Towards a Shared-storage-based Serverless Database Achieving Seamless Scale-up and Read Scale-out

Yingqiang Zhang, Xinjun Yang, Hao Chen*, Feifei Li
Jiawei Xu, Jie Zhou, Xudong Wu, Qiang Zhang
Alibaba Group
{*yingqiang.zyq, xinjun.y, ch341982, lifeifei, leroy.xjw, jarry.zj, xiangzhong.wxd, jingrui.zq*}*@alibaba-inc.com*

***Abstract*—The serverless database has recently attracted increasing attention both in industry and academia due to its high elasticity and the "pay-as-you-go" model. This paper delivers a thorough review of current shared-storage-based commercial serverless databases, pinpointing two major challenges: (1) they either experience difficulties with instance migration during scaling up or restrict the resource usage within a single physical host to avoid potential migration. (2) they lack the ability to scale out secondary nodes due to the absence of strong consistency support in secondary nodes. Based on our experience in building serverless databases, this paper proposes two fundamental requirements to address these two issues: seamless and instant migration and read scale-out. The former allows for instance migration when there are insufficient resources on the resident host during scaling up without application disruption, whereas the latter necessitates strong consistency on secondary nodes to process read requests.**

To fulfill these fundamental requirements, we propose PolarDB Serverless, a shared-storage-based serverless database achieving seamless scale-up and read scale-out. It supports read scale-out by inheriting the strong consistency feature from PolarDB, making it possible to process strongly consistent reads on secondary nodes. In the pursuit of achieving seamless migration, PolarDB Serverless introduces a transaction migration policy. It ensures there is no interruption to the application during migrations, allowing transactions to continue on the new instance without any disruptions. It also minimizes the overhead of migration, achieving a fast migration. In our evaluation, especially in the context of database migration scenarios, it's noteworthy that the migration of a database instance takes just half a second without causing any exceptions for applications. PolarDB Serverless is the first shared-storage-based serverless database supporting both seamless scale-up and read scale-out and is already commercially available at Alibaba Cloud.

***Index Terms*—cloud-native database, serverless**

I. Introduction

Cloud-native OLTP databases have emerged as vital infrastructure components for modern applications [12], [28], [37]. Most of them employ either a shared-storage architecture (such as AWS Aurora [36], Alibaba PolarDB [28] and Hyperscale [17], [27], etc) or a shared-nothing architecture (such as CockroachDB [34], TiDB [21] and OceanBase [38], etc). In recent times, the trend has been towards serverless databases, with an emphasis on delivering high elasticity and a "pay-as-you-go" model [23], [30]. A serverless database can grow automatically in fine-grained increments to meet the changing demands of an application and manage unexpected workloads that cannot be predicted or scheduled [20]. Deploying a serverless database enables users to pay only for the resources they consume, resulting in significant cost savings.

Several serverless databases have already been proposed based on shared-nothing, shared-storage, or even a novel database architecture (such as shared-memory [12]). However, since shared-storage-based databases continue to be popular and widely adopted by many customers, this paper primarily focuses on shared-storage-based databases. This paper reviewed the currently available shared-storage-based commercial serverless databases and highlighted two critical issues in them: (1) either experience difficulties with instance migration during scaling up or restrict the resource usage within a single physical host to avoid potential migration. (2) lack the ability to scale out secondary nodes due to the absence of strong consistency support in secondary nodes. In response to these issues, this paper proposes two fundamental requirements for a shared-storage-based serverless database:

(1) Seamless migration. Seamless migration is a critical aspect of the scaling-up process, involving the allocation of additional vCores. In scenarios where the current physical host lacks the necessary free resources to accommodate such scaling for a resident database instance, the database instance must either be migrated to another physical host or prevented from further scaling up. Failing to migrate the instance, as seen in some databases, violates the fundamental principle of serverless, which involves adapting allocated resources to dynamic workloads. Nonetheless, migrating the instance typically incurs a brief period of downtime. These migrations typically occur during their peak activity periods, leading to application errors due to unsuccessful transaction execution during the downtime. To avoid migration during scaling up, some serverless databases reserve excessive resources on the physical host, allowing scaling up to occur within the resident host. This approach makes the physical host's resources underutilized. It's akin to a provisioned deployment, shifting the reservation from the user side to the back-end side, ultimately resulting in increased costs for the cloud provider.

(2) Read scale-out. Shared-storage-based cloud-native databases typically consist of a single primary node responsible for handling read/write requests and one or more secondary nodes dedicated to processing read requests. However, many of

*Hao Chen is the corresponding author.

these databases face the challenge of returning stale data from secondary nodes due to asynchronous log shipment and log application processes. As a result, serverless versions of such databases often offer two endpoints. One endpoint (associated with the primary node) ensures strong consistency, while the other (linked to secondary nodes) provides only eventual consistency. This means that strongly consistent reads must be directed to the primary node, and the ability to scale out secondary nodes cannot assist in this context. When the read workload becomes intensive, it has to scale up the primary node, potentially impacting performance when involving migration. Additionally, most serverless databases impose limits on the maximum resource an instance can use. This further constrains performance and restricts resource utilization to a single physical host. On the other hand, the read-dominant workloads are very common [9], [15], [35], [37] today, making it critical to achieve the read scale-out in a serverless database.

In this work, we propose PolarDB Serverless, which is built upon shared-storage architecture, addressing the aforementioned limitations. PolarDB Serverless is the first shared-storage-based serverless database that supports both seamless cross-machine scale-up and read scale-out. It leverages PolarDB's strong consistency feature [37] to ensure strong consistency on secondary nodes. Consequently, PolarDB Serverless can provide a unique strongly consistent endpoint for users, enabling the ability to scale out secondary nodes for processing strongly consistent reads.

Crucially, PolarDB Serverless supports seamless auto-scaling. During the process of the migration, application connections remain consistently active, with the migration procedure entirely transparent to the application. The active transactions are seamlessly migrated from the old instance to the new one, without any interruptions. From the application's perspective, it continues to operate as if it were always connected to the same backend instance, ensuring uninterrupted availability. To achieve this, we propose connection maintenance and transaction migration policies within PolarDB Serverless. The connection maintenance policy relies on the proxy node's capabilities. During migration, the proxy node retains the application's connections and creates new connections to the new instances, subsequently remapping these new connections to the applications' connections. From the application's perspective, its connections are always active, and it does not encounter any exceptions but experiences a higher latency during the migration. The key design of transaction migration is to migrate the in-memory data of the old instance (*e.g.*, redo/binlog logs and some transaction-related metadata) to the new instance. PolarDB Serverless synchronize the in-memory data (*e.g.*, redo logs) at the query level to the new instance during migration via the RDMA network. After switching to the new instance, it only needs to roll back the latest unfinished query, re-execute it on the new instance and respond to the application. This process is totally transparent to applications. The applications only see a slightly higher latency for the current query, but will not receive any exceptions. The RDMA-based data transfer further speeds up the process of migration, minimizing the overhead.

Furthermore, the seamless migration design holds potential for various scenarios. It can facilitate transparent database upgrades by seamlessly migrating instances to new versions without interrupting applications. Additionally, it contributes to fast failover capabilities. In the event of a primary node failure, the secondary node can quickly assume control and continue processing active transactions, all without the application's awareness of the backend crash. Notably, both of these features have already been successfully implemented in PolarDB as part of recent releases.

We summarize our main contributions as follows:

- We reviewed some commercial serverless database designs and revealed two fundamental requirements that have been lacking in current serverless databases: seamless migrations and read scale-out.
- We propose a novel policy to implement a fast seamless migration of database instances without any interruptions.
- We design and implement PolarDB Serverless, which is the first shared-storage-based serverless database supporting both seamless instant cross-machine scale-up and read scale-out. It is already commercially available at Alibaba Cloud.
- We conducted a comprehensive evaluation of PolarDB Serverless in various scenarios, demonstrating its remarkable elasticity.

This paper is structured as follows. First, we present the background in Section II and show our lessons learned from build PolarDB Serverless in Section III. Then we provide PolarDB Serverless's overview and detailed implementation in Section IV and Section V. Next, we evaluate PolarDB Serverless in Section VI and review the related works in Section VII. Finally, we conclude the paper in Section VIII.

