COSAR: A Précis Cache Manager

Shahram Ghandeharizadeh, Jason Yap, Showvick Kalra, Jorge Gonzalez
Elham Keshavarzian, Igor Shvager, Felipe Cariño, Esam Alwagait
Computer Science Department
University of Southern California
Los Angeles, CA 90089

Abstract

Précis is a class of cache managers that stores and retrieves the final results of a time consuming operation that may use data from disparate sources. Examples include the results of an aggregate query, serialized instances of a data structure that contains results of different queries and computations used to generate a dynamic HTML page, etc. It is a high throughput, low latency key-value cache manager designed to scale a large data intensive web application. Example systems include memcached and COSt AwaRe (COSAR) cache manager. In this paper, we present the implementation details of COSAR and its variants of LRU, LRU-2, and GreedyDual-Size cache replacement techniques. We compare COSAR with memcached that employs the classical LRU replacement technique, showing the merits of its proposed replacement techniques.

1 Introduction

A Précis cache manager scales a data intensive web application to process a larger number of requests faster. It is a high throughput, low latency cache manager for key-value look ups. Examples include memcached [9] and COSAR. While COSAR is a new arrival, memcached was introduced several years ago and is in use by very large well-known sites including YouTube [4], Facebook [23], Twitter, and Wikipedia/Wikimedia among others.

A Précis cache manager is deployed at the same level of abstraction as the database management system (DBMS). A developer identifies key-value pairs using the requirements of an application. For most applications, the value is an instance of either a class or an in-memory data structure. The key may utilize one or more fields of this instance. Construction of the value requires execution of application specific logic that may involve processing of complex logic of retrieving, translating, and validating pieces of data from multiple sources such as DBMSs, Web Services, and distributed file systems. After identifying a unique key for the instance of the data structure, the developer modifies the software as follows. The software uses the key to look up the instance of the data structure in the Précis cache layer. If a value is found, the application deserializes it and proceeds to use it. Otherwise, it invokes the application specific logic to compute the value, stores the resulting key-value in Précis for future use, and processes the value. The order of the last two steps is not important and the application may store the key-value pair asynchronously. Below, we describe several specific examples.

Example 1: With a social networking web site, once a user (say Bob) logs in, the user's home page may show the profile of the user along with a list of his friends and their photos. This dynamic HTML page is generated by issuing queries to a DBMS. It may involve joins between different tables to identify the friends of Bob and their photos. Obtained results might populate a main-memory instance of a data structure incrementally. Once this instance is populated, it is used to generate the HTML page for the user's home page. Next, the system may serialize this instance (i.e. a value) and construct its unique key by concatenating the user-id, Bob, and the page-name, say Home. The application stores this key-value in the Précis cache layer. Subsequent visits of Bob to his home page causes the application software to construct the key, look up its associated value in the cache and deserialize it to obtain the main memory instance of the data structure used to construct the home page. A login by a different user, say Alice, who has no corresponding value in the Précis cache layer observes a cache miss, repeating the process outlined for Bob to generate the corresponding key-value for Alice.

When a user updates information pertaining to his or her profile, the application either deletes the key or inserts a new key-value to overwrite the old value. Deletion causes subsequent invocation of the application to observe a cache miss, compute the new value for the key, and insert it into the cache. □

Example 2: It is not uncommon to visit a web site and

encounter either an infomercial or a piece of news, named news-bytes collectively. These might be computed using the demographics of the visitor, the time of the day, and which pages the visitor has navigated thus far among others. A web site may maintain tens of demographics with thousands of news-bytes. With a large number of visitors, the software that computes the mapping between the news-bytes and demographics might not be fast enough to respond interactively. To speed up the application, one may assign a unique key for each identified demographic and associate the URL of the corresponding news-byte as a value for this key, storing the key-value pair in the Précis cache layer. When the web site is visited, the demographics of the visitor is used as the key to retrieve the URL of the news-byte for display. More complex variants of this usage scenario may 1) maintain multiple keys for one demographic group and use a function to show different news-bytes for a visitor that maps to this group, and 2) extend the value to include how the information should be presented to the user, e.g., layout of data, additional graphics and textual information, etc. □

Example 3: Consider a web site that maintains a user rating for certain products such as automobiles, books, and video and audio clips. When a visitor references a specific product, the system shows the average rating in the summary section of the page displayed for this product. This average rating might be computed by issuing an aggregate query that joins the comment table and the product table using the product-id, selecting the specific product that is being reviewed by a visitor, and computing the average rating. The developer may use the Précis cache layer to store and retrieve the key-value pair for the average rating, minimizing the number of DBMS references and freeing it to process other queries. The key for this query might be the concatenation of the product's unique id and a label that uniquely identifies the query (decided by the developer). Its associated value is the average rating retrieved from the DBMS. When a visitor posts a new commentary and a new rating for a specific product, the application may update the Précis cache in the same manner as detailed in the last paragraph of Example 1. □

Précis consists of a server and a client component. A server instance runs on a computer with substantial amount of memory. It is a shared resource deployed in a trusted environment (a data center) to communicate with many client instances. One may deploy multiple server instances. To enhance throughput and latency of each server, the client component implements techniques to (a) partition data among server instances and (b) recover from a server failure. Without a recovery mechanism, failure of a server results in degraded performance because it no longer retrieves key-value pairs. The client may interpret this as a cache miss and proceed to invoke the application specific logic to compute the missing key-value pair. To minimize the likelihood of this degraded mode of operation, client components may implement a replicated, decentralized address space across the servers, e.g., Chord [27], CAN [24], and SkiPeR [5] to name a few.

In this study, we focus on the server component of Précis. We assume the application provides a cost for each cached key-value pair. This is the time to execute the application specific logic to compute the value. A software developer may estimate it by instrumenting the application software. Such instrumentation is typical for monitoring a large web site [13]. When the total size of key-value pairs exceeds the server's available cache, the server must decide which objects should occupy its cache using a replacement technique. A technique may optimize for one or more of the following metrics:

- Précis (COSAR/memcached) execution time: Total processing time with a specific replacement strategy, excluding the time to service a request that observes a cache miss from the DBMS.
- DBMS execution time: Total DBMS service time to process those requests that observe a cache miss.
- Cache hit rate: Percentage of requests serviced using the cache.
- Byte hit rate: Percentage of bytes retrieved using the cache.

This study reports on all four metrics. However, the ideal replacement strategy must provide fast execution times and minimize the DBMS execution times. The latter is realized by caching the most time consuming, i.e. costly, values that are referenced frequently. We use cache and byte hit rates to explain the behavior of the different techniques.

In this paper, we present COSAR as a Précis cache manager with the flexibility to implement different replacement techniques. A configuration file dictates which technique is employed by COSAR. The main **contributions** of this paper are as follows. First, we introduce novel variants of LRU and LRU-2 to consider the cost of computing a key-value pair and perform admission control. We use COSAR to implement these simple variants to quantify their trade-off. A secondary contribution of this paper is to evaluate alternative techniques using metrics other than cache and byte hit rates. See Table 15 for a summary of these metrics and our findings. For example, we use the execution time of different algorithms to rank them. This comparison is possible because they are embodied by one implementation, namely COSAR. A technique that performs many unnecessary replacements is wasteful of resources, providing slow execution times. This is important because, with a burst of requests, a slow algorithm may result in significant queuing

delays that impact the response time observed by the users of a high traffic web site.

The rest of this paper is organized as follows. In Section 2, we survey the related research. Section 3 provides an overview of COSAR as a framework for the alternative replacement techniques. Sections 4, 5, and 6 describe how a function of the framework implements variants of LRU, LRU-2, and GDS replacement techniques. This discussion identifies the variant that is fastest, lowest-cost, and most adaptable to evolving access patterns, namely LRU:CA:AC, LRU-2:CA, and GDS:$\bar{q}$. We compare these with one another in Section 7. Section 8 compares memcached with COSAR. Brief conclusions and future research directions are presented in Section 9.

