LEARNING ON DISTRIBUTED TRACES FOR DATA CENTER STORAGE SYSTEMS

Giulio Zhou [*1,2] **Martin Maas** [1]

ABSTRACT

Storage services in data centers continuously make decisions, such as for cache admission, prefetching, and block allocation. These decisions are typically driven by heuristics based on statistical properties like temporal locality or common file sizes. The quality of decisions can be improved through application-level information such as the database operation a request belongs to. While such features can be exploited through application hints (e.g., explicit prefetches), this process requires manual work and is thus only viable for the most tuned workloads.

In this work, we show how to leverage application-level information automatically, by building on distributed traces that are already available in warehouse-scale computers. As these traces are used for diagnostics and accounting, they contain information about requests, including those to storage services. However, this information is mostly unstructured (e.g., arbitrary text) and thus difficult to use. We demonstrate how to do so automatically using machine learning, by applying ideas from natural language processing.

We show that different storage-related decisions can be learned from distributed traces, using models ranging from simple clustering techniques to neural networks. Instead of designing specific models for different storage-related tasks, we show that the same models can be used as building blocks for different tasks. Our models improve prediction accuracy by 11-33% over non-ML baselines, which translates to significantly improving the hit rate of a caching task, as well as improvements to an SSD/HDD tiering task, on production data center storage traces.

1 INTRODUCTION

Modern data centers contain a myriad of different storage systems and services, from distributed file systems (Howard et al., 1988; Ghemawat et al., 2003; Weil et al., 2006; Shvachko et al., 2010), to in-memory caching services (Fitzpatrick, 2004) and databases (Stonebraker & Kemnitz, 1991; Corbett et al., 2013). These services typically operate behind an RPC abstraction and are accessed by workloads that are composed of interconnected services communicating through RPCs (Gan et al., 2019a).

Storage services continuously make *decisions* that aim to optimize metrics such as cache hit rate or disk footprint. To make these decisions, the systems need to make *predictions* about future workload and system behavior. For example, caches admit objects based on their likelihood of future access (Beckmann & Sanchez, 2017; Jaleel et al., 2010), and block allocators reduce fragmentation by colocating allocations of comparable lifetime (Kim et al., 2018).

Traditionally, storage systems rely on heuristics for these decisions, such as LRU replacement policies or best-fit allocation. These heuristics exploit statistical workload properties like temporal or spatial locality, but are unable to leverage application-level signals, such as whether or not a request belongs to a temporary file. While systems can communicate hints to the storage system (e.g., using prefetch commands or non-temporal stores), manually assigning these hints is brittle, work-intensive and incurs technical debt. As such, they are most commonly used in highly tuned workloads. To apply such optimizations to the long tail of data center workloads (Kanev et al., 2016), we need to automate them.

We observe that in many cases, high-level information is already available in the system, as part of distributed tracing frameworks (Sigelman et al., 2010; Barham et al., 2003) and resource managers (Park et al., 2018; Cortez et al., 2017) that are widely deployed in data centers. Distributed traces, job names, permission names, etc. are generated automatically as part of the system's regular operation and encapsulate human-defined structure and information.

In this work, we are looking at a specific instance of this approach using a deployed production distributed tracing framework, Census (OpenCensus, 2020). Census provides a tagging API that applications use to generate arbitrary

* Work was done while at Google. [1]Google, Mountain View, CA, USA [2]Carnegie Mellon University, Pittsburgh, PA, USA. Correspondence to: Martin Maas <mmaas@google.com>.

Proceedings of the 4th MLSys Conference, San Jose, CA, USA, 2021.

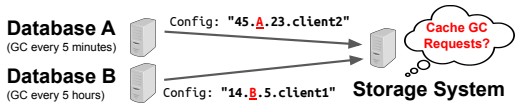


Figure 1. Distributed tracing tags contain unstructured application information that can be leveraged to make storage predictions.

key-value pair strings (Census Tags) that are automatically propagated with outgoing requests. These tags can be used to understand complex workload interactions, and for resource accounting. A side effect is that incoming requests now come with rich context that encodes the path taken to the storage system, which the system can leverage.

However, this data does not always have an explicit schema or directly encode the information required by the storage system. For instance, consider a hypothetical example of databases with two different configurations A and B, which are listed in a configuration string attached to each storage request (Figure 1). A has a garbage collection interval of 5 minutes while B has 5 hours. A caching service could leverage this information by only caching requests from the service with configuration A. However, this information is not readily available: The service needs to know that it needs to check the configuration string for the presence of A or B in a particular location, and which requests to drop.

Instead of explicitly encoding these rules, we learn them from historical trace data. We present several techniques, ranging from lookup tables to neural networks that leverage recent progress in natural language processing. A key challenge is that models become stale over time and do not transfer to new settings (e.g., a new storage system or cluster). The reason is that the model jointly has to learn 1) how to extract information from distributed traces and 2) how this information translates to predictions in a storage system. If the storage system changes, both need to be relearned from scratch. We therefore introduce a model that can be used in a multi-task learning setting, where a model can be used as a building block in different task-specific models.

We make the following contributions: 1) We demonstrate the connection between distributed tracing and storage-layer prediction tasks (Section 2) and show that strong predictive performance relies on leveraging the latent structure of unstructured distributed traces (Section 3). 2) We show that several important (and traditionally separate) storage tasks - such as cache admission/eviction, file lifetime, and file size prediction - can be learned by the same models from application-level features. 3) We present models of increasing complexity (Section 4) that represent different deployment strategies, and analyze their trade-offs. 4) We show that our models are robust to workload distribution shifts and improve prediction accuracy by 11-33% over non-ML baselines, for a range of storage tasks, improving both a caching and an SSD/HDD tiering task substantially in simulations based on production traces (Section 6).

2 Background & Related Work

Data Center Storage Systems. We use a broad definition of what constitutes a storage system. We consider any service within a warehouse-scale computer that holds data, either on persistent storage (e.g., databases, distributed file systems) or in-memory (e.g., key-value stores). Such services exist at different levels of the storage stack: managing physical storage (e.g., storage daemons in Ceph (Weil et al., 2006) or D file servers (Serenyi, 2017)), running at the file system level (e.g., HDFS (Shvachko et al., 2010)) or storing structured data (e.g., Bigtable (Chang et al., 2008)). One storage system may call into another. We assume that requests to the service are received as RPCs with attached metadata, such as the request originator, user or job names, priorities, or other information.

Prediction in Storage Systems. Storage systems employ various forms of prediction based on locally observable statistics. These predictions are often implicitly encoded in the heuristics that these systems use to make decisions. For example, LRU caches make admission decisions by implicitly predicting diminishing access probability as the time since previous access increases, while FIFO-TTL caches evict objects assuming a uniform TTL.

While such policies can model a broad range of workload patterns, they have limitations. First, a single heuristic may have difficulties modeling a mixture of workloads with different access properties, which it is more likely to encounter in warehouse-scale computers where storage services receive a diverse mix of requests resulting from complex interactions between systems. Second, they are limited by their inability to distinguish between requests based on application-level information. For example, while traditional caching approaches can distinguish between read and write requests, they do not typically distinguish based on what application-level operation the request corresponds to.

Application-Level Information in Systems. Using high-level features in storage systems is not a new idea. However, most mechanisms require that the application developer explicitly provides hints. A less explored alternative is to extract such high-level information from the application itself. Recent work has demonstrated a similar approach for cluster scheduling (Park et al., 2018), by predicting a job's runtime from features such as user, program name, etc. While cluster schedulers have a host of such features available at the time of scheduling a job, exploiting such information in storage systems is more challenging, since the predictive features are not readily available. For example, a storage request may be the result of a user contacting a front-end server, which calls into a database service, which runs a query and in turn calls into a disk server. Features may have been accumulated anywhere along this path.

