DataStax

DataStax Monday Learning

- From engineers to engineers
- Open-source only
- Hands-on experience
- Live communication with experts

DataStax

June 14th Led By Aleks and Zeke

Event Streaming Series: Ep. II
Pulsar in Action

Agenda

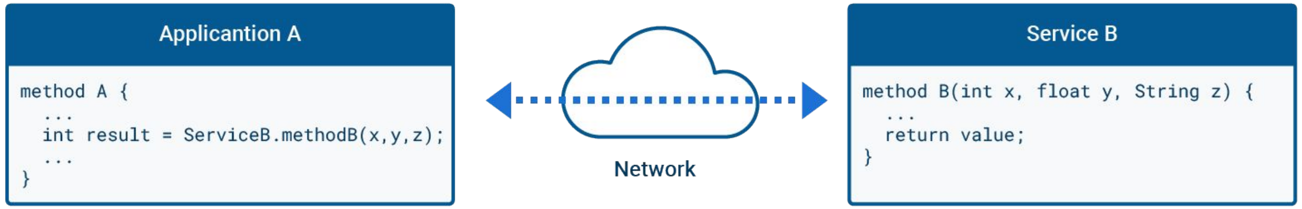
- Why Messaging
- Review Pulsar
- Messaging Patterns
- Schema Management
- SQL with Pulsar

DataStax

Why Messaging?

Remote Procedure Calls – Pain before Messaging Systems

- Remote Procedure Calls
  - Application A makes a request from service B
  - Service B responds

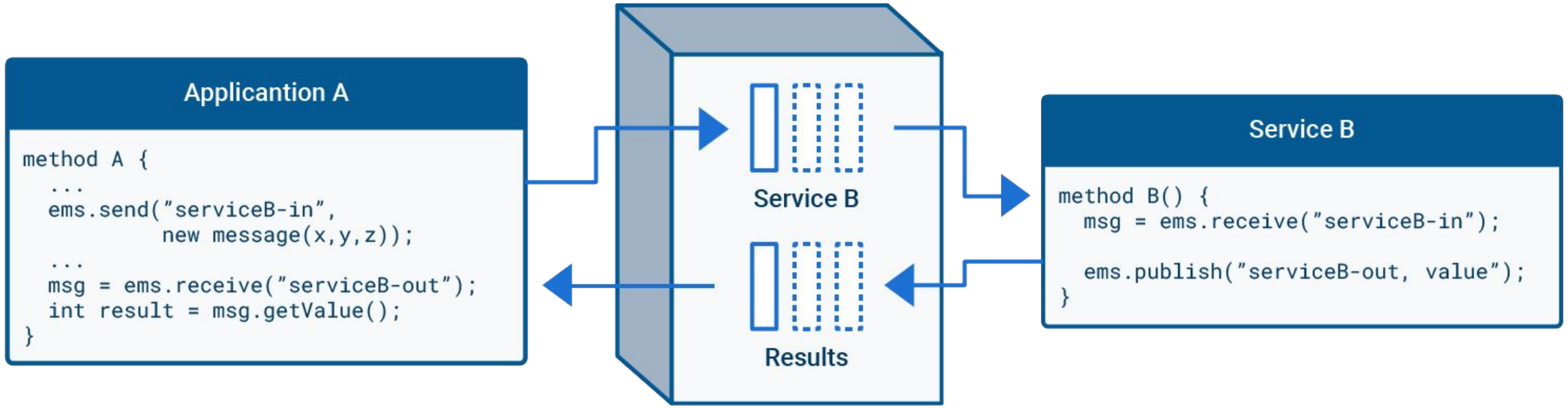


```
method A {
  ...
  int result = ServiceB.methodB(x,y,z);
  ...
}
```

```
method B(int x, float y, String z) {
  ...
  return value;
}
```

- Problems
  - Huge risk with networking
  - Needs service discovery
  - Point to Point communications
  - Slow (blocking) as you are waiting for one server to respond to another
  - If a receiver is not available, process is failed
  - One and Only One receiver

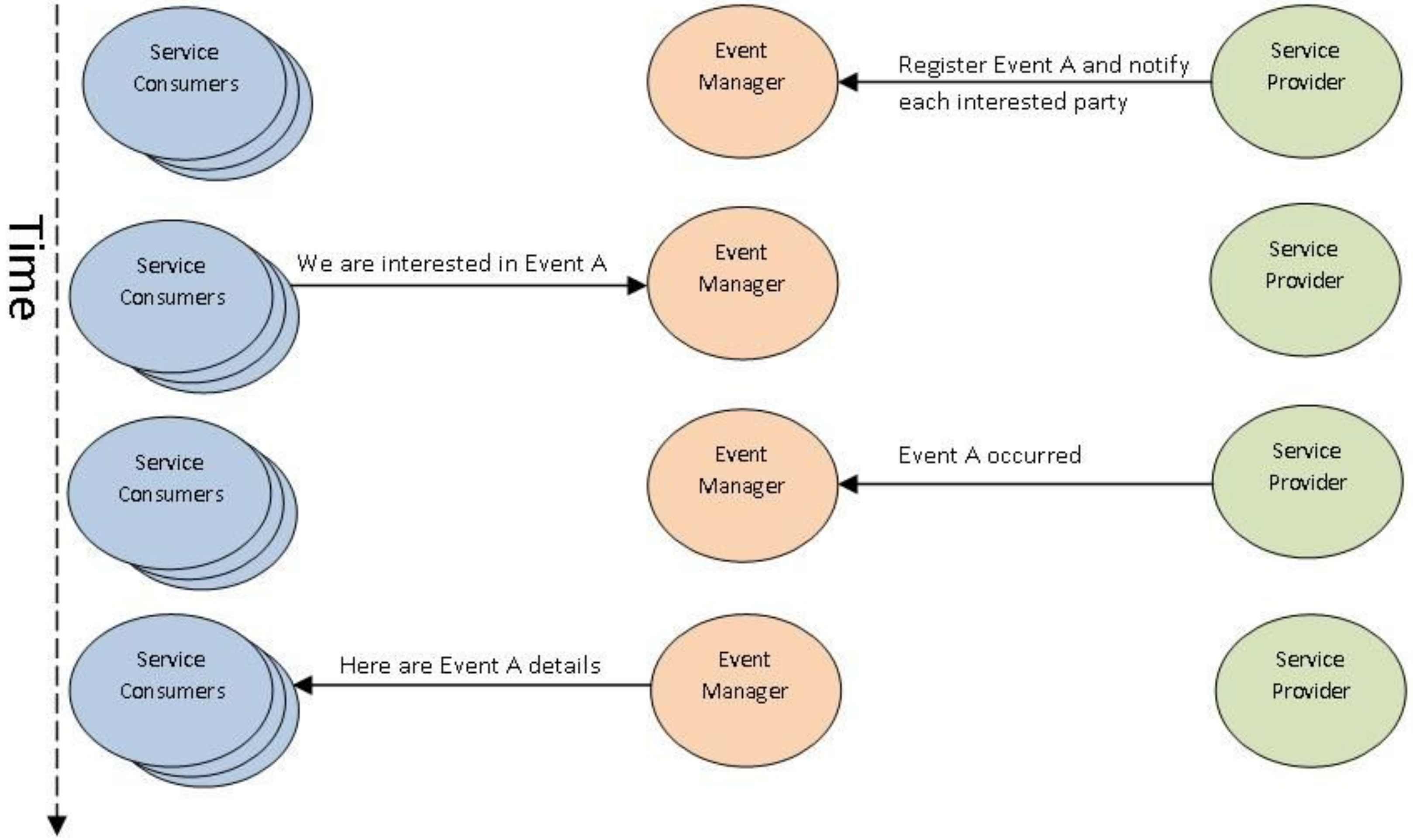
Message Systems solved RPC limits



```
method A {
  ...
  ems.send("serviceB-in",
           new message(x,y,z));
  ...
  msg = ems.receive("serviceB-out");
  int result = msg.getValue();
}
```

```
method B() {
  msg = ems.receive("serviceB-in");

  ems.publish("serviceB-out, value");
}
```

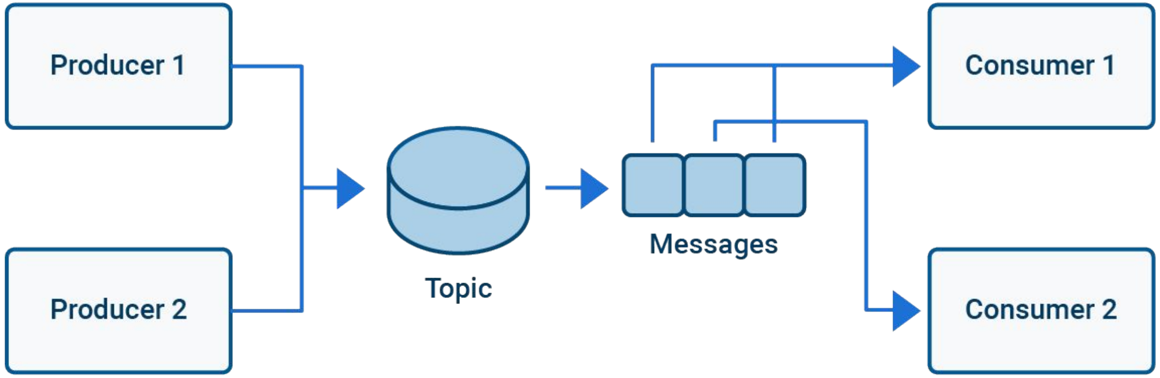
- Decouple message senders from the receivers by using a **messaging service**
- Persistent buffer to handle requests and data
- Async = no need to wait!
- Multiple Receivers!
- Perfect fit for distributed systems / microservices
- Message will be consumed by the next available service instance


