Real-time Elastic Stream Processing Model Based on K8s

Final Project Reports

***Abstract*—This report is about designing a real-time merchandise volume calculator based on stream processing. We will calculate the gross sales number of each province in China per second and calculate average sales number per order of each province in China per second. We focus on elastic property of the stream processing and use Kubernetes to provide a container-centric management environment.**

***Keywords*—stream processing; elastic; real-time; K8s**

I. Introduction

Stream processing is a technology that performs data processing in motion. Stream processing is computing data directly as it is produced or received, which can be applied to market feed processing and electrical trading in Wall Street, network and infrastructure monitoring, fraud detection and command control in military environment, etc.

Real-time guarantees response within specific time constrains. Elastic means dynamically adapt based on different cost-benefit trade-offs, for example, scale in or out due to total utilization ratio.

Kubernetes is a portable, extensible open-source platform for managing containerized workloads and services. It provides a container-centric management environment.

Stream processing can be applied to many real-world applications. And Alibaba's Singles Day real-time big screen is the most famous example. Singles Day is China's largest online shopping holiday, comparable to Black Friday and Cyber Monday in the United States but on an even larger scale. Unlike normal days, User shopping behavior changes considerably on Singles' Day. The Singles Day real-time big screen records the real-time gross merchandise volume. Real-time analysis of a tremendous volume of continuous data inflow is often needed. The value of data could be vanish immediately, or decline as time goes by. So the response time required should be as short as possible. To address these challenges, Alibaba has developed its own real-time computing framework, *Blink*. Blink bases on the true streaming approach, which considers a stream to be fundamental and treat batch as a special case of streaming (a finite stream). Stream is used instead of batch because of the difference between stream and batch. The data handled by stream computing is indefinite, while batch computing is limited to a certain fixed data. Using batch would require calculating the total sum after all transactions had closed. Stream would trace transaction values in real-time and conduct instantaneous computations, continuously updating the latest results. Stream requires checkpoint assessment and a retainment of status, and can continue to run at high speeds during failover. Batch input is sustained for a period and stored, so there is no cause for retaining status. Stream is a pre-estimation of a final result requiring a retraction of computations done in advance to make further amendments. Batch does not require the retraction process. The Singles Day real-time big screen has the following functions like calculating the total GMV in USD, calculating GMV from Mobile, calculating cross-border commerce and ranking countries or regions with completed transactions.

Inspired by the Singles Day big screen design, we will design a real-time merchandise volume calculator, which calculates the gross sales number of each province in China per second and calculate average sales number per order of each province in China per second. Elastic property will be used in the stream processing and container-centric management environment will be used based on K8s.

II. Background

A. *Stream Processing*

Stream processing is a computer programming paradigm, equivalent to dataflow programming, event stream processing, event stream processing, and reactive programming. Multiple computation units do not need to explicitly manage allocation, synchronization or communication.

Stream processing is the processing of data in motion, or in other words, computing on data directly as it is produced or received. The majority of data are born as continuous streams: sensor events, user activity on a website, financial trades, and so on- all these data are created as a series of events over time.

Before stream processing, this data was often stored in a database, a file system, or other forms of mass storage. Applications would query the data or compute over the data as needed.

Figure 1 shows a data-at-rest infrastructure before stream processing.

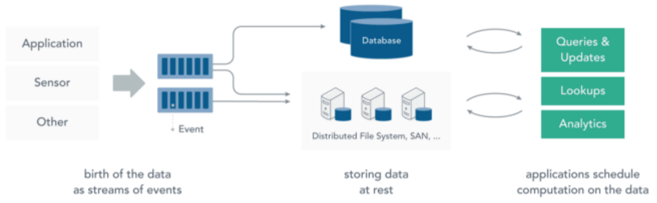

Application
Sensor
Other
Event
Database
Distributed File System, SAN, ...
Queries & Updates
Lookups
Analytics
birth of the data as streams of events
storing data at rest
applications schedule computation on the data

figure 1

In stream processing, the application logic, analytics, and queries exit continuously, and data flow through them continuously. Upon receiving an event from the stream, a stream processing application reacts to that event: it may trigger an action, update an aggregate or other statistic, or "remember" that event for future reference.

The systems that receive and send data streams and execute the application or analytics logic are called stream processors. The basic responsibilities of a stream processor are to ensure that data flows efficiently and the computation scales and is fault tolerant.

Figure 2 shows a stream processing infrastructure.

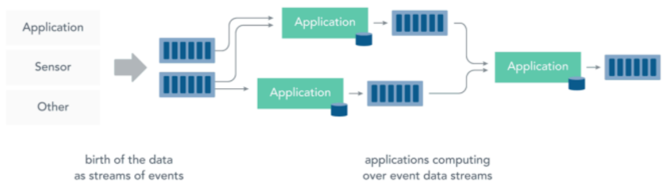

Application
Sensor
Other
Application
Application
Application
birth of the data as streams of events
applications computing over event data streams

figure 2

In stream processing, applications and analytics react to events instantly. Actions and analytics are up-to-date, reflecting the data when it is still fresh, meaningful, and valuable.

Stream processing naturally and easily models the continuous and timely nature of most data. This is in contrast to batch queries and analytics on static data. Incrementally computing updates, rather that periodic recomputation of all data fits naturally with the stream processing model.

Stream processing can handle data volumes that are much larger than other data processing systems. Data stream processing systems (SPSs). Most of the SPSs act on streams that are sequences of tuples sharing a given schema. Each tuple has a set of attributes and timestamp for schema. Each tuple has a set of attributes and timestamp for ordering.

It is desirable to query using a high level language like SQL. The existing stream processing frameworks does not support a common query language. StreamSQL and CQL are candidate query languages which could be standardized.

Streaming applications are programs that process continuous data streams. These applications have become ubiquitous due to increased automation in telecommunications, health care, transportation, retail, science, security, emergency response, and finance. Each of the research communities ultimately represents streaming applications as a graph of streams and operators, where each stream is a conceptually infinite sequence of data items, and each operator consumes data items from incoming streams and produces data items on outgoing streams.

