AnyDB: An Architecture-less DBMS for Any Workload

Tiemo Bang
TU Darmstadt & SAP SE

Norman May
SAP SE

Ilia Petrov
Reutlingen University

Carsten Binnig
TU Darmstadt

Abstract

In this paper, we propose a radical new approach for scale-out distributed DBMSs. Instead of hard-baking an architectural model, such as a shared-nothing architecture, into the distributed DBMS design, we aim for a new class of so-called architecture-less DBMSs. The main idea is that an architecture-less DBMS can mimic any architecture on a per-query basis on-the-fly without any additional overhead for reconfiguration. Our initial results show that our architecture-less DBMS AnyDB can provide significant speedup across varying workloads compared to a traditional DBMS implementing a static architecture.

1 Introduction

Motivation: Scale-out distributed architectures are used today by many academic and commercial database systems such as SAP HANA, Amazon Redshift / Aurora, and Snowflake [3, 10–12, 24] to process large data volumes, since these allow scaling compute and memory capacities by simply adding or removing processing nodes. The two predominant architectural models used in academic and commercial distributed databases are the shared-nothing (aggregated) and the shared-disk (disaggregated) architecture [22].

While the shared-nothing (aggregated) architecture provides high performance in case the data and workload are well partitionable, its performance degrades significantly under skew, overloading some resources while others are idle [5]. Moreover, dealing with requirements such as elasticity in the shared-nothing architecture is hard, since this always requires repartitioning the data even if compute is the bottleneck [9]. This renders the shared-nothing architecture less suited for modern environments such as the cloud where elasticity is a key requirement.

On the other hand, the shared-disk (disaggregated) architecture tackles the drawbacks of the shared-nothing architecture by disaggregating storage and compute [17, 24]. This disaggregation provides many new potentials especially for better skew handling as well as providing elasticity independently for compute and storage. Yet, the shared-disk (disaggregated) architecture has other downsides. One major downside is that data always needs to be pulled into the compute layer, resulting in higher latencies. While this additional latency often does not matter for OLAP workloads, it renders the shared-disk (disaggregated) architecture less suitable for OLTP workloads, which require low latency execution to reduce the potential of conflicts and provide high throughput.

Another observation is that these architectural models (shared-nothing or shared-disk) are statically baked into the system designs of today's databases [5], expecting certain workload characteristics.
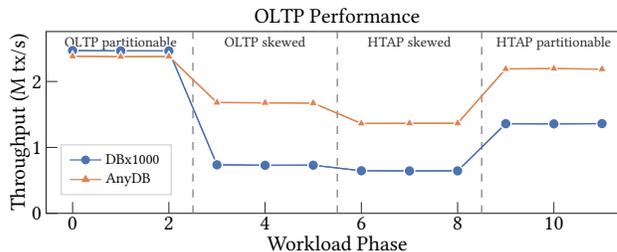
Figure 1: **Performance of AnyDB across a workload evolving from partitionable OLTP (phase 0-2), over a skewed OLTP (phase 3-5), to skewed HTAP (phase 6-8), and then to partitionable HTAP (phase 9-11). The y-axis only shows the throughput of the OLTP transactions excluding the OLAP queries in the HTAP phases.**

However, modern workloads are versatile, e.g., HTAP containing a mix of OLTP and OLAP queries [21]; even more, workloads are often evolving over time or as in the cloud are not even foreseeable for cloud providers. Consequently, databases following a static architectural model are always kind-of-a compromise and cannot provide optimal performance across a wide-spectrum of workloads.

Contributions: In this paper, we propose a radical new approach for scale-out distributed DBMSs. Instead of hard-baking an architectural model statically into the DBMS design, we aim for a new class of so-called *architecture-less DBMSs*.

The main idea of an architecture-less database system is that it is composed of a single generic type of component where multiple instances of this component "act together" in an optimal manner on a per-query basis. To instrument these generic components at runtime and coordinate the overall DBMS execution, each component consumes two streams: an event and a data stream. While the event stream encodes the operations to be executed, the data stream shuffles the state required by these events to the executing component, such that a component may act as a query optimizer at one moment for one query but for the next as a worker executing a filter or join operator. Essentially, this instrumentation of generic components by event and data streams flexibly shapes the "architecture" of an architecture-less DBMS.

A key aspect of this execution model is that by simply changing the routing of event and data streams between generic components, an architecture-less DBMS can mimic different distributed architectures and form traditional architectures as well as completely new architectures. Another important aspect is, since we decide this routing on a per-query basis, an architecture-less DBMS can simultaneously act as a shared-nothing system for one query while also acting as a shared-disk (disaggregated) DBMS for another query that runs concurrently with the first one. This opens up interesting opportunities for executing mixed workloads (e.g., HTAP) or adapting to evolving workloads.

