Datenbank Spektrum (2020) 20:43–56
https://doi.org/10.1007/s13222-019-00330-x

FACHBEITRAG

GridTables: A *One-Size-Fits-Most* H^2TAP Data Store

Vision and Concept

Marcus Pinnecke[1] · Gabriel Campero Durand[1] · David Broneske[1] · Roman Zoun[1] · Gunter Saake[1]

Received: 10 October 2019 / Accepted: 28 December 2019 / Published online: 31 January 2020
© The Author(s) 2020

Abstract
Heterogeneous Hybrid Transactional Analytical Processing (H^2TAP) database systems have been developed to match the requirements for low latency analysis of real-time operational data. Due to technical challenges, these systems are hard to architect, non-trivial to engineer, and complex to administrate. Current research has proposed excellent solutions to many of those challenges in isolation – a unified engine enabling to optimize performance by combining these solutions is still missing. In this concept paper, we suggest a highly flexible and adaptive data structure (called GRIDTABLE) to physically organize sparse but structured records in the context of H^2TAP. For this, we focus on the design of an efficient highly-flexible storage layout that is built from scratch for mixed query workloads. The key challenges we address are: (1) partial storage in different memory locations, and (2) the ability to optimize for mixed OLTP-/OLAP access patterns. To guarantee safe and well-specified data definition or manipulation, as well as fast querying with no compromises on performance, we propose two dedicated access paths to the storage.
In this paper, we explore the architecture and internals of GRIDTABLES showing design goals, concepts and trade-offs. We close this paper with open research questions and challenges that must be addressed in order to take advantage of the flexibility of our solution.

Keywords HTAP · Relational Databases · Heterogeneous Databases · Database Management · Physical Database Design

1 Introduction

In the last decade, the database research community has focused on challenges for data management and system design implied by the ongoing needs to manage and analyze web-scale, frequently changing, diverse datasets. One key challenge is to minimize the latency between operational and analytical systems [34, 45, 48]. For *Hybrid Transactional Analytical Processing* (HTAP) systems, new architectures were proposed that enable low latency analysis on real-time operational data. A good overview about this topic can be found in a recent survey by Özcan et al. [45]. A key enabling factor for HTAP systems is modern hardware: modern hardware promises novel ways for data processing of relational [14, 25] and non-relational data [42, 49], as well as benefits for several database system components, such as query optimization [26, 41]. Appuswamy et al. even suggested to use the term H^2TAP whenever hybridization of workloads is combined with heterogeneity of hardware [6], effectively emphasizing the role of modern hardware.

In previous work [50], we questioned whether current database systems on modern hardware are really future-proof and ready for H^2TAP workloads. We concluded the existence of missing synergy effects in the state-of-the-art since existing solutions are examined in isolation which leaves optimization potential unexplored and unexploited, such as unsatisfactory support of row-wise storage for co-processors, adaptive indexing across multiple devices, or an excellent online re-organization for H^2TAP workloads for cross-device databases as already studied in depth for CPU-only database systems.

In addition to that, it is not yet clear how to combine novel research suggestions in a unified system, and how such suggestions may affect or benefit from each other. In particular, our research community shows opportinities and

✉ Marcus Pinnecke
pinnecke@ovgu.de

[1] Institute of Technical and Business Information Systems, Database and Software Engineering Group, University of Magdeburg, Magdeburg, Germany

challenges of modern hardware in database systems in isolation, among them the need for analysis of novel adaptive data layouts and data structures for operational and analytical systems [6, 7, 9, 55], novel processing, storage and federation approaches on non-relational data models [11, 18, 51, 52, 62], benefits and drawbacks of porting to new compute platforms [12, 16, 33, 63], opportunities and limitations of GPUs and other co-processors as building blocks for storage and querying purposes [8, 14, 33], novel proposals for main memory databases on modern hardware [3, 15, 21, 53, 56], adaptive optimization, and first attempts towards self-managing database systems [17, 36, 43, 46].

In this paper, we aim for a novel storage engine design, called GRIDSTORE that manages relations with a data structure that we name GRIDTABLES, in order to face the challenges of H^2TAP on multiple devices by enabling the combination of established solutions so far considered in isolation. Relations in GRIDTABLES are flexibly partitioned into a set of self-contained, and placement-aware containers, called *grids*. Each grid by its own is able to perform local optimizations regarding *schema re-ordering*, to avoid cache thrashing for wide records (cf. [13] for OLAP-only), and *record organization* to optimize the data access path and minimize data redundancy (cf. [1]).

A GRIDTABLE implements a *flexible and adaptive record layout* (cf. [4, 9, 23, 55]) to allow zero-cost *null-value suppression*, to enable *the combination of logically distant record fields into physical dense blocks*, and to *perform global layout adaption*. In contrast to existing partitioning capabilities in enterprise systems, a relation can therefore be partitioned to any combination of vertical and horizontal (logical) fragments with a granularity from the table level to tuple-field values, if desired.

GRIDTABLES enable a fine-grained physical optimization of a single database by transitioning between a transactional storage, an analytical storage, and a mixed storage based on the actual usage. Transitions respect user-specific data model definitions and constraints, and are executed via local and global optimizations on the GRIDTABLE. Analytical query performance is improved by *(implicit) denormalization* (similar to a WIDETABLE, [39]), and transactional query performance is improved by *(implicit) normalization*.

We begin with an overview picture, showing a feature summary of our data store (Sect. 2). We then continue with sections containing the following contributions:

- *Requirement Analysis*. We state requirements for a storage engine matching a *One-Size-Fits-Most* design for competing access patterns and optimization goals, co-processor support and self-tuning (Sect. 3).
- *Flexible Data Storage*. We propose a stacked architecture for highly-flexible partitioning, multiple storage formats and placement options (Sect. 4).
- *Design Space Exploration*. We discuss most representative aspects in a flexible storage for H^2TAP: data storage and querying (Sect. 5).
- *Open Challenges*. With GRIDTABLES, we broaden the canvas for (autonomous) optimization, and explore optimization problems that we seek to address with our proposal, such as table partitioning and baseline heuristics (Sect. 6).

We end this paper with a discussion on related work (Sect. 7), and our conclusion (Sect. 8).

2 GridTables: Big Picture and Vision

The ultimate vision behind GRIDTABLES is to create a storage engine for H^2TAP database systems that fully supports both multi-core CPUs and many-core GPUs without making any cutbacks in terms of data freshness, isolation, and transactional consistency.

In this paper, we focus on the storage engine and on storage-engine core operations (i.e., scans and materialization) rather than on operations that fall into the domain of the query engine (and thus, are more coupled with the co-processor-aware aspects of our design).

2.1 The Need for HTAP on Modern Hardware

In this section we establish the need for an H^2TAP store on modern hardware, based on a motivating experiment. Next, we summarize key requirements for such a system. We conclude the section by outlining essential features of GridTables.

A dedicated H^2TAP system design is motivated by the observation that both operational and analytical access patterns inside a single (hybrid) workload imply *different* and (sometimes) *contradicting* optimizations, such as for physical record organization [9, 23], hot/cold data classification and handling [37], or the choice to run entire queries in parallel (i.e., *inter-query parallelism*) vs to run particular parts of a single query in parallel (i.e., *intra-query parallelism*) [50].

In the following, we show an extract of our experiments performed in our previous work [50] in Fig. 1, and invite readers interested in details beyond the scope of this paper to a read. For the ease of scoping, we limit our insights to host-based experiments only, and do not explore host-/device effects in our current paper.

