WHITE PAPER

Redis for Fast Data Ingest

Roshan Kumar, Senior Product Marketing Manager, Redis

CONTENTS

Abstract	**2**
Background	**2**
Challenges in Designing Fast Data Ingest Solutions	**2**
Handling Fast Data Ingest in Redis	**2**
Redis at the Speed of Twitter	**3**
Sample Solutions	**3**
Ingest with Redis Pub/Sub	4
Ingest with Lists	8
Ingest using Sorted Sets	11
Tips for Performance Improvement and Scaling	**17**
Conclusion	**17**

Abstract

Real-time streaming data ingest is a common requirement for many big data use cases. This whitepaper outlines how the in-memory database platform, Redis Enterprise, can solve common challenges associated with ingestion and processing of large volumes of high velocity data. Redis, because of its high performance, is a popular choice for such fast data ingest scenarios. This database achieves throughput of millions of operations/second with sub-millisecond latencies using minimal resources and offers simple implementations, enabled by its multiple data structures and functions.

Background

Collecting, storing and processing streaming data in large volume and velocity is a task that presents architectural challenges – especially in fields like IoT, e-commerce, security, communications, entertainment, finance, and retail. And since responsive, timely and accurate data-driven decision making is core to these businesses, real-time data collection and analysis are critical.

An important first step in delivering real-time data analysis is ensuring adequate resources are available to effectively capture fast data streams. While the physical infrastructure (including a high-speed network, computation, storage and memory) plays an important role here, a company's software stack must match the performance of its physical layer. Otherwise, businesses can face a massive backlog of data, missing data or incomplete, misleading data.

Challenges in Designing Fast Data Ingest Solutions

High-speed data ingestion often involves several different types of complexity:

1. **Large volumes of data sometimes arriving in bursts:** Bursty data requires a solution that is capable of processing large volumes of data with minimal latency. Ideally, it should be able to perform millions of writes per second with sub-millisecond latency, using minimal resources.
2. **Data from multiple sources/formats**: Data ingest solutions must also be flexible enough to handle data in many different formats, retaining source identity if needed and transforming or normalizing in real-time.
3. **Data that needs to be filtered, analyzed or forwarded**: Most data ingest solutions have one or more subscribers who consume the data. These are often different applications that function in the same or different locations with a varied set of assumptions. In such cases, the database not only needs to transform the data, but also filter or aggregate depending on the requirements of the consuming applications.
4. **Data coming from geographically distributed sources**: In this scenario, it is often convenient for the underlying architecture to distribute data collection nodes close to the source. The nodes themselves become part of the fast data ingest solution, to collect, process, forward or reroute ingest data.

Handling Fast Data Ingest in Redis

Many solutions supporting fast data ingest today are complex, feature-rich and over-engineered for simple requirements. Redis, on the other hand, is extremely light-weight, super-fast and easy-to-use. With clients available in over 60 languages, Redis can be easily integrated with all popular software stacks.

Redis offers data structures such as Lists, Sets, Sorted Sets and Hashes that offer simple and versatile data processing. Redis delivers over a million read/write operations per second, with sub-millisecond latency on a modest-sized commodity cloud instance, making it the most resource-efficient for large volumes of data. It also supports messaging services and client libraries in all popular programming languages, making it best suited for combining high-speed data ingest and real-time analytics. Redis Pub/Sub commands make it a message broker between publishers and subscribers, a feature often used to notify or carry messages between distributed data ingest nodes.

Redis Enterprise enhances Redis with seamless scaling, always-on availability, automated deployment and the ability to use cost-effective Flash memory as a RAM extender so that large dataset processing can be accomplished cost-effectively.

The below sections outline how to use Redis Enterprise to address common data ingest challenges.

Redis at the Speed of Twitter

To illustrate the simplicity of Redis, we'll explore a sample fast data ingest solution that gathers messages from a Twitter feed. The goal of this solution is to process tweets in real-time and push them down the pipe as they are processed.

