Towards Millions of Database Transmission Services in the Cloud

Hua Fan*
Dachao Fu*
Alibaba Group
Hangzhou, China

Xu Wang
Jiachi Zhang
Chaoji Zuo†
Alibaba Group
Hangzhou, China

Zhengyi Wu
Miao Zhang
Kang Yuan
Alibaba Group
Hangzhou, China

Xizi Ni
Guocheng Huo
Wenchao Zhou
Alibaba Group
Hangzhou, China

Feifei Li
Jingren Zhou
Alibaba Group
Hangzhou, China

{guanming.fh,qianzhen.fdc,wx105683,zhangjiachi.zjc,zuochaoji.zcj,wuzhengyi.wzy}
{yanmen.zm,yuankang.yk,xizi.nxz,guocheng.hgc,zwc231487,lifeifei,jingren.zhou}
@alibaba-inc.com

ABSTRACT

Alibaba relies on its robust database infrastructure to facilitate real-time data access and ensure business continuity despite regional disruptions. To address these operational imperatives, Alibaba developed the Data Transmission Service (DTS), which has become critical for internal applications and public cloud services alike. This paper presents a comprehensive study of the architectural innovations, resource scheduling mechanisms, and performance optimization strategies that have been implemented within DTS to tackle the significant challenges of cross-network, heterogeneous data transmission in a cost-effective manner. We explore the novel Any-to-Any (A2A) architecture, which simplifies the complexity of data paths between diverse databases and mitigates network connectivity issues, thereby significantly reducing development overhead. Additionally, we examine a dynamic network bandwidth scheduling algorithm that effectively maintains Service-Level Objectives (SLOs), complemented by a serverless mechanism that ensures efficient resource utilization. Furthermore, DTS utilizes advanced strategies such as transaction dependency tracking, hot data consolidation, and batching to enhance synchronization performance and efficiency. DTS has distilled the lessons learned from years of serving our customer base and currently supports nearly 1 million public cloud instances annually. Our evaluation results show that DTS can effectively and efficiently handle real-time data transmission in both experimental and production environments.

PVLDB Reference Format:
Hua Fan, Dachao Fu, Xu Wang, Jiachi Zhang, Chaoji Zuo, Zhengyi Wu, Miao Zhang, Kang Yuan, Xizi Ni, Guocheng Huo, Wenchao Zhou, Feifei Li, and Jingren Zhou. Towards Millions of Database Transmission Services in the Cloud. PVLDB, 17(12): 4001 - 4013, 2024.
doi:10.14778/3685800.3685822

1 INTRODUCTION

Alibaba operates a vast digital commerce service, anchored by its resilient database services, which store essential business data. This requires two key functions: First is real-time access to database information, critical for applications like advertising and search, prompting the need for services that can parse real-time database logs to satisfy the many business units demanding instant data from primary databases [25, 30, 42]. Secondly, business continuity against regional disruptions — such as power outages or natural disasters — is pivotal, demanding real-time database synchronization to secondary regions for swift operation transfer [34]. These necessities drove Alibaba to create its own Data Transmission Service (DTS) [20] in 2011, focusing on synchronization between databases (e.g., MySQL to MySQL).

As Alibaba Cloud Computing expanded, it began offering a variety of database services to the public cloud, triggering a need for migrating more than 24 different types of databases from local data centers or other cloud providers. This diversity led to a surplus of potential data transmission pathways, heavily complicating the process and increasing the development workload. Network connectivity issues further exacerbated this complexity, potentially requiring specialized programs to access private intranets, culminating in **a significant challenge: developing numerous cross-network, heterogeneous data transmission services cost-effectively.**

Managing a high volume of DTS instances poses **the challenge of resource scheduling**. This complexity arises from the need to balance and allocate network and computational resources effectively such as bandwidth, CPU, and memory among a multitude of services. Insufficient resource allocation can lead to violations of Service Level Objectives (SLOs), adversely affecting customer business operations. As the demand for real-time access to data grows, ensuring efficient resource scheduling becomes critical for maintaining service quality and reliability in DTS operations.

The third challenge that emerges is related to synchronization performance issues. This concerns the need for near-zero delay in real-time data synchronization, which is highly sought after by our customers. However, synchronization latency can be significantly affected by the performance of the target database, particularly under high-frequency updates. Factors contributing to this delay include lower concurrency in database replication compared to the source [30], performance discrepancies in updates between heterogeneous databases [18], and inefficiencies in writing to the target database.

To conquer these challenges while meeting customer and business needs, DTS was architected with several key design considerations. In this paper, we outline the architecture aimed at reducing development complexity, resource scheduling mechanisms for enhancing user experience and efficiency, and optimizations for performance of update operations. The specifics are as follows.

*Both authors contributed equally to this research.
†Also a student at Rutgers University. Work done while at an internship at Alibaba.

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685822

- DTS employs an Any-to-Any (A2A) architecture, which is a strategic design choice that allows for universal compatibility and flexibility in data transmission. This A2A approach enables DTS to interconnect any source database with any target database, transforming and translating data formats as needed. By adopting this architecture, DTS can reduce the number of potential data transmission pathways from a factorial of M source-to-N target links to a M+N configuration. On each link, DTS encapsulates network connectivity issues into predefined scenarios within the DTS framework. Users can thus select their scenario without the need for additional network programming to achieve connectivity.
- Using the optimization-based scheduling algorithm for network flow, DTS can intelligently manage and allocate bandwidth across different data transmission links. This algorithm takes into account the current network conditions, transmission priorities, and the overall demand on the system to dynamically adjust the flow of data. By doing so, it minimizes the risk of SLO violations and ensures fair distribution of network resources among all active transmissions. Moreover, *DTS serverless* dynamically alters resource allocation for each service according to the current workload and performance metrics. This adaptive resource management ensures that computational resources are allocated efficiently in real-time.
- DTS employs a series of strategies that collectively enhance performance and efficiency. These strategies include the optimization of transaction execution by tracking dependencies to maximize concurrency, the consolidation of frequently accessed data (hot data) to reduce the volume of writes, and the use of batching techniques to enhance the transfer and processing of data. These enhancements are particularly crucial in real-time synchronization scenarios, where delays can have significant downstream impacts on business operations.

In summary, this paper makes the following contributions:

(1) The adoption of an A2A architecture, when paired with predefined network connectivity scenarios, effectively simplifies the development complexity associated with DTS.
(2) Our demonstration highlights the effectiveness of the DTS's optimization-based scheduling algorithm in managing network flow, alongside its dynamic resource allocation mechanism that enables real-time adaptation to fluctuating workloads.
(3) DTS enhances performance and efficiency through an approach that encompasses tracking transaction dependencies, consolidating hot data, and implementing batching techniques, while upholding user-defined consistency standards.
(4) We showcase the real-world deployment of DTS, which supports nearly one million public cloud instances annually, thereby affirming its practicality and scalability in an industrial setting.

The remainder of this paper is structured as follows: Section 2 offers an overview of data transmission, detailing the complexities and challenges involved in managing a vast number of DTS instances. In Section 3, we delve into the architectural design, introducing the A2A architecture, and the mechanisms it utilizes for establishing network connectivity. Section 4 explores the resource scheduling solutions including the bandwidth allocation algorithm and the DTS serverless mechanism, while Section 5 delves into the optimization strategies for efficient data writing to target databases. Lastly, Section 6 evaluates DTS's performance improvements for individual instances and the collective benefits within a datacenter.

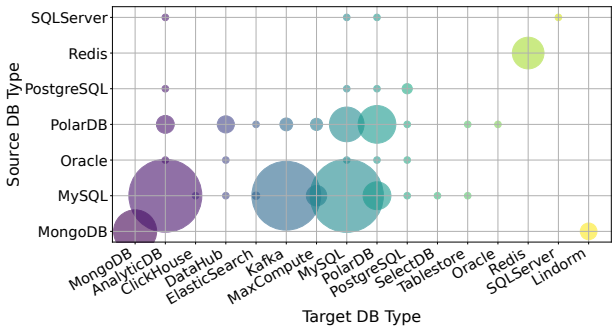


Figure 1: Diversity of Databases in DTS Instances within a Region (Circle Sizes Represent Traffic Volume)

2 BACKGROUND AND MOTIVATION

In this section, we introduce background knowledge of data transmission and three major challenges as motivations of this paper.

2.1 Data Transmission

In the domain of database research, a typical data transmission scenario entails data replication of two different databases, namely *source database* and *target database*. Based on the transmission medium, replication can be categorized into two types: physical replication, which involves the direct duplication of raw database files, and logical replication, which replays Data Manipulation Language (DML) statements on the target database. Physical replication can be readily implemented utilizing inherent features provided by database management systems, such as MySQL's Multi-threaded Replication mechanism [12]. However, its application is constrained due to its requirement for identical source and target database types. Therefore, data transmission services, such as AWS Data Migration Service (DMS) [2], Oracle's GoldenGate [13], and Fivetran [8], favor logical replication because they accommodate heterogeneous database types.

The heterogeneity of databases also compels data transmission providers to implement logical replication outside of database engines. Taking AWS DMS as an example, a transmission task consists of a source endpoint that fetches data from the source database and a target endpoint that is responsible for writing to the target database. Data transmission tasks are typically categorized, based on the fetched data, into *full* transmission tasks that transfer entire tables at once and *Changed Data Capture (CDC)* transmission tasks that replay DML statements from write-ahead logs (WALs) in real-time [32]. Despite being a mature field, the growing scale of data transmission continues to bring forth novel challenges.