II. Background

A. Shared-storage-based OLTP databases

1) Architecture: Shared-storage-based cloud-native databases, including popular solutions like AWS Aurora [36], Azure Hyperscale [17], [27] and Alibaba PolarDB [28], have gained significant traction and widespread adoption among users today. Fig. 1 illustrates the typical architecture of a shared-storage-based cloud-native OLTP database. This architecture typically features a primary node and one or more secondary nodes. The primary node is responsible for handling both read and write requests, while the secondary nodes only process read requests that do not require strong consistency. If the current primary node fails, one of the secondary nodes can be promptly promoted to take its place. Both primary and secondary nodes are monolithic database instances equipped with traditional components such as SQL parsing, transaction processing, buffer pools, and more. However, unlike conventional monolithic databases, the primary and secondary nodes share disaggregated cloud storage, ensuring fault-tolerance and data consistency. This design eliminates the need for additional storage overhead

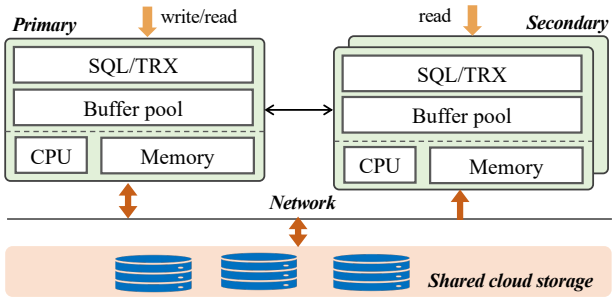

Fig. 1. The architecture of a typical shared-storage-based OLTP databases

when adding more secondary nodes, which is different from conventional primary-secondary-based database clusters, where each node maintains its own storage. Moreover, the disaggregated shared storage approach eliminates the need for data migration when migrating the database instance. Some of these databases also deploy a proxy node atop the primary and secondary nodes to facilitate load balancing, manage failover, control access, and deliver additional functionalities.

2) Stale reads on secondary nodes: Both the primary and secondary nodes maintain their respective buffer pools to enhance performance. However, since updates only happen on the primary node, the secondary node's buffer pool becomes inconsistent with the primary node's. To address this, the primary node generates corresponding logs for each update, transmits these logs to the secondary node, and subsequently, the secondary node applies these logs to update its in-memory data. For the sake of performance, the log shipment and application processes are asynchronous. Consequently, the secondary node might return stale data [37]. In such instances, the secondary nodes can only process read requests that can tolerate eventual consistency. Meanwhile, read requests requiring read-after-write consistency must be processed on the primary node. It's essential to note that this limitation is common in many shared-storage-based databases [37], such as Aurora and Azure SQL Database.

3) PolarDB: PolarDB [28] is a widely adopted cloud-native database based on disaggregated shared storage, consisting of one primary node and one or more secondary nodes. The implementation of shared storage is carried out through the utilization of PolarStore [11]. PolarDB introduces a proxy node that operates atop the primary and secondary nodes, serving several essential functions. These include read/write splitting, load balancing, connection management, failover handling, access control, and various other functionalities. PolarDB proxy typically has two active nodes for high availability. Different from other cloud-native databases, PolarDB establishes communication, such as log shipment, timestamp fetching, and other message exchanges, between primary and secondary nodes through the employment of the RDMA network. PolarDB has recently released the strong consistency feature [37] that achieves strongly consistent reads on secondary nodes with low latency. PolarDB Serverless piggybacks on this design to enable the scale-out of secondary nodes.

B. Other databases

1) Shared-nothing architecture: Another prevalent approach in database architecture is the shared-nothing architecture. Unlike the shared-storage architecture, shared-nothing architecture involves partitioning the entire database. In this setup, each node operates independently and maintains its dedicated storage. A node can only access data within its specific partition. When a transaction spans multiple partitions, it requires the use of cross-partition distributed transaction mechanisms, such as the two-phase commit policy. Scaling out the cluster in shared-nothing architecture, by introducing a new node, demands the repartitioning of the database. As part of this process, some data belonging to the new node must be migrated to the newly added node. The duration of data migration can be quite lengthy [10], depending on the total size of the database. Consequently, scaling out in shared-nothing-based databases often involves a time-consuming rebalancing procedure. This approach is not suitable for a serverless database that needs to scale out rapidly.

2) Some novel approaches: Recently, some academic approaches have explored the integration of shared memory into databases to enable elastic memory usage [12], [31], [39]. Developing a serverless database based on these designs differs from the conventional cloud-native database architecture, as the shared-memory pool offers highly flexible memory allocation and deallocation. Additionally, the secondary node can offer strong consistency through shared memory. Moreover, there are some academic trials that develop a multi-primary database based on shared storage, providing the potential to build a serverless database with multiple primary nodes. However, these proposals are still not widely adopted by users, and some of them are still in academic trials.

Considering the continued popularity and extensive adoption of shared-storage-based databases, such as AWS Aurora, Azure Hyperscale, and PolarDB, among various customers, this paper primarily focuses on serverless databases built upon shared-storage-based architectures.

C. Database logs

1) Redo/undo log: ARIES [29] is widely adopted by many databases for the purpose of rolling back uncommitted transactions and recovering committed transactions following a database crash. In the ARIES-style logging technique, each update will generate corresponding redo and undo logs. Typically, the redo log records the changes in the physical data, while the undo log maintains the previous value of a record. Importantly, the changes of undo logs also lead to the generation of corresponding redo logs. For performance consideration, many databases implement the "no-force" policy, which only forces the associated redo logs to the storage when committing a transaction and does not require the modified data to be flushed to the storage. Due to the limited buffer size, many database systems allow modified data to be written to storage before the transaction commits, a strategy known as the "steal" policy. Typically, if a modified page is going to be flushed to the storage, its corresponding redo logs (including

those for undo data) must be persistent in the storage before flushing the page. In this case, for any committed transaction, its redo log must be persistent in the storage. For the uncommitted transaction, if its modified data is committed, the page's related redo logs must be persistent in the storage. Therefore, during the recovery after a crash, the database can first apply all redo logs to recover all modified data (including undo data) that have not been flushed to storage before the crash. After applying the redo logs, committed transactions can be recovered. Subsequently, the database applies the undo logs to further roll back uncommitted data.

2) Binlog: The binary log (*binlog*) is an important feature in MySQL, serving as a record of all the modifications made to the database. It is used for a variety of workloads such as crash recovery and synchronization for replicas. It is essential to recognize that redo logs and binlogs are independent and separately stored, with the binlog not being stored in or relying on the Write-Ahead Logging (WAL) mechanism. In modern cloud-native MySQL-variant databases like Aurora and PolarDB, the binlog is no longer necessary for database recovery or secondary node synchronization. However, these databases still maintain the binlog feature for various other functionalities, such as changed data capture, auditing, and facilitating analytics. A significant number of PolarDB users opt to enable the binlog feature, underscoring its continued importance in existing cloud-native databases. The binlog is initially stored in a thread's local cache and is only appended to the global binlog file and synchronized with storage during committing a transaction. However, in the serverless databases, when migrating an active transaction to a new instance, the recovery of the binlog resident in the old instance's cache becomes a challenging aspect in ensuring seamless migration.

III. Lessons

This paper reviewed the currently available shared-storage-based serverless databases and highlighted two critical issues in them: (1) they either experience difficulties with instance migration during scaling up or restrict the resource usage within a single physical host to avoid potential migration. (2) they lack the ability to scale out secondary nodes due to the absence of strong consistency support in secondary nodes. Based on the lessons learned from building PolarDB Serverless, we proposed two fundamental requirements for a serverless database.

A. Seamless migration requirement

1) Database migration: The typical shared-storage-based databases only support single primary node. So, when the primary node's workload becomes heavy, it has to scale up by allocating more resources. However, there are some situations in which the available resources on the current physical host are not enough for the resident instances to be scaled up. In this case, it must migrate this instance to another host that has enough free resources or stop to continue scaling up. Avoiding scaling up in this situation violates the fundamental principle of serverless that dynamically allocates resources according to the changing workloads and limits the resource usage within a single physical host. Finally, it causes significant negative impacts on the applications. However, on the other hand, migrating the instances will induce a negative impact.

