Is Scalable OLTP in the Cloud a Solved Problem?

Analyzing Data Access for Distributed OLTP Architectures

Tobias Ziegler
tobias.ziegler@cs.tu-darmstadt.de
Technische Universität Darmstadt

Philip A. Bernstein
philbe@microsoft.com
Microsoft Research

Viktor Leis
leis@in.tum.de
Technische Universität München

Carsten Binnig
carsten.binnig@cs.tu-darmstadt.de
Technische Universität Darmstadt & DFKI

Abstract

Many distributed cloud OLTP databases have settled on a shared-storage design coupled with a single-writer. This design choice is remarkable since conventional wisdom promotes using a shared-nothing architecture for building scalable systems. In this paper, we revisit the question of what a scalable OLTP design for the cloud should look like by analyzing the data access behavior of different systems. We find that a shared-storage design that supports multiple writers, combined with a coherent cache, has desirable properties for building scalable OLTP systems. For instance, this design provides scalability of writes and skewed reads by caching hot tuples on demand. At the same time, this design does not require user-defined partitioning and two-phase commit in contrast to a shared-nothing design. We present the lessons learned from building a multiple-writer cache-coherent cloud OLTP DBMS to show this architecture's merits and present open research challenges.

1 Introduction

The Cloud is Taking Over. The DBMS market has shifted significantly from on-premise to the cloud in the last few years. According to a recent market report[1], in 2021 DBMS revenue in the cloud was on par with the on-premise market. Given current growth rates, the cloud DBMS market will be substantially larger by CIDR 2023. Consequently, classical DBMS vendors such as IBM and Oracle have been or are about to be overtaken by hyperscalers such as AWS, Microsoft, and Google, which have heavily invested in their cloud-native DBMSs.

Cloud-Native OLTP. Looking at cloud-native OLTP DBMSs, we see an interesting trend: Many commercially-available systems such as Amazon Aurora [53] and Microsoft Socrates [2], use a disaggregated design (shared-storage) coupled with the primary-secondary paradigm. In this design, all write transactions go to the primary node, which sends its write-ahead log to the shared storage so that secondary nodes can access it. The storage nodes use the log to reconstruct the data pages in the background. Those reconstructed pages can be read by secondary nodes on demand and thus be spawned at any time with low overhead. While this design provides important features such as hot fail-over as well as elasticity and load-balancing for read-dominated workloads, it is limited by the capacity of the primary node – particularly for write-intensive workloads, which are common in OLTP [9].

Decades of OLTP Research. The design choice of using a shared-storage architecture with a single writer node is remarkable since decades of research proposed different designs for building scalable distributed systems [21, 24, 34, 41]. For example, many early OLTP systems promoted the use of the shared-nothing architecture for scaling OLTP beyond the capacity of a single machine [21, 37]. An alternative is a shared-storage architecture with multiple read-write nodes, which has been used successfully since the 1980s. It is implemented in on-premise systems such as IBM DB2 Data Sharing [20], DEC's Rdb/VMS for VAXcluster [30], and Oracle RAC [7].

Research Question. Why have modern commercial cloud DBMSs not adopted these scalable designs? We believe there are several reasons: First, despite the substantial body of research, the rise of the cloud has disrupted our understanding of the landscape and tradeoffs of different DBMS designs. For example, it was often thought that the shared-nothing architecture is preferable for building distributed DBMSs. However, this is no longer so clear with cloud DBMSs that use shared storage. Second, even though promising shared-storage designs have been proposed and used in on-premise DBMSs, many of their core aspects crucial for a truly scalable cloud OLTP DBMS have yet to be fully explored. Third, some of the techniques for scalable OLTP are quite complex and may be risky to deploy at cloud scale. Simpler and more robust techniques are needed.

Mapping the OLTP Landscape. In this paper, we explore the design space of distributed OLTP DBMSs to help system developers to understand their fundamental differences for the cloud. More specifically, we analyze the data access path of distributed OLTP systems to understand their scalability properties. We observe that the old taxonomy of shared-storage and shared-nothing does not fully capture the scalability properties of modern cloud DBMSs. For instance, it was usually assumed that a shared-storage system allows many machines to perform updates. However, many cloud DBMSs today only allow a single read-write node, which limits the system's scalability. Therefore, as the first contribution in this paper, we distill the data access behavior of distributed OLTP DBMS designs that have been proposed over several decades. We organize the design space into three data access archetypes with different features and scalability behaviors.

Towards a Scalable Cloud OLTP DBMS. As a main result, by analyzing the different OLTP archetypes, we show theoretically and experimentally that a shared-storage design supporting multiple writers, combined with a coherent cache, has desirable properties for building scalable OLTP systems. We advocate that future cloud

[1] https://blogs.gartner.com/merv-adrian/2022/04/16/dbms-market-transformation-2021-the-big-picture/

Table 1: Data Access Archetypes for Analyzing the Asymptotic Scalability of Cloud OLTP DBMSs.

		Single-Writer	Partitioned-Writer	Shared-Writer	
				No Coherent Cache	*Coherent Cache*
	Systems based on Archetypes	AWS Aurora [53], Azure SQL HyperScale [2], PolarDB [28], AlloyDB [14]	System R* [37], Spanner [8], YugabyteDB [18], H-Store [21] CockroachDB [50]	NAM-DB [58], Sherman [54]	ScaleStore [60], Oracle RAC [40], IBM DB2 Data Sharing [20], GAM [5], DEC VAXcluster [30]
Qualit. Features	Data Access Path	Single-writer, multiple readers	One writer per partition	Writers update (remote) shared storage	Writers update cache-local data governed by coherence protocol
	System Complexity	Low	Medium	Medium	High
	Elasticity†	Only Reads	Limited	Yes	Yes
Asymptotic Scalability	Uniform Reads	Yes††	Yes	Yes	Yes
	Uniform Writes	No	Yes	Yes	Yes
	Skewed Reads	Yes††	Yes††	Yes††	Yes
	Skewed Writes	No	No	No	No

(†) Elasticity w.r.t. compute and storage separately (††) Only with the number of secondaries, i.e., replicas.

DBMSs be based on this archetype. However, many challenges exist, and building blocks still need to be added to enable a full cloud DBMS based on a shared-caching design. As a second contribution, we discuss lessons learned from building such a cloud-native system and present research opportunities for the community.

2 OLTP Data Access Archetypes

Four decades of OLTP research have led to a plethora of distributed OLTP DBMS designs. In 1985, Stonebraker provided a taxonomy for distributed DBMSs with the famous categorization into shared-nothing and shared-storage architectures that can be found in many DBMS textbooks today [48]. In a shared-nothing architecture, each node has private storage that is not accessible to other nodes. In a shared-storage architecture, all nodes have access to a single stored copy of the database.