B. Elastic Stream Processing

Stream processing is a computing paradigm that has emerged from the necessity of handling high volumes of data in real time. In contrast to traditional databases, stream processing systems perform continuous queries and handle data on-the-fly. Today, a wide range of application areas relies on efficient pattern detection and queries over streams. The advent of Cloud computing fosters the development of elastic stream processing platforms which are able to dynamically adapt based on different cost-benefit tradeoffs. Cloud computing appears to be the perfect infrastructure for realizing an elastic stream computing service, by dynamically adjusting used resources to the current conditions. In economics, elasticity describes how the change in one variable (e.g., price) influences the change in another variable (e.g., demand). Elasticity in computing defines how computing resources are varied dynamically in order to cope with environmental changes, e.g., different workloads, variations in the price of computing resource, and changing user requirements regarding quality of service.

An event is an object encoding something that is happening for the purpose of computer processing (e.g., stock tick message). Typically, events are of a certain type, have a timestamp, and hold further, more specific data. A complex event results from applying processing steps, like aggregation and filtering, to one or more other events. Events are emitted by (event) sources and consumed by (event) sinks. An (event or data) stream is linear sequence of events, usually ordered by time. Streams are usually considered (potentially) infinite, and a window is some finite portion of a stream.

Simplily speaking, there are two kinds of event type, timestamp and specific data, and complex event (typically result from applying process steps, like filter, aggregation). A linear sequence of events which is ordered by time combines stream.

Window is the finite portion of a stream. There are five different types of event stream query windows:

1) Growing window: binds data values to be available over the entire stream.

2) Single item window: only consider one event at a time and do not need to store the state of previous items.

3) Tumbling window: new windows are only created if there is currently no other open window.

4) Sliding window: a new window is always opened if s(w) holds.

5) Landmark window: once a window has been opened, it remains open indefinitely (until the end of the stream is reached or the query is explicitly closed).

Different types of event stream query windows are illustrated in Figure 3.

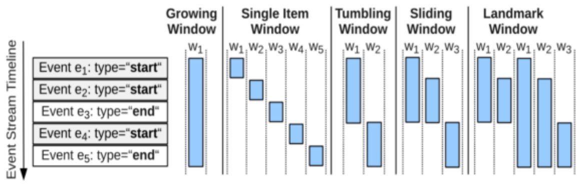


figure 3

The core artifacts and terminology are illustrated in Figure 4.

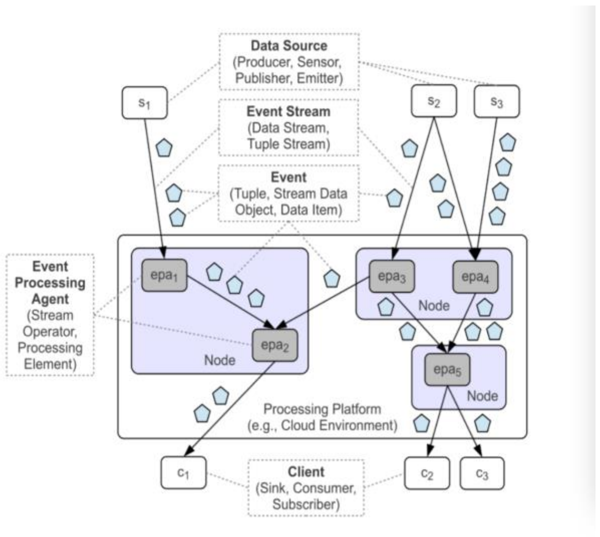

Data Source
(Producer, Sensor, Publisher, Emitter)
Event Stream
(Data Stream, Tuple Stream)
Event
(Tuple, Stream Data Object, Data Item)
Event Processing Agent
(Stream Operator, Processing Element)
s1
s2
s3
epa1
epa2
epa3
epa4
epa5
Node
Node
Node
Processing Platform
(e.g., Cloud Environment)
c1
c2
c3
Client
(Sink, Consumer, Subscriber)

figure 4

Event processing agents (EPAs): software modules that consume events, process them, and output new events. The physical deployment is achieved by mapping EPAs to concrete computing nodes.

Event processing networks (EPNs): a directed graph consisting of EPAs (as vertices) and event channels (as edges), which defines the behaviors of processing.

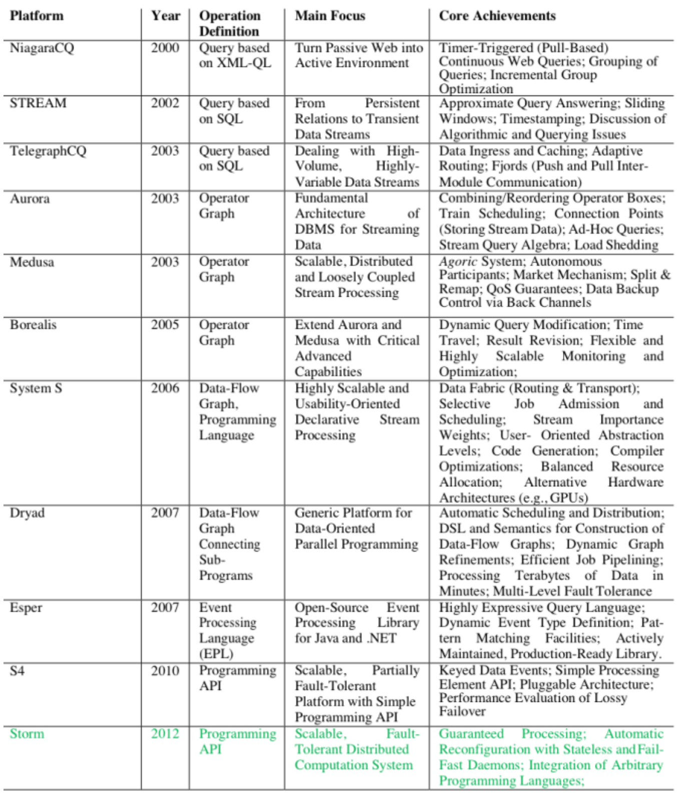
Data stream management system (DSMS): a software system whose main responsibility is execution of one or multiple EPNs.

In term of the elasticity and adaptivity in stream processing, there are three interesting points for stream processing in the Cloud: handling of stream imperfections, guaranteed data safety and delivery, automatic partitioning and scaling of applications. And there are two solutions for the above points: strategies based on single events; and more coarse-grained reorganization of the processing logic.