Setup. In our host-based experiments, we examined the effect of physical table layouts (i.e., row-store/column-store), the query parallelism policy used on the query throughput for varying access patterns, and an increasing number of tuples stored in a table. As a dataset we used

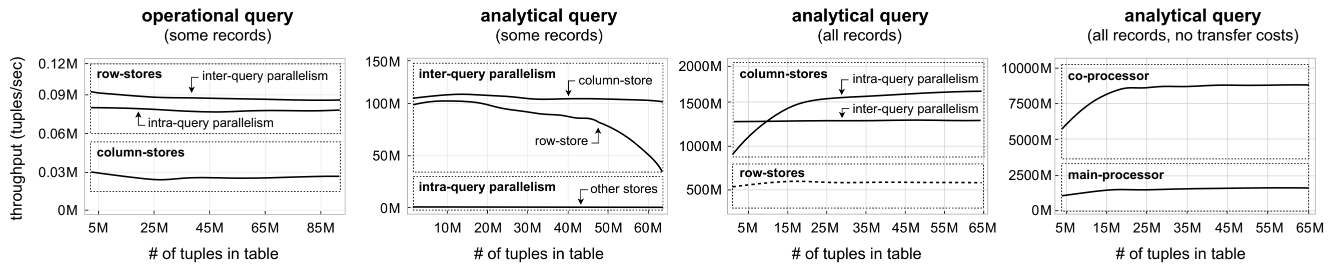

Fig. 1 Query throughput as function of data size, grouped by storage format and parallelism policy for mixed operational and analytical access patterns (running on host platform)

the *customer* and *lineitem* tables of the popular TPC-C benchmark. In detail, we issued (scan) queries computing the sum of a randomly chosen attribute (i.e., attribute-centric queries) in the *lineitem* table for all tuples and some (n = 150) tuples, and queries materializing all fields of some (m = 150) tuples (i.e., record-centric queries) in the *customer* table.

Insights. We concluded that there is no clear winner configuration: the physical storage layout and the query parallelism policy affect the query performance. For instance, due to thread-management costs, single-threaded execution is beneficial for record-centric queries as long as the number of tuples to be materialized is small. At a (system-specific) threshold on this number of tuples for the same query, changing the parallelism policy to a data-parallel execution strategy is more reasonable. Likewise, to optimize for an attribute-centric query, a columnar record layout (DSM) would be more fitting.

Consequences. In case of a mix of both query types, neither column-store nor row-store is always the best choice and it is not trivial to determine which to chose – especially if the workload changes over time. Both storage layout and query parallelism policies are sometimes tightly coupled in their optima for a specific case – but for the hybrid case, unfortunately all possible combinations of access pattern and query parallelism policy might be relevant.

3 Concrete Requirements

One cannot expect a *One-Size-Fits-All* design solving every problem in the domain of H^{2}TAP in an optimal way, as shown by Athanassoulis in 2016 for optimizations involving read times, update cost, and memory requirements at once [10]. As a consequence, we suggest a *One-Size-Fits-Most* design under the following requirements:

1. *Best for Pure Transactional Access Patterns*. Records must be quickly accessible to point queries over their primary key values. Therefore, the storage engine must support record-centric access. Read/write operations for a single tuple must be cache efficient. When issued with transactional workloads and multiple requests, the storage engine should not spend valuable CPU time in management of concurrency.
2. *Best for Pure Analytical Access Patterns*. The storage engine must support analytical queries on massive amounts of (denormalized) data without compromising the complexity of these analytical tasks. Therefore, the storage engine must support efficient range-queries in a column-centric manner.
3. *Physical Adaptiveness for* H^{2}TAP. When the system is issued with both transactional and analytical queries, the query performance should match a pure transactional system when the queries are transactional-major, and should match a pure analytical system when the queries are analytical-major. Everything between those two extremes should be smoothly interpolated. The performance penalty for accessing operational data for long-running analytical purposes should be minimized.
4. *Co-Processor Acceleration & Data Placement*. For compute-intensive analytical tasks, the engine should be able to use NUMA-styled co-processors, such as GPUs or FPGAs. In case that data is too large to be stored in the device memory of such a co-processor, the storage engine should use the co-processor on a dataset portion which fits into the device memory, and for which the largest performance gain can be expected. In fact, dismissing the use of the co-processor (e.g., by rolling back to CPU) should *not* be triggered by the data set size.
5. *Knobs for Autonomous Optimization*. The requirements mentioned above lead to a huge optimization space with an enormous amount of possible configurations. It can be expected that straight-forward user empowerment will leave optimization potential unused. Therefore, the storage engine must expose tuning knobs and informative statistics such that an external self-driving system component could instrument the storage engine to iteratively configure itself towards the most promising context-aware configuration.

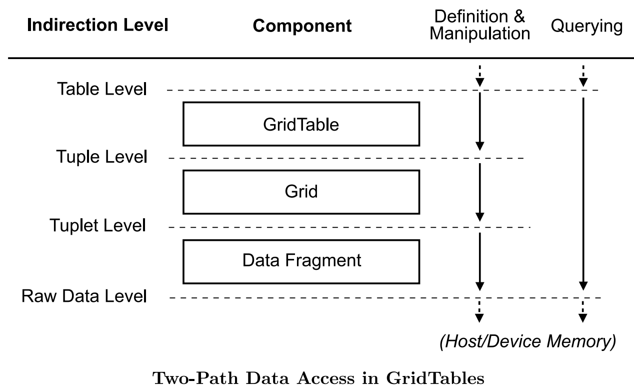

Fig. 2 Stacked architecture at a glance: indirection levels and components as well as two-way access path to raw data stored in host or device memory

In addition to the requirements mentioned above, there are a series of technical challenges and needs for H^2TAP systems on modern hardware.

Recently, Appuswamy et al. pointed to multi-socket multi-core platforms that require careful design for global shared memory, cache coherence and massive parallelism, coining the term H^2TAP as a new architecture built for this purpose [6].

In Fig. 2, we show the stacked architecture of our proposal, GRIDTABLES. Each indirection level is bundled with a particular set of level-specific functions that we explore more in detail in Sect. 5.4.

The three conceptual main components are GRIDTABLES, *grids*, and *data fragments*. From top to bottom: a GRIDTABLE is a data structure that manages multiple layouts for a relation. Each of these layouts is a combination of vertical and horizontal partitioning where a particular partition has no partitioning-related side-effects to adjacent partitions. A grid is a component that realizes one particular partition including its own physical schema, or indexes. Each grid consists of exactly one data fragment which is a plain storage implementation (such as column store or row store) for relational data that accesses host or device memory directly.

To avoid undesired effects by wrongly chosen partitions (such as splitting an OLTP-related tuple into two parts by vertical partitioning), the responsible decision process must consider a range of constraints, e.g., implied by the workload, or by service level agreements with the client. We explore related problems more in detail in Sect. 6, and study a solution option for a decision process that relies on reinforcement learning to improve from experience while seeking to avoiding execution overheads from online partitioning algorithms, in dedicated papers [19, 20].

Data access in complex structures (e.g., in GRIDTABLES) is a trade-off design space. On the one hand, a clear conceptual access path is needed that abstracts from low-level details and which solves important design-related requirements, such as reusability and understandability. In this path, safe operations and usability rather than high performance access are the goals. On the other hand, such properties come often with the cost of additional call overhead that is unacceptable for aggregation-heavy operations as typical for analytical queries over huge amounts of columnar data. For these requirements, safe operation and usability play a minor role. Therefore, a GRIDTABLE exposes two ways to access raw data, one for definition and manipulation (a safe path) and one for querying purposes (a fast path).

3.1 Definition and Manipulation

The *definition and manipulation path* adopts a carefully designed abstraction API that is engineered with the goal of a well-defined, reliable, and secure path to the data. The primary purpose of the definition and manipulation path is data loading.

Conceptually, this path abstracts from low-level raw data management over the following indirection levels: (i) the table level consists of logic that affects the table as a whole (such as snapshotting in-memory tables to secondary storage at specific intervals). The table level accesses (ii) the tuple level, which is about management of entire records that may fall into several grids (i.e., fall into several *tuplets*). This level is for reading and writing entire tuples without the need to care about how the physical organization actually looks like. The tuple level accesses (iii) the tuplet level, which abstracts from low-level operations such as seeking to particular positions in a raw byte array in DRAM. This level is used to update or read individual fields in the boundaries of a grid. Finally, each grid translates the calls from the tuplet level into low-level operations that are highly affected by the actual storage strategy at hand, the (iv) raw data level. The raw data level computes the number of bytes and the position inside the raw data that must be read/written when a particular record field is read/written via the tuplet level.

