Are Databases Fit for Hybrid Workloads on GPUs?
A Storage Engine's Perspective

Marcus Pinnecke, David Broneske, Gabriel Campero Durand[1] and Gunter Saake
University of Magdeburg
Email: {firstname.lastname}@ovgu.de & [1]campero@ovgu.de

***Abstract*—Employing special-purpose processors (e.g., GPUs) in database systems has been studied throughout the last decade. Research on heterogeneous database systems that use both general- and special-purpose processors has addressed either transaction- or analytic processing, but not the combination of them. Support for hybrid transaction- and analytic processing (HTAP) has been studied exclusively for CPU-only systems. In this paper we ask the question *whether current systems are ready for HTAP workload management with cooperating general- and special-purpose processors*. For this, we take the perspective of the backbone of database systems: the storage engine. We propose a unified terminology and a comprehensive taxonomy to compare state-of-the-art engines from both domains. We show similarities and differences, and determine a necessary set of features for engines supporting HTAP workload on CPUs and GPUs. Answering our research question, our findings yield a resolute: not yet.**

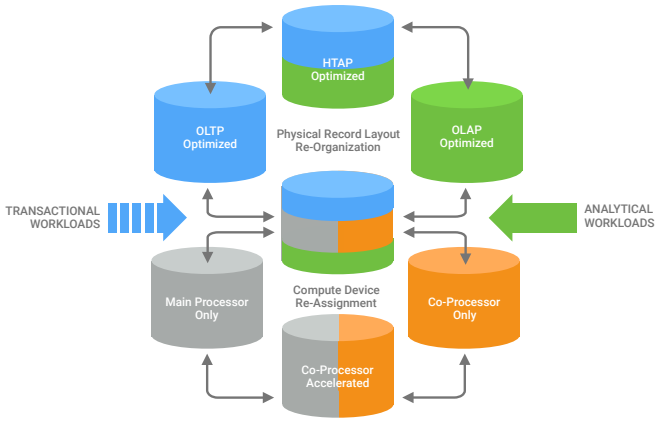

Fig. 1. Physical record layout re-organization and compute device re-assignment in database systems that manage HTAP workloads efficiently.

I. Introduction

Two challenges are being set today for database systems: continuous *physical record layout organization* and continuous *compute device assignment* in the face of mixed workload types (cf. Figure 1). On the one hand, database systems need to combine simultaneous support for analytical *and* transactional processing [1], [2], [3], [4]. Merging both processing types into one single system promises a larger business value by minimizing analytic latency and data synchronization effort [5]. On the other hand, database systems must make an optimal use of a wide range of heterogeneous processors types, such as *Graphics Processing Units* (GPUs), *Multiple Integrated Cores* (MICs), or *Field Programmable Gate Arrays* (FPGAs). Building on these heterogeneous compute platforms is necessary to overcome limitations such as the *power wall* [6]. The research on heterogeneous systems introduces design considerations into single-machine system architectures [7], [8], [9], [10], [11] that has similarities to distributed computing [12] and federated systems [13], [14]. These design considerations are driven by the following challenges: (a.i) *expensive data transfer* to and from the device memory, (a.ii) *different memory types* per compute platform, and (a.iii) *strict limitations* regarding the device memory capacity. Consequently, heterogeneous systems demand special *locality-aware* approaches able to support *column-based* placement of certain data stored in a relation [7], [10], and tailored strategies for data placement to avoid degeneration of query performance by *cache thrashing* and other side-effects during query processing [15], [16]. Database systems supporting *Hybrid Transactional/Analytical Processing workloads (HTAP)* [5] also demand special design considerations. HTAP database systems, such as HyPer [1], Peloton [2], and SAP HANA [17], address particular challenges implied by the hybridization of both analytical and transactional workload processing into one system. These challenges are: (b.i) *different data access patterns* implied by different workload types, (b.ii), *continuous physical optimization* in consideration of contradicting optimization goals, and (b.iii) *efficient processing of both workload types* without interferences between long-running ad-hoc analytic queries and massive short-living write-intensive transactional queries. Consequently, HTAP-workload systems demand special concepts for *physical storage layout handling* [18] including the capability to adapt to changes in the workload during runtime [2], [3], [19] and advanced techniques to detach analytic query execution from mission-critical transactional data [1], [20].

A storage engine is highly tailored to challenges that a database system faces and is fundamental for the entire system. In this paper we argue that currently proposed design decisions to face these challenges (a.i – iii & b.i – iii) might be complementary to each other, especially when considered from the perspective of a storage engine. We proceed with our paper as follows: We first provide background to the field of physical record organization including our experimental findings (Section II). We then contribute the following to bridge the gap between the design solutions from both fields:

- A novel storage engine design taxonomy (Section III).
- A survey and classification of state-of-the-art systems from both fields (Sections IV-A and IV-B).
- An identification of characteristics for HTAP workloads on CPU / GPU systems (Section IV-C).

While several approaches exist for supporting HTAP workloads in CPU DBMSs and for using GPUs as database co-processors, we've found that they are being treated as independent from each other. There are no uniform concepts that allow to compare the advanced design choices tailoring storage engines for both types of approaches. We end this paper with our summary in Section V.

II. Background

For a better understanding of our paper, we first introduce essential background on storage models to cover challenges regarding physical record layouts. Afterwards, we present our quantitative evaluation showing performance effects of contradicting optimizations (storage model, threading policy, and compute platform) within HTAP database systems on heterogeneous compute platforms.

A. Classic Physical Record Organization for OLTP & OLAP

Database systems that implement the relational model (e.g., Ingres [21] or System R/DB2 [22], [23] to name the earliest) are based on a physical manifestation of the concept of *relations* as suggested by Codd [24]. However, due to the 2-dimensional concept of a relation, the content has to be serialized to a format that can be stored in a linear stream of memory. The serialization of a relation encompasses the serialization of meta data and records. In fact, the way in which a relation is serialized and accessed determines the CPU cache utilization; as a result, serialization and access patterns are of special importance for optimizing query performance in hybrid-workload systems [25].

