Edge Caching for Enriched Notifications Delivery in Big Active Data

Md Yusuf Sarwar Uddin
Department of Computer Science
University of California, Irvine
CA, USA
Email: msarwaru@uci.edu

Nalini Venkatasubramanian
Department of Computer Science
University of California, Irvine
CA, USA
Email: nalini@uci.edu

***Abstract*—In this paper, we propose a set of caching strategies for big active data (BAD) systems. BAD is a data management paradigm that allows ingestion of massive amount of data from heterogeneous sources, such as sensor data, social networks, web and crowdsourced data in a large data cluster consisting of many computing and storage nodes, and enables a very large number of end users to subscribe to those data items through declarative subscriptions. A set of distributed broker nodes connect these end users to the backend data cluster, manage their subscriptions and deliver the subscription results to the end users. Unlike the most traditional publish-subscribe systems that match subscriptions against a single stream of publications to generate notifications, BAD can match subscriptions across multiple publications (by leveraging storage in the backend) and thus can enrich notifications with a rich set of diverse contents. As the matched results are delivered to the end users through the brokers, the broker node caches the results for a while so that the subscribers can retrieve them with reduced latency. Interesting research questions arise in this context so as to determine which result objects to cache or drop when the cache becomes full (eviction-based caching) or to admit objects with an explicit expiration time indicating how much time they should reside in the cache (TTL based caching). To this end, we propose a set of caching strategies for the brokers and show that the schemes achieve varying degree of efficiency in terms of notification delivery in the BAD system. We evaluate our schemes via a prototype implementation and through detailed simulation studies.**

***Keywords*-Big active data; bigdata publish-subscribe system; caching policies; TTL caching;**

I. Introduction

Messaging and information interchange are at the heart of next generation societal scale applications where large numbers of users access vast amounts of information that is being continuously generated from diverse sources (social media, sensors, websites etc.); end users and applications seek to seamlessly obtain access to these large volumes of information using query interfaces based on application and context specific needs and interests. Traditional pub-sub systems address the problem of selective message forwarding based on subscriptions to message flows, and services like Twitter, and platforms like Storm [1] and Esper [2], have scaled message forwarding to large numbers of messages and users. Scaling data is addressed in the context of *Big Data* systems that support the ingest, storage, processing of vast amounts of static and streaming data. First-generation big data management efforts have resulted in various frameworks and languages (usually layered on Hadoop) for long-running data analytics and various key-value storage management technologies [3] that provide simple but high-performance record management and access. These developments have been "passive" in nature—queries, updates, and/or analysis tasks have been scaled to handle large volumes of data.

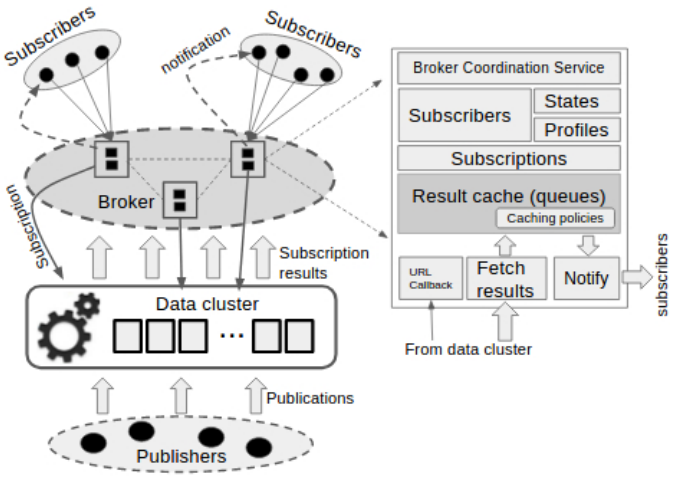


Fig. 1. Components of Big Active Data (BAD) systems.

Recent research efforts aim to move from current big data infrastructures, which largely support big *passive* data, to a new kind of big data infrastructure that provides a scalable foundation for big *active* data [4]–[6]. New techniques are required for continuously and reliably capturing big data collections continuously arising from social, mobile, web, and sensed data sources and to enable timely delivery of the right information to those users with indicated interests—a functionality that is typically enabled by publish-subscribe (pub-sub) systems. In traditional pub-sub systems, consumers (users) express interest in information through subscriptions; publications (possibly from other users) are matched against subscriptions and notifications containing information are sent to interested subscribers. In the Big Active Data (BAD) effort [7] being designed and developed at UC Irvine, we are

enabling a system that combines the capability of big data management technologies with the more traditional publish-subscribe functionalities.

Big Active Data systems position themselves distinctly in comparison to their namesake counterparts: namely pub-sub architectures, big data systems and active databases. In pub-sub, significant emphasis has been placed on ability to flexibly and easily express subscriptions (topic-based and content-based) through appropriate subscription languages. Broker based architectures have been developed to decentralize the storage of subscriptions and processing and routing of publications to subscribed users. Active database work (e.g., trigger processing [8]) and more recent work on stream processing systems with support for continuous queries can be adapted to handle subscriptions to data—but mostly for modest numbers of triggers or queries and only for limited query patterns. In contrast, Big Active Data platforms aim to deliver data of interest to a very large number of users in a timely fashion.

Fig. 1 gives a high-level overview of the physical platform and execution environment of the BAD system. At a high level, the BAD platform consists of data sources (Data Publishers) and end users (Data Subscribers). Additionally, BAD can ingest information from other external data sources and feeds, including a variety of sensors, various social data sources, news feeds, GIS information and so on. The data itself is stored and managed by a big data management system (BDMS) which runs in a data center with a cluster of hundreds (or thousands) of nodes and consists of many datasets. Users and user groups (e.g., various agencies or constituencies) are able to subscribe to data of interest declaratively, which can potentially *combine* data from *multiple* data sets both when describing their data interest criteria and when describing what they want to have delivered when conditions arise that match their interests. While the BDMS backend handles storage and processing of publications, the BAD broker network handles the task of notification management and distribution to subscribers scalably and efficiently.

Caching in Big Active Data System: As the number of end-subscribers and their interactions increase, the BAD broker network components and the BAD data cluster must incorporate new techniques to handle scalability issues. In this paper, we explore the role of broker caching as a method to support ruggedization of the broker and to enable real world scalability. Broker caching of notifications in our system has two direct benefits. First, subscriptions may be shared across subscribers and cached results can be returned from the broker reducing the overall latency for the subscribers. This is indeed the case when information needs are correlated based on interests or geographical context. Second, caching allows some degree of asynchrony among the subscribers in retrieving channel results. With caching, subscribers may remain offline when the channel results are ready to be delivered, and can receive them later on when they become online (if the cached results are not evicted by then). Additionally, given that results for matching subscriptions/channels are persistent at the big data backend, subscribers returning after a long hiatus can still retrieve notifications from the bigdata backend, albeit with increased latency. Such a persistence feature enables us to design and experiment with new storage/latency tradeoffs as systems scale while still ensuring availability of results.

