A Sneak Peek at RisingWave: a Cloud-Native Streaming Database

Yanghao Wang, Zhi Liu
Singularity Data Inc
United State
{yanghao,zhi}@singularity-data.com

ABSTRACT

This paper presents RisingWave, a new cloud-native streaming database under development. RisingWave's mission is to democratize stream processing: to make stream processing simple, affordable, and accessible. To achieve that, RisingWave treats streams as first-class citizens in a database and uses PostgreSQL compatible SQL interface. Users can create their streaming analytical task by defining materialized views. The views are incrementally maintained in the RisingWave streaming engine, and are ready to be queried directly without exporting to an external system. RisingWave is designed for the cloud. The tiered architecture decouples its components such that each component scales independently to fully leverage the cloud infrastructure and reduce the cost. We demonstrate how RisingWave can simplify users' real-time analytical missions in the modern data stack.

CCS CONCEPTS

• **Information systems → Stream management**.

KEYWORDS

Stream processing; Streaming databases

ACM Reference Format:
Yanghao Wang, Zhi Liu. 2022. A Sneak Peek at RisingWave: a Cloud-Native Streaming Database. In *The 16th ACM International Conference on Distributed and Event-based Systems (DEBS '22), June 27–30, 2022, Copenhagen, Denmark.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3524860.3543284

1 INTRODUCTION

Stream processing is a well-studied topic. Past efforts are largely devoted to developing fast, parallel, scalable, and reliable streaming systems, such as Apache Flink [9] and Spark streaming [12]. Thanks to these efforts, today's stream processing systems are running smoothly to power the real-time analytical requests across the globe, hosting thousands of applications covering advertisement recommendation, fraud detection, IoT analytics, and many others in small businesses. Can we jump to the conclusion that stream processing is a solved problem? Unfortunately, the answer is negative, as many small businesses are still complaining about the high cost of adopting streaming systems, for at least two sets of reasons:

(1) Difficult to learn. Unlike DBMSes that provide SQL as their interface, most streaming systems require users to learn a set of low-level programming APIs to manipulate the streaming data. To make things worse, typically streaming systems represent data in their raw format instead of the relational model. Users write complicated logic to transform data between streaming systems and databases.

(2) Expensive to operate. Although comprehensive scripts and Docker images are usually available for automatic deployment, the cost of deploying and maintaining a streaming system is still disastrous. Typically companies have to purchase many machines to sustain the smoothness of the cluster for the worst-case scenarios of a fluctuated workload. Furthermore, assembling an operative team to maintain the system regularly can also be challenging for hiring.

RisingWave. This paper presents RisingWave, a new cloud-native streaming database to tackle both problems. The mission of RisingWave is democratizing stream processing, making stream processing simple, affordable, and accessible to everyone.

Stream processing made simple. RisingWave provides PostgreSQL compatible SQL interface and can be seamlessly integrated with the PostgreSQL ecosystem with no code change. RisingWave treats streams as first-class citizens and allows users to compose complex queries over both streaming data and historical data declaratively and elegantly. Anyone with database experience can easily define their streaming computation tasks in SQL, without worrying about learning Java or system-specific low-level APIs.

Stream processing made affordable. RisingWave is designed for the cloud. It adopts a tiered architecture: the core components of RisingWave are fully decoupled into separate layers of services, including serving layer, meta service, compute nodes, compaction service and storage. This architecture opens the opportunity for different services to scale independently based on its own usage, without the waste of resources in the traditional architecture where computation and storage are coupled.

Stream processing made accessible. RisingWave is built from scratch in Rust, and now is open-sourced [7] under Apache License 2.0. Currently, RisingWave is still under heavy development. Everyone can participate in the design of the RisingWave project roadmap and everyone can contribute and send feedback to the community.