The process of migrating a database instance from one host to another necessitates several steps, including halting the old instance and re-establishing connections for applications to the new instance. In the context of serverless databases, these migrations typically occur during scaling up, often prompted by a shortage of available resources on the current host. This situation usually arises when the application's workload intensifies, resulting in a continuous stream of requests directed at the old instances. Consequently, halting the old instance severs these existing connections, abruptly terminating active transactions. This results in application errors indicating unsuccessful transaction execution, compelling users to retry their operations at a later time. Upon launching the new instance, it must recover the data that hasn't yet been flushed to storage before the old instance's shutdown. Additionally, it needs to roll back any aborted transactions. These processes inevitably introduce a brief period of downtime, and if a long-running transaction is in progress on the old instance, the downtime can be significantly extended.

2) Impact of migration: Database migration occasionally occurs during scaling-up, and some serverless databases even impose limits on the maximum resources that an instance can utilize. These factors make it hard to deliberately evoke instance migration during scaling up in our test. Therefore, we simulate the migration scenario by utilizing the promotion of a secondary node since migration typically involves adding a new instance as a secondary node and subsequently promoting it to the primary node. We conducted migration tests on several existing commercial serverless databases using this approach. As Aurora Serverless enjoys widespread adoption and exhibits comparatively superior migration performance among them, we present its results here to showcase the migration's impact, as depicted in Fig. 2. In this test, we run SysBench's read-write workload on the primary node. At the 30-second mark, we promote the secondary node to simulate an instance migration. The promotion process takes approximately 15 seconds to complete, while other databases we have tested even take a much longer time. During this period, the application loses its connection to the database and cannot reconnect until the migration is finished. This is typically challenging for users to accept, especially considering that it often occurs during the peak activity of the application.

3) Avoiding/alleviating migration: To minimize the impact of migrations, existing serverless databases employ strategies to reduce the frequency of migration by reserving excessive resources, including limiting the maximum hardware resources allocated to an instance and restricting the maximum number of instances running on one physical host. While these two limitations can help avoid migration or reduce the cost of migration, they result in resource underutilization, leading to increased costs for cloud providers. In an extreme scenario, consider a physical host with 128 vCores, where each instance

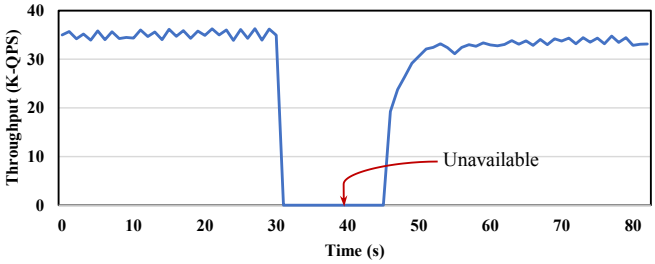

Fig. 2. The promotion of secondary node in Aurora

is capped at 64 vCores to completely prevent migration. In this setup, each host can accommodate only two instances. However, this strategy leads to a significant underutilization of the hardware resources, as these two instances often utilize only a fraction of the available resources. From the perspective of the cloud provider, this deployment is equivalent to deploying two 64-vCore provisioned database instances on a single physical host. Consequently, while users may derive benefits from this deployment, the cloud provider does not achieve cost savings. In reality, in this example, many serverless databases run more than two instances on a single physical host. This approach is justified by the relatively low probability that both instances will simultaneously reach their peak performance levels. However, once encountering a situation where there is no free resource for scaling up, the migration can result in a brief period of downtime during the peak activity of the application or force the system to cease scaling up, which would be a violation of its service commitments.

To summarize, achieving seamless migration can eliminate negative impacts on applications during the migration. Consequently, there is no need to reserve excessive resources on the physical host to avoid migration. As a result, one physical host can accommodate more instances, significantly improving resource utilization. In this case, both users and the cloud provider can benefit from the serverless database.

B. Scale-out requirements

As discussed in Section II, the secondary node in the shared-storage-based databases only supports eventual consistency. Consequently, the shared-storage-based serverless database must offer two endpoints: one ensuring strong consistency and the other providing eventual consistency, as shown in Fig. 3. In this design, when the read pressure becomes heavier, it has to scale up the primary node rather than scaling out secondary nodes. As the resource allocated for one instance usually has an upper bound, a heavy read pressure (that requires strong consistency) can scale up the primary node to the maximum capacity, and the performance can not be improved anymore, while the scale-up and scale-out abilities of the secondary nodes are neither utilized. Again, we use Aurora Serverless as an example to illustrate the auto-scaling behavior in read-dominant workloads within shared-storage-based serverless databases, as depicted in Fig. 4. Initially, the number of Aurora Capacity Units (ACUs) stays at a minimal low level because there is no pressure. However, at the 180-second mark, a substantial read-only workload is initiated on the primary node (due to the requirement for strong consistency). As a result, the number of ACUs starts to increase, eventually stabilizing at its maximum capacity of 128. We further add additional workload at the 2340-second mark. However, the number of ACUs does not increase any further because it has already reached its maximum limit and the throughput also experiences no further improvement. If the system extends its support for strong consistency on secondary nodes, it would have the capacity to scale out secondary nodes for handling heavier read workloads. Conversely, a serverless database that lacks a strong consistency guarantee is limited to scaling up the primary node and does not have the ability to scale out secondary nodes. This is particularly relevant as many applications are primarily read-intensive [9], [15], [35], including Alibaba's trading service workload (comprising 50% reads) and inventory management service workload (comprising 90% reads).

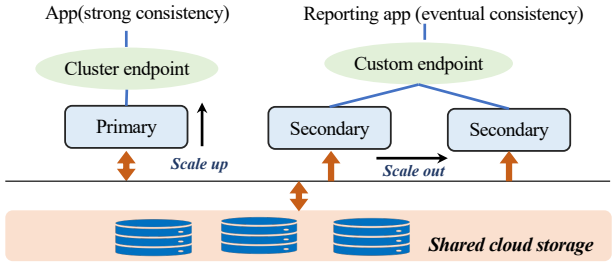

Fig. 3. The architecture of the serverless database with two endpoints [6]

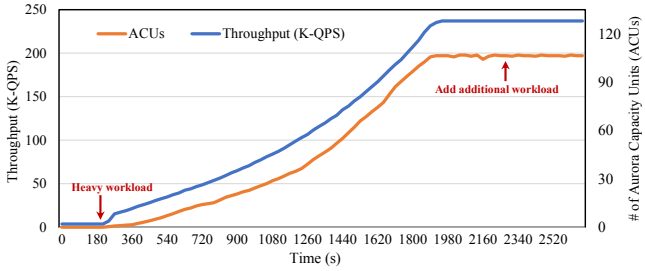

Fig. 4. The auto-scaling of Aurora Serverless in read-dominant workload

On the other hand, scaling up the primary node will also induce a higher probability of instance migration, which usually has a negative impact. So the serverless database should achieve strong consistency and consequently provide a unique endpoint for the applications. Thus it can scale up/out secondary nodes for heavier read workloads and the primary nodes only scale when the write pressure becomes heavy. Therefore, it can reduce the possibility of saturating the primary and instance migration. PolarDB supports strong consistency on secondary nodes [37], which enables PolarDB Serverless to scale out/up secondary nodes for read requests.

IV. Overview of PolarDB Serverless

PolarDB Serverless is built based on provisioned PolarDB, thus it has a similar architecture to PolarDB, as shown in Fig. 5. It adopts the disaggregated shared storage architecture, featuring one primary node to process read/write requests and one or more secondary nodes dedicated to processing read requests. Additionally, it incorporates a proxy node that operates atop both the primary and secondary nodes. This proxy node facilitates functionalities such as read/write splitting, load balancing, and failover management, as well as various serverless-related functionalities, including connection management and query statement caching. To enable dynamic resource allocation, PolarDB Serverless is equipped with a resource monitoring component. This component gathers data on resource utilization from all database instances, empowering it to make decisions regarding resource allocation or reclamation. Typically, these decisions revolve around the dynamic adjustment of memory and CPU resource utilization. Notably, given that memory usage is predominantly associated with the buffer pool within the database, PolarDB Serverless employs a buffer pool manager to resize the buffer pool, consequently impacting memory consumption. PolarDB Serverless manages resources at the unit of PolarDB Capacity Units (PCUs), where one PCU represents the hardware resources of 1 vCPU, 2GB of memory, and corresponding networking and I/O. It allocates and deallocates resources for an instance at the granularity of half a PCU. It is important to note that PolarDB Serverless only supports a single primary node. Consequently, it is feasible to scale up the primary node only, while the secondary node can be scaled up and scaled out, this is similar to the typical approach adopted by most shared-storage-based serverless databases. However, what makes PolarDB Serverless stand out are two innovative designs that deserve special attention when compared to other serverless databases: the seamless migration mechanism and the unified strongly consistent endpoint.

