Arenadata™ Streaming

Версия - v1.3-ENG

Arenadata Streaming Overview

Оглавление

Arenadata Streaming (**ADS**) – **distributed streaming platform**, which includes an integrated set of enterprise-level components based on open source solutions. The platform contains all the necessary components for streaming and processing real-time data, their transformation, interaction and storage, transmission in the semantics of “exactly-once delivery”, security and administration. Also, the platform can act as a corporate data bus and ETL tool.

The idea of a distributed streaming platform is to provide:

- Single access point – use as a corporate data bus for all your applications;
- Easy, safe and reliable way to control data flow – the ability to safely collect large data streams and efficiently manage them in real time;
- Security policies – the ability to create data streams with support for differentiation of access rights to them;
- Fast and continuous development – Develop streaming analytic applications in minutes in real time without writing a line of code.

One of the features of the implementation of the platform is the use of technology, similar to the transaction logs used in database management systems. **ADS** has the following distinctive technical qualities:

- Fault tolerance;
- Scalability;
- Distribution;
- Available equipment;
- Real time;
- Security;
- Integration;
- Simplicity and flexibility/

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.

ADS is generally used for two broad classes of applications:

- Building real-time streaming data pipelines that reliably get data between systems or applications;
- Building real-time streaming applications that transform or react to the streams of dat.

A few concepts of **Arenadata Streaming**:

- 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.

ADS has four core APIs (Pic.1.):

- The **Producer API** allows an application to publish a stream of records to one or more Kafka topics. Examples of use are given in javadocs;
- The **Consumer API** allows an application to subscribe to one or more topics and process the stream of records produced to them. Examples of use are given in javadocs;

- The **Streams API** allows an application to act as a stream processor, consuming an input stream from one or more topics and producing an output stream to one or more output topics, effectively transforming the input streams to output streams. Examples of use are given in javadocs;
- The **Connector API** allows building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems. For example, a connector to a relational database might capture every change to a table.. Examples of use are given in javadocs.

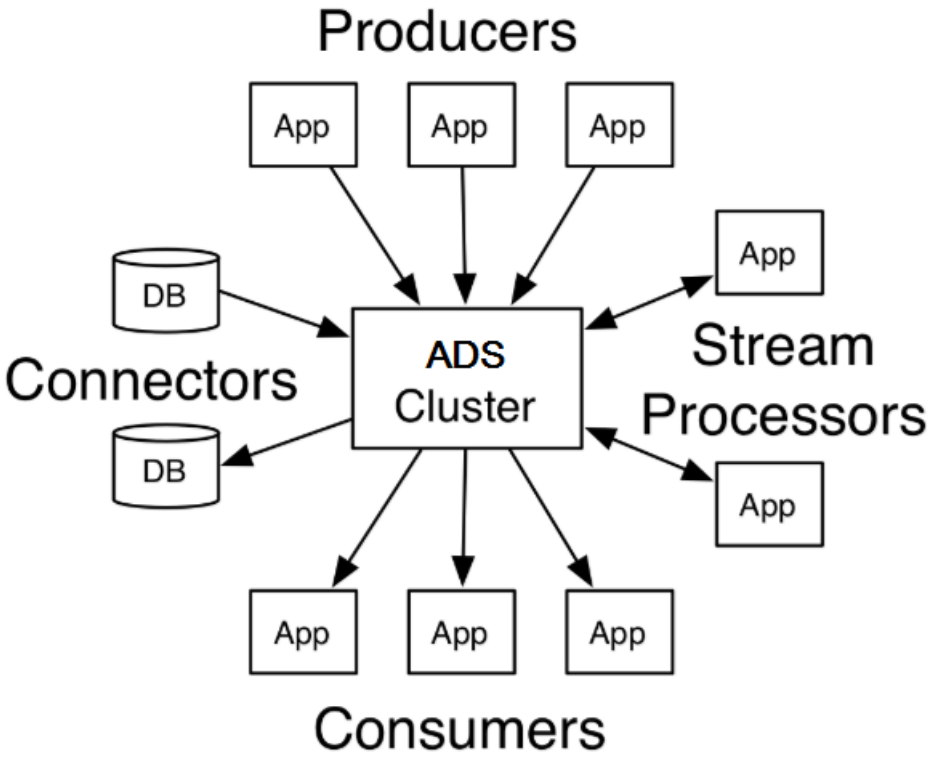

Рис.1.: ADS Platform API

Apache Kafka clients are available in many programming languages. (Clients).