Twitter data ingested by the solution is then consumed by multiple processors down the line. As shown in Figure 1, this example deals with two processors – 1. English Tweet Processor, and 2. Influencer Processor. Each processor filters the tweets and passes them down its respective channels to other consumers. This chain can go as far as the solution requires. However, in our example, we stop at the third level, where we aggregate popular discussions among English speakers and top influencers.

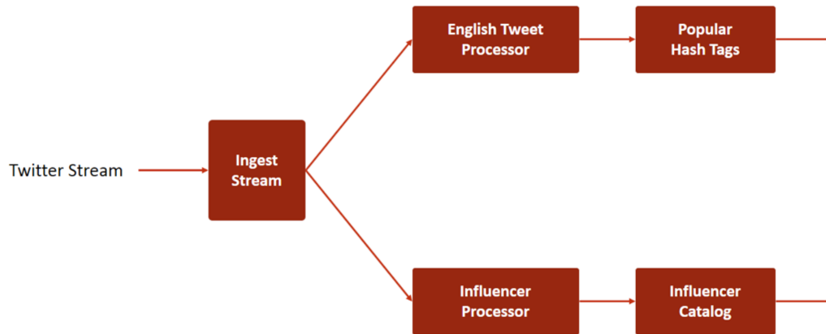

Figure 1. *Flow of the Twitter stream*

Note that:

- We are using the example of processing Twitter feeds because of the velocity of data arrival and simplicity.
- Twitter data reaches our fast data ingest via a single channel. In many cases, such as IoT, there could be multiple data sources sending data to the main receiver.

Sample Solutions

Here are three possible ways to implement this solution using Redis:

1. Ingest with Redis Pub/Sub
2. Ingest with List data structure
3. Ingest with Sorted Set data structure

Ingest with Redis Pub/Sub

This is the simplest implementation of fast data ingest. This solution uses Redis' Pub/Sub feature, which allows applications to publish and subscribe to messages. As shown in Figure 2, each stage processes the data and publishes it to a channel. The subsequent stage subscribes to the channel and receives the messages for further processing or filtering.

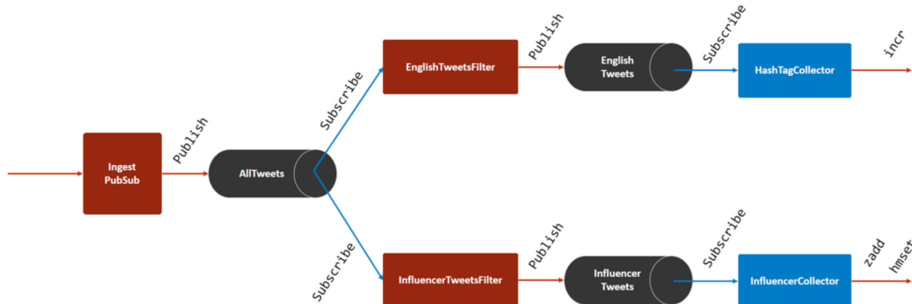

Figure 2. *Data ingest using Redis Pub/Sub*

Pros

4. Easy to implement
5. Works well when the data sources and processors are distributed geographically.

Cons

1. The solution requires the publishers and subscribers to be up all the time. Subscribers lose data when stopped, or when the connection is lost.
2. It requires more connections. A program cannot publish and subscribe to the same connection, so each intermediate data processor requires two connections – one to subscribe and one to publish. If running Redis on a DBaaS platform, it is important to verify whether your package or level of service has any limits to the number of connections.

A note about connections

If more than one client subscribes to a channel, Redis pushes the data to each client linearly one after the other. Large data payloads and many connections may introduce latency between a publisher and its subscribers. Though the default hard limit for maximum number of connections is 10,000, you must test and benchmark how many connections are appropriate for your payload.

Client output buffer limits: Redis maintains a client output buffer for each client. The default limits for the client output buffer for Pub/Sub are set as:

```
client-output-buffer-limit pubsub 32mb 8mb 60
```

With this setting, Redis will force clients to disconnect under two situations:

1. If the output buffer grows beyond 32mb
2. If the output buffer holds 8mb of data consistently for 60 seconds

This typically occurs when clients consume the data slower than it is published. Should such a situation arise, first try optimizing the consumers such that they do not add latency while consuming the data. If you notice that your clients are still getting disconnected, then you may increase the limits for the "client-output-buffer-limit pubsub" property in redis.conf. Please keep in mind that any changes to the settings may increase latency between the publisher and subscriber. Any changes must be tested and verified thoroughly.

