CPU and Cache Efficient Management of Memory-Resident Databases

Holger Pirk [#], Florian Funke [*], Martin Grund [%], Thomas Neumann [*],
Ulf Leser [+], Stefan Manegold [#], Alfons Kemper [*], Martin Kersten [#]

[#] *CWI*
Amsterdam, The Netherlands
{first.last}@cwi.nl

[*] *TU München*
München, Germany
{first.last}@in.tum.de

[%] *HPI*
Potsdam, Germany
{first.last}@hpi.uni-potsdam.de

[+] *Humboldt Universität zu Berlin*
Berlin, Germany
{last}@informatik.hu-berlin.de

***Abstract*— Memory-Resident Database Management Systems (MRDBMS) have to be optimized for two resources: CPU cycles and memory bandwidth. To optimize for bandwidth in mixed OLTP/OLAP scenarios, the hybrid or Partially Decomposed Storage Model (PDSM) has been proposed. However, in current implementations, bandwidth savings achieved by partial decomposition come at increased CPU costs. To achieve the aspired bandwidth savings without sacrificing CPU efficiency, we combine partially decomposed storage with *Just-in-Time (JiT) compilation* of queries, thus eliminating CPU inefficient function calls. Since existing cost based optimization components are not designed for JiT-compiled query execution, we also develop a novel approach to cost modeling and subsequent storage layout optimization. Our evaluation shows that the JiT-based processor maintains the bandwidth savings of previously presented hybrid query processors but outperforms them by two orders of magnitude due to increased CPU efficiency.**

I. Introduction

Increasing capacity at decreasing costs of RAM make Memory-Resident Database Management Systems (MRDBMSs) an interesting alternative to disk-based solutions [30]. The superior latency and bandwidth of RAM can boost many database applications such as Online Analytical Processing (OLAP) and Online Transaction Processing (OLTP). Unfortunately, the *Volcano*-style processing model [14] that forms the basis of most disk-based DBMSs was not designed to support such fast storage devices. To process arbitrarily wide tuples with generic operators, Volcano-style query processors "configure" the operators using function pointers that are "chased" at execution time. This pointer chasing is highly CPU inefficient [2], [6], but acceptable for disk-based systems because disk I/O costs hide the costs for function calls. Due to its CPU inefficiency, a direct port of the Volcano model to a faster storage device often fails to yield the expected performance gain. MRDBMSs performance has a second critical dimension next to I/O (cache) efficiency: CPU efficiency (see Figure 1).

To improve the CPU efficiency of MRDBMSs, the *Bulk Processing* model has been proposed [23]. Using this model, data is processed column-at-a-time which is CPU efficient but only cache efficient on data that is stored according to the *Decomposed Storage Model (DSM)* [12]. The DSM is known for its good OLAP performance but suffers from bad OLTP performance due to poor cache efficiency when (partially) reconstructing tuples. To achieve good performance for mixed (OLTP & OLAP) workloads, the hybrid or, more accurately, the *Partially Decomposed Storage Model (PDSM)* has been proposed [15]. Even though they bulk-process data partition-at-a-time, current implementations have to handle arbitrarily wide tuples (within a partition). Just like Volcano, they do this using function pointers [15]. Therefore, the current PDSM processors yield better cache efficiency than the DSM but lower CPU efficiency (Figure 1).

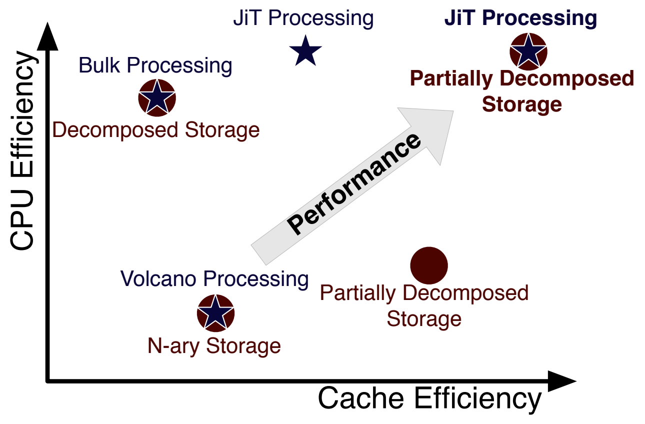


Fig. 1: Dimensions of Memory-Resident DBMS Performance

To resolve the conflict between CPU and cache efficient query processing we propose to remove the need for function pointers by combining *Just-in-Time (JiT)* compilation of queries with the PDSM. Specifically, we make the following contributions:

- We present the design and implementation of a PDSM-based storage component for HyPer, our JiT-based Database Management System (DBMS).

- We introduce a novel approach to query cost estimation and subsequent storage layout optimization for JiT-compiled queries: treating a generic cost model like a "programmable" machine that yields holistic query cost estimations using an appropriate instruction set
- We conduct an extensive evaluation of our system using existing benchmarks and compare our results to those of previously published systems [15]

The remainder of this paper is organized as follows: In Section II we present related work on CPU and cache efficient processing of memory resident data. In Section III we asses the impact of this conflict and describe how we resolve it using JiT-compiled query execution in *HyPer* [21]. In Section IV we describe the cost model and illustrate its usage for layout optimization in Section V. In Section VI we present our evaluation and conclude in Section VII.

II. Related Work

Before illustrating our approach to CPU and cache efficient memory-resident data management, we discuss previous approaches, the encountered problems and inherent tradeoffs.

A. CPU Efficient Processing

In Volcano, relational query plans are constructed from flexible operators. When constructing the physical query plan, the operators are "configured" and connected by injecting function pointers (selection predicates, aggregation functions, etc.). Although variants of this model exist, they face the same fundamental problem: operators that can change their behavior at runtime are, from a CPU's point of view, unpredictable. This is a problem, because many of the performance optimizations of modern CPUs and compilers rely on predictable behavior [19]. Unpredictable behavior circumvents these optimizations and causes hazards like pipeline flushing, poor instruction cache locality and limited instruction level parallelism [2]. Therefore, flexible operators, as needed in Volcano-style processing, are usually CPU inefficient.

To increase the CPU efficiency of MRDBMS, the database research community has proposed a number of techniques [5], [35], [20], [17]. The most prominent ones are *bulk processing* and *a-priory query compilation* . The former is geared towards OLAP applications, the later towards OLTP. Both have shortcomings for mixed workloads that we discuss in the following.

Bulk processing focuses on analytical applications and was pioneered by the MonetDB project [23], [5]. Like in Volcano, complex queries are decomposed into precompiled primitives. However, Bulk processing primitives are static loops without function calls that materialize all intermediate results [5]. For analytical applications, the resulting materialization costs are outweighed by the savings in CPU efficiency. Efforts to reduce the materialization costs have led to the vectorized query processing model [35] which constrains materialization to the CPU cache. Due to the inflexibility of the primitives, however, bulk processing is only efficient on fully decomposed relations, which are known to yield poor cache locality for OLTP applications. Using tuple clustering, compression and bank packing [20], it is possible to efficiently evaluate selections on multiple attributes in a bulk-manner. However the necessary compression hurts update performance and decompression adds to tuple reconstruction costs.

(A-priory) query compilation is advocated by, e.g., the VoltDB [17] system as a means to support high performance transaction processing on any storage model. It achieves CPU efficiency, i.e., avoids function calls, by statically compiling queries to machine code and inlining functions. The processing model supports SQL for query formulation but needs a reassembly and restart of the system whenever a query is changed or added. It also complicates the optimization of complex queries, because all plans have to be generated without knowledge of the data or parameters of the query. Both of these factors make it unsuited for OLAP applications that involve complex or ad-hoc queries.