2.2 Challenges

In this section, we introduce three major challenges that emerge as a result of the escalating scale of data transmission. These challenges are examined along three dimensions of scale: database and network diversity, task quantity, and transmission velocity.

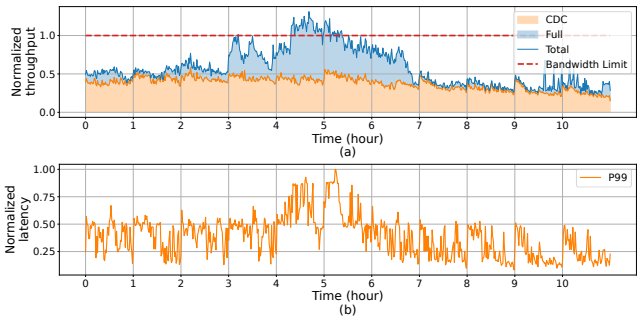


Figure 2: Sum of MBps of all DTS tasks (a) and P99 latency (b) of all CDC transmission tasks in a region.

2.2.1 Databases and Network Diversity. First, the source and target databases may encompass a wide variety of database types. As reported by DB-Engines [6], as of March 2024, there have been hundreds of cataloged database systems. Furthermore, Alibaba Cloud offers a suite of standard cloud services encompassing 24 distinct database types [1]. Various databases differ significantly in terms of their connection protocols, syntax conventions, and underlying data models, such as relational, key-value (KV), or document-oriented. Therefore, a universal data transmission tool does not exist.

Figure 1 illustrates the distribution of traffic across different pairings of source and target database types in a region of DTS. The diameter of each circle corresponds to the average data throughput observed over a defined period. The complexity of development scales to $O(N^2)$, where N represents the total number of database types.

Second, the source and target databases may be situated within heterogeneous network environments, which leads to a lack of connectivity. In Alibaba Cloud, users typically create their cloud databases within a Virtual Private Cloud (VPC) [24], which is designed to restrict external access. Meanwhile, some users create their databases within their data centers and private networks, or from other cloud providers. Therefore, it is critical for us to establish connections between the isolated network segments, while concurrently mitigating the risk of exposing sensitive data.

2.2.2 Quantity of Transmission Tasks. The escalating quantity of transmission tasks that share resources in the cloud calls for efficient utilization of bandwidth, CPU and memory. On the one hand, the escalating bandwidth demands associated with large full transmission tasks frequently result in increasing latency, which can occasionally violate Service Level Objectives (SLOs). For instance, Figure 2 shows the total throughput of all DTS instances (a), and the corresponding the 99-*th* percentile (P99) latency of all CDC transmission tasks (b). The blue area represents full tasks and the orange area represents CDC tasks. Observations indicate that: (1) a relatively strong correlation exists between latency and total throughput, and (2) there are intervals throughout the day when total bandwidth usage surpasses the designated safe limit (indicated by the dashed line). Therefore, we are looking for a fair bandwidth allocation algorithm that minimizes SLO violation and latency.

On the other hand, the workloads associated with CDC transmission tasks may significantly vary, which requires precise estimation of CPU and memory resources [39, 46]. Overestimation of resource requirements results in low resource utilization, whereas underestimation may lead to violations of SLOs. Therefore, our goal has been to develop a serverless architecture, that ensures highly efficient resource utilization and minimal SLO violations.

2.2.3 Transmission Velocity. Thirdly, for many mission-critical customers, high throughput is an essential requirement, characterized by the volume of transactions processed per second when writing to the target databases. However, in the implementation of traditional serial transaction processing methods, the transmission tasks within DTS emerge as the bottleneck, while the target database still has unused resources [28, 30, 42]. Therefore, we have been seeking for more efficient parallel data transmission mechanisms. The parallelized mechanism should not only enhance throughput but also maintain a defined level of data consistency.

3 SYSTEM

We detail the key components that make up the DTS architecture, discuss mechanisms to resolve network connectivity issues under various scenarios, and describe the Any-to-Any (A2A) model that effectively transforms data into a General Internal Representation (GIR) for seamless data interaction between different types of database systems.

3.1 Architecture

The architecture of DTS is composed of several integral components, and Figure 3 provides a detailed overview of the DTS architecture.

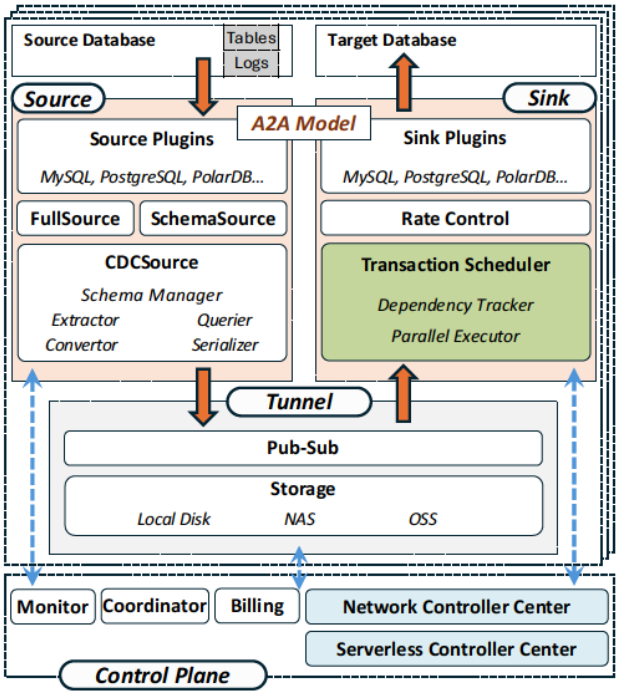


Figure 3: DTS architecture.

3.1.1 Source. The extraction component, referred to as the *Source*, is tasked with capturing both the pre-existing dataset and any subsequent modifications within the source database. It diligently

surveils the database log files for any Data Manipulation Language (DML) activities to ensure that modifications are extracted with minimal impact on the source database's operational efficiency. The modules *SchemaSource* and *FullSource* equip the Source with the capability to accurately retrieve the existing schema and the complete dataset of the source database, respectively.

The *Extractor* retrieves all database modification events from the logs. These events encompass but are not limited to, Data Definition Language (DDL) operations such as creation, alteration, or deletion of table structures, and DML operations like INSERT, UPDATE, DELETE, etc. The acquisition of this event information can be achieved through a variety of methods. In our system, the *DB Source Plugin* (detailed in §3.3) is utilized to obtain the SQL or SQL-like events. For databases that do not use DML operations, we employ alternative mechanisms to generate CDC. For instance, Redis utilizes the PSYNC command to achieve this.

In the majority of cases, the sequence of the event (or log entry) stream is consistent with the transaction commit order to ensure the persistence and consistency of data. Transaction logs, such as MySQL's binlog or PostgreSQL's WAL logs, are typically recorded in the actual sequence of transaction occurrences. However, in certain scenarios, the recorded order of logs may diverge from the actual transaction commit sequence, particularly in the case of distributed database systems or those utilizing Multiversion Concurrency Control (MVCC). The *Serializer* component reorders events according to their transaction sequence and delineates transaction boundaries within the event stream.

The *Converter* module transforms the event stream into an internal representation format known as *Records* (§3.3). Yet, when it comes to the serialization of Out-of-Row Storage fields within Records, it's imperative to employ the *Querier* module to transpose them with their corresponding field values. The Out-of-Row Storage mechanisms in databases, such as TOAST (The Oversized-Attribute Storage Technique) in PostgreSQL, LOB (Large Objects) storage in Oracle, predominantly manage the storage of large fields, such as text or Binary Large Objects (BLOBs), which may be too voluminous for direct storage within regular data rows.

3.1.2 Tunnel. After data has been successfully extracted, it must be reliably transmitted over the transmission layer, known in the DTS as the *Tunnel*.

Remarkably distinct from other transfer mechanisms, DTS has been purposefully engineered with a subscription-based interface, complemented by scalable storage for data persistence. This interface leverages the publish-subscribe (Pub-Sub) pattern, enabling multiple services or applications to subscribe to specific events or data changes, thus receiving notifications upon their occurrence.

DTS espouses the persistence of data within the Pub-Sub Tunnel for several critical reasons: (1) decoupling from the source database: the database logs can be pruned after a brief retention period, post-successful data ingestion into the tunnel. (2) simplified fault tolerance and scaling logic: with the state preserved within the tunnel, both Source and Sink components operate statelessly, simplifying the system's failover design and scale-out mechanisms. (3) deferred processing and message scheduling: some applications necessitate delayed message processing or scheduled sinks.

Consequently, the tunnel requires a scalable storage device. To address this, a three-tier hierarchical storage architecture encompassing local disks, NAS, and OSS (object storage in Alibaba Cloud) is deployed in DTS. This design thoughtfully balances storage latency for small data sets, curtails expenses, and meets the needs for scalable storage capacity.

The pub-sub system ensures at-least-once delivery semantics, and the Sink plugin will skip over duplicated records. By default, the tunnel retains data for 7 days before it is deleted. It is important to note that each source database is paired with an independent tunnel instance. This setup is highly elastic, given that instances can be created on-demand using cloud-based resources. Additionally, in the event of tunnel failures, all data can be reconstructed from the source database, ensuring strong resilience to failure.

3.1.3 Sink. Upon reaching the target system, the *Sink* component takes over. It is responsible for applying the data changes to the target database. This module must handle conflicts and reconcile data while adhering to the target database's schema and constraints. It should also be optimized for batch processing to minimize the latency and maximize throughput.