Code Design

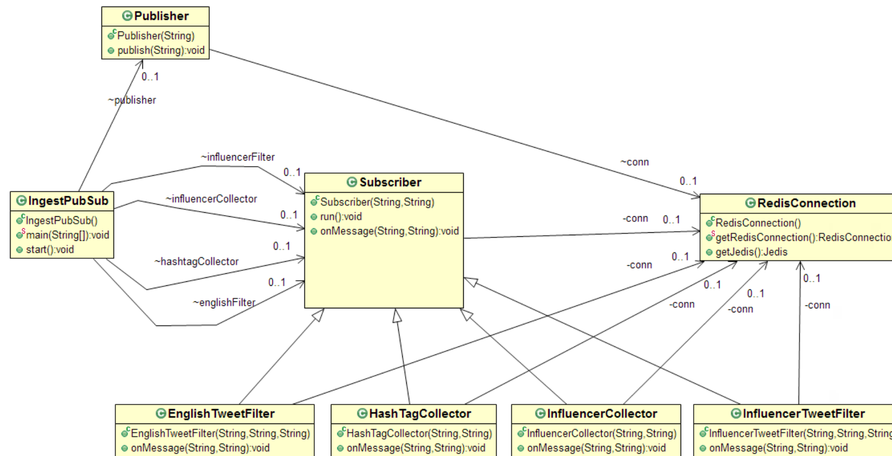

Figure 3. *Class diagram of Fast Data Ingest Solution with Pub/Sub*

This is the simplest of the three solutions described in this paper. Here are the important Java classes implemented for this solution. Download the source code with full implementation here: https://github.com/redisdemo/IngestPubSub.

1. Subscriber class is the core class of this design. Every Subscriber object maintains a new connection with Redis.

```
class Subscriber extends JedisPubSub implements Runnable{
      private String name = "Subscriber";
      private RedisConnection conn = null;
      private Jedis jedis = null;

      private String subscriberChannel = "defaultchannel";

      public Subscriber(String subscriberName, String channelName) throws Exception{
              name = subscriberName;
              subscriberChannel = channelName;
              Thread t = new Thread(this);
              t.start();
      }

      @Override
      public void run(){
              try{
                      conn = RedisConnection.getRedisConnection();
                      jedis = conn.getJedis();
                      while(true){
                              jedis.subscribe(this, this.subscriberChannel);
                      }
              }catch(Exception e){
                      e.printStackTrace();
              }
      }
```

```
        @Override
        public void onMessage(String channel, String message){
                super.onMessage(channel, message);
        }
}
```

2. Publisher class maintains a separate connection to Redis for publishing messages to a channel.

```
public class Publisher{

        RedisConnection conn = null;
        Jedis jedis = null;

        private String channel = "defaultchannel";

        public Publisher(String channelName) throws Exception{
                channel = channelName;
                conn = RedisConnection.getRedisConnection();
                jedis = conn.getJedis();
        }
        public void publish(String msg) throws Exception{
                jedis.publish(channel, msg);
        }
}
```

3. Filters (EnglishTweetFilter, InfluencerTweetFilter, HashTagCollector, InfluencerCollector) extend Subscriber, which enables them to listen to the inbound channels. Since you need separate Redis connections for subscribe and publish, each filter class has its own RedisConnection Object. Filters listen to the new messages in their channels in a loop. Here's the sample code of EnglishTweetFilter class:

```
public class EnglishTweetFilter extends Subscriber
{
        private RedisConnection conn = null;
        private Jedis jedis = null;
        private String publisherChannel = null;

        public EnglishTweetFilter(String name, String subscriberChannel,
                String publisherChannel) throws Exception{
                        super(name, subscriberChannel);
                        this.publisherChannel = publisherChannel;
                        conn = RedisConnection.getRedisConnection();
                        jedis = conn.getJedis();
                }
                @Override
```

```
            public void onMessage(String subscriberChannel, String message){
                    JsonParser jsonParser = new JsonParser();

                    JsonElement jsonElement = jsonParser.parse(message);
                    JsonObject jsonObject = jsonElement.getAsJsonObject();

                    //filter messages: publish only English tweets
                    if(jsonObject.get("lang") != null &&
                            jsonObject.get("lang").getAsString().equals("en")){
                            jedis.publish(publisherChannel, message);
                    }
            }
    }
```