2 Related Work

In this section, we compare Précis with the query cache feature of a DBMS, a key-value persistent store, a proxy-cache server using its GreedyDual-Size cache replacement technique. We also compare it to search engines and their weighted cache replacement technique, and disk paging replacement techniques.

DBMS query cache feature: At first glance, Précis may appear similar to the query cache feature of a DBMS [16]. With this feature, a DBMS caches the results of a SQL query, making subsequent execution of the same query (or a different query that subsumes the cached query) much faster. The DBMS invalidates the cached result as soon as the base table(s) referenced by the query is modified. Précis is different because its key-value pairs may correspond to arbitrarily complex application software that includes queries to different DBMSs and repositories. As illustrated in Example 1, the cached value may correspond to a serialized instance of a data structure that fuses results of several queries and application logic together. This is possible because the developer, instead of the query optimizer of a DBMS, is using the characteristics of the application to both construct and maintain the cached key-value pairs. Note that this flexibility enables the developer to implement application specific policies such as invalidate the content of Précis for a given key after a fixed number of (say 5) updates to the base table(s) used to compute its value.

Persistent key-value storage managers such as Berkeley DB [21] (BDB) and Dynamo [6] store and retrieve a key-value pair. BDB is a centralized, configurable storage manager that supports access methods such as B-tree and hash, and concurrency control and crash recovery techniques to implement ACID properties of transactions. Dynamo deploys instances of BDB on commodity servers and partitions data across them using a decentralized address space similar to Chord [27]. It replicates data to minimize loss of data in the presence of server failures.

The Précis framework is different because it is a transient cache manager that requires its client component to implement data partitioning and availability techniques. Multiple simultaneous failures with Dynamo may result in loss of data. Précis server failures impact system throughput and latency as the clients observe higher cache miss rates, causing the application software to re-compute the cached data from the relevant data sources and DBMSs.

Our Précis cache manager, COSAR, employs BDB as a main memory storage manager and uses its access methods to implement its cache replacement techniques, see Section 3. The cache replacement technique (LRU) of BDB is not used because all data resides in main memory and is not paged from the disk drive.

Proxy caches: A Précis cache manager shares similarities with a proxy cache server [19]. Both manage variable sized objects with different costs of retrieval. With a proxy cache server, cost is the time to retrieve the referenced data item from a remote server. With a Précis cache manager, cost is the time to execute a query or some software that produces a cached value. At the same time, there are several key differences. First, a proxy cache may utilize the HTTP protocol to retrieve the value associated with a URL (key) when the value is not stored in its local cache. A Précis cache manager, however, does not initiate communication with content sources and therefore cannot retrieve the value associated with a key that does not exist in its cache. It relies instead on the application to specify all key-value pairs. Thus, unlike a proxy cache server, a Précis cache manager may not implement a consistency mechanism such as those described in [14, 18, 7]. Moreover, it is not possible to combine cache replacement and consistency maintenance in the spirit of techniques such as LNC-R-W3-U [26].

Second, to maximize performance, a Précis cache manager is in close proximity to, typically in the same location (data center) as, both the DBMS servers and the application servers. A proxy cache manager might be buildings apart from its clients and thousands of miles away from the remote servers whose contents it has cached.

Default configuration of a proxy may prevent it from caching dynamic content. Even if enabled, a service provider may include the necessary headers to tell the proxy not to cache its dynamic content. Thus, a proxy is not a substitute for a Précis cache manager that expedites generation of dynamic content.

GreedyDual-Size [2] (GDS) is a cache replacement technique introduced for use with proxy caches. It considers both the cost and size of cached objects. Section 6 details this technique and considers several variants of it. When compared with our novel variant of LRU-2 (LRU-2:CA, see Section 5), LRU-2:CA is faster and incurs lower costs.

Search engines: Result and index caching have been extensively studied in the context of search engines. Result

caching enables a search engine to re-use the result of the same query issued recently by either the same or different users. With index (or list) caching, a search engine maintains the inverted lists of frequently queried search terms in main memory. These studies can be taxonomized into those that either consider [10] or do not consider the cost of a query [25, 29, 17, 8, 1]. The former is most relevant because its concept of query execution time is identical to our concept of DBMS execution time (cost).

In [10], cost is minimized by partitioning the available cache into two segments (A and B) with a different technique managing each segment (say technique A and B). A technique assigns a score to each cached query result. When inserting into a full cache, it attempts to insert into either segment. When evicting from one segment, say A, it selects the lowest scoring item according to technique A. However, before throwing this item out, it first attempts to insert it into the other cache. If the item has a high enough score according to technique B, it will evict another item from Segment B and try to insert it into Segment A. This may cause a different item to be evicted from Segment A that the algorithm inserts into Segment B. This may repeat itself several times, until an item is evicted. This algorithm is a generalization of the SDC algorithm [8]. Two hybrids of it are investigated in [10]: 1) LRU and weighted LFU, and 2) GDS (Landlord) and weighted LFU. Both show 30% to 50% cost savings relative to no caching.

A key parameter of the algorithm of [10] is the fraction of space allocated to each technique, computed by experimenting with different settings using a trace. Our variants of LRU do not partition cache space and are parameter free. They incorporate cost as an integral component in the spirit of GDS [2] and Landlord [31] techniques. A comparison of our proposed techniques with those of [10] is a future research direction.

Disk paging and ARC: The adaptive replacement cache [20] (ARC) technique continually balances between the recency and frequency features of a workload, improving cache hit rate for fix sized disk pages with a constant cost. The Précis framework is different because it assumes the workload references variable sized data items with different costs. Hence, a higher cache hit rate does not result in the best average delay experienced by the end user. It is realized by a technique that provides the best Précis execution and DBMS execution times.

3 Overview

With a Précis cache manager, when a key is inserted with different values, the last inserted key-value pair overwrites all previous ones. Some applications may require the same key with different values. For example, with cloud computing where different instances of the same application (for different customers) are sharing a COSAR server, each instance may maintain the same key with a different value. COSAR supports this using the concept of data zones. A data zone is a collection of key-value pairs with unique keys. In our example, the developer specifies a unique COSAR data zone name (e.g., customer name) for each application instance to support all applications with one COSAR server.

Figure 1 shows the pseudo-code for the general framework used to implement the alternative cache replacement algorithms. The essence of this framework consists of three components. First, the value of q associated with each object and dictated by a function specific to a replacement strategy. Second, the sorted order of q_i values to facilitate victim selection. Third, an admission control mechanism to decide whether a new object should be admitted into the cache.

We use the term *object* to refer to a logical (K, DZ, V) triplet where K is a key, DZ is the name of a data zone, and V is a value. Physically, this object is represented as three records stored in two main memory relations[1] in Berkeley DB. These two tables are named QDB and CacheDB. One record is stored in QDB and two records are stored in CacheDB, see Table 2. We describe the two tables, their record structures, and how they interact in turn.

QDB maintains the metadata required by each replacement technique to select victims. A metadata (MD) record in QDB contains the q value of each object along with its K, DZ, and metadata. A B+-tree index on the q attribute of the records provides a sorted order required by a replacement technique to select victim objects.

CacheDB is a hash-indexed relation that employs the concatenation of letter 'D', K and DZ as its key and V as its remaining attribute. When an application invokes a "Get" operation by providing a K_i and DZ_i, COSAR concatenates letter 'D', K_i and DZ_i to probe the hash index for the stored value.

To update q_i of an object i that observes a cache hit, COSAR maintains a metadata look-up (MDL) record in CacheDB. The key of this record is concatenation of the character 'Q' with K_i and DZ_i, storing q_i as its remaining attribute value. We use the character 'Q' in the key to differentiate this record from the data record in CacheDB.

