Mason Nienaber(50246244), Radhit Dedania(46802584), Tarun Sai Ganesh Nerella(78929647)
6/11/2021
CS237 UCI

Effect of Caching in a Content-Based Pub/Sub System over Kafka

Introduction

Online food ordering services have been a tremendous success in the past decade. They have emerged as a strong alternative to the tradition of visiting restaurants for eating food. Now, the users can enjoy the same delicacies from the comfort of their homes without having to endure inconvenient traveling and long queues. Notification delivery is a significant part of food order management services. Customer satisfaction depends a lot on how quickly we are able to deliver notifications to them. Most popular architecture for notifications delivery is the pub/sub architecture because it decouples the synchronization between the Restaurants and its Customers in space, time and synchronization dimensions. In this project, we develop a Content-based Pub/Sub system over Kafka and analyze the benefits of various caching techniques.

Motivation

In general, any type of publish/subscribe system consists of four core components: Publishers, Subscribers, Broker system, and the Database. Publishers publish messages to topics and Subscribers consume messages from topics. In our case, Restaurants and Customers will act as our publishers and subscribers respectively. The broker system will be an intermediary between the above two components. The database is a persistent storage which is used for storing content released by publishers and information provided by subscribers. Storage is an essential component because it is always not possible to deliver the notifications right away (as soon as a message is received) to all of its subscribers. For example, the subscriber may be disconnected, or may not be polling even though it is active. In such cases and many others, we need to temporarily store those messages so that they can be sent to the subscribers later. However, a major downside of having a database is its Latency. Accessing messages from Database produces significant delay in delivering messages to the consumers, affecting the customer experience. This is more visible when a message has a lot of subscribers; there will be one database access for delivering the message to every subscriber. To solve this problem, we introduce caching to store messages at the broker. Since cache is of limited size, we develop strategies to decide which messages to store in the cache so that the average latency is reduced.

Objectives

Our primary objective is to test various caching policies and analyze how effective they are in comparison to the no cache system. Specifically, we will be analyzing four performance metrics, Latency, Cache hit rate, Duplication factor, and Throughput. Latency (end-to-end) is the time it takes from the moment a message is published by the publisher till the time the message is received by the subscribers (subscribed to it). In our system, we assume that all the consumers are always active and, therefore, end-to-end latency gives us a good measure of how much time the Broker takes to process (matching + loading/storing from/to the cache/database) the message. Cache hit rate is the number of messages that are found in the cache when a consumer tries to access them. Due to limitations of Kafka platform, which only allows appending messages to its logs and doesn't allow any modifications to them, we implement and measure cache hits per consumer. Duplication factor is a measure of how many subscribers are interested in a publication. While duplication across multiple consumer logs might appear redundant in our current implementation (again this is due to limitation of Kafka platform), we can use this metric in combination with Cache hits to compare and contrast the various caching policies we implemented. As an example, if a policy has high cache hit rate and high duplication factor, then it means that the policy is retaining messages having more number of subscribers in the cache and offloading messages having less subscribers to the Database, therefore indicating an efficient policy. Our last metric is throughput, where we measure the average number of messages processed by the broker.
We implement three different types of static caching policies: Threshold function based, Batch averaged, and Discounted threshold. In addition, we also have the Unlimited caching case, where we assume unlimited storage available at the broker. We compute the above mentioned metrics for all the four cases and compare with the no caching case.
As a secondary objective, we are building a content based pub/sub by giving customers extra flexibility on which messages to subscribe to. Subscribers can subscribe to a particular Restaurant, a particular food item, a particular discount, or any combination of these. However, Kafka has limited functionality in that it can only match topics. We address this shortcoming by using Kafka only as a message queuing system and off-loading most of the computation to a separate Subscription module which runs the matching engine and the cache policies.

Related Works

There exist many other communication paradigms differing from publish/subscribe. Each of these paradigms address some issues relating to decoupling, be it in *time*, *space*, or *synchronization*. Some simple messaging paradigms include message passing, the backbone of distributed interaction, RPCs, which attempt to make remote invocations appear under the guise of local invocations. However, both of these couple producers and consumers in time, space, and synchronization. Notifications may be viewed as a limited form of publish/subscribe in which subscribers notify the publishers directly which distinguishes it from the publish/subscribe paradigm. Notifications provide synchronization decoupling, however this paradigm remains coupled in both time and space. In Distributed shared memory (DSM), the various hosts of a distributed system maintain some view of a common shared memory space. This model yields time and space decoupling but does not provide synchronization decoupling. Message Oriented

Middleware (MOM). also only provides time and space decoupling and does not provide synchronization decoupling.

An innovation of the aforementioned paradigm (i.e. notification systems) ultimately led to the development of contemporary publish/subscribe systems. In such publish/subscribe systems we achieve total decoupling of producers and consumers in all of time, space, and synchronization. This decoupling is desirable due to the inherent flexibility. Coupled systems are often rigid and lack many aspects needed in a dynamic distributed setting.There are six popular existing publish/subscribe systems that are described and analyzed: *SpiderCast*, *PolderCast*, *PADRES*, *Cluster-Based*, *Hermes*, and *Content-Based*.

There exists many message queuing frameworks like *ActiveMQ*, *RabbitMQ*, and *Kafka*. These systems were tested empirically and analyzed for performance in a publish/subscribe setting including a broker for the shared memory space. Two experiments were conducted including one producer and one subscriber. In both tests, Kafka proved significantly superior. For the producer test, Kafka yielded results orders of magnitude better than ActiveMQ and twice as efficient as RabbitMQ. This is likely due to Kafka's unique ability to batch many messages into one such that only one TCP/IP request is needed per batch. As well, Kafka does not wait for acknowledgements from the broker before sending the subsequent message allowing for maximum possible throughput. In the consumer test, Kafka was able to consume four times as many messages per second compared to ActiveMQ and RabbitMQ. One main reason for this is that the Kafka broker does not maintain delivery/consumer state of every message. Kafka only maintains state when the consumer explicitly commits its updated offset. Finally, Kafka also has a more efficient data storage format which mandates less bytes transmitted from broker to consumer.