Revisiting the Old Taxonomy. Although these designs are still relevant, changes in the system landscape make it worth revisiting the categorization. The reason is that many of today's systems are hard to characterize with the traditional taxonomy. For instance, many cloud-based shared-storage systems, such as Aurora single master and Azure SQL Hyperscale, only permit one update node to avoid the problem of coordinating multiple writers for buffer cache coherence. By contrast, the classical taxonomy assumed shared-storage systems allow many update nodes.

Conversely, in the classical taxonomy, it is assumed that a shared-nothing system allows only one node to update a given data item. However, many shared-nothing database systems today allow multiple nodes to update node-private replicas of the same data item. This is commonly called *multi-master*, which leads to entirely different scalability properties. Another shared-nothing design allows multiple nodes to accept update requests on a given data item and forward them to the one-and-only node that is allowed to process updates on that data item. We call this *pseudo multi-master*. The conventional taxonomy does not highlight such differences, which are important for differentiating the scalability properties of systems.

A New Taxonomy for OLTP DBMSs. Therefore, we propose a taxonomy that focuses on the data access path, which is another useful basis for capturing the scalability properties of systems. Our taxonomy allows us to analyze *asymptotic scalability*, that is, how well a system scales w.r.t the number of nodes for basic data access operations when increasing the load in the system. As shown in Table 1, we look at fundamental operations for OLTP, namely key-based reads and writes for uniform and skewed key distributions.

The old taxonomy focused heavily on transaction synchronization problems that arise in each design, e.g., whether it requires two-phase commit for transaction atomicity or global lock management to synchronize buffer caches. In our taxonomy, we instead mainly focus on the storage layer, i.e., the data access path, since it is fundamental for any OLTP DBMS to execute reads/writes scalably. In our discussions, we omit logging, recovery, and concurrency control [3, 32, 57] that are thoroughly investigated by Harding et al. [16], and Bernstein and Goodman [4]. We also omit deterministic databases [13, 31, 51]. Although they have some desirable properties, major cloud service providers do not offer them today.

We map the design space into the three data access archetypes shown in Table 1. An archetype can be considered the "phenotype of a DBMS" that determines its observable asymptotic scalability independent of its implementation details. Sections 2.1 - 2.3 describe the data access archetypes and their asymptotic scalability in more detail. Section 2.4 shows how our taxonomy is more descriptive for modern cloud DBMSs and highlights where the previous classification did not convey the data access properties. Section 2.5 discusses the effect of storage latency on the behavior of different archetypes.

2.1 Archetype: Single-Writer

In the Single-Writer archetype, a single read-write node (RW-node) processes update transactions, and multiple read-only nodes (RO-nodes) can handle read-only transactions. We depict the basic architecture of systems that implement this archetype in Figure 1. Many modern cloud DBMSs, such as AWS Aurora (single-master), Azure SQL Hyperscale, PolarDB, and AlloyDB use shared-storage. They support multiple nodes accessing a single stored database copy, but only one dedicated RW-node can execute updates. We classify such systems as Single-Writer systems.

While Figure 1 shows a Single-Writer with shared-storage, many DBMSs with private storage also fall into this category. For example, a MySQL instance in which the primary node executes reads and writes and replicates writes to read-only replicas is also categorized as Single-Writer even though the primary and replicas have node-private storage. The strictly separated data access permission is the distinguishing feature, not whether the storage is shared or private.

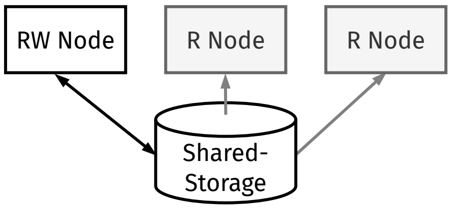

Figure 1: Archetype Single-Writer

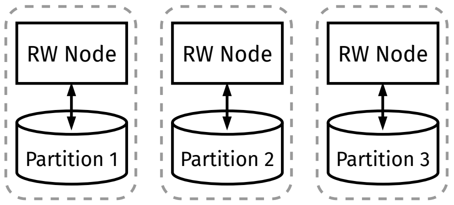

Figure 2: Archetype Partitioned-Writer

Asymptotic Scalability. The asymptotic scalability of this archetype can be summarized as follows: Writes in this archetype are limited by the capacity of the single RW-node. By contrast, read throughput can scale well by spawning more RO-nodes.

Still, the concrete performance characteristics of systems can vary due to other design decisions. For instance, cloud DBMSs with a replicated shared-storage can spawn RO-nodes very fast and react to an increase in read-only transactions. On the other hand, a mirrored instance may take longer to deploy an additional RO-node, since it requires creating another copy of the database. Furthermore, some Single-Writer systems can attain high throughput on large multi-core processors. However, their asymptotic scalability remains bounded by their limitation of a single RW-node.

Qualitative Features. This design provides important features for the cloud, such as fast failover and elasticity for read-dominated workloads. Similarly, the separation of compute and storage allows cloud vendors to handle growing data sets. At the same time, the systems' complexity is often lower since they only need to support a single-writer.

2.2 Archetype: Partitioned-Writer

In the Partitioned-Writer archetype, the database is split into partitions, each of which can be updated by only one RW-node. As shown in Figure 2, this archetype exploits data partitioning to spread the data across several nodes that are responsible for executing read-write operations on their local partition.

Like the Single-Writer design, the Partitioned-Writer archetype is a popular one for commercial cloud DBMSs. Examples include CockroachDB [25], AWS DynamoDB [45], Azure CosmosDB [33], and Spanner [8]. Most of these systems implement a coordination protocol, such as two-phase commit, to ensure atomicity of cross-partition transactions.

In our taxonomy, the data access path is the distinguishing feature, not whether the system uses instance local storage or (partitioned) disaggregated storage. The latter is commonly used in the cloud since a single partition can be dispersed to multiple storage nodes. In contrast, instance local storage typically offers lower latency albeit with a fixed storage capacity (see Section 2.5). Regardless of these implementation details, Partitioned-Writer systems provide the same asymptotic scalability as discussed next.

Asymptotic Scalability. In contrast to the Single-Writer design, DBMSs that implement the Partitioned-Writer archetype can handle uniform reads and writes in a scalable manner since they can process each partition's workload independently. However, skewed workloads can be challenging, since requests for hot keys are always processed by the same RW-node.