4. Publisher: This class has a publish method that publishes message to the required channel.

```
public class Publisher{
.
.
      public void publish(String msg) throws Exception{
              jedis.publish(channel, msg);
      }
.
}
```

5. Main class: The main class reads data from the ingest stream and posts it to the “AllData” channel. The main method of this class starts all of the filter objects.

```
public class IngestPubSub
{
.
      public void start() throws Exception{
      .
      .
              publisher = new Publisher("AllData");

              englishFilter = new EnglishTweetFilter("English Filter","AllData",
              "EnglishTweets");
              influencerFilter = new InfluencerTweetFilter("Influencer Filter","All-
Data",                       "InfluencerTweets");
              hashtagCollector = new HashTagCollector("Hashtag Collector",
              "EnglishTweets");
              influencerCollector = new InfluencerCollector("Influencer Collector",
              "InfluencerTweets");
      .
      .
}
```

Ingest with Lists

The List data structure in Redis makes implementing a queueing solution easy and straightforward. In this solution, the producer pushes every message to the back of the queue, and the subscriber polls the queue and pulls new messages from the other end.

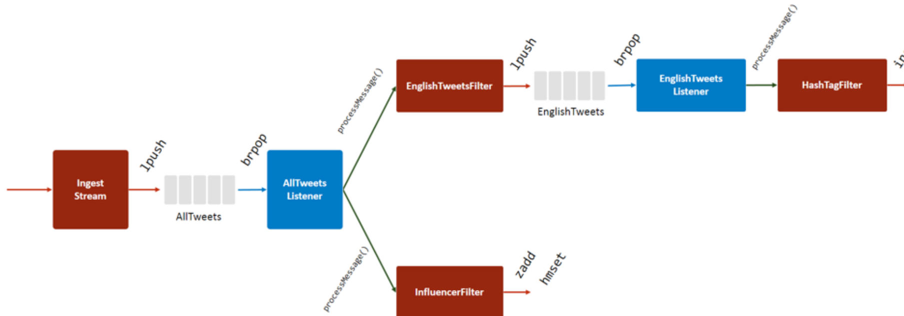

Figure 4. Fast data ingest with Redis Lists

Pros

1. This method is reliable in cases of connection loss. Once data is pushed into the lists, it is preserved there until the subscribers read it. This is true even if the subscribers are stopped or lose connection with the Redis server.
2. Producers and consumers require no connection between them.

Cons

1. Once data is pulled from the list, it is removed and cannot be retrieved again. Unless the consumers persist the data, it is lost as soon as it is consumed.
2. Every consumer requires a separate queue, which requires storing multiple copies of the data.

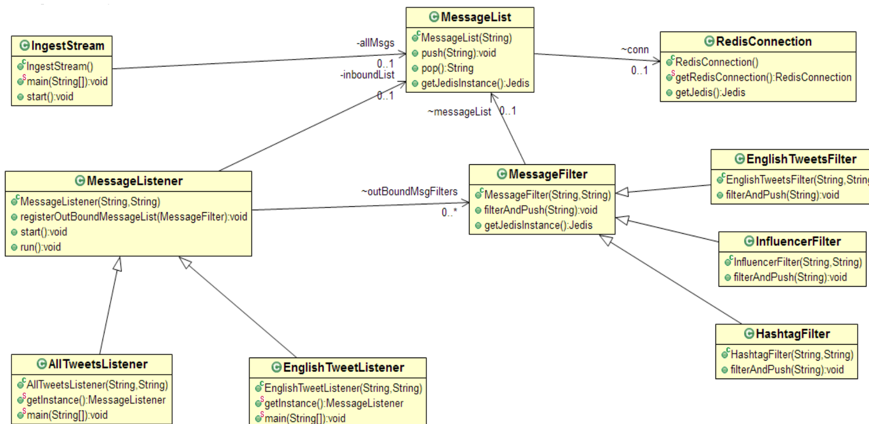

Figure 5. Class diagram of Fast Data Ingest Solution with Lists

Download the source code here: https://github.com/redisdemo/IngestList. This solution's main classes are explained below:

1. MessageList embeds the Redis List data structure. The push() method pushes the new message to the left of the queue, and pop() waits for a new message from the right if the queue is empty.

```
public class MessageList{
      protected String name = "MyList"; // Name
.
.
      public void push(String msg) throws Exception{
              jedis.lpush(name, msg); // Left Push
      }
      public String pop() throws Exception{
              return jedis.brpop(0, name).toString();
      }
.
.
}
```

2. MessageListener is an abstract class that implements listener and publisher logic. A MessageListener object listens to only one list, but can publish to multiple channels (MessageFilter objects). This solution requires a separate MessageFilter object for each subscriber down the pipe.

```
class MessageListener implements Runnable{
      private String name = null;
      private MessageList inboundList = null;
      Map<String, MessageFilter> outBoundMsgFilters =
                                    new HashMap<String, MessageFilter>();
.
.
      public void registerOutBoundMessageList(MessageFilter msgFilter){
              if(msgFilter != null){
                      if(outBoundMsgFilters.get(msgFilter.name) == null){
                              outBoundMsgFilters.put(msgFilter.name, msgFilter);
                      }
              }
      }
.
.
      @Override
      public void run(){
.
              while(true){
                      String msg = inboundList.pop();
                      processMessage(msg);
              }
.
      }
.
      protected void pushMessage(String msg) throws Exception{
              Set<String> outBoundMsgNames = outBoundMsgFilters.keySet();
```

```
            for(String name : outBoundMsgNames ){
                   MessageFilter msgList = outBoundMsgFilters.get(name);
                   msgList.filterAndPush(msg);
            }
     }
}
```

3. MessageFilter is a parent class facilitating the filterAndPush() method. As data flows through the ingest system, it is often filtered or transformed before being sent to the next stage. Classes that extend the MessageFilter class override the filterAndPush method, and implement their own logic to push the filtered message to the next list.

```
public class MessageFilter{

     MessageList messageList = null;
.
.
     public void filterAndPush(String msg) throws Exception{
            messageList.push(msg);
     }
.
.
}
```

4. AllTweetsListener is a sample implementation of a MessageListener class. This listens to all tweets on the “AllData” channel, and publishes the data to “EnglishTweetsFilter” and “InfluencerFilter”

```
public class AllTweetsListener extends MessageListener{
.
.
     public static void main(String[] args) throws Exception{
            MessageListener allTweetsProcessor = AllTweetsListener.getInstance();

            allTweetsProcessor.registerOutBoundMessageList(
                   new EnglishTweetsFilter("EnglishTweetsFilter", "EnglishTweets"));

            allTweetsProcessor.registerOutBoundMessageList(
                   new InfluencerFilter("InfluencerFilter", "Influencers"));

            allTweetsProcessor.start();
     }
.
.
}
```

5. EnglishTweetsFilter extends MessageFilter. This class implements logic to select only those tweets that are marked as English tweets. The filter discards non-English tweets and pushes English tweets to the next list.

```
public class EnglishTweetsFilter extends MessageFilter{

     public EnglishTweetsFilter(String name, String listName) throws Exception{
            super(name, listName);
     }
```

```
        @Override
        public void filterAndPush(String message) throws Exception{
                JsonParser jsonParser = new JsonParser();

                JsonElement jsonElement = jsonParser.parse(message);
                JsonArray jsonArray = jsonElement.getAsJsonArray();
                JsonObject jsonObject = jsonArray.get(1).getAsJsonObject();

                if(jsonObject.get("lang") != null &&
                        jsonObject.get("lang").getAsString().equals("en")){
                                Jedis jedis = super.getJedisInstance();
                                if(jedis != null){
                                        jedis.lpush(super.name, jsonObject.toString());
                                }
                }
        }
  }
```

Ingest using Sorted Sets

One of the concerns with the Pub/Sub method is that it can be unreliable because it is susceptible to connection loss. The challenge with the List-based solution is the problem of data duplication and tight coupling between producers and consumers. However, with Sorted Sets, both of these issues are addressed. A counter tracks the number of messages, and the messages are indexed against this message count. They are stored in a non-ephemeral state inside the Sorted Sets data structure, which is polled by consumer applications. The consumers check for new data and pull messages by running the ZRANGEBYSCORE command.