Two caching variants have been implemented in [14]: utility-driven and TTL-based:
Utility values are assigned to objects(events) based on where they reside(cache/database) using a utility function. Objects located at the database have lower utility as compared to cached ones as users experience ease in accessing objects from cache due to lower latency. Utilities are maintained on a per subscriber basis. Each subscriber has a different utility value in comparison to others for the same object. Thus, the caching policy in this case boils down to caching objects such that the overall sum of utilities(objective function) across subscribers is maximized. Depending on the implementation of utility function, caching policies vary. Three utility functions have been defined in [14]: uniform cost, size-based cost and latency-based cost. In the uniform cost case, the objective function is to maximize no. of objects that get accessed from the cache(cache hit ratio). In the size-based cost case, the objective function narrows down to maximizing the data volume(amount of bytes) returned from the cache. This utility value varies proportionally to no. of subscribers of an object. This is analogous to the widely popular LFU(least frequently used policy). The last eviction policy based on latency-cost believes that utility value of an object can be computed using the latency of retrieving the object. In this case, the objective function minimizes the access latency across objects and keeps the ones which are more likely to get accessed and have a higher database fetch latency.

Lastly, TTL caching policy sets an expiration time on each object after which they get dropped irrespective of the cache state(full or partially full). This is a way to control flow of incoming objects in the cache. Unlike eviction-based policies, this guarantees that cache evictions would be rare as TTL will ensure that cahe will never reach its capacity as time progresses unless there is a sudden surge of objects that have to be cached. It has been observed from simulations that TTL performs the best in terms of cache hit ratio, average latency and data volume fetched from cache as compared to the eviction based policies. Amongst the eviction based policies, uniform cost fares better than the other two which have similar performance on the above three metrics.

System Architecture

Our system architecture will include four key components: broker, database, publisher/subscriber, and broker management module. The database is a persistent storage which will be used only for storing content released by publishers and information provided by subscribers. The database was implemented using MySQL. Restaurants and Customers will act as our publishers and subscribers respectively. The broker component will be an intermediary between the above two components. It will consist of three modules: cache management, subscriber management, and database management. The subscriber manager will store subscriber ID and subscribed content for internal cache operations. The cache management module will create and maintain separate caches per unique subscribed content from which multiple interested subscribers can access their content. The cache management will be responsible for matching arriving published content with subscriber interests. It will also populate the subscriber unit of cache with the published content if a match occurs as the data format of published and subscribed contents will be different Finally, the database management module handles the MySQL interfacing. The database module is responsible for receiving data that has not been selected to cache, and forwarding it to a persistent store. As well, the database module is responsible for retrieving data from the persistent storage whenever the consumers notice a missing offset. If any miss occurs at broker level, the request will be propagated to the database which will incur delay in response time. The secondary aim of this project is to come up with eviction policies for caches to minimize traffic that accesses the database. We will do a comparative analysis of various caching policies with the baseline no caching scenario to understand the individual performances with respect to incoming data. As well, we will include an analysis of the system with unlimited caching. Finally, we will implement three novel policies in between and analyze the various trade-offs of each. A high level system diagram is found below in figure 1 while a more detailed system diagram illustrating the data flow is present in figure 3.

Data Structures and Record Value Representation

There exists two fundamental data structures in our system. The publication and subscription

bitmap, as well as the forward and reverse hash tables. In this work, we use bitmap as a compressed representation of the content(value field) of the publication record. In the bitmap, we concatenate the values of restaurant name, food item, and discount into one binary value (see details in figure 2). This binary value remains constant length at five bits and has values based on the possible number of restaurants, food items, and discounts respectively. In our case, we have decided the number of restaurants to be two, the number of food items to be four, and the number of discounts to be three. Thus the bitmap will always be a five bit value with the MSB corresponding to the restaurant name, the second and third MSB corresponding to the food item(s), and the fourth and fifth MSB corresponding to discount(s). Next, we included hash tables in the Broker Management Module as a way to invoke our matching algorithm. The first was a hash-table which matches input publication bitmaps to their respective subscriptions. The second is a reverse hash-table which does the inverse, it matches the subscriptions to their corresponding bitmaps. Both of these hash tables play a fundamental role in our matching.