By design, we use the definition and manipulation path for generic data load, diagnostics and debugging purposes.

4 A Stacked Architecture Concept

To address our requirements stated in Sect. 3, we provide the following features for GRIDTABLES (see Fig. 3) that are, to the point of this writing, instrumented by the client rather than by the system itself: (1) *Flexible Partitions* that allow highly-flexible intra-tuple data formats, (2) *Per-Grid Formats*, to format tuplets column-wise or row-wise, (3) *Per-Grid Storage* enabling the storage of tuplets in host or device memory, (4) *Data Packing*, enabling the storage of

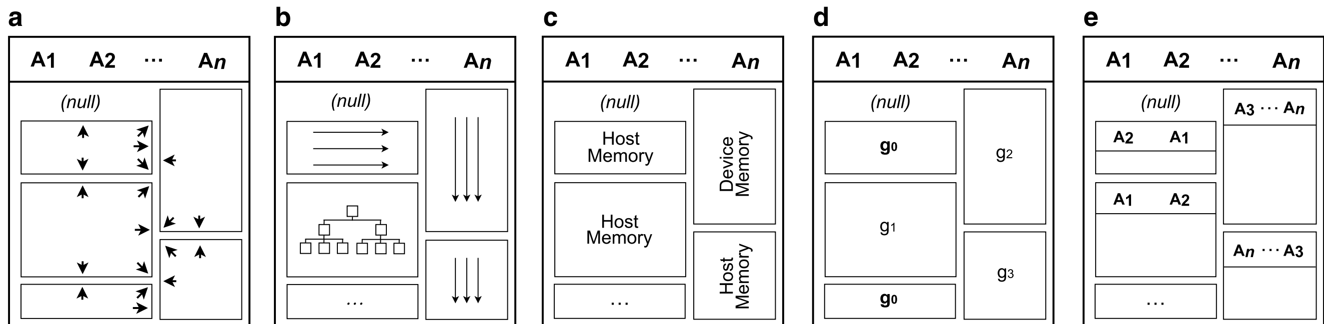

Fig. 3 Feature summary of GRIDTABLES. Flexible partitions **(a)**, per-grid formats **(b)**, per-grid storage **(c)**, data packing **(d)**, and schema-reordering **(e)**

logically distant fields in a physically contiguous manner, and (5) *Schema-Reordering*, re-ordering per-grid fields for data cache efficiency. In the following, we explore details on these features.

Flexible Partitions. Having a particular order on, or dependencies between partitions w.r.t. their definition is common for existing partitioning schemes [50], which leads to unreachable configurations in the optimization space. The feature of *flexible partitions* in a GRIDTABLE enables to partition a table in an arbitrary manner by freely defining (non-overlapping) regions in a table. In other words, the partition scheme in a GRIDTABLE does not force to partition horizontally and then vertically first (or vice versa), but allows to define partition regions independent from other existing partitions.

Removing such dependencies and order restrictions from the partitioning scheme enables a higher degree in flexibility, which in turn promises more fine-grained matches of data layout and data placement for the workload on particular regions of a table, which will lead to a better performance if the right configuration in the now broader optimization space is used.

Clearly, having freely floating partitions is not only more flexible but also more complex from both a description and optimization perspective. To limit that complexity, we restrict a GRIDTABLE to *not* have *overlapping regions*. An overlapping region may be understood as a particular portion of data that is redundantly stored on different locations and formatted differently due to different and contradicting access patterns at the same time. We assume that having exactly contradicting access patterns on a significant amount of data where there is no trend to one side of an access pattern type, is a special case for real-world workloads. Therefore, we made the design decision to disallow overlapping regions in order to prune the optimization space that must be explored. Extensions to our solution could consider replication, in future work.

Special to note is the encoding of large *null*-regions for sparse datasets in a GRIDTABLE: if there is no grid defined for a particular subset of row and columns, then this region is interpreted as containing *null*-values only. This tiny definition allows us to zero-out huge regions of *null*-data without reserving any memory for their encoding additionally.

As pointed out by Lemke et al. during their investigation of compression techniques for columnar business intelligence solutions, optimization tasks involving reordering of elements to maximize the desired effect require heuristics to be practical computable within a reasonable time [38]. The authors defined a process consisting of four stages (analysis, candidate determination, heuristic evaluation, and per-candidate application), where four different strategies for range sorting under different assumptions are used. Similar to the problems described by Lemke et al. determining the best partition for a GRIDTABLE is an $\mathcal{NP}$-complete problem and cannot be optimally solved in reasonable time. We explore this and related problems in detail in Sect. 6.

Per-Grid Formats. Per-grid formats enable each partition to organize contained record portions with complete independence from other partitions. Currently, we support uncompressed in-memory column stores and row stores, as well as a binary-search based index. Conceptually further specialized storage strategies can be added, such as compressed column stores for SSDs, or even specialized grid implementations for HDDs or long-term storage devices, such as tapes.

Per-Grid Storage. Per-grid storage enables each partition to be stored on a dedicated memory kind, if required, making the GRIDTABLE an abstract container that splits and delegates queries into grids and collects results from these grids to construct the final reply. We currently support main-memory (host memory) based partitions and partitions that are stored in the co-processors device memory. Along with the flexible partition feature, per-grid formats allow to emulate in a fine-grained manner any major storage layout presented in the literature so far. For instance, HYRISE [23] can be simulated with vertical partitioning only where each partition is either a column store or row store.

However, we are not limited by these types: our abstraction enables to store data on other memory locations that we have not yet explored, such as on SSDs or remote machines.

Data Packing. Data packing is a distinct feature in GRIDTABLES that allows to physically cluster records that are logically spread across the table. With data packing, we are able to move continuous physical memory blocks to co-processors (such as a GPUs) instead of managing several distinct memory blocks only because a user-defined structure forces us to do so. Additionally, we use data packing to decrease the memory requirements implied by organizing the GRIDTABLE structure itself: we pack data from two into one grid if both grids have the same storage location and record format, effectively reducing the number of grids that must be managed by the GRIDTABLE. Further, data packing promises to efficiently manage cold data in the long run: after analysis a GRIDTABLE may pack cold records into one grid and perform (heavy-weight) compression on this grid, or evict the grid data to SSD disks.

Schema-Reordering. Schema-reordering is a feature built for row-store-major GRIDTABLES that involve a huge amount of attributes similar to WIDETABLES but optimized for point-queries rather than range-queries. Having a best-matching ordered attribute schema for row-store records is needed for OLTP queries to optimize execution speed of queries that access a set of fields of a single record (such as the projection operator does). The reasons for an increased execution speed with a reasonable schema order is that a higher data locality of record fields that are accessed together is more cache efficient, and therefore, faster.

Schema-reordering is a per-grid capability to physically rearrange fields of records stored in that partition. The motivation behind this feature is to minimize CPU cache misses for point-queries on same records over a large subset of the records attribute set. A careful re-ordering of record fields in this case promises a higher probability to have the next field already stored in cache: when the majority of queries to that particular grid touches n out of m attributes, these n attributes are moved to the front per-record. Then, seeking between records with providing pre-fetching hints to the CPU raise the probability to have all the next n attributes already in the cache for settings in which each single records size exceeds the cache line size.

5 Organization and Storage

In this section, we focus on engineering and design challenges regarding the GRIDTABLE data structure itself. After establishing the problem statement in Sect. 5.1, we continue with our solutions in the following sections.

5.1 Problem Statement

The purpose of GRIDTABLES is to satisfy our requirements as established in Sect. 3. Namely, the support of data storage strategies optimized for analytical and transactional data access patterns along with a smooth transition between both to optimize for hybrid access patterns. Additionally, the storage engine must be ready for co-processors like GPU or FPGAs, and must expose knobs for autonomous optimization.

In Sect. 4, we depict features for which we argue that they address these requirements. For instance, flexible partitions enable fine-grained and mutable modifications on data placement and data storage strategy that can be driven by access patterns. Zero-cost *null*-value encoding, data packing and schema-reordering allow to optimize WIDETABLE-like GRIDTABLES that result from denormalization in order to optimize analytical query runtime (cf. [39]).