Also, an interesting aspect of this execution model is that it cannot only mimic architectures on the macro-level (shared-nothing vs. shared-disk) but also can adapt execution strategies on the micro-level. For example, query execution in an architecture-less DBMS can mimic various query processing models at runtime (tuple-wise pipelined vs. vectorized vs. materialized [15]) and degrees of parallelism by simply instrumenting the generic components with different event and data streams. The same holds also for other components such as transaction execution and concurrency control.

The potential of the proposed architecture-less database system is shown in Figure 1. Here, we compared the performance when running an evolving workload in a static shared-nothing architecture (blue line) based on an extended version of DBx1000 [25] with AnyDB (orange line) our prototypical implementation of an architecture-less database system. As we see, AnyDB is either able to match or outperform DBx1000 depending on whether DBx1000's static architecture happens to suit a workload or not.

Outline: The remainder of this paper is structured as follows. First, in Section 2 we give an overview of how we envision an architecture-less database system. Second, in Sections 3 and 4 we then discuss the opportunities that an architecture-less database provides for OLTP, OLAP as well as HTAP and present initial experimental results in each of these sections using our prototypical architecture-less database system AnyDB. Finally, we conclude with a discussion of future directions in Section 5.

2 An Architecture-less DBMS

In the following, we first give an overview of the general execution model of an architecture-less DBMS. Then we present how typical database workloads can be mapped to this execution model and discuss the main challenges.

2.1 Overview of Execution Model

As shown in Figure 2, the main idea of an architecture-less database system such as AnyDB is that the DBMS is composed only of generic components, so-called *AnyComponents* (ACs). These generic ACs can provide any database functionality, varying over time. That is, by routing events and their required data to an AC, the AC can act as a query optimizer in one moment and in the next moment as a query executor or any other component (e.g., log writer, etc.). This gives an architecture-less DBMS the flexibility to shift its architecture just in an instant without any downtime for reconfiguration, as we discuss later.

To execute a complete SQL query (or a transaction composed of several of database operations), multiple events and data streams are routed through AnyDB from one AC to another. For example, as outlined in Figure 2 (a), when executing a query in AnyDB calling the query optimizer is one event that can trigger follow-up events for executing the operators, e.g., scans and joins. The accompanying data streams are responsible to shuffle the required state of an event to the executing AC. Next, we focus on two important key design principles that underpin the architecture-less DBMS:

- **Fully Stateless / Active Data:** ACs are designed to be fully stateless meaning that events can be processed in any AC and all state required to execute an event is being delivered to
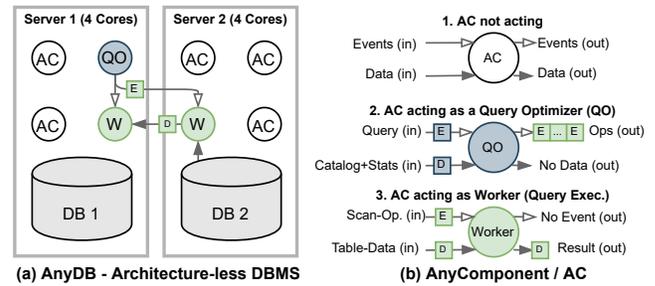

(a) AnyDB - Architecture-less DBMS   **(b) AnyComponent / AC**

Figure 2: AnyDB is an architecture-less DBMS of generic components called AnyComponents (ACs), executing arbitrary logic. ACs are instrumented by events and data streams. Depending on the incoming events an AC can act as a Query Optimizer (QO) or a Worker (W) executing a scan or a join operator or any other component, e.g., log writer, etc. An AC can also produce new data and event streams for other ACs. For example, an AC that acts as a scan operator produces a data stream with results of the scan operation.

the AC via data streams, including table data but also catalog data, statistics, and any other state. By designing ACs fully stateless, we gain a high degree of freedom as any DBMS function can be executed anywhere. This feature allows mimicking diverse architectures but also to support elasticity for all database functions individually, i.e., additional ACs can execute any DBMS function at any time. Moreover, in architecture-less DBMSs data is active, meaning that data is not pulled after an event is scheduled but it is pushed from data sources actively to the ACs before it is actually needed (We further discuss active data in Section 2.3).

- **Non-blocking / Asynchronous Execution:** A second key aspect is that ACs are executing events in a non-blocking manner. This means that an AC never waits for data of an event if data is not available yet. Instead, another event with available data is being processed. For example, a filter or a join operator is only processed once its input data, a batch of tuples, is arriving via the data stream. To provide this non-blocking execution, ACs use queues to buffer input events and data items. In addition, these queues decouple the execution between ACs as much as possible; i.e., ACs can process events asynchronously from each other. This asynchronous execution model, which is only implicitly synchronizing the execution across ACs through events and data streams, opens up many new opportunities, as we discuss later in this paper.