Time
Service Consumers
Event Manager
Service Provider
Register Event A and notify each interested party
We are interested in Event A
Event A occurred
Here are Event A details

Queues

- Multiple Producers, One Consumer
- But you can have thousands in parallel
- Consumers share work
- Perfect fit for background processing
- Easy to scale
- Easy to distribute

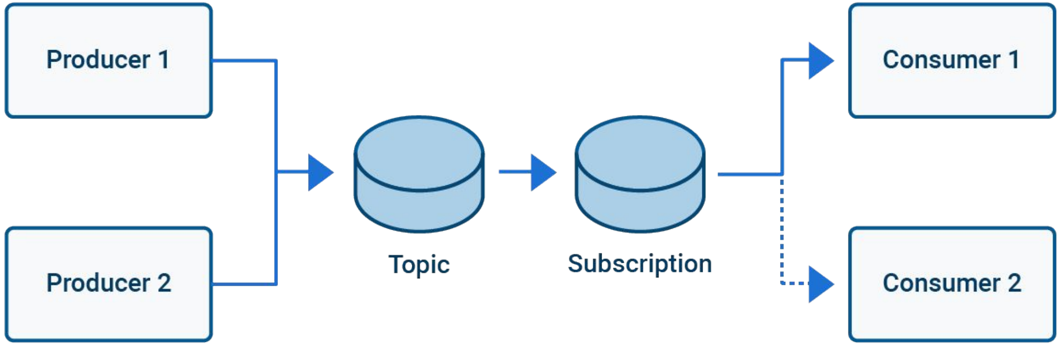
Use-Case: workers



Notifications

- Multiple Producers, Multiple Consumers
- Perfect fit for decentralized processing
- Easy to extend

Use-Case: Microservices



DataStax

Introduction to Apache Pulsar

What is Apache Pulsar

- Distributed pub-sub messaging system
  - High throughput, low latency
  - Separates compute from storage
  - Horizontally scalable
  - Streaming and queuing

- Open source
  - Originally developed at Yahoo!
  - Contributed to the Apache Software Foundation (ASF) in 2016
  - Top-level project (2018)
  - 7.2K GitHub stars, over 300 contributors

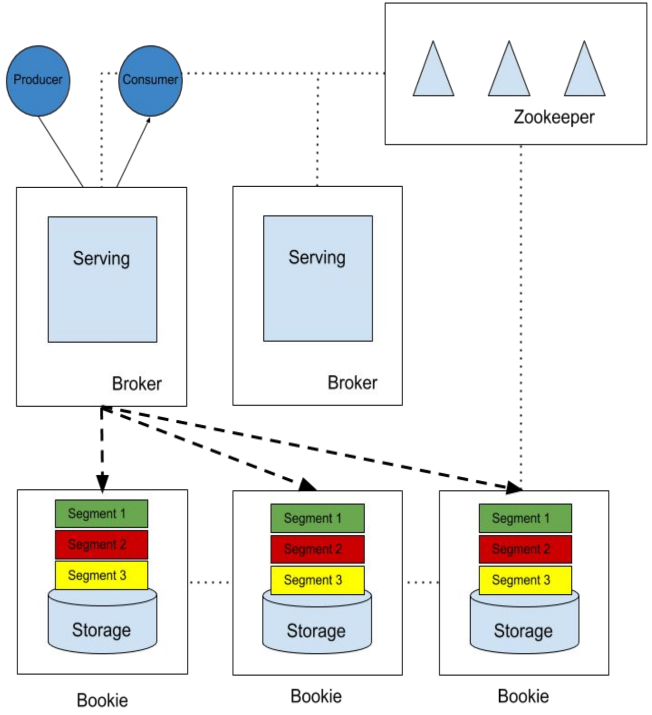
Architecture

- Distributed, tiered architecture
- Separates compute from storage

- ZooKeeper holds metadata for the cluster

- Stateless Broker handles producers and consumers

- Storage is handled by Apache BookKeeper
  - BookKeeper distributed, append-only log
  - Data is broken into segments written to multiple bookies



Why Apache Pulsar

Four Reasons Why Apache Pulsar is Essential to the Modern Data Stack

- Geo-replication
- Scaling
- Multitenancy
- Queuing (as well as Streaming)

- Cloud Native
  - K8s
  - Multi-cloud, hybrid-cloud

- Performance
  - High throughput
  - Consistent low latency

DS

Pulsar Components and Ecosystem

Zookeeper

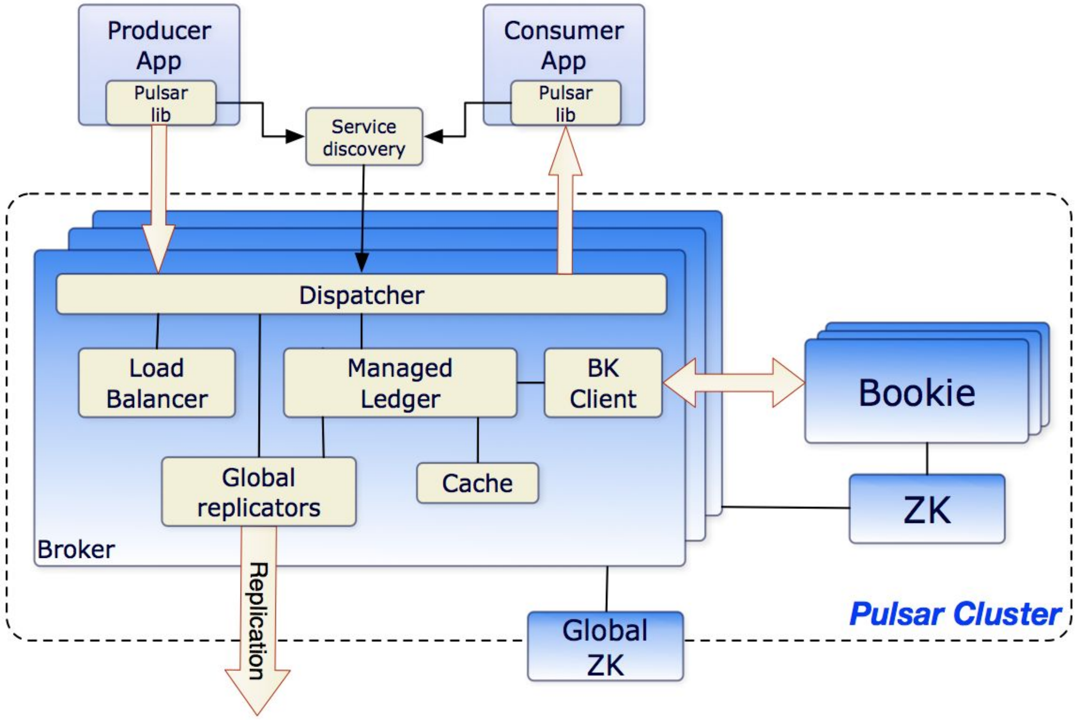
- For metadata storage, cluster configuration, and coordination

- Instance/Global level Zookeeper
  - Geo-replication
  - Configuration for tenants, namespaces, and other entities that need to be globally consistent
  - Optional

- Cluster level Zookeeper
  - Ownership metadata
  - Broker load reports
  - BookKeeper ledger metadata
  - … …

Broker

- Stateless
- Topic ownership
- Load Balancing
- Pulsar's "Brain"
  - HTTP Rest APIs for Admin tasks and topic lookup
  - TCP binary protocol for data transfers



Broker (Topic Ownership), cont'd

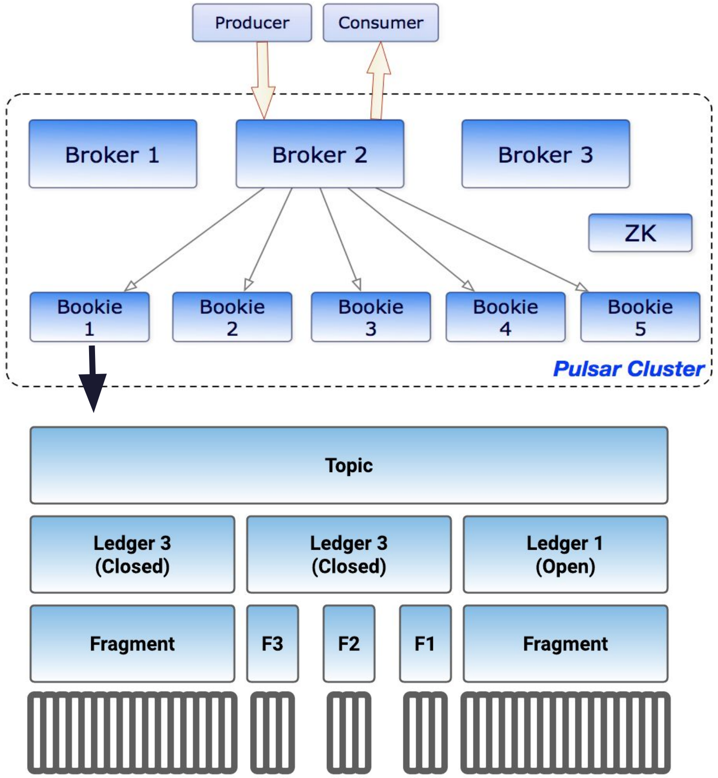
- Stateless broker makes possible of **dynamic assignment**
  - Topic ownership can change on the fly
  - Broker crash
  - Overloaded broker
- Ownership assignment granularity
  - Namespace **bundle** (subset of a namespace)
    - defaultNumberOfNamespaceBundles=4
  - Consistent hashing
    - C* ring