The challenge is to support these features in order to satisfy our requirements in one unified data structure that is both (self-)manageable and reasonable regarding its structural complexity. We classified the storage-related challenges into two groups, (1) the challenge to efficiently organize and maintain a GRIDTABLE and (2) the challenge to support unified data definition and manipulation operations in face of highly flexible partitions.

The problem of self-driven re-evaluation of a layout during runtime, a problem that we call *GridFormation*, is *not* in the scope of this paper. For interested readers, we refer to our other work that explores and investigates GridFormation in a first proposal with reinforcement learning [19, 20].

5.2 The GridTable Data Structure

In this section, we give a detailed description of the ingredients of the GRIDTABLE data structure and how these components relate to each other.

A GRIDTABLE is a type of data store for a relation R with schema $\mathcal{R}$ that segments R into non-overlapping *regions* which can be arbitrarily arranged.

A *region* is defined by two intervals: *tuple cover* and *attribute cover*, a *tuple cover* defines which tuples are contained by their row identifiers[1] (*RID*), and an *attribute cover* defines which subset of $\mathcal{R}$ falls into a particular region.

Unlike other partition schemes, GRIDTABLES allow to define regions in a nonrestrictive way: neither is a particular partitioning order enforced (such as division into subrelations first) nor is it enforced that all regions are de-

[1] A RID is a unique value referencing an entire row in a GRIDTABLE. However, the data type of these identifiers is implementation-dependent and not in focus of this concept.

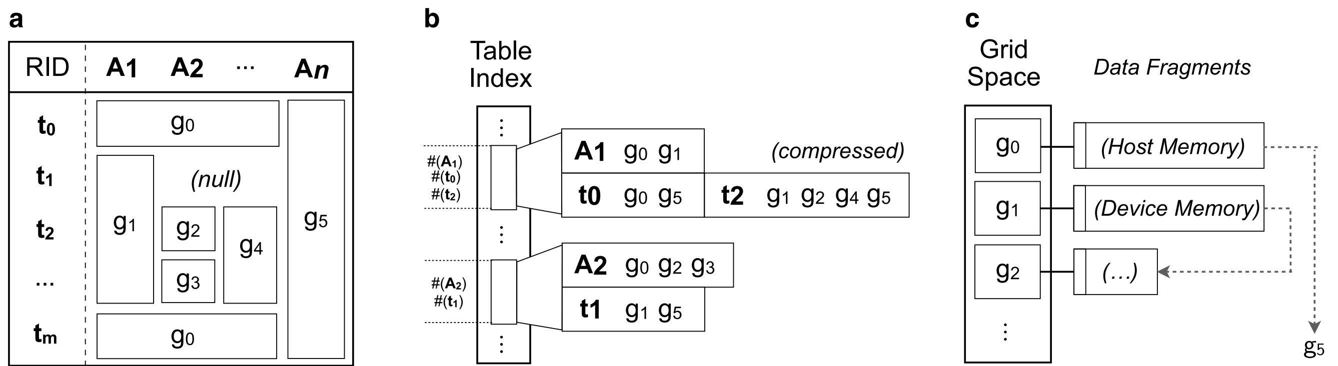

Fig. 4 Views on GRIDTABLE storage, table index, and organization

scribed. Regions can be of one out of two kinds, either *zero-outed* or *managed*.

A *zero-outed region* is a single region interpreted as a (huge) block of *null*-values. This kind is *not* described by a *grid*, i.e., the absence of a grid for a particular region defines that region as a block of *null*-only values. We visualize zero-outed regions with the label *(null)* in Fig. 4a. It's worth to note here, that lossless compression techniques, like run-length encoding, are orthogonal to zeroing-out regions, although such techniques may be used to implement that functionally alternatively.

A *managed region* is a single region interpreted as a block of data (not necessarily non-*null* data), that is owned by a *grid*.

For instance the region owned by grid g_5 covers the attribute A_n, spanning all tuples in Fig. 4a. Multiple regions can be owned by a single grid as long as these regions result from composition of vertical and horizontal partitioning of that grid. For a better understanding, see g_0 in Fig. 4a. The grid g_0 owns tuples t_0 and t_m for all attributes (with the exception of attribute A_n). It is important to note that g_0 is *logically* split into two parts (the regions) but physically g_0 is one unit (which is the basis for the technique we call *data packing*).

GRIDTABLE **Components.** Any GRIDTABLE consists of the logical schema $\mathcal{R}$ of R, its *table index* (TI), a *Grid Space*, and data for book-keeping purposes. The logical schema, that implies no order on attributes, is used to describe R according to its definition in the database. It is used in conjunction with the table index that manages regions in order to locate and instrument (e.g., call particular functions on) grids. Grids are owned by the (memory-resident) Grid Space data structure. The Grid Space (see Fig. 4c) is a dictionary data structure responsible for grid management, and especially the translation between references to grids and their implied strategies that are part of their data fragment.

As the TI in conjunction with the Grid Space act as both an organizing structure and abstraction layer, a decoupling of low-level grid-related details (such as data placement) and implementation-independent management (such as merging of grids) is feasible. One key point here is that the TI allows to poll information and statistics for particular grids and is able to revise a particular data-to-grid mapping having access to the repository of grid implementations. In productive deployments, re-partitioning must not be manual. It raises a set of research challenges for structures as flexible as GRIDTABLES, e.g., when to merge grids, when to undo uch a merge considering implications of the online execution of these operations, or how to effectively refine a chosen partitioning after data ingestion that may have a fixed partitioning policy (such as import all data as a single table-wide row grid). We explore these questions in more detail in Sect. 6, for which techniques such as database cracking [29] might be a good starting point.

Reference translation is a mission-critical operation typically invoked multiple times when multiple regions are touched during queries. Therefore, we suggest to implement a dictionary inside the Grid Space with a data structure that has constant access time (in fact, we use a plain array for that purpose). Book-keeping data ranges from memory usage, read-/write statistics, and capacity information that are used by the GRIDTABLE in order to perform diagnostics, to apply optimization tailored to the read-/write patterns (e.g., transformation to other data fragment types), or for resource management (e.g., freeing up allocated but unused space when space limits are reached).

Data Fragments. Each grid manages its contained tuplets physically in a data container, called *data fragment*.

A data fragment maps the logical schema that partially falls into the region covered by the grid to a physical schema. In addition to the logical schema, a physical schema defines the definitive order of attributes per record. This mapping between logical schema at the table level on the one hand, and the physical schema at the data fragment level on the other hand, allows us to apply a fine-grained *schema-reordering*. Schema-reordering is the capability to physically rearrange tuple fields without interference to the logical schema or other regions in the table that are not man-

aged by the grid at hand. The ultimate benefit of schema-reordering is that it enables adaptability towards request-driven physical order of fields to improve the processors data cache efficiency. More in detail, schema-reordering promises a better cache utilization by smartly ordering fields for record-centric queries on row-wise stored data when a single record exceeds the data cache line size.

In addition to the physical schema, a data fragment maintains a set of book-keeping data structures, and (abstract) operations $Op_1, Op_2, ..., Op_n$ for its *strategy* stored in the *fragment structure*.

For the purpose of this paper, we do not expand on the book-keeping component other than stating that it is mainly about statistics on data access for re-partitioning, and data histograms for query optimization. However, the $Op_1, Op_2, ..., Op_n$ along with the fragment structure are used to provide each data fragment with a specific querying strategy.

A *(row) identifier* is a unique unsigned integer that refers to a record (tuple or tuplet) in a GRIDTABLE. We use the term identifier instead of the term tuple identifier to avoid confusion with the semantics of a tuple identifier in disk-based systems, and to have a common naming for both tuple and tuplets references since they share the same concept of reference. However, in GRIDTABLES there are two kinds of identifiers, *global* and *local*. A *global identifier* identifies a single tuple in the scope of a GRIDTABLE while a *grid-local* identifier identifies a single tuplet in the scope of a grid.

