Determining SLO Violations at Compile Time

Kristal Curtis
EECS at UC Berkeley

Abstract
Fearing the fate of Friendster\(^1\), which fell out of vogue due to its unacceptable latency, providers of large-scale, interactive web services obsess about performance. Their developers thus labor to write queries whose latency will meet the company’s service quality goal, or service level objective (SLO). Since it is still very difficult to reason about latency for traditional relational database queries, today’s large-scale applications are increasingly moving from relational databases to distributed key-value stores, which are attractive due to their incremental scalability and predictable performance. While key-value stores are less convenient than databases due to their narrow interface, a recent project, the Performance-Insightful Query Language (PIQL), addresses this problem by allowing developers to express their queries declaratively and compiling the queries to key-value primitives. These developments provide an appealing setting for prediction. Using PIQL query operator models and query plans, we can predict queries’ 99th-percentile latency within 20% of the actual values.

1 Introduction
As software as a service (SaaS) has become the dominant way for many companies to provide their products, we have all grown accustomed to using a vast array of software products from both technology giants and hopeful start-ups. As applications have proliferated, users have come to expect that those on which they choose to spend time will be exciting while maintaining sub-half-second response time [11]. Applications that do not meet both of these requirements may be quickly passed over for the next craze. This means developers must endeavor to imbue their apps with all the functionality their customers demand while minimizing latency.

As they strive to provide excellent services, SaaS providers often operate with a service-level objective (SLO), which is the company’s goal for the quality of their services. SLOs can relate to aspects of a service such as performance, reliability, and durability and are pertinent whether the services are available only internally, like Google’s Bigtable [5], or also externally, like Gmail. In the case of latency SLOs, it has become common to focus not on the average request latency but rather on the tail of the latency distribution so that all users are satisfied with the service [9]. An example of such an SLO is “99% of requests must complete within 500 ms,” that is, that the 99th-percentile request latency is no greater than 500 ms.

As developers attempt to create performant, interesting applications that meet their SLOs, they naturally wish to ask what-if questions about their systems. For instance, as developers write queries that will be used to respond to user requests, they would like to reason about whether their queries’ latency will be within the company’s SLO. Developers would also like to know how to navigate the tradeoff between money spent on additional resources and improvement to the service; for example, a question of interest would be, “How much would it cost to reduce the 99th-percentile latency by 50 ms?” [15] This question is especially interesting when resources can be procured immediately through cloud computing. Also, since SaaS applications have a high rate of churn, developers are interested in reasoning about how the app will behave when new queries are added or when existing queries are executed more frequently due to surges in popularity of their corresponding features.

Due to concerns about scalability and performance, many apps are now backed by key-value stores rather than by full relational databases. Unlike monolithic databases, key-value stores provide incremental scal-

\(^1\)http://www.friendster.com
ability; capacity may be gradually increased by adding a few servers at a time. Also, while databases' expressiveness results in performance opacity, key-value stores expose a narrow interface and provide predictable performance. However, a disadvantage of this transition is that developers must now endure tedious manual data access along with index creation and maintenance where they once enjoyed the convenience of declarative queries [2].

A recent project addressing these challenges is the Scalable Consistency-Adjustable Data Store (SCADS) [1]. The physical storage layer of SCADS is a key-value store. To achieve physical data independence, SCADS provides the Performance-Insightful Query Language (PIQL) [2], whereby developers may access and manipulate application data. Approximately a subset of SQL, PIQL constrains the behavior of its queries in order to provide constant per-user performance as the application scales. While SQL queries can potentially access and manipulate all stored data, PIQL queries bound the number of data items they will consider. This restriction helps developers to write applications that scale as their user bases grow.

The SCADS combination of performance-transparent primitives with a performance-safe query language allows us to reason probabilistically about PIQL query performance at compile time. We can exploit the fact that PIQL breaks down high-level queries into a set of known query operators to predict the latency distribution of the queries. While the database community has long attempted to estimate query costs in terms of their operators, they have not used this method to predict query latency.

The main contribution of this work is the composition of operator latency models for predicting the latency distribution of PIQL queries, including the tail. Though it may be interesting in other domains to predict the average or median latency, only the tail is important here, since that is where the great majority of SLOs focus. Our predicted values for the 99th-percentile latency for several queries for an application similar to Twitter are within 10 ms, or 20%, of the actual values.