Broker (Load Balancing), cont'd

- Unload topics and bundles
  - Triggers topic re-assignment based on the current workload
  - Automatic or manual
- Bundle Split
  - Tunable thresholds
  - Automatically triggers topics unloading
- Automatic Load Shedding
  - Forces topics unloading from overloaded brokers
  - Enabled by default and can be disabled

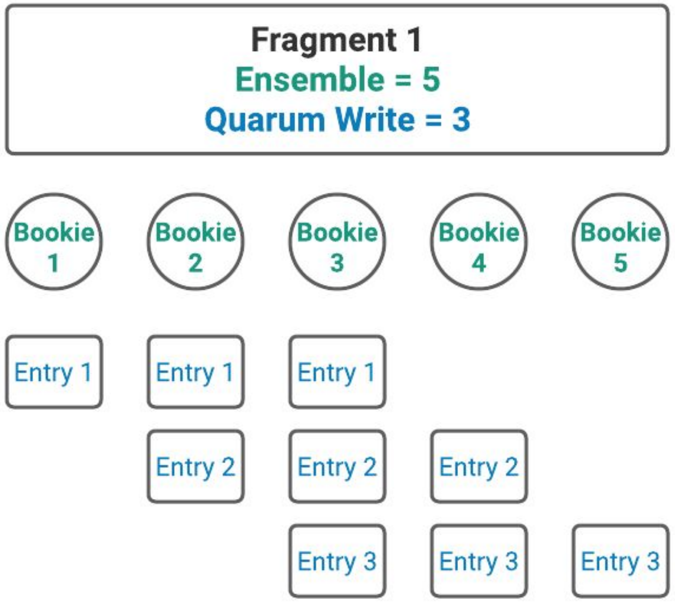
Bookkeeper (Bookie)

- Managed ledger (Topic)
  - Logical abstraction
  - A stream of ledgers
- A Ledger is created when:
  - A new topic is created, or
  - Roll-over occurs (size/time limit, ownership change)
- Append only
  - Read only after closed
- Fragments and Entries

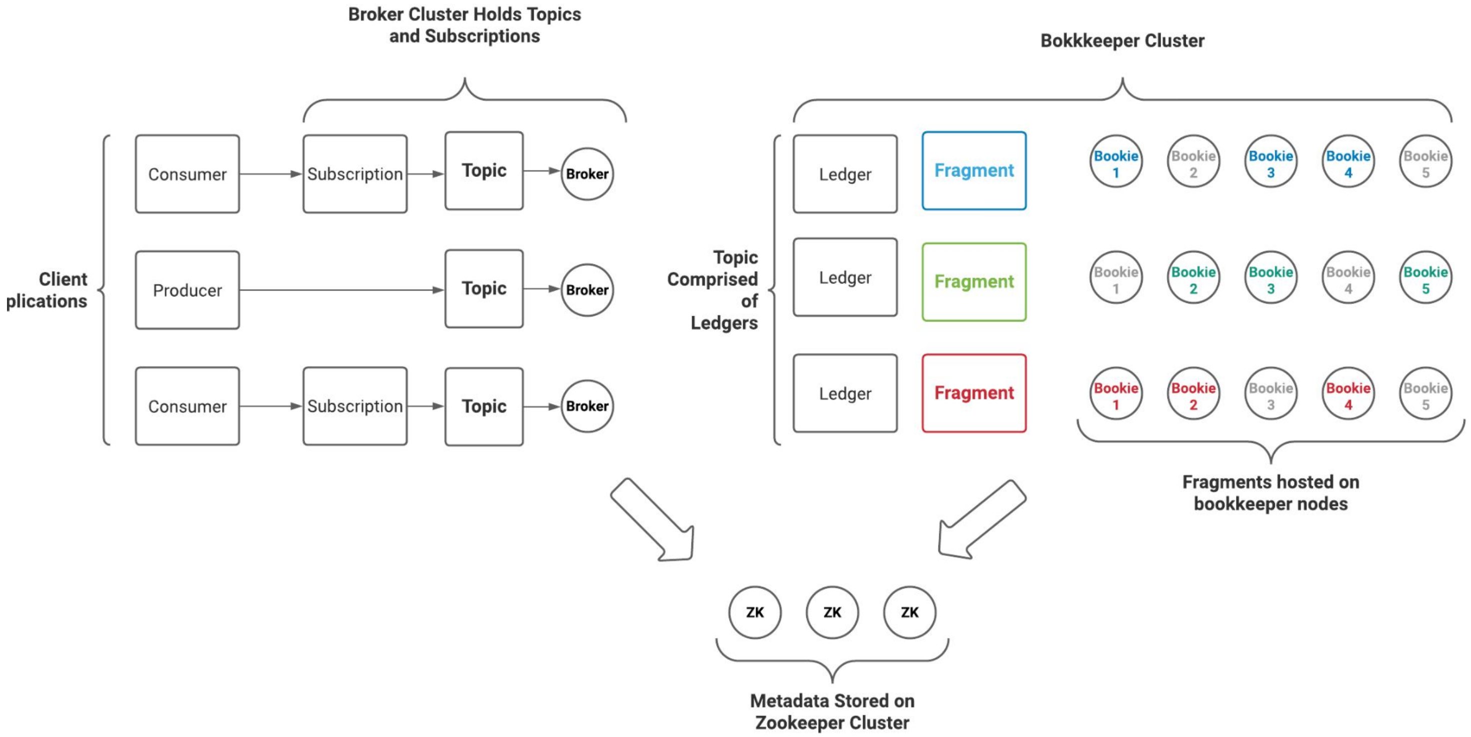


Bookkeeper (Bookie), cont'd

- Key Configuration
  - Ensemble Size (E)
    - The size of the pool of Bookies available for writes
  - Write Quorum Size (Qw)
    - The number of actual Bookies that Pulsar will write an entry to
  - Ack Quorum Size (Qa)
    - The number of Bookies that must acknowledge the write
- Rack-awareness
- WAL Journal
  - Seperate Journal and Ledger disks



Put all Together

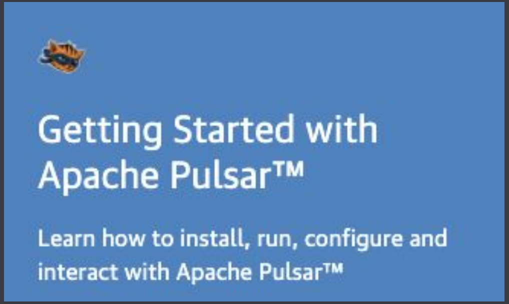


LAB TIME!

🤓 💻 👇

dtsx.io/ess-ep2-lab

Select ONLY this one →



Getting Started with Apache Pulsar™

Learn how to install, run, configure and interact with Apache Pulsar™

Message Processing

Core Concepts and Processing Model

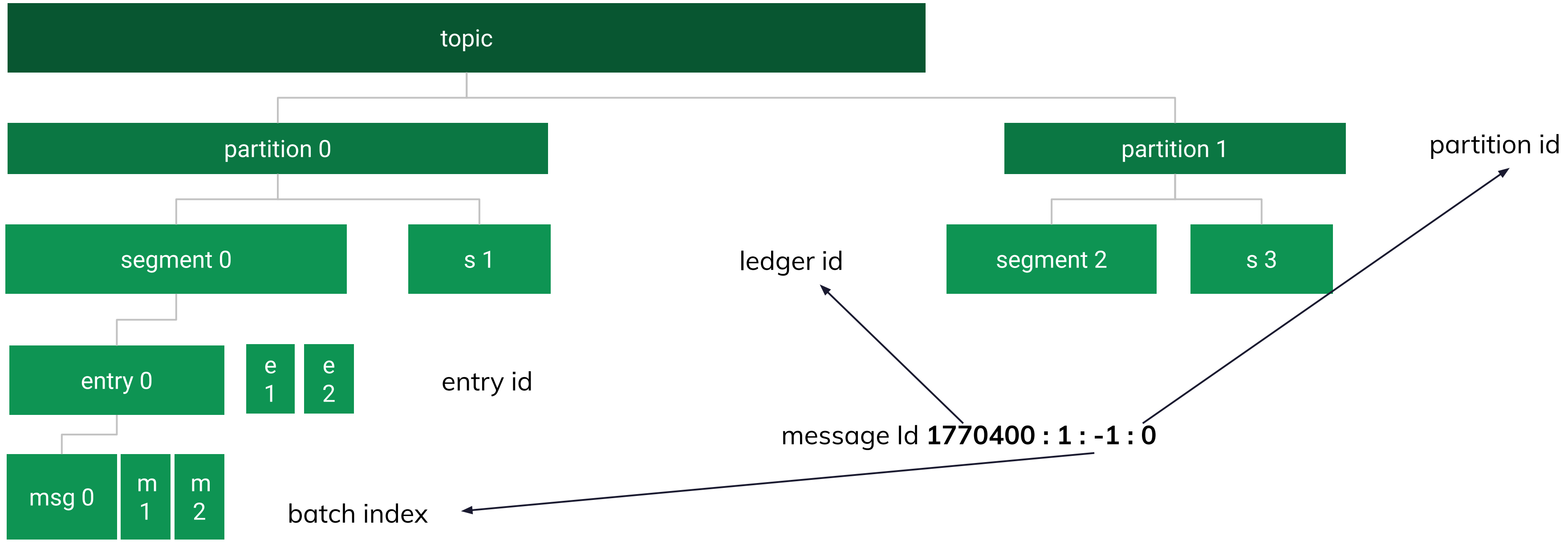
Topic

- Messages (events) are published to and subscribed from **topics**
  - Producer, consumer, reader
  - Subscription
    - Cursor (Offset)
  - Partitioned vs. non-partitioned
    - NO “Kafka Partition” constraints
  - Tenant: security context; resource quota
  - Namespace: administrative unit