One technique to handle read-skew is to couple this archetype with additional RO-nodes (e.g., per partition) similar to Single-Writer systems. However, this needs more resources for database copies and requires propagating updates to RO-nodes to keep them in sync. Since adding RO-nodes is an orthogonal mechanism that can be used for any archetype, in the rest of the paper, we discuss each archetype in its *pure* form. This way, we can identify archetypes that provide scalability under read-skew without needing additional resources.

Qualitative Features. Classical partitioned systems offered limited elasticity since adding new nodes typically called for a full repartitioning of the database. Modern cloud DBMSs have added several optimizations to address this issue, e.g., consistent hashing or fine-grained range partitioning. For example, CockroachDB uses a fine-grained range partitioning of data chunks, which avoids a complete repartitioning and improves elasticity. However, fine-grained partitioning can add complexity and overhead to locating data. To address this, CockroachDB stores data in a monolithic sorted map of key-value pairs and uses it to route queries to the responsible RW-node of a partition. Overall, the elasticity of a Partitioned-Writer system depends on the workload characteristics and the system's ability to repartition the data efficiently.

Although this archetype sounds very similar to the shared-nothing architecture, many shared-nothing DBMSs can not be classified as Partitioned-Writer, and thus, their asymptotic scalability is different. For instance, some shared-nothing systems are multi-master. While they still use partitioned storage the data is replicated and can be modified by multiple RW-nodes leading to a different scalability behavior. The next archetype Shared-Writer captures this data access pattern.

2.3 Archetype: Shared-Writer

The Shared-Writer archetype allows multiple RW-nodes to modify data items – typically by utilizing shared-storage. Several recent research OLTP systems, such as NAM-DB [58], and a few established systems, such as Oracle RAC, can be classified as Shared-Writer systems. Because data must be kept consistent across writers, these systems must address the buffer-cache coherence problem [38, 44]. There are two ways of doing this: nodes always write and read from the shared storage to get the ground truth (*No Coherent Cache*), or the nodes participate in a cache coherence protocol (*Coherent Cache*). In the following, we will drill into these two cases.

2.3.1 Shared Writer with no Coherent Caches

First, we consider Shared-Writer systems without coherent caches. As shown in Figure 3, these systems support multiple RW-nodes that write to and read from the storage layer.

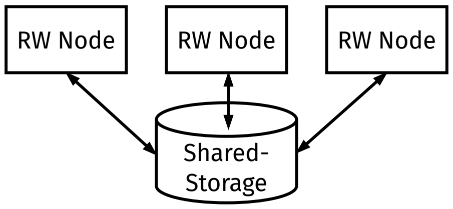

Figure 3: Archetype Shared-Writer without Coherent Cache

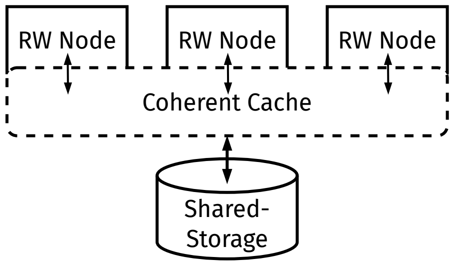

Figure 4: Archetype Shared-Writer with Coherent Cache

Asymptotic Scalability. In contrast to the Partitioned-Writer archetype in which only one RW-node writes to a partition, the Shared-Writer design allows every compute node to be an RW-node and to access every data item. Nevertheless, the Shared-Writer archetype has the same asymptotic overall scalability. For example, under access skew, the storage layer might receive requests from multiple RW-nodes and become a bottleneck, limiting the scalability of both writes and reads.

Qualitative Features. In the Partitioned-Writer design, compute and storage resources are usually scaled together. This is inflexible because the need to store more data or do more computation may change at different rates. Using a Shared-Writer design, we can scale compute and storage resources independently depending on the demand. This enables better data scalability and elasticity since any compute node can access any data item which is a desired property in the cloud.

However, one major challenge in this design is locating the data items in the storage layer. Like modern Partitioned-Writer DBMSs, modern disaggregated OLTP DBMSs use an index to locate items. The index resides remotely in storage and is accessed by compute nodes to find relevant data items. This additional requirement for indexes increases system complexity as shown in Table 1.

2.3.2 Shared-Writer with Coherent Caches

The main downside of the previous sub-archetype (No Coherent Cache) is that every data access involves a network round trip to the storage layer, even for repeated access. Caching can considerably reduce that latency by keeping recently accessed data items in the compute nodes' memory. In this archetype, we specifically consider coherent caches. These caches support shared writers and allow caching read-only data locally while keeping the data coherent (i.e., data does not become stale). Shared-Writer with coherent caches is similar to Single-Writer and Partitioned-Writer with synchronous or eager asynchronous replication since the replicas are essentially caches for reads. The difference is that the caches in this archetype are updatable.

Figure 4 shows that this archetype is, at first glance, similar to the previous Shared-Writer design. However, the coherent caches impact the asymptotic scalability as discussed in the following.

Asymptotic Scalability. This archetype has the best asymptotic scalability of all archetypes. The benefit of coherent caches is that they allow replicating frequently accessed (i.e., hot) data items on demand across multiple nodes. This workload-driven approach differs from the classical replication used in the previous archetypes. Using the coherent cache, systems of this archetype can handle read skew efficiently, as shown in Table 1.

Unfortunately, skewed writes can also not be processed in a scalable manner. This is no different than the other archetypes since each update must be exclusively handled by a single node to keep the data consistent. However, the cache coherence protocol may lead to the write privilege bouncing between compute nodes since all compute nodes can modify all data items. Solving this requires complex synchronization, a challenging problem.

Qualitative Features. While coherence protocols are non-trivial to implement efficiently, they provide many benefits. Compared to operating directly on the shared-storage, shared-caching makes accessing and maintaining remote indexes easier since index nodes are cached on demand. For instance, when an index node is updated, e.g., a B-Tree leaf, the cache-coherence protocol invalidates other copies of the index node and delivers the newest data version when the leaf is re-accessed.

Furthermore, the caching-based design provides good elasticity. Admittedly, this is more complicated than in the previous archetype without a cache since the coherence protocol must be designed accordingly (see Section 3.3). A new compute node can be added at any time, incrementally filling its cache as data items are accessed. In addition, these systems can handle easy- and hard-to-partition workloads. They do not require user-defined partitioning but still profit from partitionable workloads. Moreover, they have the flexibility to react to changes in workload distribution.

The lesson from our analysis is that a caching-based design is very promising for cloud DBMSs. Compared to the other archetypes, it has the best asymptotic scalability and provides data scalability and elasticity.