2 Related Work

This work is most similar to the cost estimation module of query optimizers, which has been an active area of work for many years. An excellent survey of this body of work is provided in [6]. The System R project laid the foundation for query cost estimation [17]. The System R cost estimation module took a bottom-up approach, where it first estimated the cost of each physical operator and then combined the operator cost estimates to predict the overall query cost. We use that approach in this work as well. Their method for determining operator cost relied on estimating the length of the operator's input data stream and then plugging this number into a cost formula. They relied on histograms over the system's stored data to estimate data stream length. Much effort has been dedicated to improving these histograms, such as [14, 7].

A weakness of previous work is that its output, a cost estimate, is often an abstract measure that does not have a clear relationship to actual latency. It is thus very difficult to get a sense for the 99th percentile latency of a query given a cost estimate. In this work, we define operator cost as actual latency (that is, wall clock time) rather than CPU time. We are not as interested in the other resources consumed, since our chief goal is to reason about query latency and whether it is acceptable according to the prescribed SLO. At this point, we hold the size of each operator's input data streams constant. We instead focus on estimation of the overall query latency, which is known to be a very difficult problem in query optimization. Our problem is simpler than query cost estimation in full relational databases due to the constrained nature of PIQL queries and the performance transparency of the underlying key-value store; thus, we hope to make progress. We do use histograms, but they are distributions over observed values of the operator latency rather than over the data. Our use of operator latency distributions allows us to consider the inherent uncertainty in operator execution time [8]. In the future, we plan to allow data stream length to vary; at that point, we will consider the work cited above.

In [3], the authors focus on cardinality estimation, which is an important aspect of query optimization. A similar feature between [3] and this work is that the authors produce a distribution over the cost of a given query plan, based on the relationship between its cost and the selectivity, rather than producing a point estimate. The authors also allow the users to specify quantiles of interest of the cost distribution based on whether they are more interested in typical or worst-case behavior. However, they do not attempt to estimate query latency.

Another related project is [10], in which the authors achieve good prediction for several query metrics
using a black box technique. The authors represent each query with two vectors: one serves as a collapsed representation of the optimizer-generated query plan and its cardinality estimates; the other records several metrics of interest, including latency and resource consumption for several different resources. Then, when given a new query, the authors use a nearest neighbors algorithm in a transformed space to find similar queries. The resource vectors for the similar queries are combined to predict the latency and other metrics for the new query. However, this technique has only been validated when each query is run in isolation on the database. Also, like all nearest-neighbors algorithms, it relies on the presence in the training set of queries that are similar to the new query. Another difference between this work and ours is that we do not require observations of queries in order to make predictions; rather, we observe operators so we can make predictions at compile time, before any queries have been observed.

The space of modeling request latency for web services is also relevant to our work. Much effort has focused on estimating the average latency of requests to a three-tier web service via queueing models, for example, [19]; however, this work does not readily extend to modeling the tail latency. Recent work has successfully modeled high latency quantiles of RUBiS\(^2\) transactions in order to assess the impact of CPU allocation/contention in a virtualized environment on response time [21]. This is meant to guide the decisions of automatic controllers like the one mentioned in [18] that dynamically adjust the allocation of virtual machines to different components in a three-tier web architecture. Our work differs in that while the authors of [21] create their models based on observations of other transactions and their resource utilization, we use models of query operators to predict the query’s latency distribution. Using our approach, we can reason about whether or not queries will result in SLO violations at compile time rather than only after the system is already deployed.

3 Methodology

Before presenting our three-step methodology, we explain several of the assumptions we make about the setting in which we attempt our predictions.

3.1 Assumptions

In the well-established three-tier web service architecture, the application, or app, tier assembles and presents content and runs business logic, while the storage tier stores persistent data. To perform its work, the app tier issues queries, which consume resources at both the app tier and the storage tier. Thus, in the presence of a high volume of user requests, contention among queries at both of these tiers could hurt performance predictability.

Thanks to cloud computing, we can easily address the contention problem at the app tier by adding servers, since they are stateless. Adding servers is straightforward; we simply monitor the workload or utilization at the app servers and spin up new ones when it exceeds a given threshold. Similarly, we can remove app servers if the workload decreases beyond the target workload range [4]. There are commercially-available products to automate this process, such as RightScale\(^3\) and Auto Scaling from Amazon Web Services\(^4\). Hence, we can maintain a target utilization on all servers at the app tier, so that even as the workload varies, the contention at the app tier stays relatively steady.