COSAR implements a replacement technique as follows. Assuming a cold cache that has a substantial amount of free space, when an object O_i is inserted for the first time, COSAR inserts its data record in CacheDB. Next, COSAR invokes the $f(K_i, DZ_i, L)$ method of its replacement strategy to generate the q_i. This, along with K_i and DZ_i, con-

[1]Termed databases by Berkeley DB. We could not use the concept of primary and secondary indicies to minimize the number of tables to one. This is because the utilization of memory drops drastically (to less than 10%) when Berkeley DB is configured with a secondary index.

Record name	Relation	Key	Value	Index	Purpose
Data	CacheDB	'D':K_i:DZ_i	V_i	Hash	Look up value V_i for the specified (K_i, DZ_i).
Meta-data look up (MDL)	CacheDB	'Q':K_i:DZ_i	q_i	Hash	Look up q_i value when (K_i, DZ_i) observes a cache hit.
Meta-data (MD)	QDB	q_i:K_i:DZ_i	metadata	B^+-tree	Identify object with the lowest q value for eviction and retrieve metadata specific to a cache replacement technique.

Table 2. COSAR database design.

Term	Definition
L	The q value of the last evicted object.
$\bar{S}$	Average object size.
K_i	Unique key for object i.
V_i	Value associated with K_i.
DZ_i	Name of data zone for object i.
S_i	Size of object i.
T_j	Time stamp of the j^{th} least recent reference to an object.
F_i	Frequency of access to object i.
q_i	Dictated by a function specific to a replacement strategy and used to select victims.

Table 1. List of terms used repeatedly and their respective definitions

```
COSAR.Get(K_x, DZ_x)
{
  Let W_x denote concatenation of K_x and DZ_x;
  Look up W_x in the cache to find V_x;
  if (V_x is found){
      q_x = Strategy.f(K_x, DZ_x, L);
      Update q_x;
      return V_x;
  }
  else return value not found;
}

COSAR.Insert(K_x, DZ_x, V_x)
{
  q_x = Strategy.f(K_x, DZ_x, L);
  if COSAR.EXISTS(K_x, DZ_x)
  {
   Overwrite old value with V_x;
   Update the old q value with q_x;
   return;
  }
  if (Strategy.Admit(q_x, L))
  {
   while(free cache space < V_x.Size)
   {
      Let O_j denote the object with minimum q_j value;
      Evict O_j from cache;
      Set L = q_j;
   }
   Store (K_x, DZ_x, V_x) and maintain its q_x value;
  }
}
```

Figure 1. A cache replacement framework.

stitute a metadata record that is inserted in the QDB. In addition, COSAR generates the MDL record to look up this MD record given its K_i and DZ_i values.

Once the cache is full and COSAR is invoked to insert a new object, COSAR invokes the admission control of the replacement strategy to decide if the object should be inserted into the cache. (If a replacement strategy lacks an admission control, it may return true to insert the object always.) To select a victim, COSAR retrieves the MD record with the lowest q value from QDB. This record contains its associated K and DZ attribute values, enabling COSAR to delete the corresponding data and MDL records in CacheDB. Next, COSAR deletes the MD record. This process is repeated until there is sufficient free space to cache the admitted object.

We analyzed a variety of databases, access patterns, and cost models to evaluate the alternative cache replacement strategies. Our observations held true across different settings. Below, we report on one collection of these parameter settings. These are based on our analysis of the content of Précis cache servers deployed at MySpace.

For our comparison, we used a synthetically generated database. It is 1 Gigabyte in size. The minimum object size is 10 bytes and we vary the maximum object size to support different distributions of object sizes. Object sizes are uniformly distributed between the minimum and the maximum sizes. Thus, the average object size is the average of the

minimum and maximum. This average dictates the number of objects in the database because the size of the database is fixed at 1 Gigabyte. For example, with a maximum object size of 2 Kilobytes, the average object size is approximately 1 Kilobyte and the database consists of 1 million objects. With a maximum object size of 20 Kilobytes, the average object size is approximately 10 Kilobytes and the database consists of 100,000 objects. We use the average size of objects, $\bar{S}$, to refer to the different databases.

A trace file issues ten million "Gets" for the different objects. It uses a Zipfian distribution with a mean μ to generate object requests. A small μ value results in a skewed access pattern while a larger μ value results in a uniform pattern of access to the objects. When a "Get" fails to find its referenced object, the client attempts to insert the referenced object into the cache. The admission control component of the cache replacement technique decides whether the object should be inserted or rejected from the cache.

We also consider a trace file that emulates an evolving pattern of access to the objects. This is realized as follows. Assume each object is assigned to a single tick mark on the x-axis. There are one million tick-marks corresponding to the number of objects, with a higher frequency of access for the object assigned to a smaller x-axis value. This means tick-mark 1 has the object with the highest frequency of access. The next frequently accessed object is the one assigned to tick-mark 2 on the x-axis.

We use a mapping function to assign objects to the tick marks of the x-axis. This function is $M(i) = (i + \delta)\%N$ where i is the object-id and N is the total number of objects in the system. To emulate an evolving access pattern, we start with $\delta = 0$ and generate α requests using a Zipfian distribution with $\mu = 0.27$. Next, we increment δ by a fixed constant Δ and generate α requests again. This process is repeated to generate a trace file with ten million requests. For example, with α=500,000 and Δ=1, we change the value of δ twenty times, incrementing it by one after every 500,000 requests. This would represent a gradual shift in the access pattern. An aggressive shift would utilize a larger value for Δ, say 10,000. We represent a shift with its α and Δ values, see Table 12.

We report on three different cost models to quantify the DBMS execution times: Constant, Exponential, and Paced Exponential (PacedExp). As suggested by its name, with the first, the DBMS execution time is a fixed constant, 1 millisecond, for all referenced objects. With Exponential, objects are partitioned into four groups and the DBMS execution time of each group is 1, 10, 100, and 1000 milliseconds. Assuming objects are numbered from 0 to N, the following function for object i realizes this cost model: $C_i = 10^{(i \bmod 4)}$. With PacedExp, 20% of objects cost 1 millisecond, 60% cost 10 milliseconds, and the remaining 20% cost 100 milliseconds. The cost function for object i is:

$$C_i = \begin{cases} 1\ msec & \text{if } i \bmod 10 \text{ is 0 or 1} \\ 10\ msec & \text{if } i \bmod 10 \text{ is 2 to 7} \\ 100\ msec & \text{if } i \bmod 10 \text{ is 8 or 9} \end{cases} \quad (1)$$

With the Constant cost model, a replacement technique that provides the highest cache hit rate minimizes the total DBMS execution time. With the Exponential and PacedExp cost models, both the cost (C_i) and the frequency of access (F_i) to the objects is important. Ideally, a technique should cache those objects with the highest $\frac{F_i \times C_i}{S_i}$ value [12] where S_i is the size of object i.

All reported COSAR (and memcached) execution times are based on a Lenovo desktop with the following specifications: Intel's 64 bit Q9400 CPU, quad core each with a rating of 2.66 GHz, 3.71 GB of memory, and the 64 bit edition of Microsoft Windows Server 2003 with Service Pack 2. This study focuses on the observed service times with one core. An investigation of system throughput and its scalability as a function of the number of cores is a future research direction, see Section 9.

Below, we detail the alternative replacement techniques.

4 Least Recently Used (LRU)

Introduced more than a quarter of a century ago, Least Recently Used (LRU) replacement policy manages the objects stored in a cache using the time stamp of their last reference. When the cache is full and the application attempts to store a new object, LRU victimizes those objects with the least recent time stamp. COSAR implements LRU by using the current clock time as the value of function f, see Figure 1 and its discussion in Section 3.