The same challenges that make it difficult to reason about storage requests make it difficult to monitor, measure and debug distributed systems in general. For this reason, data centers have long employed distributed tracing frameworks (Sigelman et al., 2010; Barham et al., 2003), which track the context of requests between systems. Distributed traces thus present an opportunity to leverage application-level features for predicting workload behavior. Unfortunately, the data gathered by these systems can be difficult to exploit, due to its high dimensionality and unstructured nature.

ML for Data Center Systems. The promise of ML for storage-related tasks is its ability to learn useful representations from large amounts of unstructured data. For example, Gan et al. (2019b) showed that it is possible to use traces to predict QoS violations before they occur. Different techniques have been proposed. For example, cheap clustering techniques (Park et al., 2018; Cortez et al., 2017) and collaborative filtering (Delimitrou & Kozyrakis, 2014) have been shown to work well for cluster scheduling while the aforementioned work on distributed traces relies on LSTM neural networks (Hochreiter & Schmidhuber, 1997). It is important to distinguish between ML for predictions/forecasting and ML for decision making. Prior work on applying ML to storage systems has sought to optimize the latter, such as learning caching policies (Kirilin et al., 2019; Song et al., 2020; Liu et al., 2020); these works improve upon heuristics while using conventional features. In contrast, we use ML to enable the use of more complex, application-level features.

Connection to Multi-Task Learning. Storage systems in data centers feature a wide range of settings. For example, caches within different systems behave differently, which means that their predictions differ as well. Decisions can also differ across database instances, or based on the hardware they run on. As a result, prediction in storage systems does not require training a single model but a myriad of them. Some systems even need to make multiple decisions simultaneously (e.g., lifetime and file size on file creation). This indicates that it is beneficial to share models between tasks, an approach known as multi-task learning (MTL).

There has been a large amount of work on MTL. Many advances in areas such as natural language processing and computer vision have come from using large amounts of data to learn general models that transfer better to new tasks. Among NLP's recent successes are the learning of general Transformer models (Vaswani et al., 2017), such as GPT-2 (Radford et al., 2019) and BERT (Devlin et al., 2018).

Usually, MTL datasets do not have a complete set of labels for each task, but are often multiple datasets (with possibly disjoint task labels) that share a common input feature space. As such, MTL is a natural fit for learning multiple storage tasks from shared distributed traces.

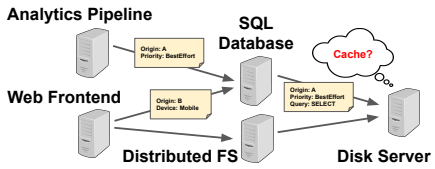


Figure 2. Census tags are free-form key-value pairs that are added by services and propagated with subsequent requests.

3 Workload Analysis with Census

In this section, we describe how the information contained in distributed traces relates to prediction tasks in storage systems, and analyze their statistical properties.

3.1 Distributed Traces and OpenCensus

Data center applications are composed of services that communicate via message passing (Gan et al., 2019a). Services often have multiple clients using them (e.g., different services accessing a database) and rely on multiple different downstream services (e.g., the database service might connect to storage servers and an authentication service). This makes analysis and resource accounting challenging: Should a request to a database be attributed to the database or one of the upstream services using it?

Census (OpenCensus, 2020) is a distributed tracing library that provides insights into such systems. It tags requests as they travel through the system (Figure 2). Census allows services to set key-value pairs (*Census Tags*) that are automatically propagated and allow a service to determine the context of each request that it receives (e.g., for resource accounting). These tags are set through an API by the service developers themselves, while being oblivious to tags added by downstream services. One of the insights of our work is that this same information represents powerful features for reasoning about the distributed system: Existing tags already capture properties of the workload that a programmer deemed important, and programmers could select new tags based on what they believe would be predictive features.

Some examples of Census tags are shown in Table 1 (obfuscated but based on real values). While this captures some common cases, this list is not exhaustive. Some Census Tags have low cardinality (they either take on a small number of values or their main information is in their presence), while others (such as transaction IDs, jobs or table names) have very large cardinality (sometimes in the tens of thousands). A human could sometimes manually write a regular expression to extract information (e.g., "[redacted]Table" might be split by "." and "-" characters and the first entry refers to a user group), but as Census tags are maintained by services themselves, there is no guarantee that they are not going to change. For storage services to make assumptions on any particular structure of these tags is inherently brittle.

Key	Example Values	Cardinality	Description
[redacted]Call	COMPACT(MINOR) | COMPACT(MAJOR) | BACKUP	Low	Request is from a particular DB operation
[redacted]CacheHit	hit | miss	Low	Whether a request was cached
[redacted]_action	EditQuery | GetIds | UpdateConfig	Medium	A particular operation from a service
user_id *	[redacted]-pipeline-services-low | [redacted]-jobs | local_[redacted]	High	A particular user group (free form)
JobId *	datacenterABC.client-job-5124.v521aef *	High	A particular job name (free form)
[redacted]Table	[redacted]-storage.[redacted]-local-[redacted].[redacted].[redacted].4523.index	High	Name of a table a query applies to
[redacted]TxnTag	AsyncService-Schedule-[redacted] | DELETE-LABEL | EDIT-[redacted]	High	Free form description of an operation

Table 1. Examples of Census tag features found in production distributed traces (adapted* and/or obfuscated).

We also noticed that the same tag does not always follow the same schema. For example, the "[redacted]TxnTag" shows a mix of different schemas depending on which service set the tag. Other tags feature a mix of capitalized/non-capitalized values, different characters to delineate different parts of the string, etc. This high degree of variance makes it difficult to manually extract information from these tags consistently.

3.2 Prediction Tasks

We now present prediction problems in storage systems that can benefit from high-level information.

File access interarrival time for caching. Predicting the time of the next access to an entry allows a cache to decide whether to admit it (Beckmann & Sanchez, 2017; Jaleel et al., 2010), and which block to evict (e.g., a block with a later time). We focus on caching fixed 128KB blocks in a production distributed file system, which is either accessed directly or used as a backing store for other storage systems, such as databases. We ignore repeated accesses under 5 seconds, to account for local buffer caches.

File lifetime until deletion. File lifetime predictions are used in different contexts (Kim et al., 2018). They are used to select storage tiers (e.g., for transient files) and can be used to reduce fragmentation. For example, some storage technologies have large erase units (e.g., SMR, NAND flash) and some storage systems are append-only (Chang et al., 2008; Ousterhout et al., 2015). Placing data with similar lifetimes into the same blocks minimizes wasted bytes, write amplification, and compaction.

Final file size. Knowing the final size of a file at the time it is allocated can improve allocation decisions. For example, it can help disk allocators pick the best block size.

Read/write ratio. Predicting the ratio of read vs. write operations is helpful for placing data. Read-only files may be candidates for additional replication while write-only files may be best stored in a log structure. This prediction can also help pick a storage medium (Section 6.4).

This list is not exhaustive. Other tasks that are not explored in this paper include 1) Resource demand forecasting when deploying a new mix of workloads (e.g., when bringing up a new cluster of machines), by recording a small number of samples characterizing the mix of workloads and then using a model to extrapolate the overall usage, and 2) Predicting workload interference (e.g., because both are I/O heavy).