In this paper, we propose, implement and evaluate two types of caching strategies for BAD systems: eviction-based and TTL (time to live)-based. In eviction-based caching, new result objects are cached on their arrival and less recently or frequently referenced objects are evicted when the cache becomes full. Different choices can exist to choose which object to drop, which we refer to as utility-driven choices. We propose a couple of dropping schemes, such as least recently used, least subscribed content, and least subscribers latency. On the other hand, TTL-based caching assigns an explicit expiration time on each cached object and evicts the object whenever the timer expires. The broker needs to choose this expiration time per object so as to keep the overall cache size bounded. We evaluated these caching strategies in different application scenarios with different workloads.

II. Related Work

Caching is a well studied concept in computing that has been studied at depth from many perspectives, including the Internet (refers to as web caching) [9], content delivery networks (CDN) [10], paging in operating systems, database buffer management, disk buffers, big-data applications [11], publish/subscribe systems [12]–[14] and information-centric networks (ICN) [15]. A comprehensive list of web caching policies is the topic of survey article by Podlipnig *et. al* [9]; more recent work [10] documents over 30 caching policies from 1993 to 2016. A vast majority of caching strategies are eviction-based; here an already cached object is evicted to admit a new object when the cache is full. Quite a few are TTL-based, such as [16], [17]. In admission based caching, such as [10], [18], incoming objects are admitted based on certain criteria (and then evicted or expired).

Eviction-based caching strategies differ in how they choose which object to drop; two popular metrics include recency (how recently has an object been accessed?) and frequency (how many times is an object requested?). Accordingly, LRU methods drop the least recent object whereas LFU methods drop the least frequent one. LFU is proven to offer the highest hit ratio [18], [19] in IRM (Independent Reference Model) load, where all requests are assumed to be independent of the past and request patterns do not change over time. LRU variants consider other metrics in addition to recency, such as object size (LRU-size), recency threshold (LRU-threshold), number of replacements (LRU-min), weighted recency (EXP1) and history of accesses (HLRU). LFU variants include LFU-aging (ages popularity over time), LFU-weighted (weighted counters), to name a few. Some policies combine both recency and frequency through a suitably chosen function, such as GD (greedy dual) [20], GDSF [21] and GD* [22]. Our proposed policies, such as LSC (least subscribed content), are variants of LFU. We also try a LRU-like approach.

TTL caches, in which eviction happens upon the expiration of a timer, are used in the Internet applications, such as Domain Name System (DNS) [16], [17]. The properties of TTL caches are formally studied in several works [23]–[25]; in contrast, LRU-based techniques are simple but hard to analyze. An interesting effort analyzes [23] the connection between TTL- and eviction-based caches using the notion of characteristics time [26]. The main question in TTL caches is to compute TTLs for objects—exact [16] and approximate [25], [27] answers have been derived. The paper [27] outlines several approaches to compute TTLs: average TTL (average over historical frequency), incremental TTL (optimized through gradient descent approach) and machine learned TTL (train a model to fit TTL).

III. The Big Active Data (BAD) System

Fig. 1 illustrates the different components within the BAD platform. BAD components provide two broad areas of functionality—bigdata management and monitoring, handled by the BAD data cluster, and notification management and distribution, handled by the BAD broker network. The BAD data cluster persistently stores all incoming publications (data items from publishers) and subscriptions (from subscribers), and perpetually executes background routines that match incoming publications with the stored subscriptions. We leverage Apache AsterixDB [28] as a foundation for backend BAD data cluster. AsterixDB has technical benefits including a rich declarative language (AQL) and a scalable distributed dataflow runtime with continuous data ingestion support. The BAD team [7] has enhanced AsterixDB with a feature called *channels*, which are instantiable versions of queries with parameters that execute perpetually in the data cluster [6].

The BAD data cluster does not directly interact with the subscribers itself; instead, a set of geo-distributed broker nodes are deployed that receive subscriptions from the end subscribers and pass them to the data cluster (in that, the brokers subscribe on their clients' behalf). The BAD broker network is managed by a Broker Coordination Service (BCS). When a new broker node joins the broker network, it registers through the BCS. A registered broker can accept clients for possible subscriptions to channels. The brokers are responsible for handling end user clients (called BAD clients), managing subscriptions to channels and delivering channel results against those subscriptions. Each broker has two parts: a "client-facing" part managing the clients and an "Asterix-facing" part handling interactions with the Asterix backend. The "Backend-facing" part uses appropriate set of REST calls (provided by the data cluster) to receive service from the data cluster whereas the client-facing part provides a set of REST calls to be used by the BAD clients to receive services from the broker.

While passing subscriptions to the data cluster, the broker registers a callback URL referred to itself (usually called a WebHook) that the data cluster invokes to notify the broker when results against that subscription is available (when a new publication matched the subscription). The actual content of the notification is implementation dependent which may contain the entire the result objects themselves and the results are immediately pushed to the broker (PUSH model) or may contain a resource handle (URL) to the result objects that can be retrieved by the broker later on (PULL model). Once subscription results are obtained by the broker, they are delivered to the corresponding subscribers. In this paper, we enhance the BAD platform with mechanisms for in-memory caching at the broker. Operationally, when a broker retrieves channel results that match subscriptions, the results are cached in the broker.

A. Subscriptions and Publication Matching in BAD

The BAD system performs 3 tasks: (a) managing and storing incoming publications as well as subscriptions (*data in*), (b) matching incoming publications (along with previously stored data if required) against the stored subscriptions (*data processing*), and (c) delivering matched publications, referred to as *results*, to the subscribers (*data out*). Managing and executing all these three components in a very scalable fashion is the ultimate challenge of BAD [6]. This paper addresses a subproblem related to caching subscription results in the broker that lies in *data out* component of the BAD system. For the sake of brevity, we outline the basic functionalities and abstractions that (a) and (b) components provide so that publish-subscribe paradigm can be supported on top of it. Scalable *data in* happens through inserting publication records into data tables with open or closed schema depending on whether the data fields and their types are apriori known or not. A SQL-like language helps with these insertions where the data records can be in JSON or XML or any other suitable format. The subscriptions are represented as *predicates* on the data fields of publications according to the interests of the subscribers. The BAD AsterixDB supports subscriptions through *parameterized channels* [6]. The *data processing* part corresponds to creating executable routines and executing them either asynchronously when required to do so or periodically with a given interval. These routines are responsible for matching publications against the subscriptions.