When it comes to strategies based on single events, the problem is the reliance on and uncontrollability of external which may lead to overload situations or inconsistencies like out-of-order event streams, resulting in the temporary or permanent inability of processing all incoming data. So we have two solutions to solve the problem. The first problem is to reordering and prioritization. We use an ordering function to give precedence to important results, and to reject or defer less important data or queries. We will get the order of results of local transmission by *PRIORITY* and *DELIVER ORDER*. We will get the order of results of the entire results set by *SUMMARIZE AS*. We will manipulate un-interesting data or queries by the prefetch and spool (P&S) technique fetching data from the input and sets aside uninteresting items to an auxiliary side-disk, denoted spooling. If an item on the side-disk becomes interesting (e.g., because ordering functions have changed), the input buffer is enriched with this item for further processing. The second solution is load shedding, which reduces the system load by dropping certain events from the stream. We have three challenges to face: when, where and how much to shed load. It is actually and optimization problem, where reducing the input load means minimizing the loss of accuracy. When the system is overload is the time shed load. We will use load coefficient, computed based on the computational costs of the stream operators and their selectivity to detect system overload. In terms of where and how much to shed load, load shedding roadmap determines the (CPU-) cycle savings coefficient at different points in the network.

When it comes to more coarse-grained reorganization of the processing logic, the most obvious form of elasticity is to scale with the input data rate and the complexity of operations – acquiring new resources when needed, and releasing resources when possible. Most operators in stream computing are stateful and cannot be easily split up or migrated. There are four approaches of topology of operators. The first approach is to model the system load as a time series X and computes the load correlation coefficient ρ_{ij} for pairs of nodes i and j. The optimization goal is to maximize the overall correlation, which has the effect that the load variance of the system is minimized. The second approach is to build a comprehensive and fine-grained model of CPU capacity and system load. The third approach is the stabilization of the buffer occupancy levels. The fourth approach is to make system support intra-query parallelism as well as intra-operator parallelism. We can also know about a distribute algorithm. Nodes are hierarchically clustered (possibly in multiple levels) and one node per cluster is elected coordinator. The placement problem is to split up into partitions for which locally optimal mappings are constructed. Given a dataflow graph tt, the algorithm first exhaustively maps all vertices of tt to nodes within the top-level cluster. Each of the cluster nodes now contains a sub-graph of tt, and the algorithm recursively refines the placement for each cluster node. Costs for migrating to a new configuration. Flux, which aims at achieving a unified utilization among the processing nodes while minimizing the number of state moves. To deal with fault-rolerance, replica based approaches (redundant processing), upstream backup (restoring lost state from predecessor nodes) and checkpointing techniques (periodically forwarding the state from a primary node to a secondary fallback node).

Figure 5 shows the comparison of Stream Processing System.



Platform	Year	Operation Definition	Main Focus	Core Achievements
NiagaraCQ	2000	Query based on XML-QL	Turn Passive Web into Active Environment	Timer-Triggered (Pull-Based) Continuous Web Queries; Grouping of Queries; Incremental Group Optimization
STREAM	2002	Query based on SQL	From Persistent Relations to Transient Data Streams	Approximate Query Answering; Sliding Windows; Timestamping; Discussion of Algorithmic and Querying Issues
TelegraphCQ	2003	Query based on SQL	Dealing with High-Volume, Highly-Variable Data Streams	Data Ingress and Caching; Adaptive Routing; Fjords (Push and Pull Inter-Module Communication)
Aurora	2003	Operator Graph	Fundamental Architecture of DBMS for Streaming Data	Combining/Reordering Operator Boxes; Train Scheduling; Connection Points (Storing Stream Data); Ad-Hoc Queries; Stream Query Algebra; Load Shedding
Medusa	2003	Operator Graph	Scalable, Distributed and Loosely Coupled Stream Processing	*Agoric* System; Autonomous Participants; Market Mechanism; Split & Remap; QoS Guarantees; Data Backup Control via Back Channels
Borealis	2005	Operator Graph	Extend Aurora and Medusa with Critical Advanced Capabilities	Dynamic Query Modification; Time Travel; Result Revision; Flexible and Highly Scalable Monitoring and Optimization;
System S	2006	Data-Flow Graph, Programming Language	Highly Scalable and Usability-Oriented Declarative Stream Processing	Data Fabric (Routing & Transport); Selective Job Admission and Scheduling; Stream Importance Weights; User- Oriented Abstraction Levels; Code Generation; Compiler Optimizations; Balanced Resource Allocation; Alternative Hardware Architectures (e.g., GPUs)
Dryad	2007	Data-Flow Graph Connecting Sub-Programs	Generic Platform for Data-Oriented Parallel Programming	Automatic Scheduling and Distribution; DSL and Semantics for Construction of Data-Flow Graphs; Dynamic Graph Refinements; Efficient Job Pipelining; Processing Terabytes of Data in Minutes; Multi-Level Fault Tolerance
Esper	2007	Event Processing Language (EPL)	Open-Source Event Processing Library for Java and .NET	Highly Expressive Query Language; Dynamic Event Type Definition; Pattern Matching Facilities; Actively Maintained, Production-Ready Library.
S4	2010	Programming API	Scalable, Partially Fault-Tolerant Platform with Simple Programming API	Keyed Data Events; Simple Processing Element API; Pluggable Architecture; Performance Evaluation of Lossy Failover
Storm	2012	Programming API	Scalable, Fault-Tolerant Distributed Computation System	Guaranteed Processing; Automatic Reconfiguration with Stateless and Fail-Fast Daemons; Integration of Arbitrary Programming Languages;

figure 5

C. *Master-Slave Database*

A database is “slaved” to a “master” when it receives a stream of updates from the master in near real-time, functioning as a copy. The “slaves” must simply apply the changes that the master validated and approved.

MySQL replication is a process that enables data from one MySQL database server (the master) to be copied automatically to one or more MySQL database servers (the slaves). It is usually used to spread read access on multiple servers for scalability, although it can also be used for other purposes such as for failover, or analyzing data on the slave in order not to overload the master.

As the master-slave replication is a one-way replication (from master to slave), only the master database is used for the write operations, while read operations may be spread on multiple slave databases. What this means is that if master-slave replication is used as the scale-out solution, you need to have at least two data sources defined, one for write operations and the second for read operations.