The **NiFi** service as part of the platform **ADS** is a powerful tool for building scalable oriented data routing graphs and their conversion. Some of the high-level features and goals **NiFi**:

- Web user interface: + Development, management and monitoring in a single interface;
- Flexible configuration depending on needs: + Loss resistance or guaranteed delivery; + Low latency or high bandwidth; + Dynamic prioritization; + The ability to change the flow at run time;
- Origin of data: + Tracking data flow from start to finish;
- Expansion of functionality: + The ability to create your own processors and much more; + Ensuring rapid development and effective testing;
- Security: + SSL, SSH, HTTPS, encrypted content, etc.; + Multi-tenant authorization and internal authorization/policy management.

The documentation contains storage concepts for the **Arenadata Streaming** platform, warranties and recommendations for using **ADS**. The section is proposed for reading before proceeding to the direct installation of the system.

Important: Contact information support service – e-mail: info@arenadata.io

Глава 1

Storage concept

Let's first dive into the core abstraction **ADS** provides for a stream of records – the topic.

A topic is a category or feed name to which records are published. Topics in **ADS** are always multi-subscriber; that is, a topic can have zero, one, or many consumers that subscribe to the data written to it. For each topic, the platform maintains a partitioned log that looks like this Pic.1.1.

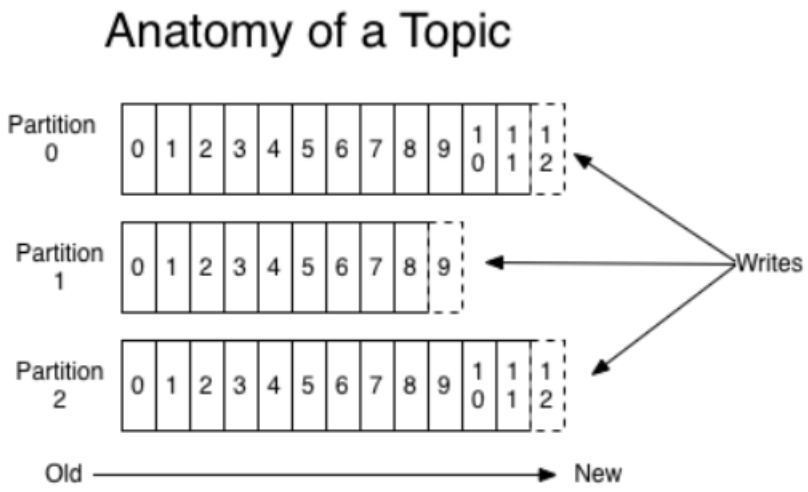

Рис.1.1.: Partitioned log

Each partition is an ordered, immutable sequence of records that is continually appended to – a structured commit log. The records in the partitions are each assigned a sequential *id* number called the *offset* that uniquely identifies each record within the partition.

ADS persists all published records – whether or not they have been consumed – using a configurable retention period. For example, if the retention policy is set to two days, then for the two days after a record is published, it is available for consumption, after which it will be discarded to free up space. Platform's performance is effectively constant with respect to data size so storing data for a long time is not a problem (Pic.1.2.).

In fact, the only metadata retained on a per-consumer basis is the offset or position of that consumer in the log. This offset is controlled by the consumer: normally a consumer will advance its offset linearly as it reads records, but, in fact, since the position is controlled by the consumer it can consume records in any order it likes. For example a consumer can reset to an older offset to reprocess data from the past or skip ahead to the most recent record and start consuming from "now".

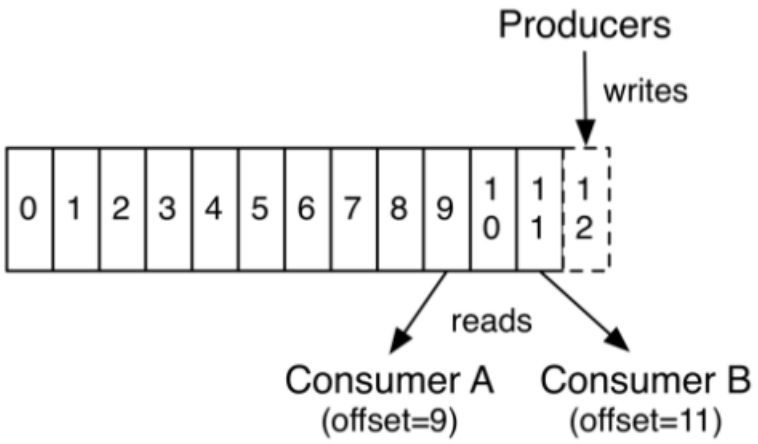

Рис.1.2.: Offset logic

This combination of features means that **ADS** consumers are very cheap – they can come and go without much impact on the cluster or on other consumers. For example, you can use our command line tools to "tail" the contents of any topic without changing what is consumed by any existing consumers.

The partitions in the log serve several purposes. First, they allow the log to scale beyond a size that will fit on a single server. Each individual partition must fit on the servers that host it, but a topic may have many partitions so it can handle an arbitrary amount of data. Second they act as the unit of parallelism—more on that in a bit.