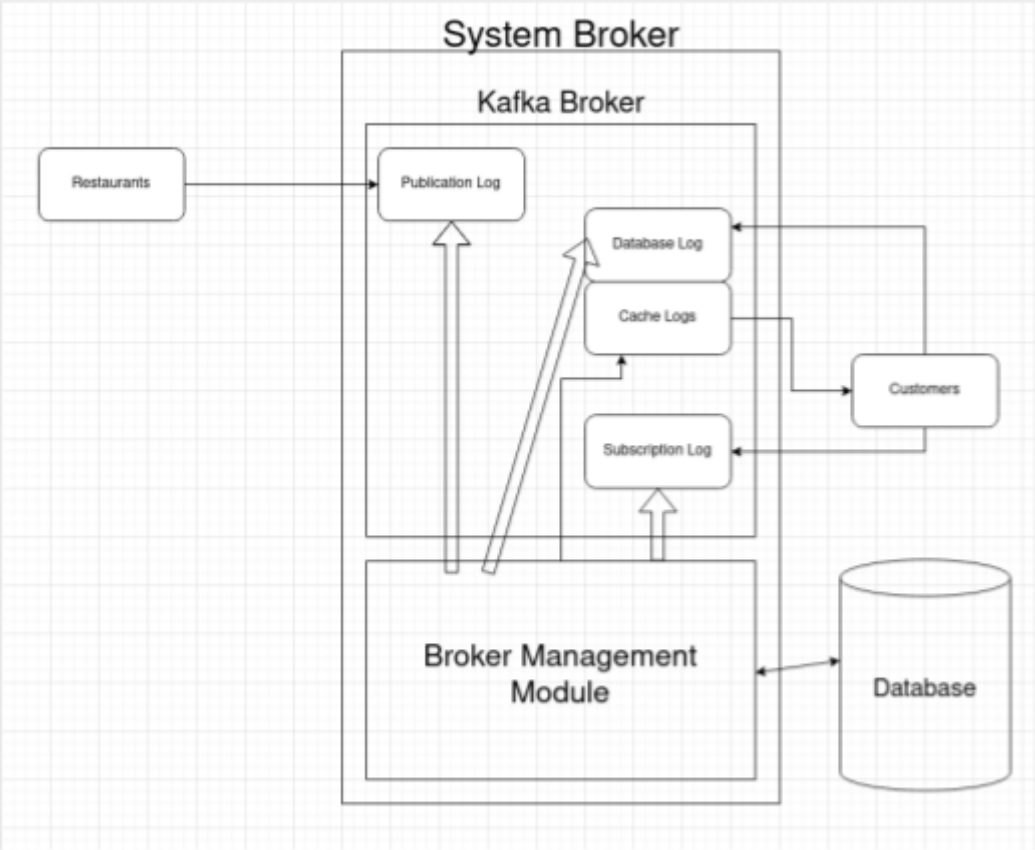

Figure 1: Block Diagram of System

Bitmap	Meaning
10001	R1,F0,D1
00110	R0,F1,D2

Figure 2: Bitmap and Representation Meaning

Figure 3: Detailed System Architecture Diagram

Bitmap(key)	Subscribers(value)
10001	S0,S2
00110	S1

Subscribers(key)	Bitmap(value)
S0	10001
S1	00110
S2	10001

Figure 4: Hash-table and reverse hash-table example

Data Flow

The data flow of the system will be described using the above diagram in Figure 2. The broker management module handles most of the processing of data in the system. The first instance of data flow is the polling of the publication log in which the broker management module obtains any publications from the restaurants. Next, the publications are fetched and hashed in a way such that publications can easily be matched to subscriptions. The data is then forwarded to the cache controller where we may select one of five caching policies to determine the next step. If the record is to be cached, then the cache controller module forwards the subscription to the respective consumer log. In the case that the caching function elects not to cache the record, it is forwarded to the MySQL database and stored in the form of key-value. In step five, the consumers continuously pull any data to be fetched from their corresponding consumer log and forward any missing offset values to the shared database log. In this log, values that were not elected to be cached are stored where they are polled by the broker management module to be ultimately retrieved from the database. Next, the poller forwards the retrieved offsets to the database connector. From here, the database connectors interface with the database to obtain the necessary key-value pair(s) in steps nine and ten. Next, the retrieved database information is forwarded to the correct consumer log using the subscriber ID. Finally, to ensure that an infinite loop does not result due to the database offset not being the expected offset, the database log is only utilized if the unexpected offset is greater than the expected offset. Otherwise, this record is one that has already been retrieved from the database.

Caching Techniques

As mentioned, our system provides five base caching functions with various parameters available for tuning and optimization. There exists two trivial caching functions used as edge cases for testing. These are: NO caching, and unlimited caching. We created these policies to test the performance of our system in either respective case. These two cases provide useful benchmarks to compare our remaining three policies to. Each of our policies should provide some metrics in between the aforementioned edge cases.

The first novel policy employed is called the *Threshold* policy. This policy has two tuning parameters: *lambda, mu*. The policy caches the record if the existing threshold (initialized to ZERO) is less than or equal to the input record's number of subscribers. The policy then updates the threshold for the next record to the sum of the current threshold and the product of *lambda* and the input number of subscribers. Conversely, if the number of subscribers is less than the current threshold, this record will be sent to a persistent store in the database. The threshold function is then updated to the difference between the existing threshold and the product of parameter *mu* and the input number of subscribers. An example of this policy can be seen below in figure 3. The motivation for the design of this policy is as follows. For a certain threshold, if the current input record is not cached due to its subscriber count being less than it, then maybe the current threshold is comparatively high and hence, should be brought down (threshold - *mu**(current subscriber count)) before the next record is processed based on subscriber count. Conversely, if the current subscriber count was greater than the threshold, we would add the current input record to the cache but maybe that happened because the threshold is comparatively low and hence, we should increase it (threshold + *lambda**(current subscriber count)) before a new input record comes in for a decision.

The second novel policy implemented is called the *Mean* policy. This policy has one tuning parameter, the batch size. Based on the batch size, this policy samples *n* most recent records up until the number of records sampled equals the batch size parameter and then averages subscriber count values corresponding to all those input records in that batch. That average value is then set as the new threshold value for the next batch size. Similarly, this threshold value is used in the same way as the first novel policy whereas if the threshold is less than or equal to the number of subscribers of the incoming record, it will be cached. Otherwise, it is passed to a persistent store in the database. The motivation for this design lies in the fact that a workload stabilises within a certain time after initial setup. If dynamic addition/removal of publishers & consumers is allowed, there would be many transition phases in the observed attribute of interest (no. of subscribers) of the published record. In such cases, it is highly inaccurate to rely on a single value for threshold computation as it may introduce unknown variance during those transitory phases. To mitigate this problem, a batch of inputs (instead of a single one) can be considered for threshold updation which would nullify this variance by averaging out the observed attribute over the batch. This policy is still relevant for cases when no dynamic addition/removal of entities is allowed as using the average of a batch for some operation (here, threshold updation) is always more reliable than a single input.

Finally, the third novel policy is called the *Buffer* policy. This policy has two turning parameters: *alpha, gamma*. This policy involves buffering the previous record's value and using that buffered value to update the new threshold. The new threshold is updated at each step (i.e. batch size of 1) but uses both the passed value, and buffered value to update. The updated threshold value is computed by the sum of two products. The first product is the parameter *gamma* and the subscriber count of the current input. The second product is the parameter *alpha* and the buffered value (subscriber count for the previous input; just before the current one). The motivation for this approach comes from the field of Reinforcement Learning. In case of charting out a motion plan for a single actor system (the motion plan is expressed as a finite state machine where each state holds a reward value), that transition edge is selected which leads to a state (say, s1) from where the states with higher reward values can be accessed though s1 might have a lower reward value. The overall objective is to select a sequence of states such that total reward obtained on the motion path is optimized. Hence, it becomes necessary to consider reward values of subsequent states too as there might be an optimal path from s1(current state) to other states that has highest overall reward value as compared to other paths though s1 might have the lowest reward value in the system. Similarly, in our case, we believe that inputs are not totally independent and hence, taking into account the past behaviour can lead to a stable and more accurate threshold value as the current input may be an outlier in which case it will disrupt the threshold computation.

Figure 5: Example of *Threshold* Policy

Figure 6: Batched Mean policy

Figure 7: Buffered Policy

Experiments

We conducted experiments relating to various metrics including: cache hit rate, duplication factor, end-to-end latency, and throughput. For each experiment we used a randomly generated list of input publication files ranging in size from 500 to 10000 records. Using these file sizes, and the aforementioned metrics, we generated four plots that can be seen below comparing caching policies.