The original LRU does not consider the cost incurred to retrieve an object from the DBMS. Moreover, it has no admission control. We introduce LRU:Cost Aware (LRU:CA) as a variant that computes the value of an object as the product of its cost and the time stamp (T_1) of the current reference, $f = T_1 * C_i$. This magnifies the recency of a reference to an expensive object, enabling it to remain cache resident for a longer period of time.

With the Exponential and PacedExp cost models for DBMS service times, LRU:CA provides substantial savings in the incurred costs when compared with LRU. This is shown in the middle and last columns of Table 3. For example, with 512 MB of cache space, the total DBMS execution time with LRU:CA is twice as fast as those seen with LRU. This enhancement is at the expense of lower cache and byte hit rates, see Table 4. This is justified because enhancing cache hit rate is inappropriate when it fails to enhance service time.

We considered a variant of LRU:CA that incorporates the following admission control criterion: It maintains the q

Cache Size (MB)	Constant Cost Model (Seconds)			Exponential Cost Model (Seconds)			PacedExp Cost Model (Seconds)		
	LRU	LRU:CA	LRU:CA:AC	LRU	LRU:CA	LRU:CA:AC	LRU	LRU:CA	LRU:CA:AC
32	8,210	8,213	8,213	2,278,830	1,980,290	1,993,000	215,281	196,012	193,143
64	7,604	7,608	7,608	2,110,420	1,716,520	1,735,710	199,306	174,626	170,859
128	6,815	6,822	6,822	1,891,050	1,353,900	1,384,680	178,423	146,346	141,013
256	5,789	5,801	5,801	1,606,380	878,285	912,226	151,663	111,097	104,309
512	4,901	4,916	4,916	1,359,830	580,923	582,948	128,520	85,382	82,227
1024	3,249	3,278	3,278	901,080	359,935	361,661	85,388	53,937	52,310
1536	2,078	2,126	2,126	576,863	299,929	300,033	54,469	37,565	36,164

Table 3. Total cost (DBMS execution time) with LRU, LRU:CA and LRU:CA:AC and 3 different cost models, $\bar{S}$=1K.

Cache Size (MB)	Cache Hit Rate			Byte Hit Rate			COSAR Execution Time (Seconds)		
	LRU	LRU:CA	LRU:CA:AC	LRU	LRU:CA	LRU:CA:AC	LRU	LRU:CA	LRU:CA:AC
32	17.90	8.36	7.78	17.43	7.76	7.20	749	702	255
64	23.96	11.20	10.50	23.50	10.63	9.96	738	705	261
128	31.85	15.42	14.51	31.42	14.89	14.01	718	689	276
256	42.11	22.14	21.10	41.73	21.70	20.68	662	692	258
512	50.99	28.65	27.84	50.66	28.30	27.50	627	750	276
1024	67.51	46.99	46.35	67.29	46.81	46.26	582	660	397
1536	79.22	62.30	61.93	79.07	62.12	61.83	552	631	459

Table 4. A comparison of LRU with LRU:CA and LRU:CA:AC with the Exponential cost model, $\bar{S}$=1K.

value of the last evicted object, L in Figure 1, and does not admit object i with a q_i value lower than L. We name this variant LRU:CA:AC. Table 4 shows this variant provides cache and byte hit rates comparable to LRU:CA. However, its execution time to process 10 million requests is significantly faster than the other two alternatives. With small cache sizes, it is almost 3 times as fast because it does not perform unnecessary evictions such as a replacement that brings an object into the cache only to have this object evicted by the next insert operation. With larger cache sizes, it services a larger number of objects from the cache that is comparable to the other two alternatives, outperforming them by a smaller margin.

With the Constant cost model (see Table 3), the function f used by LRU:CA and LRU:CA:AC reduces to the expression used by LRU, $f = T_1$. Thus, all three provide identical performance.

With an evolving access pattern, both LRU:CA and LRU:CA:AC continue to provide a lower total cost when compared with LRU. LRU:CA:AC does not outperform LRU:CA always. With the PacedExp cost model, LRU:CA:AC is slightly better than LRU:CA. With the Exponential cost model, the reverse is true. With both, the percentage difference is small. Most often it is less than 1%. In a few cases, it is as high as 10%. These small differences might be attributed to experimental noise.

Both LRU:CA:AC and LRU:CA are resilient to both slow (small values of Δ) and drastic (large Δ and α values) changes in access patterns. When compared with themselves using a non-evolving access pattern (Δ=0 and α=0), the percentage difference is less than 15%.

We also analyzed a variant of LRU:CA that employs cost per byte to compute the value of q, i.e. $f(K_i, DZ_i) = T_1 \times \frac{C_i}{S_i}$. This variant is worse than LRU:CA with small cache sizes (less than 1 Gigabyte) and provides negligible improvements with large cache sizes (1 Gigabyte and larger). We revisit this variant with GDS, see Section 6.

In summary, LRU:CA:AC is a superior alternative to LRU because its execution is significantly faster, its total DBMS execution time is lower, and it adapts to evolving access patters quickly. With the Constant cost model, it reduces to LRU.

5 LRU-2

Since the introduction of the LRU-K [22] technique, there have been several implementations of LRU-2 [15, 30]. This section presents a simple implementation of LRU-2 using the framework of Figure 1. Next, we extend it to incorporate cost and the concept of admission control. These variants are named LRU-2:CA and LRU-2:CA:AC, respectively. LRU-2:CA minimizes total DBMS execution time. However, we were surprised to find its extension with admission control (LRU-2:CA:AC) perform poorly. In Section 5.1, we present an implementation of the admission control techniques of [15, 30]. With limited amount of memory, these variants fail to minimize total DBMS execution time.

COSAR implements LRU-2 by maintaining a base time stamp, β, that is delta time units prior to when the system started[2]. When an object is first referenced, COSAR uses β as the second time stamp for this object, $T_2 = \beta$, and the current time stamp as the least recent time stamp for the object, T_1 = Clock Time. When an object is referenced a second time, we update its time stamp, stored in a MD record in QDB, as follows: a) T_2=T_1, and b) T_1 = Clock Time. We define $f(K_i, DZ_i)$ as the average[3] of these two time stamps, $f(K_i, DZ_i) = \frac{T_1+T_2}{2}$.

LRU-2:CA, cost aware variant of LRU-2, defines $f(K_x, DZ_x)$ by multiplying the cost of an object x, C_x, with the average of its two time stamps, $f(K_x, DZ_x) = C_x \times \frac{(T_1+T_2)}{2}$. Tables 5 and 6 present a comparison of LRU-2 with LRU-2:CA, showing LRU-2:CA reduces the total DBMS execution time significantly. Moreover, with the Constant cost model, LRU-2:CA provides identical performance to LRU-2 because its definition of $f(i)$ reduces to be identical to LRU-2.

Similar to the discussion of LRU in Section 4, we considered a variant of LRU-2:CA with the following admission control. It maintains the q value of an evicted object (L) and does not admit object i when $q_i < L$. This variant is named LRU-2:CA:AC. The incurred cost with this technique is as bad as LRU-2 with cache sizes smaller than 256 MB, see Table 6. With larger cache sizes, LRU-2:CA:AC becomes better than LRU-2. However, it remains significantly worse than LRU-2:CA. The explanation for this is as follows. Once the cache becomes full and the first object is victimized (establishing the value q_j for admission control), very few objects are admitted. This is because LRU-2:CA:AC employs β to compute the average time stamp computed for a referenced object that does not reside in the cache, preventing these objects from being admitted into the cache.

With larger cache sizes, the number of cache resident objects increases. This reduces the threshold (q value of the last victimized object) used by admission control, enabling LRU-2:CA:AC to improve.

LRU-2:CA:AC performs very few insertions and replacements, explaining its fast execution times in Table 5.

In summary, among the different alternatives for LRU-2, LRU-2:CA is most appropriate for objects with different

[2]Recall that the contents of the cache disappear when COSAR shuts down. Hence, it is acceptable for the value of β to be reset each time COSAR starts up.