persistent://my-tenant/my-namespace/my-topic

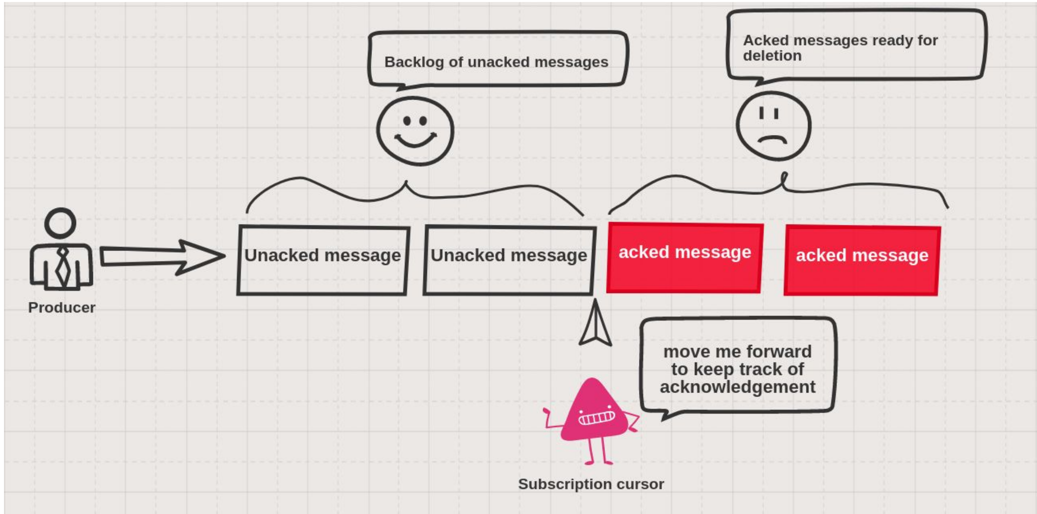
Message

- Topic is a logic channel
- Messages are organized as a linear log
- Each message has a message Id
  - ledger-id, entry-id, batch-index, and partition-id



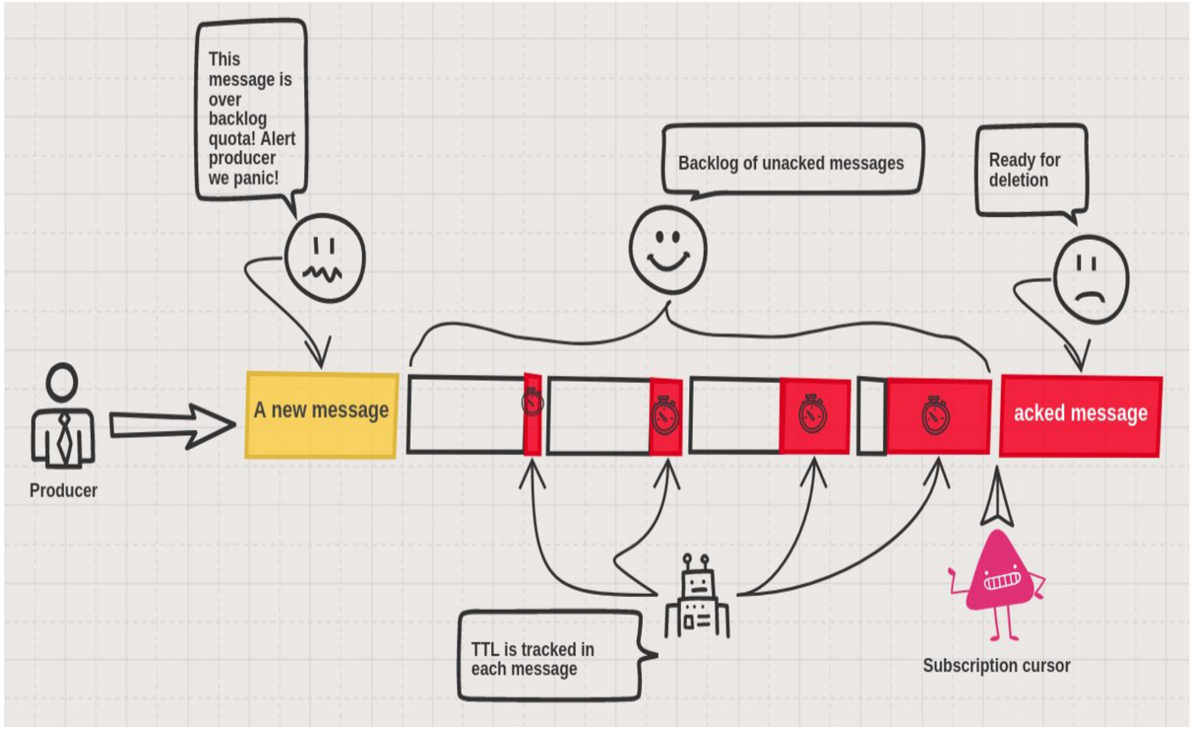
Message TTL and Retention

- Pulsar keeps all messages until they are acknowledged
  - Hence TTL
  - By default TTL is indefinite
    - Hence backlog quota to prevent indefinite growth
- Pulsar does not intend to keep acknowledged messages or messages in a topic with no subscription
  - Hence retention policy


Backlog of unacked messages
Acked messages ready for deletion
Producer
Unacked message
Unacked message
acked message
acked message
move me forward to keep track of acknowledgement
Subscription cursor