In the event of conflicts, if a batch transaction fails, the Sink will automatically revert to processing updates on a non-batch basis. Our conflict handling strategies are fully configurable. When conflicts occur, we allow users to choose among IGNORE, INTERRUPT and OVERWRITE options to handle the conflicts [21]. DTS employs best practices for logical replication. For instance, to address conflicts arising from retries, it executes update operations by combining delete and replace actions, replacing values as absolute rather than relying on statement replay. This approach ensures consistency. Furthermore, DTS defines various error codes along with corresponding retry strategies (e.g., resubmit, reconnect). Should failures persist, the process will ultimately escalate to require manual intervention, ensuring resilience and reliability in data handling.

The *Transaction Scheduler* module within the Sink leverages sophisticated conflict resolution techniques to bolster concurrent data processing, thereby enhancing the overall throughput. Although both the Source and Sink have transaction reordering modules — the transaction serializer and transaction scheduler, respectively — these two modules should not coexist on either the Source or the Sink. A Source can be linked to multiple Sinks; for example, an OLTP database can synchronize with both OLTP and OLAP databases. If the serializer function, which reorders transactions to follow the commit order, is placed on the Sink side, it will be executed multiple times. Furthermore, the outcomes of the transaction scheduler depend on runtime conditions and configurations, as OLTP and OLAP workloads might require different settings. Incorporating such specific requirements on the Source side would therefore be inappropriate. The mechanisms deployed within the Transaction Scheduler to enhance the performance of the Sink component are elaborated in §5.

Given that source data and the target database may often possess divergent data schemes, the Sink module may entail executing precise data transformations and mappings. This alignment guarantees that data is accurately conformed to the target schema. This crucial functionality is also facilitated by the database plugin component as detailed in the A2A (§3.3).

In addition, the DTS's *Rate Control* module, embedded within the Sink, enables each DTS link to adhere to precise line rates governed by the control plane, thus regulating the Sink's reading speed from the Tunnel.

3.1.4 Control Plane. The Control Plane plays a pivotal role in orchestrating the data transmission services, encompassing the initialization, scheduling, monitoring, billing, and termination of data migration or synchronization tasks.

The *Monitor* module vigilantly tracks the health and performance of the DTS instance, providing timely alerts and detailed logs for effective error resolution. The *Coordinator* module oversees the entire lifecycle of each DTS job, ensuring smooth execution and transitions between different stages. The *SCC (Serverless Controller Center)* module is responsible for managing the control flow concerning serverless operations, which will be expounded upon in the Section 4.2. The *Network Control Center* module executing the network allocation algorithm (§4.1).

3.2 Network Connectivity

In practice, the source and target databases, as well as DTS instances are often isolated within diverse network environments. For instance, source and target databases might be located within Alibaba's Virtual Private Cloud (VPC) [24], VPCs from other cloud providers, classic networks [19], and Internet Data Center (IDC). And DTS instances are typically deployed within a designated VPC, enabling us to enhance resource utilization through condensed scheduling practices. Meanwhile, customers have varying expectations regarding costs and data privacy. As a result, we have developed and offer a variety of bridging mechanisms to interconnect these network environments, each with tiered costs and levels of data privacy to meet diverse customer needs.

In the remainder of this section, we introduce the bridging mechanisms implemented in six representative scenarios. To present, Figure 4 depicts the network systems with the components, and network connections represented by directed arrows to indicate their initiation points.

S1: VPC to VPC. Figure 4 (a) illustrates the most common scenario in Alibaba Cloud, which involves transferring data from one VPC to another within the same region. In such cases, once authorized by the user, the DTS Source and Sink respectively establish network connections to the source and target database through VPC NAT gateways [17] (pink arrows).

S2: Classic network to VPC. Figure 4 (b) illustrates another common scenario, where the source database is hosted within classic networks. To access the source database, we deploy a cluster within the classic network to host DTS instances. Users can grant network access (yellow arrow) by reconfiguring their security groups accordingly.

S3: Cross region. Figure 4 (c) illustrates a scenario where the source and target databases are hosted in different geographic regions. In such cases, we leverage the Cloud Enterprise Network (CEN) service [4] (blue arrow) to facilitate high-speed transmission across regions. The DTS Source and Tunnel are typically co-located since they can be shared by tasks that involves transferring data to different regions. However, once the CEN becomes the bottleneck rather than the target database, we will co-locate Sink with Source and Tunnel. In this way, cross-region bandwidth consumption can be minimized, benefiting from the optimizations of Sink (see § 5).

S4: IDC/other cloud VPC to Alibaba Cloud. Figure 4 (d) illustrates a crucial scenario in which users aim to migrate data from their on-premises data centers (IDCs) or other cloud providers to Alibaba Cloud. Users have the option to set up an Express Connect [7] (green arrow) by laying a physical network cable from their sources to Alibaba Cloud's nearest access point. Alibaba Cloud offers a global network of access points [11] to support such connections, enabling direct, reliable, and high-bandwidth connectivity. In practice, the DTS access the Express Connect via a virtual switch (vSwitch) [16] within another Alibaba Cloud VPC, ensuring the security.

S5: Public Internet to Alibaba Cloud. In the scenario similar to **S4**, Figure 4 (e) illustrates a more cost-efficient approach where users make their databases accessible over the public network. This allows DTS Source to access the source database through public Internet connections (purple arrow). Despite the potential cost savings, this method is generally discouraged due to the increased risks to data security that are inherent with exposing sensitive data to the public Internet.

S6: Database gateway (DG). To reduce security risks associated with exposing databases to the public network and the high costs of Express Connect, Figure 4 (f) illustrates a method utilizing DG [5]. Users should deploy a DG agent in their private network. Concurrently, a DG server is deployed within an Alibaba Cloud VPC. The DG agent creates an encrypted connection (red arrow) to the DB server, while also accessing the source database over the private network. This approach enhances the security of data transmission, as it avoids exposing the database directly to the public network and ensures that data is encrypted during transit between the user's private network and Alibaba Cloud.

3.3 A2A Model

The A2A (Any-to-Any) Model transforms modification events from databases into a General Internal Representation (GIR), an intermediate format that is independent of any specific database or data format. This representation facilitates easier exchange of data across diverse database systems and applications, enhancing data portability and system interoperability.

The A2A Model enables the development of DTS to be modular, utilizing a framework and plugins that can be developed independently. The DTS framework focuses on the consistent handling of GIR, while database developers only need to concentrate on the creation and testing of plugins for their specific databases. Based on our real-world production experience, this development model has significantly reduced the time required to support new database types, from three weeks to just one week.

3.3.1 Compatibility. Designing a GIR that can accommodate the logical logs of various database systems demands a thorough and inclusive approach. It involves finding a common ground that encapsulates the shared characteristics while respecting the unique differences of each database system. In the context of Any2Any, data is abstracted into individual *records*, where each record is composed of a *Schema*, *Operation*, *BeforeImage*, and *AfterImage*. The

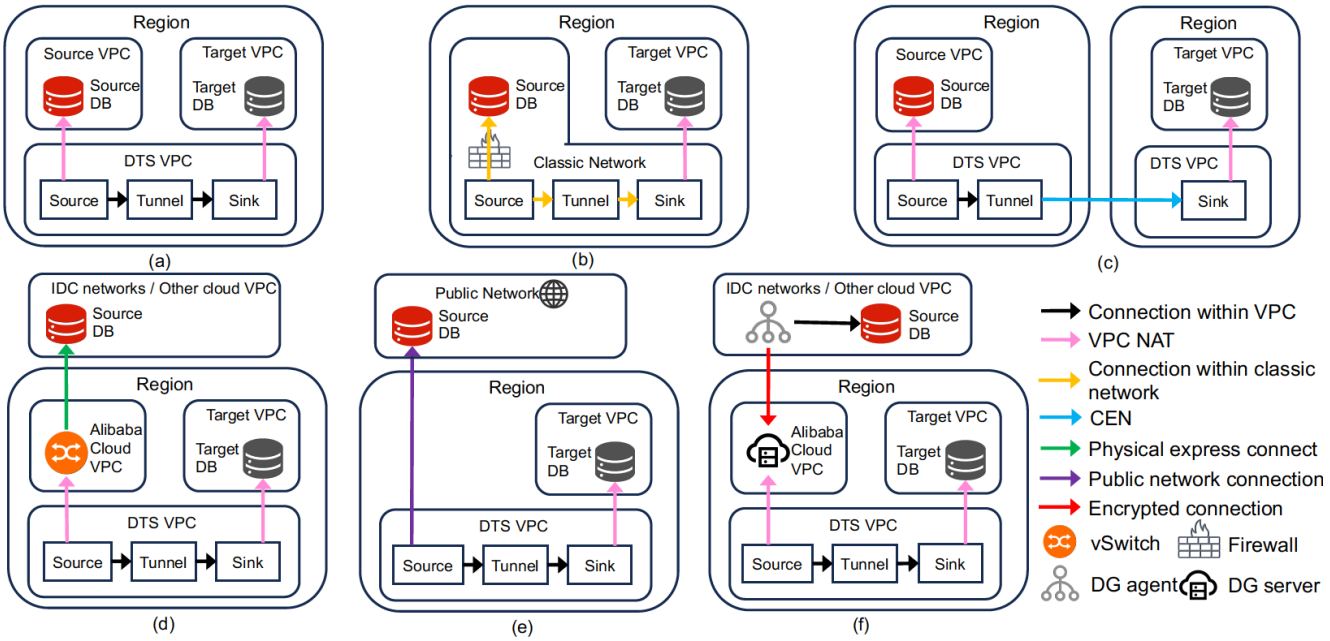


Figure 4: Six representative scenarios of DTS network.