At a first glimpse, the execution model of an architecture-less DBMS seems to have similarities with existing approaches such as scalable stream processing systems, function-as-a-service (FaaS) or serverless DBMSs. However, there are crucial differences.

(1) First and foremost, while AnyDB also uses streams as a major abstraction, AnyDB is different from stream processing engines, since we target classical database workloads that process relations but employ streams as a vehicle to on-the-fly adapt the database architecture on a per-query basis. Still, our approach benefits from
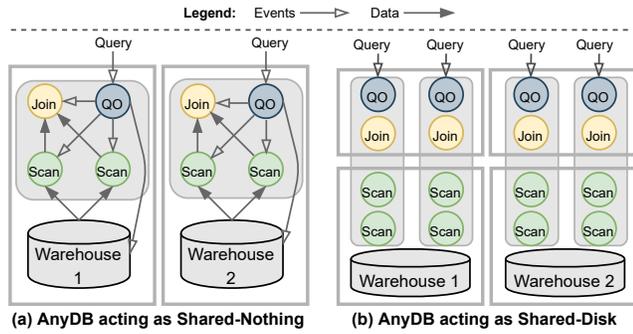
Figure 3: AnyDB can mimic diverse architectures simply by using different routing schemes for events and data streams. In (a), two servers act as a shared-nothing database while in (b) additional resources (i.e., two servers with additional 4 ACs per server) are added and AnyDB acts as a disaggregated architecture to deal with a higher query load. For simplicity, we only show the events and data streams for (a). The gray-shaded boxes around the ACs, however, indicate in (b) which ACs execute events of the same query.

techniques of scalable stream processing such as efficient data routing or for implementing fault-tolerance in AnyDB [8].

(2) Similar to function-as-a-service (FaaS), AnyDB relies on a fully stateless execution model to provide elasticity. However, in architecture-less DBMSs data (i.e., state) does not come as an after-thought. In FaaS as offered today, a function is scheduled first and then data must be pulled in from storage before the execution can actually start [20]. Instead, as mentioned before, in architecture-less DBMSs data is active, meaning that data is actively pushed from data sources to the ACs before the event is actually being processed. Moreover, while architecture-less DBMSs logically disaggregate the DBMS execution into small functions like function-as-a-service, we still allow executing events in a physically aggregated manner and also allow shipping events to the data to make use of locality.

(3) Finally, there also exist serverless DBMS offerings such as Amazon Aurora Serverless [1, 24], Snowflake Serverless, Azure SQL Database Hyperscale [2, 18]. These aim to provide elasticity (similar to FaaS), though, they still rely on static architectures and are thus restricted to very distinct workloads (e.g., OLAP only). This is very different from our architecture-less approach that can optimally adapt to a given workload and provide elasticity at the same time.

2.2 Supporting OLAP and OLTP

In the following, we give a brief overview of how the execution model above can be used to execute OLAP and OLTP workloads. Later in Sections 3 and 4, we discuss the opportunities for these workloads arising from our execution model.

Supporting OLAP: The basic flow when executing an OLAP query in an architecture-less DBMS is shown in Figure 3. The initial event is typically a SQL query that is sent from a client to any AC of the DBMS – never mind which one – which then acts as the Query Optimizer (QO). The main task of the QO is to come up with an

efficient execution plan like a traditional query optimizer in a static DBMS architecture.

In contrast to traditional optimizers, however, the QO in an architecture-less DBMS produces an event stream and initiates the data streams that instrument the ACs for query execution. Importantly, the QO also determines the routing of a query's events and data through the architecture-less DBMS. Consequently, by these routing decisions the QO defines the DBMS architecture perceived by individual queries.

For example, as shown in Figure 3 (a), if a query touches only one partition and there is moderate load in the system, then the QO can route events of a query such that the architecture-less DBMS acts as a shared-nothing architecture. However, in case the query load in the system increases, servers with additional ACs are added and the architecture-less DBMS executes queries in a disaggregated mode simply by routing events differently, as shown in Figure 3 (b).

Supporting OLTP: The basic flow of executing OLTP transactions is similar to executing OLAP read-only queries. Transactions are also decomposed into event and data streams where routing decisions define the architecture. A key difference to read-only OLAP, however, is that in OLTP (1) transactions need to update state and (2) concurrently running transactions need to coordinate their operations to guarantee correct isolation. Both these aspects are discussed below in the following (cf. *Concurrency and Updates*).