Organization. The rest of this paper organizes as follows. Firstly section 2 gives an overview of RisingWave, including the application scenario and the system architecture. Secondly, section 3 elaborates on the technical design of each component in RisingWave. Finally,

DEBS '22, June 27–30, 2022, Copenhagen, Denmark
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9308-9/22/06…$15.00
https://doi.org/10.1145/3524860.3543284

Section 4 describes our demonstration plan for our viewers.

2 RISINGWAVE OVERVIEW

In this section, we give an overview of RisingWave. RisingWave is a cloud-native streaming database. It treats the stream as a first-class citizen in a database and can be declared in an extended SQL grammar. After connecting to upstream stream sources, users can create their streaming analytical task by simply defining materialized views, which are incrementally and efficiently maintained in the RisingWave streaming engine. Then the views are ready to serve users' queries immediately.

PostgreSQL Interface. RisingWave targets to fully support the PostgreSQL interface. In particular, this includes: (1) The PostgreSQL relational model. RisingWave supports most PostgreSQL data types, including basic types and compositional types. (2) Query statements. Currently, RisingWave supports a large set of PostgreSQL standard user queries, including 20 of 22 TPC-H queries. (3) Client protocol. Users can use their installed psql clients to access RisingWave directly without installing an extra client.

Sources. Sources are upstream systems that RisingWave can read data from. Currently, RisingWave accepts data from sources like Apache Kafka [1], Apache Pulsar [2], AWS Kinesis [3], Redpanda [6], and Debezium [4] Change Data Capture. After defining the sources and establishing the connection, RisingWave starts reading from sources and processing them continuously. RisingWave accepts both stream data in JSON, Protobuf, and Avro format, and the raw streaming data will be transformed into relational data directly during their ingestion.

Materialized views. RisingWave allows users to define materialized views over both tables and sources, such that the materialized views capture the accumulative analytical result of the streams. In addition, RisingWave extends its SQL grammar and natively supports streaming window functions over sources, including the tumbling window, hop window, and sliding window. RisingWave periodically refreshes materialized views incrementally using its high-performance distributed streaming engine, and users can query those materialized views anytime directly to extract analytical insights from up-to-date data, without dumping results to an external system.

Consistency. RisingWave guarantees *snapshot based consistency* over materialized views, such that users can expect sensible answers from queries, without worrying the erroneous answer due to the eventual consistency [11]. In particular the guarantee is twofold, and we give formal definitions below. (1) Given a query Q at timestamp t, the system returns a result S such that $Q(D_{t'})$ is *consistent* with a *snapshot* $D_{t'}$ at timestamp $t^{'}$, where $t^{'} \leq t$. Here we say D_t is a snapshot at timestamp t if it includes all tuples ingested into the system before t, and exclude all tuples after time t, and a query result S is consistent with $D_{t'}$ if the result is equivalent with evaluating Q on $D_{t'}$ directly. (2) For any two timestamps t_1 and t_2 with $t_1 \leq t_2$, assume the result for Q on t_1 and t_2 are consistent with snapshots $D_{t_1'}$ and $D_{t_2'}$ on $t_1^{'}$ and $t_2^{'}$ respectively, then $t_1^{'} \leq t_2^{'}$. That is, later queries will always get results consistent with later snapshots.

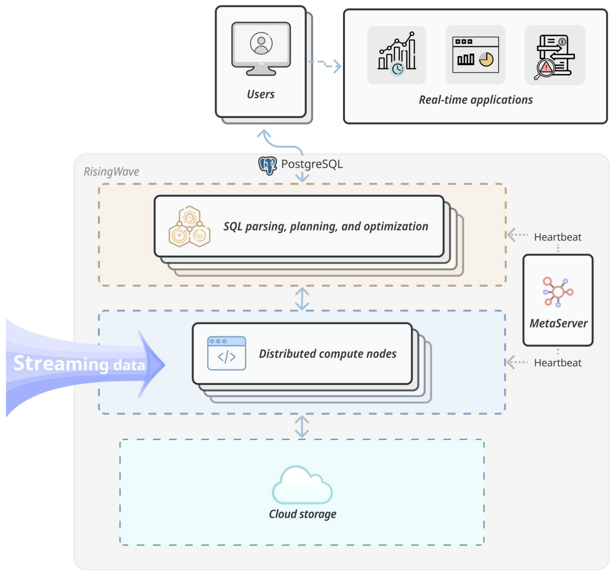

Figure 1: The architecture of RisingWave

3 SYSTEM DESIGN

In this section we describe the detailed technical designs of RisingWave. We start with the architecture, and elaborate the details in each component respectively.