The data in a relation can be serialized following an *N-ary storage model*[1] *(NSM)* [26] or a *Decomposed storage model (DSM)* [18]. In NSM, data is formatted as a sequence of records, i.e., all fields of a record r_x are stored sequentially before the process is repeated with the successor r_{x+1}. NSM is the foundation of row-oriented storage engines. In contrast in DSM, data is formatted as a sequence of columns, i.e., all fields of a certain column c_x are stored in a sequence, before this process is repeated with the next column c_{x+1}. DSM is the foundation of column-oriented storage engines. Data inside a relation R can be formatted following a certain physical record layout, i.e., NSM or DSM. The physical record layout satisfies the question on *how* data is stored. Another question is, *where* the data is stored, e.g., on main-memory or on hard drive. Whether NSM or DSM is the more suitable format to store data in R depends on *how* the data in R *is accessed* rather than where it is actually stored [2].

Historically, NSM was the first format employed for (transactional) relational databases, because the main application areas for database systems (e.g., communication, finance, travel, manufacturing, and process control [27]) had a *record-centric* data access pattern: each read / update operation in a transaction accesses a *small* subset of the records of a relation, and it also accesses a *large* subset of fields per record. For a better understanding, consider the following query Q_1:

$$Q_1 : \textbf{SELECT } * \textbf{ FROM } R \textbf{ WHERE } pk = c;$$

The query Q_1 asks for all fields of all records in a relation R whose field pk equals a certain constant value c. Assuming the attribute pk is a (non-compound) primary key, the database system can efficiently identify exactly *one* record without scanning the entire relation. Once the record is found, *all* fields are materialized for the result. This extreme case is an example of a record-centric data access pattern. NSM combined with the *Volcano*-style processing model suits well for this access pattern in case the costs for function calls can be hidden by data access costs. More specifically, NSM works well for disk-based systems, but has limited CPU data cache efficiency for main-memory systems [28], [29].

In contrast, DSM is utilized for database systems which are issued with an *attribute-centric* data access pattern: operations access a *large* subset of relation's records, and a *small to tiny* subset of fields per record. For a better understanding, consider the following query Q_2:

$$Q_2 : \textbf{SELECT } sum(a) \textbf{ FROM } R;$$

The query Q_2 asks for the sum of all record values regarding the attribute a of a relation R. Typically, the database system runs an aggregation by accessing *all* records in R considering exactly *one* attribute (i.e., all values for a). This extreme case is an example of an attribute-centric data access pattern. DSM is typically employed in analytic processing systems where mostly aggregations and groupings are executed on read-only data, while benefiting from late materialization and improved compression rates [30]. DSM combined with a *Bulk*-style processing model is a good match for analytic processing in main-memory databases due to improved CPU data cache efficiency [29], [31].

B. Contradicting Optimization Goals within HTAP Workloads

Despite some common beliefs, Plattner et al. showed in 2009 that update-intensive tasks of transaction processing can be efficiently executed in DSM-powered main-memory database systems [32]. Today, it is known that neither DSM nor NSM is always the best choice [2], [3], [19], [29]. The reason for this is in the contradicting access pattern of HTAP workloads. The chosen physical record layout has a direct impact on the query execution performance, since the format affects which parts of the data are co-located and loaded in advance by hardware data prefetchers. If data is misplaced, the penalty is (i) a cache miss that requires to load the desired data first from main memory to higher cache hierarchies, and (ii) an unnecessary loading of additional data into the cache that might force an eviction of useful data [33]. This does not only apply to CPU caches, but also to the GPU's counterparts. Since GPU cache sizes are far more limited and graphics cards offer on-chip local caches in addition, data placement must be especially considered for GPU-based systems [10].

To emphasize the impact of (a) different physical storage layouts, (b) different compute platforms, and (c) different threading policies on the performance of (1) attribute-centric- and (2) record-centric queries, we share some findings resulting from our latest experiments[2]. We run both materialization and summing on records stored in the *customer*- resp. *item*

[1] The concept of NSM is also termed *slotted pages* in the literature.

[2] Source code is public available: https://github.com/PantheonDBMS see Pantheon-Research/Public/Storage-Engine/20170000D00HTAP/

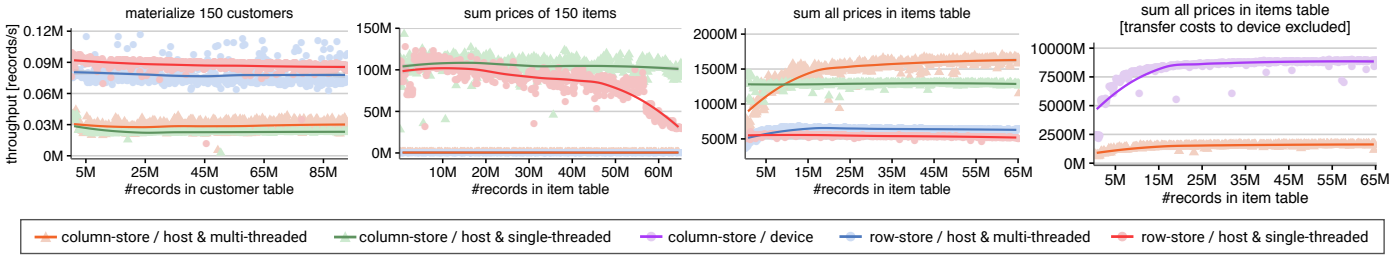

Fig. 2. Different attribute- and record-centric operations executed on the same tables of the TCP-C bechmark dataset. None of the solutions is optimal for HTAP workloads w.r.t. the storage layout, the threading policy or the data placement. The consequence is a space of choices that must be considered by the storage- and execution engine.