Schema specifies the database and table to which the record belongs, along with column information. The Operation delineates the type of transaction performed, BeforeImage represents the data state prior to the operation, and AfterImage depicts the data state subsequent to the action. This design covers the shared logical log formats and ensures compatibility across multiple database systems. In situations where a BeforeImage is not available in the WAL, DTS will attempt to retrieve it using the capabilities of the source database (e.g., Flashback in Oracle, MongoDB, etc.). If this is not feasible, DTS will resort to using the default NoneValue in the record. This NoneValue will then be processed accordingly by the plugin of each target database.

3.3.2 Precision. In DTS, precision and the value range of each data type are imperative. Generally, the most basic data types fall into three categories: numeric, temporal, and character. For numeric data types, commonly used floating-point data types such as Double and Float are directly converted into the *Double* type. High-precision data types such as Number or Decimal are transformed into *BigDecimal*. Int/BigInt types are represented using *BigInteger* to cater to their varied signed/unsigned ranges. Temporal data types are addressed using *UnixTimestamp*, which represents time in milliseconds since the epoch, and *DateTime*, which incorporates timezone and offset information. Special data formats are converted into the BinaryObject type, encapsulated as byte array accompanied by an enum type. Similarly, textual data is converted into the StringValue type, consisting of byte array and a CharSet specification. Upon reaching the Sink, the handling of the byte array is determined according to its associated CharSet type. This approach ensures that the GIR can precisely capture the breadth of data types and their respective nuances as found in various database systems.

3.3.3 Extensibility via Database Plugins. The A2A framework conceptualizes each plugin as offering two distinct capabilities: data extraction through *AnySource* and data insertion via *AnySink*. These capabilities correspond to the specific functions the plugins perform within the data flow process. This abstraction not only simplifies the interaction with different databases but also ensures that each plugin is precisely aligned with the demands of the data transfer tasks it facilitates.

The DTS dynamically loads the appropriate plugin in line with the task's configuration. Within the plugin, a data transfer task is abstracted into a DataFlow. A DataFlow, in essence, is a configurable stream of data that is currently composed of three key components: SchemaFetcher, SchemaBlaster, and RecordRangeScheduler.

SchemaFetcher is responsible for retrieving the structural details of the source and destination databases, tables, and columns. SchemaBlaster takes on the role of segmenting the acquired tables into more manageable slices or partitions. RecordRangeScheduler then steps in to schedule these newly created table partitions for data transfer. The actual transfer of data from AnySource to AnySink operates at the granularity of these slices, known as RecordRanges. This granular approach ensures both the efficiency and manageability of data transfers. By structuring the architecture in this manner, the A2A framework attains significant extensibility.

4 RESOURCE SCHEDULE

This section presents our designs for resource scheduling in DTS, including bandwidth allocation (§4.1) and DTS serverless (§4.2).

4.1 Bandwidth Allocation

Within each Alibaba Cloud region, thousands of DTS transmission tasks may execute concurrently, creating a complex network of data flows that depend on a shared network infrastructure. Those include elephant flows of full tasks with lower-level SLOs (i.e., latency) and mouse flows of CDC tasks with higher-level SLOs. Sometimes, the rapidly increasing flow may surpass the bandwidth limit granted by Alibaba's internal network infrastructures, resulting in SLO violations of CDC tasks. This drives us to refine the scheduling process for network flows across all DTS Tunnels.

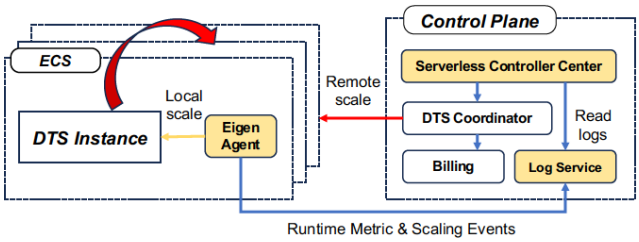


Figure 5: DTS serverless architecture

The domain of network scheduling has been extensively researched. Network scheduling [23] typically involves the following stages implemented in network routers: (1) *classify* that decides which waiting line the arriving packet will join, (2) *queue* that dictates the sequencing of packet transmission, such as Earliest Deadline First (EDF) [43] and Fixed Priority Scheduling (FPS) [27] and [44], and (3) *schedule* that manages the transmission rates by calling packets from queues (i.e., bandwidth allocation). The schedule algorithms include heuristic algorithms, such as Equal Share (i.e., round robin), Proportional Fair Share [29] and Max-Min Fair Share [33], and optimization-based algorithms, such as Hedera [26].

In our scenario, the challenge lies in the fact that traditional methods of network scheduling are largely dependent on the underlying network topology and the ability to measure, classify, and manage traffic at intermediate points within the network. Therefore, we chose to implement bandwidth allocation algorithms by periodically regulating the data fetching rate of the Sinks (i.e., the Rate Control module in Figure 3). In this paper, we present an optimization-based algorithm that minimizes a practically valuable self-defined metric *SLO violation*.

DTS Transmission Latency First, we define the latency of a DTS instance i at timestamp t:

$$\text{latency}_i = d_i + \frac{\max(0, (a_i - b_i)T)}{b_i} + \frac{\text{buffer}}{b_i} \tag{1}$$

The latency consists of three terms. The first term (i.e.,d_i) represents a constant latency that is caused by non-bandwidth issues (e.g., SQL analyzing and transaction processing). The second term represents the latency introduced through a period of T (a_i and b_i represent the rates of data generation and transmission). The final term corresponds to the queued data volume at timestamp t.

SLO Violation Based on latency_i, we then evaluate, then minimize the SLO violation by a hinge-loss function

$$\text{loss}_i = \alpha_i \max(0, \text{latency}_i - \text{SLO}_i) \tag{2}$$

where α_i is a loss weight that depends on the priorities of the tasks. The problems are solved by the classic quadratic programming solvers, such as Sequential Least Squares Programming (SLSQP) [38]. In practice, we improve the performance (to less than 10 seconds) of the solvers by proactively reserving bandwidth for a collection of mouse flows (e.g., less than 1 MBps).

4.2 DTS Serverless

In analyzing the online trouble ticket data for DTS instances, we have identified that 30% of events with high latency were attributed to insufficient resources, such as CPU and memory limitations. Normally, the events require the manual intervention of scaling up, which is operationally costly. This called for the enhancement of DTS's elasticity, leading to the introduction of the DTS serverless. This architecture offers automated scaling in response to workload fluctuations, enabling a pay-as-you-go pricing model that optimizes users' costs.

DTS Serverless supports both horizontal and vertical auto-scaling of the *Source* and *Sink* components. Specifically, *Source* can be scaled out/in through being paired with the partitioned tables in the source databases, and *Sink* can be scaled out/in through being paired with *Sources*. Furthermore, both *Source* and *Sink* can be scaled up/down based on the number of threads and CPU cores, allowing for flexible resource management depending on computational demands.

Implementation The architecture of DTS serverless is depicted in Figure 5. It is founded on Alibaba's cluster management system, Eigen [35]. Within Eigen, we leverage some critical functions, including Eigen Agent, Log Service, Serverless Controller Center (SCC), and then re-implement some components, such as DTS Coordinator and DTS Billing. In Figure 5, yellow boxes represent the components from Eigen, and white boxes represent the components re-implemented by DTS.

Within a cluster composed of Elastic Compute Service (ECS) nodes, an instance of the Eigen Agent is deployed on each node. The Eigen Agent plays a crucial role in the management of DTS instances (i.e., Kubernetes pods). It is tasked with the continuous monitoring of runtime metrics for each instance, making informed decisions regarding scaling, and executing resource scheduling tasks. The Eigen Agent within DTS supports two distinct categories of reactive auto-scaling: local scaling and remote scaling.

- **Local Scaling**: Depicted by the yellow arrow, this method involves adjusting the resource allocation on the current node where the DTS instance is running. If the instance requires more resources and there is sufficient capacity available on the local node, it can scale up without the need to move to a different node. After the completion of a scaling operation, the Eigen Agent communicates the outcome to the DTS Coordinator. This update is conveyed through the log service and SCC (blue arrows), ensuring that the DTS Coordinator is informed of the current state of resource allocation and can maintain an overall view of the system's performance and resource utilization.
- **Remote Scaling**: When the current node cannot accommodate the necessary scaling changes, for instance, when there is a lack of available resources for scaling up, the system resorts to remote scaling. Upon receiving a request from the Eigen Agent through the log service and SCC (blue arrows), the DTS Coordinator takes action to re-schedule the DTS instance remotely. As depicted by the red arrows, this involves migrating the DTS instance to another node within the cluster that has the required resources.