3.1 Architecture

The overall architecture of RisingWave is depicted in Figure 1. RisingWave consists of two computation engines in its kernel: the batch engine and the streaming engine. The batch engine responds to user issued ad-hoc queries via traditional query processing techniques, while the streaming engines builds workflows to continuously update materialized views. To host both computation engines, RisingWave decouples its kernel into a few sets of services: frontend, compute engine, storage service, compaction service, and meta service.

3.2 Frontend nodes

At the serving layer is a set of frontend nodes. These nodes connect to PostgreSQL clients and answer users' requests directly. Each frontend node hosts its independent SQL parser, query planner and query scheduler. For metadata used during the planning phase, the frontend periodically receives the update of metadata from the centralized meta service via push-based notifications. Therefore the planner can access the local cache on the frontend directly, without pulling the metadate from meta service.

After parsing via a unified parser, different SQL statements are redirected into batch engine and streaming engine accordingly based on their functionality. For ad-hoc queries, the frontend works in three steps. (1) Generating a logical plan from the parsed abstract syntax tree. (2) Optimizing the logical plan into an execution plan. And (3) scheduling the execution plan on compute nodes. For creating materialized view statements, the frontend builds a streaming workflow also in three steps. (1) Generating a logical plan. This step is shared with the batch engine. (2) Building a streaming workflow

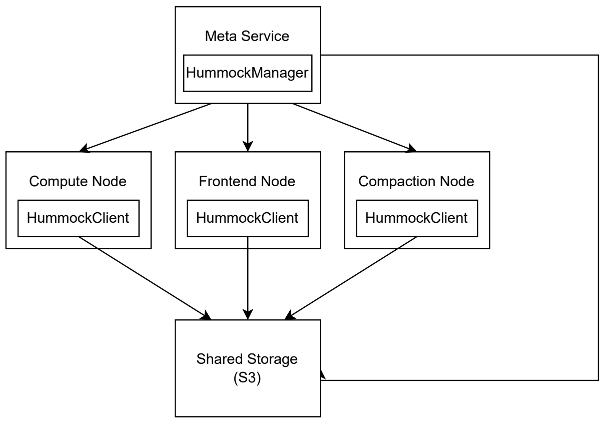

Figure 2: The architecture of Hummock

from the logical plan. In this step, the logical streaming plan is fragmented and mapped into streaming executors. Then executors are duplicated for parallel execution. (3) Scheduling and deploying the workflow on the compute nodes.

3.3 Compute nodes

At the core of RisingWave kernel is its compute nodes. The cluster of compute nodes host both the batch engine and the streaming engine.

Batch engine. The batch engine follows the modern design for massive parallel query processing. This include a vectorizied query execution engine and an exchange service for shuffling data between compute nodes. Queries are also evaluated on each compute node in multi-threads.

Streaming engine. The RisingWave streaming engine is based on actor model [10] in concurrent programming. The frontend builds a set of actors, such that each actor has no internal parallelism, and responds to the received messages according to its local states only. Therefore the streaming engine is inherently concurrent and fully takes advantage of multi-core CPUs via Rust asynchronous runtime.

Checkpoint. RisingWave uses Asynchronous Barrier Snapshot [8] to generate consistent snapshots periodically for two purposes: refreshing materialized views and failure recovery. Therefore RisingWave guarantees that the accessed data from the storage is always consistent when querying materialized views. For failure recovery, whenever the streaming engine crashes, the system globally rollback to a previous consistent snapshot. During the recovery process, the system rebuilds all actors and resets their states to the last consistent checkpoint.