The simplest replication can be used to backup valuable data. As changes are stored on the master copy of the database, those same changes are being forwarded to the slave database. For HA environments, RDM will cooperate with an external HA manager to perform failover or failback functionality.

One of the considerations in any distributed database management system is the latency in data consistency between the master and the slave. This consideration is important in assessing the trade-off between performance and the amount of data the application designer is willing to lose in the event of a catastrophic failure on the master database.

Figure 6 shows the Master-Slave Database.

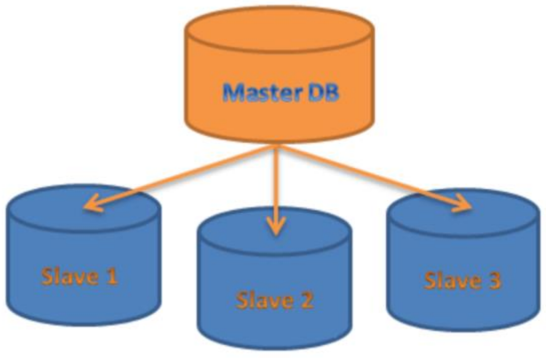


figure 6

Many distributed databases provide synchronous and asynchronous mirroring modes with the slave. In synchronous mode, the slave is part of the update transaction. In asynchronous mode the updates arrive at the slave in a delayed manner. In the latter mode if the master fails before the updates are shipped, the slave will still have the old values. This window of data inconsistency between the master and the slave depends on the periodicity and throughput speed of the update transfers between the master and the slave.

D. *ZooKeeper*

ZooKeeper is a service for coordinating processes of distributed applications. Since ZooKeeper is part of critical infrastructure, ZooKeeper aims to provide a simple and high performance kernel for building more complex coordination primitives at the client. It incorporates elements from group messaging, shared registers, and distributed lock services in a replicated, centralized service. The interface exposed by ZooKeeper has the wait-free aspects of shared registers with an event-driven mechanism similar to cache invalidations of distributed file systems to provide a simple, yet powerful coordination service.

ZooKeeper provides to its clients the abstraction of a set of data nodes (znodes), organized according to a hierarchical name space. The znodes in this hierarchy are data objects that clients manipulate through the ZooKeeper API. Hierarchical name spaces are commonly used in file systems. It is a desirable way of organizing data objects, since users are used to this abstraction and it enables better organization of application meta-data.

There are two types of znodes that a client can create. One kind is regular node, the other is ephemeral node. Clients manipulate regular znodes by creating and deleting them explicitly. Clients create ephemeral znodes, and they either delete them explicitly, or let the system remove them automatically when the session that creates them terminates (deliberately or due to a failure).

ZooKeeper implements watches to allow clients to receive timely notifications of changes without requiring polling. When a client issues a read operation with a watch flag set, the operation completes as normal except that the server promises to notify the client when the information returned has changed. Watches are one-time triggers associated with a session; they are unregistered once triggered or the session closes. Watches indicate that a change has happened, but do not provide the change.

Leader election in ZooKeeper can be done in the following way: The node that gets to create some node first becomes the leader; others give up and issue a watch when such a node disappears. The created znode must be ephemeral so that if the leader were to die, automatically this znode will be deleted and a new election round can be performed. Naturally, the rest of processes must set a watch to be informed of the znode deletion.

Figure 7 illustrates ZooKeeper hierarchical name space.

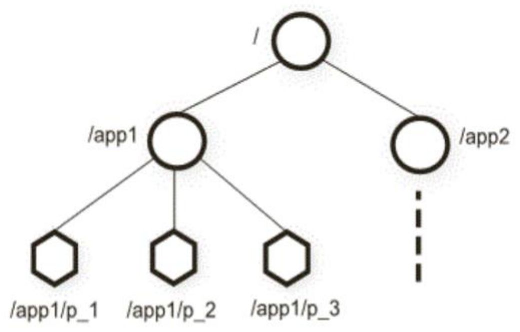


figure 7

We will use python client for ZooKeeper, Kazoo.

E. Kafka

Apache Kafka is a distributed streaming platform. A streaming platform has three key capabilities, publish and subscribe to streams of records, similar to a message queue or enterprise messaging system, store streams of records in a fault-tolerant durable way, process streams of records as they occur.

Kafka is generally used for two broad classes of applications: building real-time streamimg data pipelines that reliably get data between systems or applications, building real-time streaming applications that transform or react to the streams of data.

Kafka is run as a cluster on one or more servers that can span multiple datacenters. The Kafka cluster stores streams of records in categories called topics. Each record consists of a key, a value, and a timestamp.

The partitions of the log are distributed over the servers in the Kafka cluster with each server handing data and requests for a share of the partitions. Each partition is replicated across a configurable number of servers for fault tolerance. Each partition has one server which acts as the “leader” and zero or more servers which act as “followers”. The leader handles all read and write requests for the partition while the followers passively replicate the leader. If the leader fails, one of the followers will automatically become the new leader. Each server acts as a leader for some of its partitions and a follower for others so load is well balanced within the cluster.

In Kafka a stream processor is anything that takes continual streams of sales and shipments, and output a stream of reorders and price adjustments computed off this data. It is possible to do simple processing directly using the producer and consumer APIs. However for more complex transformations Kafka provides a fully integrated Streams API. This allows building applications that do non-trivial processing that compute aggregations off o streams or join streams together. This facility helps solve the hard problems this type of application faces: handling out-of-order data, reprocessing input as code changes, performing stateful computations, etc.

Figure 8 shows Kafka structure.

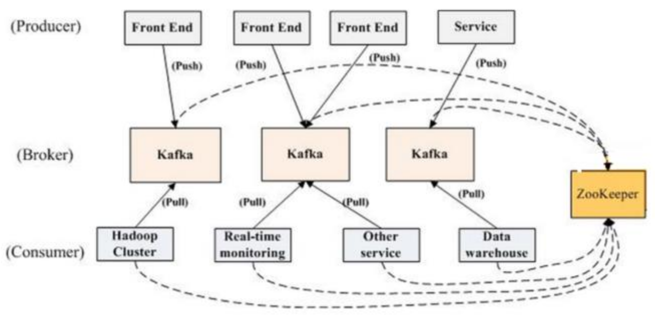


figure 8

F. Etcd

Etcd is a distributed key value store that provides a reliable way to store data across a cluster of machines. It’s open-source and available on GitHub. Etcd gracefully handles leader elections during network partitions and will tolerate machine failure, including the leader. The applications can read and write data into etcd. A simple use-case is to store database connection details or feature flags in etcd as key value pairs. These values can be watched, allowing your app to reconfigure itselt when they change. Advanced uses take advantage of the consistency guarantees to implement database leader elections or do distributed locking across a cluster of workers.