The partitions of the log are distributed over the servers in the **ADS** cluster with each server handling 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.

ADS MirrorMaker provides geo-replication support for your clusters. With **MirrorMaker**, messages are replicated across multiple datacenters or cloud regions. You can use this in active/passive scenarios for backup and recovery; or in active/active scenarios to place data closer to your users, or support data locality requirements.

Producers publish data to the topics of their choice. The producer is responsible for choosing which record to assign to which partition within the topic. This can be done in a round-robin fashion simply to balance load or it can be done according to some semantic partition function (say based on some key in the record).

Consumers label themselves with a consumer group name, and each record published to a topic is delivered to one consumer instance within each subscribing consumer group. Consumer instances can be in separate processes or on separate machines. If all the consumer instances have the same consumer group, then the records will effectively be load balanced over the consumer instances. If all the consumer instances have different consumer groups, then each record will be broadcast to all the consumer processes (Pic.1.3.).

A two server **ADS** cluster hosting four partitions (*P0-P3*) with two consumer groups. Consumer group *A* has two consumer instances and group *B* has four.

More commonly, however, we have found that topics have a small number of consumer groups, one for each "logical subscriber". Each group is composed of many consumer instances for scalability and fault tolerance. This is nothing more than publish-subscribe semantics where the subscriber is a cluster of consumers instead of a single process.

The way consumption is implemented in **ADS** is by dividing up the partitions in the log over the consumer instances so that each instance is the exclusive consumer of a "fair share" of partitions at any point in time. This

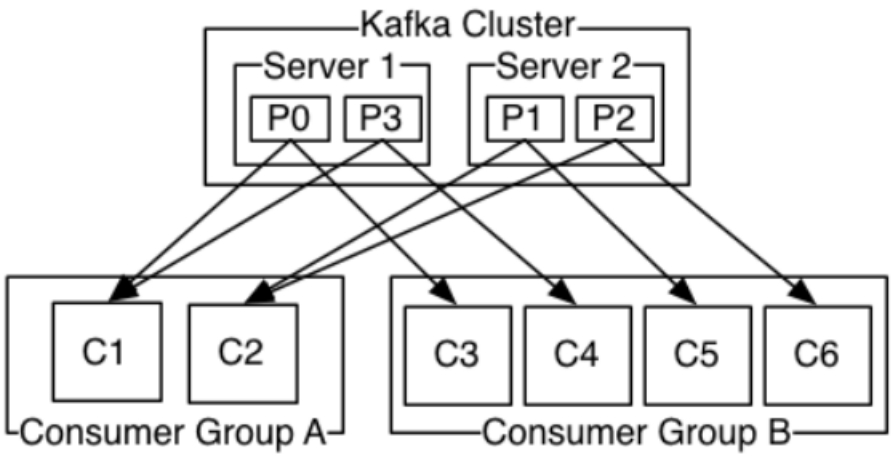

Рис.1.3.: Consumer groups

process of maintaining membership in the group is handled by the **ADS** protocol dynamically. If new instances join the group they will take over some partitions from other members of the group; if an instance dies, its partitions will be distributed to the remaining instances.

ADS only provides a total order over records within a partition, not between different partitions in a topic. Per-partition ordering combined with the ability to partition data by key is sufficient for most applications. However, if you require a total order over records this can be achieved with a topic that has only one partition, though this will mean only one consumer process per consumer group.

You can deploy **ADS** as a multi-tenant solution. Multi-tenancy is enabled by configuring which topics can produce or consume data. There is also operations support for quotas. Administrators can define and enforce quotas on requests to control the broker resources that are used by clients.

Глава 2

Guarantees

At a high-level **ADS** gives the following guarantees:

- Messages sent by a producer to a particular topic partition will be appended in the order they are sent. That is, if a record *M1* is sent by the same producer as a record *M2*, and *M1* is sent first, then *M1* will have a lower offset than *M2* and appear earlier in the log;
- A consumer instance sees records in the order they are stored in the log;
- For a topic with replication factor *N*, we will tolerate up to *N-1* server failures without losing any records committed to the log.

Глава 3

Recommendations for use

3.1 ADS as a Messaging System

How does ADS's notion of streams compare to a traditional enterprise messaging system?

Messaging traditionally has two models: queuing and publish-subscribe. In a queue, a pool of consumers may read from a server and each record goes to one of them; in publish-subscribe the record is broadcast to all consumers.

Each of these two models has a strength and a weakness. The strength of queuing is that it allows you to divide up the processing of data over multiple consumer instances, which lets you scale your processing. Unfortunately, queues aren't multi-subscriber—once one process reads the data it's gone. Publish-subscribe allows you broadcast data to multiple processes, but has no way of scaling processing since every message goes to every subscriber.