3.3 Analyzing Census Tag Predictiveness

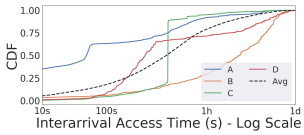
After introducing the prediction tasks, we now demonstrate that Census tags are predictive of some of these tasks.

Dataset. We analyze Colossus (Serenyi, 2017) file system traces sampled from 5 different clusters at Google. The data is longitudinally sampled at a per-file granularity. Our traces contain over a trillion samples per cluster and we are analyzing traces from a period of three years. The clusters contain different proportions of various workload types. All our requests are sampled at the disk servers backing the distributed file system and contain file metadata as well as Census Tags associated with each request. Note that these disk servers back other storage services, such as databases.

Features. Broadly, the features provided through Census Tags fall into four categories: 1) Census tags that indicate a particular category of request (e.g., a DB operation), 2) numerical information (e.g., an offset), 3) medium and high cardinality labels that can contain unstructured data (e.g., project IDs, table names, etc.) and 4) high cardinality labels that may or may not be predictive (e.g., timestamps or transaction numbers). We are interested in the predictiveness of these features. Note that there is information about requests that these features do not capture. For example, we only consider one request at a time.

We can phrase our prediction problems as follows: Given a set of Census Tags and their associated values $X = \{x_1, x_2, \ldots, x_n\}$ where x_i is the ith (unordered) key-value string pair, predict a label Y (e.g., interarrival time, lifetime, etc.). We refer to X as a *Census Tag Collection (CTC)*.

Entropy Analysis. To measure the predictive ability of Census Tags, we look at the distribution of values for each of the tasks we aim to predict (e.g., interarrival times) and compare this distribution to the same distribution conditioned on different values of particular Census Tags. Figure 3a shows an example where we conditioned the distribution of interarrival times on a particular Census tag "[redacted]Key", which describes what type of operation a request belongs to. We show distributions for four arbitrary values of this tag.



(a) Overall and per-tag value interarrival time CDFs.

Task	Entropy	Cond.
Interarrival Time	4.748	3.474
File Lifetime	7.303	6.575
Final File Size	3.228	2.538
R/W Fraction	1.874	1.476

(b) Per-task entropy, overall and avg. conditioned across all tags.

Figure 3. Conditioning the distribution on Census tags significantly reduces entropy, indicating that they are predictive.

There is a visible difference between the distributions depending on the specific value, and the average distribution (the dotted line) captures neither of them. A way to measure this effect more formally is by computing the information entropy of the overall distribution (shown in Figure 3b) and compare it to the conditional entropy (the weighted average over the entropies when conditioning on Census tag values). The difference between the two is known as the mutual information (or information gain), which measures the predictiveness of Census Tag collections for the distribution.

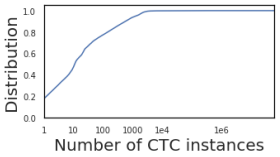
Transferability of Census Tags. In order to use Census tags in predictions, we need to show that the information they provide transfers – i.e., values recorded in one setting can be used to predict values in a different setting. We are interested in two particular types of transferability:

1. **Across time:** We want to be able to use past traces to make predictions months or years in the future.
2. **Across clusters:** We want to use traces recorded in one cluster to make predictions in other clusters.

For predictions to be transferable, traces must either share features or have a similar latent structure (e.g., there exists a similar relationship between the keys and values even if they are named differently). To analyze transferability, we conducted a study comparing the keys and values found in two different clusters and between traces 9 months apart (Figure 4a). We find that 1) only a small fraction of requests have a CTC that occurs exactly in the original trace, but 2) most requests have at least one key-value pair that was seen in the original trace. This is true both across time and across clusters, and indicates that an approach that only records CTCs in a lookup table will degrade over time and is of limited use across clusters. Meanwhile, it shows that complex approaches can potentially extract more information.

High-Cardinality Tags. One example of tags that do not transfer directly are high-cardinality keys capturing information that changes over time or between clusters. For example, new accounts or database tables are added over time and different clusters host different workloads. Tags that directly include these identifiers as values will therefore differ. This is visualized in Figure 4b which plots the number of times each CTC is observed in a 1.4B entry trace.

(a) How many CTCs match across time and clusters.



(b) How often each CTC appears in the data set (one cluster).

Figure 4. Transferability (a) and cardinality (b) of Census tags.

18% of CTCs are observed only once and 2/3 of CTCs are observed at most 30 times, pointing to high-cardinality keys.

However, many of these tags can still contain information. For example, a username may be composed of a particular system identifier and a prefix (e.g., "sys_test54" vs. "sys_production") and table names often have hierarchical identifiers (e.g., a format such as "type.subtype.timestamp"). Only using exactly matching strings would therefore lose important information. We need to extract information from within these strings, which resembles natural language processing tasks. Such techniques enable proper information sharing between known values as well as generalization to new values that have not been seen before. Of course, there are also high-cardinality keys that carry little information – e.g., unseen UIDs. This has similarities to ML applied to code (Karampatsis & Sutton; Shrivastava et al., 2020), where tokens are often highly specific to the context in which they appear.

3.4 Distribution-based Storage Predictions

Intuitively, we would expect a predictor for the storage prediction tasks from Section 3.2 to predict one value for Y (e.g., the expected lifetime of a file) given a CTC X. However, there is inherent uncertainty in these predictions: 1) features do not always capture all details in the system that determine the file's lifetime, and 2) effects outside the control of the system, such as user inputs, affect the predictions. For many predictions, there is not a single value that we could predict that is correct most of the time. Similar to the work by Park et al. (2018) on cluster scheduling, we therefore predict a *probability distribution* of values. This distribution can then be consumed by the storage system directly, similar to EVA (Beckmann & Sanchez, 2017): For example, a cache could evict a cache entry with a high variance in its distribution of interarrival times in favor of an entry with low interarrival time at low variance.

To perform distribution-based predictions, data within the traces needs to be pre-aggregated. Specifically, we need to take all entries in the trace with the same X and compute the distributions for each of the labels Y that we want to predict (interarrival times, lifetimes, etc.). To do so, we can collect a histogram of these labels for each CTC. We note a large skew: Some CTCs appear many orders of magnitude

more often than others, and 18% of entries show up only once (Figure 4b). This can be explained by two effects: 1) Some systems account for a much larger fraction of requests than others, and 2) The lower the cardinality of Census tags set by a system, the lower the number of different CTCs associated with this system.

Fitting Lognormal Distributions. One approach to use the histograms for input features is to use them in a lookup table, which covers low-cardinality cases. However, as we have seen, some Census Tags have high cardinality and we therefore need to predict them using models that have the ability to generalize to previously unseen values. For these tags, we therefore need to represent the output distribution in a way that we can train a model against.

Gaussians (or mixtures of Gaussians) are often used to model this type of output distribution. For example, they are used for lifetime prediction in survival analysis (Fernandez et al., 2016). In particular, we consider lognormal distributions and show that they are a suitable fit for our data. They are a popular choice for modeling reliability durations (Mullen, 1998). In contrast to other similar distributions (such as Weibull and log-logistic), the parameters that maximize the lognormal likelihood can be estimated in closed form. Figure 5 shows examples of fitting lognormals to the pre-aggregated distributions for several CTCs. To measure how well the fitted distributions match the real data, we use the Kolmogorv-Smirnov (KS) distance, which measures the maximum deviation between CDFs. The overall KS distance of fitting lognormals to our data is 0.09-0.56.