B. Cache Efficient Storage

Having eliminated the function call overhead through one of these techniques, memory bandwidth is the next bottleneck [5]. One way to reduce bandwidth consumption is compression in its various forms [33] (dictionary compression, run-length encoding, etc.). However, this is beyond the focus of this paper and orthogonal to the techniques presented here. In this paper, we study (partial) decomposition of database tables to reduce bandwidth waste through suboptimal co-location of relation attributes. Full decomposition, i.e., storage using the DSM [12] yields significant bandwidth savings for analytical applications on disk-resident databases. Unfortunately, full decomposition has a negative impact on intra-tuple cache locality and therefore transaction processing performance [1]. Consequently, it is suboptimal for applications that generate mixed OLTP/OLAP workloads. To improve cache efficiency for such applications, the *hybrid* or, more accurately, the *Partially Decomposed Storage Model (PDSM)* has been proposed [15]. In this model, database schemas are decomposed into partitions such that a given workload is supported optimally. Applications that can benefit from this technique include mixed OLTP/OLAP applications like real-time reporting on transactional data, non-indexed search, or the management of schemas with tables in which a tuple may describe one of multiple real-life objects. Such schemas may result from, e.g., mapping complex class hierarchies to relations using Object Relational Mapping (ORM).

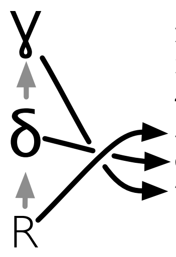
III. CPU and Cache Efficient Data Management

By definition, partially decomposed data involves at least one N-ary partition (otherwise we would refer to it as fully decomposed). As already described, MRDBMS need flexible operators to process arbitrarily wide tuples in a single scan. To achieve such flexibility, current query processors [15] rely on function pointers which causes CPU inefficiency. In this section we demonstrate the impact of this problem using a practical example. Following that, we describe our approach that employs JiT compilation to overcome this problem.

(a) SQL

```
Query:   select sum(B), sum(C), sum(D), sum(E) from R where A = $1
Schema:  create table R(A int, C int, ..., P int)
```

(b) Relational Algebra



(c) C-Implementation on Partially Decomposed Relation

```
1  void query(const struct{int A[SIZE]; v4si B_to_E[SIZE];
2                          int F_to_P[];}* R,
3             v4si* sums, const int c){
4
5    for (int t_id = 0; t_id < SIZE; ++t_id)
6      if(R->A[t_id] == c)
7        *sums += R->B_to_E[t_id];
8  }
```

Fig. 2: Representations of the example query

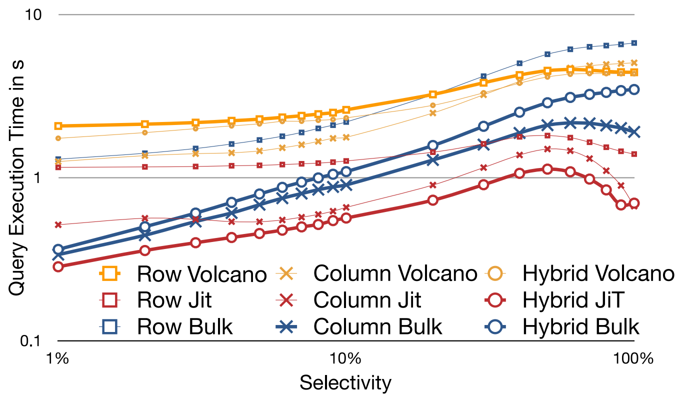


Fig. 3: Costs of the Example Query on 25 M tuples (1.6GB)

A. The Impact of Storage and Processing Model

To evaluate the impact of storage and processing model, we implemented a typical select-and-aggregate query (Figure 2a) in C. We implemented the query to the best of our abilities according to the different processing and storage strategies[1] and measured the query evaluation time while varying the selectivity of the selection predicate (Figure 3).

In our *bulk processing* implementation, the first operator scans column `A` and materializes all matching positions. After that, each of the columns `B` to `E` are scanned and all the matching positions materialized. Finally, each of the materialized buffers are aggregated. This is CPU efficient but cache inefficient for high selectivities. The *Volcano-style* implementation consists or three functions (scan, selection, aggregation), each calling the underlying repetitively to produce the next input tuple. The resulting performance (independent of the storage model) of the Volcano model indicates that it is, indeed, inappropriate for such a query on memory-resident data. The *JiT-compiled* query was implemented according to the HyPer compilation model [27] and is displayed in Figure 2c in the version that is executed on the PDSM data.

[1]The partially decomposed representation was hand-optimized

The selectivity-dependent advantage of bulk- and JiT-compiled processing is consistent with recent work [32]. The figure also shows that, across all selectivities, our implementation of the JiT-compiled query on partially decomposed data outperforms the other approaches. This observation led us to the following claim:

JiT compilation of queries is an essential technique to support efficient query processing on memory-resident databases in N-ary or partially decomposed representation.

In the remainder we study how to combine the advantages of PDSM and JiT-compiled queries in a relational Database Management System (DBMS).

B. Partially Decomposed Storage in HyPer

HyPer is our research prototype of a relational MRDBMS. To compete with the bulk processing model in terms of CPU efficiency, HyPer relies on JiT-compilation of queries [27]. Whilst DBMS compilers have a long history [3], [8], up to recently [27], [24], [31], the focus has been flexibility and extensibility rather than performance. The idea is to generate code that is directly executable on the host system's CPU. This promises highly optimized code and high CPU efficiency due to the elimination of function call overhead.

The code generation process is described in previous work [27] and out of scope of this paper. To give an impression of the generated code, however, Figure 2 illustrates the translation of the relational algebra plan of the example query (Figure 2a) to C99-code. The program evaluates the given query on a partially decomposed relation `R`. The relation `R`, the output buffer `sums` and the selection criteria `c` are parameters of the function. In this example, every operator corresponds to a single line of code (the four aggregations are performed in one line using the vector intrinsic type `v4si`). The scan yields a loop to enumerate all tuple ids (Line 5). The selection is evaluated by accessing the appropriate value from the first partition in an `if` statement (Line 6). If the condition holds, the values of the aggregated attributes are added to the global sum (Line 7). It is apparent that no overhead in storage or executed code is generated. All operators are merged into a single for-loop. Values enter the CPU registers once and do not leave them until they are no longer needed. In practice, the compiler does not generate C-code but equivalent LLVM-assembler which is compiled into machine code by the LLVM-compiler library [25].

As demonstrated in previous work [27], the generated code achieves the CPU efficiency of the bulk processing model without the need for expensive intermediate materialization. More importantly for our case, however, JiT-compilation makes the N-ary storage model viable for memory-resident databases. Since the generated code is static, it is very predictable and allows the respective optimizations by the CPU and the compiler. Since HyPer already has an N-ary storage backend, developing a backend for PDSM is straightforward. We extended the catalog to support multiple vertical partitions within a single relation and the compiler to generate accesses to the respective partitions rather than the relation. As with earlier systems, the main challenge is to determine the appropriate decomposition for a given schema and workload. We will discuss our approach to this problem in the next sections.

IV. Query Cost Estimation

In addition to the processing model, the query performance on partially decomposed data also depends on the choice of the decomposition/layout. Like earlier approaches [15], we focus on cost-based optimization to find an appropriate layout for a given workload. Since in memory-resident data processing no cost factor clearly dominates, a hardware-conscious cost model is needed. Since memory-resident bulk processors face similar challenges there is already a body of research in hardware-conscious cost modeling for main memory databases [26], [15]. For JiT-compiled queries, however, query evaluation is more complicated: as described in Section III-B query operators are interleaved, resulting in complex and irregular memory access. Simply adding the costs of the operators [26] or neglecting non-scan operators [15] would yield inaccurate estimates. To achieve more accurate estimates, we developed a "programmable" holistic cost model based on the existing Generic Cost Model [26], using its atoms as instructions.

In the rest of this section we briefly motivate the need for a complex model, introduce the Generic Cost Model [26] as well as our extensions and the use for hardware and storage-layout aware cost estimation of queries.