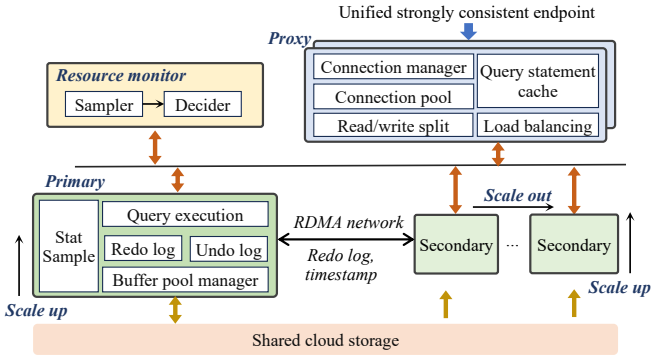

Fig. 5. The architecture of PolarDB Serverless

A. Seamless migration

PolarDB Serverless implements the seamless migration to support migrating a database instance seamlessly from one physical host to another without interruptions in cases where there are insufficient free resources to scale up the resident instance. The two fundamental designs for seamless migration are connection maintenance and transaction migration.

1) Connection maintenance: The proxy node is responsible for managing connection access. All applications are directly connected to the proxy node. The proxy node verifies access rights for applications' connections and establishes a corresponding connection to the database instances for each of them. During a database migration, the proxy node establishes new connections to the new instances, closing connections to the old instances after shutting them down. Subsequently, it remaps the applications' connections to the new instances. Therefore, the connections between applications and the proxy node remain active without any disruptions.

2) Transaction migration: Transaction migration in PolarDB Serverless allows the continuation of the old instance's uncommitted transactions on a new instance without interruptions. The essential aspect of this approach is to recover the transaction's state and its associated modifications on the new instance. With the potential of ongoing query streams from applications, we restore both the transaction state and relevant changes at the query level. This design ensures that only the most recent query needs to be rolled back and re-executed on the new instance. To facilitate this, we utilize the proxy node to cache the latest query statement for each transaction. Because the data changes are all recorded by redo logs, migrating the redo log to the new instance can enable the new instance to recover these changes. For in-flight queries, their changes should be rolled back on the new instance by applying the corresponding undo logs. To recover the active transaction's in-memory state, it also requires migrating the related metadata to the new instance. To exactly roll back the latest query, the proxy node caches the latest query's undo log number for each connection. When the database node returns a query's response to the proxy node, the corresponding undo log number is stored with the response and the proxy node will cache it. For efficiency, we utilize RDMA for data transfers, and a data preload strategy is implemented for further improvement. The details are introduced in the subsequent section.

B. Read scale-out

PolarDB Serverless inherits the strong consistency feature from the provisioned PolarDB. This attribute enables PolarDB Serverless to process the reads on the secondary nodes while still guaranteeing strong consistency. In this case, it can scale out the secondary nodes to improve read performance. Each PolarDB Serverless can provide a unified endpoint with strong consistency. All applications can direct their requests to this unique endpoint. When facing heavier read workloads, PolarDB Serverless can effectively scale out the secondary nodes in the background to maintain optimal performance.

V. Implementations

A. Seamless migration

PolarDB Serverless introduces a connection maintenance design to ensure that connections remain active for applications during the migration process. Another important aspect

of seamless migration is the migration of active transactions to the new instance. The seamless migration requires that transactions are not aborted, allowing applications to send requests to the database as usual. This, in turn, requires the database system to be capable of recovering the state of active transactions on the new instance, enabling the new instance to seamlessly process the subsequent queries of the transaction.

1) Connection maintenance: Connection maintenance plays an important role in maintaining active connections for applications throughout the instance migration process. This design leverages the capabilities of the proxy node. PolarDB Serverless provides a unified endpoint for applications via the proxy node. The applications are all connected to the proxy node rather than directly to the database instance. The proxy node manages a connection pool that stores all the connections established with the applications. Upon receiving a connection request from the application, the proxy node first establishes the connection with the application. It then attempts to establish connections with the backend database instances for that application. In cases where the database has multiple secondary nodes, the proxy node establishes connections with each of these nodes. Consequently, a single connection between the application and the proxy may map to multiple connections to database instances, each connected to a different node. During instance migration, the connections between the proxy node and the migrating instance are disconnected. However, the corresponding connection between the application and the proxy node remains active. The application can still send requests using this connection. However, the requests are temporarily cached on the proxy node and will be forwarded to the new instance after migration is completed. After migration, the proxy node establishes new connections with the new instance and remaps the connection to the application connection. If there are cached requests on that connection, the proxy node sends them to the new instance. Throughout the migration, the application can continue sending requests. However, if a request is sent via the affected connection, it may experience a longer response time because the proxy node must wait for the migration to complete before forwarding the request to the new instance. From the application's perspective, the application is always connected to an alive instance, but with the potential for increased query latency when using the affected connection.

2) Live transaction migration: Transaction migration is a critical part of seamless and instant migration. It supports the uncommitted transaction to continue to be executed on the new instance. In an interactive transaction model, where a transaction comprises multiple queries, the database receives the queries sequentially. If the migration happens during a transaction's processing, it will lose the in-memory data (such as undo/redo logs and related metadata) and the updates made by the transaction's prior queries have been successfully responded to. So, to support the live transaction migration, we should guarantee the changes made by the prior successful queries can be still seen by the new instance. The uncommitted transaction's state can be recovered by the new instance. The live transaction migration involves two important parts: data migration and transaction recovery.

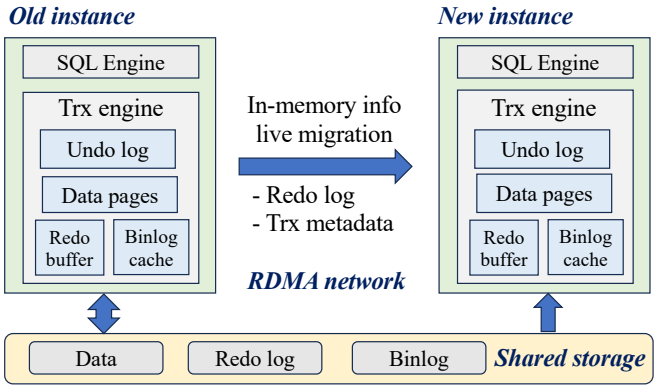

Fig. 6. In-memory data synchronization during instance migration

Cache synchronization. The most critical data component is the in-memory redo log, which reflects all updates that occurred on the old instance. Migrating the in-memory redo log to the new instance ensures that the new instance has all the changes that happened on the old instance. Consequently, all changes initiated by successful queries are visible on the new instance. For in-flight queries that are still executing at the time of migration, their changes must be recovered on the new instance. Achieving this rollback necessitates migrating the corresponding undo logs. In PolarDB Serverless, undo logs are organized into pages. To minimize data traffic, we opt to migrate the redo log associated with the undo data rather than directly transferring entire undo pages. In addition to data, certain in-memory data structures (e.g., transaction objects) and related metadata must also be migrated to the new instance. Some users may enable binlog in their databases, in which case the in-memory binlog requires migration as well. Fig. 6 shows the data migration process during the instance migration. Once the new instance is prepared, it will send some metadata (such as its log buffer address) to the old instance. Subsequently, the old instance will start to remotely write its corresponding redo logs to the new instance's log buffer. During the migration, the old instance is still serving applications. In this case, when the old instance finishes executing one query, it will send the query's related redo logs (including the undo data's redo) to the new instance before acknowledging the applications. This can guarantee the new instance has all the related redo logs of the successful queries.