3.5 Case Study: Database Caching Predictor

We demonstrate the insights from this section using a database system as an example. One Census tag associated with this system indicates the high-level operation associated with it (Figure 3a). This information can be used to make decisions about admission and eviction. For example, consider values A, C and D of this particular Census tag. While the average (dotted) CDF of interarrival times for requests increases slowly (note the log scale), indicating that the interarrival time is difficult to predict, requests with values A/C/D are more predictable: The vertical jump in C shows that 3/4 of requests with this tag have an interarrival time of 15 minutes, indicating it has a periodicity of 15 minutes. Meanwhile, we see that 2/3 of requests for A take less than 1 minute before they are accessed again, and for D the same is true for a 5 minutes interval.

We can exploit this information in a caching policy that does not evict these requests for the first 1, 5 and 15 minutes after their last access. Afterwards, we treat them the same as other requests. We can also do something similar for values such as B where the distribution shows that interarrival times

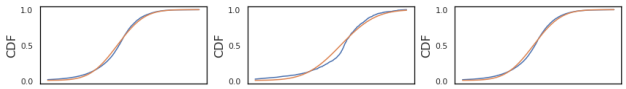


Figure 5. Fitting lognormal distributions to CTC CDFs.

are much longer than for other requests. For example, we could avoid admitting these entries to the cache at all, or prioritize them for eviction.

4 Machine Learning for Census Tags

We now demonstrate a set of learning techniques to achieve transferability across clusters and over time. We assume that all models are compiled and directly linked into the storage server, running either on the CPU, or on an accelerator such as a GPU or TPU. When a request is received by a storage system, its CTC is represented as an unordered set of string key-value pairs. The prediction problem is formally defined as follows: Given a CTC X, predict the parameters of its lognormal distribution for a given task, $Y = (\mu_Y, \sigma_Y)$.

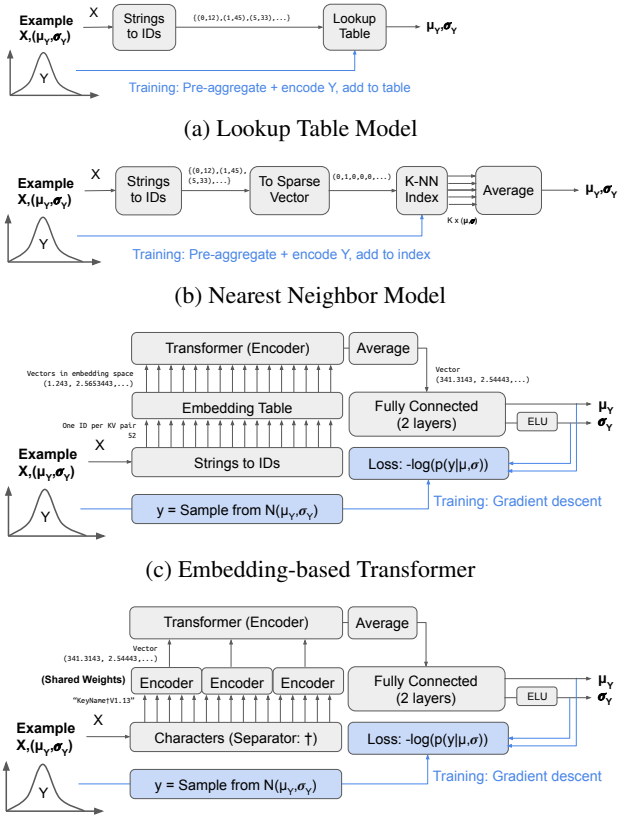
4.1 Lookup Table Model

The simplest prediction approach is a lookup table (Figure 6a) where a canonical encoding of the CTC is used to index a static table that maps CTCs to Y. The table is "trained" by collecting the target distribution histograms from a training set, pre-aggregating them, and computing the mean and standard deviation of a lognormal distribution that fits the data. CTCs are encoded by assigning each unique key and value in the training set an ID and looking them up at inference time. Keys in the CTC are sorted alphanumerically, ensuring that the same CTC always results in the same encoding.

CTCs not found in the table can be handled by substituting the overall distribution of the training set. As shown in Section 3.3, the entropy of this set is much larger than the entropy conditioned on a particular CTC (and is therefore not very predictive), but represents the best we can do. Note that the lookup table can become very large, and it is often necessary to remove rare CTC entries. There are different ways to implement such a lookup table. For example, it could be implemented as a hashtable or represented by a decision tree (Safavian & Landgrebe, 1991), which is an equivalent but potentially more compact representation.

4.2 K-Nearest Neighbor Model

Improving upon how the lookup table handles *unseen* CTCs, the k-nearest neighbor approach (Figure 6b) makes predictions for these entries by combining predictions from CTCs that are close/similar. We implement an approximate k-NN method that uses as its distance metric the number of differing Census Tags between two CTCs. We encode CTCs as a sparse binary vector where each entry denotes whether a



(a) Lookup Table Model

(b) Nearest Neighbor Model

(c) Embedding-based Transformer

(d) Hierarchical Raw Strings Transformer

Figure 6. The different models (blue is training).

particular Census Tag key-value pair is present. Distance can thus be cheaply computed as the squared L2 distance between sparse binary vectors (which can reach a dimensionality of millions). This approach allows us to make use of existing approximate nearest neighbor libraries that are highly optimized for sparse vectors. We choose K=50 in our experiments, since binary vectors may have many neighbors of equal distance. For instance, a CTC that has a different value for one Census Tag may get matched against a number of CTCs that have distance 2. To compute the predictions, we aggregate the chosen nearest neighbors. The mean μ_Y is simply the weighted average over the means of the individual neighbors. The standard deviation σ_Y is computed by summing two components: (1) the weighted average over the variance of each neighbor, and (2) the weighted squared distance between the individual means and the overall mean.

This approach resembles strategies that have been used in cluster scheduling (Park et al., 2018; Cortez et al., 2017). In contrast to a lookup table, it has more generalization ability, but it is still unable to extract information from high-cardinality Census tags. Imagine a tag where values are of format `<query-type>.<timestamp>`. Here, "query type" captures information that we want to extract. Since "timestamp" will take on a different value for every request, each entry will result in a different CTC. This, in turn, means that 1) the lookup table grows very large and 2) each entry only has a single data point associated with it. Instead of a histogram of values, the "distribution" associated with this CTC is therefore a single point mass with $\sigma = 0$. This makes it impossible for the model to generalize, since the nearest neighbor approach has no way of knowing that the different values are identical except for the timestamp.

4.3 Neural Network Model

Handling these high-cardinality cases necessitates a model that can parse the strings that constitute the key-value pair. While a nearest neighbor approach can learn simple connections between predictions and Census tags (e.g., "if tag A has value B, the file is short-lived"), it cannot learn more complex and non-linear interactions (e.g., "if tag A has an even number as value, then the file size is small"). To push the limits of learning these more complex connections, we use a neural network-based approach. Note that in practice, this neural network would not run at every prediction but be used as a fall-back for lookup table entries where no example can be found (and therefore runs rarely).

A simple approach would be to encode keys and values as IDs (similar to the lookup table), feed them into a feed-forward network, and train against the Y from pre-aggregation. However, this approach still has no capability to generalize to unseen values nor high-cardinality keys that only have a single data point associated with them. We address these problems by combining two approaches:

1. We build on recent advances in natural language processing to train networks operating on raw strings. Specifically, we use a Transformer (Vaswani et al., 2017) model that uses an attention mechanism to consume a sequence of inputs (e.g., character strings comprising each Census tag) and maps them to an embedding (i.e., a learned encoding).
2. To handle CTCs with a single point, we do not train against (Y_μ, Y_σ) directly, but use Mixture Density Networks (Bishop, 1994) to let the model fit a Gaussian.