To execute channel functionalities, BAD provides two types of channels: *continuous* channels that generate matching results asynchronously whenever new data records match subscriptions and *repetitive* channels that get executed every so often at a given periodic interval. The construction of these channels and managing their runtime environment are the central piece of *data processing* engine in BAD, whose exact detail is beyond the scope of this paper. This paper relies on a certain abstraction of the backend data cluster in terms of subscriptions and getting the matched results back. This abstraction is as follows: the data cluster receives subscription requests (channel name and parameter values) and returns a unique subscription identifier. Each subscription does also specify a callback URL which is invoked by the data cluster each time a new result object (matching publications) is generated and are stored in a result dataset. The records in the result dataset contains the subscription identifier for which

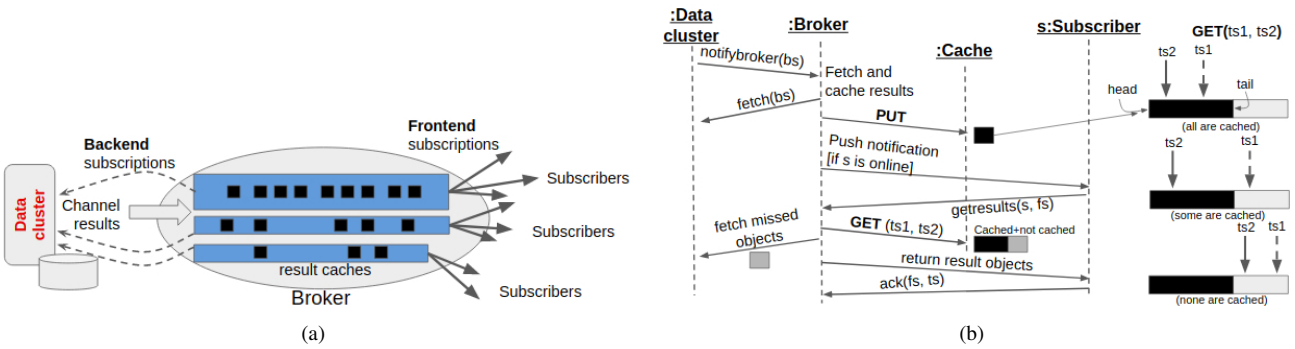


Fig. 2. Sequence diagram of subscriber retrieving channel results.

they are generated and are appropriately timestamped so that records can be retrieved in the order they were produced.

B. Distributed broker network

The central part of the *data out* component consists of a set of distributed broker nodes that connect the end subscribers to the data cluster. The broker receives subscriptions from the subscribers (the end users), stores them in a subscription table and forwards them to the data cluster. When notified, the broker pulls the new channel results from the data cluster and stores them in an in-memeory cache. A notification subsystem notifies the associated subscriber when a new result object is added to the cache. The cache grows and shrinks when new result objects are added and results are consumed by the subscribers. The various caching policies dictate how to manage these result objects, particularly to decide which object to evict when the cache is full or to determine TTL (time-to-live) values set on the result objects.

C. Delivery of Notifications in BAD

As the broker receives subscriptions from the subscribers, it does not necessarily forward them all to the data cluster. The broker suppresses subscriptions when multiple subscribers subscribe to the same channel with the same set of parameters (they are effectively asking for the same results but started at different times). In that, the broker makes only one subscription back to the data cluster and shares the channel results among the subscribers. These are called *backend* subscriptions. That means, a set of *frontend* subscriptions can be merged into a single *backend* subscription. For each of the backend subscriptions, the broker maintains an in-memory *result cache* to store matched results against that particular subscription.

Each result cache is a *sorted list* of objects ordered in the descending order of their timestamps as new objects are pushed at the head and old objects are deleted from the tail when needed. For each backend subscription, the broker keeps track of the *last* timestamp upto which the broker already fetched the results from the data cluster. Upon notifications, the broker fetches results after that marker, puts them in the corresponding cache and updates the timestamp accordingly. Similar kind of timestamp markers are maintained individually for all frontend subscriptions.

Fig. 2 shows the interactions among the data cluster, broker and subscribers as subscribers retrieve results for their subscriptions. As the broker puts fetched result objects in the cache, the cache may exceed the allowed limit (i.e., the cache becomes full). In that, the broker chooses a candidate cache (based on the dropping policy applied) and drops the tail object from that cache. After caching the results, the broker sends a push notification to all associated subscribers (if they are online) giving them a signal that new results are available to retrieve. The subscribers then make `getresults` call to retrieve the new results (or all the pending results since last retrieved). After retrieving the results, the subscriber sends an ACK to the broker confirming the consumption of results. The broker treats ACKs as cumulative in the sense that acknowledging the last object indicates that all objects before that are considered consumed.

In response to a `getresults` call, the broker first checks whether the requested objects are in the cache. Three cases can happen here: i) all objects are in the cache (the broker returns the cached objects), ii) some objects are in the cache and rests are not (the missing objects are retrieved from the data cluster and returned along with the cached ones), and iii) all objects are missed (all are retrieved from the data cluster).

Algorithm 1 shows different cache routines. GETRESULTS routine is invoked by a subscriber with a front subscription, which in turn calls GET routine of the associated result cache. This may invoke **fetch** routine that fetches results from the data cluster (in case the requested objects are not in the cache). The **fetch** call is specified by three parameters: the backend subscription for which the results will be fetched and the two end values of the requesting timestamp interval. The last boolean parameter specifies whether the right end of the interval is closed. The missed objects when retrieved from the data cluster are not cached again in the result cache because they may not be sharable by other subscribers any more.

IV. Caching Policies for Big Active Data

We take two base approaches of caching: utility-driven eviction and TTL-based expiration. While the utility-driven caching decides *which* object to evict when the cache is full, the TTL-based caching determines *when* to drop objects. In

Algorithm 1 Cache operations

function GETRESULTS(s, fs)
  bs = backend subscription associated with fs
  $cache$ = cache associated with bs
  bts, fts = latest timestamp for bs and fs
  return $cache$.GET(bts, fts)
end function
function PUT($cache, obj$)
  $obj.subs = cache.subs$
  $cache$.append(obj)
  while $cache$ is full **do**
    ch = choose-cache-to-drop-from()
    drop $ch.tail$
  end while
end function
function GET($cache, ts_1, ts_2$)
  $head = cache.head.ts$
  $tail = cache.tail.ts$
  if $tail \leq ts_1 \leq ts_2 \leq head$ **then**
    /* All requested objects are in the cache */
    $cached = \{o | o \in cache \wedge ts_1 \leq o.ts \leq ts_2\}$
    return $cached$
  else if $ts_1 < tail \wedge ts_2 \leq head$ **then**
    /* Some are in the cache and some are not */
    $missed =$ **fetch**($cache.bs, ts_1, tail, false$)
    $cached = \{o | o \in cache \wedge tail \leq o.ts \leq ts_2\}$
    return $missed + cached$
  else if $ts_2 < tail$ **then**
    /* All are missed */
    $missed =$ **fetch**($cache.bs, ts_1, ts_2, true$)
    return $missed$
  end if
end function
function ACK(s, fs, ts)
  $cache$ = cache associated with fs
  Update timestamp associated with fs
  $o.subs$.remove(s) $\forall o \in cache \wedge o.ts \leq ts$
end function
function SUBSCRIBE($s, channel, params$)
  Check if a backend subscription, bs, exists
  with the same $channel$ and $params$;
  if not create one and init a cache
  $cache$ = cache associated with bs
  $cache.subs$.add(s)
  Create frontend sub, fs, attached to bs and $cache$
  return fs
end function
function UNSUBSCRIBE(s, fs)
  Detach s from backend subscription of fs
  $cache$ = cache associated with fs
  $cache.subs$.remove(s)
  $o.subs$.remove(s) $\forall o \in cache$
end function

the following, we describe these two techniques in the context of big active data (BAD) systems.