Transaction reconstruction. On the new instance, changes made by the unfinished queries need to be rolled back and re-executed. The critical task is precisely rolling back the changes made by an individual query while also restoring the latest query statement. To address this challenge, we leverage the capabilities of the proxy node. As illustrated in Fig. 7, the proxy node caches the latest query for each transaction, along with its associated metadata. When the proxy node receives a query from the application, it initially caches the most

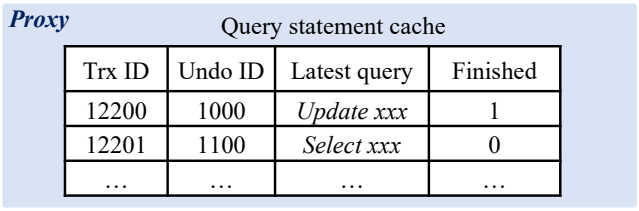

Fig. 7. The query statement cache on proxy

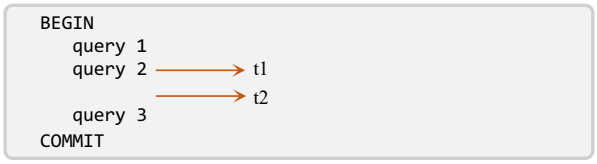

Fig. 8. An example of a transaction

recent query for each transaction in its memory, marking it as unfinished. On the database side, it includes the maximum undo log ID generated by the current query in the response to the proxy node. When the proxy node receives a query's response from the database, it associates this ID with the query and marks the query as finished. If a query is marked as finished, it signifies that its updates have all been migrated to the new instance, obviating the need for further recovery actions on the new instance. However, if it's marked as unfinished, it indicates that this query was in progress on the old instance when the old instance was shut down. In such cases, it's necessary to roll back the changes made by the query. This is achieved by applying the undo log, starting from the transaction's latest undo log and ending at the query's undo log ID that is stored on the proxy node. After the rollback, the query can be sent to the new instance for re-execution.

Lock state synchronization. Like many databases, PolarDB Serverless implements multi-version concurrency control and defaults to read committed snapshot isolation. This approach allows reads to access the committed version without necessitating row locks, while writes lock rows for updates by obeying a two-phase locking protocol. Essential to transaction migration is the transfer of lock states between instances. To facilitate this, PolarDB Serverless embeds the transaction ID within the row's payload upon modification, a practice also common in databases like Oracle and PostgreSQL. In this mechanism, a transaction intending to modify a row first checks the embedded transaction ID. If this ID indicates an uncommitted state, it signifies that the row is locked by another active transaction, and thus, the current transaction must wait. Once the locking transaction commits, the waiting transaction can proceed by writing its transaction ID into the row, effectively acquiring the lock. During migration, once we finish recovering the data on the new instance, the lock state should be also recovered accordingly.

Example of transaction resuming . Fig. 8 gives an example of a transaction comprising three queries, sent to the database sequentially. If the migration occurs at $t1$, the secondary query is still in progress on the old instance. Since the proxy node keeps track of the highest undo log ID associated with the last finished query, the new instance can apply undo logs from the newest ID down to that specific ID to roll back changes made by the secondary query. Subsequently, it re-executes the secondary query on the new instance, and the following query will be also sent to the new instance. If the migration takes place at $t2$, the secondary query has already successfully responded to the application and is marked as finished on the proxy node. In this case, there is no need to roll back this query. If the third query arrives after migration, it can be directly sent to the new instance. If it arrives during migration, the third query is temporarily cached on the proxy node and marked as incomplete. It will be forwarded to the new instance once it is ready.

Binlog migration. When the binlog is enabled for some specialized tasks, transaction migration should support binlog migration as well. Unlike redo logs, binlogs don't have a unified buffer. Each thread maintains its own binlog buffer, appending to it during transactions. Upon transaction committing, this buffered binlog will be written to the global binlog file on the storage. In exceptional cases, a transaction may generate a large amount of binlog, spanning several GBs. Storing such extensive binlogs in memory isn't feasible. They are temporarily stored on storage and appended to the global binlog file during committing. During migration, uncommitted transactions necessitate in-memory binlog data transfer to the new instance, and any on-storage temporary binlog files may require remapping.

Fig. 9 details the binlog migration process. At the beginning of the migration, the new instance will allocate an amount of memory. The old instance then requests memory addresses for transactions from the new instance. Subsequently, it transmits its in-memory binlog to the new instance's binlog cache via the one-sided RDMA interface. Some transactions might have saved portions of their binlogs as temporary storage files. To identify these on the new instance, temporary files are tagged with unique transaction IDs. The related metadata within these files, like persistency offset, is stored with the related in-memory binlog for migration. During the migration, every query, upon completion, ensures its binlog gets written remotely to the new instance. After migration, the new instance assigns its node transaction objects and retrieves the respective binlogs from the cache and temporary files.

B. Migration speedup

Transaction migration prevents interruptions and reduces the cost associated with rolling back uncommitted transactions on the new instance. However, it is essential to further reduce the cost of migration. PolarDB Serverless accelerates this process using two strategies. The first is the RDMA-based data migration. The RDMA network is already a fundamental infrastructure at Alibaba and highly co-design with PolarDB. During migration, the related in-memory data (*e.g.*, redo log, binlog, and the necessary metadata) are transferred to the new instance via the one-sided RDMA interface. The other one is

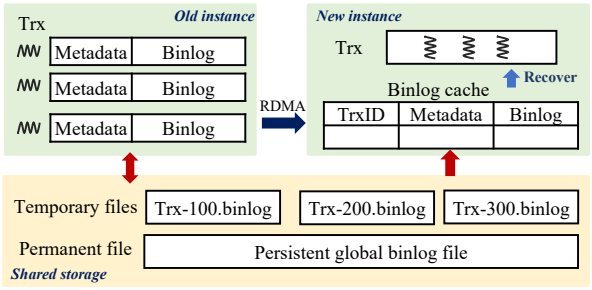

Fig. 9. The migration of binlog

to warm up the new instance in advance. It includes preloading the necessary redo/undo/binlog logs from the storage and parsing them in advance if possible. This could save a lot of I/O overheads and CPU cycles during the migration. Leveraging these optimizations, migration can be completed in under half a second, as depicted in Fig. 10 in Section VI.

C. Schedule policy

PolarDB Serverless supports the strong consistency, enabling the secondary node to process the strongly consistent reads. In this case, it is possible to implement the read/write split in PolarDB Serverless. This will bring more opportunities for the scheduling of scaling up primary/secondary nodes and scaling out secondary nodes.

1) Scaling up primary node: PolarDB Serverless supports only a single primary node, allowing for scaling up but not scaling out. Scaling up an instance can potentially trigger instance migration, which can impact performance. To mitigate this, there are several strategies in PolarDB Serverless. The first strategy is to scale out the secondary node instead of scaling up the primary. In cases where the primary node experiences increased pressure, especially with concurrent read requests, PolarDB Serverless can expand by adding a secondary node and redirecting the read requests to it. This approach helps alleviate the pressure on the primary node and may mitigate the need to scale up the primary node. The second strategy accelerates migration by preparing a new instance in advance. This preparation occurs when the primary node's capacity is increasing, and the available resources on the current host have decreased to 10%. This new instance is added to the database cluster in the role of a secondary node and receives the primary node's redo logs as discussed in Section V-A2. The primary node also synchronizes its buffered data with the new instance if there's sufficient time. When migration is initiated, the new instance is swiftly promoted to the new primary since it already possesses the necessary redo logs and some hot data in its memory. The third strategy involves proactive instance migration. At Alibaba, clear and consistent periodic usage patterns have been observed in many applications. For example, some databases consistently scale up to a specific capacity at around the same time each day. Predicting such instances' need for migration during their peak activity times can enable pre-migration to prevent migration-related performance issues during those periods. The last strategy is to avoid migrating the primary node by scaling out the secondary nodes of another cluster. For instance, when planning to migrate the primary node of *cluster-A* for scaling up and simultaneously discovering that a secondary node of *cluster-B* is using a significant amount of resources on the current host, the secondary node can be scaled down, freeing up resources for scaling up the primary node of *cluster-A* instead of migrating it. To ensure the performance of *cluster-B*, additional secondary nodes are added to *cluster-B* before scaling down its secondary nodes.

2) Scaling up/out secondary nodes: In PolarDB Serverless, secondary nodes can provide strongly consistent reads, enabling the ability to scale out secondary nodes for handling read-intensive workloads. However, the decision of whether to scale up a secondary node or scale out depends on two considerations. One approach is to minimize the number of secondary nodes. Having more nodes in the database cluster will complicate the management and waste the bandwidth. So we prefer to initially attempt to scale up a secondary node and resort to scaling out only if scaling up is not feasible in certain situations. The second one is to minimize the likelihood of migration. When the free resources on the current host are reduced to 20%, we will not scale up the secondary node on the current host, and try to scale out. This can avoid the potential migration for the resident primary node.