Strategy Events that trigger either local or remote scaling are dictated by specific strategies. The prevalent triggers are instances when CPU/memory utilization or processing speed, measured by rows per second (RPS), exceeds or descends below predetermined thresholds. By default, we apply the 90-50 rule: the DTS instance astutely escalates its resource allocation, increasing it twofold when the metric attains 90% of the upper limit; conversely, the DTS instance reduces its resource consumption by half when the metric drops to 50% of the established lower threshold. These thresholds

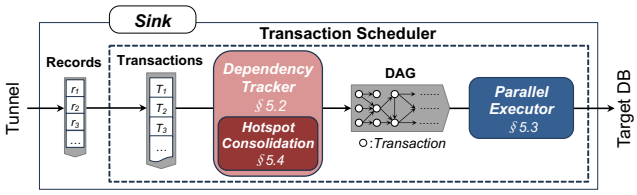


Figure 6: Transaction Scheduler Processing Flow in the Sink**.**

are set to ensure that the instances are running with optimal resource allocation, balancing performance and cost-efficiency. When a trigger event occurs, the Eigen Agent responds accordingly by initiating the appropriate scaling action to maintain system stability and performance.

5 SINK OPTIMIZATION

In the data transmission link of DTS, the computational models for the Source and Tunnel are predominantly IO-heavy, rarely becoming bottlenecks for throughput. However, rapidly writing to a replica database presents significant challenges. It is well recognized that the execution parallelism in a replica database is lower than that in the primary database [22, 30]. As a result, it's impossible to guarantee bounded replication lag under strong consistency for asynchronous database replication [30].

Consequently, the DTS service defaults to ensuring *eventual consistency* for CDC tasks. This means that updates to the source database will ultimately be synchronized with the target database. Eventual consistency allows for non-conflicting transactions to be executed in parallel and potentially out of order. This ensures that the parallelism of DTS writing to the target database can be greater than or equal to the parallelism of the source database, preventing unbounded replication lag.

To achieve rapid replaying of change operations to the target database, Sink utilizes a *Transaction Scheduler* which manages the transaction execution order and implements optimization strategies to boost throughput. The following sections will explore the optimization employed by Sink in detail.

5.1 Transaction Scheduler

The overall processing flow in the transaction scheduler is depicted in Figure 6. A continuous sequence of transactions is extracted from a stream of records retrieved through the Tunnel. During this process, transactions that exceed a predefined configuration limit—typically those containing thousands of records—are intentionally segmented into smaller, more manageable subsets. Such strategic division is crucial for preventing system congestion that could result from processing exceptionally large transactions. As transactions enter our system, the Dependency Tracker (§5.2) performs a thorough analysis of transaction dependencies. This analysis leads to creating of a Directed Acyclic Graph (DAG), which outlines the dependencies among the transactions awaiting replay.

Utilizing the DAG, the Parallel Executor (§5.3) is able to pinpoint and isolate groups of transactions devoid of conflicts. Such transactions are not only capable of being replayed in parallel — thereby maintaining data consistency — but they also gain from a variety of optimizations that users can tailor to their needs. Among these optimizations, batching stands out for its ability to significantly improve the efficiency of network communications and bolster the execution performance of transactions within the target database. In scenarios of synchronization of data from an Online Transaction Processing (OLTP) system to a data warehouse, many customers opt for the *TableAggregation* feature. This feature intelligently consolidates updates and groups insertions by table. Leveraging the inherent columnar organization of data warehouses, *TableAggregation* facilitates more efficient write operations and augments concurrency across tables.

Furthermore, for users who are less concerned with tracking the intermediate states of updates, the configuration option of hotspot consolidation (§5.4) presents a compelling solution. This optimization is designed to minimize frequent changes to specific data hotspots, thereby streamlining the update process and reducing the opportunity for performance bottlenecks.

The transaction scheduler adopts a single-producer-multiple-consumer model, consisting of a single *fetching thread* and multiple *working threads*. The fetching thread persistently retrieves records from the Tunnel, subsequently assembling them into a First-In-First-Out (FIFO) transaction queue, while simultaneously integrating these transactions into the dependency graph. In parallel, the working threads autonomously extract transactions from the dependency graph, executing them concurrently on the target database. This approach to parallelism significantly reduces synchronization latency and elevates throughput between Sink and the target database, enhancing overall system efficiency.

Algorithm 1: Dependency Tracker Procedure

Data: dependency graph G
Input: A transaction T extracted from Tunnel.
Function `TrackDependency(`T`)`**:**
  Add vertex T to G
  foreach *transaction* T_i *in graph* G **do**
    foreach *record* r_j *in* T *and records* r_k *in* T_i **do**
      if $(r_j.PK == r_k.PK)$ **or** $(r_j.UK == r_k.UK)$ **then**
        Add edge from T_i to T in G;
        break ; `// break internal for-loop`

5.2 Dependency Tracker

The Dependency Tracker is designed to identify transaction dependencies. These dependencies are primarily determined by conflicts involving the *primary keys (PK)* and *unique keys (UK)* of records between transactions, as these keys are critical in maintaining data integrity. For example, if two transactions, T_1 and T_2, both involve updates to records with the same primary key, and T_1 is executed before T_2 in the source database, they are considered to have an innerdependency and must be replayed to the target database in the original order. A directed acyclic graph (DAG) is employed to accurately map transactional interdependencies. In this graph, each vertex represents a distinct transaction, and an edge from vertex T_1 to vertex T_2 indicates that T_2 depends on the prior execution of T_1. Throughout the remainder of this paper, we use "DAG" and "dependency graph" interchangeably.

Algorithm 1 delineates the process by which the dependency graph is constructed. It accepts as input a transaction and an existing DAG and yields an updated DAG that incorporates the new

Algorithm 2: Parallel Executor Procedures

```
   Data: dependency graph G
1  Function ConflictFreeTxns():
2      T_set ← vertices having no incoming edges in G ;
3      foreach vertex v in T_set do
4          Remove v and its outgoing edges from G;
5      return T_set;
6  Function ParallelTask():
7      T_set ← ConflictFreeTxns(G);
8      parllel foreach transaction T_i ∈ T_set
9          Execute T_i ;
10 Function BatchTask():
11     T_set ← ConflictFreeTxns(G);
12     T_batch ← initialize new transaction;
13     foreach transaction T_i ∈ T_set do
14         T_batch ← T_batch ∪ T_i;
15     Execute T_batch;
16 Function TableAggregationTask():
17     T_set ← ConflictFreeTxns(G);
       // map from TableNames to Transactions
18     map ← ∅;
19     foreach transaction T_i ∈ T_set do
20         foreach record r ∈ T_i do
21             map[r.table].Append(r);
22     parllel foreach pair p in map
23         Execute map[p.table] ;
```

transaction. The key of this procedure is the identification of any pair of transactions that share a common PK or UK. Throughout this process, the records contained within the new transaction are examined, with their primary and unique keys being extracted. Upon discovering a dependency, an edge is added to the DAG, connecting the preexisting conflicting transaction to the new transaction. This connection signifies that the new transaction must be executed subsequently to maintain the integrity of the data.

5.3 Parallel Executor

The Parallel Executor retrieves a set of conflict-free transactions and executes them concurrently. Depending on the user's configuration, the executor can employ various strategies—or even a combination thereof—to expedite execution.

Algorithm 2 presents the pseudocode of these procedures. The function `ConflictFreeTxns` is designed to identify and process transactions that are primed for immediate parallel execution, as determined by the current state of the DAG. Specifically, a transaction with no incoming edges is independent of other transactions and, thus, eligible for concurrent execution. After such a transaction is retrieved from the graph, all its outgoing edges are also removed.

In simple scenarios, these conflict-free transactions are executed in parallel by the *task threads* , as demonstrated in the function `ParallelTask`. More commonly, however, the *BatchTask* is employed to pursue higher throughput. Batching is particularly efficacious in scenarios characterized by a multitude of small transactions. By consolidating these transactions into a single, larger transaction, batching substantially reduces the overhead associated with multiple database invocations.

The TableAggregationTask is specifically customized for scenarios where the target database is an OLAP system, taking full advantage of the system's ability to perform high-throughput operations on specific tables. This task adeptly capitalizes on the OLAP system's strengths by consolidating updates and batching insertions based on their target table. The TableAggregation feature effectively improves write performance by optimizing for the columnar storage architecture prevalent in data warehouses. Furthermore, it enhances table-level concurrency, thereby facilitating more efficient parallel processing of database operations.

5.4 Hotspot Consolidation

In scenarios where users are primarily concerned with the final state following a series of updates, we offer an optimization known as *Hotspot Consolidation*. This technique streamlines the update process by amalgamating multiple alterations to the same record into a single update. It significantly decreases the quantity of records that need to be replayed, which is particularly advantageous when there are frequent updates to a small subset of the data.

This feature must identify all updates eligible for consolidation while maintaining the correct order, achieving this with minimal overhead and without compromising parallelism. The dependency tracker satisfies all these conditions with only minor modifications. Consequently, when two conflicting transactions are detected at line 6 of algorithm 1, a further check determines if they can be consolidated. If they are eligible, the updates are merged accordingly.