2.3 Key Challenges

There are different key challenges to enable efficient execution in an architecture-less DBMS. One of them is the optimal routing of events and data for a given workload. Another one is to handle concurrency and updates. In the following, we briefly discuss the main ideas how we aim to address these challenges. Some of these ideas are already built into our prototype AnyDB while others represent future routes of research.

Event and Data Routing: As mentioned before, a key challenge of an architecture-less DBMS is to decide how to handle a query and how to route its events, as part of query optimization. Depending on requirements of an application (e.g., latency guarantees), load in the system, and the workload, the query optimizer has to define an optimal event routing. In our current prototype, we do not focus on this problem but use an optimal decision to showcase the potential of our approach. We believe however that this is an interesting avenue for learned query optimizers.

A second challenging aspect is the efficient data routing. As mentioned before, this aspect is important for latency hiding. We utilize the decoupling of data streams from events in our execution model to solve this challenge. The main observation is that in DBMS execution one often knows which data is accessed way ahead of time before the data is actually being processed. For example, complex OLAP queries need to be optimized and compiled, often taking up to 100ms in commercial query optimizers in our experience, while we already know which tables contribute to a query before query optimization. In AnyDB we make use of this fact and initiate data streams as early as possible. Once initiated a data stream actively pushes data to the AC where, for example, a filter operator will be executed once query optimization finished. We call this feature *data*

beaming as data is often available at an AC before the according event arrives, entirely hiding latencies of data transfers. We analyze the opportunities of data beaming for OLAP later in Section 4.

Concurrency and Updates: In general, updates are supported in AnyDB by event streams directed towards the storage which ingests these events and produces acknowledgment events when the updates have been processed, as required for transaction coordination in OLTP. A major challenge in handling updates thus is to hide latencies of updates as much as possible. Again, to hide latencies of updates and decrease the overall latency of executing a transaction, operations of one transaction are represented as events and executed asynchronously by ACs. For example, an update can be sent to the storage by one AC while other (independent) operations of the transaction can progress on other ACs. Only the commit operation at the end of a transaction needs to know if the write successfully persisted and thus needs to wait for the acknowledgment event coming from the storage. As we show later in Section 3, this asynchronous model for OLTP provides many interesting opportunities and results in higher performance under various workloads.

Another challenge that is harder to solve is to efficiently handle concurrency. A naïve way would be to implement a lock manager using events representing lock operations and data streams providing the state of the lock table. A more clever way, however, is to rethink concurrency protocols and route events and data streams such that their processing order already captures the requirements of a particular isolation level for concurrency control, as we discuss later in Section 3.

Fault-Tolerance and Recovery: Fault-tolerance and recovery are two major challenges any DBMS needs to address. For an architecture-less DBMS this is a challenge due to the asynchronous (decoupled) execution of multiple ACs where individual ACs might fail.

Again, a naïve approach would be to implement standard write-ahead logging by sending log events from ACs to durable storage. For recovery the DBMS could be stopped and the log could be used to bring the DBMS into a correct state. Again, in an architecture-less DBMS we believe that we can do better and learn from the streaming community. For example, as the entire execution of a DBMS is represented as streams, another direction is to make the streams reliable, such that upon AC failure the streams (events and data) can be rerouted to another AC [8]. Applying these ideas is again an interesting avenue of future research.

3 Opportunities for OLTP

In the following, we discuss the various opportunities emerging from an architecture-less DBMS when executing OLTP workloads and show initial results when compared to existing execution models of static architectures (shared-nothing and shared-disk). For all initial experiments in this paper, we use the two dominant transactions of the TPC-C benchmark (i.e., payment and new-order) [23].

3.1 Opportunity 1: Duality of Disaggregation

As indicated earlier in Figure 1, for partitionable OLTP workloads an architecture-less DBMS can achieve nearly the same throughput as an (aggregated) shared-nothing architecture. Key to this is the duality of disaggregation in the architecture-less DBMS. The
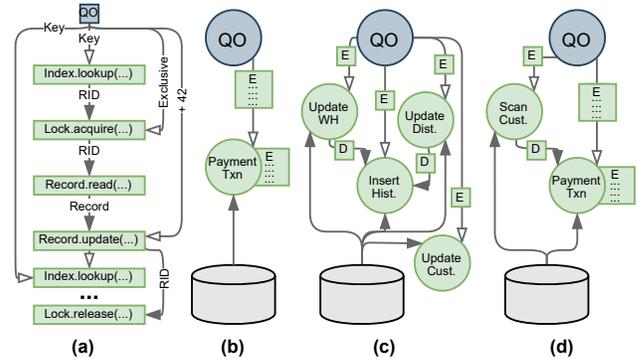