[3]An alternative is to use T_2, i.e. $f(K_i, DZ_i) = T_2$. In our experiments, this alternative provides a 10% to 15% lower cache and byte hit rates than using the average of the two time stamps.

Cache Size (MB)	Cache Hit Rate			Byte Hit Rate			COSAR Execution Time (Seconds)		
	LRU-2	LRU-2:CA	LRU-2:CA:AC	LRU-2	LRU-2:CA	LRU-2:CA:AC	LRU-2	LRU-2:CA	LRU-2:CA:AC
32	23.46	9.43	10.50	22.91	8.88	10.24	593	665	149
64	28.94	11.77	14.63	28.43	11.23	14.33	580	665	143
128	35.55	15.16	21.21	35.07	14.67	20.87	562	654	167
256	43.64	22.05	30.97	43.24	21.64	30.64	506	648	219
512	50.99	36.55	40.03	50.63	36.26	39.70	479	576	273
1024	65.24	57.87	62.07	65.00	57.58	61.75	424	550	327
1536	76.39	76.89	78.43	76.24	76.72	78.26	337	500	398

Table 5. A comparison of LRU-2 with LRU-2:CA and LRU-2:CA:AC using the Exponential cost model, $\bar{S}$=1K.

Cache Size (MB)	Constant Cost Model (Seconds)			Exponential Cost Model (Seconds)			PacedExp Cost Model (Seconds)		
	LRU-2	LRU-2:CA	LRU-2:CA:AC	LRU-2	LRU-2:CA	LRU-2:CA:AC	LRU-2	LRU-2:CA	LRU-2:CA:AC
32	7,659	7,661	8,056	2,122,760	1,903,340	2,220,800	200,713	187,531	211,007
64	7,110	7,116	7,415	1,969,360	1,691,360	2,031,670	186,365	169,500	193,036
128	6,443	6,456	6,554	1,785,430	1,409,420	1,777,740	169,350	145,848	169,385
256	5,624	5,636	5,420	1,562,380	950,551	1,448,880	147,374	113,153	139,233
512	4,910	4,918	4,444	1,359,360	436,546	1,175,970	128,404	82,969	113,321
1024	3,459	3,456	2,620	966,689	304,500	689,492	90,380	45,405	66,769
1536	2,355	2,394	1,493	656,737	286,360	404,240	61,723	34,182	39,376

Table 6. Total cost (DBMS execution time) with LRU-2, LRU-2:CA and LRU-2:CA:AC and 3 different cost models, $\bar{S}$=1K.

Cache Size (MB)	Cache Hit Rate			Byte Hit Rate			COSAR Execution Time (Seconds)		
	LRU-2	LRU-2:TA	LRU-2:TSA	LRU-2	LRU-2:TA	LRU-2:TSA	LRU-2	LRU-2:TA	LRU-2:TSA
32	23.5	21.3	20.8	22.9	20.8	20.2	557	501	504
64	28.9	26.4	26.4	28.4	25.9	25.8	534	489	485
128	35.5	34.9	34.9	35.0	34.4	34.3	505	453	449
256	43.8	46.2	46.2	43.4	45.8	45.8	470	406	405
512	50.9	56.0	56.0	50.5	55.7	55.7	432	357	357
1024	65.5	73.3	69.7	65.2	72.4	68.5	361	296	292
1536	76.7	84.8	85.0	76.6	84.7	84.8	299	252	249

Table 7. A comparison of LRU-2 with LRU-2:TA and LRU-2:TSA with the Constant cost model, $\bar{S}$=1K.

costs.

5.1 Temporal and size-based admission control

This section presents extensions of LRU-2 with temporal [15] and size based [30] admission control. Using the Constant cost model, we show temporal admission works well when COSAR maintains substantial amount of historical data. At the end of this section, we extend these techniques with cost, showing they fail to minimize total DBMS execution time with small cache sizes.

The original 2Q [15] implementation of LRU-2 includes the concept of admission control where the very first reference to an object generates its history in one queue. Only the second reference admits the object into the cache. We implemented this variant by modifying our implementation such that the first reference to an object, detected by a lack of a MD record, generates the MD record in QDB without bringing the object into the cache. The second reference to an object, detected by the presence of a MD record in QDB, admits the object into the cache. We name this strategy LRU-2 with temporal admission, LRU-2:TA.

A size based admission control technique is described in [30]. With this extension, once O_i satisfies the temporal admission control, it is inserted into the cache with the probability $p_i = 1 - \frac{S_i - S_{min}}{2 \times (S_{max} - S_{min})}$ where S_{max} and S_{min} are the maximum and minimum size of objects in the cache respectively, and S_i is the size of O_i. This equation gives preference to smallest objects being admitted into the cache. If O_i is the largest object, its p_i will equal one half. There is a high likelihood of this object being admitted into the cache with two consecutive references for it. We named this variant as LRU-2 with temporal and size admission control, LRU-2:TSA.

Table 7 shows a comparison of these variants of LRU-2 with the Constant cost model. LRU-2:TSA is almost the same[4] as LRU-2:TA. Based on this, we focus on LRU-2:TA and exclude LRU-2:TSA.

With cache sizes smaller than 256 MB, LRU-2 provides a higher cache and byte hit rates when compared with LRU-2:TA. The reverse holds true with caches sizes of 256 MB and larger. This is because LRU-2:TA maintains more historical information with larger cache sizes, enabling it to make better eviction decisions. This results in fewer replacements, enabling LRU-2:TA to provide a better execution time. With smaller cache sizes, the execution time of LRU-2:TA is better than LRU-2 because it performs fewer retrievals due to its lower cache and byte hit rates.

The amount of historical information has a significant impact on the performance of LRU-2:TA. This is specially true with evolving patterns of access to the data. To demonstrate this, we considered two different implements of LRU-2:TA. A practical one that assumes QDB competes with CacheDB for the available cache space. And, a theoretical LRU-2:TA with infinite cache space for QDB. These are labeled as "LRU-2:TA (Finite)" and "LRU-2:TA (Infinite)" in Tables 8 and 9. These tables show two different evolving access[5] patterns: Shift Δ=10K every α=100K traces and α=500K traces. The α=100K trace changes more frequently than α=500K.

The theoretical LRU-2:TA maintains a few hundred thousand historical records in QDB that enables it to make intelligent replacement decisions. It provides the most desirable values for our quantifiable metrics: its cache and byte hit rates with 32 MB of memory are better than those of LRU-2 with 512 MB of memory, see Table 8. With cache sizes less than 256 MB, the practical implementation of LRU-2:TA maintains one tenth of historical records of its theoretical counter-part, reducing the quality of its replacement decision. This results in a significantly lower cache and byte hit rates, demonstrating the sensitivity of this technique to the amount of historical records.

We considered LRU-2 Cost Aware with temporal admission control (LRU-2:TA:CA), and LRU-2 Cost Aware with temporal and size admission control (LRU-2:TSA:CA). Similar to LRU-2:CA, both employ $f(K_x, DZ_x) = C_x \times \frac{(T_1+T_2)}{2}$. LRU-2:TSA:CA performs the same as LRU-2:TA and is excluded from further discussion.

LRU-2:TA:CA incurs a higher cost when compared with LRU-2:CA. With cache sizes of 1 Gigabyte and smaller, LRU-2:TA:CA is almost as bad as LRU-2. The explanation for this is as follows. When COSAR starts, it admits all inserted objects until its cache is full. Once evictions start, the cache is occupied by a mix of inexpensive and expensive (low and high cost) objects. With LRU-2:TA:CA, for a high cost object to evict a low cost one, its references must be close enough in time for its historical (MD) record to exist in QDB when the second reference is issued. With a finite amount of memory, this is a rare event for those expensive objects with a moderately high frequency of access, preventing COSAR from materializing expensive objects. LRU-2:CA does not have this limitation because it always admits an object into the cache on its first reference, enabling expensive ones with moderately high frequency of access to become cache resident.