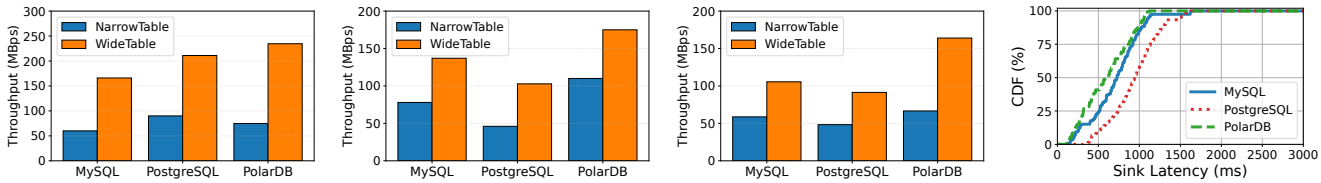
6 EVALUATION

In this section, we present the evaluation results of DTS. Our evaluation commences with a series of micro-benchmark experiments conducted within a controlled environment as detailed in §6.1. This initial phase is designed to preclude any environmental distortions, thereby enabling a precise assessment of the end-to-end performance as well as a detailed performance breakdown of individual DTS components. As it is impracticable to perform large-scale experiments that could potentially lead to violations of numerous DTS instances in a production setting, §6.2 involves simulations based on traces of online network traffic. Finally, we substantiate the practical effectiveness of DTS in providing serverless services and managing substantial data transfers through two case studies conducted within real-world production environments. Our evaluation focused on the following questions:

- How does DTS perform under heterogeneous databases, particularly in terms of read operations from the source database and write operations to the target database?
- What impact does the write optimization from Sink (§5) have on write operations in OLTP and OLAP databases?
- How is bandwidth effectively allocated to multiple DTS instances on cloud platforms to better satisfy users' SLOs?
- How does DTS perform in a production environment under spiky or massive transmission workloads?

6.1 Micro-Benchmark

6.1.1 Setup. Our micro-benchmarks utilized DTS instances and database instances acquired from Alibaba Cloud. The employed Alibaba RDS instances acted as the source databases and OLTP



(a) end to end throughputs for Full tasks (b) Source throughputs for CDC tasks (c) Sink throughputs for CDC tasks (d) latency distribution under 20MBps

Figure 7: DTS **Performance Across Three Database Types with NarrowTable and WideTable Configurations**

target databases, were configured with 8 vCPU cores and 32GB of RAM, hereinafter referred to as "8C32G". The default configuration for DTS is 1C1G for both the Source and Tunnel, and 4C16G for the Sink. The OLAP target databases utilize the more robust ADB [48] 24C32G configuration. These components were interconnected using dual VPC NATs as depicted in scenario (a) of Figure 4.

Workloads. Our experiments employed Sysbench [15] to generate database workloads, with 100 tables of 100,000 rows each. We evaluated two row sizes commonly found in actual workloads: NarrowTable (200-byte rows) and WideTable (100-KB rows), both of which consist of four columns. To replicate hotspot update scenarios accurately, we applied a Zipfian distribution to the update operations on each table, while the selection of tables was uniformly distributed. This setup is designed to emulate typical e-commerce situations where the stock of hot-sale products is frequently updated, and overall product data is distributed across multiple shards for load balancing.

6.1.2 Full Tasks. For full task transmission, the Source and Sink components are interconnected, as data pulled from the source database is immediately written into the target database via Sink (bypassing the necessity for a Tunnel), a standard configuration in database migration scenarios. Therefore, we use the end to end throughput (MBps, MB per second) for three different databases to evaluate the performance of full tasks. The results depicted in Figure 7(a) show that for NarrowTable, all three DTS instances can achieve a throughput exceeding 50 MBps. When it comes to WideTable, each instance can surpass 150 MBps in throughput. Notably, PostgreSQL exhibits the best performance for NarrowTable because DTS takes advantage of bulk loading via the `COPY` command, while PolarDB shows superior performance for WideTable.

6.1.3 CDC Tasks. We begin by presenting an ablation study on the individual submodules, demonstrating the throughput limitations separately at the Source and Sink ends. Subsequently, we investigate the latency on end-to-end transmission under heavy workloads.

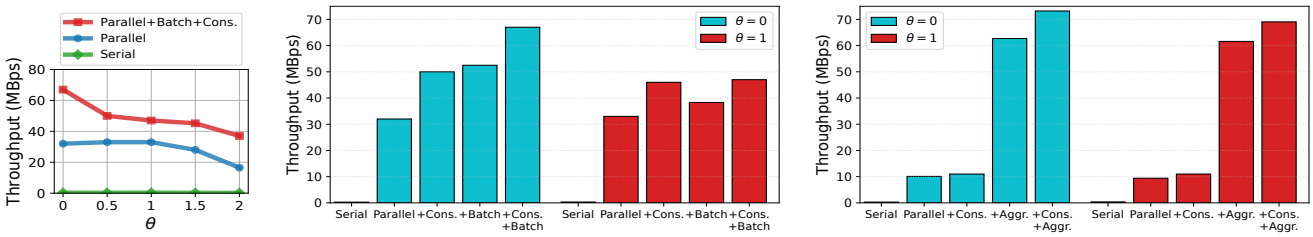
Source. Figure 7(b) presents the throughput of Source module for three different databases. Remarkably, even with a modest allocation of 1 core and 1 GB memory, Source can still achieve high throughput. For example, our Source can transfer data from MySQL at 78MBps, which meets or exceeds the needs of most deployments (according to daily statistics, 99% of running DTS instances exhibited a peak throughput of less than 36.5MBps). Moreover, to investigate the performance ceiling of DTS, for PostgreSQL we escalated the source RDS instance to 32 cores 128GB RAM to prevent the source database writing throughput from becoming the bottleneck. The results indicate that Source can efficiently capture change data and generate records at a high rate.

Sink. Figure 7(c) presents the throughput of the Sink module for three different databases. The sink module of PostgreSQL exhibits the best performance, reaching as high as 47.6 MBps. Notably, for PostgreSQL, the maximum throughput of Sink is higher than that of Source. This is because the *BatchTask* (see § 5.3) collects a batch of transactions to write at once, resulting in higher throughput.

End to End Latency. Figure 7(d) presents the latency distribution when the Sink throughput is 20MBps. It is evident that 90% of the records for MySQL and PolarDB can be written within 1 second, even under high throughput conditions. PostgreSQL exhibits a slightly higher latency (around 1.3 seconds) due to the performance related to the "Replication Slot" in the source database. Overall, this demonstrates that DTS can synchronize data in a timely manner.

6.1.4 Sink *optimization strategies.* We conducted performance tests under various Sink optimization strategies described in §5. The *Serial* mode writes to the target database only in a sequential manner; *Parallel* mode utilizes the dependency tracker detailed in (§5.2); *Batch* and *Consolidation* add the *BatchTask* (§5.3) and hotspot consolidation (§5.4) optimization on top of the dependency tracker, respectively; *Aggregation* mode employs the *TableAggregationTask* (§5.3). Figure 8(a) illustrates the throughput of DTS under various coefficient values θ of the Zipfian distribution. A θ value of 0 indicates a uniform distribution of updates across all entries, while a θ value of 2 indicates a high concentration of updates on specific entries. The *Serial* method yields a modest throughput that does not exceed 0.33 MBps, equivalent to an RPS (Records Per Second) of 882. In contrast, the *Parallel* mode consistently achieves a performance increase of more than 200 times. Moreover, with additional *Batch* and *Consolidation* optimizations, the throughput is doubled compared to the *Parallel* mode alone.

We conducted additional experiments on both OLTP and OLAP target databases. Figure 8(b) presents the results for the OLTP target database, MySQL. For OLTP, strategies such as consolidation and batching significantly increased throughput by approximately 1.6 times when updates were evenly distributed. Nevertheless, as update hotness rose (i.e., $\theta = 1$), the interdependence among transactions grew, reducing the number of non-conflicting transactions that could be batch-processed and, consequently, attenuating the benefits of batching on throughput. In contrast, our investigation into OLAP databases (e.g., ADB [48]), presented in Figure 8(c), revealed that the *Aggregation* mode was extremely effective, potentially amplifying data synchronization throughput to data warehouses by a factor of six. This significant enhancement is detailed



(a) Sink to MySQL varying skewness (b) Breakdown of Optimization: Sink to MySQL (c) Breakdown of Optimization: Sink to ADB

Figure 8: DTS **performance under various** Sink **optimizations, using MySQL as sources.**

in §5.3. It is important to note that OLAP database performance, particularly in terms of writing to the 100 tables involved in our experiments, faced inherent limitations without the use of *Aggregation* mode. This limitation resulted in a less discernible contrast in throughput benefits between hotspot consolidation and varied update concentrations.

6.2 Simulation on Network Trace

We conducted simulation experiments to evaluate bandwidth allocation algorithms using a dataset of 3983 real-world CDC tasks, of which 122 had data generation rates above 1 MBps, demanding a total bandwidth of 921 MBps. And the remaining 3861 tasks required 183 MBps. Our experimental design involved two stages: initially ensuring bandwidth for the majority of lower-rate tasks. And next, applying different bandwidth allocation algorithms among the 122 high-rate tasks with a quantity of full tasks. The SLO for CDC tasks was set at 60 seconds, while full transmission tasks, averaging 500 GB in size, had an SLO of 1 hour. All CDC tasks have been assigned a cost weight of 1, while full tasks are assigned lower weights to reflect their relative priority. The period is set at 10 seconds. Algorithm performance is evaluated across three dimensions: bandwidth limitation, buffer ratio, and full-CDC throughput ratio. Specifically, the buffer ratio of a CDC task i is defined as $\frac{\text{buffer}}{a_i \times \text{SLO}_i} \times 100\%$, where a_i represents data generating rate.

Figure 9 illustrates the impact of varying bandwidth limits on (a) average loss and (b) normalized latency, under a fixed buffer ratio constraint of 100% with 0 full transmission task. The red dotted lines mark the sum of data generation rate of all CDC tasks. Figure 9 (c) and (d) illustrate the effects of buffer ratios on the average loss and normalized latency, respectively, under a fixed bandwidth constraint of 800 MBps with 0 full transmission task. Figure 9 (e) and (f) present the impact of changing the full-CDC throughput ratio on the same metrics, under the bandwidth constraint of 800 MBps and a buffer ratio constraint of 100%.