A. Cost Factors on modern CPUs

To achieve high memory access performance, modern CPUs incorporate a complex memory hierarchy (see Figure 4). Multiple levels of caches and TLBs speed up repetitive accesses to data items (or data items located on the same cache line or TLB-Block). In addition, many CPUs have prefetching units that speculatively load data items before they are accessed.

1) Prefetching: While transferring and processing a fetched cache line in the CPU, the next accessed cache line is anticipated by a *Prefetching Unit*. If the confidence is high enough, a fetch instruction is issued to the memory system and the cache line loaded into a slot of the *Last Level Cache (LLC)*. A correctly prefetched cache line may hide memory access latency behind the time spent processing the data whilst incorrect prefetching a) causes unnecessary traffic on the memory bus and b) may evict a cache line that should have stayed cache-resident. Due to these potentially harmful effects, prefetching units generally follow a cautious strategy

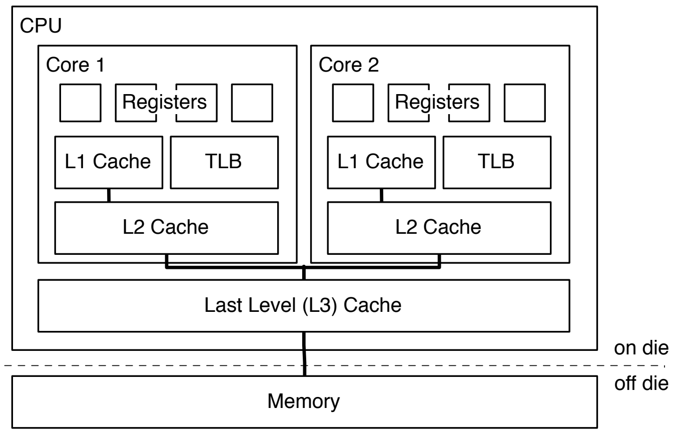


Fig. 4: Memory Structure of an Intel Nehalem System

Notation	**Description** [26]
	Parameters
`R.n`	The number of tuples, values or tuple fragments stored in a relation/partition `R`
`R.w`	The data width of tuple, value or tuple fragment
`u`	The number of data words of a data item that are accessed when performing an access pattern (`u` $<$ `R.w`)
B_i	The access granularity (block size) of cache i
l_i	The block access latency of cache i
	Atoms
`s_trav (R.n,R.w)`	The sequential traversal of a memory region of width `R.w` with unconditional access to every single of the `R.n` data items
`r_trav (R.n,R.w)`	The traversal of a memory region of width `R.w` with unconditional access to every single of the `R.n` data items in random order
`rr_acc (R.n,R.w,r)`	The repetitive (`r` times) access of one of the `R.n` data items of a memory region of width `R.w`
	Algebraic Operators
$P_1 \odot P_2 \odot \ldots \odot P_n$	The concurrent execution of the Access Patterns P_1 to P_n
$P_1 \oplus P_2 \oplus \ldots \oplus P_n$	The sequential execution of the Access Patterns P_1 to P_n
	Intermediate Metrics
M_i^s	The number of sequential access cache misses induced on cache i
M_i^r	The number of random access cache misses induced on cache i

(a) Atoms and Operators

access pattern of the example query
s_trav(26214400,4) $\odot$ rr_acc(26214400,16,262144) $\odot$ rr_acc(1,16,262144)

(b)

TABLE I: Overview of the Generic cost model

when issuing prefetch instructions.

Prefetching strategies: Prefetching strategies vary among CPUs and are often complex and defensive up to not issuing any prefetch instructions at all. In our model, we assume an *Adjacent Cache Line Prefetching with Stride Detection* strategy that is, e.g., implemented in the Intel Core Microarchitecture [18]. Using this strategy, a cache line is prefetched whenever the prefetcher anticipates a constant stride. Although this seems a naïve strategy, its simplicity and determinism make it attractive for implementation as well as modeling. More complex strategies exist, but usually rely on the (partial) data access history of the executed program. These are generally geared towards more complex operations (e.g., high dimensional data processing or interleaved access patterns) yet behave similar to the *Adjacent Cache Line* Prefetcher in simpler cases like relational query processing.

B. The Generic Cost Model

The Generic Cost Model is built around the concept of *Memory Access Patterns*, formal yet abstract descriptions of the memory access behavior that an algorithm exposes. The model provides atomic access patterns, an algebra to construct complex patterns and equations to estimate induced costs. Although the model is too complex for an in-depth discussion here, Table Ia provides a brief description. We refer the interested reader to the original work [26] for a detailed description.

To illustrate the model's power, consider the example in Table Ib which is the access pattern of the example query (see Figure 2a) on partially decomposed data for a selectivity of 1%. To evaluate the given query, the DBMS scans column `A` by performing a sequential traversal of the memory region that holds the integer-values of a (`s_trav(A) = s_trav(26214400,4)`[2]). Concurrently ($\odot$), whenever the condition holds, the columns `B, C, D, E` are accessed. This is modeled as a `rr_acc` on the region with `r` reflecting the number of accessed values (`r` can be derived from the selectivity). For every matching tuple, the output region has to be updated which yields the last atom: a `rr_acc` of a region which holds only one tuple but is accessed for every matching tuple. This algebraic description of the executed program has proven useful for the prediction of main memory join performance [26].

One may notice, however, that the access pattern in Table Ib is not an entirely accurate description of the actual operation: the `rr_acc` for the access of `B, C, D, E` is not fully random since the tuple can be assumed to be accessed in order. In the next section, we illustrate our extensions to the model that allow modeling of such behavior.

C. Extensions to the Generic Cost Model

When modeling JiT-compiled queries on a modern CPU we encountered several shortcomings of the original model. The

[2]Depending on the context we will use either numeric parameters or relation identifiers when denoting atomic access patterns. While using relation identifiers is not strictly speaking correct, the numeric parameters are generally easily inferred from the relation identifiers.

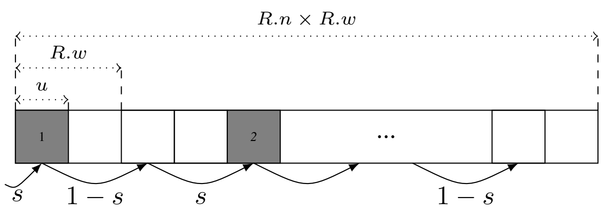


Fig. 5: `s_trav_cr`

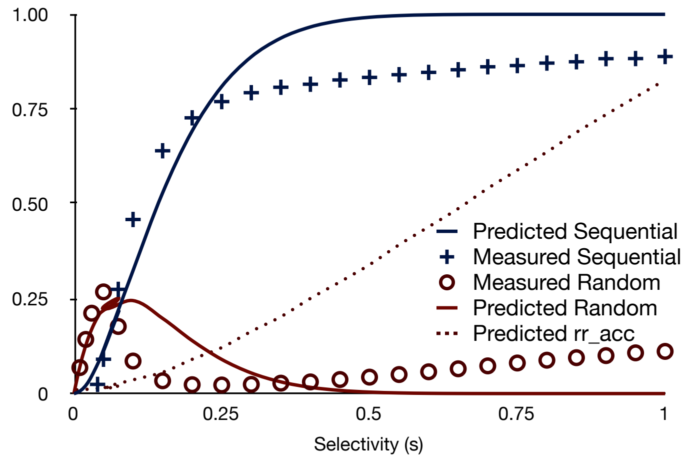


Fig. 6: Prediction Accuracy of `s_trav_cr` vs. `rr_acc`

first has been hinted at in the last section: the inadequacy of the model for selective projections. We overcome this problem by introducing a new access pattern, the *Sequential Traversal/Conditional Read*. We also report two modifications we applied to the model to improve the accuracy of random access estimation and the impact of prefetching. While an understanding of the new access pattern is crucial for the rest of this paper, the other two are extensions that do not change the nature of the model.