Figure 9 illustrates the application of etcd in Kubernetes.

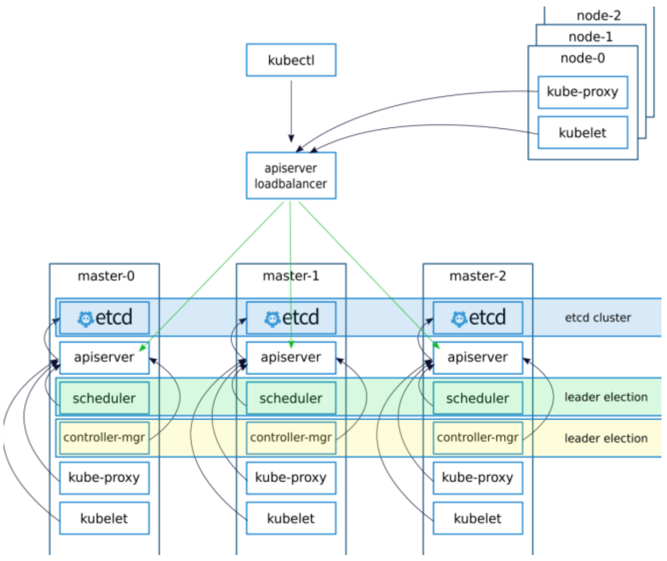


figure 9

Etcd is written in Go which has excellent cross-platform support, small binaries and a great community behind it. Communication between etcd machines is handled via the Raft consensus algorithm.

Latency from the etcd leader is the most important metric to track and the built-in dashboard has a view dedicated to this. In the testing, severe latency will introduce instability within the cluster because Raft is only as fast as the slowest machine in the majority. You can mitigate this issue by properly tuning the cluster. Etcd has been pre-tuned on cloud providers with highly variable networks.

Etcd is a consistent and highly-available key value store used as Kubernetes' backing store for all cluster data. We should always have a backup plan for etcd's data for the Kubernetes cluster. We should run etcd as a cluster of odd members. Etcd is a leader-based distributed system. Ensure that the leader periodically send hearbeats on time to all followers to keep the cluster stable. Keeping stable etcd clusters is critical to the stability of Kubernets clusters.

In a word, etcd is a distributed reliable key-value store for distributed key locking, storing configuration, keeping track of service live-ness and other scenarios.

G. Load-balancing

Load-balancing improves the distribution of workloads across multiple computing resources. Load balancing aims to optimize resource use, maximize throughput, minimize response time and avoid overload of any single resource.

Load balancing is dividing the amount of work that a computer has to do between two or more computers so that more work gets done in the same amout of time and, ingeneral, all users get served faster. Load balancing can be implemented with software. Typically, load balancing is the main reason for computer server clustering.

A load balancer is a device that acts as a reverse proxy and distributes network or application traffic across a number of servers. Load balancers are used to increase capacity (concurrent users) and reliability of applications. They improve the overall performance of applications by decreasing the burden on servers associated with managing and maintaining application and network sessions, as well as by performing application-specific tasks.

HAProxy is a free, very fast and reliable solution offering high availability, load balancing , and proxying for TCP and HTTP-based applications. It is particularly suited for very high traffic web sites and powers quite a number of the world's most visited ones. Over the years it has become the de-factor standard opensource load balancer, is now shipped with most mainstream Linux distributions, and is often deployed by default in cloud platforms.

Figure 10 illustrates the HAProxy well.

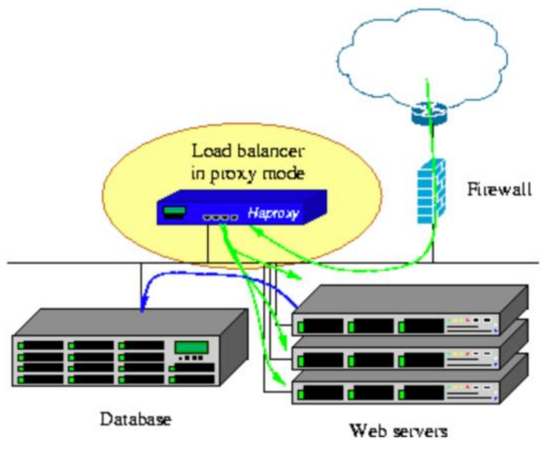


figure 10

H. Sarsa

Sarsa (State-action-reward-state-action) is an algorithm for learning a Markov decision process policy, used in the reinforcement learning area of machine learning. It was proposed by Rummery and Niranjan in a technical note with the name "Modified Connectionist Q-Learning" (MCQ-L).

The following are hyperparameters. Learing rate (alpha) determines to what extent newly acquired information overrides old information. A factor of 0 will make the agent not learn anything, while a factor of 1 would make the agent consider only the most recent information. Discount factor (gamma) determines the importance of future rewards. A factor of 0 makes the agent "opportunistic" by only considering current rewards, while a factor approaching 1 will make it strive for a long-term high reward. If the discount factor meets or exceeds 1, the Q values may diverge. Since SARSA is an iterative algorithm, it implicitly assumes an initial condition before the first update occurs. A low (infinite) initial value, also known as "optimistic initial conditions", can encourage exploration: no matter what action takes place, the update rule causes it to have higher values than the other alternative, thus increasing their choice probability.

I. *Flux*

Flux use process-pair technique to achieve quick fail-over and high availability. Model CQ operators are as deterministic state-machines. In order to maintain the consistency of replicas, we will do the following things. By replicating input stream during normal processing, upon failure, and after recovery. We should concern the following two things, maintaining input ordering and maintaining consistency during failure and after recovery, the former is easy while the latter is hard, because connections can lose in-flight data and operators may not be perfectly synchronized. We want to achieve loss-free, which is no tuple in the input stream sequence are lost and achieve dup-free, which is no tuple in the input stream sequence are duplicated.