The data show that the DTS Optimization-Based Scheduling approach consistently results in lower loss. However, it does not consistently outperform the Max-Min Fair Share method regarding normalized latency (e.g., when the full-CDC ratio is 80%). It is reasonable since the optimization-based scheduling algorithm prioritizes loss over latency. This approach seeks to distribute bandwidth more equitably, tolerating increased latency provided it remains within the bounds of the specified SLO. Moreover, both Max-Min Fair Share and DTS Optimization-Based Scheduling exhibit improvements over Proportional Fair Share algorithm, which was the baseline of DTS, across these metrics.

6.3 Production Case Studies

6.3.1 Severless Case. In Figure 10, we showcase the performance of a serverless DTS instance in production over a 24-hour period. We record the throughput, measured in Rows Per Second (RPS) and depicted in blue, alongside the instance's RPS upper threshold, shown in orange. Additionally, we present the latency in the lower portion of the figure. The orange line largely covers the blue line, indicating that the automatic scaling system efficiently handles the load, while the latency trend remains consistent. A notable exception occurs around 5 AM when a spike in latency coincides with batch maintenance tasks on the source database, causing a lag on the source side. This event triggers a rapid fourfold increase in resource allocation within one minute, successfully managing the higher workload and preventing further delays in transmissions. This case study underscores the remarkable efficiency of the serverless DTS instance: it utilizes only 25.5% of the resource capacity required by a large-scale DTS instance that runs continuously, while maintaining end-to-end latency below one second in over 99.8% of cases.

6.3.2 Massive Data Case. Figure 11 illustrates DTS's proficiency in managing change data transmission across regional databases over a 24-hour period. The graphed throughput, measured in MBps, alongside the corresponding latency, highlights the system's capability to handle large volumes of data. Notably, throughput spikes dramatically at midnight, coinciding with scheduled database maintenance activities such as data cleaning. Furthermore, the system adeptly copes with significant throughput increases—reaching 75MBps around noon and in the evening—as a result of frequent updates from the source databases. Despite these variations, DTS consistently keeps latency under 4 seconds, affirming its ability to sustain seamless real-time data transmission. This reliable performance emphasizes DTS's robustness in fulfilling the requirements of heavy data transmission workloads, ensuring that data remains up-to-date and accessible within the target databases.

7 RELATED WORK

Commercial Database Transmission Service on Cloud. Several cloud providers offer database transfer services. Amazon Web Services (AWS) [2], Microsoft Azure [3], and Google Cloud [9] all provide Database Migration Services (DMS) that support both

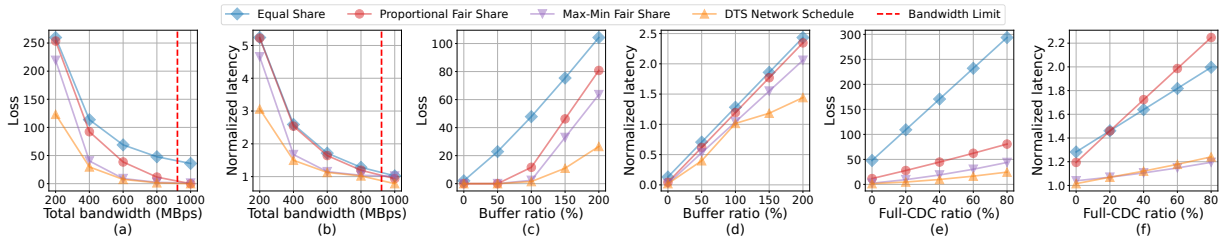


Figure 9: The performance of different bandwidth allocation algorithms.

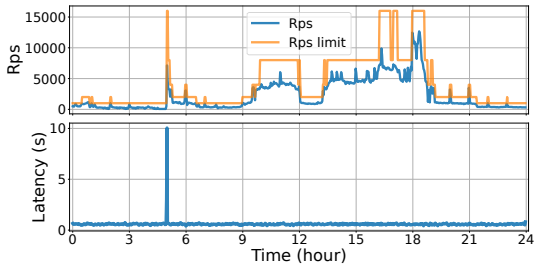


Figure 10: A serverless DTS instance over a 24-hour period.

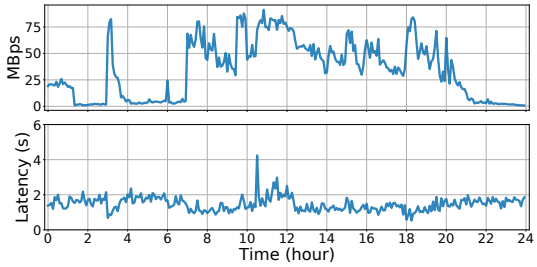


Figure 11: Real DTS performance for a 24-hour period.

homogeneous and heterogeneous migrations, as well as change data replication. Furthermore, Oracle's GoldenGate [13] facilitates high-performance data replication and can be utilized beyond the confines of the Oracle ecosystem. In contrast to these offerings, Alibaba's DTS distinguishes itself in terms of supported databases, network environments, architectural design, and functionality. For example, AWS's DMS requires a dedicated *replication instance* for each migration task, which is responsible for extracting data from the source database and loading it into the target database. Conversely, DTS deconstructs the process into three discrete modules: *Source*, *Tunnel*, and *Sink*. This modular architecture allows for each component to be deployed independently, thus aligning with the specific requirements of Alibaba Cloud's clientele.

Three party companies specializing in data integration and management such as Informatica [10], Fivetran [8], and Qlik [14] also offer database transmission services on cloud, with a particular focus on data integration and Extract-Transform-Load (ETL) processes. Debezium is an open-source tool for Change Data Capture (CDC) and data synchronization. While Debezium and DTS share similar functionalities in data synchronization, DTS supports more complex data synchronization and transmission scenarios, including data migration, data synchronization, data subscription, and real-time data processing. Additionally, as part of a cloud service, DTS provides a comprehensive commercial solution with high functionality integration, making it suitable for enterprise-grade applications with a user-friendly graphical interface and automation tools. In contrast, Debezium still requires considerable manual tuning and configuration for optimal performance.

Database Replication and Synchronization. To provide efficiency database replication in distributed environment and reduce the lag gap of primary/backup database , a lot of concurrency control protocol and system have emerged in both academia and industry [28, 30, 31, 40–42, 47]. These work proposed novel protocol and designs which require modifying the original database systems or proposed new database system. In contrast, our work aims to provide a generic solution to efficiently interconnect any source database with any target database, independent of reliance on native support from the databases themselves.

Cloud Resources Management. The scheduling of resources within cloud environments to facilitate database replication for multi-tenant architectures, while adhering to Service Level Objective (SLO) requirements, is studied in [37, 45]. Additionally, [36] explores resource management in cloud networking through economic and pricing perspectives. As a cloud-based service, DTS primarily concentrates its resource management on bandwidth allocation and serverless architecture to ensure reliable and high-speed transmission. This focus is designed to meet the diverse requirements of multiple tenants and to manage the bandwidth demands of simultaneous transmission tasks effectively.

8 SUMMARY

This paper provides a comprehensive analysis of Alibaba's Data Transmission Service (DTS), a database infrastructure that facilitates data migration and real-time data synchronization. We highlighted the novel Any-to-Any (A2A) architecture which simplifies the complexity of data transmission across heterogeneous databases and under various network connectivity scenarios. Additionally, we examine a dynamic network bandwidth scheduling algorithm that effectively maintains Service-Level Objectives (SLOs), complemented by a serverless mechanism that ensures efficient resource utilization. We detail key performance optimization strategies, such as transaction dependency tracking, hot data consolidation, and batching, which are proven to significantly boost the efficiency of DTS. Our evaluation results show that DTS can handle real-time data transmission effectively and efficiently in both experimental and production environments.

REFERENCES

[1] [n.d.]. Alibaba Cloud Database Services Empower Your Business. https://www.alibabacloud.com/en/product/databases Accessed: 2024-04-30.

[2] [n.d.]. AWS Database Migration Service. https://aws.amazon.com/dms/ Accessed: 2024-04-30.

[3] [n.d.]. Azure Database Migration Service. https://azure.microsoft.com/products/database-migration Accessed: 2024-04-30.

[4] [n.d.]. Cloud Enterprise Network (CEN). https://www.alibabacloud.com/product/cen Accessed: 2024-04-30.

[5] [n.d.]. Database Gateway. https://www.alibabacloud.com/help/en/database-gateway Accessed: 2024-04-30.

[6] [n.d.]. DB-Engines Ranking. https://db-engines.com/en/ranking Accessed: 2024-04-30.

[7] [n.d.]. Express Connect. https://www.alibabacloud.com/product/express-connect Accessed: 2024-04-30.

[8] [n.d.]. Fivetran. https://www.fivetran.com/ Accessed: 2024-04-30.

[9] [n.d.]. Google Cloud Database Migration Service. https://cloud.google.com/database-migration Accessed: 2024-04-30.

[10] [n.d.]. informatica. https://www.informatica.com/ Accessed: 2024-04-30.

[11] [n.d.]. Locations of access points. https://www.alibabacloud.com/help/en/express-connect/user-guide/locations-of-access-points Accessed: 2024-04-30.

[12] [n.d.]. MySQL Replication. https://dev.mysql.com/doc/mysql-replication-excerpt/5.7/en/ Accessed: 2024-04-30.

[13] [n.d.]. Oracle GoldenGate. https://www.oracle.com/integration/goldengate/ Accessed: 2024-04-30.