Figure 4: Duality of Disaggregation. (a) shows how a transaction is logically disaggregated into individual events for each operation. (b) shows physically aggregated execution of events by routing the stream to one AC. (c) shows event routing for *fully* intra-transaction parallel execution. (d) shows *balanced* intra-transaction parallel execution.

architecture-less DBMS distinguishes logical disaggregation of the DBMS design and physical disaggregation of the DBMS execution. Logically, the DBMS is entirely disaggregated into independent fine-grained functionality interacting via events and data streams. However, while the logical execution is disaggregated into many small events, physically the events can still be executed in an aggregated manner if desired. This opens up the opportunity to achieve high data locality if desired as all events can be executed close to the data, e.g., a partition of the database.

For example, an OLTP transaction may consist of an event stream like in Figure 4 (a). Yet, this logical disaggregation does not mandate disaggregated execution. In the contrary, any sub-sequence of these events can be physically aggregated and executed by a single AC. As shown in Figure 4 (b), the entire event stream of a transaction can be executed by a single AC. In fact, this physical aggregation of events establishes a shared-nothing architecture that performs on par with the static shared-nothing architecture of DBx1000 as shown earlier in Figure 1.

The Gist: Through the duality of logical vs. physical disaggregation, we believe that an architecture-less DBMS can efficiently mimic diverse architectures ranging from entirely aggregated shared-nothing to fine-grained disaggregated as required, simply shifting between those by adapting event and data routes.

3.2 Opportunity 2: Execution Strategies

Along with the freedom of achieving different architectures on the macro-level, the execution model in an architecture-less DBMS also provides broad freedom to layout parallel execution strategies in an optimal manner on the micro-level.

Generally, as explained earlier, transactions are represented as event streams flowing through the architecture-less DBMS. Importantly for transaction execution, this event-based execution allows diverging from typical execution models in OLTP that aim for *inter*-transaction parallel execution and allows investigating also other forms of parallelism. For example, event-based execution naturally

brings opportunities to ad hoc parallelize execution within a single transaction to achieve *intra*-transaction parallelism, especially when contention prohibits inter-transaction parallel execution.

The efficiency of this freedom to change the transaction execution on the micro-level becomes also visible in Figure 5. When running a partitionable OLTP workload in the first phase (0-2), AnyDB mimics not only a shared-nothing architecture but also uses classical inter-transaction parallelism. Afterwards it uses intra-transaction parallelism in the second phase (3-5) for the highly skewed, contended OLTP workload, where 100% of TPC-C payment transactions operate on one warehouse only.

The baseline DBx1000 in this experiment uses a shared-nothing model and is thus bound by the resources that are assigned to one partition resulting in 0.7 M txn/s. In such a case, our architecture-less DBMS allows to simply shift from inter- to intra-transaction parallelism by routing events of a single transaction to several ACs, i.e., shifting from Figure 4 (b) to (c). Such intra-transaction parallel execution may accelerate transactions in contended OLTP workload, as already proven by architectures such as DORA [19].

However, just like any design decisions in a static architecture, also static intra-transaction parallelization does not always prove beneficial either. For example, Figure 5 shows that with naïve intra-transaction parallelism, where every independent operation of a transaction is farmed out to a different AC, AnyDB only achieves 0.8 M txn/s (orange squares), barely exceeding the inter-transaction parallel execution of the DBx1000 baselines (blue lines). The main reason is that the overhead of parallelizing within a transaction dominates the execution.

The architecture-less DBMS addresses the challenging parallelization of transactions in the following way: generally, the representation of transactions as event streams allows the architecture-less DBMS to route independent sub-sequences of events (i.e., sub-sequences of operations) to multiple ACs for parallel execution. Thus, the challenge in an architecture-less DBMS is to split a transaction into suitable sub-sequences of events and route them to different ACs, balancing the amount of work versus overhead. In our experiments, for example, we partition the TPC-C payment transaction into one sub-sequence with several brief update statements and a second sub-sequence with a long range scan, as depicted in Figure 4 (d).

Finding an optimal splitting and routing of event sequences of transactions depends on many factors. One important point is that the individual sub-sequences of operations have similar execution latencies, such that the overall latency of the transaction is minimized. Finding such an optimal routing of events is an opportunity for learning query optimization and scheduling, as mentioned before. As shown in Figure 5, with this balanced intra-transaction parallelization AnyDB achieves 1.2 M txn/s (orange triangle) with only 2 ACs, outperforming the baseline (blue lines) and AnyDB's naïve parallelization (orange square) by 3.2x and 3x in throughput per thread, respectively.