table of the popular TCP-C benchmark, and consider record-centric resp. attribute-centric data access pattern. In our setting, a customer record has a size of 96 bytes for 21 fields, and an item record has a size of 20 bytes for 4 fields + 8 bytes for the *price* field. We assume that the entire database can be kept in main memory. We vary the storage model, threading policy and compute platform. Operator execution follows the bulk-style processing model with late materialization. For the host platform, in case of multi-threaded execution, we fix to 8 threads with blockwise partitioning of the input data (i.e., each thread operates on one exclusive and subsequent list of input positions where each position refers to a certain tuplet in the corresponding input table). In case of single-threaded execution, there is no thread management involved at all. Thus, single-threaded execution runs sequentially on the main thread. On the device platform, we executed an optimized parallel reduction kernel[3] to calculate the sum of price fields. We configured the kernel to run with at least 1024 blocks (each having 512 threads). The final reduction was performed with 1 block and 1024 threads on the device, too. Although required to compute the answers to our test queries, we exclude the effort for join processing in our reports since these costs are orthogonal to our purposes. More in detail, we consider costs starting right after the output (i.e., sorted position lists) of the last directly preceding join operator is available.

In Figure 2, we depict the results from executing our experiments on commodity hardware[4]. The physical storage layout, the threading policy and the compute platform all affect the query performance; and there is no clear winner: (i) on a tiny number of records (i.e., OLTP-style queries), sequential execution outperforms multi-threaded execution since thread-management costs dominate, (ii) for record-centric operations, the NSM format outperforms the DSM format since NSM is more cache friendly here, (iii) for attribute-centric operations (OLAP-style), the DSM format outperforms the NSM with an argument similar as for (ii), and (iv) once the *price*-column is stored in device memory, a GPU outperforms a CPU (both with columnar storage), since a GPU exposes massive parallelism and higher throughput.

[3]The kernel based on a great tutorial by Mark Harris (chief technologist for GPU computing software at NVIDIA), see https://github.com/parallel-forall.

[4]x86_64 host w/ Intel Core i7-6700HQ CPU2.60GHz, 4 cores on single socket, L1/L2/L3 cache size: 32K/256K/6144K. 2x 8GB SODIMM synchronous main memory, running UBUNTU 16.04.1 LTS, host-compiler: CLANG++ 3.8 w/ O3 enabled; device: CUDA 8.0, capability 5.0, 4044 MBytes global memory, 5 multiproc. w/ 128 Cores/MP, L2: 2MB, max 1024 threads/block, no shared memory w/ host, compiler NVCC 8.0.44 w/ O3 enabled targeting 5.0 virtual GPU architecture

We outline two challenges for both employed compute platforms for HTAP workloads: (i) physical storage layout on the host, and (ii) under-utilization of the device. To overcome some limitations of contradicting access patterns by bridging between DSM and NSM for CPU-platforms, a storage model called *Partially Decomposed Storage Model* (PDSM) was proposed in 2010 [3]. In PDSM, a relation is (disjointly) partitioned into a set of *sub-relations* by using vertical partitioning [34]. PDSM is implemented within the HYRISE storage engine to achieve good performance for mixed-workloads. However, pure-PDSM aims to use bulk-style processing of partitions for improving data-cache efficiency compared to NSM. As shown by Arulraj et al. in 2016, PDSM is less efficient than DSM for several cases [2]. For GPU-platforms, it is important to understand that the GPU must always be kept busy to avoid under-utilization in face of its massive parallelism capabilities. Although He et al. suggested a bulk-processing model for transactions without user interaction (cf., [35]), it is currently unclear whether a fallback to the host platform is reasonable for general-purpose transaction processing.

III. Classification Properties and Taxonomy

While the definition of a *flexible storage model* (FSM, [2]) is sufficient to understand that an FSM format is somewhere between the NSM and the DSM approach; we argue it is too general for building a taxonomy of existing storage engines. Consequently, we propose a series of more fine-grained concepts. In concrete, we advance a system to consider the capability of HTAP-workload storage engines regarding their *layout* and *fragment* management.

To enable a uniform classification, we suggest both concepts, *layouts* and *fragments*, as a generalization from a magnitude of terms presented in the literature (e.g., cf. "column group" in [19] and "container" in [3]). We define a relation similar to common understanding with the following extensions: relations can have multiple alternative layouts; a layout is a complete relation divided into a set of possibly overlapping fragments. A fragment spans a "gapless" region of data in a relation. The per-tuple portion that falls inside a given fragment is called a *tuplet*. A *sub-relation* is a fragment of a relation R where all layouts in R are exclusively managed by vertical fragmentation.

We present a visualization of our terminology in Figure 3. In Figure 4 on Page 5 we depict an overview of the taxonomy we define using our terminology. We proceed with a more

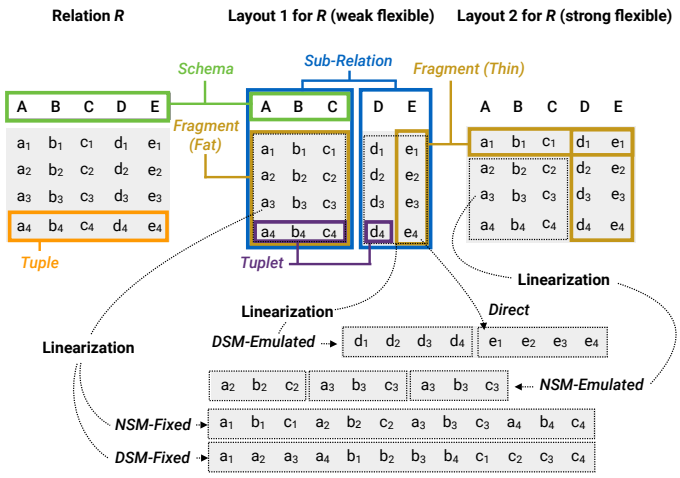

Fig. 3. Terminology used in this paper. A relation R can have multiple layouts each describing R in terms of several fragments (thin or fat). Pure-vertically partitioned layouts are called sub-relations. A tuple fragment in a fragment is called tuplet. Depending on the fragment type, the linearization type varies for NSM and DSM.

detailed introduction of properties to classify storage engines based on the concepts of layouts and fragments:

Layout handling. If a storage engine limits a relation R to have exactly one layout, then R has a *single layout*. Otherwise R is *multi-layout*. Storage engines can *emulate* a multi-layout property for a relation R by holding relations $R_1, R_2, ..., R_k$ under the same name, but relations in R have pair-wise different fragments (e.g., different storage models, or data locations) following a data replication strategy.