6 GreedyDual-Size (GDS)

GreedyDual-Size [2] (GDS) is an extended version of the GreedyDual algorithm [31] (GD). The original algorithm

[4] In some experiments (not reported here), LRU-2:TSA is slightly better because it gives preference to caching smaller objects.

[5] Both LRU-2:TA and LRU-2:TSA provide comparable results with LRU-2:TSA being slightly better. Hence, we show the performance with LRU-2:TA only.

Cache Size (MB)	Cache Hit Rate			Byte Hit Rate			COSAR Execution Time (Seconds)		
	LRU-2	LRU-2:TA (Finite)	LRU-2:TA (Infinite)	LRU-2	LRU-2:TA (Finite)	LRU-2:TA (Infinite)	LRU-2	LRU-2:TA (Finite)	LRU-2:TA (Infinite)
32	7.9	5.0	20.8	7.9	5.0	20.8	648	579	757
64	8.9	6.8	25.1	8.9	6.8	25.1	652	568	729
128	11.1	10.8	31.2	11.1	10.8	31.2	647	556	684
256	15.4	18.4	39.7	15.4	18.4	39.7	636	538	629
512	20.7	28.0	47.6	20.7	28.0	47.6	607	507	570
1024	37.1	52.1	65.7	37.2	52.2	65.7	530	437	451
1536	57.4	72.2	80.5	57.4	72.2	80.5	422	373	333

Table 8. Impact of memory on LRU-2 with alternative admission control mechanisms (Constant cost model) and evolving pattern of access Δ=10K shift every α=100K requests, $\bar{S}$=1K.

Cache Size (MB)	Cache Hit Rate			Byte Hit Rate			COSAR Execution Time (Seconds)		
	LRU-2	LRU-2:TA (Finite)	LRU-2:TA (Infinite)	LRU-2	LRU-2:TA (Finite)	LRU-2:TA (Infinite)	LRU-2	LRU-2:TA (Finite)	LRU-2:TA (Infinite)
32	12.0	8.7	23.9	11.9	8.6	23.9	626	568	736
64	13.9	12.2	29.6	13.8	12.1	29.6	623	543	699
128	17.8	18.6	36.9	17.8	18.5	36.9	611	524	644
256	25.2	27.4	46.3	25.2	27.4	46.3	580	495	582
512	34.1	38.8	54.5	34.0	38.8	54.5	530	442	519
1024	56.6	66.7	71.3	56.6	66.7	71.3	414	336	401
1536	75.5	83.0	85.6	75.5	83.0	85.6	313	275	277

Table 9. Impact of memory on LRU-2 with alternative admission control mechanisms (Constant cost model) and evolving pattern of access Δ=10K shift every α=500K requests, $\bar{S}$=1K.

considers fix-sized pages that incur different costs to read from a secondary storage device into the cache. When a page j is brought into cache, this algorithm maintains a variable q_j for this page and assigns it the cost, C_j, associated with bringing this page into the cache. When a replacement must be made, the page with the smallest q_k value, q_{min}, is replaced. Moreover, the value of q_i for every page i is reduced by q_{min}. If a page j observes a cache hit, its q_j is restored to the cost C_j of reading it into the cache. Hence, recently referenced pages have a larger fraction of their original cost than those that have not been accessed for a long time.

The GDS algorithm extends GD by incorporating the sizes of the cached objects. This is achieved by setting the value of q_j to $\frac{C_j}{S_j}$ when object j is referenced. S_j is the size of object j.

At the first glance, GDS (and GD) appear to require N subtractions when a replacement is made where N is the number of cached objects. An efficient implementation maintains an "inflation" value L and adds it to all future settings of q_i. This is reflected in the pseudo-code of Figure 1 where L is assigned the q_k value of the victim object that is replaced. It is used to define function f of GDS as follows, $f(K_i, DZ_i) = \frac{C_i}{S_i} + L$.

Table 10 shows the cache and byte hit rates with GDS. It includes a variation of GDS that maintains the average $\bar{q}$ value of the objects in the cache and updates the value of those objects with a q_i value lower than $\bar{q}$. This variation is named GDS:$\bar{q}$. When compared with GDS, it improves the service time of COSAR without degrading its performance when considering other metrics. It is most beneficial with larger cache sizes because a larger fraction of cache resident objects have q_i values greater than $\bar{q}$.

We analyzed a variant of GDS:$\bar{q}$ that includes the following admission control policy: When a new object is inserted, admit it only if its q_i value is greater than L, i.e. the q value of the last evicted object. This variant is named GDS:$\bar{q}$:AC. With the Constant cost model, GDS:$\bar{q}$:AC shows significant degradation because it is inconsistent with the design of the algorithm. However, with non-uniform cost models (Exponential and PacedExp), this variant provides comparable (if not better) performance. This is shown in Table 11 where we report on the percentage improvement in total DBMS execution time observed with GDS:$\bar{q}$:AC when compared with GDS, $100 \times \frac{C_{GDS} - C_{GDS:\bar{q}:AC}}{C_{GDS}}$. A negative number means GDS:$\bar{q}$:AC is inferior to GDS. These results show GDS:$\bar{q}$:AC to be slightly better than GDS in most cases with non-uniform cost models. This is important because the execution time of GDS:$\bar{q}$:AC is several times faster than GDS, almost identical to those shown in Figure 10.

With an evolving pattern of access, GDS:$\bar{q}$:AC adapts when changes are gradual and slow, i.e. small Δ values and large α values. It is less adaptive with changes that are drastic, i.e., either large Δ values, small α values, or both, see Table 12. GDS and GDS:$\bar{q}$ adapt more readily.

Cache Size (MB)	Constant Cost Model	Exponential Cost Model	PacedExp Cost Model
32	-4%	-2%	0%
64	-5%	0%	2%
128	-5%	3%	6%
256	-6%	7%	14%
512	-8%	6%	23%
1024	-15%	0%	12%
1536	-18%	0%	-5%

Table 11. Percentage improvement observed with GDS:$\bar{q}$:AC when compared with GDS. A negative value means GDS:$\bar{q}$:AC is inferior to GDS.

We also investigated a variant of GDS that considers recency of references in its cost function, $f(K_i, DZ_i) = \frac{TS_i \times C_i}{S_i} + L$. Its incurred costs and execution time is almost identical to GDS:$\bar{q}$. When this variant is extended with an admission control, it behaves almost the same as GDS:$\bar{q}$:AC, namely, it provides a fast execution time at the expense of failing to adapt to drastic changes in access patterns.

In summary, while the execution time of GDS:$\bar{q}$:AC is impressive, we find GDS:$\bar{q}$ to be more appropriate because its incurred costs are as competitive as GDS:$\bar{q}$:AC and it adapts more readily to evolving access patterns.

7 A Comparison of alternative replacement techniques

Table 15 provides a summary of the different techniques and their tradeoffs. This section focuses on those techniques that adapt to drastic changes in access pattern. We believe this is an important property because most organizations are unaware of the patterna of access to their data and whether it changes drastically.