In order to achieve invariants, we use intermediate operators that connect existing operators in a replicated dataflow. Between every producer-consumer operator pair, we interpose these intermediate operators to coordinate copies of the produce and consumer. Each tuple has a sequence number. Figure 11 shows flux design and normal case protocol.

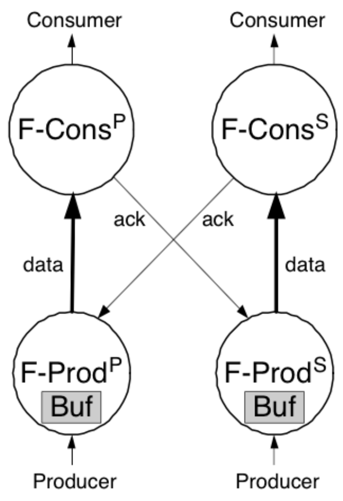


figure 11

The F-Prod is like Ex-Prod in Exchange and F-Cons is like Ex-Cons in Exchange. For each F-Cons instance in the primary dataflow, there is a corresponding F-Cons instance in the secondary dataflow, and likewise for the F-Prod instances. We call each of these combination as a partition pair.

In normal cases as in figure 4, F-Cons behaves same as S-Cons except it maintains two connections for multiple producers. For each tuple received from a connection to a primary F-Prod, it sends an ack of the tuple's sequence number to the corresponding secondary F-Cons, before considering the tuple for any further operations. F-Prod only forward utple to primary F-Cons and only receives ack from secondary F-Cons. For primary F-Prod, once a tuple has been forwarded to primary F-Cons and acked by secondary F-Cons, it is evicted from its internal buffer.

In terms of Flux take-over, if F-ProdP failed, F-ConsP sends "reverse" message to F-ProdS; if F-ConsP failed, F-ProdP sends "reverse" message to F-ConsS.

In terms of Flux Catch-up, before F-Cons starts moving state, it sends a "pause" to all producers, once F-Cons received all "pause" acks, it begins move state. Both F-ConsP and F-ConsS broadcast to all producer partitions a csync message with corresponding checkpoint version number. Like the ingress, if a csync arrives from the active connection, F-Prod resets the buffer and modifies del to account for the new replica. Before csync arrives, the primary forwarding connection is paused for both F-Prod replicas. If the primary connection is connected to the active replica and a csync arrives, the connection immediately is unpaused. If the primary connection goes to a standby replica, the connection is restarted only after the csync from the secondary has arrived. Both F-prod operators broadcast a psync to all remaining F-Cons. F-Prod can have either one or both F-Cons remaining. F-Prod still broadcasts the psync but if it is forwarding data to its secondary, it stops immediately and begins processing acks instead. To avoid missing acks or sending redundant acks, F-Cons cannot activate a connection upon receiving a psync unless the corresponding psync from the replica has arrived.

J. *Kubernetes*

Kubernetes (commonly stylized as K8s) is an open-source system for automating deployment, scaling and management of containerized applications that was originally designed by Google and now maintained by the Cloud Native Computing Foundation. It aims to provide a "platform for automating deployment, scaling, and operations of application containers across clusters of hosts. It works with a range of container tools, including Docker.

Kubernetes follows the master-slave architecture. The components of Kubernetes can be divided into those that manage individual node and those that are part of the control plane.

The Kubernetes Master is the main controlling unit of the cluster that manages its workload and directs communication across the system. The Kubernetes control plane consists of various components, each its own process, that can run both on a single master node or on multiple masters supporting high-availability clusters. The various components of Kubernetes control plane are as follows: etcd, API server, Scheduler, Controller manager. Kubernetes node, also known as Worker or Minion, is a machine where containers (workloads) are deployed. Every node in the cluster must run a container runtime such as Docker, as well as the below-mentioned components, for communication with master for network configuration of these containers. We have terminology like Kubelet, Container, Kube-proxy, cAdvisor.

Figure 12 shows the architecture of Kubernetes.

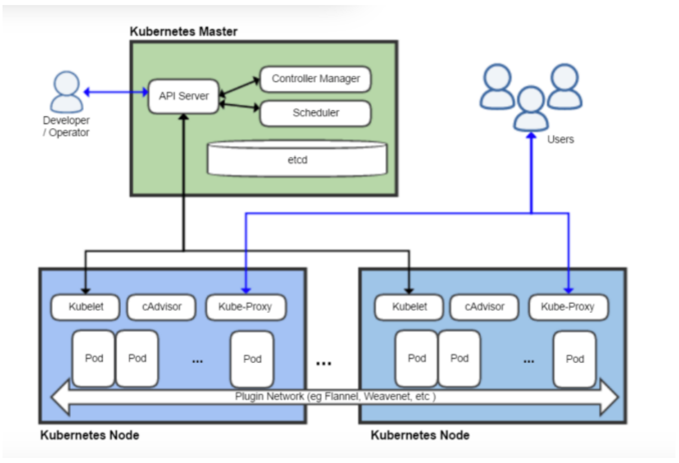


figure 12

K. *Keeperalived*

Keepalived is a routing software. The main goal of this project is to provide simple and robust facilities for loadbalancing and high-availability to Linux system and Linux based infrastructures. Loadbalancing framework relies on well-known and widely used Linux Virtual Server (IPVS) kernel module providing Layer4 loadbalancing. Keepalived implements a set of checkers to dynamically and adaptively maintain and manage loadbalanced server pool according their health. On the other hand high-availability is achieved by VRRP protocol. VRRP is a fundamental brick for router failover. In addition, Keepalived implements a set of hooks to the VRRP finite state machine providing low-level and high-speed protocol interactions. Keepalived frameworks can be used independently or all together to provide resilient infrastructures.

Figure 13 shows the structure of Keeperalived.