Layout flexibility. A storage engine is *inflexible* if it supports only one fragment per layout. Otherwise the storage engine is called *flexible*. A flexible storage engine is *weak* if all layouts apply the same partitioning technique to define fragments (either horizontal or vertical fragmentation). A weak storage engine always satisfies that its fragments are either in a vertical fragmentation or in a horizontal fragmentation. A flexible storage engine is *strong* if it supports layouts that combine vertical and horizontal partitioning to define fragments. If the definition of a fragment has side-effects to adjacent fragments (e.g., forcing a certain partitioning) in the context of a strong flexible layout, or if the order of the partitioning is pre-defined, then the layout flexibility is called *constrained*. Otherwise it is called *unconstrained*.

Layout adaptability. During runtime, a flexible storage engine might react to changes in the workload and adapt fragments of a certain layout. If a storage engine supports this dynamic re-organization of layouts, the storage engine's layout adaptability is *responsive*. Otherwise (or in case the storage engine is inflexible), it is called *static*.

Data location. Tuplets are stored on a certain storage medium, such as main-memory, device-memory, or flash drive. If all tuplets are stored exclusively in the main memory, then the fragment's data locality is called *host-memory-only*, conversely it is *device-memory-only* (or *secondary-memory-only*) if all tuplets are not stored in the main memory (e.g., they are stored exclusively in a compute platform's memory, or on disk). If the data location is *host-memory-only* or *device-memory-only*, the data locality is *centralized*. If the storage engine supports data locations that are neither *host-memory-only* nor *device-memory-only*, the data location is called *mixed* and the data locality is *distributed*.

Fragment linearization properties. A fragment of a relation can be *fat* or *thin*. A fragment is fat iff it contains at least two tuplets and at least two attributes in its schema. Since a fat fragment is two-dimensional, it must be *linearized* in order to be stored into one-dimensional memory. Linearization is sequentially arranging tuplets by either the NSM or DSM format. If a storage engine supports fat fragments but is restricted to either NSM or DSM, then the linearization is *NSM-fixed* or *DSM-fixed*. If the storage engine supports NSM or DSM for fat fragments, the linearization is *variable*. A fragment is thin iff it is not fat. Since a thin fragment is one-dimensional it does not require linearization. In this case, the linearization property is called *direct*. Flexible storage engines can emulate NSM-fixed or DSM-fixed linearization, by either horizontal or vertical fragmentation of a layout into thin-only fragments, and then applying direct linearization. This technique is called *NSM-emulated* or *DSM-emulated*. If this emulation does not cover the entire schema of the relation (i.e., some fragments remain fat), this technique is called *variable DSM-fixed partially NSM-emulated* if remaining fat fragments are DSM-fixed linearized or *variable NSM-fixed partially DSM-emulated* if remaining fat fragments are NSM-fixed linearized.

Please note, the difference between linearization of a fat fragment with DSM, and linearization of n thin fragments with DSM-emulated: the first stores *all* per-column fields of tuplets in *one subsequent* block of memory, while the latter stores column fields of tuplets in n different memory blocks (one per column). The latter appears for concepts where columns are equivalent to multiple distinct vectors, while the former appears for concepts where columns are stored in one single vector. The same applies for NSM resp. NSM-emulated.

Fragment scheme. In multi-layout relations, there are more fragments than are actually required to cover the tuples of a relation. A *replication*-based approach holds copies of tuplets (e.g., with different storage model formats) that cannot be referenced between fragments of multiple layouts (e.g., when the format definition of the data storage model is contradictory). A *delegation*-based approach restricts the access of certain regions from certain layouts, since some tuplets are exclusively stored in certain layouts. As a consequence, there is no data redundancy between layouts for non-shared data regions. However, storage engines using a delegation-based approach must manage delegation policies to avoid undefined behavior.

In the following Section IV, we employ our proposed taxonomy, to provide an account on storage engines.

IV. Survey and Classification

Several promising storage engines have been proposed in the last decades. In this section we survey some storage engines (Section IV-A) and database systems (Section IV-B) to classify them regarding the properties that we suggest in Section III. We provide a summary on our classification in Table I. In

	Layout handling	Layout flexibility	Layout adaptability	Data location	Fragment linearization	Fragment scheme	Processor support	Workload support	Date / Paper
PAX	single	inflex.	static	Host + Disc centr.	fat, DSM-fixed	-	CPU	HTAP	2002 [25]
FRAC. MIRRORS	built-in multi	inflex.	static	Host + Disc distr.	fat, NSM+DSM-fixed	replication	CPU	HTAP	2002 [36]
HYRISE	single	weak flex.	respons.	Host + Host centr.	fat, variable	-	CPU	HTAP	2010 [3]
ES^2	built-in mult.	strong flex.	respons.	Host. + distr.	fat, DSM-fixed	delegated	CPU	HTAP	2011 [37]
GPUTX	single	weak flex.	static	Dev. + Dev. centr.	thin, DSM-emulated	-	GPU	OLTP	2011 [35]
H_2O	single	weak flex.	respons.	Host + Host centr.	v. NSM-fixed p. DSM-emul.	-	CPU	HTAP	2014 [19]
HYPER	single	strong flex.	respons.	Host + Host centr.	thin, DSM-emulated	-	CPU	HTAP	2015 [38]
COGADB	built-in multi	weak flex.	static	Mixed + distr.	thin, DSM-emulated	replication	CPU / GPU	OLAP	2016 [16]
L-STORE	single	strong flex.	respons.	Host + Host centr.	DSM-emulated	delegated	CPU	HTAP	2016 [39]
PELOTON DBMS	built-in mult.	strong flex.	respons.	Host + Host centr.	fat, variable	delegated	CPU	HTAP	2016 [2]

TABLE I. SUMMARY OF SURVEY ORDERED BY DATE (HOST = HOST MEMORY, DEV = DEVICE MEMORY).