Amongst the candidate techniques (all except for LRU-2:CA:AC and GDS:$\bar{q}$:AC), LRU:CA:AC provides the best observed COSAR execution times. For example, with 512 MB of memory, LRU:CA:AC is several times faster than GDS:$\bar{q}$ and LRU-2:CA, compare Table 4 with Tables 5 and 10. This difference becomes larger as we increase the average object size ($\bar{S}$) from 1K to 10K. LRU-2:CA and GDS:$\bar{q}$ are more than 4 times slower than LRU:CA:AC with $\bar{S}$=10K and approximately 2.8 times slower with $\bar{S}$=1K. This is because LRU-2:CA and GDS:$\bar{q}$ write larger objects into the cache with each insertion. The admission control of LRU:CA:AC does not insert all the objects in the cache,

Cache Size (MB)	Cache Hit Rate			Byte Hit Rate			COSAR Execution Time (Seconds)		
	GDS	GDS:$\bar{q}$	GDS:$\bar{q}$:AC	GDS	GDS:$\bar{q}$	GDS:$\bar{q}$:AC	GDS	GDS:$\bar{q}$	GDS:$\bar{q}$:AC
32	17.30	17.39	13.88	13.17	13.34	7.78	752	745	150
64	23.57	23.70	19.63	18.42	18.66	11.51	723	721	175
128	31.92	32.09	28.21	25.65	25.99	17.70	695	685	206
256	42.76	42.93	39.23	35.58	35.95	26.68	679	653	255
512	52.19	52.29	47.89	44.77	45.05	34.15	646	615	295
1024	69.51	69.67	63.95	63.30	63.57	51.66	583	547	353
1536	80.06	81.47	75.81	76.15	77.91	67.60	511	465	376

Table 10. A comparison of GDS, GDS:$\bar{q}$ and GDS:$\bar{q}$:AC with the Constant cost model, $\bar{S}$=1K.

Cache Size (MB)	Δ =0 α=0			Δ =1 α=1K			Δ=10K α=100K		
	GDS	GDS:$\bar{q}$	GDS:$\bar{q}$:AC	GDS	GDS:$\bar{q}$	GDS:$\bar{q}$:AC	GDS	GDS:$\bar{q}$	GDS:$\bar{q}$:AC
32	1,992,680	1,986,030	2,041,810	2,002,890	1,997,910	2,276,380	2,071,060	2,065,250	2,570,150
64	1,722,780	1,719,210	1,724,430	1,721,590	1,717,490	1,908,040	1,816,130	1,810,670	2,383,220
128	1,372,560	1,370,300	1,337,350	1,374,160	1,371,860	1,428,180	1,491,900	1,489,410	2,040,380
256	931,466	931,778	866,725	931,857	932,055	913,878	1,053,650	1,053,920	1,433,060
512	586,488	586,557	551,742	588,205	588,956	573,071	699,662	700,211	853,889
1024	312,325	310,154	311,198	312,832	310,444	312,864	331,928	330,217	351,871
1536	288,622	287,689	288,069	288,771	287,974	288,485	298,338	296,317	299,637

Table 12. Total cost (DBMS execution time in Seconds) of GDS, GDS:$\bar{q}$ and GDS:$\bar{q}$:AC with the Exponential cost model and varying access pattern, $\bar{S}$=1K.

outperforming the other techniques by a wider margin as we increase $\bar{S}$. This difference holds true with different patterns of access to the data, i.e. different μ values for the Zipfian distribution.

Second, LRU-2:CA is the best technique when considering the incurred costs, i.e. DBMS execution times. It minimizes incurred costs several folds with large object sizes ($\bar{S}$=10K). This difference decreases with smaller object sizes ($\bar{S}$=1K) and more skewed access patterns. The explanation for this is as follows. There are fewer objects with $\bar{S}$=10K when compared with $\bar{S}$=1K because the database size is fixed. (There are approximately ten times fewer objects.) This magnifies the significance of LRU-2:CA making better caching decisions with larger $\bar{S}$ values.

All three techniques adapt to evolving access patterns. With drastic changes in access patterns (large Δ values), LRU:CA:AC and GDS:$\bar{q}$ adapt quicker than LRU-2:CA, minimizing the incurred costs. The difference between LRU:CA:AC (GDS:$\bar{q}$) and LRU-2:CA ranges from a few percentage to a factor of two. It is smallest with the Exponential cost model and become larger with PacedExp[6].

The average service time of a system consists of COSAR and DBMS execution times. If DBMS execution time dominates then one may minimize it by deploying COSAR configured with LRU-2:CA. However, if the execution time of COSAR dominates[7] then COSAR should be configured with LRU:CA:AC because it is faster than LRU-2:CA.

8 A Comparison with memcached

This section presents a comparison of memcached with COSAR. We focus on version v1.2.6 [3] of memcached server, the latest version that runs on the Windows operating system. This and other versions of memcached implement the LRU replacement policy. There are several providers of memcached client and we used the one provided by Tangent [28].

We use the non-uniform cost models (Exponential and PacedExp) to compare COSAR with memcached. This is because LRU:CA:AC of COSAR reduces to LRU with the Constant cost model, see Section 4.

Table 13 shows the incurred costs with COSAR and memcached for different cache sizes when the average object size is 10K, $\bar{S}$=10K. The numbers shown in parentheses pertain to the percentage difference between COSAR and memcached, $100 \times \frac{C_{memcached} - C_{COSAR}}{C_{COSAR}}$, with a negative number implying COSAR is inferior to memcached.

[6]This difference is largest with the Constant cost model.

[7]A service provider may deploy COSAR with a large amount of memory to minimize accesses to the DBMS.

Cache Size (MB)	Exponential Cost Model (Seconds)		PacedExp Cost Model (Seconds)	
	COSAR	memcached	COSAR	memcached
32	1,796,640 (19.1%)	2,138,940	176,142 (14.9%)	202,316
64	1,452,450 (33.7%)	1,941,570	145,956 (25.2%)	182,745
128	997,470 (67.9%)	1,674,680	106,536 (47.9%)	157,538
256	582,540 (127.9%)	1,327,490	79,132 (57.8%)	124,882
512	400,323 (111.7%)	847,601	67,051 (18.9%)	79,695
1024	85,110 (66.1%)	141,399	15,490 (-13.6%)	13,376
1536	29,268 (-0.1%)	29,244	2,792 (-1.2%)	2,759

Table 13. Total cost (DBMS execution time) with COSAR and memcached using non-uniform cost models, $\bar{S}$=10K. Numbers in parentheses are the percentage difference between COSAR and memcached, $100 \times \frac{C_{memcached} - C_{COSAR}}{C_{COSAR}}$.

COSAR performs significantly better with cache sizes of 512 MB and smaller. This is because its replacement policy (LRU:CA:AC) stages those objects with the highest costs that are likely to be accessed repeatedly in the cache. The standard LRU policy used by memcached cannot do the same.

The case for LRU:CA:AC becomes even more convincing when one considers the utilization of the available cache space. Berkeley DB, the key-value store of COSAR, utilizes approximately 50% of the cache space while the main memory key-value store of memcached utilizes 80% of the cache space. This is reflected in the higher cache and byte hit rates observed with memcached, see Table 14. By considering both the cost and recency of references, LRU:CA:AC enables COSAR to minimize costs, outperforming memcached with 30% less memory for cache sizes smaller than 512 MB.

With larger memory sizes, both COSAR and memcached cache a larger number of objects. With memcached, this reduces the cache miss rate for all objects including those with a high cost. Its 30% higher utilization of cache space enables memcached to outperform COSAR. With cache sizes larger than 1.25 GB of memory, all inserts into memcached succeed, making the reported cache and byte hit rates reach their theoretical[8] upper bound.

The reported execution times for COSAR and memcached in Table 14 are not comparable. This is because while memcached uses the socket layer to communicate between the client and server processes running on the same PC, COSAR is a collection of libraries that are linked with the client software and incur no message passing overhead.

Note that the reported percentage differences between COSAR and memcached in Table 13 are conservative due to our use of LRU:CA:AC. They are higher when COSAR is configured with either LRU-2:CA or GDS:$\bar{q}$. For example, with 512 MB of memory and the Exponential cost model, COSAR configured with LRU-2:CA provides more than 600% reduction in total DBMS execution time when compared with memcached. This is because, as shown in Table 15, LRU-2:CA and GDS:$\bar{q}$ are superior to LRU:CA:AC when minimizing total DBMS execution time.