The neural network architecture is based on typical models used in NLP to process character and word tokens. We present two versions: 1) an embedding-based version that resembles the approach above of feeding token-encoded key-value pairs directly into the model, and 2) an approach that parses raw strings of key-value pairs. The model architecture is similar in both cases and relies on learning *embedding representations* (Mikolov et al., 2013), learned mappings from a high-dimensional input to a latent representation in some other – usually lower-dimensional – space.

Embedding-Based Transformer Model. For this version (Figure 6c), we start from a CTC $X = \{x_1, x_2, \ldots, x_n\}$

where x_i is the ith (unordered) key-value string pair – encoded as one-hot encoded vectors based on their IDs – and pass each x_i through a single embedding layer $\phi : \mathbb{N} \rightarrow \mathbb{R}^m$ to create a set of embedding vectors $V = \{\phi(x_1), \phi(x_2), \ldots, \phi(x_n)\}$. V is then passed into the Transformer encoder $M : \mathbb{R}^{n \times m} \rightarrow \mathbb{R}^{n \times m}$ and its output is averaged to produce the shared output embedding $S = \sum_{i=1}^{n} M(Y)_i$ where $S \in \mathbb{R}^m$. Finally, this output embedding is passed through an additional 2-layer fully connected network to yield the outputs $Y = (\mu_Y, \sigma_Y)$. The last layer producing σ uses an ELU activation (specified by Mixture Density Networks).

Hierarchical Raw Strings. This version (Figure 6d) operates directly on raw strings, where each character is encoded as a one-hot vector of dimensionality 128 (127 characters and one special character to separate key and value). Each key-value pair x_i is encoded as a sequence of such one-hot vectors ($x_i \in \mathbb{R}^{k_i \times 128}$), and the characters are passed through an embedding layer, yielding an $\phi(x_i) \in \mathbb{R}^{k_i \times m}$, where k_i is the length of the i-th key-value pair. Each $\phi(x_i)$ is then passed through a Transformer encoder – all these encoders' weights are shared (i.e., this encoder learns how to parse an individual key-value pair). The outputs of these encoders are then passed into another encoder, which now aggregates across the different key-value pairs (i.e., it learns connections between them). As before, the output is then averaged and passed through two fully-connected layers.

Mixture Density Networks. Because the goal is to predict the distribution associated with each CTC, we must choose a loss function that allows the model to appropriately learn the optimal parameters. Consider if we used squared distance to learn the mean and standard deviation of a log-normal distribution. While squared error may be appropriate for learning the mean, it is not for the standard deviation. For instance, squared error is symmetric, and underestimating the standard deviation by 0.1 has a much larger effect on error than overestimating by 0.1. Additionally, a model trained with squared error will not learn the correct standard deviation from an overpartitioned dataset (e.g., if all CTCs had $\sigma = 0$, the model would learn $\sigma = 0$).

Mixture Density Networks (Bishop, 1994) were designed to address this problem. Instead of fitting (μ_Y, σ_Y) directly to the label, the predicted (μ, σ) are used to compute the likelihood that the label y came from this distribution:

$$loss = -\log\left(\frac{1}{\sqrt{2\pi}\sigma} \times \exp\left[-\frac{(y-\mu)^2}{2\sigma^2}\right]\right)$$

Note that now instead of training against a distribution Y, we need to train against a specific label y from this distribution. We therefore sample y from Y at every training step. In high-cardinality cases where $\sigma = 0$, all of these samples will be the same, while in cases where we have enough data points, the samples match the distribution.

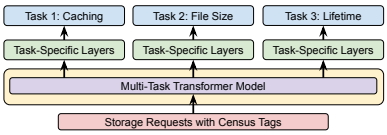


Figure 7. Using the Transformer in multi-task learning.

Multi-Task Learning. While the Transformer model is shown for a single task, the front-end (encoder) part of the model could be reused in a multi-task setup (Figure 7). The fully connected layers at the end of the network can be replaced by different layers for each task. The Transformer could then be jointly trained on multiple storage tasks.

5 Implementation Details

We prototyped and evaluated our models in a simulation setup driven by production traces. We pre-process these traces using large-scale data processing pipelines (Chambers et al., 2010) and run them through our models.

Lookup Table. The lookup table performance is calculated using our data processing pipelines. We aggregate across CTCs, perform predictions for each CTC and then weight by numbers of requests that belong to each CTC.

K-Nearest Neighbors. We build on the ScaNN nearest-neighbor framework that uses an inverted index method for high-performance k-nearest neighbor search (Guo et al., 2020). We use this framework to conduct an offline approximate nearest neighbors search with K=50. Most of this pipeline is shared with the lookup table calculation.

Transformer. We implement our Transformer models in TensorFlow (Abadi et al., 2016) and run both training and evaluation on TPUs (Jouppi et al., 2017), using the Tensor2Tensor library (Vaswani et al., 2018). We use the following hyperparameters: `{num_hidden units=64, num_hidden_layers=2, num_heads=4}` and a sinusoid positional embedding. We train using a weighted sampling scheme to ensure that CTCs occur approximately as often as they would in the actual trace.

6 Evaluation

We evaluate our models on traces. We start with microbenchmarks based on synthetic traces that demonstrate the ability of our models to generalize to unseen CTCs. We then evaluate our models on production traces from Google data centers. Finally, we show a simulation study that applies our models to two end-to-end storage problems, cache admission and SSD/HDD tiering.

Tags	Len. P/D	Samples	Table	K-NN	Transformer
Separate	2 / 5	1,000	8.00	0.001	0.008
Separate	2 / 5	10,000	8.00	0.000	0.015
Separate	10 / 20	1,000	8.00	0.000	0.047
Separate	10 / 20	10,000	8.00	0.000	0.005
Combined	2 / 5	1,000	8.00	8.000	0.034
Combined	2 / 5	10,000	8.00	8.000	0.003
Combined	10 / 20	1,000	8.00	8.000	0.017
Combined	10 / 20	10,000	8.00	8.000	0.006

Table 2. Mean squared error (MSE) on a synthetic microbenchmark that combines an information-carrying (P)refix with a (D)istractor of a certain length, in the same or separate tags.

6.1 Microbenchmarks

To demonstrate the ability of our models to learn information in high-cardinality and free-form Census tag strings, we construct a synthetic data set for the interarrival time task. We create 5 overlapping Gaussian clusters with means $\mu = \{1, 3, 5, 7, 9\}$ and $\sigma = 1$. Requests from the same cluster are assigned a shared prefix and a randomly generated distractor string (in real Census tags, this might be a timestamp or UID). The goal is for the model to learn to ignore the distractor string and to predict the parameters of each cluster based on the observed shared prefix. We experiment with two different setups: 1) the prefix and distractor are in *separate* Census tags, and 2) the prefix and distractor are *combined* in the same Census tag. For the former, the model has to learn to ignore one particular Census tag, for the latter, it has to extract part of a string.