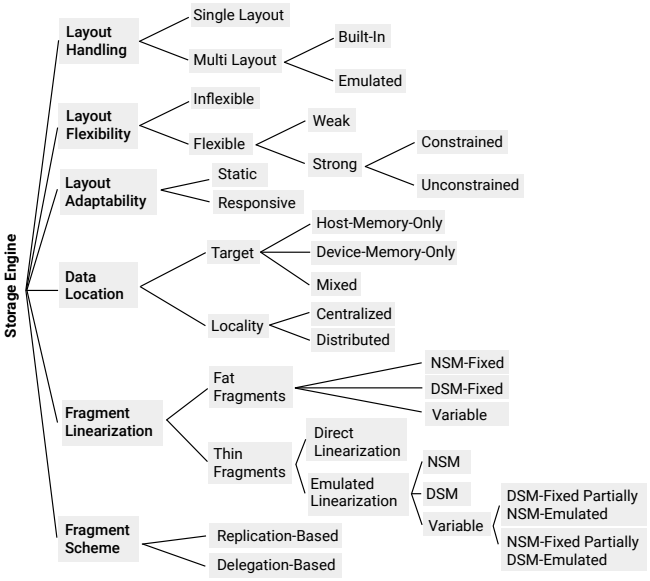

Fig. 4. Taxonomy on classification properties of storage engines.

Section IV-C we provide a wrap-up of our findings w.r.t. hybrid workload management in CPU / GPU database systems.

A. Storage Engines

Next, we survey notable storage engines proposed by early research (e.g., PAX) and more recent research (e.g., ES^2).

1) PAX: With the PAX storage model [25], Ailamaki et al. proposed a page-level decomposition storage model in the context of disk-based database systems that try to get the best of both storage models DSM and NSM. Conceptually, a relation has one layout that is horizontally split in n fat fragments where n is determined by the page size. Each fat fragment is afterwards linearized using a DSM-fixed approach. Therefore, PAX is a single-layout storage approach based on horizontal fat fragments using DSM-fixed linearization. PAX has a static layout adaptability since neither the fragmentation strategy nor the linearization technique can be changed. PAX was designed for disk-based systems powered by a database buffer manager. Consequently, the primary storage is the hard disk drive. However, the working set is kept in main-memory and PAX was evaluated on a single machine. Although both of these properties are not inherently required for PAX, the original concept relates to a host-only data location with centralized data locality on the secondary storage.

2) Fractured Mirrors: An early approach from 2002 to manage conflicting linearizations models (i.e., NSM vs DSM) in HTAP workloads for disk-based database system is the replication-based inflexible multi-layout *fractured mirrors* approach by Rösch et al. [36]: the idea is to have two *logical* copies of a relation with each possessing its own storage model rather than having two physical copies of the relation on two disks. In fractured mirrors, a relation has two layouts, with one fat fragment each that spans the entire schema of the relation, and which is linearized using the NSM (or DSM) format. In detail, fractured mirrors hold a number of NSM-styled pages and n additional DSM-styled pages where n is the number of attributes in the schema of the relation. Thus, the relation is physically replicated at page level. Fractured mirrors considers the data skew on multiple disks while guaranteeing data mirroring in case of physical failures of a single disk. With fractured mirrors, the pages of both fragments are distributed on disks such that each disk holds a copy of the relation but both fragments are equally represented on all disks. Thus, fractured mirrors uses an NSM-fixed/DSM-fixed technique.

3) HYRISE*:* In 2010, Grund et al. proposed a weak flexible storage engine in the context of host-only data with centralized storage [3]. A relation in HYRISE is laid out by n sub-relations which are called *containers*. Each container in HYRISE is formatted as a list of continuous memory blocks. A sub-relation can vary regarding the number of attributes the sub-relation schema contains. In addition, each sub-relation can be formatted using NSM or DSM. Since HYRISE manages fat fragments, it can apply NSM or DSM linearization for tuplets. HYRISE supports both linearization techniques for all fragment types, and variable linearization on fat fragments. With the aim of improving co-location of data and cache efficiency for HTAP workloads, HYRISE supports an automatic re-adapting of per-sub-partition widths. Therefore, the storage engine in HYRISE is responsive to workload changes. However, HYRISE follows a single layout approach since a relation has a certain layout at a time.

4) ES^2: The system EPIC is an elastic power-aware cloud platform for data-intensive applications in the context of distributed computing. The motivation behind this platform is to enable efficient management of HTAP workloads for cloud computing. One notable property of EPIC is its intentional use of the relational data model instead of the dominating key-value data model for transactional cloud platforms. This design decision is driven by the requirements for analytic

processing (as part of HTAP processing) in the cloud. In 2011, Cao et al. provided insights into EPIC's elastic storage engine that is designed for large cluster of shared-nothing commodity machines, ES^2 [37]. ES^2 supports relations to be fragmented via both vertical and horizontal partitioning. Fragment re-adaption is continuously executed based on query workload traces. The fragmentation strategy is built-in and consists of two steps. First (but optional), if columns are frequently accessed together, then these columns are moved into one new physical sub-relation. This strategy allows to hide less-frequently accessed columns, which improves cache-efficiency resp. reduces I/O costs for attribute-centric data access. Second, each such sub-relation is automatically split into further fragments (called partitions) by horizontal partitioning. The latter step allows to minimize the number of workers that access multiple compute nodes by placing certain partitions intentionally at a certain node. Record-centric data access is managed with distributed secondary indexes. Thus, EPIC is powered by a constrained strong flexible storage engine. Since ES^2 distributes both indexes and partitions to nodes in the cluster, it exploits a delegation-based fragment scheme. However, for load balancing and fault tolerance, data can also be replicated. The backbone for data storage in ES^2 is a slightly modified Hadoop distributed file system (DFS) that is used as a raw-byte device to which PAX-formatted tuplets are written. Hence, the storage engine of EPIC exposes a distributed location of data that is stored on the host's compute platform memory or disk, and which inherits the fragmentation linearization property of PAX.