1) Sequential Traversal/Conditional Read: The example in Table Ib already indicated a problem of the cost model: If a memory region is scanned sequentially but not all tuples are accessed, most DBMSs expose an access pattern that cannot be accurately modeled using the atoms in Table Ia. In the example, we resorted to modeling this operation using a `rr_acc` which is not appropriate because a) the region is traversed from begin to end without ever going backwards and b) no element in the region is accessed more than once. While this is not an issue for the original purpose of the cost model, i.e., join optimization, it severely limits its capabilities for the holistic estimation of query costs and subsequent optimization of the storage layout.

We developed an extension to accurately model this behavior. A new atom, the *Sequential Traversal with Conditional Reads* (`s_trav_cr`), captures the behavior of selective projections using the same parameters as `s_trav` yet also incorporates the selectivity of the applied conditions s. Figure 5 gives a visual impression of this Access Pattern: The Region R is traversed sequentially in $R.n$ steps. In every step, u bytes are read with probability s and the iterator unconditionally advances $R.w$ bytes. This extension provides the atomic access pattern that is needed to accurately model the query evaluation of the example in Table Ib: The `rr_acc([B, C, D, E])` becomes a `s_trav_cr([B, C, D, E], s)` with the selectivity $s = 0.01$.

To estimate the number of cache misses from these parameters we have to estimate the probability P_i of accessing a cache line when traversing it. P_i is equal to the probability that any of the data items of the cache line is accessed. It is independent of the capacity of the cache but depends on the width of a cache line (i.e., the block size and thus denoted with B_i). Assuming a uniform distribution of the values over the region, P_i can be estimated using Equation 1. For non-uniformly distributed data the model can be extended with a different formula.

$$P_i = 1 - (1 - s)^{B_i} \tag{1}$$

However, to estimate the induced costs, we have to distinguish random and sequential misses. Even though not explicitly stated, we believe that the distinction between random and sequential cache misses in the original model [26] was introduced largely to capture non-prefetched and prefetched misses. Thus, we model them as such. Assuming an *Adjacent Cache Line Prefetcher*, the probability of a cache line to be a sequential, i.e., prefetched, cache miss is the probability of a cache line being accessed with the preceding cache line being accessed as well. Since these two events are statistically independent, the probability of the two events can simply be multiplied which yields Equation 2 for the probability of a cache miss being a sequential miss (P_i^s).

$$P_i^s = \left(1 - (1 - s)^{B_i}\right)^2 \tag{2}$$

Since any cache miss that is not a sequential miss is a random miss, the probability for a cache line to be a random miss P_i^r can be calculated using Equation 3.

$$\begin{aligned} P_i^r = P_i - P_i^s &= 1 - (1 - s)^{B_i} - \left(1 - (1 - s)^{B_i}\right)^2 \\ &= (1 - s)^{B_i} - (1 - s)^{2B_i} \end{aligned} \tag{3}$$

Equipped with the probability of an access to a cache line we can estimate the number of cache misses per type using Equation 4.

$$M_i^x = P_i^x \times \frac{R.w \times R.n}{B_i} \text{ for } x \in \{r, s\} \tag{4}$$

Prediction Accuracy: To get an impression of the probability for P_i^r and P_i^s with varying s consider Figure 6. The percentage of random and sequential misses increases steeply with the selectivity in the range from $0 < s < 0.05$. After that point, the number of random misses declines in favor of more sequential misses.

To assess the quality of our prediction, we implemented a selective projection in C and measured the induced cache misses using the CPU's Performance Counters. The Nehalem CPU Performance Counters only count *Demand/Requested* L3 cache misses as misses. Misses that are triggered by the prefetcher are not reported, which allows us to distinguish them when measuring. The sequential misses are simply the number of reported L3 accesses minus the reported L3 misses. The random misses are the reported L3 misses. In addition to the predicted, Figure 6 also shows the measured cache misses. The Figure shows that the prediction overestimates the number of random misses for mid-range selectivities and underestimates for very high selectivities. However, the general trend of the prediction is reasonably close to the measured values. To illustrate the improvement of the model as achieved by this new pattern, the figure also shows the predicted number of accessed cache lines when modeling the pattern using a `rr_acc` instead of `s_trav_cr`. It shows that a) the `rr_acc` highly underestimates the total number of misses and b) does not distinguish random from sequential misses. Especially for low selectivities, the model accuracy has improved greatly.

2) Prefetching aware Cost Function: In its original form the Generic cost model [26] distinguishes random and sequential misses (M_i^r and M_i^s respectively) and associates them with different but constant weights (relative costs) to determine the final costs. These weights are determined using empirical calibration rather than detailed inspection and appropriate modeling. This is sufficient for the original version of the model because access patterns induce almost exclusively random or exclusively sequential misses. Since our new atom, `s_trav_cr`, induces both kinds of misses, however, we have to distinguish the costs of the different misses more carefully.

For this purpose, we propose an alternative cost function. Where Manegold et al. [26] simply add the weighted costs induced at the various layers of memory, we use a more sophisticated cost function to account for the effects of prefetching. Since the most important (and aggressive) prefetching usually happens at the LLC, we change the cost function to model it differently.

Prefetching is essentially only hiding memory access latency behind the activity of higher storage and processing layers. Therefore, its benefit highly depends on the time it takes to process a cache line in the faster memory layers. Following the rationale that execution time is determined by memory accesses the costs induced at the LLC are reduced by the costs indexed at the faster caches and the processor registers (which we consider just another layer of memory). If processing the values takes longer than the LLC-fetching, the overall costs are solely determined by the processing costs and the costs induced at the LLC are 0 — the application is CPU-bound. The overall costs for sequential misses in the LLC (in our Nehalem system the Level 3 Cache, hence T_3^s) can, thus, be calculated using Equation 5.

$$T_3^s = max\left(0, M_3^s \cdot l_3 - \sum_{i=0}^{2} M_i \cdot l_{i+1}\right) \tag{5}$$

Following [26], the costs (in CPU cycles) for an access to level i (i.e., a miss on level $i-1$) will be denoted with l_i. Since we regard the CPU's registers as just another level of memory, l_1 denotes the time it takes to load and process one value and M_0 the number of register values that have to be processed.

The overall costs T_{Mem} are calculated by summing the weighted misses of all cache layers except the LLC. The costs for prefetched LLC misses are calculated using Equation 5 and added to the overall costs in Equation 6.

$$T_{Mem} = \sum_{i=0}^{2} M_i \cdot l_{i+1} + T_3^s + M_3^r \cdot l_4 + \sum_{i=4}^{N} M_i \cdot l_{i+1} \quad (6)$$

3) Random Accesses Estimation: To estimate the number of cache misses that are induced by a Repetitive Random Access (`rr_acc`), the work of Manegold et al. includes an equation to estimate the number of unique accessed cache lines (I) from the number of access operations (r) and the number of tuples in a region ($R.n$). While mathematically correct, this equation is hard to compute due to heavy usage of binomial coefficients of very large numbers. This makes the model impractical for the estimation of operations on large tables. For completeness, we report a different formula that we used here.

This problem, the problem of distinct record selection, has been studied extensively (and surprisingly controversial). Cardenas [7], e.g., gives Equation 7 for to estimate the distinct accessed records when accessing one of $R.n$ records r times. Whilst challenged repeatedly for special cases [13], [34], [9], we found the equation yields virtually identical results to the equation from the original cost model while being much cheaper to compute.

$$\mathbf{I}\left(r, R.n\right) = R.n \cdot \left(1 - \left(1 - \frac{1}{R.n}\right)^r\right) \quad (7)$$

Equipped with a model to infer costs from memory access patterns, we can estimate the costs of a relational query by translating it into the memory access pattern algebra. In the rest of this section we will describe this process.

D. Modeling JiT Query Execution

Due to the instruction-like character of the access pattern algebra, we can treat it like a programmable machine with each atomic access pattern forming an instruction. Thus, generating the access pattern for a given physical query plan is similar to generating the actual code to perform the query (see [27] for a description). To generate the access pattern, the relational query plan is traversed in pre-order from its root (see Figure 7) and the appropriate patterns emitted according to Table II. Just like statements in a program, the emitted patterns are appended to the overall pattern.