We compute the MSE to indicate how close the predicted log-normal parameters (μ, σ) were to the ground truth. An error of 0 indicates that we perfectly recovered the distribution, while an error of 8 $(= (2 \times 2^2 + 2 \times 4^2 + 0)/5)$ corresponds to always predicting the average of all means. We also vary the number of samples per cluster between 1,000 and 10,000 to study how many samples are needed to learn these parameters. The lookup table is unable to learn either case, since it can only handle exactly matching CTCs (Table 2). K-Nearest Neighbor (with K=∞) can correctly predict the separate cases, but fails on the combined cases since it cannot look into individual strings. Finally, the neural network successfully learns all cases. We find that 10K samples per class were sufficient to learn a predictor that stably achieves an error close to 0 and does not overfit. This data shows how our models are able to learn successively more information, representative of the actual traces.

6.2 Prediction Latency

A key question for deployment is the models' latency. In practice, the lookup table will be used to cover the vast majority of cases and the more expensive models only run when a CTC is not in the table (the result is added for the next time the CTC is encountered). This gives the best of both worlds – resilience to drift over time and across clusters, and high performance. Evaluating on file creation requests, we found that after one month, only 0.3% of requests had CTCs that were never seen before. We measured our lookup table at 0.5 μs per request and the largest Transformer model at 99 ms (entirely untuned; we believe there is headroom to reduce this significantly). The average latency with the Transformer is therefore 310 μs, which is fast enough to run at relatively rare operations like file creation (e.g., the SSD/HDD tiering case). For more frequent operations (e.g., block reads/writes), we would use the cheaper models, whose latency can be hidden behind disk access.

6.3 Production Traces

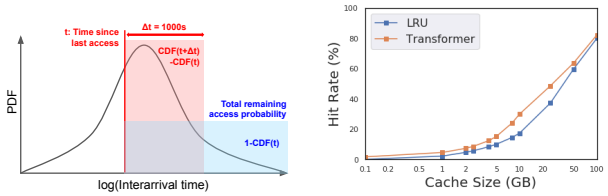
We now evaluate our models on real production traces. Our evaluation consists of two main components: evaluating model generalization error, and demonstrating end-to-end improvements in simulation.

As discussed in Section 3.3, we would like models to generalize 1) across long time horizons, and 2) across clusters. We train models on a 3-month trace and evaluate their generalization on a 3-month trace from the same cluster 9 months later, and a 3-month trace from the same time period on a different cluster. We find that models perform well within the same cluster and less (though acceptably) well across clusters. We measure both the weighted and unweighted error, and show that simple models are sufficient to learn the head of the distribution while more complex models are better at modeling the long tail. We also find that while there is some drift in each CTC's intrinsic statistics, generalization error across time is largely due to unseen CTCs, indicating that a model can be stable over time. More details about the results and error metrics can be found in the Appendix.

6.4 End-to-End Improvements

We now show two case studies to demonstrate how the CDF predictions can be used to improve storage systems. While our simulations use production traces and are inspired by realistic systems, they are not exact representations of any real Google workload. Additional evaluation of variation within some of these results is provided in the Appendix.

Cache Admission and Eviction. We implement a cache simulator driven by a consecutive time period of 40M read requests from our production traces. These are longitudinal traces to a distributed file system; while our simulation does not model any specific cache in our production system, this is equivalent to an in-memory cache in front of a group of servers that are handling the (small) slice of files represented by our traces. Such caches can have a wide range of hit rates (Albrecht et al., 2013), depending on the workload mix and upstream caches. As such, these results are representative for improvements one might see in production systems. The approach is similar to prior work on probability-based



(a) Calculating Utility (b) Cache Hit Rate

Figure 8. Using predictions in caching.

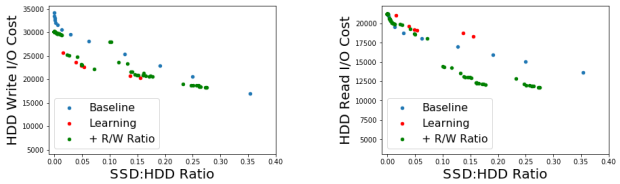


Figure 9. Using predictions in SSD/HDD tiering.

replacement policies such as EVA (Beckmann & Sanchez, 2017) or reuse interval prediction (Jaleel et al., 2010). The main difference is that we can predict these reuse intervals more precisely using application-level features.

We consider a cache that operates at a 128KB fixed block size granularity and use an LRU admission policy as the baseline; LRU is competitive for our distributed file system caching setup, similar to what is reported by Albrecht et al. (2013). Our learning-based policy works as follows: At every access, we use our model to predict (μ_Y, σ_Y) of the lognormal distribution associated with this request. We store these parameters in the block's metadata, together with the timestamp of the last access to the block. We now define the utility of a block as the probability that the next access to the block is within the next Δt = 1,000s (Δt is configurable). This value can be computed in closed-form (Figure 8a):

$$Utility(t, \mu, \sigma) = \frac{CDF(t + \Delta t|\mu, \sigma) - CDF(t|\mu, \sigma)}{1 - CDF(t|\mu, \sigma)}$$

We logically arrange the blocks into a priority queue sorted by increasing utility. When we insert a block into the cache, we compute its utility and add it to the priority queue. If we need to evict a block, we pick the entry at the front of the queue (after comparing it to the utility of the new block). We therefore ensure that we always evict the block with the lowest utility/probability of access. Note that this utility changes over time. Recomputing all utilities and sorting the priority queue at every access would be prohibitive – we therefore only do so periodically (e.g., every 10K requests). An alternative is to continuously update small numbers of entries and spread this work out across time. Figure 8b shows that the model improves the hit rate over LRU by as much as from 17% to 30%.

SSD/HDD Tiering: We perform a simple simulation study to demonstrate how predictions can be used to improve SSD/HDD tiering. We assume a setup similar to Janus (Albrecht et al., 2013) where an HDD tier is supplemented with an SSD tier to reduce HDD disk I/O utilization (since spinning disks are limited by disk seeks, fewer accesses for hot and short-lived data means that fewer disks are required). Our baseline is a policy that places all files onto SSD initially and moves them to HDD if they are still alive after a specific TTL that we vary from 10s to 2.8 hours. We use 24 hours of the same traces as in the previous example and compute the average amount of live SSD and HDD memory, as well as HDD reads and (batched) writes as a proxy for the cost.

We use our model to predict the lifetime of newly placed files (Figure 9). We only place a file onto SSD if the predicted $\mu + n \times \sigma$ is smaller than the TTL (we vary $n = 0, 1$). After the file has been on SSD for longer than $\mu + m \times \sigma$ ($m = 1, 2$), we move it to the HDD. This reduces write I/O at the same SSD size (e.g., by ≈20% for an SSD:HDD ratio of 1:20) or saves SSD bytes (by up to 6×), but did not improve reads. We also used the model to predict read/write ratio (Section 3.2) and prefer placing read-heavy files on SSD. This keeps the write savings while also improving reads.

7 Future Work

Systems Improvements. The caching and storage tiering approach has applications across the entire stack, such as selecting local vs. remote storage, or DRAM caching. Other prediction tasks in storage systems that can benefit from high-level information include read-write ratio, resource demand forecasting, and antagonistic workload prediction. Our approach could also be applied to settings other than storage systems (e.g., databases), and to other (non-Census) meta-data, such as job names.

Modeling Improvements. Our approach could benefit from a method for breaking up Census Tags into meaningful sub-tokens. In addition to the bottom-up compression-based approaches used in NLP such as Byte Pair Encoding (BPE, Gage (1994)) or WordPiece (Wu et al., 2016), we may also want to identify known important tokens ahead of time, e.g. "temp" or "batch". This would improve our model's ability to generalize to unseen Census Tags and new clusters. On the distribution fitting side, log-normal distributions are not ideal in all scenarios. For instance, distributions that are bounded or multi-modal are respectively better handled using a Beta distribution or Mixture-of-Gaussians.