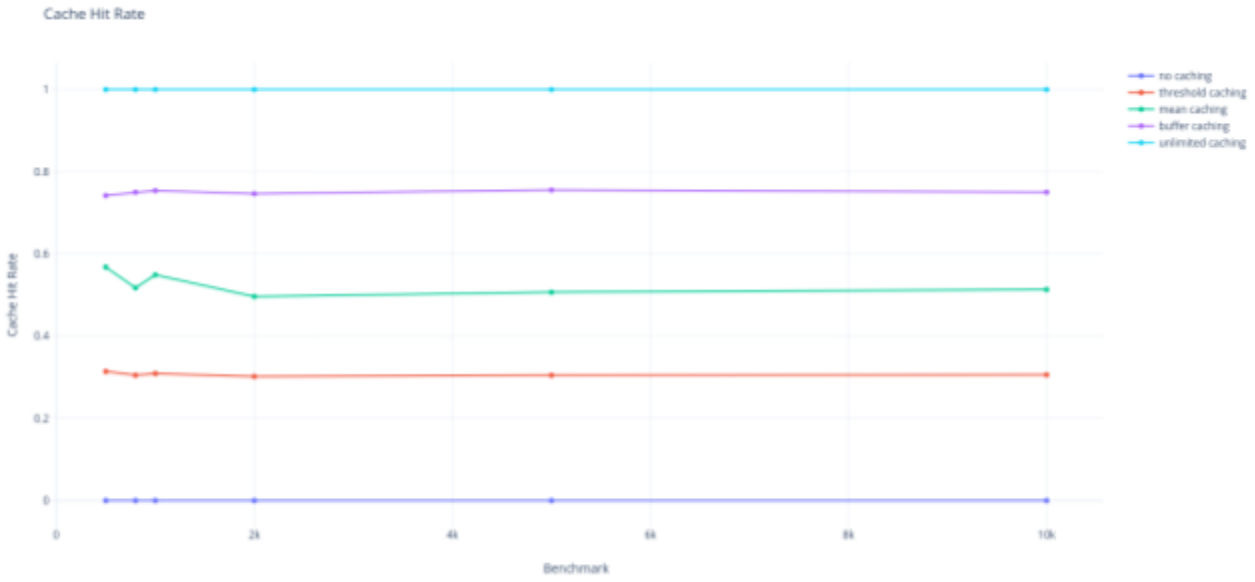

Figure 8: Cache Hit Rate by Policy

In the above figure, we obtain the cache hit rate for each of the five tested policies. As expected, the cache hit rate is best when the policy is set to unlimited caching. However, the best novel policy is unanimously buffer caching. This policy maintains an average cache hit rate around 0.8 for all six benchmark values.

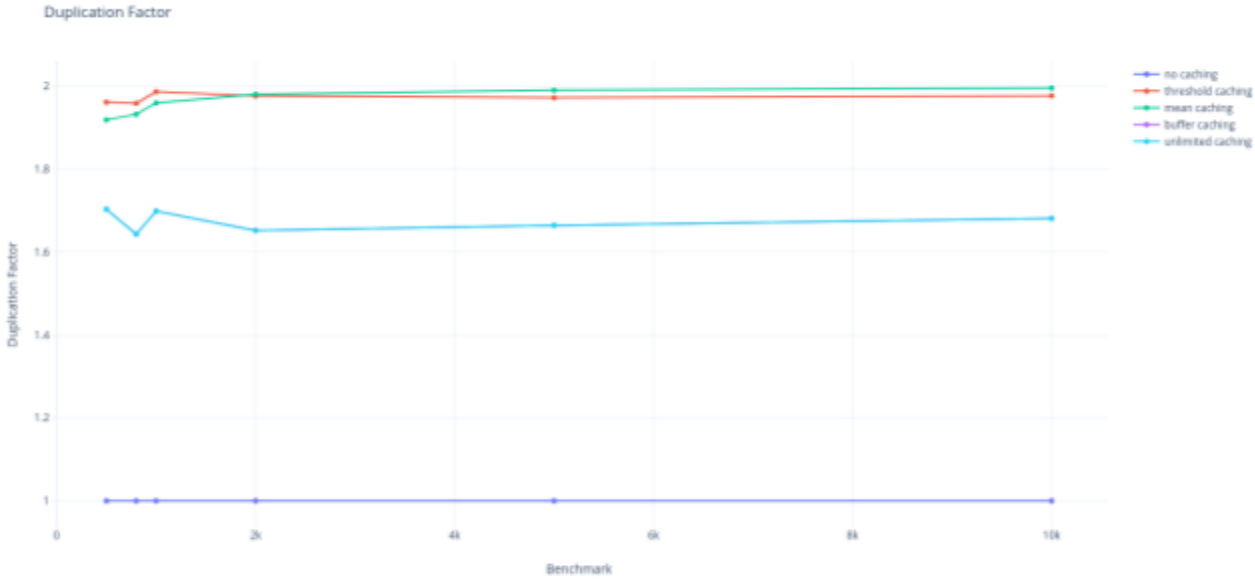

Figure 9: Duplication Factor by Policy

In the above figure, we empirically test the duplication factor for each of the five policies. Notably, two policies are about equivalent when strictly analyzing duplication factors. These two policies are both novel policies including: mean caching and threshold caching. Both of these novel policies have duplication factors very close to the maximum of 2.0.

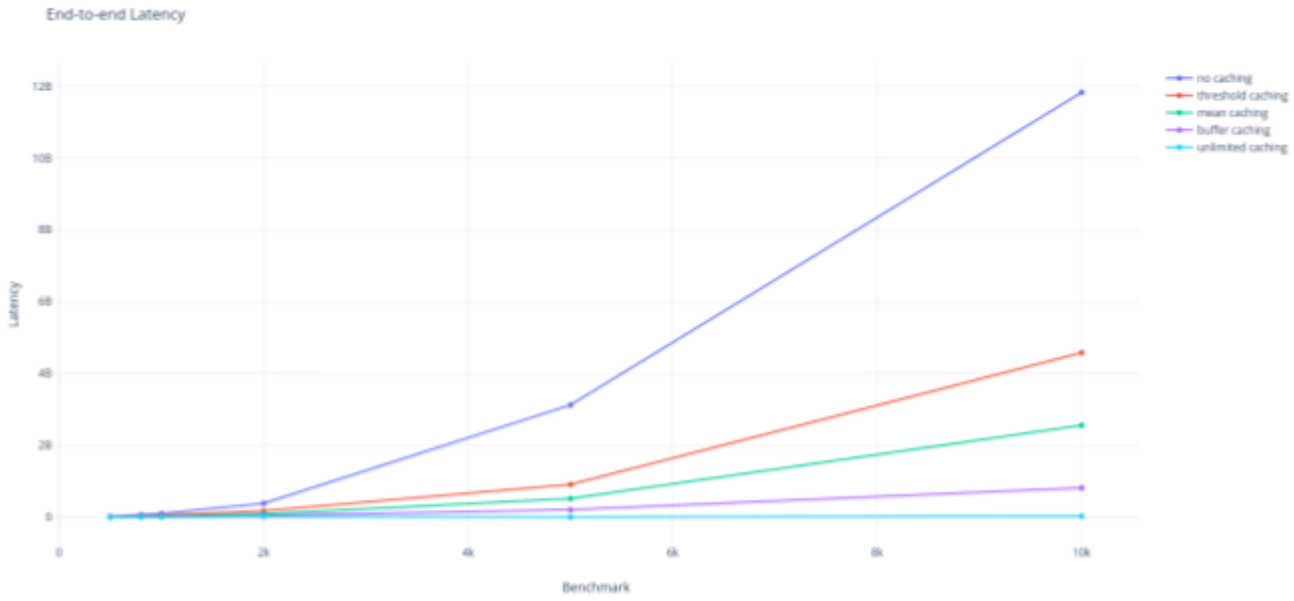

Figure 10: End-To-End Latency by Policy

The above figure contains empirical testing results of end-to-end latency for each of the five policies. As expected, no caching results in the highest end-to-end latency and this effect gets much more palpable as the number of input records increases. In our experiments, when the input record size was 10,000 the latency was immediately obvious as the difference between no caching and unlimited caching was obvious.