The consumer group concept in **ADS** generalizes these two concepts. As with a queue the consumer group allows you to divide up processing over a collection of processes (the members of the consumer group). As with publish-subscribe, **ADS** allows you to broadcast messages to multiple consumer groups.

The advantage of ADS's model is that every topic has both these properties—it can scale processing and is also multi-subscriber—there is no need to choose one or the other.

ADS has stronger ordering guarantees than a traditional messaging system, too.

A traditional queue retains records in-order on the server, and if multiple consumers consume from the queue then the server hands out records in the order they are stored. However, although the server hands out records in order, the records are delivered asynchronously to consumers, so they may arrive out of order on different consumers. This effectively means the ordering of the records is lost in the presence of parallel consumption. Messaging systems often work around this by having a notion of "exclusive consumer" that allows only one process to consume from a queue, but of course this means that there is no parallelism in processing.

ADS does it better. By having a notion of "parallelism – the partition – within the topics", **ADS** is able to provide both ordering guarantees and load balancing over a pool of consumer processes. This is achieved by assigning the partitions in the topic to the consumers in the consumer group so that each partition is consumed by exactly one consumer in the group. By doing this we ensure that the consumer is the only reader of that partition and consumes the data in order. Since there are many partitions this still balances the load over many consumer instances. Note however that there cannot be more consumer instances in a consumer group than partitions.

3.2 ADS as a Storage System

Any message queue that allows publishing messages decoupled from consuming them is effectively acting as a storage system for the in-flight messages. What is different about **ADS** is that it is a very good storage system.

Data written to **ADS** is written to disk and replicated for fault-tolerance. **ADS** allows producers to wait on acknowledgement so that a write isn't considered complete until it is fully replicated and guaranteed to persist even if the server written to fails.

The disk structures **ADS** uses scale well – **ADS** will perform the same whether you have *50 KB* or *50 TB* of persistent data on the server.

As a result of taking storage seriously and allowing the clients to control their read position, you can think of **ADS** as a kind of special purpose distributed filesystem dedicated to high-performance, low-latency commit log storage, replication, and propagation.

3.3 ADS for Stream Processing

It isn't enough to just read, write, and store streams of data, the purpose is to enable real-time processing of streams.

In **ADS** a stream processor is anything that takes continual streams of data from input topics, performs some processing on this input, and produces continual streams of data to output topics.

For example, a retail application might take in input 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 **ADS** provides a fully integrated Streams API. This allows building applications that do non-trivial processing that compute aggregations off of 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.

The streams API builds on the core primitives **ADS** provides: it uses the producer and consumer APIs for input, uses **ADS** for stateful storage, and uses the same group mechanism for fault tolerance among the stream processor instances.

Глава 4

The core concepts of NiFi

NiFi's fundamental design concepts closely relate to the main ideas of Flow Based Programming – FBP. Here are some of the main **NiFi** concepts and how they map to **FBP**.

Таблица4.1.: The core concepts of NiFi

NiFi Term	FBP Term	Description
FlowFile	Information Packet	A FlowFile represents each object moving through the system and for each one, NiFi keeps track of a map of key/value pair attribute strings and its associated content of zero or more bytes
FlowFile Processor	Black Box	Processors actually perform the work. In eip terms a processor is doing some combination of data routing, transformation, or mediation between systems. Processors have access to attributes of a given FlowFile and its content stream. Processors can operate on zero or more FlowFiles in a given unit of work and either commit that work or rollback
Connection	Bounded Buffer	Connections provide the actual linkage between processors. These act as queues and allow various processes to interact at differing rates. These queues can be prioritized dynamically and can have upper bounds on load, which enable *back pressure*
Flow Controller	Scheduler	The Flow Controller maintains the knowledge of how processes connect and manages the threads and allocations thereof which all processes use. The Flow Controller acts as the broker facilitating the exchange of FlowFiles between processors
Process Group	subnet	A Process Group is a specific set of processes and their connections, which can receive data via input ports and send data out via output ports. In this manner, process groups allow creation of entirely new components simply by composition of other components

This design model, also similar to seda, provides many beneficial consequences that help NiFi to be a very effective platform for building powerful and scalable dataflows. A few of these benefits include:

- Lends well to visual creation and management of directed graphs of processors;

- Is inherently asynchronous which allows for very high throughput and natural buffering even as processing and flow rates fluctuate;
- Provides a highly concurrent model without a developer having to worry about the typical complexities of concurrency;
- Promotes the development of cohesive and loosely coupled components which can then be reused in other contexts and promotes testable units;
- The resource constrained connections make critical functions such as *back-pressure* and *pressure release* very natural and intuitive;
- Error handling becomes as natural as the happy-path rather than a coarse grained catch-all;
- The points at which data enters and exits the system as well as how it flows through are well understood and easily tracked.