Clearly, the more fine-gained a relation becomes, the higher the cost for book-keeping this information, and the more effort during processing. Hence, the actual partition choice must be bound given some user limits on space consumption and the partitioning impact on query processing performance. We take a deeper look at these challenges in Sect. 6.

5.3 Strategy Abstraction Design

To be extensible towards novel strategies, we intentionally draw the abstraction layer of strategies and data fragments over abstract functions.

Abstract functions fall into the following categories:

1. *Raw Operations.*
2. *Cursor-Based Operations.*
3. *Indirection-Level Bridging.*

Each (query-related) function in (1) operates on a bulk of tuplets to minimize the per-tuplet function-call overhead. Non-query-related operations in (2) involve moving fields cursors and tuplet cursors, per-field reading and writing for the definition and manipulation path (see Sect. 5.4). Query-related operations in (3) are basically used for invocation of full-scan operations and point-query operations over a set of tuplets for the query path.

5.4 Definition and Manipulation Path

This section is about the definition and manipulation path in GRIDTABLES. This path is intended for generic data load, diagnostics and debugging purposes. Directly speaking, querying is done via the query path. This completely bypasses the definition and manipulation path to get rid of the complexity involved with that indirection. For a disk-based system where accessing secondary storage dominates this indirection costs, one can speculate to utilize the definition and manipulation path also for query processing – especially since the query path cannot be implemented for that system kind without major changes. Considering the data definition and manipulation path for disk-based querying is an interesting but yet unexplored application of GRIDTABLES which is out of the scope of this paper.

From a main-memory storage engine perspective, the definition and manipulation path is required and used exactly for the purpose it was designed for: correct definition and manipulation of data stored in an environment that does not guarantee physical order of elements nor shared memory between elements.

Level-Specific Operations. In Sect. 4, we provided a high-level view on the stacked architecture for GRIDTABLES, visualized in Fig. 2. In this section, we show level-specific functions to operate on components in that architecture, and to navigate from one layer to another.

1. *Table Level (TL).* A GRIDTABLE exposes operations to insert, update, remove, and query records abstracting from the table partition. Any request to insert, to update, or to remove tuples is delegated to those grids that own the specific region that should be altered.
2. *Tuple Level (TPL).* Similar to typical tuple-based processing of tuple-at-a-time models, a tuple cursor is opened at table level and used to iterate through all tuples stored in the table. This iteration potentially involves jumping from one grid to another. The logic for these jump operations is transparent to the caller such that the tuple level is abstracted away from the partitioning structure below the tuple level.
3. *Tuplet Level (TTL).* A tuple is already broken down into several tuplets that fall into several grids. A single grid owns portions of several (physical) tuples that may span several regions in the GRIDTABLE. A tuplet is a conceptual abstraction from lower-level stored fields to get rid of low-level data management, i.e., each tuplet consists of a fixed set of fields that can be randomly accessed independent on their actual physical storage. Tuplet fields may be spread across multiple locations but the tuplet

level exposes a unified way to read and write these fields creating the illusion of a dense object.
4. *Raw Data Level (RDL)*. Records are actually physically stored and queried highly dependent on the strategy at hand. The raw data level is responsible for two actions: (1) to provide the functionality defined at the tuplet level in order to hide from low level details (e.g., seeking to a certain memory address), and (2) to provide one or more late-materialization scan flavors to efficiently restrict the GRIDTABLES content given a user-defined predicate.

We explicitly note here, that some major aspects (such as efficient primary key uniqueness checks, recovery and failover, concurrency issues and transaction control, or cache coherence and latency management for data on co-processors) are not discussed or only slightly touched in this paper. The reason for this is our strong focus on the table data structure in isolation, such that we have to defer this required discussion to future work.

6 Open Challenges

At the point of this writing, GRIDTABLES are a novel concept to enable a unified storage engine in the huge design space of an H^2TAP database system.

The core question is *how to instrument* GRIDTABLES capabilities in a way that the system itself smartly and autonomously tunes *multiple* knobs *at once* to calibrate itself to the best possible performance in one instant in time.

In order to answer the question about self-driven instruction of partitioning schemes as flexible as GRIDTABLES, we formulate the following eight open research challenges that can be researched in isolation, which are, therefore, given here without any particular order.

Record Organization Problem. Given a GRIDTABLE R, a set $\mathcal{Q}$ of n queries on R and a cost function f that determines the costs to access fields in R in order to answer the query set $\mathcal{Q}$.

The problem is to find a layout $L(R)$ such that f is minimal for all queries in $\mathcal{Q}$ at once.

This problem cannot be solved efficiently in its optimal version in a feasible amount of time. However, a good solution to this problem enables to autonomously determine a suitable layout for one particular time span in which $\mathcal{Q}$ is issued (cf. [4, 9, 23] for work towards this direction).

Data Placement Problem. Given a GRIDTABLE R, an update-ratio α, a workload by a set Q of queries having a portion of α update operations contained, a cost function f_Q that determines the costs of accessing fields in R for Q, and a cost function f_{up} that determines the costs for updates on the device memory.

The problem is to find a layout $L(R)$ for R such that f_Q is minimal for all queries in Q at once, and that minimizes the f_{up} for those data fragments that are stored in the device memory for varying α.

Data Fragments can be placed in a device memory (e.g., the co-processors device memory) and processed by that device. The structure of GRIDTABLE enables a fine-grained data placement of tuplets in the device, e.g., multiple parts of a single column, disjunct regions of multiple columns, or particular blocks of data.

In case of a read-only workload ($\alpha \equiv 0$), the data placement problem is the Record Organization Problem. In case of any non-read-only workload ($\alpha > 0$), selecting the device as storage- and processing-place for some data only yields higher performance if the cost penalty for update propagation to the device is low. Whether this penalty is low or not (even for write-only workloads with $\alpha \equiv 1$), depends whether the selected data is target of updates in Q or not.

A reasonable solution to the problem is to minimize the surface of data in R stored in the device that is updated by Q but to maximize the surface of data in R not modified by Q stored in the device to increase the processing performance.

Transition Cost Problem. Given a GRIDTABLE R, and a layout $L_0(R)$ for R that is a solution of the Record Organization Problem for one particular time instant t_0, and two time instants t_1, t_2 with $t_2 > t_1 > t_0$.

The problem is to determine (or forecast) layouts $L_1(R)$ and $L_2(R)$ as a solution of the Record Organization Problem for t_1 resp. t_2, to compute the transition costs $c_{0\to1}$ for a transition from $L_0(R)$ to $L_1(R), c_{0\to2}$ for a transition $L_0(R)$ to $L_2(R)$, and $c_{1\to2}$ for a transition $L_1(R)$ to $L_2(R)$, and to decide at a time instant $t_* \in (t_0, t_2]$ whether to change from $L_0(R)$ to $L_1(R)$, or to change from $L_0(R)$ to $L_2(R)$ considering $c_{0\to1}, c_{0\to2}$, and $c_{1\to2}$, or to perform no change at all.

In simpler words, this *online* problem describes the decision act with which the system performs a change in the layout towards a more suitable layout. The interesting challenge in this problem is that the optimal layout changes over time (due to changes in the workload), and that the benefit of a transition might be hidden by the costs it implies. These costs may contain time considerations for copying and re-formatting actions of data in memory, costs for data movement operations between devices, and more.

Roughly speaking, staying too long on one particular layout or being too slow in layout adaption leaves performance opportunities untouched. At the same time, too aggressive changes will lead to sub-optimal performance compared to moderate or to slow changes due to transition costs.

A suitable solution to this problem must balance the trade-off such that more suitable layouts are adapted as fast as possible, while – at the same time – the number of sub-

optimal performance runs (due to transition costs) must be minimized.

Read Set Labeling Problem. Given a workload $\mathcal{W}$ consisting of N queries with a particular ratio α of transactional and analytical operations where α is unknown to the system.

The problem is to find and optimally classify regions in the read set W *(i.e., the fields accessed for reads or writes) in order to mark them as attribute-centric or row-centric operation regions.*

Obviously, this is not a trivial problem efficiently to solve in an optimal way since PARTITION is already $\mathcal{NP}$-complete. However, having this classification promising a good hint for a layout optimizer which then immediately is able to compare current regions in a GRIDTABLE matching the regions in the read set.