2.4 Categorizing Systems with Archetypes

In the following, we discuss how our new taxonomy can be used for categorizing the scalability properties of existing systems.

New Taxonomy Focuses on Data Access. As mentioned before, we see the archetypes introduced in our taxonomy as themes that better help developers understand the scalability properties of DBMSs. This is in contrast to the old taxonomy, which often falls short in classifying systems. A good example where the old taxonomy fails to express the system's scalability is FaRM [10] from Microsoft.

With the old taxonomy, this system cannot be easily classified as a shared-nothing or a shared-storage design rendering the discussion of the scalability properties difficult. FaRM provides a shared-storage abstraction in which workers can read data from every other node. Thus, for reads FaRM behaves as a shared-storage system. However, updates are shipped to the partition owner as in a shared-nothing. So it is unclear if the scalability characteristics are inherited from shared-storage or shared-nothing.

Instead, we can categorize FaRM as a Partitioned Writer using our taxonomy. That is, while every node can read from every other node, the writes are performed by the RW-nodes, which own the partition. Therefore, the system's asymptotic scalability is inherited from the Partitioned Writer design.

Multiple Archetypes for One System. Another interesting DBMS is Aurora multi-master [12, 27, 46], in which all DB nodes can be writers. This contrasts with Aurora's single-master, in which only one DB node can act as a writer. While Aurora single-master fits our Archetype of a single-writer and has the above-discussed characteristics, Aurora multi-master is a Shared-Writer system since it allows multiple compute nodes as writers.

Aurora multi-master allows each RW-node to read and write the whole database. It uses optimistic concurrency control (OCC) to detect conflicts with operations from other writers. The underlying storage system is very similar to Aurora single-master [53], except that it performs the OCC test. A writer sends each update to all storage replicas with a copy of the data it wrote. A storage replica accepts the first write it receives for the latest page version and rejects ones that arrive later. The write succeeds if a quorum of storage servers accepts it. Otherwise, the write fails, and the transaction that issued the write aborts.

Writers send their committed updates to other writers, which the other writers use to update their cache. This is how Aurora maintains a weak form of cache coherence. It is weak in the sense that stale versions of an updated page are not immediately invalidated in all caches. It is similar to update propagation on replicas in a single-writer system, but for a different purpose. Here, update propagation increases the chance that future writes by a node operate on fresh data and, hence, will be accepted by the OCC check.

Caching in Single-Writer Systems. A third interesting system is PolarDB Serverless [6]. While traditional PolarDB is very similar to AWS Aurora due to using log servers and log replay on the RO-nodes, PolarDB Serverless uses a cache-coherence protocol to update the RO-nodes. Yet, the fact that they use a cache coherence protocol does not turn PolarDB Severless into a shared-writer system since the data access path only allows a single RW-node.

Optimizations for Partitioned-Writer. A further relevant research direction is shared-nothing databases that dynamically repartition data. Several papers repartition the shards online when (severe) workload imbalances are detected [11, 23, 29, 42, 49, 55]. This process balances the load at runtime while minimizing the impact on running transactions. This allows Partitioned-Writer databases to behave similarly to the shared-writer archetype with a coherent cache in that they can scale better than Single-Writer under skew. However, the fundamental difference is that only one RW-node can process operations on a given partition; thus, adapting to changing workloads requires expensive repartitioning. As such, very skewed workloads can still overload a partition and limit scalability.

2.5 The Importance of Latency

So far, our discussion of scalability has focused exclusively on throughput. However, latency is also an important factor for OLTP performance since it influences lock holding times and other factors that can affect the throughput of an OLTP DBMS. In the following, we first discuss factors that influence latency and ways it affects throughput. We then discuss considerations on how to reduce it.

The Importance of Latency to Throughput. Latency in an OLTP DBMS depends very much on workload characteristics. Transactions that need to navigate the relationships between tuples of different tables can require many storage accesses. Tuples have complex and sometimes subtle relationship graphs in the form of joins, foreign keys, being modified by the same transaction, being part of the same group by clause, matching the same filters, etc. Evaluating those relationships and maintaining their integrity typically requires many separate sequential memory accesses, which drives up response and lock holding time, which reduces throughput.

Transaction latency could be seen as independent of throughput scalability. But in practice higher transaction latency is quite problematic at scale. For example, many distributed relational DBMSs adopt the protocol of an existing relational DBMS, such as PostgreSQL, to support existing libraries and tools. These protocols are typically synchronous and interactive, which effectively bounds achievable throughput to the number of sessions divided by average response time. Increasing the number of sessions to achieve higher throughput is not always practical and generally has adverse effects such as increasing memory pressure and resource contention. Higher response time also leads to longer lock holding time, which further limits achievable concurrency and throughput. Finally, tools and libraries built for a single-node relational DBMS rarely anticipate high response times. For instance, web development frameworks routinely do a dozen sequential queries for a single page load because they expect queries to return in less than a millisecond. In the cloud, that can take much longer. Custom libraries and protocols could help achieve scalability in the presence of high response times, but that limits applicability. Thus, even if throughput is a more important user requirement than latency, optimizing latency is essential to reach the throughput goal.

Cutting Latency by Push-down. Storage latency contributes to transaction latency and hence can limit throughput scalability. Storage latency in the cloud is relatively high, since there is rarely enough capacity to have all servers in close proximity, e.g., in the same rack. Moreover, close proximity might be undesirable since it degrades availability due to correlated failures. One way to reduce storage latency is by using node-local storage instead of disaggregated storage. However, this is typically not desirable in the cloud since this prevents not only independent scalability of the storage but also hinders other properties, such as elasticity. A technique that helps to reduce latency in systems that use a disaggregated (shared) storage design is to push down the access to storage servers. While this is harder to do in a shared-storage DBMS that uses a simple file system, it is conceivable with a shared-storage DBMS with a custom-built storage system, such as Aurora and SQL Server Hyperscale. However, computation push-down risks consuming too much of the storage system's limited compute capacity, thereby increasing the latency of simple storage accesses. Exploring query and update push-down to the storage layer of OLTP systems is thus potentially an interesting research opportunity.

Cutting Latency by Caching. Another direction in a shared-storage system that can mitigate the impact of storage latency is keeping the working set in a node-local cache which is possible for all archetypes as discussed before. This cache can avoid distributed processing of transactions and thus remote data accesses. Moreover, caching can be used by both RW-nodes and RO-nodes. However, caching designs come with their own challenges. For example, to ensure that read-only replicas remain fresh, they need to apply the log of committed updates to their cached data. Hence, cutting down the latency of log replay is one interesting challenge. Another research opportunity in disaggregated (shared-storage) designs is to find efficient ways to mitigate storage latency when the working set does not fit in the cache.