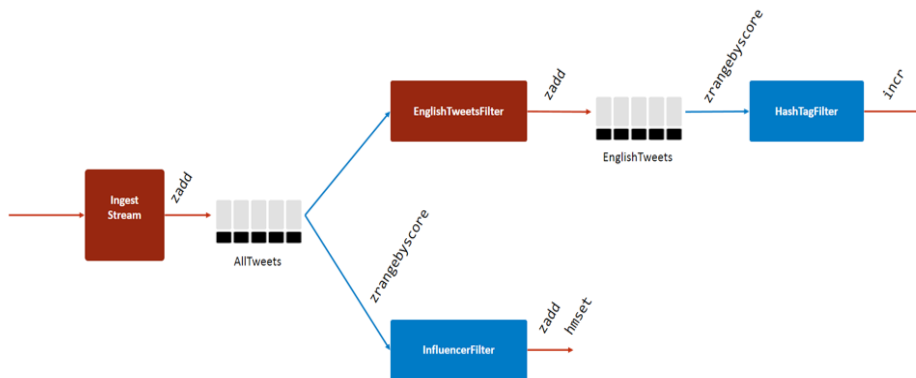

Figure 6. *Fast data ingest with Sorted Sets and Pub/Sub*

Unlike the previous two solutions, this one allows subscribers to retrieve historical data when needed, and consume it more than once. Only one copy of data is stored at each stage, making it ideal for situations where the consumer to producer ratio is very high. However, this approach is more complex and less cost effective when compared with the last two solutions

Pros

1. It can fetch historical data when needed, because retrieved data is not removed from the Sorted Set.
2. The solution is resilient to data connection losses, since producers and consumers require no connection between them.
3. Only one copy of data is stored at each stage, making it ideal for situations where the consumer to producer ratio is very high.

Cons

1. Implementing the solution is more complex.
2. More storage space is required, as data is not deleted from the database when consumed.

Code Design

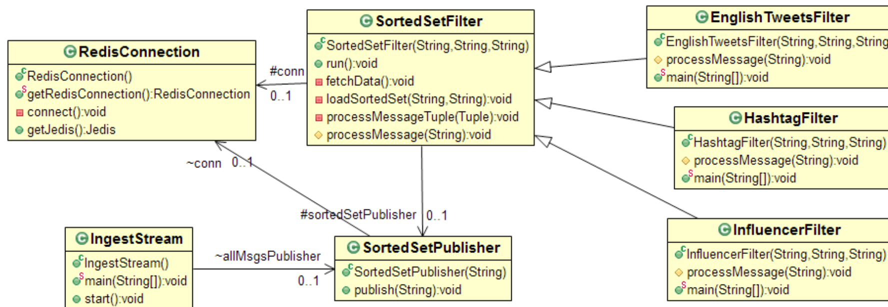

Figure 7. *Class diagram of Fast Data Ingest Solution with Sorted Sets*

Download the source code here: https://github.com/redisdemo/IngestSortedSet. The main classes are explained below:

1. SortedSetPublisher inserts a message into a Sorted Set and increments the counter that tracks new messages. In many practical cases the counter can be replaced by the timestamp.

```
public class SortedSetPublisher
{
      public static String SORTEDSET_COUNT_SUFFIX = "count";

      // Redis connection
      RedisConnection conn = null;

      // Jedis object
      Jedis jedis = null;

      // name of the Sorted Set data structure
      private String sortedSetName = null;

      /*
       * @param name: SortedSetPublisher constructor
       */
      public SortedSetPublisher(String name) throws Exception{
              sortedSetName = name;
              conn = RedisConnection.getRedisConnection();
              jedis = conn.getJedis();
      }
      /*
       */
```

```
	public void publish(String message) throws Exception{
		// Get count
		long count = jedis.incr(sortedSetName+":"+SORTEDSET_COUNT_SUFFIX);

		// Insert into sorted set
		jedis.zadd(sortedSetName, (double)count, message);
	}
}
```