Wide-Partitioning Problem. Given m tables $R = \{R_1, R_2, ..., R_n\}$, and a series of m read/write queries $Q_1, Q_2, ..., Q_m$ on these n tables with ratio r, a cost function f_Q that determines the costs to access fields in R to answer Q, a cost function f_σ that determines the costs to access fields for a rewritten (scan) query σ for Q on $R_1 \bowtie R_2 \bowtie ... \bowtie R_n$, a cost function $f_\bowtie$ that determines the costs to construct a WideTable $R^* = (R_1 \bowtie R_2 \bowtie ... \bowtie R_n)$, and a cost function $m_Q(X)$ that determine maintenance costs to update table X if Q manipulates data in X.

The problem is to find $i_0 \in \{1, 2, ..., m\}$ *such that*

$$f_\bowtie(R) + \sum_{i=i_0}^{m} f_{\sigma_i}(R^*) + m_{\sigma_i}(R^*) < \sum_{i=i_0}^{m} f_{Q_i}(R) + m_{\sigma_i}(R)$$

(if such an i_0 *exists)*

Informally, the problem is to determine a particular threshold i_0 in time during processing of $Q_1, Q_2, ..., Q_m$ for which it is cheaper to take the effort in constructing R^* once, and then continue with WideTable scans compared to straightforward execution $Q_{i_0+1}, ..., Q_m$ (by also considering update costs after that threshold).

As shown by Bian et al. in their work on WIDE TABLES, rewriting a query Q on R to σ on R^* yields excellent performance improvements for pure analytical processing [13]. In context of H^2TAP this technique therefore promises an excellent performance gain for the analytical part of queries. However, naive adaptions of WideTables is likewise a bottleneck due to memory limitations to hold the denormalized table R^*. Additionally, since H^2TAP inherently implies additional *data writes*, update costs of R^* compared to (potentially normalized) tables in R must be taken into account. Finally, H^2TAP systems are online systems rather than pure analytical offline systems. Therefore, a solution i_0 once found, is immediately target for being re-evaluated once time passes m. Perhaps, a decomposition of R^* back into R will be the better option then.

Attribute Ordering Problem. Given a grid G in a layout $L(R)$ of a GRIDTABLE R, and m sets of queries $Q_1, Q_2, ..., Q_m$ for G at time $t_1, t_2, ..., t_m$ ($t_1 < t_2 < ... < t_m$), a function f_{miss} that determines the number of (processor data-) cache-misses for a query set Q in G given a fixed (physical) order of tuplet fields in G defined by the order of attributes $A_1, A_2, ..., A_n$ in the schema of G.

The problem is to find a sequence of permutations $(\sigma(A_1), \sigma(A_2), ..., \sigma(A_n))_i$ *for* $i = 1, 2, ..., m$ *to physically re-order tuplet fields in* G *such that* f_{miss} *is minimal for* $t_1, t_2, ..., t_m$ *with* $(\sigma(A_1), \sigma(A_2), ..., \sigma(A_n))_k$ *is used at time* t_k *for query set* Q_k *(*$1 \leq k \leq m$*).*

The interesting aspect of this problem is that queries in a query set Q are neither required to read/write a particular subset of tuplets in G, of tuplet fields or of fields in common in a particular order. Consequently, this problem ranges from trivial configurations (such as entire Q reads all tuplets fields of all tuplets in natural order) to contradicting configurations (such as the first half of Q reads all tuplet fields of all tuplets in natural order, while the second half of Q does the same but in inverse natural order).

Finding an optimal solution $\sigma(A_1), \sigma(A_2), ..., \sigma(A_n)$ for a given Q is challenging, especially for an *online* sequence of queries as given in the problem statement.

Null-Region Maximization Problem. Let R be a a sparse GRIDTABLE, $L(R)$ a layout for R, Q a set of queries, f a cost function that determines the costs for accessing fields in R, and ε a small threshold from the domain of costs.

The problem is to re-order tuplets in R *and to re-order attributes in the schema of each grid in* $L(R)$ *such that for the new layout* $L^*(R)$ *holds:* $L^*(R)$ *maximizes the regions that contains null-only values (e.g., by minimizing the number of null-only regions), and the costs for* Q *using* f *in* $L(R)$ *are the same as the costs for* $L^*(R) \pm \varepsilon$.

A region in a GRIDTABLE R that completely covers a *null*-value block of data, does not require additional space for encoding these *null*-values (cf. Sect. 5.2). This potentially saves space in very sparse data sets. Given the way how regions and grids are managed in a GRIDTABLE, the most memory efficient configuration is a small number of regions that are *null*-value data data only, but each of these regions cover a maximum number of values.

Compression Problem. Given a layout $L(R)$ of a GRIDTABLE R with k grids $G_1, G_2, ...G_k$, a set C of n compression techniques $C = \{c_1, c_2, ..., c_n\}$, a set Q of queries $Q = \{Q_1, Q_2, ..., Q_m\}$ with a query performance of p, and a user-defined lower bound $\tau < p$ that sets the least acceptable query performance.

The problem is to determine a candidate set $X \subseteq C$ and $j = 1, ..., k$ permutation $\pi^j : \{1, 2, ..., n\} \rightarrow \{1, 2, ..., n\}$ such that for each $i \in \{1, 2, ..., k\}$ the per-grid compression $(c^i_{\pi(n)} \circ \cdots \circ c^i_{\pi(2)} \circ c^i_{\pi(1)})(G_i)$ minimizes the space requirements for the entire layout $L(R)$ while the query performance for Q must not drop below the threshold τ.

In the challenging aspect of this problem is not only the determination of the $i = 1, ..., k$ permutations π_i to determine the order in which a particular set of compression techniques are applied one after the another to compress a particular grid, which by its own is a computational expensive problem, but that this decision must run for k grid concurrently while there is a (potential) variety of access pattern in Q and queries in Q must not access all grids in $L(R)$ equally.

In sum, a unified architecture promises the best of all worlds. For instance, the synergy of the compression problem and the attribute ordering problem promises a better solution than both in isolation (cf. [38]). To fulfill the promise of a truly unified architecture, our stated open research challenges must be continuously solved at once during runtime, which is a challenging task.

7 Related Work

The field of adaptive data stores is a hot research topic with a series of novel approaches, such as the popular *database cracking* [22, 24, 28, 29], its variations and analysis [30, 58–60], advanced partitioning [32, 44, 61] or adaptive resp. holistic indexing [5, 47, 57]. Latest research is done on navigation through the entire data structure design space, and systems adapting to workload and hardware by using machine learning [20, 27, 31], or Just-In-Time data structures as proposed by Kennedy et al. [35]. On the other side of the spectrum, there are also advanced techniques operating on fixed data layouts, such as PAX [2] or Fractured Mirrors [54].

An academic database system that pioneers a notable amount of H^2TAP features for the relational model is HYPER [34]. Originally motivated by the challenge to engineering an H^2TAP system with competitive performance to pure operational and pure analytical systems by using the UNIX's `fork` system call, its storage engine nowadays supports combined horizontal and vertical partitioning including advanced compression of cold data [37]. However, this is in contrast to the partition technique in GRIDTABLES: while HYPER forces vertical partitioning to a relation first, in our approach it is up to the system whether to start first with horizontal partitioning, or vertical partitioning instead.

A young system is L-STORE, a main-memory H^2TAP database system that supports historic queries [55]. L-STORE is powered by a storage engine that performs physical reformatting of tuples on-demand. For this, the primary data container incorporates multiple base pages and tail pages that are used to form an actual tuple. A relation is managed by sub-relations such that each attribute of a table is mapped to one vertical fragment. Although GRIDSTORE does not support time-travel (historic) queries in the sense of L-STORE, the flexibility of GRIDSTORE allows to mimic partitioning to pure-vertical fragments.