Note that no operator produces a pattern when entering an operator node for the first time. All operations are performed when data flows into the operator, i.e., when leaving the operator in the traversal process. Joins (hash) behave slightly differently since data flows into them twice, once for each

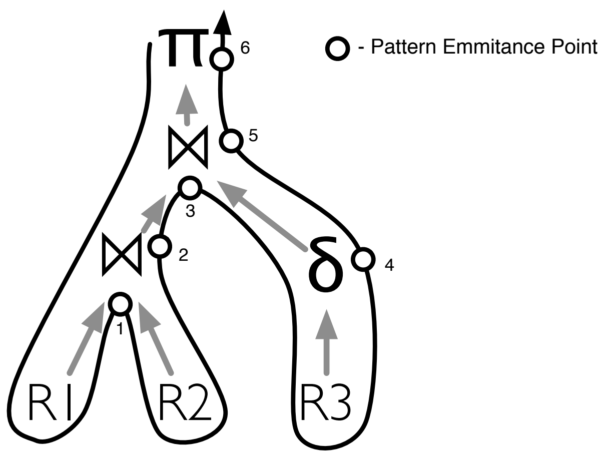


Fig. 7: Plan Traversal

Operation	Emitted Pattern
Select	`s_trav_cr(s, [attributes])` $\odot$
Join (Push) *Hash-Build*	`r_trav([ht_attributes])` $\oplus$
Join (Push) *Hash-Probe*	`rr_acc([ht_attributes])` $\odot$
Join (Pull) *Hash-Build*	`s_trav_cr(s, [ht_attributes])` $\odot$ `r_trav([ht_attributes])` $\oplus$
Join (Pull) *Hash-Probe*	`s_trav_cr(s, [ht_attributes])` $\odot$ `rr_acc([ht_attributes])` $\odot$

GroupBy (Pull)	`s_trav_cr(s, [group_attributes]`
	`+ [aggr_attributes])` $\odot$
	`rr_acc([group_attributes] +`
	`[aggr_attributes])`

GroupBy (Push)	`s_trav_cr(s, [group_attributes]`
	`+ [aggr_attributes] -`
	`[pushed_attributes])` $\odot$
	`rr_acc([group_attributes] +`
	`[aggr_attributes])`

Project (Push)	`s_trav_cr(s, [attributes]`
	`- [pushed_attributes])` $\odot$
	`s_trav([attributes])`

Project (Pull)	`s_trav_cr(s, [attributes])` $\odot$ `s_trav([attributes])`
Sort (Push)	`s_trav_cr(s, [attributes]` `- [pushed_attributes])` $\odot$ `s_trav([attributes])` $\oplus$ `rr_acc([attributes])` $\oplus$
Sort (Push)	`s_trav_cr(s, [attributes])` $\odot$ `s_trav([attributes])` $\oplus$ `rr_acc([attributes])` $\oplus$

TABLE II: Operators and their Access Pattern

of its two children[3]. Thus, they emit patterns twice: once for building the internal hashtable and once for probing it. When emitting patterns, we distinguish *pulling* and *non-pulling operators*. Operators that are within a pipeline fragment that has a join on its lower end are non-pulling, others are pulling. The rationale behind this is that operators placed above a join (e.g., the outer join in Figure 7) do not have to explicitly fetch their input because join operators essentially push tuples into a pipeline fragment as a result of the hash-probing (this effect is explained in detail in [27]). Operators in pull-mode (e.g., the

[3]n-way joins are represented by n nodes

selection in Figure 7) have to pull/read their input explicitly.

As an example, consider the inner join operator in Figure 7: when entering a subtree rooted at the node, no pattern is emitted and the traversal continues with its left child. When leaving the left subtree (Mark 1), a pattern is generated that reflects the hash-building phase of the hashjoin. Since its left child is a base table it has to pull tuples into the hashtable, thus, it emits a `s_trav(R1)` and a concurrent ($\odot$) `r_trav(ht)` of the hashtable. Since the hash-build causes materialization, it breaks the pipeline and marks that by appending a sequence operator $\oplus$ to the pattern. After that, the processing continues with its right child. When leaving the subtree (Mark 2), the hashtable is probed (again in pull-mode) and the tuples pushed to the next hashtable (Mark 3). The emitted pattern at Mark 2 is `s_trav(R2)` $\odot$ `rr_acc(ht)`.

Using this procedure, we effectively program the generic cost model using the access patterns as instructions. While being a simple and elegant way to holistically estimate JiT-compiled query costs, it also allows us to estimate the impact of a change in the storage layout. In the next section, we will use this to optimize the storage layout for a given workload.

V. Schema Decomposition

Finding the optimal schema decomposition is an optimization problem in the space of all vertical partitionings with the estimated costs as objective function. Since the number of vertical partitionings of a schema grows exponentially with the size of the tables, attribute-based partitioning algorithms like the one used in the Data Morphing Approach [16] are impractical due to the high optimization effort (linear with the number of partitionings, thus exponential with the number of attributes). Instead we apply an algorithm that takes the queries of the workload as hints on potential partitionings of the schema. Chu et al. [10] proposed two such algorithms. The first, *OBP*, yields optimal solutions but has exponential effort with regard to the number of considered queries. The second, *BPi* only approaches the optimal solution but has reduced costs (depending on a selectable threshold down to linear). We will apply the BP*i* algorithm to approach the optimal solution.

A. Binary Partitioning

OBP and BP*i* generate potential solutions by iteratively partitioning a table according to what is called a *reasonable cut*. In the original work, every query yields one reasonable cut for every accessed table. The cut is a set of two distinct attribute sets: the ones that are accessed in the query and the ones that are not. The query in our initial example (see Figure 2a), e.g., would yield the cut $\{\{A, B, C, D, E\}, \{F, .., P\}\}$. For workloads with multiple queries, the set of a reasonable cuts also contains all cuts that result from cutting the relation multiple times. Thus, the solution spaces grows exponential with the number of queries.

However, this definition of a reasonable cut is oblivious to the actual access pattern of the query. It does not reflect the fact that different attributes within a query may be accessed in a different manner. E.g., in the initial example $\{\{A\}, \{B, C, D, E\}, ...\}$ is not considered a reasonable cut and therefore never considered for decomposition because the attributes are accessed in the same query. If, however, the selectivity of the condition was 0, $\{B, C, D, E\}$ would never be accessed and storing them in one partition with A would hurt scan performance. It is therefore reasonable to also consider a partitioning that divides attributes that are accessed within one query but in different access patterns. Thus, we consider what we call *Extended Reasonable Cuts* as potential solutions. These are generated from the access patterns rather than the queries. An extended reasonable cut is made up from all attributes that are accessed together within an atomic pattern or in concurrently ($\odot$) executed atomic access patterns of the same kind. E.g., `s_trav(a)` $\odot$ `s_trav(b)` would lead to one cut while `s_trav(a)` $\odot$ `r_trav(b)` would lead to two. For concurrent `s_trav_cr` the case is slightly more complicated. Depending on the selectivity s_1 and s_2 in `s_trav_cr(a,`s_1`)` $\odot$ `s_trav_cr(b,`s_2`)` , the values of a and b may or may not be accessed together. If, e.g., $s_1 = s_2 = 1$, the traversed attributes are always accessed together and do not have to be considered for decomposition. If the selectivity is less than 1, we have to consider all possible cuts: $\{\{\{A\}, ...\}, \{\{B\}, ...\}, \{\{A, B\}, ...\}\}$.

Based on the solution space of all reasonable cuts, BP*i* employs a Branch and Bound strategy to reduce the search space. The schema is iteratively cut according to a (randomly) selected cut. This cut is considered for inclusion in the solution by estimating its cost improvement. If the improvement is above a user defined threshold, the cut is considered for inclusion and the algorithm branches into two cases: including or excluding the cut in the solution. If the cost improvement is below that threshold the cut is not considered for inclusion and the subtree pruned. While sacrificing optimality this reduces the search space and thus the optimization costs.