2. SortedSetFilter class is a parent class that implements logic to learn about new messages, pull them from the database, filter them, and push them to the next level. Classes that implement custom filters extend this class and override the processMessage() method with a custom implementation.

```
public class SortedSetFilter extends Thread
{
	// RedisConnection to query the database
	protected RedisConnection conn = null;

	protected Jedis jedis = null;

	protected String name = "SortedSetSubscriber"; // default name

	protected String subscriberChannel = "defaultchannel"; //default name

	// Name of the Sorted Set
	protected String sortedSetName = null;

	// Channel (sorted set) to publish
	protected String publisherChannel = null;

	// The key of the last message processed
	protected String lastMsgKey = null;

	// The key of the latest message count
	protected String currentMsgKey = null;

	// Count to store the last message processed
	protected volatile String lastMsgCount = null;

	// Time-series publisher for the next level
	protected SortedSetPublisher SortedSetPublisher = null;

	public static String LAST_MESSAGE_COUNT_SUFFIX="lastmessage";

	/*
	 * @param name: name of the SortedSetFilter object
	 * @param subscriberChannel: name of the channel to listen to the
	 * availability of new messages
```

```
	 * @param publisherChannel: name of the channel to publish the availability of
	 * new messages
	 */
	public SortedSetFilter(String name, String subscriberChannel,
					String publisherChannel) throws Exception{
		this.name = name;
		this.subscriberChannel = subscriberChannel;
		this.sortedSetName = subscriberChannel;
		this.publisherChannel = publisherChannel;
		this.lastMsgKey = name+":"+LAST_MESSAGE_COUNT_SUFFIX;
		this.currentMsgKey =
		subscriberChannel+":"
		+SortedSetPublisher.SORTEDSET_COUNT_SUFFIX;
	}

	@Override
	public void run(){
		try{

			// Connection for reading/writing to sorted sets
			conn = RedisConnection.getRedisConnection();
			jedis = conn.getJedis();

			if(publisherChannel != null){
				sortedSetPublisher =
					new SortedSetPublisher(publisherChannel);
			}

			// load delta data since last connection
			while(true){
				fetchData();
			}
		}catch(Exception e){
			e.printStackTrace();
	}
}

/*
 * init() method loads the count of the last message processed. It then loads
 * all messages since the last count.
 */
private void fetchData() throws Exception{
	if(lastMsgCount == null){
		lastMsgCount = jedis.get(lastMsgKey);
		if(lastMsgCount == null){
			lastMsgCount = "0";
		}
	}
```

```
        String currentCount = jedis.get(currentMsgKey);

        if(currentCount != null && Long.parseLong(currentCount) >
                                Long.parseLong(lastMsgCount)){
                loadSortedSet(lastMsgCount, currentCount);
        }else{
                Thread.sleep(1000); // sleep for a second if there's no
                                // data to fetch
        }
}


//Call to load the data from last Count to current Count
private void loadSortedSet(String lastMsgCount, String currentCount)
                                                        throws Exception{
        //Read from SortedSet
        Set<Tuple> CountTuple =
                jedis.zrangeByScoreWithScores(
                        sortedSetName, lastMsgCount, currentCount);

        for(Tuple t : CountTuple){
                processMessageTuple(t);
        }

}

// Override this method to customize the filters
private void processMessageTuple(Tuple t) throws Exception{
        long score = new Double(t.getScore()).longValue();
        String message = t.getElement();
        lastMsgCount = (new Long(score)).toString();
        processMessage(message);

        jedis.set(lastMsgKey, lastMsgCount);
}

        protected void processMessage(String message) throws Exception{
                //Override this method
        }
}
```