VI. Evaluation

A. Objectives

While serverless databases are a relatively recent addition to the market, there currently exists no standard benchmark for their evaluation. Drawing on our expertise in developing serverless databases and insights from customer feedback, we highlight several critical performance-related aspects. We then evaluate PolarDB Serverless based on these criteria by answering the following question.

- Can PolarDB Serverless achieve a fast seamless cross-machine scaling up without interruptions?
- Does PolarDB Serverless automatically scale out secondary nodes for read-heavy bursty workloads?
- Does PolarDB Serverless adjust resource allocation dynamically based on varying workloads?

B. Setup

1) Test platform: Since PolarDB Serverless is already commercially available at Alibaba Cloud, our evaluations are all conducted in the Alibaba public cloud environment. In our test, the underlying physical machines are equipped with 2 Intel Xeon Platinum 8369B CPU and 1TB DDR4 DRAM, running CentOS-7 OS. These physical machines are connected by a 50Gbps Mellanox ConnectX-4 network. In PolarDB Serverless, it measures the capacity of an instance with the unit of PolarDB Capacity Unit (PCU), where one PCU indicates the hardware resource of 1 vCore, 2GB memory, and corresponding networking and I/O.

2) Baseline: Aurora Serverless is one of the most popular serverless databases that is based on shared-storage architecture. It is naturally our baseline. The testing within Aurora Serverless is conducted on the Amazon Web Service and adopts its default parameters. We test Aurora Serverless with its cluster endpoint which guarantees strong consistency. In Aurora Serverless, it measures the capacity of an instance with the unit of Aurora Capacity Unit (ACU). Each ACU is a combination of approximately 2GB of memory, corresponding CPU, and networking.

3) Workloads: Since the serverless database is used for dynamic workloads, we use the SysBench [24], a popular benchmark stressing DBMS systems, to generate a dynamic workload in our evaluations. In most databases, we configure the SysBench with 32 tables and each has 1M records.

C. Migration performance

As seamless migration is one of the most important features of PolarDB Serverless, we first evaluate the instance migration performance during scale-up, comparing it to Aurora Serverless. In serverless databases, the instance migration occasionally occurs during scale-up. Some databases often reserve some spare resources on the physical host for potential scale-ups. Moreover, Aurora Serverless caps the maximum ACUs of an instance at 128. These make it challenging to intentionally trigger instance migration during our tests. Typically, instance migration involves adding a new instance as a secondary node and then promoting it to the primary node. Therefore, to simulate the migration scenario, we perform a test by promoting a secondary node to the primary node.

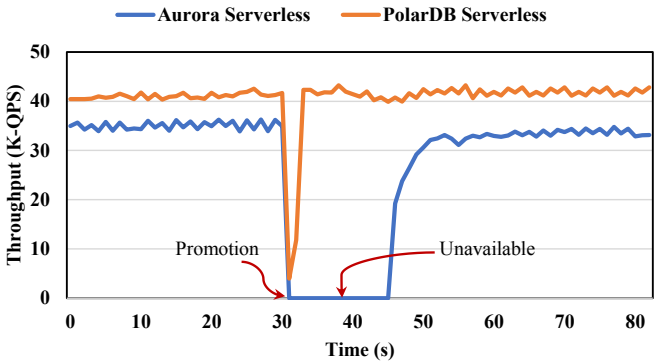

Fig. 10. The performance of secondary node promotion

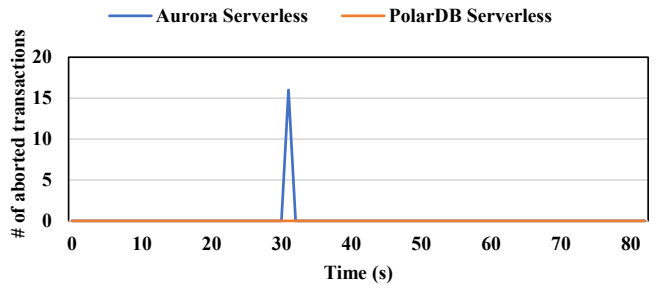

Fig. 11. The number of aborted transactions during secondary node promotion

```
[ 24s ] thds: 16 tps: 1751.99 qps: 35047.78 (r/w/o: 24535.85/7007.96/3503.98) lat (ms,95%): 11.45 err/s: 0.00 reconn/s: 0.00
[ 25s ] thds: 16 tps: 1757.06 qps: 35160.27 (r/w/o: 24605.89/7040.26/3514.13) lat (ms,95%): 11.04 err/s: 0.00 reconn/s: 0.00
[ 26s ] thds: 16 tps: 1736.93 qps: 34696.63 (r/w/o: 24299.04/6923.73/3473.86) lat (ms,95%): 12.08 err/s: 0.00 reconn/s: 0.00
[ 27s ] thds: 16 tps: 1714.90 qps: 34243.00 (r/w/o: 23953.60/6859.60/3429.80) lat (ms,95%): 11.65 err/s: 0.00 reconn/s: 0.00
[ 28s ] thds: 16 tps: 1742.01 qps: 34859.15 (r/w/o: 24413.10/6962.03/3484.01) lat (ms,95%): 10.84 err/s: 0.00 reconn/s: 0.00
[ 29s ] thds: 16 tps: 1742.10 qps: 34840.92 (r/w/o: 24383.34/6973.38/3484.19) lat (ms,95%): 11.65 err/s: 0.00 reconn/s: 0.00
[ 30s ] thds: 16 tps: 1756.95 qps: 35184.05 (r/w/o: 24641.34/7028.81/3513.91) lat (ms,95%): 11.04 err/s: 0.00 reconn/s: 0.00
FATAL: mysql_drv_query() returned error 2013 (Lost connection to MySQL server during query) for query 'UPDATE sbtest17 SET k=k+1 WHERE id=49770'
FATAL: mysql_drv_query() returned error 2013 (Lost connection to MySQL server during query) for query 'UPDATE sbtest8 SET k=k+1 WHERE id=49929'
FATAL: mysql_drv_query() returned error 2013 (Lost connection to MySQL server during query) for query 'COMMIT'
FATAL: mysql_drv_query() returned error 2013 (Lost connection to MySQL server during query) for query 'UPDATE sbtest18 SET k=k+1 WHERE id=50423'
FATAL: mysql_drv_query() returned error 2013 (Lost connection to MySQL server during query) for query 'UPDATE sbtest25 SET k=k+1 WHERE id=51142'
FATAL: mysql_drv_query() returned error 2013 (Lost connection to MySQL server during query) for query 'COMMIT'
(last message repeated 3 times)
FATAL: mysql_drv_query() returned error 2013 (Lost connection to MySQL server during query) for query 'UPDATE sbtest2 SET k=k+1 WHERE id=50205'
FATAL: mysql_drv_query() returned error 2013 (Lost connection to MySQL server during query) for query 'COMMIT'
(last message repeated 1 times)
```

Fig. 12. The errors that were reported during secondary node promotion in Aurora Serverless

In this test, we first run the SysBench's read-write workload on the primary node. Upon the primary node reaching a certain scale, we simulated a migration triggered by scale-up by promoting the secondary node to the primary. Fig. 10 shows the results for PolarDB Serverless and Aurora Serverless. At the 30-second mark, we promote the secondary node to the primary. We can find that PolarDB Serverless only experiences a momentary throughput dip, restoring performance within half a second. Such fast migration benefits from our careful designs with RDMA-based data transferring. In contrast, Aurora Serverless required 15 seconds for the new primary node to become available. What's more, during the migration, Aurora Serverless aborted all application connections, preventing new connections until promotion completion. Fig. 11 gives the number of the aborted transactions. During the test, there are no aborted transactions in PolarDB Serverless because it supports transaction migration. The unfinished transaction can continue to be executed on the new instance. But, in Aurora Serverless, when the old instance shuts down, the connections are aborted and the corresponding transactions are also aborted, reporting 16 aborted transactions because there are only 16 connections in our testing. Fig. 12 further presents the client-side screen snapshot during the promotion in Aurora Serverless, depicting connection losses. However, PolarDB Serverless reports no errors, and the connection remained uninterrupted. With PolarDB Serverless, applications behaved as if continuously connected, still dispatching requests. These requests were temporarily cached on the proxy node and processed by the new primary.