VI. Evaluation

The benefits of cache-conscious storage do not only depend on the query processing model but also on characteristics of the workload, schema and the data itself. It is, therefore, not enough to evaluate the approach only on a single application. We expect wider tables and more diverse queries to benefit more from partial decomposition than specialized queries on narrow tables. To support this claim we evaluated using three very different benchmarks.

1) The SAP Sales and Distribution Benchmark that was used to benchmark the HYRISE system [15]. We consider the schema relatively generic in the sense that it support multiple use cases and covers business operations for enterprises in many different countries with different regulatory requirements.
2) The CH-benchmark [11], a merge between the TPC-C and TPC-H benchmarks, modeling a very specific use case: the selling and shipping of products from warehouses in one country.
3) A custom set of queries on the CNET product catalog dataset [4] designed to reflect the workload of such

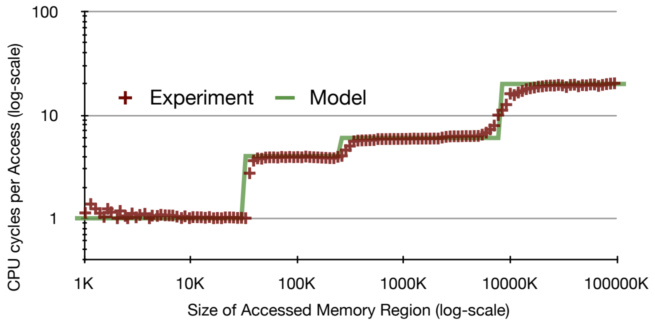


Fig. 8: The configuring experiment

Level	Capacity	Blocksize	Access Time
L1 Cache	32 kB	8 B	1 Cyc
L2 Cache	256 kB	64 B	3 Cyc
TLB	32 kB	4 kB	1 Cyc
L3 Cache	8 MB	64 B	8 Cyc
Memory	48 GB	64 B	12 Cyc

TABLE III: Parameters used for the model

a product catalog web application. Due to the variety of attributes of the different products, we consider the schema very generic.

Since the non-hybrid HyPer system has been shown to be competitive to established DBMSs (VoltDB and MonetDB) [22] we will focus our evaluation on the partial decomposition aspect of the system.

A. Setup

We evaluated our approach on a system based on the Intel Nehalem Microarchitecture as depicted in Figure 4 with 48 GB of RAM. The 4 CPUs were identified as "Intel(R) Xeon(R) CPU X5650 @ 2.67 GHz" running a "Fedora 14 (Laughlin) Linux" (Kernel Version: 2.6.35.14-95.fc14.x86_64). Since the model relies on more detailed parameters to predict costs we will discuss how to obtain these in more detail.

Training the Model: The cost estimation relies on parameters that describe the memory hierarchy of a given system. To a large extent, these parameters can be extracted from the specification or read directly from the CPU using, e.g., cpuinfo_x86[4]. The reported information includes the capacities and blocksizes of the available memory layers but not their respective latency. To determine these experimentally, we, inspired by the Calibrator[5] of the Generic Cost Model, implemented the following experiment in C to determine the latencies: calculate the sum of a constant number of values varying the size of the memory region that they are read from (and thus the number of unique accessed values). Figure 8 shows the execution time in cycles per summed value as a function of the size of the accessed memory region. The latencies of the different memory layers become visible whenever the size of the accessed region exceeds the capacity of a memory layer. The latencies can be determined from this graph manually or automatically by fitting the curve to the data points. Table III lists the parameters of the cost model and their values for our system.

Besides providing the needed parameters, this experiment illustrates the significance of a hardware-conscious cost model. If we only counted misses on a single layer (e.g., only processed values or only L2-misses) we would underestimate the actual costs. This observation is what triggered the development of the Generic Cost Model in the first place [26]. Due to space limitations we focus our evaluation on the high level impact of partial decomposition rather than the accuracy of the cost model. In addition to the original validation [26], we performed an extensive study of the extended generic cost model and demonstrated it's validity on current hardware [28]. We determined an appropriate layout for each of our three benchmarks using our extended BP*i*.

B. The SAP-SD Benchmark

The SAP Sales and Distribution (SD) Benchmark was used to evaluate the HYRISE system [15] and illustrates a performance gain of partial decomposition in a moderately generic case. We consider this benchmark to cover the middle ground between the highly specialized CH-benchmark and the very generic CNET Products case. We implemented the benchmark using the reported queries [15] on publicly available schema information[6]. We filled the database with randomly generated data, observing uniqueness constraints where applicable.

Decomposition: Due to space limitations, we cannot cover the optimization process in detail here[7]. To give an impression of the optimization process we briefly discuss the decomposition of the `ADRC` table of the SD benchmark (see Table IV). Query 1 and 3 of the benchmark (see Table IVa) operate on that table. Query 1 scans `NAME1` and (conditionally) `NAME2` to evaluate the selection conditions and Query 3 scans `KUNNR`. The extended reasonable cuts that originate from their plans are listed in Table IVb. The optimization yields the decomposition as listed in Table IVc: The first three partitions support the scans of the queries efficiently. Since `NAME2` is only accessed if `NAME1` does not match the condition, these are decomposed. `KUNNR` is stored in the third partition to support Query 3. The fourth partition supports the projection of Query 1 and the last partition the projection of Query 3.

Results: For reference, we compare the performance of the JiT-compiled queries to the processing model of the HYRISE system. HYRISE uses a bulk-oriented model but still relies on function calls to process multiple attributes within one partition. It therefore suffers from the same CPU inefficiency as the Volcano model. Figure 9 shows the results for queries one to twelve of the benchmark. We observed that, in general, JiT-compiled queries have similar relative costs on different layouts as volcano processed ones. However, the processing costs of the HYRISE processor are much higher (note the

[4]`http://osxbook.com/book/bonus/misc/cpuinfo_x86/cpuinfo_x86.c`

[5] `http://www.cwi.nl/~manegold/Calibrator`

[6]`http://www.se80.co.uk`, `http://msdn.microsoft.com/en-us/library/cc185537(v=bts.10).aspx`

[7]We are currently working to make the optimizer publicly available.

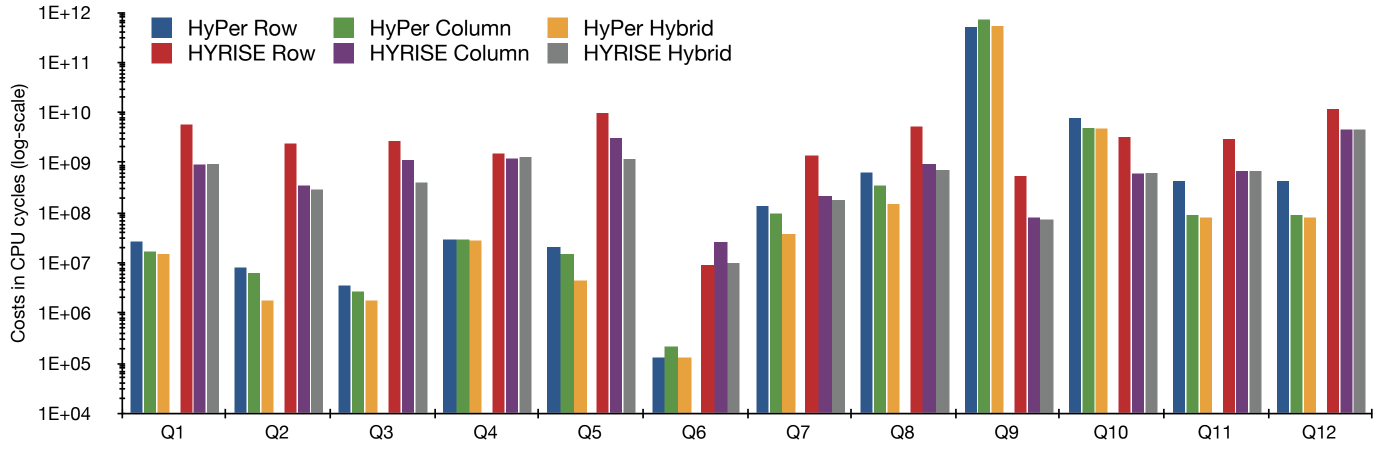


Fig. 9: Hybrid Storage Performance with and without JiT compilation

Q1	`select ADDRNUMBER, NAME_CO, NAME1, NAME2, KUNNR from ADRC where NAME1 like $1 and NAME2 like $2;`
Q3	`select * from ADRC where KUNNR = $1`