The Gist: Generally, we envision, that the freedom of the execution models on the micro-level in an architecture-less DBMS can enable new parallel transaction execution models spanning any design between pure inter-transaction to aggressive (fine-grained) intra-transaction parallelism on a per-transaction level.
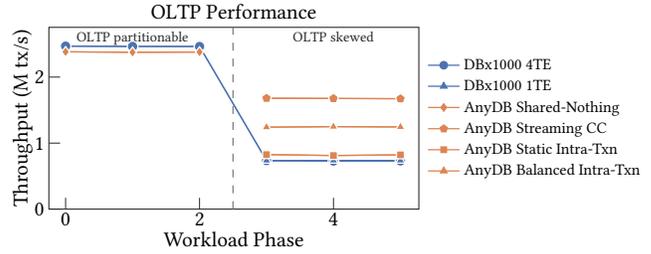

Figure 5: OLTP performance of AnyDB versus the shared-nothing DBx1000 under partitionable and skewed OLTP. In phases 0-2 AnyDB acts as a shared-nothing DBMS using an inter-transaction parallel execution model while in phases 3-5 AnyDB acts as a shared-disk DBMS using an intra-transaction parallel execution model. Note that for DBx1000, 4 transaction executors (TEs) perform like a single TE due to high contention between transactions.

3.3 Opportunity 3: Concurrency Control

In OLTP workloads, especially under high contention, concurrency control (CC) causes significant coordination effort and challenges efficient parallelization [4]. In an architecture-less DBMS, the event-based nature of transactions provides the opportunity to transform CC to a streaming problem. Thereby, the architecture-less DBMS can improve the efficiency of traditional CC schemes and opens many opportunities for novel CC schemes, as discussed next.

Transforming Traditional Concurrency Control: Interestingly, many traditional CC protocols are stream-like already and thus benefit from a direct mapping to the asynchronous (non-blocking) execution model of the architecture-less DBMS. For example, a pessimistic lock-based CC scheme [6] needs to match incoming lock requests with its lock state. This can be mapped to a streaming join on an event stream containing lock requests and a data stream containing the lock state of the requested item. Similarly, verification in optimistic CC protocols [16] joins the read/write set of a transaction which is one data stream with the current state of the database which is another data stream. Despite these benefits for traditional CC protocols, the architecture-less DBMS offers opportunities for novel coordination-free CC schemes vastly outperforming the traditional approaches, as explained in the following.

Novel Streaming Concurrency Control: The key idea of rethinking CC schemes is that they can be enabled by efficiently ordering events of (conflicting) transactions flowing through the architecture-less DBMS, rather than actively synchronizing execution of concurrent transaction using traditional CC schemes causing high coordination overhead especially under high contention. Here, the streaming execution of transactions brings new opportunities despite high contention.

Concurrency control in AnyDB can be implicitly and coordination-free encoded into event routes. That is, for consistency of concurrent transactions it suffices to route their events in a consistent order through ACs which execute the conflicting operations. For example, considering two TPC-C payment transactions accessing the same warehouse, AnyDB can guarantee consistency by simply
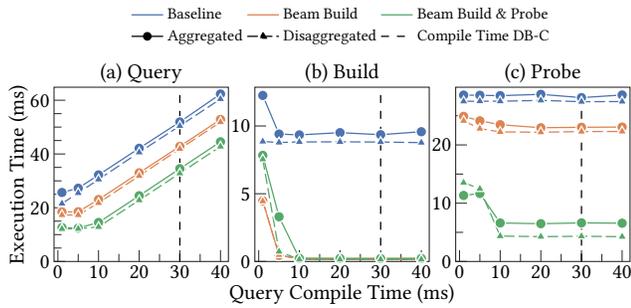
Figure 6: Data beaming can effectively shorten query execution for disaggregated execution of OLAP workloads and in the best case hide latencies of data shuffling completely.

routing their events to all involved ACs in the same order. Thereby, AnyDB enables intra-transaction parallelism, routing independent events through different ACs and also provides CC without the need to actively synchronize operations at the same time.

Note that event-ordering does not violate our non-blocking (asynchronous) execution model of ACs. Operations (i.e., events) of conflicting transactions simply remain in the ACs' input queues for ordering while other events can still be executed. In Figure 5, we see that this instantiation of AnyDB called *streaming CC* (orange pentagon), yields 1.7 M txn/s for TPC-C payment under high contention (phases 3-5). This is much closer to the performance of the uniform (partitionable) execution of TPC-C payment in phases 0-2.

The Gist: Along the properties of event streams, we envision novel CC protocols, that avoid active synchronization, as discussed. Moreover, the streaming CC enables new directions where events are gradually rerouted depending on the load, e.g., at first all events for a specific transaction are routed to a single AC until this AC becomes overloaded. Then AnyDB may transparently repartition event streams while still guaranteeing consistent event order.