Latency and Throughput are Important. In summary, our analysis of the throughput scalability of archetypes is only part of the story. When latency is taken into account, the tradeoffs can be even more complex. This is also reflected in the discussions of the following section.

3 Towards a Scalable Cloud OLTP DBMS

Section 2 showed that a design with shared-writers and coherent caches, in the following called shared-cache systems, provide the best asymptotic scalability and are a solid foundation for modern cloud OLTP DBMSs. This section delves deeper into that design.

There was a large body of research on shared-cache DBMSs in the early 1990's [19, 26, 38, 39, 47]. Several commercial OLTP DBMSs of that era used that design, such as IBM DB2 Data Sharing [20], DEC's Rdb/VMS for VAXcluster [30], and Oracle RAC [7]. However, these approaches were not designed for the cloud. Some systems relied on proprietary hardware, such as the coupling facility of IBM, that limit multi-cloud support since the custom interconnection hardware is unlikely to be supported in other vendors' clouds.

Meanwhile hardware has been evolving. Today, we rely on data center networks and large multicore servers. Customizable processors, such as FPGAs and ASICs, are becoming commonplace. To minimize cost, multitenancy is usually needed. And although Oracle is actively working on optimizing Oracle RAC for the cloud[2], the research community has not been focusing on shared-cache DBMSs for some years. Given these trends, we believe it is time to revisit, adapt, and improve upon existing research to design a cloud-native DBMS. We are not the only researchers with this sentiment. Mohan pointed out in his keynote that much of the work done in this area can now be reused in cloud DBMSs [35].

In the past year, some of us have started to build a new cloud shared-caching OLTP DBMS called ScaleStore[3][60]. In the following, we first discuss ScaleStore's design and then present research opportunities.

3.1 A Blueprint for a Shared-Caching DBMS

A key feature of shared-caching systems is that they do not rely on statically partitioned data. Instead, compute nodes cache data dynamically based on the access patterns of a workload. For instance, consider the index shown in Figure 5(i), which consists of pages P1-P5. We can see that the three compute nodes can cache different index pages depending on their workload.

[2]https://www.oracle.com/technetwork/database/options/clustering/overview/new-generation-oracle-rac-5975370.pdf

[3]https://github.com/DataManagementLab/ScaleStore

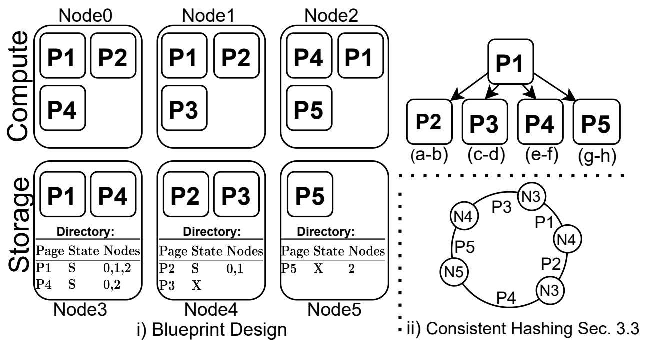

Figure 5: Shared-Cache Design of ScaleStore

Invalidation-based Coherence Protocol. Because an index page can be cached on multiple nodes (e.g., the root of the index P1), a coherence protocol is necessary. Thus, modifying a page in one node makes all copies of that page in other nodes obsolete. The coherence protocol has to ensure that these changes are detected so that nodes can access the most recent page. ScaleStore uses a *directory-based invalidation-based protocol* which provides sequential consistency at page-granularity. Thus, whenever a node plans to modify a page, other nodes must invalidate the page. However, instead of broadcasting the modification intent to all nodes, we use directories to track nodes that currently cache the page. Consequently, we send the invalidations only to the required nodes. Note that recently proposed memory coherence protocols [56, 59] are attractive alternatives to a directory coherence protocol and need to be evaluated in a distributed DBMS.

Directory per Storage Node. Every storage node is a directory for some of the pages as shown in Figure 5(i). Storage nodes keep track of those pages that are stored on them. For instance, Node 3 is the directory of P1 and P4. The directories are mainly responsible for managing the metadata for the invalidation-based cache coherence protocol. In Figure 5(i) this metadata is depicted in the tables on the storage nodes and reflects the state of the compute caches. For instance, P5 is held in exclusive-mode by Node 2 whereas P1 is cached on all compute nodes in shared-mode. Since tracking on a per-tuple granularity would be prohibitively expensive, we organize multiple tuples in fixed-size pages to amortize the bookkeeping overhead. These pages (e.g., 4 KB) are kept coherent across all nodes.

Supporting Arbitrary OLTP Workloads. For cloud DBMSs one important requirement is that they can support arbitrary (and potentially changing) workloads in a scalable manner. The page-based organization of data is a perfect fit for supporting primary and secondary indexes to access data efficiently. Those indexes allow the user to perform lookups, updates, and short-range scans very efficiently. Furthermore, the beauty of a page-based index organization is that every node of a distributed B-Tree is automatically backed by the coherence protocol. Thus, all participating compute nodes that currently cache a page observe changes to this page. In addition, the coherence protocol allows compute resources to dynamically cache the hot parts of the index (such as the inner B-Tree nodes). As a result, if the workload changes, the cache will adapt automatically.

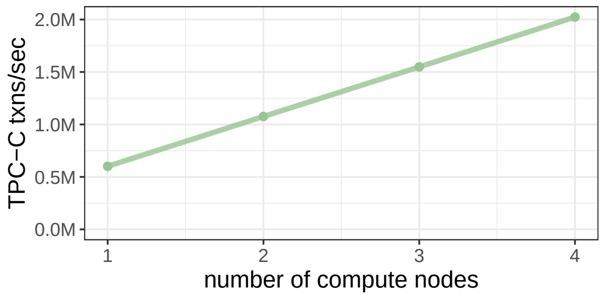

Figure 6: TPC-C Scale-Out Performance of ScaleStore with 4 Compute and Storage Nodes all Connected with RDMA. We use 20 Warehouses per Compute Node. Every Compute Node uses 20 Threads.

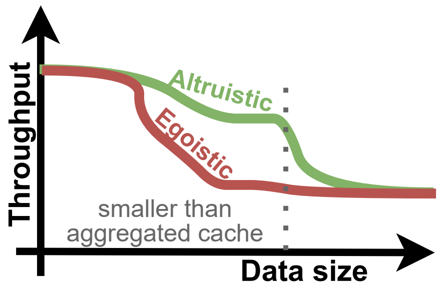

Figure 7: Effect of Egoistic vs. Altruistic Eviction when the Working Set is Smaller than the Aggregated Memory of Compute Nodes.