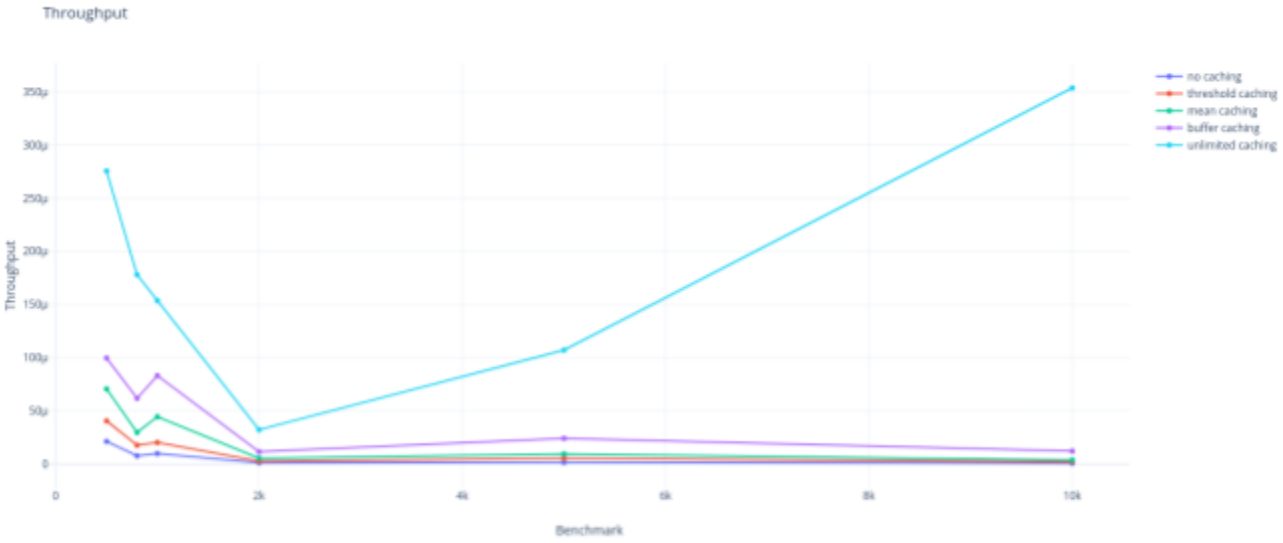

Figure 11: Throughput by Policy

The last of the four experiments comparing policies is shown above. The throughput metric is obtained by inverting the end-to-end latency and accounting for the number of input records. As expected, when the policy is configured to unlimited caching the throughput reaches a

maximum and far supersedes any other policy's throughput. Similar to latency, this effect is most obvious as the number of records increases.

As well, we conducted three more experiments, one per novel caching policy. In each of these experiments we attempted to tune the respective parameters within the cache policy to achieve the optimal parameterization of each policy. These policies were tested on the randomly generated publication input file of size 1000. The metric of interest (i.e. the y-axis in the below graphs) is latency. The three plots are included below. Additionally, all seven plots will be discussed in detail in the following analysis section of this paper.

Figure 12: Threshold Policy Optimization by Parameters

The first of the three optimization experiments involved the *threshold* policy. In this experiment we empirically tested varying values of the tuning parameters. As illustrated by the above graph, the latency was minimized when the (gamma,alpha) tuple was (0.8,0.2) and (0.3,0.1). Thus, these values appear to be optimal for this specific policy implementation.

Figure 13: Mean Policy Optimization

The second policy, the *mean* policy was also optimized on its one parameter: batch size. The batch size varied between 20 and 500 and the end-to-end latency was compared. As illustrated in the above graph, the optimal batch size appears to be 500. Notably, the end-to-end latency consistently decreases as the batch size increases. However, this may not be an entirely accurate representation of the policy since this batch size has an inherent trade-off with cache hit rate.

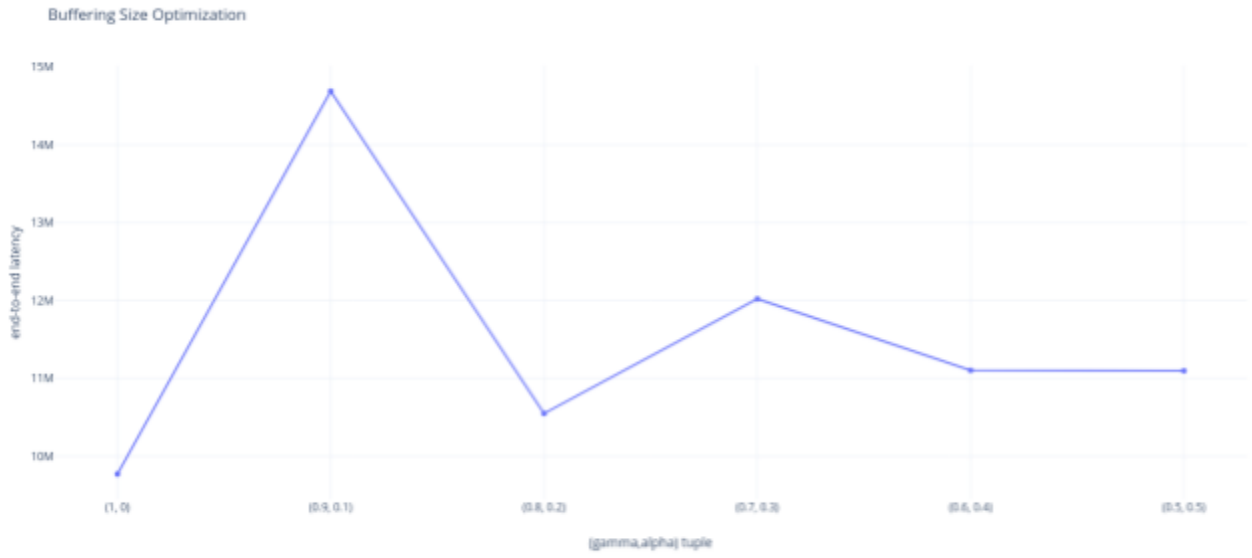

Figure 14: Buffering Size Policy Optimization

Finally, the last policy, the *buffering* policy, was also optimized on its two parameters, the (gamma,alpha) tuple. The end-to-end latency appears to reach a minimum when the tuple is

(1,0) and also reaches local minima around (0.8,0.2). In the optimal case (1,0) the policy does not account for the buffered value and instead only uses the input value.