[8]Cache and byte hit rates do not reach 100% because each experiment starts with an empty cache.

9 Conclusions and future research directions

This paper presents the design and implementation of COSAR, a Précis cache server. COSAR supports a variety of replacement techniques that consider the cost of obtaining a cached object. Table 15 shows a summary of the alternative techniques and their tradeoffs. Our comparison of COSAR with memcached provides additional motivation for the proposed replacement techniques.

In the near future, we are extending COSAR in several ways. First, we are replacing Berkeley DB as the storage manager of COSAR with Everest [11] as the transactional properties of Berkeley DB are not required by COSAR. Moreover, Everest prevents fragmentation of memory and enhances its utilization. Our objective is to approximate the 80% utilization observed by memcached without sacrificing other performance metrics.

Second, we are investigating the role of mass storage devices and their tradeoffs in the context of Précis caches. The current software release of COSAR supports a mass storage device (such as either a magnetic or flash disk) as an extension of memory, victimizing objects from memory by writing them to its disk(s). We are developing a framework that enables a developer to decide whether COSAR should use its mass storage device or ignore it by reporting a cache miss to the application. The key tradeoff is the overhead of retrieving the referenced object from the local storage of COSAR versus invoking the necessary software (that issues

Cache Size (MB)	Cache Hit Rate		Byte Hit Rate		Execution Time (Seconds)	
	COSAR	memcached	COSAR	memcached	COSAR	memcached
32	9.61	22.90	9.01	15.42	325	1,329
64	13.08	30.09	12.46	24.54	298	1,237
128	17.72	39.73	17.08	35.34	307	1,275
256	22.20	52.23	21.55	49.07	289	1,244
512	24.62	69.49	23.97	67.69	222	1,142
1024	48.04	94.90	47.32	94.41	327	1,065
1536	98.70	98.95	98.71	98.95	535	1,007

Table 14. A comparison of memcached with COSAR (LRU:CA:AC) using the Exponential cost model, $\bar{S}$=10K.

Technique	Design details		Behavior		
	f	Admission control	Adapts to drastic changes in access patterns	COSAR execution time	DBMS execution time
LRU	T_1	No	Yes	Bad	Bad
LRU:CA	$T_1 \times C_i$	No	Yes	Bad	Very Good
LRU:CA:AC	$T_1 \times C_i$	Yes	Yes	Very Good	Very Good
LRU-2	$\frac{T_1+T_2}{2}$	No	Yes	Good	Bad
LRU-2:CA	$\frac{T_1+T_2}{2} \times C_i$	No	Yes	Not Bad	**Excellent**
LRU-2:CA:AC	$\frac{T_1+T_2}{2} \times C_i$	Yes	No	**Excellent**	Not Bad
GDS	$\frac{C_i}{S_i} + L$	No	Yes	Not Bad	Very Good
GDS:$\bar{q}$	$\frac{C_i}{S_i} + L$	No	Yes	Good	Very Good
GDS:$\bar{q}$:AC	$\frac{C_i}{S_i} + L$	Yes	No	**Excellent**	Sometimes Good

Table 15. Summary of alternative techniques and their tradeoffs.

DBMS queries) to re-compute it.

Third, the performance study of this paper focused on the service time. Our implementation of COSAR is thread-safe, supporting multiple threads executing on a multicore CPU simultaneously. We intend to study the throughput of COSAR and its scalability as a function of the number of cores.

References

[1] R. A. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The impact of caching on search engines. In *SIGIR*, pages 183–190, 2007.

[2] P. Cao and S. Irani. Cost-Aware WWW Proxy Caching Algorithms. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97)*, Monterey, CA, 1997.

[3] Jellycan Code. memcached v1.2.6, July 25, 2009, http://code.jellycan.com/memcached/.

[4] C. D. Cuong. YouTube Scalability. Google Seattle Conference on Scalability, June 23 2007.

[5] A. Daskos, S. Ghandeharizadeh, and R. Shahriari. SkiPeR: A Family of Distributed Range Addressing Spaces for Peer-to-Peer Systems. In *15th International Conference on Software Engineering and Data Engineering*, July 2006.

[6] 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. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220. ACM Press, 2007.

[7] A. Dingle and T. Partl. Web Cache Coherence. In *In Fifth International WWW Conference*, 1996.

[8] T. Fagni, R. Perego, F. Silvestri, and S. Orlando. Boosting the Performance of Web Search Engines: Caching and Prefetching Query Results by Exploiting Historical Usage Data. *ACM Trans. Inf. Syst.*, 24(1):51–78, 2006.

[9] B. Fitzpatrick. Distributed Caching with Memcached. *Linux Journal*, 2004(124), 2004.

[10] Q. Gan and T. Suel. Improved Techniques for Result Caching in Web Search Engines. In *WWW*, pages 431–440, 2009.

[11] S. Ghandeharizadeh, D. Ierardi, and R. Zimmermann. An Online Algorithm to Optimize File Layout in a Dynamic Environment. *Information Processing Letters Journal*, 57:75–81, 1996.

[12] S. Ghandeharizadeh and S. Shayandeh. Greedy Cache Management Techniques for Mobile Devices. In *First International IEEE Workshop on Ambient Intelligence, Media, and Sensing (AIMS)*, 2007.

[13] S. Ghemawat, H. Gobioff, and S. Leung. The Google File System. In *SOSP '03: Proceedings of nineteenth ACM SIGOPS symposium on Operating systems principles*. ACM Press, 2003.

[14] J. Gwertzman and M. Seltzer. World-Wide Web Cache Consistency. In *In Proceedings of the 1996 USENIX Technical Conference*, pages 141–151, 1996.

[15] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 439–450, 1994.

[16] A. Keller and J. Basu. A Predicate-Based Caching Scheme for Client-Server Database Architectures. *VLDB Journal*, 5(1):35–47, 1996.

[17] R. Lempel and S. Moran. Predictive Caching and Prefetching of Query Results in Search Engines. In *WWW*, pages 19–28, 2003.

[18] C. Liu and P. Cao. Maintaining Strong Cache Consistency in the World-Wide Web. In *In Proceedings of the Seventeenth International Conference on Distributed Computing Systems*, pages 445–457, 1998.

[19] A. Luotonen and K. Altis. World-Wide Web Proxies. *Computer Networks and ISDN Systems*, 27(2):147–154, 1994.

[20] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *FAST*. USENIX, 2003.

[21] M. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 183–191, 1999.

[22] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, 1993.

[23] Saab P. Scaling memcached at Facebook, http://www.facebook.com/note.php?note_id=39391378919, December 2008.

[24] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A Scalable Content-Addressable Network. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 161–172, August 2001.

[25] P. C. Saraiva, E. de Moura, R. C. Fonseca, W. Meira Jr., B. A. Ribeiro-Neto, and N. Ziviani. Rank-preserving two-level caching for scalable search engines. In *SIGIR*, pages 51–58, 2001.

[26] J. Shim, P. Scheuermann, and R. Vingralek. Proxy Cache Algorithms: Design, Implementation, and Performance. *IEEE Transactions on Knowledge and Data Engineering*, 11(4), 1999.

[27] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, pages 149–160, San Diego, California, August 2001.

[28] Tangent. memcached client, http://tangent.org/552/libmemcached.html.

[29] Y. Xie and D. R. O'Hallaron. Locality in Search Engine Queries and Its Implications for Caching. In *INFOCOM*, 2002.

[30] C. Yang, K. Y. Lee, Y. D. Chung, M. Kim, and Y. Lee. An Effective Self-Adaptive Admission Control Algorithm for Large Web Caches. *IEICE Transactions*, 92-D(4):732–735, 2009.

[31] N. E. Young. On-line Caching as Cache Size Varies. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1991.