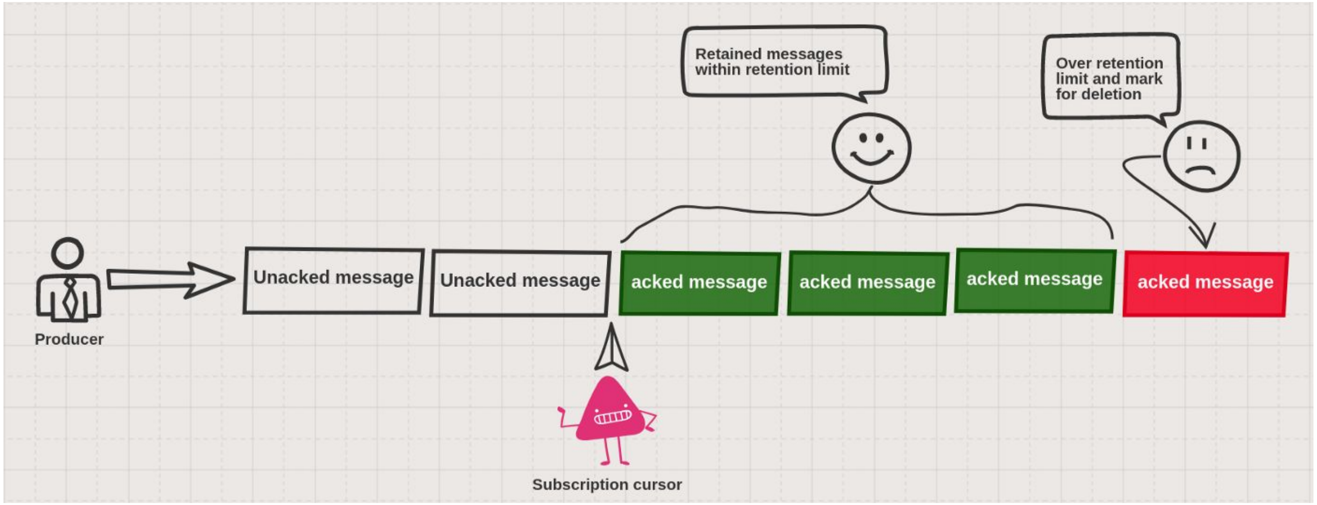
Backlog and TTL

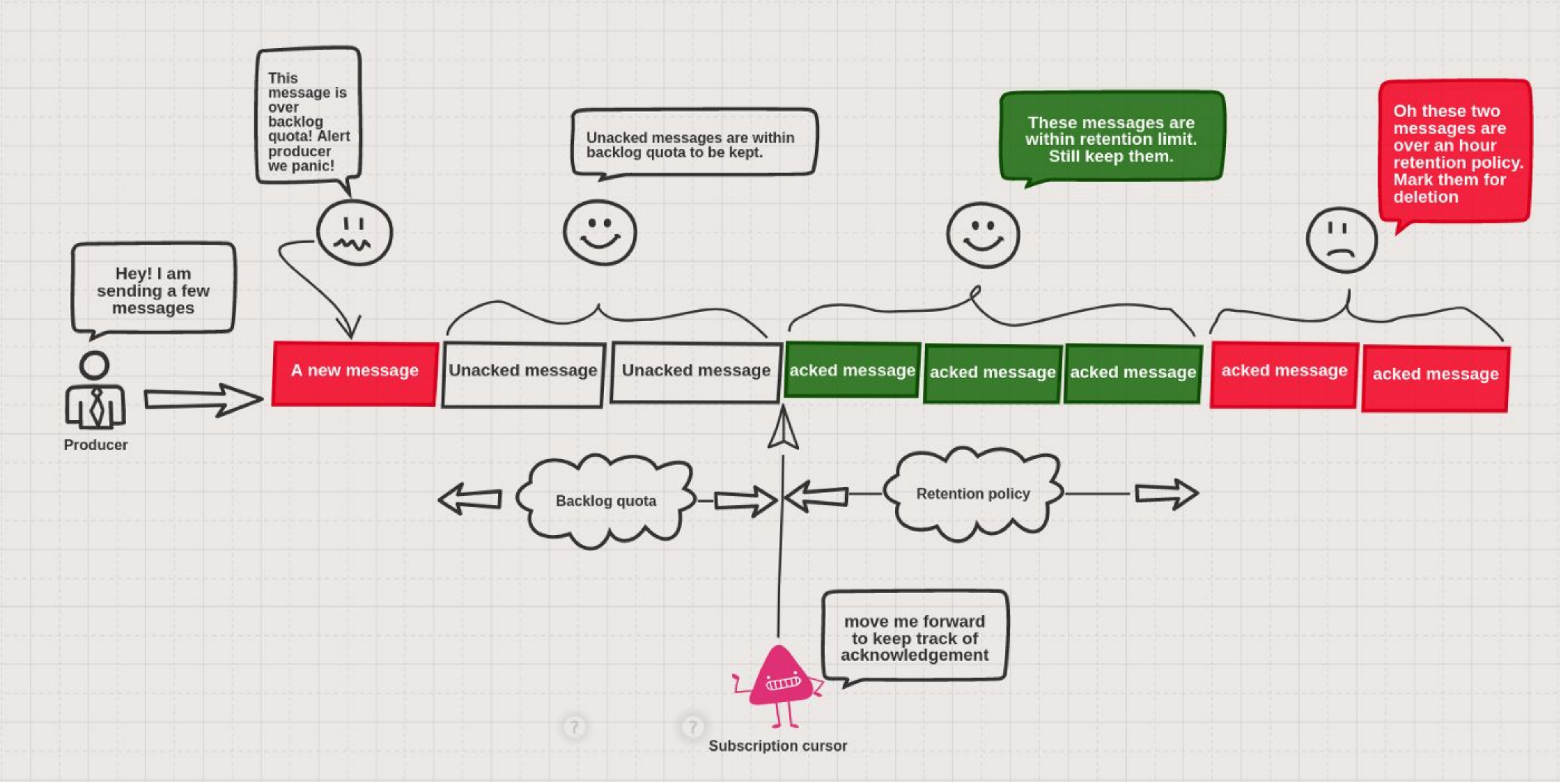
- Backlog
  - all the unacked messages of a subscription
  - the slowest subscription determines the size
  - stats (backlogSize, msgBacklog)
- TTL
  - namespace setting
- Policy to manage backlog quota breach
  - producer_request_hold
  - producer_exception
  - consumer_backlog_eviction


This message is over backlog quota! Alert producer we panic!
Backlog of unacked messages
Ready for deletion
Producer
A new message
acked message
TTL is tracked in each message
Subscription cursor

Retention policy

- Retain acknowledged messages
- Namespace settings
  - defaultRetentionTimeInMinutes and defaultRetentionTimeInSize
- Message deletion at per ledger basis, not operate against individual messages
  - Unacked message in a ledger prevent deletion




This message is over backlog quota! Alert producer we panic!
Unacked messages are within backlog quota to be kept.
These messages are within retention limit. Still keep them.
Oh these two messages are over an hour retention policy. Mark them for deletion
Hey! I am sending a few messages
Producer
A new message
Unacked message
Unacked message
acked message
acked message
acked message
acked message
acked message
Backlog quota
Retention policy
move me forward to keep track of acknowledgement
Subscription cursor

Storage

- Ledger is message storage operation unit
    - Deletion is at per ledger basis
    - Message is not deleted individually

- Theoretically, Storage size = backlog size + retention size
- Storage size = sum of all the ledgers that still contain at least a message satisfy backlog or retention policy rule
- Disk size > storage size
    - Data striping
    - BK compaction trigger

Tiered Storage

- Pulsar is tiered: compute, storage
- Further tier in storage
  - Offload older messages (ledgers)
  - S3, Google Cloud Storage, Azure Blob, HDFS
- Transparent to client
- Supports event sourcing
- Offers longer message retention period
- Savings:
  - Cloud storage is significantly cheaper than SSDs
  - Further tiering (infrequent access, glacier)

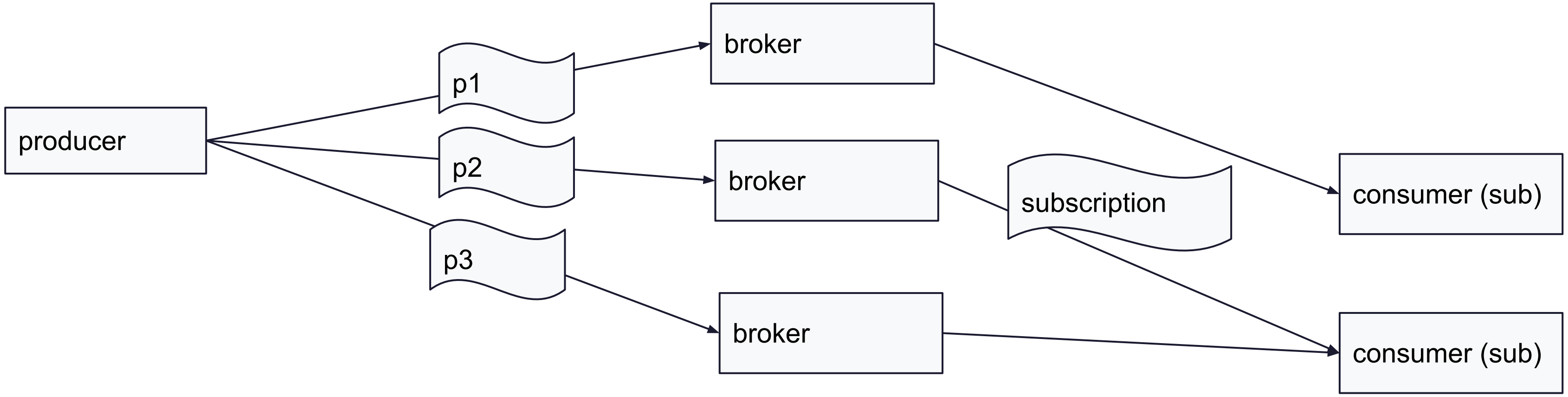
Producer

- Send messages to broker(s)
- A local message queue
  - maxPendingMessages - queue size
  - blockIfQueueFull
  - queue management
    - working with broker using receipt
- Message batching
  - Batches tracked and stored as a single unit
  - Redelivery concern of unacked message
    - `batchIndexAcknowledgeEnable`

Partitions

- Partitions if necessary
- Partitions are also unit of parallelism so allow for scaling high volume topics
- Message routing
  - Routing mode
    - Partitions hash by key, round robin, or custom by message producing
  - Hashing Scheme
- Can dynamically update partitions in a topic
- All topics will be becoming partitioned topic
- Message ordering guarantee
  - messages from the same producer
  - messages with the same key in the same partition

Producer cont'd



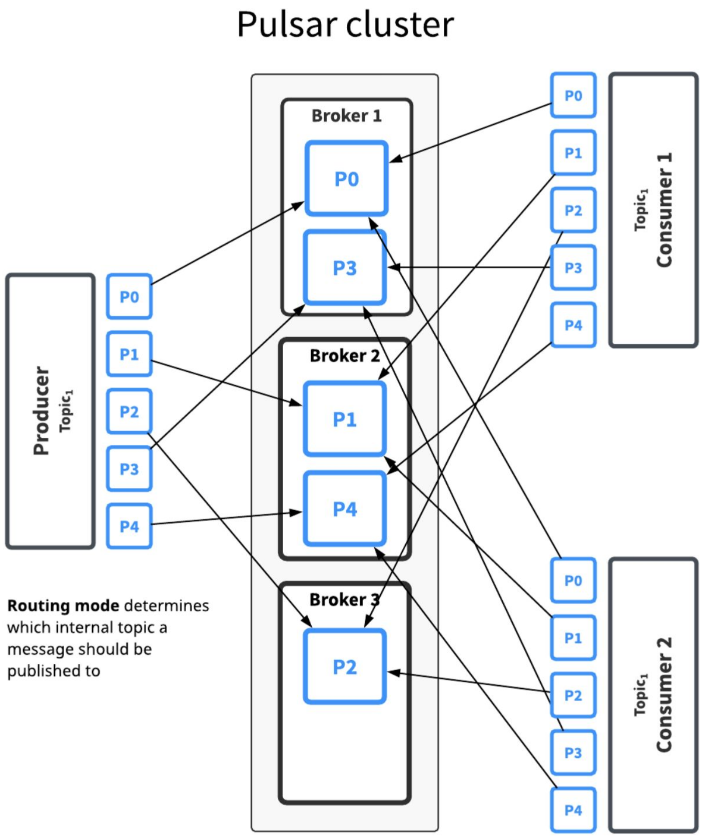
Two consumers with the same shared subscription

Producer

- Attach to a topic and publish to a broker
- Sync vs. Async send
- Message compression
  - LZ4, ZLIB, ZSTD, SNAPPY
- Batching
  - Batches tracked and stored as a single unit
  - Acknowledgement
    - As a whole
    - Individual: Batch index acknowledgement (since 2.6.0)
- Chunking
  - Publish-payload threshold size
  - Mutually exclusive with batching
  - Applies only to exclusive and fail-over subscription model (message ordering is required)
  - Consumer maintains a buffer and combines them together when all chunked messages have been collected

Partitioned Topic

- Partition is not bound by the number of brokers!
- Implemented as N internal topics
- Producer side:
  - Parallelism control
  - Increase throughput
  - Routing Mode
    - With key - based on key hashing
    - No key
      - Round-robin
      - Single-partition
      - Custom-partition



Routing mode determines which internal topic a message should be published to

Subscription mode determines which consumer(s) a message should be delivered to

Subscriptions and consuming messages

- Consumer uses subscription tracks message consumption
- Subscription cursor is a reference pointer of message Id to track current consumption
    - use cursor to keep track of many things (replay, skip, rewind, seek, reader, subscription, ledger deletion, and etc.)