A. Utility-driven Caching

As a basis for utility-driven caching, we argue that, in the eye of a subscriber the *utility* of retrieving of an object from the data cluster, when a cache miss occurs, has strictly *lower* utility than retrieving the same object from the cache. This is because retrieving result objects from the data cluster may take additional latency. The general objective of caching policies is to cache those objects so that the sum of utilities across all subscribers is maximized. We propose a set of utility-driven caching policies that differ the way they define the utility functions.

At any given time, let there be N result caches, one for each backend subscription, for K subscribers. Let $S(i)$ be the set of subscribers attached to result cache i and $O(i)$ be the list of objects in the cache in the descending order of their timestamps where $head$ and $tail$ denote the head (the newest object) and the tail object (the oldest object) of the list respectively. We denote any arbitrary object in a cache as o_{ij}. In the subsequent discussions, we use symbols, i, j and k to denote cache, object and subscriber respectively.

Every object maintains a set of subscribers to which the object is to be delivered. Let this set be $S(i,j)$ object o_{ij}. Note that $S(i,j)$ (subscribers attached to an object) may not be the same as $S(i)$ (subscribers attached to the cache). This is because a subscriber only receives result objects *after* its subscription. So, if a cache already had objects before a particular subscription is made, those objects would not contain this particular subscriber in their subscriber list, but the new results onward would contain the subscriber.

Let z_{ijk} be a binary indicator variable to denote whether object o_{ij} is to be delivered to subscriber k. Let $U_c(i,j,k)$ be the utility of object $o(i,j)$ with respect to subscriber k when the subscriber retrieves the object from the cache and $U_d(i,j,k)$ denote the same if the object is retrieved from the data cluster. The caching decision corresponds to find which objects to cache so as to:

$$\max \sum_i \sum_j \sum_k (x_{ij} z_{ijk} U_c(i,j,k) + (1 - x_{ij}) z_{ijk} U_d(i,j,k))$$

$$\text{s.t.}, \sum_i \sum_j x_{ij} s_{ij} \leq B$$

where s_{ij} is the size of object o_{ij}, B is the allowed cache size and x_{ij} is a binary decision variable indicating whether o_{ij} is cached or not. We can rearrange terms in the objective function and eliminate constant terms to have the following:

$$\sum_i \sum_j x_{ij} \sum_k z_{ijk} \left(U_c(i,j,k) - U_d(i,j,k)\right) \qquad (1)$$

Let us denote $\Delta(i,j,k) = U_c(i,j,k) - U_d(i,j,k)$ as the utility difference of an object when the object is retrieved from the cache vs from the data cluster. This is effectively the *value*

of caching of the object with respect to a particular subscriber. We, therefore, have the following:

$$\max \sum_i \sum_j x_{ij} \sum_k z_{ijk} \Delta_u(i,j,k)$$
$$s.t., \sum_i \sum_j x_{ij} s_{ij} \leq B$$

We can define the value of each individual object as: $\phi_{ij} = \sum_k z_{ijk} \Delta(i,j,k)$, which is the sum of values obtained by all subscribers attached to the object. Consequently, we have:

$$\max \sum_i \sum_j x_{ij} \phi_{ij}, \text{ s.t.,} \sum_i \sum_j x_{ij} s_{ij} \leq B \tag{2}$$

Given the values and sizes of all candidate objects, the above is a standard 0/1-Knapsack problem (which is known to be NP-hard). We can use the classical greedy heuristic: choose objects in *decreasing* order of *value-size* ratios, i.e., $\frac{\phi_{ij}}{s_{ij}}$ to fill the cache. Alternatively, when the cache is full, we drop the object that has the least value-size ratio, which gives the general dropping policy:

$$\textit{drop the object that has the least value of } \frac{\phi_{ij}}{s_{ij}}$$

Since we maintain a sorted list of objects in each cache and always drop objects from the tail, not all objects but the tail object needs to be considered as the candidate object. That means, we effectively do not need to iterate over j. Consequently, we can define the value and size of cache i, denoted by ϕ_i and s_i respectively, in terms of its tail object and can then drop from the cache that has the smallest value-size ratio. Therefore, the general dropping policy stands out to be:

$$\textit{drop from the cache that has the least value of } \frac{\phi_i}{s_i}$$

Note that the above choice of considering only the tail object per cache reduces the complexity of choosing objects to the linear order of the number of caches instead of the number of objects. By using appropriate data structure (e.g., heap), this can be implemented in logarithmic order. As the number of caches can be magnitude order lower than the number of objects, the technique scales well.

Now we show how utility gain, $\Delta(i,j,k)$, is computed. Recall that this is the "value" of caching of an object in the sense how much utility gain it achieves when it is retrieved from the cache compared to being fetched from the data cluster. It indirectly corresponds to how "costly" it is to retrieve the object from the data cluster if the object is ever dropped from the cache. There can be different choices for assigning these costs. One choice can be considering all objects equally costly (uniform cost). Another choice can be that the cost would depend on the size of the object. This is because larger objects are costlier to get fetched from the data cluster (cost depends on size). The cost could also depend on how much *additional* delay it incurs when the object needs to be fetched from the data cluster (cost depends on latency).

1) Uniform cost: All objects are equally costly, that is, all objects are of equal value. That means, $\Delta(i,j,k) = 1$. With this choice, the objective function of the caching problem becomes:

$$\sum_i \sum_j \sum_k x_{ij} z_{ijk} \tag{3}$$

which effectively asks to cache objects so that the *number* of objects returned from the cache is maximized. That is to say to maximize the cache hit ratio. In this, the value of an object is given by: $\phi_{ij} = \sum_k z_{ijk}$, that is, the number of subscribers attached to the object. Let $f_{ij} = \sum_k z_{ijk}$. Hence, the object that has lowest subscribers to size ratio is dropped.