Proof of Concept. With the current design ScaleStore can already execute arbitrary OLTP workloads. We implemented a full-fledged TPC-C benchmark using our B-Trees as primary and secondary indexes. All experiments are run in read uncommitted mode, since ScaleStore does not yet implement full transaction semantics. The invalidation-based coherence protocol as described above only provides sequential consistency on a page-level. Figure 6 shows the results for a setup with 4 compute and 4 storage nodes connected with RDMA. We use 20 warehouses and threads per compute node. As we can observe, the throughput of ScaleStore scales with the number of nodes for the write-heavy TPC-C benchmark. This underlines the asymptotic scalability of the storage engine as discussed in Section 2.

3.2 Caching and Eviction Go Hand in Hand

Modern cloud data center networks are fast – which means that new pages may be added to the caches of compute nodes at very high rates. Thus, shared-cache systems need to evict cold, i.e., unused, pages quickly while making sure that hot pages remain in the cache. Otherwise, the performance may be impaired significantly. Although many good page replacement algorithms are known for single-node systems, optimal performance in a distributed setting of a shared caching system requires different algorithms.

Why is this hard? Consider an example in which a working set of 150GB is processed on 4 nodes with a capacity of 50GB each (total aggregated 200GB). For the sake of simplicity, assume that we perform uniform point lookups. Clearly, the working set should fit in the aggregated cache of the compute nodes. Consequently, no eviction to the secondary storage (i.e., storage nodes) should be triggered – in theory.

However, when every node only uses a local eviction strategy, latencies will increase, which affects the performance as illustrated in Figure 7. The main reason for this non-obvious behavior lies in the egoistic caching strategy of each node. For instance, if page *A* is cached on all 4 nodes, four slots are used instead of just one. In the extreme, if every page was cached 4 times, we could only effectively store 50 GB. This means that the nodes cannot cache all the data and thus pages may be evicted to secondary storage instead of simply dropping the cached copies. Note that the nodes do not know which copies are cached on other nodes as they only have their local view. The next time a node reads such a page from secondary storage the latency is much higher than expected. After all, it would be much faster to read from the remote caches than from secondary storage.

Research Opportunity: Altruistic Eviction. A theoretical solution was proposed 30 years ago [26, 47]. However, we are not aware of any system that implements these ideas. Even commercial systems such as Oracle RAC use a sub-optimal local strategy [7]. The main roadblock is that the proposed altruistic approaches rely on global knowledge, such as page access frequencies and the number of cached copies of each page. However, global knowledge is too expensive to maintain in high-performance distributed systems. Hence, designing a robust and adaptive heuristic could be an interesting research opportunity.

3.3 Elasticity

Pay-as-you-go pricing is the de-facto standard in the cloud. Therefore, support for compute and storage elasticity is a major competitive and operational advantage. Scaling the compute layer comes naturally in a shared-caching system since new compute nodes can join any time. However, achieving elasticity of the storage layer needs a more careful design. In the following, we sketch ideas on how to achieve elasticity in order to scale the storage layer up and down at runtime.

Research Opportunity: Elastic Storage & Caching. To achieve elasticity on the storage layer of ScaleStore, some of the state must be moved whenever a new storage node joins or old one leaves. To relocate pages in storage, consistent hashing [22] can be used. In consistent hashing, all nodes are organized in a hash ring which represents hash-ranges as illustrated in Figure 5(ii). Every node in the ring is assigned to a specific range, based on *the hash of the page identifiers*. The key idea is to leverage the storage layer in a way such that every physical storage node is placed randomly multiple times on this ring and thereby responsible for multiple of those ranges. In our example from Figure 5 we can see that Node 3 is placed twice on the ring and manages P1 and P4. Once a storage node leaves the cluster, all its pages are moved to the neighbors in the ring. For instance, when Node 3 leaves, P1 needs to be moved to Node 4 and P4 to Node 5. Conversely, whenever a new node joins it gets some of the pages from its neighbors. Clearly, using a consistent-hashing scheme provides elasticity and has been used in the past. However, while the data pages can be moved asynchronously in the background, moving the directory (and its tracking data) requires more attention. For example, an inconsistency can arise if two directories manage access to the same page and they grant exclusive access to different nodes. An open question is thus how to update the directory state at run-time.

3.4 ACID Guarantees

Currently, ScaleStore is merely a distributed storage engine that provides page-level sequential consistency but without transaction support. Therefore, let us next discuss how to support ACID transactions on top of an elastic caching-based DBMS.

Built-In Atomicity and Consistency. A key challenge of distributed DBMSs is to ensure all nodes agree on whether a particular transaction commits. In shared-nothing systems, this is achieved using a two-phase commit protocol (2PC) [15]. In a shared-caching system, every transaction becomes a local transaction since all of the transaction's data can be cached at one node. Thus, 2PC is not required [23]. Consistency is straightforward too since every compute node can validate declarative constraints locally by simply loading and accessing the data from the cache/remote nodes.

Research Opportunity: Isolation. There is a rich literature on locking-based concurrency control in shared-cache systems such as [19, 39, 43]. For instance, DEC's Rdb/VMS [19] uses a distributed lock manager with techniques to reduce the number of network messages. It uses lock de-escalation which means that processes will first acquire a lock on a large granule (e.g., a table) so as to permit operation on pages or records without additional lock requests.

Along the same lines, Rahm's Primary Copy Locking scheme [43] reduces network messages by integrating the concurrency control with an *on-request invalidation coherence scheme*. In contrast to our scheme, which invalidates as soon as they are outdated, on-request invalidation leaves outdated pages in the cache. Thus every page access must validate that the page is still up-to-date. This is achieved by comparing a cached page's version number with its version number on the storage node, and incrementing the version number on updates. Naturally, a lock request can be piggy-backed with the required validation message. E.g., when a node wants to read P1 in the cache, it sends a message with the page version and the shared-lock request to the storage node.

Mohan and Narang [39] describe LP-Locking, a technique that avoids this piggybacked message altogether by granting local lock requests as long as the page is cached on the compute node. This approach is a good fit for ScaleStore's modus operandi as pages are cached as long as there is no conflict (i.e., invalidation). However, their design works at page-granularity, not record-level, which may impair concurrency in highly parallel systems. We see two possible approaches to address this difficulty: investigate new techniques such as contention split [1], which splits pages at contention points to achieve higher concurrency despite page-level locking; or use more involved schemes based on record-level locking [38] which, however, require more network messages and may increase latency. We think an evaluation-based comparison of these approaches on modern hardware is an interesting avenue for future work.