5) H_2O: With H_2O, Alagiannis et al. present a weak flexible storage engine that is capable of managing DSM and NSM for a single relation, responding to changes in the workload. Relations in H_2O are organized by n sub-relations created using a horizontal (i.e., weak-flexible) partitioning. Each fragment is per default a fat fragment linearized using NSM-fixed. However, if the number of attributes of a sub-relation is set to one, the fragment becomes a thin fragment that is directly linearized. In fact, if a relation with m attributes is split into m sub-relations, the DSM storage is emulated. Therefore, H_2O uses a variable NSM-fixed partially DSM-emulated linearization. Layouts in H_2O are responsive to changes in the workload during runtime by lazily applying a new layout after evaluating alternative layouts from a pool. However, since H_2O does neither support overlapping partitions nor multiple layouts for a single relation at a fixed time, H_2O is a single layout approach. As originally proposed by Alagiannis et al., H_2O is a storage engine for data stored in centralized host-only memory.

B. Database Systems

Next, we survey systems focusing on host/device memory.

1) GPUTX*:* A single transaction is a small and simple task that might underutilize the parallelism available in modern graphics cards. With GPUTX [35], He et al. propose an in-memory relational database prototype for transaction workload processing on graphics cards that addresses this issue by bulk-processing of transactions. GPUTX is powered by a storage engine that is tailored to the characteristics of graphics cards, e.g., the transfer costs from host to device memory and vice versa. A relation in GPUTx is organized by n thin fragment sub-relations. Since GPUTX is a proof-of-concept of GPU-based transaction processing, its weak-flexible storage manager does not consider multiple layouts. Since the storage engine of GPUTX addresses a sub-relation approach only, it cannot change the layout of a relation. Thus, the layout adaptability of GPUTX is static. GPUTX manages a result pool in host-memory that retrieves copies from the device-memory. Since the use of host-memory is required to deliver processing results to users but relations are stored and processed in device-memory, GPUTX uses a secondary-only data location.

2) HYPER*:* The key motivation behind the engineering of HYPER was to build an HTAP-workload database system with a competitive performance compared to dedicated systems specialized for a single workload-type [1]. The storage engine of HYPER was re-newed in 2012 by Funke et al. to support combined horizontal and vertical partitioning, i.e., contributing a flexible storage engine [38]. In HyPer, a relation is physically organized by a hierarchy of partitions, chunks and vectors. A partition in HYPER is a sub-relation, i.e., HYPER applies first vertical partitioning to a relation. A resulting sub-relation is further split into horizontal (inner) fragments (called chunks). Therefore, HYPER applies a constrained strong flexible layout to relations, since a relation is compound of multiple fragments having side-effects to each other. One such side-effect is the dictation of boundaries of chunks. However, a chunk in a sub-relation is organized as a set of vectors. Each vector represents exactly one attribute of the sub-relation's schema. Thus, a vector in HYPER is a thin fragment. Since the entire relation is organized that way, there are no fat fragments left. Consequently, HYPER applies a DSM-emulated fragment linearization approach. To the best of our knowledge, HYPER applies dynamic re-organization of fragments in the layout of relations but does not manage non-emulated multiple layouts. Hence, HYPER is powered by a single-layout storage engine. In addition, HYPER's storage engine is responsive to changes in an HTAP workload [38].

3) COGADB*:* With COGADB, Breß et al. proposed a cross-device CPU/GPU database system for analytic processing, featuring a weak flexible storage engine which is similar to GPUTX [7]. In contrast to GPUTX, COGADB addresses the problem of query plan generation in heterogeneous architectures following a hardware-oblivious paradigm. COGADB features a self-adapting query optimizer (HYPE) that learns cost models and balances the workload between all compute devices [8]. Since data movement to and from device memory is a notable bottleneck, COGADB allows thin fragment sub-relations of a relation to be kept on host-memory, device-memory, or on both memory locations using a replication-based approach. As a result, COGADB's storage engine supports mixed data locations with distributed data locality. COGADB follows an "all or nothing" approach for moving a thin fragment (i.e., the i-th column of a relation) from host to device memory: either there is enough space for the column in the device memory, or not. If there is enough space, the column is placed in the device memory. Otherwise a fallback operation is scheduled that leaves the column in host memory. If the column fits into device memory, COGADB applies several strategies to handle side-effects (e.g., cache trashing or heap contention) during query processing on graphics cards [16]. In its current version, COGADB's storage engine exposes multiple layouts on a relation but applies

exclusively vertical fragmentation to a set of columns.

4) L-STORE*:* In early 2016, Sadoghi et al. present the main-memory database system L-STORE that was designed to manage HTAP workloads with the capability of historic querying [39]. The underlying strong flexible layout responsive storage engine features demand-driven changes of the physical storage layout of tuples, optimizing either for write or for read operations. In L-STORE, a relation is encoded by three components: a set of *base* pages, a set of *tail* pages and a *page dictionary*. Base and tail pages are the primary data container for tuple fields. A pair of base and tail pages form a single attribute column of a relation. Both together, the base and tail pages in such a pair, contain all field data for the corresponding attribute for all contained tuples. Thus, L-STORE manages a relation by a set of sub-relations where each attribute in the relation's schema corresponds to a single vertical fragment. Since the mapping between attribute and vertical fragment cannot be changed, L-STORE exposes a single layout architecture. However, each fragment is further split individually into two parts: the upper read-only (and compressed) base page part and the lower append-only tail page part. An attribute field of a tuple is a reference to a value in the corresponding base page part of the relation to which the tuple belongs, rather than a concrete value. This design enables a fine-grained control of attribute values. When the value of a field for a certain tuple (called *base record*) is modified, a new tuple (called *tail record*) is appended to the relation. This tail record shares the same references to base page values as its out-dated counterpart (i.e., its base record) with one exception: the modified field. The modified field points to a newly added value in the tail page part. The book-keeping between pages and records is in the responsibility of the page dictionary. The page dictionary also hides the information from its clients whether a certain record is made of base or tail pages. L-STORE applies DSM-emulated fragment linearization to satisfy attribute-centric query performance requirements. For record-centric queries, L-STORE requires to dereference values that are spread between multiple fragments. This might cause additional cache misses in direct comparison to records that are formatted using plain NSM. However, the deep integration of historic data handling is a notable feature of the L-STORE storage engine.