3.4 Storage service

We build Hummock, a cloud-native storage service, that provides the unified key-value interface for storing and accessing data in RisingWave. By combining shared cloud storage with tiered caches, Hummock guarantees the low latency, linearizability, infinite capacity, and strong reliability for the state management in RisingWave simultaneously.

The architecture of Hummock is given in Figure 2. Hummock consists of a manager service on the meta service, local Hummock clients on each worker node (including compute node, frontend node, and compactor node), and a shared storage to store Sorted String Tables (SSTs). The shared storage relies on cloud storage service, *e.g.,* AWS S3.

Hummock is optimized for the streaming workload. For write request, Hummock optimizes its write path in a similar fashion with LSM-tree. Hummock client encodes the write batch from shared buffer into SSTs and upload them into the shared storage, and invokes a compaction job if necessary. The compaction service is also decoupled from the Hummock client on the compute node to achieve flexible elasticity. For read requests, Hummock extensively uses tiered cache to bridge the gap between the latency of memory and S3. Since stream executors only access their local state, Hummock can avoid the expensive coordination between different clients while guaranteeing strong consistency.

3.5 Meta service

The meta service is a centralized service that stores metadata, *i.e.,* the **true state** of the cluster, *e.g.,* the system catalog, cluster membership, the status of committed epoch etc. Both frontend nodes and compute nodes periodically synchronize with the meta service to access up-to-date metadata. This enables other services to be elastic as other nodes are essentially stateless. The meta service must survive any failure without compromising consistency, thus we choose etcd [5] for the durable storage of metadata, as etcd provides strong consistency over its key-value storage.

The meta service also serves as the centralized coordinator of the distributed services. For example, the meta service oversees the scheduling of streaming workflow, scheduling compaction jobs, issuing and tracking epoch barriers, and coordinating the failure recovery procedure.

4 DEMONSTRATION

In this demonstration, we invite users to play with RisingWave and show how RisingWave can significantly simplify users' efforts when developing their real-time analytic applications.

During our demonstration, we will invite users to access RisingWave via local psql clients and issue SQL statements, including declaring sources, creating materialized views, and querying real-time results. A Kafka service will also be deployed as the upstream source. In addition to console clients, RisingWave also offers a dashboard (shown in Figure 3) for users to speculate the streaming workflow for maintaining materialized views, and monitor the running status of the cluster. As a showcase, we will present the following example on RisingWave.

Example 1: Advertisement platforms and advertisers want to measure the impact of the displayed advertisements by calculating the click transformation rate (CTR) of all displayed advertisements in real-time, such that advertisers can change their bids immediately without wasting money.

In the traditional stream processing stack, the advertiser usually set up a streaming platform, *e.g.,* Flink, to transform the streaming data from its raw format to a relational model, and dump the data to an OLAP system. Then the advertiser can get the CTR by running adhoc aggregation queries in the OLAP system. This process involves

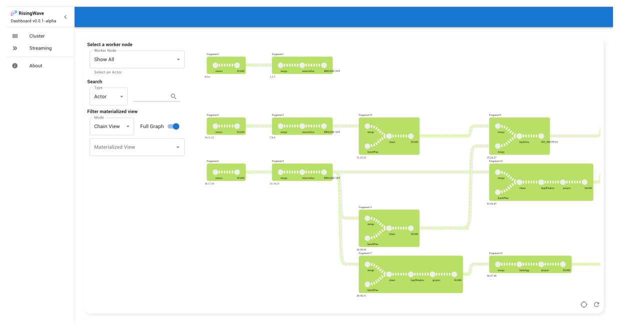

Figure 3: The dashboard of RisingWave showing the streaming workflow

two systems and two sets of API.