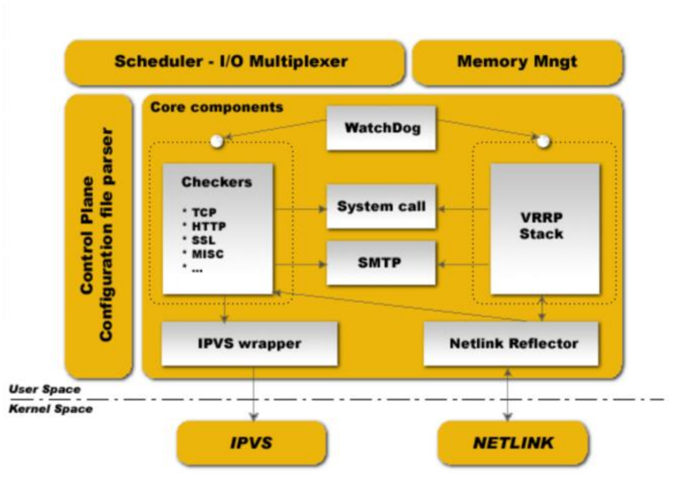


figure 13

L. *MapReduce*

MapReduce is a programming model and an associated implementation for processing and generating large data sets that operates via map and reduce functions.

MapReduce is going to solve the problem about how to perform computations on large input data and distribute them across thousands of machines while hiding the messy details. The goals are to parallelize computation, distribute data, handle failures and replace special purpose computations.

The computation is processed under the following procedure: Map takes in an input key/value pair and produces a set of intermediate key/value pair. Shuffle/Sort groups together all intermediate values associated with the same intermediate key I and passes them to the reduce function. Reduce accepts an intermediate key I and a set of values for that key and merges them together to form a possibly smaller set of values.

Figure 14 illustrates the execution of MapReduce.

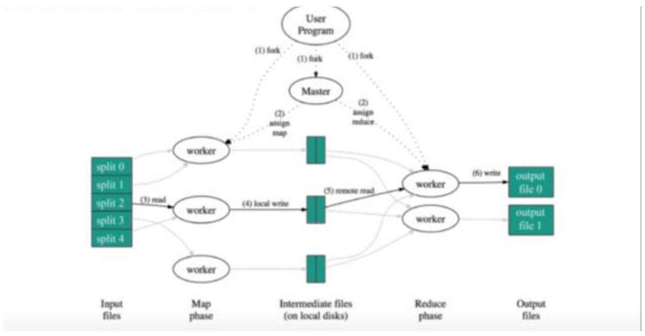


figure 14

III. Design

In our whole final projects, we have the following parts: DataSourcePart, K8sPart, Master-slave database and ZooKeeper part.

In DataSourcePart, we have DataGenerator.py, which is used to generate input data. Generated data are stored under DataSource directory, each file contains 1000000 random numbers. And there are 50 data source files. We also have DataSpoutContainers.py and SpoutingData.py.

In K8sPart, we have MysqlOperations.py to make python implementation to cooperate with database operations. We have Egress.py to implement Egress operator and Ingress.py to implement Ingress operator. Operator.py is used to implement distribute data, calculating, etc. Output.py is used to implement the total sum, total mean, total maximum and total minimum. We also have Master.py to create deployment.

In MySQL part, we have Master and Slave. And we also use ZooKeeper to cooperate operators.

A. Architecture

Figure 15 shows the general system architecture of our design.

A pod represents a running process on the cluster. Pod is a group of one or more containers with shared storage and network, and a specification of how to run the containers. Pods are the smallest deployable units of computing that can be creating and managing in Kubernetes. A pod encapsulates an application container or in some cases multiple containers, storage resources, a unique network IP, and options that govern how the container or containers should run.

A Kubernetes service is an abstraction which defines a logical set of Pods and a policy by which to access them, which sometimes will be called a micro-service. Here, an ingress operator gets input data from source while an egress operator forwards the result to the top-most operator. Tuple is the unit of input data that is composed of timestamp and data. And S_1 to S_n are data source.

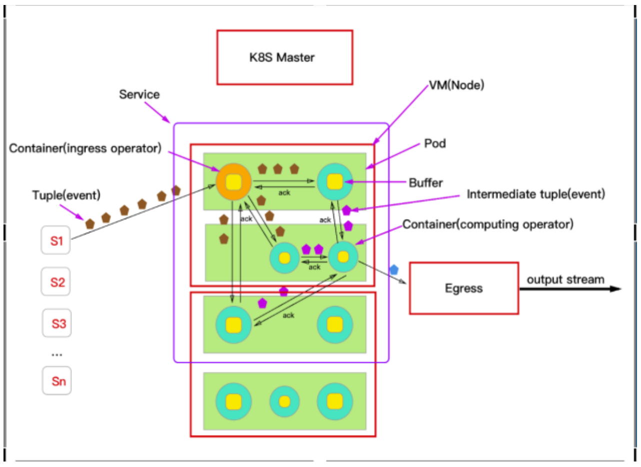

K8S Master
Service
VM(Node)
Container(ingress operator)
Pod
Buffer
Tuple(event)
Intermediate tuple(event)
Container(computing operator)
ack
S1
S2
S3
...
Sn
Egress
output stream

Figure 15: General System Architecture

B. Work in Normal Case

- Ingress operator gets input data from data source.
- Ingress operator immediately distributes data to multiple computing operators. We use a load balancer (like HAproxy) here. Ingress maintain a buffer, whenever ingress receives tuple, it stores timestamp and tuple in buffer and maintains the data until it receives ack message from downstream computing operators.
- Computing operators start computation work as long as they receive new tuples; computing operators also maintain a buffer for fault-tolerance. After computation, computing operator transfer the intermediate tuple to downstream operator.
- Egress operator receives the result and transfers it to output.

C. Auto-scaling

We will use reinforcement learning (Sarsa Algorithm) as our auto-scaling strategy.

Our idea is to make K8s master node maintains a lookup table for each node, and master node needs to measure the utilization of worker node every once in a while. Node is a single virtual or physical machine in a Kubernetes cluster. And a cluster is a group of nodes firewalled from the Internet, which are the primary compute resources. The lookup table is used for deciding which action to take based on historical feedback. After master node making decision, it chooses an action to modify the cluster scale.

For scaling in, master can just build new container to increase system throughput. For scaling out, master node would check if any other node has enough capacity of handle the container that has lowest resource utilization. If so, our idea is to use container live migration to migrate the container to new node. If not, master gives up scaling out.