(a) Queries

```
{{NAME1},{NAME2},{NAME1,NAME2},
{KUNNR},{NAME_CO,ADDRNUMBER},
{ADDRNUMBER,NAME_CO,NAME1,NAME2,KUNNR},
{ADDRNUMBER,NAME_CO,KUNNR},{*}}
```

(b) Extended Reasonable Cuts

```
{{NAME1},{NAME2},{KUNNR},{ADDRNUMBER,NAME_CO},{*}}
```

(c) Solution

TABLE IV: Decomposition of the ADRC-table

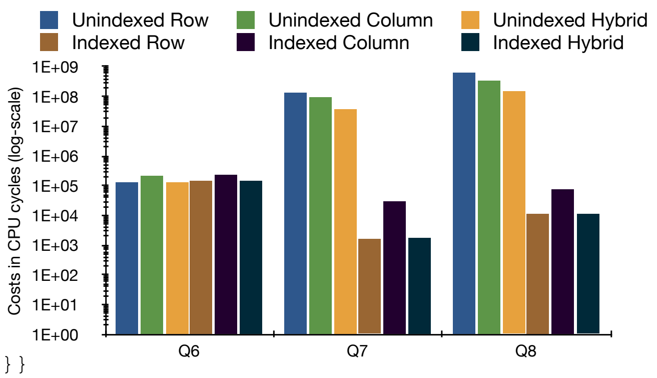


Fig. 10: Hybrid Storage With and Without Indexes

log-scale) than the costs of the JiT-compiled queries. For scan-heavy queries like, e.g., Query 1, this can go beyond an order of magnitude. This confirms our expectations, since it reflects the performance advantage of bulk- over Volcano-processors that has been reported for memory-resident databases [5].

Two queries, 9 and 10, show significantly worse performance in the HyPer system than in the competitor. In both cases, the HYRISE system uses metadata information about the data (Implicit ordering) for query plan optimization that are not exploited by HyPer. Another notable fact is that the only modifying query of the benchmark, Query 6, is much cheaper in HyPer. Being designed with update/insert performance in mind, insert queries in HyPer are processed in an almost bulk-insert like manner. For bulk inserts/appends, the performance penalty of decomposed over N-ary storage is less severe (in our case, ca. 60 %). This leads to the observed good transactional insert performance.

Indexes: It has been claimed that column-stores do not benefit from indexes due to cheap column-scans that can be used for tuple retrieval [29]. While Column-Scans are hard to avoid for search-like queries like Query 1, queries that are mere identity-selects may benefit more from indexes in addition to decomposition. In the SD benchmark, e.g., Queries 7 and 8 are instances of such queries. To investigate the benefit of indexed selects on various storage layouts we created supporting indexes (hash indexes for primary keys and one RB-Tree on `VBAP(VBELN)`) for these queries using the same storage strategy. We also looked at the impact the maintenance of these indexes has on the modifying Query 6. Figure 10 shows the results of these experiments. We found that the performance penalty for index maintenance at inserts (Query 6) was negligible. Queries 7 and 8, that had to rely on scans to locate matching tuples in the absence of indexes, experience a performance boost of more than 1,000x in a column- and more than 10,000x in a row-store. Since the query costs are now largely determined by the tuple reconstruction the row-store is out-performing the column-store by about an order of magnitude. This indicates that whilst partial decomposition improves scan performance for, e.g., aggregations or full-text-search, indexes are better suited for tuple retrieval.

C. The CH-Benchmark

Our second benchmark, the CH-Benchmark was designed to simulate a use case that involves analytical as well as transactional operations, thus creating a conflicting benchmark. We, therefore, started out expecting a significant improvement

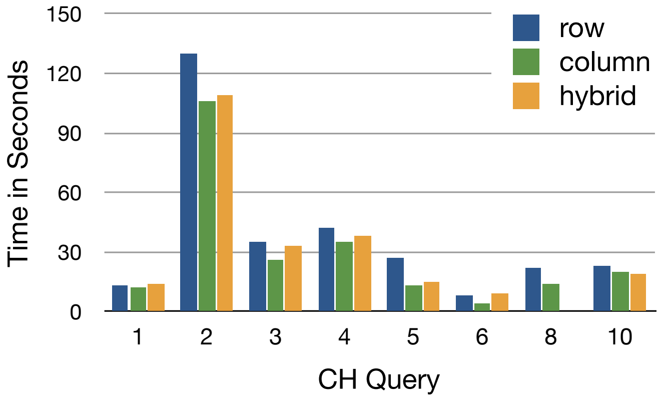


Fig. 11: Query Evaluation time

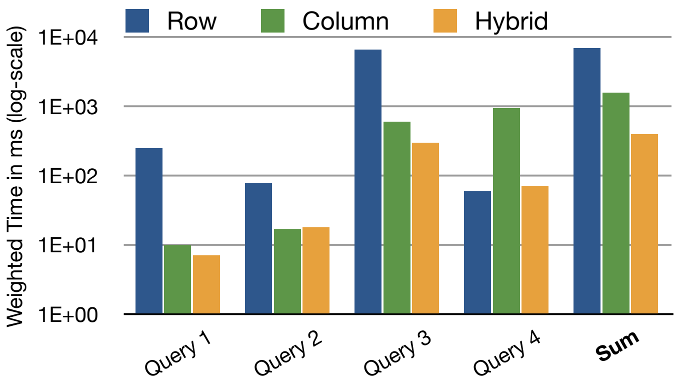


Fig. 12: CNET Results

in the overall workload performance. However, as depicted in Figure 11, the benefit of partial decomposition is not as high as we initially expected. We also noticed that even a full decomposition (DSM) does only yield an improvement of about 30 percent in comparison to N-ary storage for the analytical queries. This seemingly stands in contrast to previously published results that report orders of magnitude difference between row- and column-stores for analytical queries [5]. This divergence indicates that other factors than the storage strategy contribute to the superior analytical performance of column-stores: the CPU efficiency of the simple, tight loops of a bulk query processor. Since JiT-compilation always generates tight loops, there is little to be gained from decomposed storage. It is not that the evaluated column-store implementation is deficient but the row-store implementation leaves little room for improvement in this benchmark.

D. The CNET-Products Benchmark

The CNET Products Data Set [4] is a description of the properties of the product catalog of the CNET review site. Since it contains data on many different products, the catalog relation is very wide (almost 3000 attributes) but sparsely populated (the average tuple contains 11 non-null values). However some attributes like manufacturer, name and category are set for all tuples and can be used for analytics. Schemas like this occur frequently when mapping object oriented class hierarchies to relational tables due to the lack of inheritance in the relational model. In this case, all classes in a hierarchy and their attributes are mapped to the same table. Such schemas make good candidates for partial decomposition. To fill the CNET schema, we implemented a generator to create relational data according to the reported properties[8].

Unfortunately, the CNET Products Data Set Description only reports on the properties of the data. It does not provide an application or queries on top of that data. Inspired by the functionality of the CNET products website, we created a set of four queries (see Table V) to simulate the load of a web application. The first three queries correspond to a user navigating the catalog to get an overview of the available data. Even though they focus on end-users, they are, by character, analytical queries. The results of the first two queries may be cached in, e.g., a materialized view and, consequently, have a low frequency in our benchmark. The third query relies on user input and is, thus, harder to cache. Since it is a browsing query we assign it medium frequency. The fourth query shows the details of a particular product given its primary key. This query simulates a direct link to a product page (potentially from an external site) and is an OLTP-style lookup. For websites this is a very common operation, hence its frequency is much higher than that of the other queryies.