The previous schemes rely on pessimistic CC. Another direction is to evaluate modern optimistic CC schemes [52]. Especially for ScaleStore, such schemes are interesting as they avoid all shared-memory writes for read operations. That is, no pages must be invalidated and thus no network messages must be sent at all. AWS Aurora uses an optimistic scheme where conflicts are detected during log replay on storage servers. This solution is simpler than pessimistic lock protocols. However, the delay in detecting conflicts significantly limits throughput of transactions on write-hot data, since it leads to many aborts.

Research Opportunity: Durability & Recovery. Since ScaleStore organizes data in pages, the standard ARIES-style logging and recovery mechanism [36] of disk-based DBMSs is applicable. Typically, this involves a single sequential log for all concurrent transactions. Having all nodes write sequentially to a central log is clearly not scalable. A scalable alternative is decentralized logging, where every node writes its private log file. Since every compute node can potentially touch any page, the changes for a single-page can be dispersed across multiple local logs. Consequently, in case of a failure, those log files must be merged in the storage layer, either during recovery or online at run-time. Mohan and Narang [38] describe this approach and provide ARIES-style logging for shared-cache systems. As shown by Haubenschild et al. [17], single-node ARIES performance and scalability can be improved with techniques like Remote Flush Avoidance (RFA). Therefore, we believe that existing schemes [38] should be revisited to further improve performance and scalability in a modern shared-cache system.

3.5 Cloud Infrastructure and Services

There are also many challenges and opportunities when building an OLTP DBMS that arise from the fact that come with modern cloud infrastructures and services.

Cloud Services. So far, we have assumed that the compute and storage layers are under the control of the DBMS. However, there are now many opportunities to utilize off-the-shelf cloud services instead. Those services may be more cost-effective than hand-rolled solutions and often come with high availability guarantees. For instance, AWS's object store S3 offers built-in replication. A shared-caching DBMS could use such a storage service to replace the storage layer. Of course, S3 does not offer the directory functionality, which then must be moved to the compute node or even decoupled completely into its own layer. Unfortunately, using such a proprietary service comes with its own technical challenges. For instance, S3's latency is in the tens of milliseconds which is unacceptable in latency-critical workloads such as OLTP. One could circumvent this by equipping the compute nodes with SSDs having microsecond latency to store warm data and only use S3 to read cold data or save checkpoints/log asynchronously.

Multi-Cloud Support. A further complication is that customers want multi-cloud support. Thus, when using a storage service, a DBMS must support the storage services of multiple vendors such as AWS, Microsoft, and Google. This adds code complexity not only due to different APIs but also due to different performance, availability, and cost characteristics. Another challenge is that data center network latency varies across cloud vendors. RDMA offers single-digit microseconds latency but it is only available in Microsoft Azure and only for certain virtual machine types. EFA is AWS's low-latency network offering, which has 10× higher latency than RDMA [61]. Therefore, a latency-adaptive coherence and eviction protocol is required. That is, buffer-to-buffer communication may not be faster than reading from a VM's local SSD, which means

that the coherence protocol should load and cache pages to local SSD. Conversely, the eviction should adapt for the use of the egoistic variant (cf. Section 3.2), which is inefficient if buffer-to-buffer communication is fast, but preferable if remote accesses are costly.
Hyperscaler Opportunities. Hyperscalers have many possibilities to co-design services and infrastructure. For instance, Microsoft rolled their own internal low-latency logging service called XLOG [2]. Another trend is that data center networks are equipped with programmable switches and network cards. This opens the possibility to embed the coherence protocol and conflict resolution inside the network. For instance, the directory can be placed inside the switches and thus be completely transparent to the DBMS.
Multitenancy and Workload Placement. Another exciting avenue of work is multitenancy and workload placement. A scalable system allows multiple tenants to execute their workload in the same instance, necessitating fairness in resource allocation and isolation between tenants. Of course, this is a challenging task that requires policies and strategies but enables better resource utilization. Especially coupled with workload placement which could allow pairing customer workloads that are more CPU intensive with other customer workloads which require more I/O. The flexibility of shared-caching systems allows to provision additional compute instances to avoid SLA violations. Therefore, such a system is predestined to run autonomously in the cloud, e.g., as a backend for Database-as-a-Service.

4 Summary

The title of this paper asked if scalable OLTP in the cloud is a solved problem. To approach this question, we analyzed the data access path of different distributed OLTP systems and devised a taxonomy that helps to characterize their asymptotic scalability. We found that most modern cloud systems are either based on the single-writer or the partitioned-writer paradigm. While these architectures certainly have their merits, we believe that the elasticity and scalability properties of a shared-cache design (shared-writer with coherent caches) is a better foundation for building cloud-native OLTP DBMSs. In this paper, we presented our findings and lessons learned from building such a DBMS. However, as we outlined in this paper, many interesting research opportunities must be solved to build a full-fledged shared-cache DBMS for the cloud.

Acknowledgments

This work was partially funded by the German Research Foundation priority program 2037 (DFG) under the grants BI2011/1 & BI2011/2, the DFG Collaborative Research Center 1053 (MAKI), the BMBF and the state of Hesse as part of the NHR Program. We also thank hessian.AI and DFKI for the support.

We thank Marco Slot for insights about the importance of low latency access, and the anonymous CIDR reviewers for many excellent suggestions on improving our original submission.

References

[1] Adnan Alhomssi and Viktor Leis. 2021. Contention and Space Management in B-Trees. In *CIDR*.

[2] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. 2019. Socrates: The New SQL Server in the Cloud. In *SIGMOD*.

[3] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. 2013. Tango: distributed data structures over a shared log. In *SOSP*.

[4] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *Computing Surveys* (1981).

[5] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. 2018. Efficient Distributed Memory Management with RDMA and Caching. *PVLDB* (2018).

[6] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. 2021. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *SIGMOD*.

[7] Sashikanth Chandrasekaran and Roger Bamford. 2003. Shared Cache - The Future of Parallel Databases. In *ICDE*.

[8] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* (2013).

[9] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *VLDB* (2013).

[10] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *NSDI*.

[11] Aaron J. Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, and Amr El Abbadi. 2015. Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases. In *SIGMOD*.

[12] Steve Abraham Eric Boutin. 2019. AWS re:Invent: Amazon Aurora Multi-Master: Scaling out database write performanc. https://www.youtube.com/watch?v=p0C0jakzYuc

[13] Jose M. Faleiro, Daniel Abadi, and Joseph M. Hellerstein. 2017. High Performance Transactions via Early Write Visibility. *PVLDB* (2017).