System always chooses the action that has the highest value. Sarsa updates the reward for the previous state and action Q(si-1) based on an immediate reward r and the expected future reward for the current state Q(si):

$$Q(s_{i-1}) = \left(1 - \alpha(s_{i-1})\right)Q(s_{i-1}) + \alpha(s_{i-1})(r + \beta * Q(s_i)) \qquad (1)$$

$\alpha(s_{i-1})$ and β are learning factors describing the influence of the immediate and the future reward respectively. And r is the immediate reward. r = 1 + |(last measured utilization - target utilization) / target utilization| - |(current measured utilization - target utilization)/target utilization|.

D. Special Conditions that May Happen

- Data may be out of order due to delay, we handle this problem by using a time window and a delay pardon mechanism.
- We use Replication Controller in K8s to implement fault-tolerance in our system to achieve high availability feature.
- We use flux as the fault-tolerance

IV. Results

After having different attempts, we have our final implementation. We have our whole structure implementation and the detailed Micro batch map reduce implementation as the following.

All buffer data are stored in MySQL. We use Master-Slave module to achieve HA. We also use ZooKeeper to cooperate operators.

A. Whole Structure

The figure 16 shows our final implementation about the whole structure.

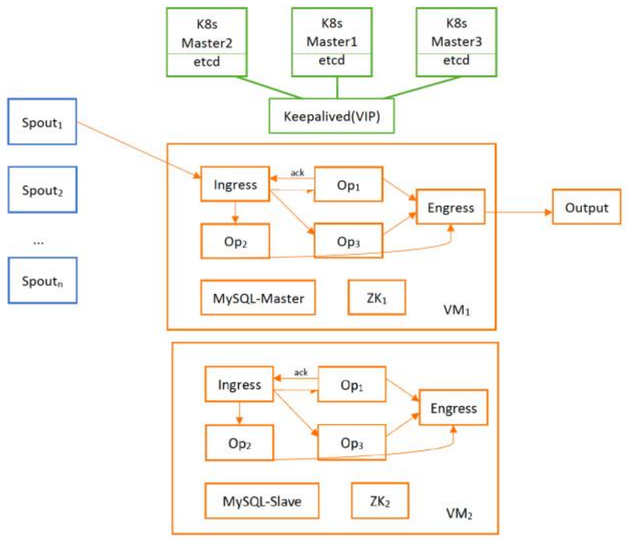


figure 16

B. *Detailed Micro Batch Map Reduce Process*

The figure 17 shows our detailed micro batch map reduce process.

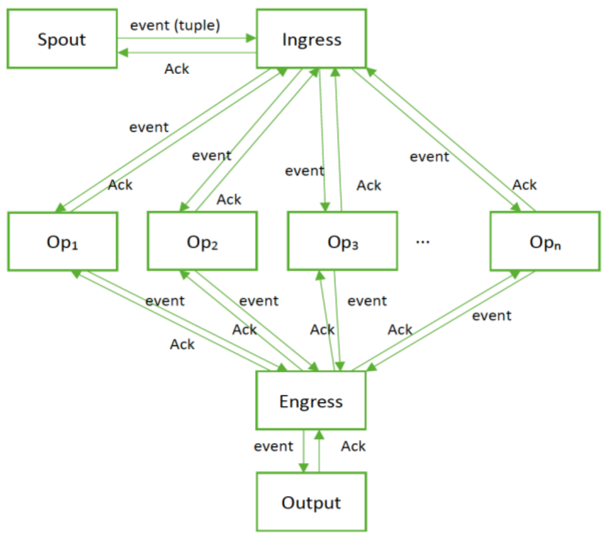


figure 17

C. *Open-source Challenges*

Until now, we only implemented our design successfully on the local machine, which means we still have a long way to go to finish our implementation. We will complete our whole design later in the future.

References

[1] Hummer, W., Satzger, B., & Dustdar, S. (2013). Elastic stream processing in the Cloud. *Wiley Interdisc. Rew.: Data Mining and Knowledge Discovery*, *3*, 333-345.

[2] Heinze, T., Pappalardo, V., Jerzak, Z., & Fetzer, C. (2014). Auto-scaling techniques for elastic data stream processing. *2014 IEEE 30th International Conference on Data Engineering Workshops*, 296-302.

[3] Shah, M.A., Hellerstein, J.M., & Brewer, E.A. (2004). Highly-Available, Fault-Tolerant, Parallel Dataflows. *SIGMOD Conference*.

[4] Hirzel, M., Soulé, R., Schneider, S., Gedik, B., & Grimm, R. (2013). A catalog of stream processing optimizations. *ACM Comput. Surv.*, *46*, 46:1-46:34.

[5] Whittier, J. C., Nittel, S., Plummer, M. A., & Liang, Q. (2013, November). Towards window stream queries over continuous phenomena. In *Proceedings of the 4th ACM SIGSPATIAL International Workshop on GeoStreaming* (pp. 2-11). ACM.

[6] Loesing, S., Hentschel, M., Kraska, T., & Kossmann, D. (2012). Stormy: an elastic and highly available streaming service in the cloud. *EDBT/ICDT Workshops*.

[7] Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Xing, Y., & Zdonik, S. B. (2003, January). Scalable Distributed Stream Processing. In *CIDR* (Vol. 3, pp. 257-268).

[8] Lin, W., Fan, H., Qian, Z., Xu, J., Yang, S., Zhou, J., & Zhou, L. (2016, March). StreamScope: Continuous Reliable Distributed Processing of Big Data Streams. In *NSDI* (Vol. 16, pp. 439-453).

[9] Kumbhare, A. G., Simmhan, Y., & Prasanna, V. K. (2014, May). Plasticc: Predictive look-ahead scheduling for continuous dataflows on clouds. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on* (pp. 344-353). IEEE.

[10] Stonebraker, M., Çetintemel, U., & Zdonik, S.B. (2005). The 8 requirements of real-time stream processing. *SIGMOD Record*, *34*, 42-47.

[11] Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., & Stoica, I. (2013, November). Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*(pp. 423-438). ACM.

[12] Akidau, T., Balikov, A., Bekiroğlu, K., Chernyak, S., Haberman, J., Lax, R., ... & Whittle, S. (2013). MillWheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, *6*(11), 1033-1044.

[13] Qian, Z., He, Y., Su, C., Wu, Z., Zhu, H., Zhang, T., ... & Zhang, Z. (2013, April). Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems* (pp. 1-14). ACM.