Consumer queue and queue management (internal stuff)

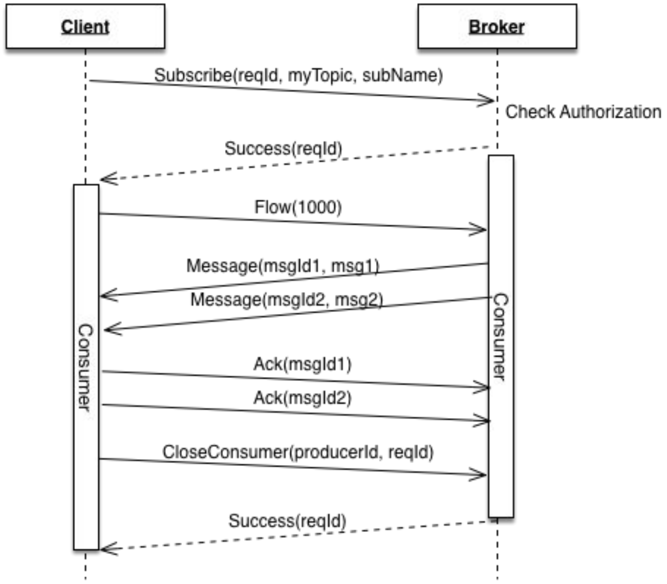
Demo cursor, ledger, producers and subscriptions and consumers
pulsar stats and stats-internal

Consumer (cont'd)

- Subscribe to multiple topics
  - Same namespace
  - Topic list
  - Topic pattern of regex
- Subscription initial positions
  - Earliest, latest
- replicateSubscriptionState (geo-replication)

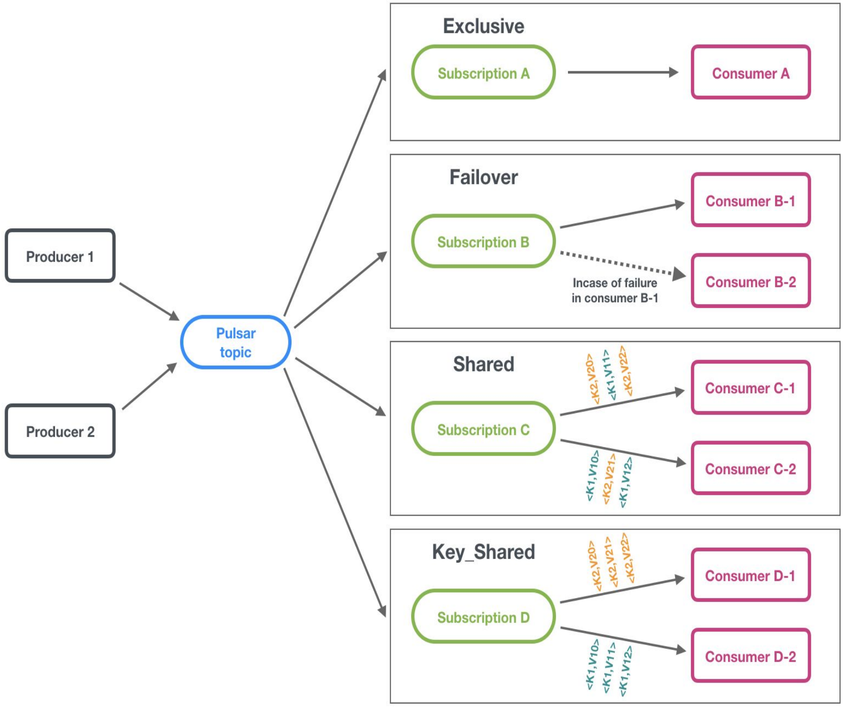
Consumer

- Attach to a **subscription** and consume from a broker
    - Subscription cursor
- Sync vs. Async receive
    - Client library listener interface
- Consumer requests to receive messages via Flow control
    - Receiver queue
- Acknowledgement
    - More details later...



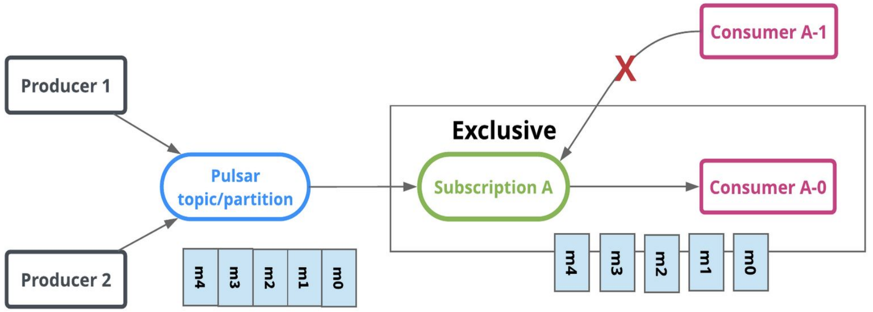
Consumer Subscription

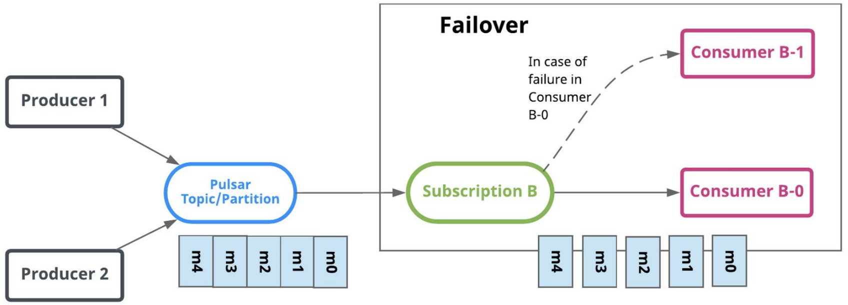
- No Kafka "Consumer Group"
- No Kafka "Partition" constraints/limitations
- Subscription Modes
    - Determines message delivery to consumers
    - **No difference between non-partitioned and partitioned topic**



Subscription Mode

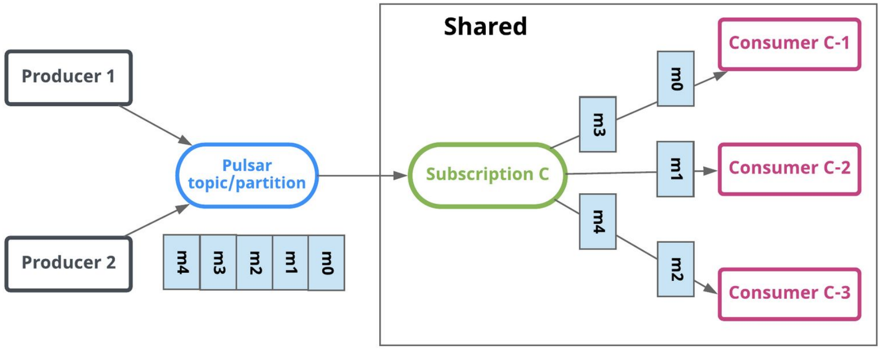
- Exclusive Mode (default)
  - Only one consumer is allowed to attach to a subscription.
- Failover
  - Allows Multiple consumers attached to the same subscription
  - Only one master consumer
    - Highest priority level
    - Alphabetical order of consumer names
  - Messages are delivered to the master consumer only
  - Deliver messages to the next consumer in line when the master fails
    - Subsequent messages
    - Non-acked messages





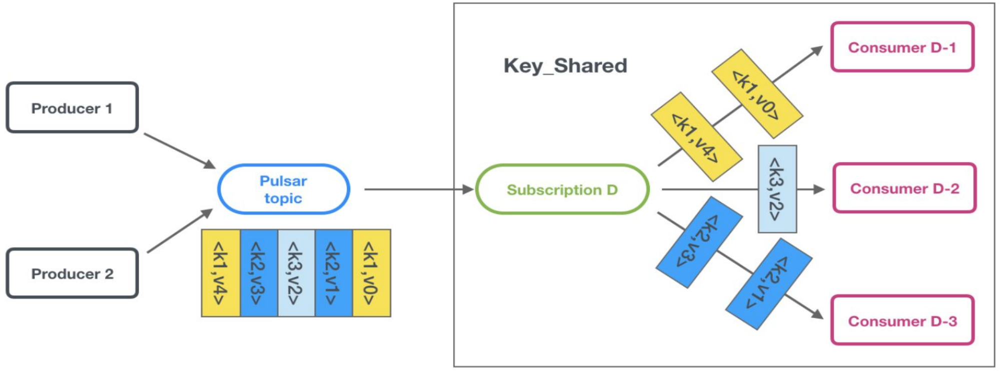
Subscription Mode, continued

- Shared/Round-robin Mode
  - Allow multiple consumers attached to the same subscription
  - Message distributed in round-robin fashion among consumers
  - Any message is only distributed to one consumer
  - When a consumer fails, all sent messages to it that are not acked will be rescheduled to the remaining consumers.