[14] Google. 2022. AlloyDB. https://cloud.google.com/alloydb

[15] Jim Gray and Leslie Lamport. 2006. Consensus on transaction commit. *TODS* (2006).

[16] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. 2017. An Evaluation of Distributed Concurrency Control. *VLDB* (2017).

[17] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. 2020. Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines. In *SIGMOD*.

[18] Yugabyte Inc. 2022. YugabyteDB. https://www.yugabyte.com/

[19] Ashok M. Joshi. 1991. Adaptive Locking Strategies in a Multi-node Data Sharing Environment. In *VLDB*.

[20] Jeffrey W. Josten, C. Mohan, Inderpal Narang, and James Z. Teng. 1997. DB2's Use of the Coupling Facility for Data Sharing. *IBM Syst. J.* (1997).

[21] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-store: a high-performance, distributed main memory transaction processing system. *VLDB* (2008).

[22] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *STOC*.

[23] Antonios Katsarakis, Yijun Ma, Zhaowei Tan, Andrew Bainbridge, Matthew Balkwill, Aleksandar Dragojevic, Boris Grot, Bozidar Radunovic, and Yongguang Zhang. 2021. Zeus: locality-aware distributed transactions. In *EuroSys*.

[24] Donald Kossmann. 2000. The State of the art in distributed query processing. *ACM Comput. Surv.* (2000).

[25] Cockroach Labs. 2022. CockroachDB. https://www.cockroachlabs.com/product/

[26] Avraham Leff, Joel L. Wolf, and Philip S. Yu. 1993. Replication Algorithms in a Remote Caching Architecture. *IEEE Trans. Parallel Distributed Syst.* (1993).

[27] Justin Levandoski. 2019. HPTS 2019: Aurora Multi-Master. http://www.hpts.ws/papers/2019/aurora-multimaster-hpts2019.pdf

[28] Feifei Li. 2019. Cloud native database systems at Alibaba: Opportunities and Challenges. *Proc. VLDB Endow.* 12, 12 (2019), 2263–2272. https://doi.org/10.14778/3352063.3352141

[29] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. 2016. Towards a Non-2PC Transaction Management in Distributed Database Systems. In *SIGMOD*.

[30] David Lomet, Rick Anderson, TK Rengarajan, and Peter Spiro. 1992. How the Rdb/VMS data sharing system became fast. *DEC Cambridge Research Lab Technical Report CRL* (1992).

[31] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: A Fast and Practical Deterministic OLTP Database. *PVLDB* (2020).

[32] Hatem A. Mahmoud, Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2014. MaaT: Effective and scalable coordination of distributed transactions in the cloud. *PVLDB* (2014).

[33] Microsoft. 2022. Azure Comsmos DB. https://azure.microsoft.com/en-us/products/cosmos-db/

[34] Umar Farooq Minhas, David B. Lomet, and Chandramohan A. Thekkath. 2011. Chimera: data sharing flexibility, shared nothing simplicity. In *IDEAS*.

[35] C. Mohan. 2022. Modern Cloud DBMSs Vindicate Age Old Work on Shared Disk DBMSs Keynote at SMBDB. https://db.cs.pitt.edu/smdb2022

[36] C. Mohan, Don Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *TODS* (1992).

[37] C. Mohan, Bruce G. Lindsay, and Ron Obermarck. 1986. Transaction Management in the R* Distributed Database Management System. *TODS* (1986).

[38] C. Mohan and Inderpal Narang. 1991. Recovery and Coherency-Control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared Disks Transaction Environment. In *VLDB*.

[39] C. Mohan and Inderpal Narang. 1992. Efficient Locking and Caching of Data in the Multisystem Shared Disks Transaction Environment. In *EDBT*.

[40] Oracle. 2022. Oracle RAC. https://www.oracle.com/de/database/real-application-clusters/

[41] M. Tamer Özsu and Patrick Valduriez. 1991. *Principles of Distributed Database Systems*.

[42] Andrew Pavlo, Carlo Curino, and Stanley B. Zdonik. 2012. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD*.

[43] Erhard Rahm. 1986. Primary copy synchronization for DB-Sharing. *Inf. Syst.*

[44] Erhard Rahm. 1991. Recovery Concepts for Data Sharing Systems. In *International Symposium on Fault-Tolerant Computing*.

[45] Amazon Web Services. 2022. Amazon DynamoDB. https://aws.amazon.com/dynamodb/

[46] Amazon Web Services. 2022. Aurora Multi-Master Documentation. https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-multi-master.html

[47] Markus Sinnwell and Gerhard Weikum. 1997. A Cost-Model-Based Online Method for Ditributed Caching. In *ICDE*, W. A. Gray and Per-Åke Larson (Eds.).

[48] Michael Stonebraker. 1985. The Case for Shared Nothing. In *HTPS*.

[49] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-Store: Fine-Grained Elastic Partitioning for Distributed Transaction Processing. *PVLDB* 8 (2014).

[50] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *SIGMOD*.

[51] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*.

[52] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *SOSP*.

[53] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *SIGMOD*.

[54] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *SIGMOD*.

[55] Xingda Wei, Sijie Shen, Rong Chen, and Haibo Chen. 2017. Replication-driven Live Reconfiguration for Fast Distributed Transaction Processing. In *USENIX*.

[56] Xiangyao Yu, Hongzhe Liu, Ethan Zou, and Srinivas Devadas. 2016. Tardis 2.0: Optimized Time Traveling Coherence for Relaxed Consistency Models. In *PACT*.

[57] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sánchez, Larry Rudolph, and Srinivas Devadas. 2018. Sundial: Harmonizing Concurrency Control and Caching in a Distributed OLTP Database Management System. *PVLDB* (2018).

[58] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Tim Harris. 2017. The End of a Myth: Distributed Transaction Can Scale. *PVLDB* (2017).

[59] Jin Zhang, Xiangyao Yu, Zhengwei Qi, and Haibing Guan. 2022. Falcon: A Timestamp-based Protocol to Maximize the Cache Efficiency in the Distributed Shared Memory. In *IPDPS*.

[60] Tobias Ziegler, Carsten Binnig, and Viktor Leis. 2022. ScaleStore: A Fast and Cost-Efficient Storage Engine using DRAM, NVMe, and RDMA. In *SIGMOD*.

[61] Tobias Ziegler, Dwarakanandan Bindiganavile Mohan, Viktor Leis, and Carsten Binnig. 2022. EFA: A Viable Alternative to RDMA over InfiniBand for DBMSs?. In *DaMoN*.