Another direction is taken for the development of the database system PELOTON [9]: its storage engine is built from ground up to support a novel tile-based architecture that manages tables in terms of tile groups. Each such group is a horizontal fragment which may be further vertically partitioned into (inner) partitions called logical tiles. The partition schema of PELOTON is more restrictive than the one we present in this paper, but shares important ideas such as the autonomous self-adaption of the layout depending on workload optimization. One special feature of PELOTON is its ability to forecast changes in the workload and to trigger adaption proactively [40]. At this point of this writing, GRIDTABLES do *not* support the orthogonal feature of forecasting, or adopting learned optimization models, but we are researching in this direction [19].

An adaptive storage engine veteran is HYRISE [23], which organizes a relation by n sub-relations, called containers. Each container holds a certain amount of attributes: when a container incorporates exactly one attribute, the sub-relation becomes de facto a columnar format. HYRISE allows both formats for records columnar and row-wise. This storage engine automatically changes the number of attributes particular containers own in order to improve cache efficiency in face of changing workloads. Similar, the H_2O [4] storage engine manages both, columnar and row-wise formatted partitions for a single table following a strict horizontal partitioning similar to HYRISE. H_2O applies changes in that the partitioning is done in a lazy fashion when compared to HYRISE by applying a new partitioning schema after careful evaluation in the background. GRIDTABLES and both, HYRISE and H_2O share the required idea of autonomous adaption of partitions without manual tuning by a human administrator. However, the space of potential partitions for a single table in GRIDTABLES is far larger compared to these approaches since GRIDTABLES allows for an arbitrary order of horizontal and vertical partitioning.

8 Conclusion

In this paper, we propose a novel concept to manage records for H^2TAP database systems, called GRIDTABLES. We showed how mixed workloads affect the query performance. Then, we stated requirements for an H^2TAP

store, and showed our proposal of a *stacked architecture* built on a set of well-engineered indirection levels for secure, safe and well-defined data access in face of arbitrary data placement and formatting. Based on our concept for a One-Size-Fits-Most architecture, we explored our list of formally defined *open research challenges* that focus on automatic instrumentation of GRIDTABLE features: (i) *Read Set Labeling* to label workload parts as analytical resp. transactional, (ii) *Record Organization* to find layout for table to optimize for read set, (iii) *Wide-Partitioning* to decide on (de-)normalize action for infinite horizon (iv) *Data Placement* to find optimal placement of data, (v) *Attribute Ordering* to find optimal order of attributes, (vi) *Null Maximization* to find maximum regions for *null*-data, and (vii) *Transition Costs* to approximate data moving & partitioning action costs, and (viii) *Compression* to compress grids individually while not sacrificing performance. To fulfill the promise of best performances, we motivated for further investigation these eight open research challenges for storage structures as flexible as GRIDTABLES.

Acknowledgements This work was partially funded by the German Research Foundation (DFG grant no.: SA 465/50-1 and SA 465/51-1), and the de.NBI Network (grant no.: 031L0103). Thanks to Mahmoud Mohsen, Iya Arefyeva, Anusha Janardhana Rao, Andreas Meister, and Thomas Leich.

Funding Open Access funding provided by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Abadi DJ, Madden SR, Hachem N (2008) Column-stores vs. row-stores: how different are they really? In: ACM SIGMOD SIGMOD'08, Vancouver, June 9–12, 2008, pp 967–980
2. Ailamaki A, DeWitt DJ, Hill MD (2002) Data page layouts for relational databases on deep memory hierarchies. VLDB J 11(3):198–215
3. Ailamaki A, Liarou E, Tözün P, Porobic D, Psaroudakis I (2014) How to stop under-utilization and love multicores. In: IEEE International Conference on Data Engineering ICDE 2014, Chicago, March 31–April 4, 2014, pp 1530–1533
4. Alagiannis I, Idreos S, Ailamaki A (2014) H2O: a hands-free adaptive store. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data SIGMOD/PODS'14, Snowbird, 2014 Association for Computing Machinery, New York, pp 1103–1114
5. Alvarez V, Schuhknecht FM, Dittrich J, Richter S (2014) Main memory adaptive indexing for multi-core systems. In: Proceedings of the Tenth International Workshop on Data Management on New Hardware SIGMOD/PODS'14: International Conference on Management of Data, Snowbird, June, 2014 ACM, New York, p 3
6. Appuswamy R, Karpathiotakis M, Porobic D, Ailamaki A (2017) The case for heterogeneous HTAP. In: 8th Biennial Conference on Innovative Data Systems Research 8th Biennial Conference on Innovative Data Systems Research, Chaminade, January 8–11, 2017 (EPFL-CONF-224447)
7. Arefyeva I, Broneske D, Pinnecke M, Bhatnagar M, Saake G (2017) Column vs. row stores for data manipulation in hardware oblivious CPU/GPU database systems. In: GvDB Workshop 29th Workshop on Foundations of Databases, Blankenburg, 30.5.–2.6.2017, pp 24–29
8. Arefyeva I, Broneske D, Durand GC, Pinnecke M, Saake G (2018) Memory management strategies in CPU/GPU database systems: a survey. In: BDAS 2018 14th International Conference, BDAS 2018, Held at the 24th IFIP World Computer Congress, WCC 2018, Poznan, September 18–20, 2018
9. Arulraj J, Pavlo A, Menon P (2016) Bridging the archipelago between row-stores and column-stores for hybrid workloads. In: SIGMOD'16 SIGMOD'16, San Francisco, June 26–July 01, 2016, pp 583–598
10. Athanassoulis M, Kester M, Maas L, Stoica R, Idreos S, Ailamaki A, Callaghan M (2016) Designing access methods: the RUM conjecture. In: Proc. 19th International Conference on Extending Database Technology (EDBT) EDBT, Bordeaux, March 15–18, 2016
11. Bača R, Krátký M, Holubová I, Nečaský M, Skopal T, Svoboda M, Sakr S (2017) Structural XML Query Processing. ACM Computing Surveys (CSUR) 64:1–41
12. Becher A, Lekshmi B, Broneske D, Drewes T, Gurumurthy B, Meyer-Wegener K, Pionteck T, Saake G, Teich J, Wildermann S (2018) Integration of FPGas in database management systems: challenges and opportunities. Datenbank Spektrum 18(3):145–156
13. Bian H, Yan Y, Tao W, Chen LJ, Chen Y, Du X, Moscibroda T (2017) Big wide table layout optimization based on column ordering and duplication. In: SIGMOD '17: Proceedings of the 2017 ACM International Conference on Management of Data SIGMOD/PODS'17: International Conference on Management of Data, Chicago, May, 2017 ACM, New York, pp 299–314
14. Breß S, Beier F, Rauhe H, Sattler KU, Schallehn E, Saake G (2013) Efficient co-processor utilization in database query processing. Inf Syst 38(8):1084–1096
15. Breß S, Köcher B, Funke H, Zeuch S, Rabl T, Markl V (2018) Generating custom code for efficient query execution on heterogeneous processors. VLDB J 27(6):797–822
16. Broneske D, Köppen V, Saake G, Schäler M (2018) Efficient evaluation of multi-column selection predicates in main-memory. IEEE Trans Knowl Data Eng. https://doi.org/10.1109/TKDE.2018.2825349
17. Chaudhuri S, Narasayya V (2007) Self-tuning database systems: a decade of progress. In: VLDB '07 VLDB '07, Vienna, September 23–28, 2007, pp 3–14
18. Durand GC, Pinnecke M, Broneske D, Saake G (2017) Backlogs and interval timestamps: building blocks for supporting temporal queries in graph databases. In: EDBT/ICDT
19. Durand GC, Pinnecke M, Piriyev R, Mohsen M, Broneske D, Saake G, Sekeran M, Rodriguez F, Balami L (2018) Gridformation: towards self-driven online data partitioning using reinforcement learning. In: aiDM'18: Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management SIGMOD/PODS '18: International Conference on Management of Data, Houston, June, 2018 ACM, New York, pp 1–7