D. Elasticity performance

1) Read scale-out: PolarDB Serverless delivers strong (read-after-write) consistency for secondary nodes, facilitating performance enhancement through secondary node scaling. However, the secondary node in the other serverless database, such as Aurora Serverless, can not serve strongly consistent reads. The read performance will be bound by the single primary node. In this test, we evaluate the scalability of the secondary node in the PolarDB Serverless, compared to Aurora Serverless, shown in Fig. 13. Initially, no workloads run, with both PolarDB Serverless and Aurora Serverless operating on minimal hardware. We use the SysBench to generate a read-dominant workload with high pressure (500 threads) and begin running at the 50-second mark. Aurora

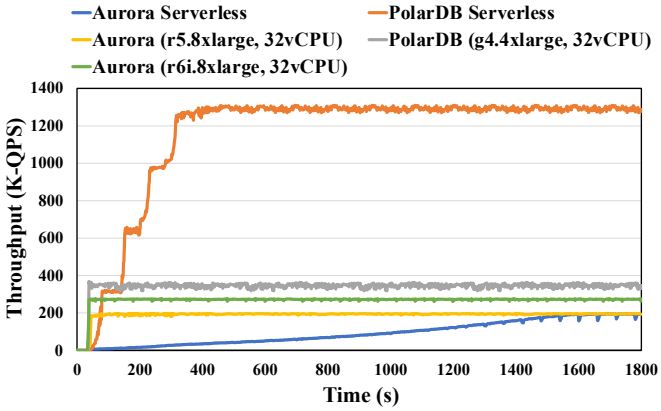

Fig. 13. The elasticity under read-dominant workload

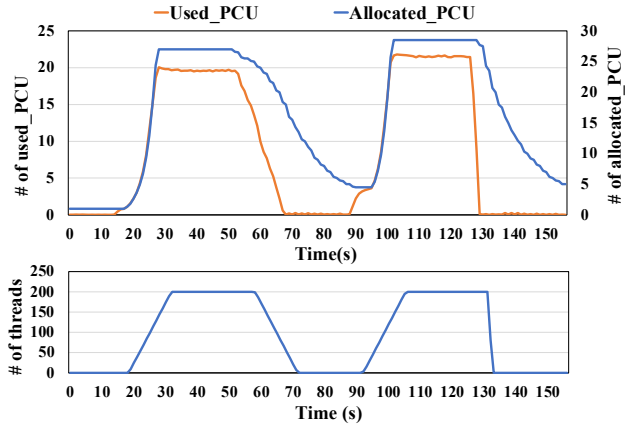

Fig. 14. Changes in PCU under different pressures

Serverless routes all requests to its primary node due to strong consistency demands, gradually scaling up with increasing throughput. Eventually, its performance is sustained at 200 K-QPS because it is already scaled up to the maximum capacity (128 ACU). However, PolarDB Serverless scales rapidly in response to the workload, showing an immediate throughput rise. Although it caps PCUs at 32 per instance and a single node can not accommodate this heavy workload, it can scale out secondary nodes to handle read requests, reaching a peak throughput of 1.3M QPS.

Given Aurora Serverless's undisclosed hardware and distinct implementation from PolarDB, direct performance comparisons may not be fair here. Fig. 13 juxtaposes the performance of 32vCPU provisioned PolarDB and Aurora, providing a context for serverless performance. While Aurora Serverless performs similar to its 32vCPU provisioned counterpart (*r5.8.xlarge*), and slightly underperforms the other 32 vCPU provisioned counterpart (*r6i.8xlarge*). This is because their underlying hardwares are different. PolarDB Serverless outperforms, achieving roughly 2.5 times the throughput of its 32 vCPU provisioned version. This is because PolarDB Serverless can scale out secondary node to improve read performance (while ensuring strong consistency) but it is not possible in Aurora Serverless.

2) Dynamically resource allocation: Serverless databases, by design, dynamically adjust resources based on dynamic workload. The intensive workloads lead the database to utilize nearly all its resources, triggering a scale-up. Subsequently, additional resources are allocated, and the database eventually stabilizes, utilizing these resources at a predetermined level (defaulted to 80% in PolarDB Serverless). This test focuses on the interplay between allocated and consumed PCUs in PolarDB Serverless during workload variations.

As PolarDB Serverless allocates the resource at the granularity of PCU, we explored the dynamic behavior of PolarDB Serverless, focusing on its dynamic PCU allocation and utilization, as depicted in Fig. 14. The bottom graph illustrates the workload pressure over time, represented by the number of client threads. The top graph mirrors this with corresponding PCU data. Initially, with minimal database stress, both allocated and used PCUs are low. However, as we gradually intensify the workload pressure starting from the 20-second mark, the utilization of PCUs increases significantly, almost exhausting all of the allocated PCUs. Accordingly, the scheduling policy attempts to assign more PCUs to bring the used PCUs back down to the predefined percentage. Nonetheless, the workload pressure increases at a faster rate than the PCU allocation can accommodate. Consequently, the used PCUs closely match the allocated PCUs during this period. Once the workload stabilizes after the 20-second mark, PCU allocation ceases, and the used resource percentage remains at the predetermined value. Subsequently, when the workload pressure subsides after the 60-second mark, the allocated PCUs are gradually reclaimed. It's worth noting that resource deallocation in PolarDB Serverless tends to proceed at a more measured pace compared to allocation, primarily to ensure optimal performance. As the workload pressure once again starts to incrementally rise around the 90-second mark, it exhibits a pattern similar to the previous occurrence.

VII. Related Works

1) Shared-storage-based databases: Shared-storage-based cloud-native databases, such as AWS Aurora [36], Azure Hyperscale [17], [27], Azure Socrates [3] and Alibaba PolarDB [28], have gained widespread adoption and popularity among users. Typically, these databases consist of a primary node and one or more secondary nodes. The primary node is responsible for handling both read and write requests, while the secondary nodes focus on processing read requests that do not require strong consistency. The primary and secondary nodes share disaggregated cloud storage. It eliminates the need for additional storage overhead when adding secondary nodes and removes the requirement for data migration when migrating the database instance. However, challenges arise when developing serverless versions of these databases. They encounter difficulties with instance migration during scaling up and face limitations in scaling out secondary nodes due to the absence of strong consistency support in secondary nodes.

2) Shared-nothing-based database: Shared-nothing architecture is another prevalent approach. Popular examples of

databases utilizing the shared-nothing architecture include Spanner [14], CockroachDB [34], OceanBase [38], TiDB [21], DynamoDB [1], Cassandra [4], FoundationDB [5] and PolarDB-X [10]. Unlike the shared-storage architecture, the shared-nothing architecture involves partitioning the entire database. Each node operates independently and maintains its dedicated storage. A node can only access data within its specific partition. When a transaction accesses multiple partitions, it requires the use of distributed transaction mechanisms, such as the two-phase commit policy. Scaling out such databases, by introducing a new node, demands the repartitioning of the database. The duration of data migration can be quite lengthy [10], depending on the total size of the database. As a result, this approach is not suitable for a serverless database that requires rapid scaling out.

3) Serverless database: Several OLTP serverless databases have been introduced recently. They are typically based on either the shared-storage architecture (such as Aurora Serverless [2]) or the shared-nothing architecture (*e.g.*, CockroachDB Serverless [26]). Some innovative approaches have also emerged (e.g., shared memory) [12]. However, these serverless databases have certain limitations. Shared-memory-based approaches are still primarily experimental in academia and have not yet reached commercial availability The majority of commercial databases continue to rely on shared-nothing or shared-storage architectures. Shared-nothing-based databases often require data migration when adding new nodes during scaling out, which can be time-consuming and challenges the goal of providing second-level elasticity in a serverless database. Existing shared-storage-based serverless databases face issues related to instance migration and strong consistency. PolarDB Serverless is designed to address these challenges and aims to provide a serverless database with seamless auto-scaling capabilities. There are also some NoSQL serverless databases (*e.g.*, Firestore [23]), but they are not suitable for OLTP applications. This paper mainly focuses on the OLTP database.

4) Online migration: Some works also focus on online migration. First of all, to achieve an even data distribution between nodes, several works try to optimize data migration, so as to minimize performance impact and service interruption during data migration, typical approaches include Remus [22], CockroachDB [34], Microsoft Citus [16], Greenplum [13], Amazon Redshift [7] and E-Store [33]. On the other hand, there are also some efforts to optimize live migration in a multi-tenant database, so as to achieve live migration with effectively zero down-time and ensure minimal impact on transaction execution, e.g., Albatross [18], ProRea [32], Slacker [8], Zephyr [19]. Also, live migration approaches for key-value stores are proposed [25]. However, the process of migration operation is very time-consuming and complex [10], and hence migration operations in these systems will still interrupt online service. Unlike the above work, PolarDB Serverless achieves seamless migration through flexible design of connection maintenance and live transaction migration. Also, PolarDB serverless supports seamless horizontal scaling out according to actual workloads, and ensures that no errors and online service will not be interrupted during scaling out.