[14] [n.d.]. Qlik. https://www.qlik.com/ Accessed: 2024-04-30.

[15] [n.d.]. sysbench. https://github.com/akopytov/sysbench Accessed: 2024-04-30.

[16] [n.d.]. VPCs and vSwitches. https://www.alibabacloud.com/help/en/vpc/user-guide/vpcs-and-vswitchs/ Accessed: 2024-04-30.

[17] [n.d.]. What is a VPC NAT gateway. https://www.alibabacloud.com/help/en/nat-gateway/user-guide/what-is-a-vpc-nat-gateway Accessed: 2024-04-30.

[18] 2024. AWS DMS: Challenges & Solutions Guide. https://www.integrate.io/blog/aws-dms-challenges-solutions-guide/ Accessed: 2024-04-30.

[19] 2024. Classic Network. https://www.alibabacloud.com/help/en/ecs/user-guide/classic-network Accessed: 2024-04-30.

[20] 2024. Data Transmission Service: Data Migration and Synchronization - Alibaba Cloud. https://www.alibabacloud.com/en/product/data-transmission-service Accessed: 2024-04-30.

[21] 2024. DTS-Reserve parameter. https://www.alibabacloud.com/help/en/dts/developer-reference/reserve-parameter-description Accessed: 2024-04-30.

[22] 2024. Parallel Recovery - PostgreSQL wiki. https://wiki.postgresql.org/wiki/Parallel_Recovery Accessed: 2024-04-30.

[23] 2024. Queuing and Scheduling. https://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst9400/software/release/16-6/configuration_guide/qos/b_166_qos_9400_cg/b_166_qos_9400_cg_chapter_01.html#concept_whp_jdb_p1b Accessed: 2024-04-30.

[24] 2024. Virtual Private Cloud (VPC). https://www.alibabacloud.com/product/vpc Accessed: 2024-04-30.

[25] Daniel Abadi, Rakesh Agrawal, Anastasia Ailamaki, Magdalena Balazinska, Philip A. Bernstein, Michael J. Carey, Surajit Chaudhuri, Jeffrey Dean, AnHai Doan, Michael J. Franklin, Johannes Gehrke, Laura M. Haas, Alon Y. Halevy, Joseph M. Hellerstein, Yannis E. Ioannidis, H. V. Jagadish, Donald Kossmann, Samuel Madden, Sharad Mehrotra, Tova Milo, Jeffrey F. Naughton, Raghu Ramakrishnan, Volker Markl, Christopher Olston, Beng Chin Ooi, Christopher Ré, Dan Suciu, Michael Stonebraker, Todd Walter, and Jennifer Widom. 2016. The Beckman report on database research. *Commun. ACM* 59, 2 (2016), 92–99. https://doi.org/10.1145/2845915

[26] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. 2010. Hedera: dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation* (San Jose, California) *(NSDI'10)*. USENIX Association, USA, 19.

[27] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. 1991. Hard Real-Time Scheduling: The Deadline-Monotonic Approach. *IFAC Proceedings Volumes* 24, 2 (1991), 127–132.

[28] Dennis Butterstein, Daniel Martin, Knut Stolze, Felix Beier, Jia Zhong, and Lingyun Wang. 2020. Replication at the speed of change: a fast, scalable replication solution for near real-time HTAP processing. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3245–3257. https://doi.org/10.14778/3415478.3415548

[29] Jon Crowcroft and Philippe Oechslin. 1998. Differentiated end-to-end Internet services using a weighted proportional fair sharing TCP. *ACM SIGCOMM Computer Communication Review* 28, 3 (1998), 53–69.

[30] Jeffrey Helt, Abhinav Sharma, Daniel J. Abadi, Wyatt Lloyd, and Jose M. Faleiro. 2022. C5: Cloned Concurrency Control That Always Keeps Up. *Proc. VLDB Endow.* 16, 1 (2022), 1–14. https://doi.org/10.14778/3561261.3561262

[31] Chuntao Hong, Dong Zhou, Mao Yang, Carbo Kuo, Lintao Zhang, and Lidong Zhou. 2013. KuaFu: Closing the parallelism gap in database replication. In *ICDE 2013*, Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou (Eds.). IEEE Computer Society, 1186–1195. https://doi.org/10.1109/ICDE.2013.6544908

[32] IBM. [n.d.]. Change-capture replication and full-refresh copying. https://www.ibm.com/docs/en/idr/10.2.1?topic=tables-change-capture-replication-full-refresh-copying Accessed: 2024-04-30.

[33] Srinivasan Keshav and S Kesahv. 1997. *An engineering approach to computer networking: ATM networks, the Internet, and the telephone network*. Vol. 116. Addison-Wesley Reading.

[34] Feifei Li. 2019. Cloud-native database systems at Alibaba: opportunities and challenges. *Proc. VLDB Endow.* 12, 12 (aug 2019), 2263–2272. https://doi.org/10.14778/3352063.3352141

[35] Ji You Li, Jiachi Zhang, Wenchao Zhou, Yuhang Liu, Shuai Zhang, Zhuoming Xue, Ding Xu, Hua Fan, Fangyuan Zhou, and Feifei Li. 2023. Eigen: End-to-End Resource Optimization for Large-Scale Databases on the Cloud. *Proc. VLDB Endow.* 16, 12 (aug 2023), 3795–3807. https://doi.org/10.14778/3611540.3611565

[36] Nguyen Cong Luong, Ping Wang, Dusit Niyato, Yonggang Wen, and Zhu Han. 2017. Resource Management in Cloud Networking Using Economic Analysis and Pricing Models: A Survey. *IEEE Commun. Surv. Tutorials* 19, 2 (2017), 954–1001. https://doi.org/10.1109/COMST.2017.2647981

[37] Riad Mokadem and Abdelkader Hameurlain. 2020. A data replication strategy with tenant performance and provider economic profit guarantees in Cloud data centers. *Journal of Systems and Software* 159 (2020). https://doi.org/10.1016/J.JSS.2019.110447

[38] Jorge Nocedal and Stephen J Wright. 2006. Quadratic programming. *Numerical optimization* (2006), 448–492.

[39] Olga Poppe, Qun Guo, Willis Lang, Pankaj Arora, Morgan Oslake, Shize Xu, and Ajay Kalhan. 2022. Moneyball: Proactive Auto-Scaling in Microsoft Azure SQL Database Serverless. *Proc. VLDB Endow.* 15, 6 (feb 2022), 1279–1287. https://doi.org/10.14778/3514061.3514073

[40] Lin Qiao, Kapil Surlaker, Shirshanka Das, Tom Quiggle, Bob Schulman, Bhaskar Ghosh, Antony Curtis, Oliver Seeliger, Zhen Zhang, Aditya Auradkar, Chris Beaver, Gregory Brandt, Mihir Gandhi, Kishore Gopalakrishna, Wai Ip, Swaroop Jagadish, Shi Lu, Alexander Pachev, Aditya Ramesh, Abraham Sebastian, Rupa Shanbhag, Subbu Subramaniam, Yun Sun, Sajid Topiwala, Cuong Tran, Jemiah Westerman, and David Zhang. 2013. On brewing fresh espresso: LinkedIn's distributed data serving platform. In *SIGMOD 2013*. ACM, 1135–1146. https://doi.org/10.1145/2463676.2465298

[41] Dai Qin, Angela Demke Brown, and Ashvin Goel. 2021. Caracal: Contention Management with Deterministic Concurrency Control. In *SOSP 2021*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 180–194. https://doi.org/10.1145/3477132.3483591

[42] Dai Qin, Ashvin Goel, and Angela Demke Brown. 2017. Scalable Replay-Based Replication For Fast Databases. *Proc. VLDB Endow.* 10, 13 (2017), 2025–2036. https://doi.org/10.14778/3151106.3151107

[43] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. 1990. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Trans. Comput.* 39, 9 (1990), 1175–1185.

[44] Cha Hwan Song, Xin Zhe Khooi, Raj Joshi, Inho Choi, Jialin Li, and Mun Choon Chan. 2023. Network Load Balancing with In-network Reordering Support for RDMA. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 816–831.

[45] Flávio R. C. Sousa and Javam C. Machado. 2012. Towards Elastic Multi-tenant Database Replication with Quality of Service. In *IEEE Fifth International Conference on Utility and Cloud Computing*. IEEE Computer Society, 168–175. https://doi.org/10.1109/UCC.2012.36

[46] Rebecca Taft, Nosayba El-Sayed, Marco Serafini, Yu Lu, Ashraf Aboulnaga, Michael Stonebraker, Ricardo Mayerhofer, and Francisco Andrade. 2018. P-Store: An Elastic Database System with Predictive Provisioning. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 205–219. https://doi.org/10.1145/3183713.3190650

[47] Guozhang Wang, Lei Chen, Ayusman Dikshit, Jason Gustafson, Boyang Chen, Matthias J. Sax, John Roesler, Sophie Blee-Goldman, Bruno Cadonna, Apurva Mehta, Varun Madan, and Jun Rao. 2021. Consistency and Completeness: Rethinking Distributed Stream Processing in Apache Kafka. In *SIGMOD 2021*. ACM, 2602–2613. https://doi.org/10.1145/3448016.3457556

[48] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, and Chengliang Chai. 2019. AnalyticDB: real-time OLAP database system at Alibaba cloud. *Proc. VLDB Endow.* 12, 12 (aug 2019), 2059–2070. https://doi.org/10.14778/3352063.3352124