Figure 12 shows the results of the CNET-benchmark. For the analytical queries decomposed storage outperforms N-ary storage as expected. Query 3 shows a slight performance benefit from collocating `id` and `name` over full decomposition. The fourth query performs best on an N-ary relation but only shows slight degradation on a partially decomposed relation. Overall, the partial decomposition model performs more than an order of magnitude better than the N-ary mode and almost 4x better than the fully decomposed storage.

[8] `http://www.cwi.nl/~holger/generators/cnet`

VII. Conclusion

Partial decomposition is a promising means to reduce the data access costs in a MRDBMS. To fully exploit its potential, however, it is crucial to avoid sacrificing CPU efficiency for savings in bandwidth. JiT-Compiled queries naturally avoid any CPU-overhead and are, therefore, a natural match to the Partially Decomposed Storage Model. By combining these techniques we achieved the promised bandwidth savings without the CPU overhead at query evaluation time. We found orders of magnitude gain when replacing a hybrid DBMS based on flexible, Volcano-like operators by a system that JiT-compiles queries.

Whilst partial decomposition does not degrade performance, the benefit depends largely on the schema and workload of the database. As a rule of thumb we found that, the more generic/wide a schema and scan- and projection-heavy a workload is, the higher the benefit of a partial decomposition. For a very generic database schema like the CNET-dataset or the SAP SD benchmark, the improvement can be significant (factor 3 and

Query	Frequency	Description
`select category, count(*) from products group by category`	1	Give overview of all categories with product counts
`select (price_from/10)*10 as price, count(*) from products where category = $1 group by price order by price;`	1	Drill down to a category and show price ranges
`select id, name from products where category=$1 and (price_from/10)*10 = $2`	100	Show a Listing of all products in a category for the selected price range
`select * from products where id=$1`	10,000	Show available Details of a selected Product

TABLE V: The Queries on the CNET Product Catalog

more). We therefore believe that workload-conscious storage optimization is an interesting field for further research.

Beyond schema decomposition there are a number of other workload-conscious storage optimizations. Especially with the focus on sparse data the *storage as dense key-value lists* is an option that may save storage space and processing effort. We also expect such a key-value storage to be easier to integrate into existing column-stores than a new processing model like JiT. *Partial compression* may work well when data is not sparse but has a small domain and might be a good application for our hardware-conscious cost model. Another area is *online/adaptive reorganization* of the decomposition strategy and *Query-Layout-Co-Optimization*.

Acknowledgments

The work reported here has partly been funded by the EU-FP7-ICT project TELEIOS.

References

[1] D. Abadi, D. Myers, D. DeWitt, and S. Madden. Materialization strategies in a column-oriented dbms. In *ICDE '07*, pages 466–475. IEEE, 2007.

[2] A. Ailamaki, D. DeWitt, M. Hill, and D. Wood. Dbmss on a modern processor: Where does time go? In *VLDB '99*, pages 266–277, 1999.

[3] D. Batory. Concepts for a database system compiler. In *Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 184–192. ACM, 1988.

[4] J. Beckham. The cnet e-commerce data set. *University of Wisconsin, Madison, Tech. Report*, 2005.

[5] P. Boncz, S. Manegold, and M. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB '99*, pages 54–65, 1999.

[6] P. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR '05*, pages 225–237, 2005.

[7] A. Cárdenas. Analysis and performance of inverted data base structures. *Communications of the ACM*, 18(5), May 1975.

[8] D. Chamberlin, M. Astrahan, M. Blasgen, J. Gray, W. King, B. Lindsay, R. Lorie, J. Mehl, T. Price, F. Putzolu, et al. A history and evaluation of system r. *Communications of the ACM*, 24(10):632–646, 1981.

[9] T.-Y. Cheung. Estimating block accesses and number of records in file management. *Communications of the ACM*, 25(7), Jul 1982.

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

[11] R. Cole, F. Funke, L. Giakoumakis, W. Guy, A. Kemper, S. Krompass, H. Kuno, R. Nambiar, T. Neumann, M. Poess, et al. The mixed workload ch-benchmark. In *DBTest '11*, page 8. ACM, 2011.

[12] G. P. Copeland and S. N. Khoshafian. A decomposition storage model. In *ACM SIGMOD '85*, pages 268–279, New York, NY, USA, 1985. ACM.

[13] G. Diehr and A. Saharia. Estimating block accesses in database organizations. *IEEE TKDE*, Jan 1994.

[14] G. Graefe. Volcano-an extensible and parallel query evaluation system. *IEEE TKDE*, 6(1):120–135, 1994.

[15] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. Hyrise: a main memory hybrid storage engine. *VLDB '10*, 4(2):105–116, 2010.

[16] R. Hankins and J. Patel. Data morphing: An adaptive, cache-conscious storage technique. In *VLDB '03*, pages 417–428. VLDB Endowment, 2003.

[17] S. Harizopoulos, D. Abadi, S. Madden, and M. Stonebraker. Oltp through the looking glass, and what we found there. In *ACM SIGMOD '08*, pages 981–992. ACM, 2008.

[18] R. Hedge. Optimizing application performance on Intel Core microarchitecture using hardware-implemented prefetchers, 2007.

[19] J. Hennessy and D. Patterson. *Computer architecture: a quantitative approach.* Morgan Kaufmann Pub, 2011.

[20] R. Johnson, V. Raman, R. Sidle, and G. Swart. Row-wise parallel predicate evaluation. *VLDB '08*, 1(1):622–634, 2008.

[21] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE '11*, pages 195–206, 2011.

[22] A. Kemper and T. Neumann. One size fits all, again! the architecture of the hybrid oltp&olap database management system hyper. *Enabling Real-Time Business Intelligence*, pages 7–23, 2011.

[23] M. Kersten, S. Plomp, and C. Berg. Object storage management in goblin. *IWDOM '92*, 1992.

[24] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE '10*, pages 613–624. IEEE, 2010.

[25] C. Lattner and V. Adve. The llvm instruction set and compilation strategy. *CS Dept., Univ. of Illinois at Urbana-Champaign, Tech. Report UIUCDCS*, 2002.

[26] S. Manegold, P. Boncz, and M. L. Kersten. Generic database cost models for hierarchical memory systems. In *VLDB '02*, pages 191–202. VLDB Endowment, 2002.

[27] T. Neumann. Efficiently compiling efficient query plans for modern hardware. In *VLDB '11*, VLDB, 2011. VLDB, VLDB Endowment.

[28] H. Pirk. Cache conscious data layouting for in-memory databases. Master's thesis, Humboldt-Universität zu Berlin, available at http://oai.cwi.nl/oai/asset/19993/19993B.pdf, 2010.

[29] H. Plattner. A common database approach for oltp and olap using an in-memory column database. In *ACM SIGMOD '09*, pages 1–2. ACM, 2009.

[30] H. Plattner and A. Zeier. In-memory data management, 2011.

[31] J. Rao, H. Pirahesh, C. Mohan, and G. Lohman. Compiled query execution engine using jvm. In *ICDE'06*, pages 23–23. Ieee, 2006.

[32] J. Sompolski, M. Zukowski, and P. Boncz. Vectorization vs. compilation in query execution. In *DaMoN '11*, pages 33–40. ACM, 2011.

[33] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. *ACM SIGMOD Record*, 29(3):55–67, 2000.

[34] S. Yao. Approximating block accesses in database organizations. *Communications of the ACM*, 20(4), Apr 1977.

[35] M. Zukowski, P. A. Boncz, N. Nes, and S. Héman. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Engineering Bulletin*, 28(2):17–22, June 2005.