Analysis

The first graph(Figure 8) plots the cache hit rate v/s benchmarks(varying count of publication records) for all the five caching policies. It can be observed from the plot that cache hit rate remains nearly constant for each individual policy across different benchmarks. Among the policies, no caching has a hit rate of 0 and unlimited caching has a hit rate of 1 as expected. Among our novel policies, threshold policy has a hit rate of about 0.3 whereas the metric for mean policy lies around 0.5 and for buffer policy it is close to 0.7. In our work, as the online cache functions are used which make decisions on the fly and stand by it in future, the hit rate is synonymous to the actual cache content as whatever is stored in a cache log is eventually going to be consumed by the corresponding subscriber because each consumer cache log only keep relevant records which have been filtered out by our broker depending on the received subscriptions. Thus, more hit rate means more cache content and hence, more cache size. Thus, a higher hit rate is beneficial in terms of latency but expensive in terms of memory consumption for storing records.

The second graph(Figure 9) plots the duplication factor(no. of replicated records in cache/cache_hits) v/s benchmarks(varying count of publication records) for all the five caching policies. An interesting pattern is observed. The duplication factor fluctuates for lower benchmarks values(till 2000) and then remains nearly constant throughout till 10000. In every testing benchmark, we have three consumers with two of them sharing a common subscription and the remaining one having subscribed to something else. Thus, duplication factor can range between 1(record kept only once) and 2(two copies of the common record kept in individual queues) with it being 1 if all the publication records have unique subscriptions or no caching happens though the records brought back from database will again face duplication but in this case duplication will be inevitable and won't be due to caching and it being 2 when all publication records correspond to the common subscription (have 2 subscribers in our case). Duplication factor of no caching is 1 by definition as nothing is kept in cache initially and everything will be retrieved from the database eventually but we count the duplicate records only during caching so it will be 1 in this case. It can be observed that duplication factor for unlimited caching is significantly lower than the remaining three policies even when its cache_hits = 1 because the no. of incoming publications that have shared subscribers constitute 70% of the total publication records and hence, duplication factor becomes (1*0.3 + 2*0.7)/1.0 = 1.7. For the other three policies that we designed, the duplication factor is high and nearly same though the cache hits are 0.3(threshold policy), 0.5(mean policy) and 0.7(buffer policy) respectively because the fraction of shared records cached increases and follows the order: buffer policy > mean policy > threshold policy. The fluctuation can be explained in terms of the fraction of shared publications in the testing benchmarks. If the fraction is high, the duplication factor is high and vice-versa.

For the third and fourth plot(Figure 10 & 11), the performance metric measured is average end-to-end latency (from publication till consumption including database access if the record was not cached) per record and average throughput(no. of records fetched per second) respectively. Average throughput is the inverse of average end-to-end latency and therefore, exactly opposite trend will be observed as compared to the trend in average latency graph. From the average latency plot, it can be seen that the following order holds: unlimited caching < buffer policy < mean policy < threshold policy < no caching. In case of unlimited caching, all records are cached so the latency is lowest and 100 times less than no caching case where it is highest because everything is stored at the database. For the remaining three policies, we mentioned while discussing the results of cache hit rate in the first paragraph that cache hit rate is synonymous with the actual content cached. Thus, more cache hit rate means more content being stored in cache and less latency for retrieval of those records. As threshold policy had hit rate 0.3 , mean policy had hit rate 0.5 and buffer policy had hit rate 0.7, the average latency is lower for buffer policy than mean policy which has even lower latency than threshold policy. The trend in throughput is the reverse(as explained above) with the following order: unlimited caching > buffer policy > mean policy > threshold policy > no caching. A peculiar feature about plot for average threshold is that it is a V-shaped curve with the lowest occurring at benchmark 1000 because the fraction of shared publication records varies per benchmark. The fraction was the lowest for 2000 and hence, the duplication factor also dipped for that benchmark as compared to its neighbours 1000 and 5000.
We now try to optimize each of the policies individually by varying the policies parameters and find out the optimal values which give the best end-to-end latency values.

For the Threshold policy (Fig 12) , we obtained the latencies for seven tuples of (mu, lambda) = {(0.2, 0.3), (0.1, 0.3), (0.5, 0.5), (0.3, 0.1), (0.3, 0.2), (0.2, 0.8), (0.8, 0.2)}. The lowest end-to-end latency was observed for mu = 0.8, lambda = 0.2 and the highest latency was observed for mu = 0.8, lambda = 0.8. These results match our expectations because, lambda controls the fraction of the current value that gets added to the threshold when it is a cache hit. Therefore, the higher the lambda is, the higher the threshold will be incremented for cache hits. This in turn means that the number of cache hits decreases and more blocks get sent to the database and hence higher the average latency. Similarly, mu controls the fraction of the current value that gets decremented from the threshold when there is a cache miss. Larger mu means that threshold will be lowered by a lot whenever there is a cache miss. This inturn increases the number of cache hits and more blocks will be stored in the cache leading to a lower average latency. Therefore, to get the best performance out of this policy, mu should be kept high and lambda should be kept low.

For the Mean Policy, we obtained the latencies for various batches starting from 20, 30, 50 upto 500. The results showed that latency monotonically decreased with increase in batch size. The reason why the latency is low for high batch sizes is very clear. In the policy, the threshold is initially set to 0 and then after every batch_size number of iterations, the threshold is set to the average value of the previous batch_size elements. Therefore, if we have batch_size of 500, the threshold remains at 0 until we reach the 500th iteration, the halfway point in our 1000 sized benchmark. So the first 500 elements are always cached. The remaining 500 are ethen decided

based on the average value of the previous 500 elements. So therefore, although it appears from the results that having a higher batch_size is better, it isn't the best choice and we should look at the dataset size also before deciding an optimal batch_size.