2) Cost depends on size: In this approach, we define the value as $\Delta(i,j,k) = s_{ij}$, which leads to maximizing:

$$\sum_i \sum_j \sum_k x_{ij} z_{ijk} s_{ij} = \sum_i \sum_j x_{ij} f_{ij} s_{ij} \tag{4}$$

This is to choose objects so that the amount of bytes returned from the cache is maximized (maximizing cache hit byte). In this scheme, the object that has the fewest subscribers ($\min f_{ij}$) is dropped. Note that this policy is in fact a variant of the popular LFU (least frequently used) policy.

3) Cost depends on latency: In this scheme, we assume the additional latency to retrieve the object from the data cluster contributes to its cost. Let l_{ij} denote the latency of retrieving the object from the data cluster to the broker. In that, utility gain is $\Delta(i,j,k) = l_{ij}$ and the value function is $\phi_{ij} = l_{ij} f_{ij}$. Hence, the object with the least value of $\frac{l_{ij} f_{ij}}{s_{ij}}$ is dropped.

Consequently, we can have the three caching policies as shown in Table I.

TABLE I
UTILITY-DRIVEN CACHING POLICIES

Utility, $\Delta(i,j,k)$	Caching value, ϕ_{ij}	Dropping criteria	Name
Uniform, 1	$\sum_k z_{ijk} = f_{ij}$	$\min \frac{f_{ij}}{s_{ij}}$	LSCz
Size, s_{ij}	$f_{ij} \times s_{ij}$	$\min f_{ij}$	LSC
Latency, l_{ij}	$f_{ij} \times l_{ij}$	$\min \frac{f_{ij} l_{ij}}{s_{ij}}$	LSD

B. TTL (Time-to-Live) Caching

TTL cache sets an explicit *expiration time* on cached objects and drop them immediately after the expiration time elapses. The amount of time an object is held at the cache is referred to as TTL (Time-to-Live). The value of TTLs ultimately controls the size of the caches. If TTLs are set large, objects are held for longer durations allowing subscribers to come and retrieve them, but it raises the cache size. Smaller TTL values keeps the caches in smaller sizes but the subscribes may now not have enough time to retrieve their objects. Intuitively, the TTL value should depend on at what rate new results are added to the caches and at what rate results are consumed from the caches. By setting TTL values in an appropriate way, the total size of the caches can be kept bounded within the allowed cache size.

Instead of assigning TTLs to an individual object, we assign TTL to each cache and drop the tail object from the cache at a

periodic interval determined by the TTL. Let new results arrive in cache i at a rate λ_i bytes per second and objects are dropped at a rate η_i. Recall that an object is dropped when *all* of the subscribers attached to the object have retrieved the object unless the drop is due to the TTL expiration. The net growth of the cache, denoted by ρ_i, is given by the difference between the arrival rate and the consumption rate, that is, $\rho_i = (\lambda_i - \eta_i)^+$, where $x^+ = \max(0, x)$. Since objects are held in the cache for T_i seconds after which they are automatically dropped, at this growth rate the cache grows up to $\rho_i T_i$. That is, the expected size of cache i is $\rho_i T_i$. We want the total size of all caches to be bounded within B, which requires the following to hold:

$$\sum_{i=1}^{N} \rho_i T_i = B \tag{5}$$

A generalized solution to (5) can be of the form:

$$T_i = \omega_i \frac{B}{\rho_i} \tag{6}$$

where each cache can have a weight, ω_i and $\sum_i \omega_i = 1$. One way to assign weights can be in proportion to the number subscribers each cache is attached to, that is, for $n_i = |S(i)|$, $\omega_i = \frac{n_i}{\sum_i n_i}$, which gives TTL values as follows:

$$T_i = \frac{n_i}{\sum_{i=1}^{N} n_i} \frac{B}{\rho_i} \tag{7}$$

The broker keeps track of the incoming data rate and the consumption rate of each cache (by calculating moving averages over time) and computes TTLs for all caches at a certain interval, say every 5 minutes. It is important to note that unlike utility-driven caching, TTL caching does not guarantee that the total cache size would always remain within B. In that, the size constraint holds only in *expectation* sense and the cache can in fact grow beyond B if T_i's are set longer than their exact values. This can happen because the broker approximates the growth rates, ρ_i's, only to a certain accuracy without knowing their true values. For comparison purpose, we try an eviction version of TTL where the broker drops objects only when the cache is full and chooses the object that will expire soon (the earliest in the future) unless none of the objects have expired yet, otherwise the object that has already expired furthest in the past is dropped. We refer to this version of caching policy as EXP (drop earliest to be expired).

V. Simulation Results

We conduct an extensive simulation to show the performance variations among different caching schemes at scale. We wrote a discrete event simulator using Python that mimics the behavior of the broker (manages subscriptions and deliver channel results) as well as the backend data cluster (generates results at different rates for different channels). Table II shows the simulation settings. For each run, we created up to 1000 backend subscriptions to the data cluster and allow 10,000 subscribers to attach to any of them (following a random joining process). Each subscriber remains ON and OFF for mean durations of 20 and 30 minutes respectively following a lognormal distribution (the choice is due to [29], [30]). Each run is for six hours and result data are averaged over ten independent runs. While in real setting the number of subscribers may be millions with each subscription lasting for weeks or months, we scaled everything down in our simulation so that the experiments can be conducted within a bounded time.

TABLE II
Simulation settings

Setting	**Value**
No of subscribers	10000
Subscription per subscriber	10
No of unique subscriptions	1000
Subscription duration	Lognormal(1, 2) minutes
Result object size	Uniform(1KB, 500KB)
Allowed cache size	50MB — 500MB
Result object arrival	Poisson, rate 1 per 10–60sec
Subscriber ON duration	Lognormal(1.753, 1.425)
Subscriber OFF duration	Lognormal(2.415, 1.11)
Broker to data cluster bandwidth	10MB/s
Broker to subscriber bandwidth	1MB/s
RTT (broker to data cluster)	500ms
RTT (broker to subscribers)	250ms

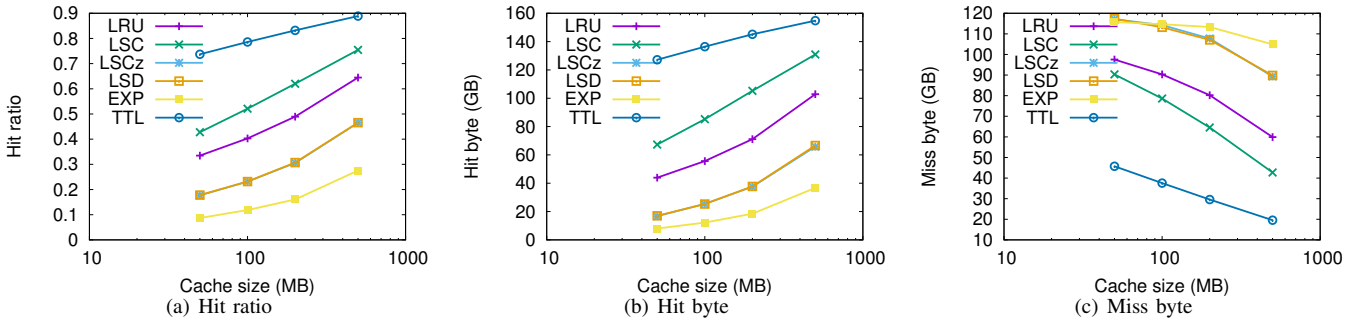
We implemented the following caching policies:

Scheme	Dropping criteria
LRU	drop from the least recently accessed cache
LSC	drop object with the fewest subscribers
LSCz	LSC normalized by object size
LSD	drop object with the least delay-size ratio
EXP	earliest object to be expired
TTL	drop objects when TTL expires

We measure the following performance metrics: *hit ratio*, *hit byte*, *miss byte*, *latency* and *fetch*. The hit ratio is the fraction of result objects that are served from the cache out of the total requests made by the subscribers. We also measure the amount of data served from the cache as hit byte. It is expected that the caching should improve subscribers latency to retrieve subscription results from the broker because a cache hit results in retrieving the results from the broker where a cache miss leads to higher delay due to additional latency of retrieving the missed objects from the data cluster. We measure the average latency across all subscribers as subscriber latency. The amount of total data bytes fetched from the data cluster is also an important performance metric as higher cache misses leads to higher data fetching. The metric, miss byte, accounts the amount of bytes fetched *only* due to cache misses.

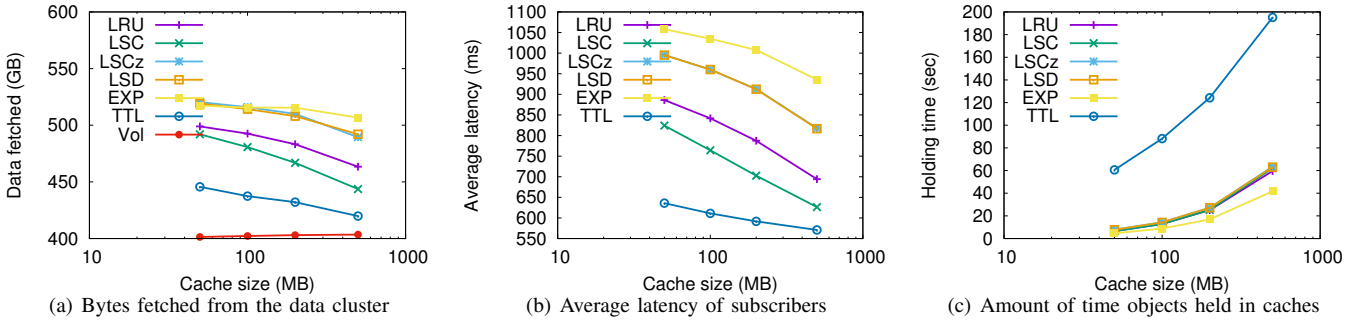
A. Performance of caching policies

Fig. 3 and Fig. 4 shows performance results of various caching policies as we vary the total cache size. We observe that as we increase the total cache sizes, the hit ratio and hit byte increase in all schemes, and subscriber latency and the amount of data fetched from the data cluster decreases. As we can observe, TTL has the highest hit ratio, LSC has higher hit ratio than LRU and EXP has the lowest hit ratio.



(a) Hit ratio (b) Hit byte (c) Miss byte

Fig. 3. Hit ratio, hit byte, and miss byte across caching policies.



(a) Bytes fetched from the data cluster (b) Average latency of subscribers (c) Amount of time objects held in caches

Fig. 4. Average latency of subscribers and the total amount of data fetched from the data cluster

In turns out that LSCz and LSD have very similar hit ratio results. Apparently, eviction version of TTL caching, that is, EXP does not perform well. Generally, whichever scheme has the higher hit ratio results in higher hit byte, lower miss byte, lower subscriber latency and lower fetch.

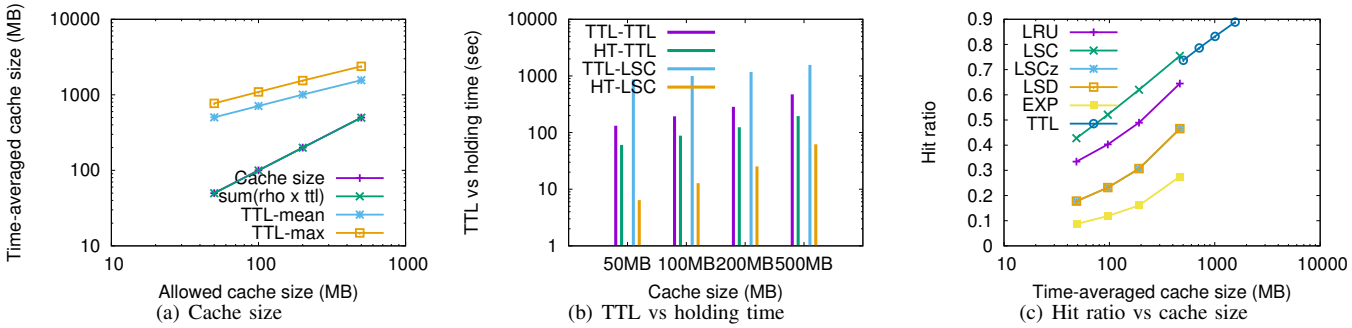
Fig. 4(a) shows fetch across different caching schemes. The line marked as 'Vol' denotes the total amount of data produced in the data cluster in response to all subscriptions. This is the base amount that the broker needs to pull from the data cluster (irrespective of caching policy used) to populate the content of the caches. The additional fetches are attributed to cache misses which is shown in Fig. 3(c) as miss byte. We observe that TTL scheme offers the lowest subscriber latency followed by LSC and then LRU. Note that in addition to RTTs among the broker and subscriber, the latency values add up the processing times as well as the data transfer times. We also measure how much time the caches *hold* objects in them (the time gap between an object is cached and dropped). We see, from Fig. 4(c), that the TTL cache holds objects for longer duration than other schemes and EXP holds for the shortest duration. This is why EXP performs worse than other schemes.

An important observation is that TTL-based caching has superior performance than eviction-based techniques. This is achieved, however, at an additional cost of slightly higher total cache size. Recall that TTL cache does not strictly adhere to the allowed cache size, rather tries to keep cache size bounded in an *expectation* sense. Therefore, on some occasions, the caches may grow beyond the allowed cache size. To show this, we measure the time-averaged cache size and the maximum cache size in all schemes (Fig. 5(a)). Time-averaged cache size is a weighted mean of cache sizes where the cache size is weighted by the amount of time the cache remains at that size. Maximum cache size denotes the largest size the cache ever grew. We observe that while eviction-based schemes limit themselves within the allowed sizes, TLL cache indeed exceeds the limit. We also show $\sum_i \rho_i T_i$, which fits almost exactly the allowed size (as per (5)) indicating that computed TTLs are consistent with the given cache size. It will be interesting to observe if holding times match with TTL values. Intuitively, these two values should match closely in TTL caches, which we in fact see in Fig. 5(b), compared to the same for LSC scheme. Note that the holding time can be less than TTL value as objects can be dropped when consumed by all subscribers (even before their TTL expires).