Subscription Mode, continued

- Key-Shared Mode
  - Allow multiple consumers attached to the same subscription
  - Message distributed among consumers by message keys
    - Hash range space divided among consumers (*keySharedPolicy*)
      - Consistent auto split hash
        - Automatic rebalance hash range rebalance among consumers
      - Sticky hash range
        - Manual hash range calculation and rebalance
  - Messages with the same key are delivered to the same consumer



Message Consumption Flexibility

- Fan-out pub-sub
  - Multiple consumers
  - Unique subscription name per consumer
  - Subscription mode: exclusive
- Queuing
  - Multiple consumers
  - Share the same subscription name
  - Subscription mode: shared, or key-shared
- Even more flexible options
  - Combine the above two somehow

Multi-topic Subscription

- By default, one consumer subscribes to one topic
- Multi-topic subscription is doable:
  - Explicit list of topics
  - Regex pattern of topic names
- Belong to the same namespace

Reader

- Read from topic directly (no subscription)
- Need to explicitly specify message ID
    - pulsar.EarliestMessage()
    - pulsar.LatestMessage()
    - pulsar.DeserializeMessageID(lastSavedId)
- Read API
    - HasNext()
    - Next()

Transaction

- New in Pulsar 2.7.0
  - Enable to consume, process, and produce messages in one atomic operation
    - Atomic multi-topic(partition) writes
    - Atomic multi-subscription acknowledgement
  - Transaction coordinator
  - Transaction log (Pulsar topic)
  - Transaction ID (128-bit; highest 16-bit for coordinator)

Message Deduplication (Producer)

- When enabled, each message produced on Pulsar topics is persisted to disk ***only once***
    - Even if the message is produced more than once
- Handled automatically on the server side
- Can be enabled at different levels (disabled by default)
    - Broker (config. file)
    - Namespace (pulsar-admin cli)
    - Topic (pulsar-admin cli)
- Producer client requirements
    - Producer name
    - No message timeout

Consumer (cont'd) – shared subscription – DLQ

```
Consumer<byte[]> consumer = pulsarClient.newConsumer(Schema.BYTES)
            .topic(topic)
            .subscriptionName("my-subscription")
            .subscriptionType(SubscriptionType.Shared)
            .deadLetterPolicy(DeadLetterPolicy.builder()
                 .maxRedeliverCount(maxRedeliveryCount)
                 .build())
            .subscribe();
```

Dead letter Topics use the following naming standards

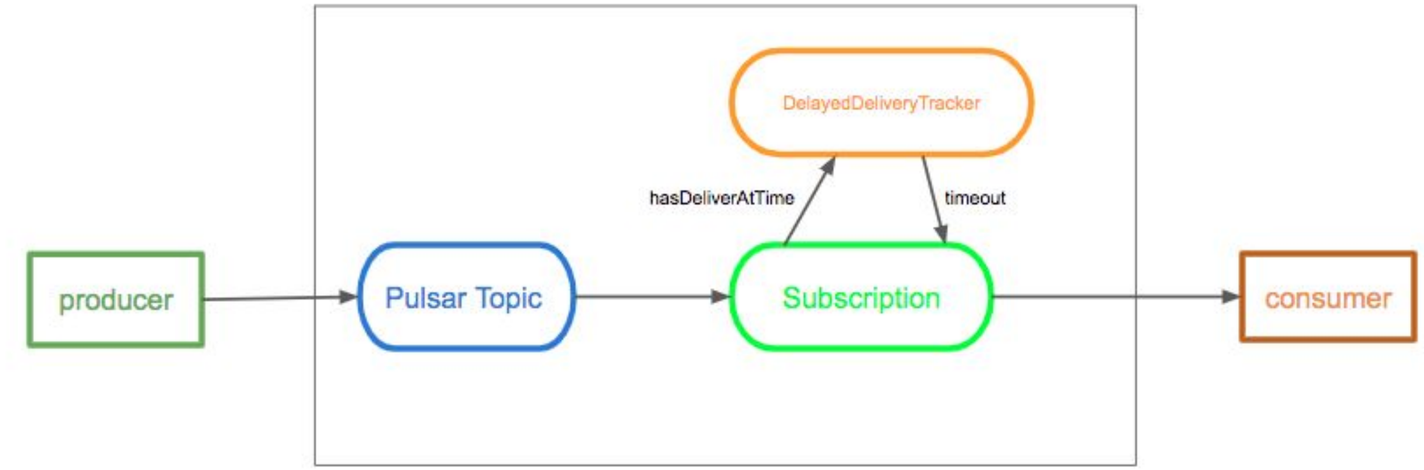
```
<topicname>-<subscriptionname>-DLQ
```

Consumer (cont'd) - shared subscription - Consumer Priority

```
setPriorityLevel
public void setPriorityLevel(int priorityLevel)

Sets priority level for the shared subscription consumers to which broker gives more priority while
dispatching messages. Here, broker follows descending priorities. (eg: 0=max-priority, 1, 2,..)
In Shared subscription mode, broker will first dispatch messages to max priority-level consumers if they
have permits, else broker will consider next priority level consumers.

If subscription has consumer-A with priorityLevel 0 and Consumer-B with priorityLevel 1 then broker will
dispatch messages to only consumer-A until it runs out permit and then broker starts dispatching messages to
Consumer-B.

 Consumer PriorityLevel Permits
 C1       0             2
 C2       0             1
 C3       0             1
 C4       1             2
 C5       1             1


 Order in which broker dispatches messages to consumers: C1, C2, C3, C1, C4, C5, C4
```

Delayed Message

- Producer can send a delayed message that will be delivered after a delay time by broker.
- Only available on shared subscription



```
// message to be delivered at the configured delay interval
producer.newMessage().deliverAfter(3L, TimeUnit.Minute).value("Hello Pulsar!").send();
```

Message Acknowledgement (Consumer)

- A message is persistently stored and deleted only after all the subscriptions have acknowledged it.
- Individual acknowledgement
- Cumulative acknowledgement
  - Acknowledge the last message
  - Upstream message won’t be re-delivered
  - Not applicable with Shared subscription mode
- Acknowledgement timeout
  - Automatic re-delivery
  - *acktimeout*
- Negative acknowledgement
  - Explicit re-delivery notification
  - Preferred over Acknowledgement timeout
    - More precise control
    - Avoid invalid re-delivery when consumer processing time is longer than *acktimeout*

Message Acknowledgement, continued

- Dead letter topic (DLT)
  - Allows the consumption of new messages when old messages are failed to be consumed
  - A dedicated, separate topic (DLT)
- Retry letter topic (RLT)
  - Automatic retry after a configurable delay time
  - Disabled by default
  - Original topic and a separate, dedicated retry letter topic (RLT)

Message Retention and Expiry (refresher)

- By default, brokers do:
  - Immediately delete all messages that are acknowledged
  - Persistently store all unacknowledged messages
- Retention Policy
  - Store messages that have been acknowledged
  - Namespace level
- Expiry Policy
  - Set TTL for messages that have not yet been acknowledged
  - Namespace level

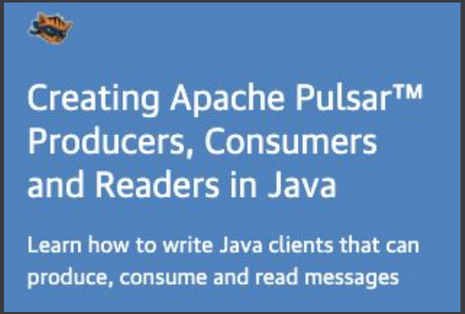
DS

LAB TIME!

🤓 💻 👇

dtsx.io/ess-ep2-lab

Select ONLY this one →



Creating Apache Pulsar™ Producers, Consumers and Readers in Java

Learn how to write Java clients that can produce, consume and read messages

DS

Schema Management With Pulsar

Schema Registry

- Data governance
- Type safety is important when producers/consumers are decoupled
- Built-in schema registry (stored in bookkeeper)
- Clients/REST API can register a schema for a topic
    - Primitive: string, integer
    - Complex: Key/Value, Avro, JSON
- Pulsar stores the current schema, ensures producers and consumers conform
- Schema validation
- Supports schema evolution

Python Example

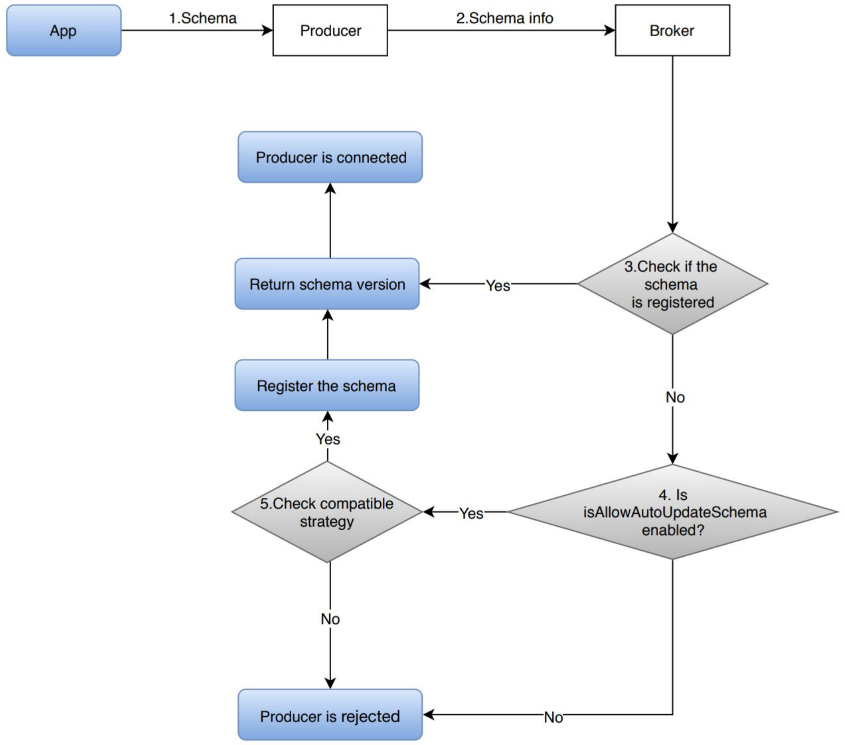
```
import json
import datetime
import urllib.request, json
import pulsar


from pulsar.schema import *

class Covid19(Record):
    date = String()
    confirmed = Integer()
    deaths = Integer()
    recovered = Integer()
    country = String()
```