8 Conclusion

Our results demonstrate that information in distributed traces can be used to make predictions in data center storage services. We demonstrated three models – a lookup table, k-NN and neural-network based approach – that, when combined, extract increasing amounts of information from Census tags and significantly improve storage systems.

ACKNOWLEDGEMENTS

We want to thank Richard McDougall for supporting this project since its inception and providing us with invaluable feedback and technical insights across the storage stack. We also want to thank David Andersen, Eugene Brevdo, Andrew Bunner, Ramki Gummadi, Maya Gupta, Daniel Kang, Sam Leffler, Elliot Li, Erez Louidor, Petros Maniatis, Azalia Mirhoseini, Seth Pollen, Yundi Qian, Colin Raffel, Nikhil Sarda, Chandu Thekkath, Mustafa Uysal, Homer Wolfmeister, Tzu-Wei Yang, Wangyuan Zhang, and the anonymous reviewers for helpful feedback and discussions.

REFERENCES

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283, Savannah, GA, November 2016. USENIX Association. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

Albrecht, C., Merchant, A., Stokely, M., Waliji, M., Labelle, F., Coehlo, N., Shi, X., and Schrock, E. Janus: Optimal flash provisioning for cloud storage workloads. In *Proceedings of the USENIX Annual Technical Conference*, pp. 91–102, 2560 Ninth Street, Suite 215, Berkeley, CA 94710, USA, 2013. URL https://www.usenix.org/system/files/conference/atc13/atc13-albrecht.pdf.

Barham, P., Isaacs, R., Mortier, R., and Narayanan, D. Magpie: Online modelling and performance-aware systems. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, 2003.

Beckmann, N. and Sanchez, D. Maximizing cache performance under uncertainty. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 109–120, 2017.

Bishop, C. M. Mixture density networks. 1994. URL http://publications.aston.ac.uk/id/eprint/373/.

Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R., Bradshaw, R., and Nathan. Flumejava: Easy, efficient data-parallel pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 363–375, 2 Penn Plaza, Suite 701 New York, NY 10121-0701, 2010. URL http://dl.acm.org/citation.cfm?id=1806638.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

Cortez, E., Bonde, A., Muzio, A., Russinovich, M., Fontoura, M., and Bianchini, R. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pp. 153–167, New York, NY, USA, 2017. Association for Computing Machinery. URL https://doi.org/10.1145/3132747.3132772.

Delimitrou, C. and Kozyrakis, C. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pp. 127–144, New York, NY, USA, 2014. Association for Computing Machinery. URL https://doi.org/10.1145/2541940.2541941.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Fernandez, T., Rivera, N., and Teh, Y. W. Gaussian processes for survival analysis. In *Advances in Neural Information Processing Systems 29*, pp. 5021–5029. Curran Associates, Inc., 2016. URL http://papers.nips.cc/paper/6443-gaussian-processes-for-survival-analysis.pdf.

Fitzpatrick, B. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

Gage, P. A new algorithm for data compression. *The C Users Journal*, 12(2):23–28, 1994.

Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., Bruno, A., Hu, J., Ritchken, B., Jackson, B., Hu, K., Pancholi, M., He, Y., Clancy, B., Colen, C., Wen, F., Leung, C., Wang, S., Zaruvinsky, L., Espinosa, M., Lin, R., Liu, Z., Padilla, J., and Delimitrou, C. An open-source benchmark suite for microservices and their hardware-software implications for cloud edge systems.

In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pp. 3–18, New York, NY, USA, 2019a. Association for Computing Machinery. URL https://doi.org/10.1145/3297858.3304013.

Gan, Y., Zhang, Y., Hu, K., Cheng, D., He, Y., Pancholi, M., and Delimitrou, C. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pp. 19–33, New York, NY, USA, 2019b. Association for Computing Machinery. URL https://doi.org/10.1145/3297858.3304004.

Ghemawat, S., Gobioff, H., and Leung, S.-T. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

Guo, R., Sun, P., Lindgren, E., Geng, Q., Simcha, D., Chern, F., and Kumar, S. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*, 2020. URL https://arxiv.org/abs/1908.10396.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., and West, M. J. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):51–81, 1988.

Jaleel, A., Theobald, K. B., Steely, S. C., and Emer, J. High performance cache replacement using re-reference interval prediction (rrip). *SIGARCH Comput. Archit. News*, 38(3):60–71, June 2010. URL https://doi.org/10.1145/1816038.1815971.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-l., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45(2):1–12, June 2017. URL https://doi.org/10.1145/3140659.3080246.

Kanev, S., Darago, J. P., Hazelwood, K., Ranganathan, P., Moseley, T., Wei, G.-Y., and Brooks, D. Profiling a warehouse-scale computer. *ACM SIGARCH Computer Architecture News*, 43(3):158–169, 2016.

Karampatsis, R.-M. and Sutton, C. Maybe deep neural networks are the best choice for modeling source code. *arXiv preprint arXiv:1903.05734*.

Kim, T., Hahn, S. S., Lee, S., Hwang, J., Lee, J., and Kim, J. Pcstream: Automatic stream allocation using program contexts. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, Boston, MA, July 2018. USENIX Association. URL https://www.usenix.org/conference/hotstorage18/presentation/kim-taejin.

Kirilin, V., Sundarrajan, A., Gorinsky, S., and Sitaraman, R. K. Rl-cache: Learning-based cache admission for content delivery. In *Proceedings of the 2019 Workshop on Network Meets AI ML*, NetAI'19, pp. 57–63, New York, NY, USA, 2019. Association for Computing Machinery. URL https://doi.org/10.1145/3341216.3342214.

Liu, E., Hashemi, M., Swersky, K., Ranganathan, P., and Ahn, J. An imitation learning approach for cache replacement. In III, H. D. and Singh, A. (eds.), *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 6237–6247. PMLR, 13–18 Jul 2020. URL http://proceedings.mlr.press/v119/liu20f.html.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

Mullen, R. E. The lognormal distribution of software failure rates: application to software reliability growth modeling. In *Proceedings Ninth International Symposium on Software Reliability Engineering (Cat. No.98TB100257)*, pp. 134–142, 1998.

OpenCensus. Opencensus. https://opencensus.io/, 2020. URL https://opencensus.io/.

Ousterhout, J., Gopalan, A., Gupta, A., Kejriwal, A., Lee, C., Montazeri, B., Ongaro, D., Park, S. J., Qin, H., Rosenblum, M., Rumble, S., Stutsman, R., and Yang, S. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3), August 2015. URL https://doi.org/10.1145/2806887.

Park, J. W., Tumanov, A., Jiang, A., Kozuch, M. A., and Ganger, G. R. 3sigma: Distribution-based cluster scheduling for runtime uncertainty. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery. URL https://doi.org/10.1145/3190508.3190515.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8), 2019.

Safavian, S. R. and Landgrebe, D. A survey of decision tree classifier methodology. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(3):660–674, 1991.

Serenyi, D. PDSW-DISCS Keynote: From GFS to Colossus: Cluster-Level Storage at Google. http://www.pdsw.org/pdsw-discs17/slides/PDSW-DISCS-Google-Keynote.pdf. 2017.

Shrivastava, D., Larochelle, H., and Tarlow, D. On-the-fly adaptation of source code models using meta-learning. *arXiv preprint arXiv:2003.11768*, 2020.

Shvachko, K., Kuang, H., Radia, S., Chansler, R., et al. The hadoop distributed file system. In *MSST*, volume 10, pp. 1–10, 2010.

Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., and Shanbhag, C. Dapper, a large-scale distributed systems tracing infrastructure. 2010. URL https://research.google/pubs/pub36356/.

Song, Z., Berger, D. S., Li, K., and Lloyd, W. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pp. 529–544, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-13-7. URL https://www.usenix.org/conference/nsdi20/presentation/song.

Stonebraker, M. and Kemnitz, G. The postgres next generation database management system. *Communications of the ACM*, 34(10):78–92, 1991.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.

Vaswani, A., Bengio, S., Brevdo, E., Chollet, F., Gomez, A. N., Gouws, S., Jones, L., Łukasz Kaiser, Kalchbrenner, N., Parmar, N., Sepassi, R., Shazeer, N., and Uszkoreit, J. Tensor2tensor for neural machine translation. *arXiv preprint arXiv:1803.07416*, 2018.

Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pp. 307–320, USA, 2006. USENIX Association.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

APPENDIX

A PREDICTION ERROR EVALUATION

Kolmogorov-Smirnov Distance. Since our models predict distributions rather than specific values, we use KS distance (the maximum deviation between CDFs of each CTC) to compare their accuracy. In other words, KS distance is determined by the point at which one distribution maximally lags behind another in terms of cumulative probability mass, and can be viewed as a way of measuring worst-case error. In our traces, we encounter several CTCs much more often than others. We thus report both weighted and unweighted error, due to two real-world considerations: 1) error should be lower on more common workloads for which we have more data and whose optimization has a larger overall effect, 2) optimizing for the common cases can be done statically whereas the strength of a model should lie in its ability to generalize well to new (and less common) examples.

In-depth error analysis on production traces. As mentioned in Section 3.3, we need models to generalize 1) across long time horizons, and 2) across clusters. We evaluate generalization by training our models on a 3-month trace and evaluating on a 3-month trace from the same cluster 9 months later, and a 3-month trace from the same time period on a different cluster. Table 3 reports both weighted and unweighted error: The former indicates the overall accuracy, but hides the long tail of workloads. For example, if 90% of requests have the same CTC and this CTC can be predicted by a lookup table, this will dominate the results. Meanwhile, the unweighted error tells us about the accuracy of the model on the long tail of CTCs. We would like both errors to be low – in practice, common CTCs will be handled by a lookup table, while more complex models handle the long tail. Our data supports this: As the model complexity increases, the error decreases. For reference, we also report the "oracular" error of directly fitting the log-normal distributions to the data used for evaluation. Note that this does *not* minimize KS distance, which measures the maximum divergence of each CDF, not the average.

Evaluating drift over time. To better understand the relative contributions of temporal drift in the statistics v.s. unseen CTCs to the overall error, we separately compute the KS distance of CTCs that have been seen in the training set, as well as their proportion (Figure 11). In all cases, we observe that known CTCs exhibit some increase in test error compared to the training error. However, unseen CTCs still remain the predominant source of error, comprising a larger portion of all CTCs and generally incurring higher test error; this is shown by the fact that the known CTC test error is much lower than the overall error shown in Table 3.

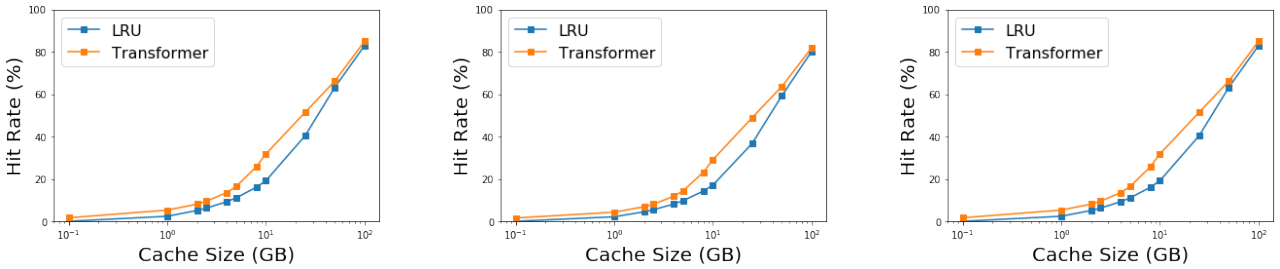
We conclude that infrequent periodic retraining is necessary, but doing so 1-2 times per year (or less) might be sufficient, and training may start from a previous snapshot that has already learned existing Census tags.

B END-TO-END RESULT STABILITY

To ensure that results are consistent between different runs of the end-to-end experiments, we evaluate our caching model on different time slices. We do so by offsetting the start of the test period to fall into different parts of a 24-hour period, to account for variations throughout the day (Figure 10). The end-to-end evaluation results are largely consistent with each other and Figure 8b.

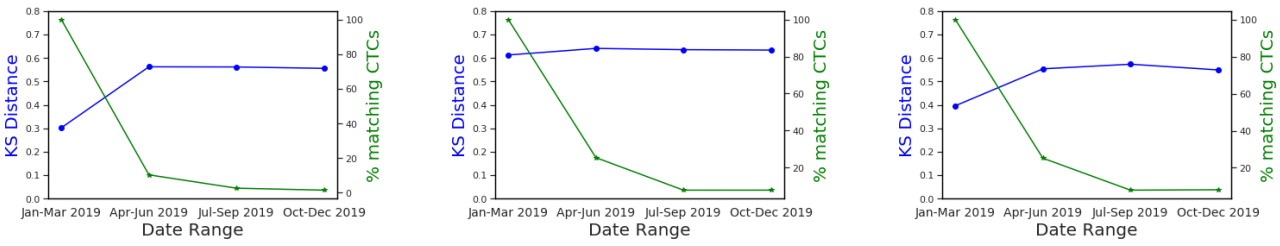
	Weighted by #requests with same CTC						Unweighted (every observed CTC counts once)					
	Across Time			Across Clusters			Across Time			Across Clusters		
Model	**IT**	**FS**	**LT**	**IT**	**FS**	**LT**	**IT**	**FS**	**LT**	**IT**	**FS**	**LT**
Oracular Predictor	0.09	0.41	0.13	0.14	0.51	0.19	0.28	0.53	0.35	0.34	0.56	0.33
Lookup Table	0.33	0.45	0.33	0.50	0.54	0.42	0.58	0.69	0.68	0.70	0.67	0.75
Nearest Neighbor	0.27	0.47	0.30	0.53	0.53	0.37	0.62	0.66	0.56	0.71	0.66	0.71
Embedding Transformer	0.29	0.39	0.27	0.47	0.49	0.39	0.64	0.62	0.58	0.68	0.64	0.73
Raw String Transformer	0.26	0.38	0.22	0.45	0.47	0.35	0.60	0.63	0.58	0.67	0.63	0.66

Table 3. Generalization KS Distance (IT=Interarrival Times, FS=File Size, LT=Lifetime).



(a) First time interval (b) Second time interval (c) Third time interval

Figure 10. Additional evaluation of the caching end-to-end experiment. We show that the results are stable across multiple data points by looking at three time intervals spanning the three different parts of a 24-hour cycle. While the precise hit rates vary slightly between these different time intervals, the improvements of the learned strategy over the baseline remain similar.



(a) Interrarrival times. (b) Final file size. (c) File lifetime.

Figure 11. The unweighted error (in KS distance) of known CTCs over time as well as their proportion of all observed CTCs. The statistical properties for the same CTC are relatively stable over time but the number of matching CTCs drops quickly, demonstrating limitations of the lookup table approach.