5) PELOTON*:* Recently, Arulraj et al. suggest a multi-layout storage engine with a *tile-based* architecture in the context of main-memory database systems issued with HTAP workloads. Their proposal is implemented in the PELOTON database [2]. In a tile-based architecture, a relation is represented in terms of *tile groups*. A tile group is a horizontal fragment. Each fragment in a tile group is further vertically fragmented into (inner) fragments called *logical tiles*. Similar to HYPER, this design is a constrained strong flexible layout approach with the same argument as for HYPER. The difference to HYPER is the order of vertical resp. horizontal fragmentation at the logical-tile level. However, in the tile-based architecture, logical tiles contain references to values stored in several *physical tiles*. The authors argue for this concept, which they call *layout transparency* (LT). LT enables to abstract from tuplets in a logical tile. This means, fragment linearization is done in a physical tile rather than in a logical one. A physical tile is a fat fragment incorporating tuplets from several layouts from different relations. Tuplets in physical tiles can be physically formatted using NSM or DSM. Thus, the tile-based architecture exposes a variable fragment linearization. Unfortunately, the authors do not explicitly state how the data in logical tiles is actually linearized. However, their presented storage engine was evaluated in PELOTON which is a main-memory-focused database system. Thus, their approach is primary-only centralized regarding the data location. The tile-based architecture exposes a delegation-based fragment scheme, i.e., tuplets between several layouts of several relations can be shared due to the logical tiles abstraction and sharing of tuplet values in terms of physical tiles.

C. Wrap-Up & Consequences

Based on our in-depth examination of storage engines, we can conclude that none of today's database systems are ready to process HTAP workloads employing both CPU and GPU. This holds on both directions: latest research on flexible storage approaches w.r.t. HTAP-workloads in main-memory fails to consider graphics cards as storage medium (and GPUs as processing unit). Conversely, none of today's GPU-powered database systems combine analytic- and transaction processing with HTAP workload processing. This distinction is reflected in the design and capabilities of storage engines for these systems. Clearly, none of the HTAP-workload main-memory database systems are aware of characteristics of graphics cards; especially they cannot consider operator or data placement to graphic cards for query processing. Likewise, no CPU / GPU database system has a storage engine capable to fulfill the needs of HTAP-workload processing (e.g., layout flexibility, or more advanced concurrency control). To contribute to bridging this gap, we next present our suggestion for a reference storage engine design: (1) at least constrained strong flexible layout support, (2) layout responsive to changes in workloads, (3) mixed data location and distributed data locality, (4) fragmentation linearization that cover NSM and DSM, (5) built-in multi layout handling for relations, and (6) fragment scheme supports delegation.

V. SUMMARY

In this paper we ask the question whether current database systems are ready for HTAP workloads on CPU and GPU. Our question is driven by two facts and one observation. First, recent research on GPU processing in databases has shown potentials to increase the overall query execution performance, constituting a promising solution to address technical limitations, such as the power wall. Second, the hybridization of analytic and transaction processing database systems is an increasing need in modern data management, and much has been done to provide advanced storage engines that dynamically solve the physical optimization contradictions from combining both workload types. Our observation is that both research areas are to-date distinct from each other. This is critical since it prevents support for HTAP workloads on database systems powered by both CPU and GPU. By considering the latest concepts for storage management from both domains, we can answer our question with a resolute *not yet*. Since a storage engine is one of the most fundamental components in any database system, it is only natural that to move research forward, one must first appraise what has been done so far in both research areas and what can be expected when merging

the concerns of both to enable HTAP workloads on CPU and GPU. For this, we introduced a terminology based on *layout*- and *fragment*-management which generalizes techniques presented in the literature, and enables a conceptual comparison of diverse storage engines presented in the literature. On top of this terminology, we introduced relationships and hierarchies for terms of this taxonomy. We use both, the presented terminology and taxonomy, to examine state-of-the-art storage engines and database systems from a conceptual perspective. We conclude that none of today's storage engines are capable to process HTAP workloads on CPU and GPU, achieving the same benefits that one dedicated system for a single domain could do. Finally, we present a reference storage engine design that covers the ideal storage engine capabilities drawn from both, HTAP workloads on CPU and OLTP or OLAP processing on GPU. These design characteristics must be considered for competitive HTAP workload processing on CPU and GPU.

VI. Acknowledgment

We thank our reviewers for their valuable feedback, and the DFG for funding (DFG; grant no.: SA 465/50-1).

References

[1] A. Kemper and T. Neumann, "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots," in *ICDE*. IEEE Computer Society, 2011, pp. 195–206.

[2] J. Arulraj, A. Pavlo, and P. Menon, "Bridging the archipelago between row-stores and column-stores for hybrid workloads," in *SIGMOD*, vol. 19, 2016, pp. 57–63.

[3] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudré-Mauroux, and S. Madden, "HYRISE - a main memory hybrid storage engine," *PVLDB*, vol. 4, no. 2, pp. 105–116, 2010.

[4] P. Rösch, L. Dannecker, G. Hackenbroich, and F. Färber, "A storage advisor for hybrid-store databases," *PVLDB*, vol. 5, no. 12, pp. 1748–1758, 2012.

[5] M. Pezzini, D. Feinberg, N. Rayner, and R. Edjlali, "Hybrid transaction/analytical processing will foster opportunities for dramatic business innovation," *Gartner*, 2014.