Python Example (continued)

```
client = pulsar.Client('pulsar://pulsar:6650')

producer = client.create_producer(topic='covid19',schema=AvroSchema(Covid19))


count=0
for key in covid:
    print("Processing all entries for -> "+ key)
    for i in covid[key]:
        record = Covid19(date=i['date'],country=i['country'],confirmed=i['confirmed'],deaths=i['deaths'],recovered=i['recovered'] )
        producer.send( partition_key=record_date, content=record )
        count+=1

client.close()
```

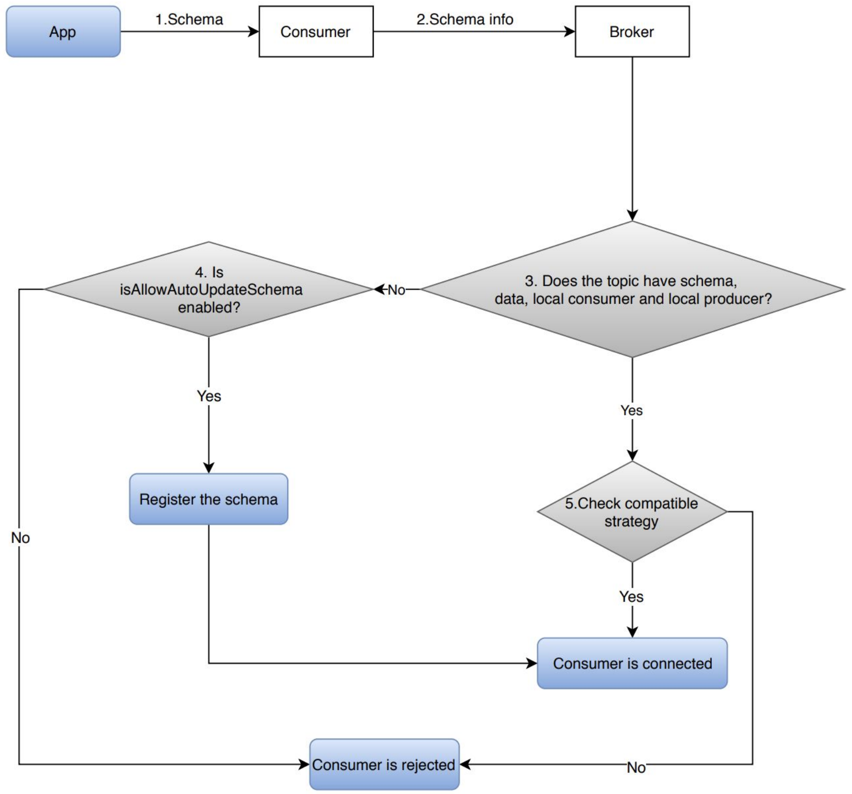
Scheme management

Producer side how schema is managed



Schema management

Consumer side how schema is managed


App
1.Schema
Consumer
2.Schema info
Broker
3. Does the topic have schema, data, local consumer and local producer?
No
4. Is isAllowAutoUpdateSchema enabled?
Yes
Register the schema
No
Yes
5.Check compatible strategy
Yes
Consumer is connected
Consumer is rejected
No

SQL on Pulsar Topics

Pulsar SQL

- Pulsar messages must have well-defined schema
  - Pulsar has a built-in schema registry
  - Schema per topic
  - Only available for Java, Python, CGo and C++ clients
- Presto
  - Distributed SQL engine
  - Coordinator and workers
- Presto-Pulsar connector
  - Workers retrieve data from bookies
- Presto/Pulsar SQL client
  - SELECT, SHOW, DESCRIBE

Pulsar Sql Java Example

Define a Data Schema

```
public class JobSearchRequest {
    private String traceId;       // Identify a job search request.
    private Long userId;          // Who sent the job search request, the value can be null.
    private Date requestTime;     // Request time of the search request.
    private int cityId;           // City ID that user specified.
    private String keywords;      // Keywords that user input of the search request.
}
public class JobSearchResponse {
          private String traceId;       // Identify a job search request.
          private long jobId;           // The returned job ID for the user.
}
```

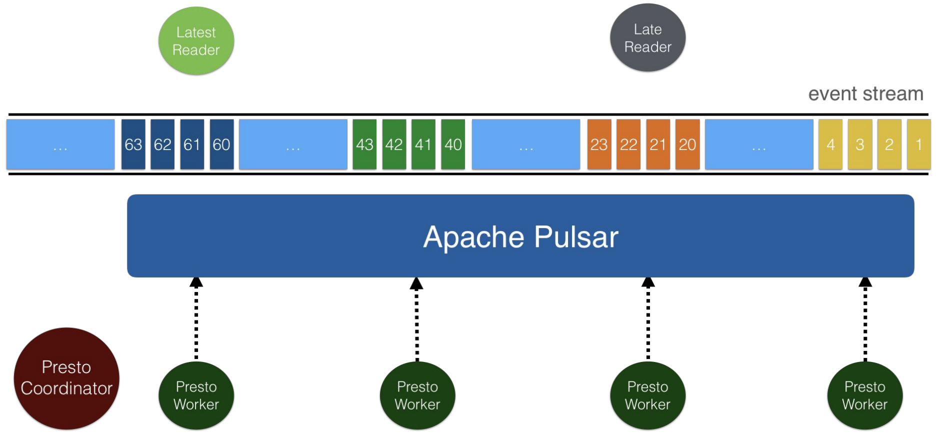
To display the top ten keywords in today's job search requests

```
SELECT count(*) as c, keywords FROM pulsar."search/job".requests WHERE requestTime = current_date
GROUP BY keywords ORDER BY c DESC LIMIT 10;
```

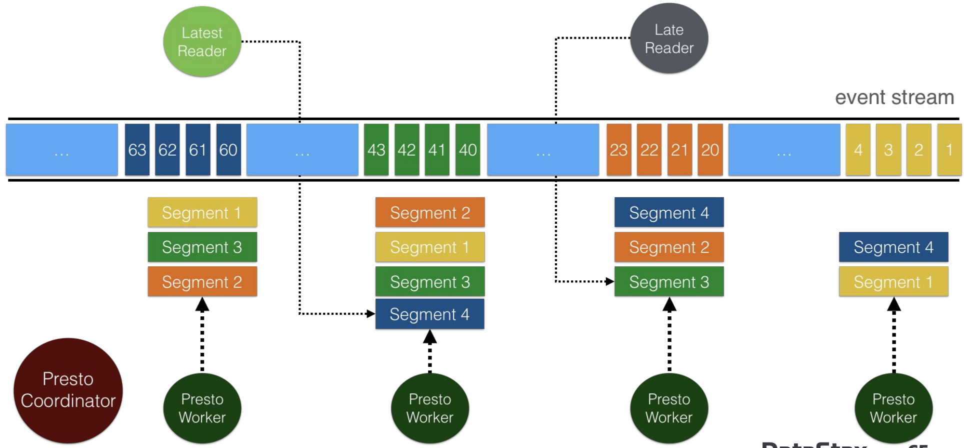
Display Top Jobs that yielded no job

```
SELECT traceId FROM pulsar."search/job".requests EXCEPT SELECT traceId FROM
pulsar."search/job".response;
```

Presto Architecture on Pulsar



```
presto> SELECT * FROM system.runtime.nodes;
 node_id |        http_uri         | node_version | coordinator | state
---------+-------------------------+--------------+-------------+--------
 1       | http://192.168.2.1:8081 | testversion  | true        | active
 3       | http://192.168.2.2:8081 | testversion  | false       | active
 2       | http://192.168.2.3:8081 | testversion  | false       | active
```



Understanding Presto-Pulsar Terminology

Presto	Pulsar
catalog	N/A
schema	namespace
table	topic
row	message

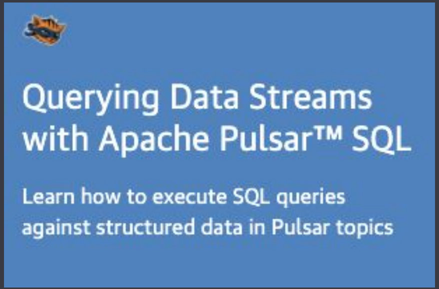
DS

LAB TIME!

🤓 💻 👇

dtsx.io/ess-ep2-lab

Select ONLY this one →



Querying Data Streams with Apache Pulsar™ SQL

Learn how to execute SQL queries against structured data in Pulsar topics

QUIZ TIME!

DataStax

SUBSCRIBE NOW: dtsx.io/ess-ep3

NEXT WEEK:

June 14th, same time!

Episode III: Pulsar in Action

Led By Aleks and Zeke

DataStax

SUBSCRIBE NOW: dtsx.io/nosql-1606

THIS WEEK:

16th / 17th of June

Introduction to NoSQL Databases

Led By Cedrick, David, Ryan and Aleks

DataStax

Cassandra Day India

Mark the date: July the 10th

One day, two tracks, over 12 hours of pure data knowledge with the best NoSQL experts and guest speakers. Boost your career and reach the new level with DataStax Developers!

DataStax

Cassandra Day India

Call for Papers is open: dtsx.io/cdindia-cfp

DS

THANK YOU!