4 Opportunities for OLAP and HTAP

Previously, we have described that the execution model of an architecture-less DBMS provides many opportunities for OLTP. In the following, we discuss further opportunities for OLAP as well as mixed HTAP workloads.

Especially for OLAP, operations encoded in the event streams are data intensive (e.g., a join of two large tables). Therefore, data streams must efficiently bring data to wherever events are executed as if data access was local to facilitate the architecture-less DBMS. While this aspiration of "omnipresent" data appears challenging, we observe that in DBMSs one often knows data to be accessed way ahead of time before actually processing it. For example, in OLAP data is only accessed and processed after several milliseconds of query optimization and compilation. Hence, we propose *data beaming*, a technique initiating data streams early and pushing data to ACs where events will be executed.

In the following experiment, we demonstrate the effect of data beaming with a simple OLAP query. Based on CH-benCHmark Q3 [7], our query reports all open orders for all customers from

states beginning with "A" since 2007 via 3 (filtered) scans and 2 joins. In Figure 6, we display the effect of data beaming in several degrees for this query: (1) In blue, the baseline does not utilize beaming, but instead passively pulls in data when needed. (2) In orange, the build sides are beamed during query compilation, then joins are executed. (3) In green, build and probe sides are beamed.

To detail the efficiency of data beams, we implement two variants: one where data beams only need to shuffle locally (not over the network) and one where they shuffle data across the network. The solid lines in Figure 6 demonstrate the runtime using local beaming via shared-memory queues [14] (e.g., to hide NUMA latencies) and the dashed lines show the beaming across the network for a disaggregated architecture using DPI-flows [13]. On the x-axis as reference point for query compile time, 30ms marks the time taken by a commercial DBMS (DB-C) to compile this query.

Figure 6 (a) shows that the overall query execution time with beaming is only slightly higher than query compile time (green line), whereas without beaming (blue line) the query execution time has additional latency of 20ms, since data transfer is not overlapped with query compile time. In detail, Figures 6 (b) and (c) show the individual effects of data beaming on the build and probe side (without query compilation overhead), respectively. We see that beaming can reduce the runtime of the smaller build side almost to 0ms. For the larger probe side, beaming also reduces the runtime from 30ms to less than 10ms. Notably, the disaggregated architecture (which needs to shuffle data across the network) performs even better than the aggregated architecture, as DPI offloads event and data transfers to the InfiniBand NICs acting as a co-processor in AnyDB.

The previous experiment demonstrated the utility of data beaming to hide data transfer latencies in OLAP workloads. Besides hiding transfer latencies, data beaming can also be used to achieve other goals such as resource isolation, e.g., for HTAP workloads. The idea is that in HTAP workloads, we can use data beams to route data intensive analytical queries to additional compute resources disaggregated from storage while latency-sensitive transactions are executed close to the data.

The HTAP workloads in Figure 1 (phase 6-11) outline such scenarios, where the OLAP query of the previous experiment is executed in parallel to the OLTP workload. Here, AnyDB executes the OLAP query independently of the OLTP workload, only sharing storage resources, whereas the OLAP query in the static DBx1000 uses the same transaction resources for OLAP queries as for the OLTP workload. Thereby, AnyDB simultaneously provides higher OLTP and OLAP performance than DBx1000.

The Gist: In general, we envision AnyDB to establish flexible architectures through described data beaming as well as optimal per query/transaction event routing, opening up new paths to hybrid architectures and supporting various types of deployments.

5 Conclusions and Future Directions

In this paper, we have proposed architecture-less DBMSs, a radical new approach for scale-out distributed DBMSs. In addition to the discussed opportunities, we see many further interesting research directions arising from the architecture-less approach:

Elasticity for Free: Flexible event routing and data beaming open up opportunities, apart from resource isolation, for transparent elasticity without additional latencies. Considering events are self-contained and state is always beamed, elasticity for execution of event streams just means consistent routing of events and their state to an elastic number of ACs. Even more, as mentioned before since all events and data are both delivered as streams to ACs, these streams could be repartitioned or rerouted to distribute load in the system adaptively.

Transparent Heterogeneity: The stateless execution model in conjunction with the opportunity for elasticity further facilitates transparent (ad hoc) integration of heterogeneous compute resources per query, including but not limited to accelerators (e.g., FPGAs or GPUs) and programmable data planes (e.g., programmable NICs or switches). Moreover, since events fully describe what to do and data streams deliver all required state, event execution (*how to do it*) can be specialized for FPGAs, etc. without any impact nor dependencies on the rest of the DBMS.

