get_char_value() const;
    ...
    void          get_string_value(char* string, const uint bufsize) const;
    decimal       get_decimal_value() const;

}
```

BIBLIOGRAPHY

[1] Mike W. Blasgen, Morton M. Astrahan, Donald D. Chamberlin, JN Gray, WF King, Bruce G. Lindsay, Raymond A. Lorie, James W. Mehl, Thomas G. Price, Gianfranco R. Putzolu, et al. "System R: An architectural overview." In: *IBM systems journal* 20.1 (1981), pp. 41–62.

[2] Edgar F Codd. "A relational model of data for large shared data banks." In: *Communications of the ACM* 13.6 (1970), pp. 377–387.

[3] Transaction Processing Council. *TPC-C Benchmark Specification*. 2016. URL: http://www.tpc.org/tpcc/default.asp (visited on 11/25/2016).

[4] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. "SAP HANA database: data management for modern business applications." In: *ACM Sigmod Record* 40.4 (2012), pp. 45–51.

[5] MariaDB Foundation. *MariaDB*. URL: https://mariadb.org/ (visited on 11/17/2016).

[6] Hector Garcia-Molina and Kenneth Salem. "Main memory database systems: An overview." In: *IEEE Transactions on knowledge and data engineering* 4.6 (1992), pp. 509–516.

[7] Goetz Graefe. "Query evaluation techniques for large databases." In: *ACM Computing Surveys (CSUR)* 25.2 (1993), pp. 73–169.

[8] Goetz Graefe, Haris Volos, Hideaki Kimura, Harumi Kuno, Joseph Tucek, Mark Lillibridge, and Alistair Veitch. "In-memory performance for big data." In: *Proceedings of the VLDB Endowment* 8.1 (2014), pp. 37–48.

[9] Theo Härder. "DBMS architecture-the layer model and its evolution." In: *Datenbank-Spektrum* 13 (2005), pp. 45–57.

[10] Theo Härder and Erhard Rahm. *Datenbanksysteme: Konzepte und Techniken der Implementierung*. Springer-Verlag, 1999.

[11] Donald E. Knuth. "Computer Programming as an Art." In: *Communications of the ACM* 17.12 (1974), pp. 667–673.

[12] Konstantinos Krikellas, Stratis D Viglas, and Marcelo Cintra. "Generating code for holistic query evaluation." In: *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. IEEE. 2010, pp. 613–624.

[13] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation." In: *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE. 2004, pp. 75–86.

[14] Stefan Manegold, Martin L Kersten, and Peter Boncz. "Database architecture evolution: mammals flourished long before dinosaurs became extinct." In: *Proceedings of the VLDB Endowment* 2.2 (2009), pp. 1648–1653.

[15] John McCallum. *Memory Prices 1957 to 2016*. 2016. URL: http://www.jcmit.com/memoryprice.htm (visited on 10/03/2016).

[16] Thomas Neumann. "Efficiently compiling efficient query plans for modern hardware." In: *Proceedings of the VLDB Endowment* 4.9 (2011), pp. 539–550.

[17] Jun Rao, Hamid Pirahesh, C Mohan, and Guy Lohman. "Compiled query execution engine using JVM." In: *22nd International Conference on Data Engineering (ICDE'06)*. IEEE. 2006, pp. 23–23.

[18] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. "Vectorization vs. compilation in query execution." In: *Proceedings of the Seventh International Workshop on Data Management on New Hardware*. ACM. 2011, pp. 33–40.

[19] Michael Stonebraker and Ariel Weisberg. "The VoltDB Main Memory DBMS." In: *IEEE Data Eng. Bull.* 36.2 (2013), pp. 21–27.

[20] Scott Wasson. *Intel's Xeon 5600 processors - Westmere-EP adds two more cores to an already-potent mix*. 2010. URL: http://techreport.com/review/19196/intel-xeon-5600-processors/4 (visited on 11/07/2016).

[21] Paul R. Wilson. "Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Spaces on Standard Hardware." In: *SIGARCH Comput. Archit. News* 19.4 (July 1991), pp. 6–13. ISSN: 0163-5964. DOI: 10.1145/122576.122577. URL: http://doi.acm.org/10.1145/122576.122577.

[22] Marcin Zukowski, Peter A Boncz, Niels Nes, and Sándor Héman. "MonetDB/X100-A DBMS In The CPU Cache." In: *IEEE Data Eng. Bull.* 28.2 (2005), pp. 17–22.

DECLARATION OF AUTHORSHIP

I, Stefan Hemmer, declare that this thesis titled, "Key-Value Storage Engines in Relational Databases" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Kaiserslautern, December 2016

Stefan Hemmer