VI. Prototype Implementation and Results

We build a small scale BAD prototype system on a three-node data cluster based on AsterixDB [7] with a single broker node. Our broker node is a RESTFul HTTP server written in Python3 using the Tornado web framework [31].

Fig. 6 shows the interactions among the brokers, the broker coordination service (BCS), the data cluster and the subscribers. The BCS takes care of registering brokers when a new broker joins to the system. When a subscriber comes to the system, it contacts to the BCS and the BCS returns the IP address and port of a suitable broker that the subscriber can connect. The subscriber then interacts with the broker through a set of REST APIs, whereas the broker talks to



(a) Cache size (b) TTL vs holding time (c) Hit ratio vs cache size

Fig. 5. Comparing eviction-based caching policies with TTL caching.

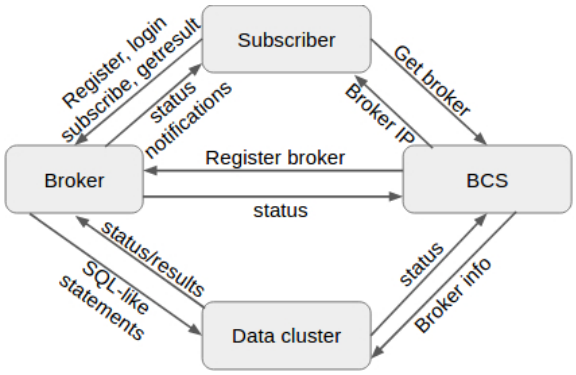


Fig. 6. Interactions among components of Big Active Data (BAD) systems

TABLE III
CHANNELS

Channel (params)	Predicates	P (s)
emrgn (E)	Events of type 'E'	10
emrgn (L)	— near loc 'L'	20
emrgn (E, L)	— of type 'E' near loc 'L'	30
emrgn-ws (E, L)	— of type 'E' near 'L' with shelters	60
emrgn-nm (U)	— near user 'U'	10
emrgn-nm (E, U)	— of type 'E' near user 'U'	10
emrgn-nm (E, U)	— of type 'E' near 'U' with shelters	10

the data cluster through AQL (Asterix Query Language). The subscriber programs are written in Python3 (using Tornado web framework) and they receive notifications from the broker through Websockets. We also tested desktop and Android clients that receive notifications from the broker through a RabbitMQ message queue server and GCM (Google Cloud Messaging) respectively.

A. Evaluation of the Prototype System for Caching

Although our prototype system does work correctly end-to-end, the current deployment is not fully capable of running very scaled experiments, especially not to the scale required to show the efficacy of caching. Therefore, the results reported here are mainly to validate the proof-of-concept rather not to represent the measurements of the actual system and its performance. The code for the broker and the experiment setup can be obtained at [32].

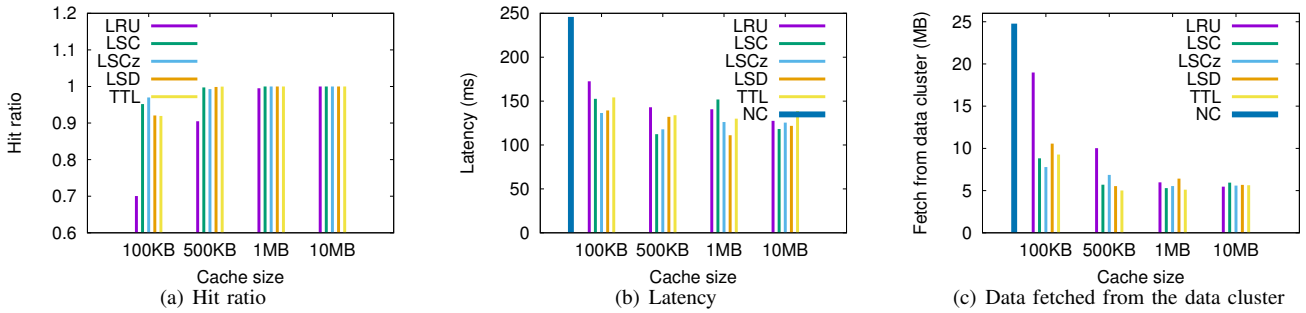
We use the following publish-subscribe usecase related to receiving notifications during emergencies in a city. Subscribers are interested about certain type of emergencies, such as tornado, flood, and shooting, happening in certain locations as expressed by different repetitive channels shown in Table III with their corresponding periods.

We produce a synthetic but random trace of subscribers interaction in the system, namely a series of timestamped activities such as login, logout, subscribe to parameterized channels and unsubscribe from the channels. These traces are then played back by a driver program. We also set a publisher that publishes geo-tagged and timestamped emergency reports and shelter information at an interval of around every 10 seconds (publications are text strings of size 200-1000 bytes). We let subscribers (randomly) move on the city and publish their locations. We run the experiments with 400 subscribers with nearly 3500 frontend subscriptions and 800 backend subscriptions. For each setting, we provide the same trace to all competing caching schemes and run the trace for an hour.

Fig. 7 shows the different results for the testbed deployment. We also show results without any caching (NC) appeared at the far left. We can see from the figures that having a cache in the broker significantly reduces the subscriber latency as well as the total number of bytes retrieved from the data cluster. We also observe that, as cache size grows hit ratio increases in all caching schemes, and both latency and bytes fetched decline. At this small scale of evaluation, LSC appears way better than LRU in terms of hit ratio. Again, TTL-based caching does slightly better at lower cache sizes. It is interesting to see that even a small cache size (100KB) results in high latency drop (around 50% drop) with a high hit ratio (above 50%). This is perhaps due to the fact that some subscriptions are very popular (due to Zipfian subscription model we used), so caching those objects results in higher hit ratio.

VII. Conclusion

In this paper, we develop intelligent caching techniques for a big data publish subscribe system. The BAD platform at UCI uniquely integrates big data management systems and distributed broker architectures for pub-sub to enable augment and enrich notifications and deliver them to a very large number of subscribers as quickly and efficiently as



(a) Hit ratio (b) Latency (c) Data fetched from the data cluster

Fig. 7. Hit ratio, subscribers latency and the amount of data fetched from the data cluster in the BAD prototype system.

possible. A range of system level issues arise in addressing scale and dynamicity in these systems - in addition to broker caching methods that alleviate backend and network overhead, techniques to manage state information for dynamic reconfiguration are being developed. These include methods for handling failures and support for efficient load balancing as loads on the brokers change dynamically as a result of events. This will enable us to get closer to our vision of implementing a scalable rich notification platform that enables "petabytes to megafolks in milliseconds".