Crossing Clouds and More: Finally, data beaming is an interesting concept generally hiding data transfer cost not only within but also across data centers. This opens up opportunities for DBMS deployments across on-premise, cloud offerings, and the edge without paying significant latencies for data transfer. For example, an architecture-less DBMS for HTAP workload may run transactions for daily business on-premise and may ad hoc beam data to cloud resources for sporadic reporting, combining the benefits of both platforms efficiently.

References

[1] Amazon Web Services. Amazon Aurora Serverless, 2020. https://aws.amazon.com/rds/aurora/serverless/, accessed 2020-11-10.
[2] P. Antonopoulos, A. Budovski, C. Diaconu, A. H. Saenz, J. Hu, H. Kodavalla, D. Kossmann, S. Lingam, U. F. Minhas, N. Prakash, V. Purohit, H. Qu, C. S. Ravella, K. Reisteter, S. Shrotri, D. Tang, and V. Wakade. Socrates: The new sql server in the cloud. In *SIGMOD'19*. ACM, 2019.
[3] J. Appleby. SAP HANA - scale-up or scale-out hardware? https://blogs.saphana.com/2014/12/10/sap-hana-scale-scale-hardware/, accessed 2020-06-20.
[4] T. Bang et al. The tale of 1000 cores: An evaluation of concurrency control on real(ly) large multi-socket hardware. In *DaMoN'20*. ACM, 2020.
[5] T. Bang, I. Oukid, N. May, I. Petrov, and C. Binnig. Robust performance of main memory data structures by configuration. In *SIGMOD'20*. ACM, 2020.
[6] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2), June 1981.
[7] R. Cole, F. Funke, L. Giakoumakis, W. Guy, A. Kemper, S. Krompass, H. Kuno, R. Nambiar, T. Neumann, M. Poess, K.-U. Sattler, M. Seibold, E. Simon, and F. Waas. The mixed workload ch-benchmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems*. ACM, 2011.
[8] B. Del Monte, S. Zeuch, T. Rabl, and V. Markl. Rhino: Efficient management of very large distributed state for stream processing engines. In *SIGMOD'20*. ACM, 2020.
[9] A. J. Elmore, V. Arora, R. Taft, A. Pavlo, D. Agrawal, and A. El Abbadi. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *SIGMOD'15*. ACM, 2015.
[10] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA database–an architecture overview. *IEEE Data Eng. Bull.*, 35(1), 2012.
[11] A. K. Goel, J. Pound, N. Auch, P. Bumbulis, S. MacLean, F. Färber, F. Gropengiesser, C. Mathis, T. Bodner, and W. Lehner. Towards scalable real-time analytics: an architecture for scale-out of OLxP workloads. *Proc. VLDB Endow.*, 8(12), 2015.
[12] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan. Amazon Redshift and the case for simpler data warehouses. In *SIGMOD'15*. ACM, 2015.
[13] A. Gustavo, C. Binnig, I. Pandis, K. Salem, J. Skrzypczak, R. Stutsman, L. Thostrup, T. Wang, Z. Wang, and T. Ziegler. DPI: the data processing interface for modern networks. In *CIDR*, 2019.
[14] B. Hu and J. DeLong. Folly single-producer-single-consumer queue, 2019. https://github.com/facebook/folly/blob/d2c64d94c7e892925a02a080c886ab3df3f5c937/folly/ProducerConsumerQueue.h, accessed 2020-11-10.
[15] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11(13), Sept. 2018.
[16] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2), June 1981.
[17] S. Loesing, M. Pilman, T. Etter, and D. Kossmann. On the design and scalability of distributed shared-data databases. In *SIGMOD'15*. ACM, 2015.
[18] Microsoft. Hyperscale service tier, 2020. https://docs.microsoft.com/en-us/azure/azure-sql/database/service-tier-hyperscale, accessed 2020-11-10.
[19] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *Proc. VLDB Endow.*, 3(1), 2010.
[20] M. Perron, R. Castro Fernandez, D. DeWitt, and S. Madden. Starling: A scalable query engine on cloud functions. In *SIGMOD '20*. ACM, 2020.
[21] I. Psaroudakis, F. Wolf, N. May, T. Neumann, A. Böhm, A. Ailamaki, and K.-U. Sattler. Scaling up mixed workloads: a battle of data freshness, flexibility, and scheduling. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 2014.
[22] M. Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1), 1986.
[23] The Transaction Processing Council. TPC-C benchmark (revision 5.9.0), 2007. http://www.tpc.org/tpcc/spec/tpcc_current.pdf.
[24] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon Aurora: Design considerations for high throughput cloud-native relational databases. In *SIGMOD'17*. ACM, 2017.
[25] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, 8(3), Nov. 2014.