3. EnglishTweetsFilter is a custom filter that extends SortedSetFilter with its own custom filter to select only tweets that are marked as English.

```
public class EnglishTweetsFilter extends SortedSetFilter
{
      /*
       * @param name: name of the SortedSetFilter object
       * @param subscriberChannel: name of the channel to listen to the
       * availability of new messages
       * @param publisherChannel: name of the channel to publish the availability
       * of new messages
       */
      public EnglishTweetsFilter(String name, String subscriberChannel,
                             String publisherChannel) throws Exception{
              super(name, subscriberChannel, publisherChannel);
      }

      @Override
      protected void processMessage(String message) throws Exception{

              //Filter; add them to a new time series database and publish
              JsonParser jsonParser = new JsonParser();

              JsonElement jsonElement = jsonParser.parse(message);
              JsonObject jsonObject = jsonElement.getAsJsonObject();

              if(jsonObject.get("lang") != null &&
                   jsonObject.get("lang").getAsString().equals("en")){
                     System.out.println(jsonObject.get("text").getAsString());
                     if(sortedSetPublisher != null){
                             sortedSetPublisher.publish(jsonObject.toString());
                     }
              }
      }

      /*
       * Main method to start EnglishTweetsFilter
       */
      public static void main(String[] args) throws Exception{
              EnglishTweetsFilter englishFilter =
                     new EnglishTweetsFilter(
                             "EnglishFilter", "alldata", "englishtweets");
              englishFilter.start();
      }
```

Tips for Performance Improvement and Scaling

4. **Pipelining publish or push commands**
   A Redis command from a client to the server follows the request/response model of the TCP/IP protocol, which means that the client waits for the server's response for every command before proceeding to the next command. Pipelining allows a client program to run multiple commands without waiting for the response. With this technique, a data ingest solution can publish or push multiple messages at once to reduce the number of roundtrips between the client and the server.
5. **Polling versus asynchronous notification**
   The first two examples in this paper rely on asynchronous communication between the producers and the consumers. However, depending on the data flow scenario (for example, cases involving a continuous flow of data), the consumers may be designed to pull a new set of data as they complete the processing of their current set. For example, you could replace a BRPOP call with multiple RPOP within a pipeline in the "Fast data ingest using Redis Lists" example.
6. **Failover for Pub/Sub**
   Redis Enterprise delivers high availability for Pub/Sub. In the event the master goes down, a slave takes over the job of being the message broker, thus preserving continuity to the service on the backend. However, the publishers and subscribers will temporarily lose their connection with the server. Many popular Redis client libraries offer a feature to automatically reconnect and reestablish connections.
7. **Redis Enterprise Flash**
   Redis Enterprise Flash technology enhances Redis to run on a combination of RAM and more cost-effective Flash memory. For large datasets, this is the most cost-optimal way to run Redis with the same sub-millisecond latencies and extremely high throughput. This would be ideal for fast data ingest using Sorted Sets as a time-series database. Since with this approach, the data isn't removed from the Sorted Set when consumed, the size of a database can easily grow into multiple gigabytes if not terabytes. With Redis Enterprise Flash, all of this data does not need to consume RAM, it can be stored in a combination of RAM and Flash memory.

Conclusion

When using Redis for fast data ingest, its data structures and pub/sub functionality offer numerous options for implementation. Each one has its advantages– a summary of the related pros and cons is below. This table also outlines which method is appropriate for which type of application:

Fast data ingest with	Pros	Cons	Ideal scenarios
Pub/Sub	• Easy to implement • Producers and consumers are decoupled	• Not resilient to connection loss • Requires many connections	e-commerce workflows, job and queue management, social media communication, gaming, collection of logs
Lists	• Easy to implement • Data is not lost when the subscriber loses connection	• Tight coupling of producers and consumers • Data is duplicated for each consumer, hence not ideal for some scenarios	financial transactions, gaming, social media, IoT, fraud detection
Sorted Sets	• Allows time-series queries • Efficient for cases where one client has numerous consumer	• Consumes more space • Complex to implement and maintain	IoT transactions, financial transactions, metering

700 E El Camino Real, Suite 250
Mountain View, CA 94040
(415) 930-9666
redis.com