For the Buffering policy, we obtained the altencies for various combinations of (gamma, alpha) = {(1, 0), (0.9, 0.1), (0.8, 0.2), (0.7, 0.3), (0.6, 0.4), (0.5, 0.5)}. From the graph we can observe that the latency is lowest for gamma = 1, alpha = 0, but however, there is not much difference between the various combinations. We now discuss some boundary cases. Gamma determines the fraction of the current value that contributes to the threshold, and alpha determines the fraction of the previous value that contributes to the threshold. We always choose gamma, alpha such that gamma + alpha = 1. Now if gamma = 1, alpha = 0, that means that the threshold doesn't depend on the previous value and is always set to the current value. Therefore, the block will always be a cache hit and we get a very low latency. On the other hand, if we have gamma = 0.5, alpha = 0.5, then it means that both current value and previous value contribute equally to the threshold, a mean of the two. This case is similar to the batch mean policy with a batch size of 2. Overall, there isn't much optimization over the parameters of this policy that we can do because latencies don't vary much, however in comparison to other policies, this policy performs the best.

Comparing the three novel policies in terms of access latency, hit rate & fraction of shared publications cached, we notice that order for average latency as well as hit rate is buffer policy < mean policy < threshold policy while for fraction of shared publications cached (cache hit rate * duplication factor) the value is almost equal for the three. A good policy can be defined as one having less but more valuable cache content. Here, the value is the reuse amongst subscribers. Therefore, whichever policy has less cached fraction of total records but more duplication factor is better as more duplication factor implies more subscriber sharing for the cached content. Threshold policy keeps 0.3 fraction of total records but the subscriber reuse is comparable to the other two so it is the best policy according to our opinion. Mean policy comes next with cached fraction of 0.5 and buffer policy is last with cached fraction of 0.7 (shared cache content is almost identical for all three). Though buffer policy has a lower latency as compared to other two, due to memory constraints it seems unfeasible. Note that priority is given to respect for memory constraints & valuable content over latency for comparing the above three policies.

Discussion & Comments

During our system architecture design phase, various conflicting designs for our system meeting the required objectives were explored and their benefits and shortcomings weighed before narrowing down on one for implementation. Design space was not explored for scalability with respect to the publishers as that was guaranteed by the use of a single common publication log across all publishers which can also allow dynamic addition or removal of a publisher from the network without any extra overhead. Design for the matching engine was also straightforward with the use of hash tables and it did not have much scope for exploration. Albeit, the consumer side had a wide variety of potential designs. Some of which were scalable with consumers while

others were not but had their own pros. Designs were investigated for providing scalability with respect to consumers within certain limitations of bandwidth and quality of service metrics. One such design that was considered was using a single cache log across all consumers. This design would ensure that our system is scalable even with respect to consumers just as it is for publishers as all consumers will now fetch from the common log instead of their individual cache queues. The benefit of this design is that it eliminates the need for duplication of blocks (keeping multiple copies of the same record in different consumer logs) as the block can be fetched by all from the same common log. This would have highlighted the benefits of caching even more in the face of limited memory availability as it would have promoted block reuse across subscribers and the space that was initially consumed by these duplicated logs can be used to store more distinct blocks which would eventually decrease the average access latency. However, the drawback of this design is that it puts onus on the consumer to filter out blocks of interest as it would have to consume all of them in the common log irrespective of their relevance due to a limitation of Apache Kafka Consumer API which does not allow selective fetch from a topic(common log). Therefore, all consumers will retrieve all blocks from the common log thereby putting unnecessary load on network bandwidth.

Another design that we surveyed was of using a pointer or reference block in the consumer log instead of the original block containing subscribed content. The motivation of using pointers & references came from transportation of image content(huge size) over the network in a compressed format to reduce the usage of network bandwidth and hence, increase overall throughput. Pointers & references serve as an alias of the original block and serve a type of compression technique though not in literal sense. The original blocks can be stored at some special memory region pointed/referred to by the value of the pointer/reference block. The advantage of using this design is that initially consumers only have to fetch the pointer/reference blocks which are of small size and therefore, can be accessed much more easily without any significant delay. The consumers can then access the required content as and when needed by accessing the special memory region to retrieve it without having to fetch everything before it is actually required. Additionally, special memory region can be placed in a protected domain and hence, only authenticated consumers would be allowed to fetch the actual content thus enabling security aspects for this design. Regardless of these benefits, the double overhead of fetching data(once the pointer block & then the original content block) has to be borne by the consumer. This design will have performance issues in the longer run due to double fetch and authentication services which might make use of locks. Nevertheless, it is quite good for addressing memory issues in the broker.

We also checked the feasibility of the design that would notify consumers of the missing records in their cache log and would prompt them to directly fetch those from the external database. Though it would relieve broker from the burden of accessing database and the network connection from Kafka broker to consumers of excessive bandwidth, still the impact of this design on the quality of user experience will be drastic as the users will have to endure high latency while fetching content from the database and also incur additional access processing overhead which might be more suited for broker as it is a high-end machine than consumers who might be using some resource-constrained device like smartphones to access their

subscriptions. Therefore, we concluded that it would be better if our broker would access the database on their behalf as the access would be faster than consumers due to more availability of network resources and the user experience would remain intact as they will not feel the presence of a backend database even while retrieving the missing records as those will be placed in their cache log by the broker after fetching them from the database.

Finally, we chose to proceed with small separate cache logs per consumer so that unnecessary blocks are not fetched as opposed to the shared cache log and hence, the network bandwidth remains available for use by multiple simultaneous consumers. We would have preferred going with either a shared log of original records or record of pointers/references for the sake of scalability but Apache kafka does not support selective offset fetch and retrieval from a special memory region respectively. Hence, we were left with no choice apart from this one. The shortcoming of this design is that duplication of records takes up extra memory space in cache logs which could instead have been used for storing more distinct records and puts a processing cost on the broker to write the duplicated records to individual queues of interested consumers. The benefit is that user experience is not affected and it also provides better isolation between different consumers as each of them has a separate queue. Authorization service can also be built on top of this design.

As far as the platform is concerned, we also explored Apache Flink and KSQL towards the end of our implementation to see if either of them is more compatible to our preferred design than Apache Kafka. Apache Flink is a cluster computing & data processing platform like Hadoop that can also be used for processing streams also. It also allows importing from and exporting data to external sources like databases in the same way as Kafka. The benefit of using Apache Flink over Kafka is that the throughput & end-to-end latency measures are far better than than obtained when using Kafka but still it will not help us with our preferred design of a shared cache log as there is no inherent random-access storing mechanism supported by it which was the case with Kafka also. Kafka is better in that it at least provides append-only logs for persistent storage. But, Apache Flink mostly operates on files and uses external MQs like RabbitMQ for providing the same functionality as a Kafka topic log which is an additional overhead on the application developer in terms of integration with Flink. On the other hand, KSQL is a streaming SQL engine for Apache Kafka. As we are doing processing outside Kafka Streams, even KSQL would be irrelevant for our purpose. We can shift our processing to an inherent Kafka Streams API in which case KSQL would be better as the code can be written using SQL-like declarative language as opposed to Java. But, it still does not address our shared log issue. Perhaps, we might have to look for frameworks having their own random access stores in order to resolve this issue and then implement our preferred design on top of it to guarantee consumer scalability.