[6] S. Borkar and A. A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[7] S. Breß, "The design and implementation of CoGaDB: A column-oriented GPU-accelerated DBMS," *Datenbank-Spektrum*, vol. 14, no. 3, pp. 199–209, 2014.

[8] S. Breß and G. Saake, "Why it is time for a HyPE: A hybrid query processing engine for efficient GPU coprocessing in DBMS," *VLDB PhD Workshop*, vol. 6, no. 12, pp. 1398–1403, 2013.

[9] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake, "GPU-accelerated database systems: Survey and open challenges," *TLDKS*, vol. 15, pp. 1–35, 2014.

[10] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander, "Relational query coprocessing on graphics processors," *ACM Trans. Database Syst.*, vol. 34, no. 4, pp. 1–39, Dec. 2009.

[11] M. Pinnecke, D. Broneske, and G. Saake, "Toward GPU accelerated data stream processing," in *Proc. GI-Workshop GvDB*. GI, 2015, pp. 78–83.

[12] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen *et al.*, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *Journal of Parallel and Distributed computing*, vol. 61, no. 6, pp. 810–837, 2001.

[13] A. P. Sheth and J. A. Larson, "Federated database systems for managing distributed, heterogeneous, and autonomous databases," *ACM Computing Surveys (CSUR)*, vol. 22, no. 3, pp. 183–236, 1990.

[14] M. Pinnecke and B. Hoßbach, "Query optimization in heterogenous event processing federations," *Datenbank-Spektrum*, vol. 15, no. 3, pp. 193–202, 2015.

[15] G. Chen, X. Shen, B. Wu, and D. Li, "Optimizing data placement on GPU memory: A portable approach," *IEEE TC*, vol. PP, no. 99, pp. 1–1, 2016.

[16] S. Breß, H. Funke, and J. Teubner, "Robust query processing in co-processor-accelerated databases," in *SIGMOD*. ACM, 2016, pp. 1891–1906.

[17] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "SAP HANA database: Data management for modern business applications," *SIGMOD Rec.*, vol. 40, no. 4, pp. 45–51, 2012.

[18] G. P. Copeland and S. N. Khoshafian, "A decomposition storage model," *SIGMOD Rec.*, vol. 14, no. 4, pp. 268–279, 1985.

[19] I. Alagiannis, S. Idreos, and A. Ailamaki, "H_2O: A hands-free adaptive store," in *SIGMOD*. ACM, 2014, pp. 1103–1114.

[20] T. Neumann, T. Mühlbauer, and A. Kemper, "Fast serializable multi-version concurrency control for main-memory database systems," in *SIGMOD*. ACM, 2015, pp. 677–689.

[21] M. Stonebraker, G. Held, E. Wong, and P. Kreps, "The design and implementation of INGRES," *ACM Trans. Database Syst.*, vol. 1, no. 3, pp. 189–222, Sep. 1976.

[22] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson, "System R: Relational approach to database management," *TODS*, vol. 1, no. 2, pp. 97–137, Jun. 1976.

[23] D. J. Haderle and R. D. Jackson, "IBM database 2 overview," *IBM Syst. J.*, vol. 23, no. 2, pp. 112–125, Jun. 1984.

[24] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, Jun. 1970.

[25] A. Ailamaki, D. J. DeWitt, and M. D. Hill, "Data page layouts for relational databases on deep memory hierarchies," *VLDB Jour.*, vol. 11, no. 3, pp. 198–215, 2002.

[26] R. Ramakrishnan and J. Gehrke, *Database management systems*. McGraw-Hill, 2000.

[27] J. Gray and A. Reuter, *Transaction processing: Concepts and techniques*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.

[28] G. Graefe, "Volcano – an extensible and parallel query evaluation system," *TODS*, vol. 6, no. 1, pp. 120–135, Feb. 1994.

[29] H. Pirk, F. Funke, M. Grund, T. Neumann, U. Leser, S. Manegold, A. Kemper, and M. Kersten, "CPU and cache efficient management of memory-resident databases," in *ICDE*. IEEE Computer Society, 2013, pp. 14–25.

[30] D. J. Abadi, S. R. Madden, and N. Hachem, "Column-stores vs. row-stores: How different are they really?" in *SIGMOD*. ACM, 2008, pp. 967–980.

[31] M. L. Kersten, S. Plomp, and C. A. van den Berg, "Object storage management in goblin." in *IWDOM*, M. T. Özsu, U. Dayal, and P. Valduriez, Eds. Morgan Kaufmann, 1992, pp. 100–116.

[32] H. Plattner, "A common database approach for OLTP and OLAP using an in-memory column database," in *SIGMOD*. ACM, 2009, pp. 1–2.

[33] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, "DBMSs on a modern processor: Where does time go?" *PVLDB*, pp. 266–277, 1999.

[34] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis, "Weaving relations for cache performance," *PVLDB*, pp. 169–180, 2001.

[35] B. He and J. X. Yu, "High-throughput transaction executions on graphics processors," *PVLDB*, vol. 4, no. 5, pp. 314–325, Feb. 2011.

[36] R. Ramamurthy, D. J. DeWitt, and Q. Su, "A case for fractured mirrors," *PVLDB*, pp. 430–441, 2002.

[37] Y. Cao, C. Chen, F. Guo, D. Jiang, Y. Lin, B. C. Ooi, H. T. Vo, S. Wu, and Q. Xu, "ES^2: A cloud data storage system for supporting both OLTP and OLAP," in *ICDE*. IEEE, 2011, pp. 291–302.

[38] F. Funke, A. Kemper, and T. Neumann, "Compacting transactional data in hybrid OLTP&OLAP databases," *PVLDB*, vol. 5, no. 11, pp. 1424–1435, Jul. 2012.

[39] M. Sadoghi, S. Bhattacherjee, B. Bhattacharjee, and M. Canim, "L-Store: A real-time OLTP and OLAP system," *CoRR*, vol. abs/1601.04084, 2016.