ACKNOWLEDGMENT

This work is supported by NSF Award 1447720.

REFERENCES

[1] "Storm," 2014. [Online]. Available: http://storm.incubator.apache.org/
[2] "Event series intelligence: Esper and nesper," 2014.
[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.
[4] S. Jacobs et al., "A BAD demonstration: Towards big active data," *Proc. VLDB Endow.*, vol. 10, no. 12, pp. 1941–1944, Aug. 2017.
[5] M. J. Carey, S. Jacobs, and V. J. Tsotras, "Towards situational awareness on big data: A BAD approach," in *International Workshop on Real-time Analytics in Multi-latency, Multi-Party, Metro-scale Networks*, 2017.
[6] ——, "Breaking BAD: a data serving vision for big active data," in *ACM International Conference on Distributed and Event-based Systems*. ACM, 2016, pp. 181–186.
[7] "Big Active Data (BAD)." [Online]. Available: http://asterix.ics.uci.edu/bigactivedata
[8] C. Carnes, J. B. Park, and A. Vernon, "Scalable trigger processing," in *International Conference on Data Engineering*, ser. ICDE '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 266–.
[9] S. Podlipnig and L. Böszörmenyi, "A survey of web cache replacement strategies," *ACM Comput. Surv.*, vol. 35, no. 4, pp. 374–398, Dec. 2003.
[10] D. S. Berger, R. K. Sitaraman, and M. Harchol-Balter, "Adaptsize: Orchestrating the hot object memory cache in a content delivery network," in *USENIX NSDI*. Boston, MA: USENIX Association, 2017, pp. 483–498.
[11] Yaxiong Zhao, Jie Wu, and Cong Liu, "Dache: A data aware caching for big-data applications using the MapReduce framework," *Tsinghua Science and Technology*, vol. 19, no. 1, pp. 39–50, feb 2014.
[12] V. Sourlas, G. S. Paschos, P. Flegkas, and L. Tassiulas, "Caching in content-based publish/subscribe systems," in *IEEE GLOBECOM*. IEEE, 2009, pp. 1–6.
[13] M. Diallo, S. Fdida, V. SoOPTurlas, P. Flegkas, and L. Tassiulas, "Leveraging Caching for Internet-Scale Content-Based Publish/Subscribe Networks," in *IEEE ICC*. IEEE, jun 2011, pp. 1–5.
[14] V. Sourlas, G. S. Paschos, P. Flegkas, and L. Tassiulas, "Mobility Support Through Caching in Content-Based Publish/Subscribe Networks," in *IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE, 2010, pp. 715–720.
[15] G. Zhang, Y. Li, and T. Lin, "Caching in information centric networking: A survey," *Computer Networks*, vol. 57, no. 16, pp. 3128–3141, nov 2013.
[16] J. Jung, A. W. Berger, and H. Balakrishnan, "Modeling TTL-based internet caches," in *IEEE INFOCOM*, vol. 1. IEEE, 2003, pp. 417–426.
[17] N. C. Fofack, P. Nain, G. Neglia, and D. Towsley, "Performance evaluation of hierarchical TTL-based cache networks," *Computer Networks*, vol. 65, no. Supplement C, pp. 212 – 231, 2014.
[18] G. Einziger, R. Friedman, and B. Manes, "TinyLFU: A highly efficient cache admission policy," *ACM Trans. Storage*, vol. 13, no. 4, pp. 35:1–35:31, Nov. 2017.
[19] S. Traverso, M. Ahmed, M. Garetto, P. Giaccone, E. Leonardi, and S. Niccolini, "Unravelling the Impact of Temporal and Geographical Locality in Content Caching Systems," *IEEE Transactions on Multimedia*, vol. 17, no. 10, pp. 1839–1854, oct 2015.
[20] P. Cao and S. Irani, "Cost-aware WWW proxy caching algorithms," in *USENIX Symposium on Internet Technologies and Systems*. USENIX Association, 1997, pp. 18–18.
[21] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin, "Evaluating content management techniques for web proxy caches," *SIGMETRICS Perform. Eval. Rev.*, vol. 27, no. 4, pp. 3–11, Mar. 2000.
[22] S. Jin and A. Bestavros, "GreedyDual* web caching algorithm: Exploiting the two sources of temporal locality in web request streams," in *International Web Caching and Content Delivery Workshop*, 2000, pp. 174–183.
[23] M. Dehghan, L. Massoulie, D. Towsley, D. Menasche, and Y. C. Tay, "A utility optimization approach to network cache design," in *IEEE INFOCOM*. IEEE, apr 2016, pp. 1–9.
[24] V. Martina, M. Garetto, and E. Leonardi, "A unified approach to the performance analysis of caching systems," *IEEE INFOCOM*, vol. 1, no. 3, pp. 1–28, 2014.
[25] F. B. Sazoglu, B. B. Cambazoglu, R. Ozcan, I. S. Altingovde, Ö. Ulusoy, F. B. Sazoglu, B. B. Cambazoglu, R. Ozcan, I. S. Altingovde, and Ö. Ulusoy, "Strategies for setting time-to-live values in result caches," in *ACM CIKM*. New York, New York, USA: ACM Press, 2013, pp. 1881–1884.
[26] H. Che, Y. Tung, and Z. Wang, "Hierarchical web caching systems: Modeling, design and experimental results," *IEEE J.Sel. A. Commun.*, vol. 20, no. 7, pp. 1305–1314, Sep. 2006.
[27] S. Alici, I. S. Altingovde, R. Ozcan, B. B. Cambazoglu, and Ö. Ulusoy, "Adaptive Time-to-Live Strategies for Query Result Caching in Web Search Engines," in *European Conference on IR Research: Advances in Information Retrieval*. Springer, Berlin, Heidelberg, 2012, pp. 401–412.
[28] "AsterixDB." [Online]. Available: https://asterixdb.apache.org
[29] F. Benevenuto, T. Rodrigues, M. Cha, and V. Almeida, "Characterizing user behavior in online social networks," in *ACM SIGCOMM conference on Internet measurement conference*. ACM, 2009, pp. 49–62.
[30] J. Jiang, C. Wilson, X. Wang, W. Sha, P. Huang, Y. Dai, and B. Y. Zhao, "Understanding latent interactions in online social networks," *ACM Transactions on the Web (TWEB)*, vol. 7, no. 4, p. 18, 2013.
[31] "Tornado web framework." [Online]. Available: http://www.tornadoweb.org
[32] "BAD caching code." [Online]. Available: https://bitbucket.org/yusufsarwar/badbroker