With RisingWave, the analytical tasks can be easily done by a few lines of SQL statements. Firstly, we connect the upstream system with RisingWave by creating source statements. Assume that there are two streams ad_impression and ad_click which can be consumed from Kafka topics ad_impression and ad_click respectively. Then the SQL statement for stream ad_impression is given below.

```
create  source ad_impression (
        bid_id BIGINT,ad_id BIGINT,
        impression_timestamp BIGINT )
with (  'connector' = 'kafka',
        'kafka.topic' = 'ad_impression',
        'kafka.brokers' = 'redpanda : 9092',
        'kafka.scan.startup.mode' = 'earliest'
) ROW FORMAT JSON;
```

In the SQL statement above, the configurations of Kafka are specified in the with clause, and the mapping from JSON fields to the relational schema is defined in the source header. In the above example, source ad_impression will produce a stream of 3-tuples continuously, where in each tuple the first column is extracted from bid_id field and cast into int64 type. Similarly, one can also define the source ad_click, for which we omit the SQL clause here due to the space limitation.

After creating sources, we can directly define materialized views on top of created sources. An example materialized view definition is given below.

```
create  materialized view ad_ctr as(
select  ad_clicks.ad_id as ad_id,
        ad_clicks.clicks_count :: NUMERIC/
        ad_impressions.impressions_count as ctr
from (
        select  ad_impression.ad_id as ad_id,
                count(*) as impressions_count
        from ad_impression
        group by ad_id
) as ad_impressions
join (
        select  ai.ad_id as ad_id,
                count(*) as clicks_count
        from ad_click as ac
        left join ad_impression as ai
                on ac.bid_id = ai.bid_id
        group by ai_id
) as ad_clicks
on ad_impressions.ad_id = ad_clicks.ad_id;
```

The materialized view above calculates the CTR of every appeared advertisement based on the latest impressions and clicks. Given the view above, users also search the CTR of any advertisement by directly querying ad_ctr in RisingWave, without any external database.

□

5 FUTURE WORK

There is much to be done before delivering a production-ready streaming database. We would like to enhance RisingWave in the following aspects. (1) Richer functionality support. Currently, RisingWave supports only a small fraction of PostgreSQL SQL standards. We will be adding more basic functionalities including more SQL commands, full-fledged access control, user-defined functions and much more. (2) Schemaless source. Currently, users still need to define how source data is mapped from nested format to flat tables upon ingestion. To simplify the users' usage, we will support schemaless sources in RisingWave, such that RisingWave can automatically infer the source schema from received raw data, and users can speculate the data schema and define views after the data arrive. (3) Adaptive stream processing. A major technical challenge is to optimize the pipeline for stream processing. Unlike ad-hoc queries for which the database can collect statistics in advance, for stream processing, no knowledge about the underlying stream is known when creating sources.The technical challenge here is how to automatically adapt the streaming pipeline on-the-fly.

ACKNOWLEDGEMENTS

We would like to thank Yu Fu, Xiangyu Hu, Zhanxiang Huang, Renjie Liu, Dr. Yingjun Wu, Chi Zhang, Tianyi Zhuang and many others for their help and valuable discussions, and thank Heng Ma, Zhenzhou Pang, Weihao Wang and Tao Wu for their help in preparing the demonstration.

REFERENCES

[1] 2022. Apache Kafka. *https://kafka.apache.org/.*
[2] 2022. Apache Pulsar. *https://pulsar.apache.org/.*
[3] 2022. AWS Kinesis. *https://aws.amazon.com/kinesis/.*
[4] 2022. Debezium. *https://debezium.io/.*
[5] 2022. etcd. *https://etcd.io/.*
[6] 2022. Redpanda. *https://redpanda.com/.*
[7] 2022. RisingWave source code. *https://github.com/singularity-data/risingwave.*
[8] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. 2015. Lightweight asynchronous snapshots for distributed dataflows. *arXiv preprint arXiv:1506.08603* (2015).
[9] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).
[10] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence.* 235–245.
[11] Frank McSherry. 2020. Eventual Consistency isn't for Streaming. *https://materialize.com/eventual-consistency-isnt-for-streaming/.*
[12] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles.* 423–438.