Future Work

Although we designed and implemented a fully functional content-based publish-subscribe system with caching at broker, there are many potential extensions to it. We created a

publish-subscribe system that runs with a fixed number of publishers and consumers. There are no dynamic additions and leaves during runtime. Initially, we planned to allow new publisher additions or leaves using a registry log wherein each incoming/outgoing publisher would write a message to notify our broker management module about the same but due to lack of time we could not implement and test it. Secondly, the content of subscription and publication can be more rich. Currently, we are only allowing consumers to specify their interest in a particular attribute by selecting a value from a small set of possible values corresponding to each attribute. We can remove this constraint by allowing the user to choose any value for an attribute in their subscriptions, maintaining a unique identifier for each distinct value observed in the subscriptions and modifying the key-value records of the hash tables(used in matching engine) on the fly. Another expansion of our work could be in the area of caching policies. In this work, we only used online threshold caching policies which make decisions on the go and do not reverse it in future. It was supposed to be compatible with the append-only storage logs provided by Kafka. It does not take into account the memory constraints in terms of a limited cache size. A future aspect of research would be to introduce some notion of size constraints on top of these policies to accurately cater to needs of a limited cache store. In our present system, the subscriptions once set by a subscriber cannot be changed over due course of time. Therefore, another future direction of work could be to allow subscribers to modify their subscriptions over time to allow more user control and flexibility as it happens in a real life publish/subscribe system. All the extensions listed above do not constitute an exhaustive list of achievable add-ons. There might be many more that we might have missed here but could play a critical role in improving the quality of service of the overall system.

Conclusion

In this project work, we implemented a content based publish-subscribe system over Apache Kafka platform to deliver notifications about restaurants, food items, discounts, etc. to interested subscribers. Our primary objective was to compare the performance of caching policies in presence of a limited storage at Kafka broker and an unlimited storage at an external database. The metrics used for measuring performance consisted of latency, throughput, cache hit rate, duplication factor, etc. The Dramatic performance gap (100x) between unlimited caching and no caching in terms of average end-to-end latency for various benchmarks justified our hypothesis of benefits of broker caching in a publish/subscribe system. It was observed from various experiments that threshold policy performed better than mean policy which in turn was more effective than buffer policy in terms of fraction of content cached(should be low to respect memory constraints) & its sharing value across subscribers(should be high to ensure reasonable reuse). We let the consumers express their interest using content-based subscriptions and restaurants publish events containing information about food items. We decoupled the subscribers from publishers through a broker middleware which is built on top of the Kafka Broker. Our system is scalable with an increase in publishers as all of them publish in parallel to the same log. However, it does not extend this flexibility to consumers as a distinct small cache log is maintained on a per consumer basis and hence, a surge in consumers leads to additional overhead involved in creating new cache logs and managing them. Kafka Broker is

mainly used as a persistence fault-tolerant storage platform for both subscriptions and publications(original and cached). Our broker takes care of polling, managing subscriptions, event matching, access to the database(on behalf of the consumer) and writing to cache logs, thus providing convenience to both producers and consumers as they have to write and read from a single location respectively. Another benefit of our system is its high availability. Publishers and consumers are not expected to be online always. They can write and read records respectively at any time of their choice. To conclude, our content-based publish-subscribe system built over Apache Kafka platform is highly available, scalable with publishers, enhances user experience(low data access latency) and fault-tolerant(due to intrinsic kafka replication) framework.

Github Repository (Source Code)

https://github.com/MNienaber2727/CS237_caching_pubsub

References

1) Sax M.J. (2018) Apache Kafka. In: Sakr S., Zomaya A. (eds) Encyclopedia of Big Data Technologies. Springer, Cham. Apache Kafka
2) Kreps, Jay, Neha Narkhede, and Jun Rao. "Kafka: A distributed messaging system for log processing.", Proceedings of the NetDB 2011
3) John, Liu. "A Survey of Distributed Message Broker Queues" University of Waterloo 2017
4) Eugster, Patrick Th, et al. "The Many Faces of Publish/Subscribe", ACM computing surveys 2003
5) Jacobsen, Hans-Arno, et al. "The PADRES publish/subscribe system.", 2006
6) Kiani, Knappmeyer, et al. "Effect of Caching in a Broker Based Context Provisioning System." EuroSSC 2010.
7) Chockler, Gregory, et al. "Spidercast: a scalable interest-aware overlay for topic-based pub/sub communication", DEBS 2007
8) Setty, Vinay, et al. "Poldercast: Fast, robust, and scalable architecture for P2P topic-based pub/sub" ACM International Conference on Distributed Systems Platforms and Open Distributed Processing 2012
9) Hojjat Jafarpour, Sharad Mehrotra and Nalini Venkatasubramanian. " A Fast and Robust Content-based Publish/Subscribe Architecture", IEEE NCA 2008
10) Carzangia, Rosenblum, et al. "Design of a Scalable Event Notification Service: Interface and Architecture", University of Colorado 1998
11) Adaya, Cooper, et al. "Thialfi: A Client Notification Service for Internet-Scale Applications" Google, inc.
12) Pietzuch, Bacon. "Hermes: A Distributed Event-Based Middleware Architecture", University of Cambridge 2002
13) Baldoni, Marchetti, et al. "Content-Based Publish-Subscribe over Structured Overlay Networks", Sapienza University of Rome 2005

14) M. Y. S. Uddin, and N. Venkatasubramanian. "Edge Caching for Enriched Notifications Delivery in Big Active Data", ICDCS 2018
15) Tim Berglund (2020) “Intro to Apache Kafka: How Kafka Works.”https://www.confluent.io/blog/apache-kafka-intro-how-kafka-works/
16) “Kafka Connect” docs.confluent.io. https://docs.confluent.io/platform/current/connect/index.html
17) “Java JDBC mySQL connector” https://www.vogella.com/tutorials/MySQLJava/article.html
18) “ mySQL database” https://www.vogella.com/tutorials/MySQL/article.html
19) “Kafka with Java” https://developer.okta.com/blog/2019/11/19/java-kafka