VIII. Conclusion

This paper reviews existing serverless databases and introduces two critical requirements for a serverless database: seamless migration and strong consistency. Seamless migration enables the migration of database instances when there are insufficient free resources on the physical host during scaling up. In such cases, physical hosts don't need to reserve excessive resources for potential scaling up, resulting in lower total ownership costs. Meanwhile, strong consistency empowers secondary nodes to handle read requests requiring strong consistency, efficiently utilizing the scaling-out capabilities of secondary nodes to process these requests. This also decreases the probability of scaling up the primary node and avoids potential migration. To meet these vital requirements, we introduce PolarDB Serverless, a serverless database ensuring seamless scale-up and read scale-out. PolarDB Serverless inherits its strong consistency ability from PolarDB-SCC [37] to support read scale-out. To enable seamless migration, PolarDB Serverless proposes connection maintenance and transaction migration policies. By combining these two designs, it supports the seamless scale-up. In our evaluation of PolarDB Serverless, especially in the context of database migration scenarios, it's worth noting that PolarDB Serverless completes the migration of a database instance in just half a second without causing any exceptions for applications.

Acknowledgment

We are grateful to anonymous reviewers for their constructive feedback and valuable suggestions.

References

[1] Amazon. DynamoDB. https://aws.amazon.com/dynamodb.

[2] Amazon. Amazon Aurora Serverless. https://aws.amazon.com/rds/aurora/serverless/, 2021. "[accessed-October-2023]".

[3] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, et al. Socrates: The New SQL Server in the Cloud. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1743–1756, 2019.

[4] Apache. Cassandra-3.11.4. https://github.com/apache/cassandra/tree/cassandra-3.11.4.

[5] Apple. FoundationDB. https://www.foundationdb.org.

[6] AWS. Instant and fine-grained scaling with Amazon Aurora Serverless v2. https://pages.awscloud.com/rs/112-TZM-766/images/2022_0608-DAT_Slide-Deck.pdf, 2022. "[accessed-October-2023]".

[7] Amazon AWS. Overview of managing clusters in Amazon Redshift. https://docs.aws.amazon.com/redshift/latest/mgmt/managing-cluster-operations.html.

[8] Sean Barker, Yun Chi, Hyun Jin Moon, Hakan Hacigümüş, and Prashant Shenoy. "cut me some slack": Latency-aware live migration for databases. In *Proc. of ACM EDBT*, 2012.

[9] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. TAO: Facebook's Distributed Data Store for the Social Graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, 2013.

[10] Wei Cao, Feifei Li, Gui Huang, Jianghang Lou, Jianwei Zhao, Dengcheng He, Mengshi Sun, Yingqiang Zhang, Sheng Wang, Xueqiang Wu, et al. PolarDB-X: An Elastic Distributed Relational Database for Cloud-Native Applications. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 2859–2872. IEEE, 2022.

[11] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. PolarFS: an Ultra-low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database. *Proceedings of the VLDB Endowment*, 11(12):1849–1862, 2018.

[12] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, et al. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2477–2489, 2021.

[13] Jeffrey Cohen, John Eshleman, Brian Hagenbuch, Joy Kent, Christopher Pedrotti, Gavin Sherry, and Florian Waas. Online Expansion of Large-Scale Data Warehouses. *Proc. of VLDB Endow.*, 4(8):1249–1259, August 2011.

[14] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, and Peter Hochschild. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.*, 31(3), August 2013.

[15] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's Globally Distributed Database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[16] Umur Cubukcu, Ozgun Erdogan, Sumedh Pathak, Sudhakar Sannakkayala, and Marco Slot. Citus: Distributed PostgreSQL for Data-Intensive Applications. In *Proc. of ACM SIGMOD*, 2021.

[17] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. Automatically Indexing Millions of Databases in Microsoft Azure SQL Database. In *Proceedings of the 2019 International Conference on Management of Data*, pages 666–679, 2019.

[18] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud Using Live Data Migration. *Proc. of VLDB Endow.*, 4(8):494–505, May 2011.

[19] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *Proc. of ACM SIGMOD*, 2011.

[20] Fauna. Introduction to serverless databases. https://fauna.com/blog/intro-to-serverless-databases, 2022. "[accessed-October-2023]".

[21] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. TiDB: A Raft-based HTAP Database. *Proceedings of the VLDB Endowment*, 13(12):3072–3084, 2020.

[22] Junbin Kang, Le Cai, Feifei Li, Xingxuan Zhou, Wei Cao, Songlu Cai, and Daming Shao. emus: Efficient Live Migration for Distributed Databases with Snapshot Isolation. In *Proc. of ACM SIGMOD*, 2022.

[23] Ram Kesavan, David Gay, Daniel Thevessen, Jimit Shah, and C Mohan. Firestore: The NoSQL Serverless Database for the Application Developer. In *Proceedings of the 2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 3367–3379, 2023.

[24] Alexey Kopytov. Sysbench. https://github.com/akopytov/sysbench.

[25] Chinmay Kulkarni, Aniraj Kesavan, Tian Zhang, Robert Ricci, and Ryan Stutsman. Rocksteady: Fast migration for low-latency in-memory storage. In *Proc. of ACM SOSP*, 2017.

[26] Cockroach Labs. A serverless database for your most demanding applications. https://www.cockroachlabs.com/lp/serverless-database-mc/, 2022. "[accessed-October-2023]".

[27] Willis Lang, Frank Bertsch, David J DeWitt, and Nigel Ellis. Microsoft Azure SQL Database Telemetry. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 189–194, 2015.

[28] Feifei Li. Cloud-native Database Systems at Alibaba: Opportunities and Challenges. *Proceedings of the VLDB Endowment*, 12(12):2263–2272, 2019.

[29] Chandrasekaran Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, 1992.

[30] Olga Poppe, Qun Guo, Willis Lang, Pankaj Arora, Morgan Oslake, Shize Xu, and Ajay Kalhan. Moneyball: Proactive Auto-Scaling in Microsoft Azure SQL Database Serverless. *Proceedings of the VLDB Endowment*, 15(6):1279–1287, 2022.

[31] Chaoyi Ruan, Yingqiang Zhang, Chao Bi, Xiaosong Ma, Hao Chen, Feifei Li, Xinjun Yang, Cheng Li, Ashraf Aboulnaga, and Yinlong Xu. Persistent Memory Disaggregation for Cloud-Native Relational Databases. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 498–512, 2023.

[32] Oliver Schiller, Nazario Cipriani, and Bernhard Mitschang. Prorea: Live database migration for multi-tenant rdbms with snapshot isolation. In *Proc. of ACM EDBT*, 2013.

[33] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-Store: Fine-Grained Elastic Partitioning for Distributed Transaction Processing Systems. *Proceedings of the VLDB Endowment*, 8(3):245–256, 2014.

[34] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. Cockroachdb: The Resilient Geo-distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1493–1509, 2020.

[35] The Transaction Processing Council. TPC-E Benchmark. http://tpc.org/tpce/, 2007. "[accessed-October-2023]".

[36] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon Aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1041–1052, 2017.

[37] Xinjun Yang, Yingqiang Zhang, Hao Chen, Chuan Sun, Feifei Li, and Wenchao Zhou. PolarDB-SCC: A Cloud-Native Database Ensuring Low Latency for Strongly Consistent Reads. *Proceedings of the VLDB Endowment*, 16(12):3754–3767, 2023.

[38] Zhenkun Yang, Chuanhui Yang, Fusheng Han, Mingqiang Zhuang, Bing Yang, Zhifeng Yang, Xiaojun Cheng, Yuzhong Zhao, Wenhui Shi, Huafeng Xi, et al. OceanBase: a 707 Million tpmC Distributed Relational Database System. *Proceedings of the VLDB Endowment*, 15(12):3385–3397, 2022.

[39] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Xinjun Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, et al. Towards Cost-effective and Elastic Cloud Database Deployment via Memory Disaggregation. *Proceedings of the VLDB Endowment*, 14(10):1900–1912, 2021.