20. Durand GC, Piriyev R, Pinnecke M, Broneske D, Gurumurthy B, Saake G (2019) Automated vertical partitioning with deep reinforcement learning. In: European Conference on Advances in Databases and Information Systems ADBIS 2019, Bled, September 8–11, 2019 Springer, Cham, pp 126–134
21. Garcia-Molina H, Salem K (1992) Main memory database systems: an overview. IEEETrans Knowl Data Eng. https://doi.org/10.1109/69.180602
22. Graefe G, Halim F, Idreos S, Kuno H, Manegold S, Seeger B (2014) Transactional support for adaptive indexing. VLDB J 23(2):303–328
23. Grund M, Krüger J, Plattner H, Zeier A, Cudre-Mauroux P, Madden S (2010) HYRISE: a main memory hybrid storage engine. Proc VLDB Endow https://doi.org/10.14778/1921071.1921077
24. Halim F, Idreos S, Karras P, Yap RH (2012) Stochastic database cracking: towards robust adaptive indexing in main-memory column-stores. Proc VLDB Endow 5(6):502–513
25. He B, Yu JX (2011) High-throughput transaction executions on graphics processors. Proc VLDB Endow. https://doi.org/10.14778/1952376.1952381
26. Heimel M, Kiefer M, Markl V (2015) Self-tuning, GPU-accelerated kernel density models for multidimensional selectivity estimation. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp 1477–1492. https://doi.org/10.1145/2723372.2749438
27. Idreos S, Dayan N, Qin W, Akmanalp M, Hilgard S, Ross A, Lennon J, Jain V, Gupta H, Li D et al (2019) Design continuums and the path toward self-designing key-value stores that know and learn. In: Biennial Conference on Innovative Data Systems Research (CIDR 2019)
28. Idreos S, Kersten ML, Manegold S (2007) Updating a cracked database. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data SIGMOD ’07, pp 413–424
29. Idreos S, Kersten ML, Manegold S et al (2007) Database cracking. CIDR 7:68–78
30. Idreos S, Papaemmanouil O, Chaudhuri S (2015) Overview of data exploration techniques. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM, New York, pp 277–281
31. Idreos S, Zoumpatianos K, Hentschel B, Kester MS, Guo D (2018) The data calculator: data structure design and cost synthesis from first principles and learned cost models. In: Proceedings of the 2018 International Conference on Management of Data. ACM, New York, pp 535–550
32. Jindal A, Dittrich J (2011) Relax and let the database do the partitioning online. In: International Workshop on Business Intelligence for the Real-Time Enterprise. Springer, Berlin, Heidelberg, pp 65–80
33. Karnagel T, Habich D (2017) Heterogeneous placement optimization for database query processing. It Inf Technol:117–123. https://doi.org/10.1515/itit-2016-0048
34. Kemper A, Neumann T (2011) HyPer: a hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In: ICDE, pp 195–206
35. Kennedy O, Ziarek L (2015) Just-in-time data structures. In: CIDR
36. Kraska T, Beutel A, Chi EH, Dean J, Polyzotis N (2017) The case for learned index structures. In: CoRR, pp 489–504
37. Lang H, Mühlbauer T, Funke F, Boncz PA, Neumann T, Kemper A (2016) Data blocks: hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In: ACM SIGMOD, pp 311–326
38. Lemke C, Sattler KU, Färber F (2009) Kompressionstechniken für spaltenorientierte BI-Accelerator-Lösungen. In: Datenbanksysteme in Business, Technologie und Web (BTW)–13. Fachtagung des GI-Fachbereichs “Datenbanken und Informationssysteme” (DBIS)
39. Li Y, Patel JM (2014) Widetable: an accelerator for analytical data processing. In: VLDB, pp 907–918
40. Ma L, Aken DV, Hefny A, Mezerhane G, Pavlo A, Gordon GJ (2018) Query-based workload forecasting for self-driving database management systems. In: ACM SIGMOD, pp 631–645
41. Meister A, Breß S, Saake G (2015) Toward GPU-accelerated database optimization. Datenbank Spektrum. https://doi.org/10.1007/s13222-015-0184-3
42. Mueller R, Teubner J, Alonso G (2009) Streams on wires: a query compiler for FPGas. Proc VLDB Endow 2(1):229–240
43. Neumann T, Radke B (2018) Adaptive optimization of very large join queries. In: Proceedings of the 2018 International Conference on Management of Data. ACM, New York, pp 677–692
44. Olma M, Karpathiotakis M, Alagiannis I, Athanassoulis M, Ailamaki A (2017) Slalom: coasting through raw data via adaptive partitioning and indexing. Proc VLDB Endow 10(10):1106–1117
45. Özcan F, Tian Y, Tözün P (2017) Hybrid transactional/analytical processing: a survey. In: SIGMOD, pp 1771–1775
46. Pavlo A, Angulo G, Arulraj J, Lin H, Lin J, Ma L, Menon P, Mowry T, Perron M, Quah I, Santurkar S, Tomasic A, Toor S, Aken DV, Wang Z, Wu Y, Xian R, Zhang T (2017) Self-driving database management systems. In: CIDR
47. Petraki E, Idreos S, Manegold S (2015) Holistic indexing in main-memory column-stores. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM, New York, pp 1153–1166
48. Pezzini M, Feinberg D, Rayner N, Edjlali R (2014) Hybrid transaction/analytical processing will foster opportunities for dramatic business innovation. Gartner, https://www.gartner.com/en/documents/2657815/hybrid-transaction-analytical-processing-will-foster-opp
49. Pinnecke M, Broneske D, Saake G (2015) Toward GPU accelerated data stream processing. In: GvDB Workshop, pp 78–83
50. Pinnecke M, Broneske D, Durand GC, Saake G (2017) Are databases fit for hybrid workloads on GPus? A storage engine’s perspective. In: HardBD/ICDE, pp 1599–1606
51. Pinnecke M, Campero Durand G, Zoun R, Broneske D, Saake G (2019) Protobase: it’s about time for backend/database co-design. In: BTW. Gesellschaft für Informatik, Bonn
52. Pinnecke M, Hoßbach B (2015) Query optimization in heterogenous event processing federations. Datenbank Spektrum 15(3):193–202
53. Pohl C, Sattler KU (2018) Joins in a heterogeneous memory hierarchy: exploiting high-bandwidth memory. In: Proceedings of the 14th International Workshop on Data Management on New Hardware. ACM, New York, p 8
54. Ramamurthy R, DeWitt DJ, Su Q (2003) A case for fractured mirrors. VLDB J 12:89–101
55. Sadoghi M, Bhattacherjee S, Bhattacharjee B, Canim M (2016) L-store: a real-time OLTP and OLAP system. In: CoRR, pp 540–551
56. Sartakov V, Weichbrodt N, Krieter S, Leich T, Kapitza R (2018) STANlite – a database engine for secure data processing at rack-scale level. In: 2018 IEEE International Conference on Cloud Engineering (IC2E). IEEE, Piscataway, pp 23–33
57. Schuh S, Dittrich J (2015) AIR: adaptive index replacement in Hadoop. In: 2015 31st IEEE International Conference on Data Engineering Workshops. IEEE, Piscataway, pp 22–29
58. Schuhknecht FM, Dittrich J, Linden L (2018) Adaptive adaptive indexing. In: 2018 IEEE 34th International Conference on Data Engineering (ICDE). IEEE, Piscataway, pp 665–676
59. Schuhknecht FM, Jindal A, Dittrich J (2013) The uncracked pieces in database cracking. Proc VLDB Endow 7(2):97–108
60. Schuhknecht FM, Jindal A, Dittrich J (2016) An experimental evaluation and analysis of database cracking. VLDB J 25(1):27–52

61. Schuhknecht FM, Khanchandani P, Dittrich J (2015) On the surprising difficulty of simple things: the case of radix partitioning. Proc VLDB Endow 8(9):934–937
62. Seidemann M, Seeger B (2019) ChronicleDB: a high-performance event store. ACM Trans Database Syst 44(4):1–45
63. Zhang K, Wang K, Yuan Y, Guo L, Li R, Zhang X, He B, Hu J, Hua B (2017) A distributed in-memory key-value store system on heterogeneous CPU-GPU cluster. In: VLDB, pp 729–750