Contestion at the storage tier cannot be so easily alleviated: we cannot just add servers at the storage tier due to additional complexity involving data movement, replication, and/or partitioning. However, there have been successful efforts to harness the elasticity of cloud computing for storage. The basic idea is for the storage system to provide a small repertoire of primitives - usually \textit{get}, \textit{get range}, and \textit{put} - and a model-based control system that can add and remove storage nodes and move or copy data as needed to rebalance load and maintain predictable performance of those primitives. We assume that a system like this is available, such as the one described in [18].

\(^2\)http://rubis.ow2.org/
\(^3\)http://www.rightscale.com/
\(^4\)http://aws.amazon.com/autoscaling/
3.2 Approach

PIQL queries map to three levels of execution: PIQL queries call operators such as sort and prefixGet, some of which spawn key-value primitives such as get and get_range. All operators are executed at the app tier, while some also create work at the storage tier. Figure 1 illustrates this three-level execution scheme. Though a typical application requires many queries, these queries can be executed by assembling a subset of a relatively small set of simple query operators.

![Figure 1: Three levels of execution involved in PIQL queries, with examples of each level.](image)

Our hypothesis is that if we can model the latency of each operator, and if we know how the operators combine to produce the query, we can reason about the latency distribution of the query itself. Our three-step methodology, which we illustrate in Figure 2 and describe below, is as follows:

- Produce a model for each operator.
- Combine the operator models to produce a sampler for each query.
- Use each query’s sampler to predict its latency distribution.

The first step is to produce a model for each of the nine PIQL query operators. We gather the data used for modeling each operator by observing the operator in isolation, as this provides convenience and incremental modeling if new query operators are added. Since we assume that contention at the app tier will be insignificant due to our ability to add app servers as needed, we hypothesize that an operator will not behave much differently as the operator mix at the app tier changes. Thus, our models gathered from a context where the operator mix is homogeneous should extend to scenarios with heterogeneous operator mixes. In Figure 2, the operator distributions appear at the top. They are represented here by histograms. We have two queries to emphasize that the operator models are reused across queries; thus, it is crucial to construct models that are independent of any particular query.

The second step is to produce a latency sampler for each query of interest in the target application, that is, the application whose queries we wish to model. To do so, we combine the operator models according to the physical query plan generated by the PIQL compiler. Note that unlike a traditional query optimizer, the PIQL compiler produces a single execution plan for each query; if it were to produce multiple, we could easily extend our methodology to choose the best one. At this point, we assume independence among operators; that is, we assume that the latency of one operator is not influenced by the latency of another operator. Thus, this step is simple. For query plans (or plan sections) that are serial, we add up the latency samples from the individual operator models, while for parallel plans or plan sections, we take the maximum.
Third, we use each query’s sampler to predict its latency distribution. Each sample gives an estimate for the query latency. Taking many samples results in a distribution over the latency, reflecting the uncertainty in the query’s running time. Rather than a single value, this distribution serves as our prediction of the query’s running time. A latency distribution is more useful for reasoning about the performance of a query during the application’s execution since the query’s running time will exhibit significant variance. Also, we are interested in the overall performance of the query rather than the latency of a given run of the query. Thus, we would like our distributions to apply as the parameters to the query vary. However, at this point, we do not attempt to address variations in query selectivity. In Figure 2, the query latency distributions appearing at the bottom of the figure would be obtained from the queries’ respective samplers.

Our goal is to determine whether the query will meet its SLO; to do so, we inspect the predicted latency distribution. We are chiefly concerned at this point with detecting violations of SLOs that are posed in terms of high quantiles of the query latency distribution; thus, if the SLO stipulates that “99% of queries should complete in under 500 ms,” and if the 99th-percentile latency of our predicted distribution is less than 500 ms, we predict that the query will be completed within its SLO. Thus, it is crucial that we perform good prediction for not only the bulk of the query’s latency distribution but for the tails as well. In Figure 2, the 99th-percentile latency has been marked on the queries’ latency distributions and can be compared to the queries’ SLOs.

Figure 2: Modeling process for PIQL queries. First, we create models for each PIQL operator. Then, for each query, we combine the operator models according to the query plan to create a latency sampler. Taking many draws from the query’s sampler allows us to estimate the query’s latency distribution, from which we can predict whether the query will meet its SLO.
4 Experimental Results

To evaluate our methodology, we modeled the latency of several queries for SCADr, a Twitter\(^5\) clone developed at our lab. In Section 4.1, we detail how we created the operator models. In Section 4.2, we explain how the operator models were used to model SCADr queries and present the results of our modeling.

4.1 Operator Modeling

The first step was to produce operator models for each of the nine PIQL operators; we describe the operators in Table 1. We adopted a data-driven approach to modeling; thus, we needed to gather many observations of the execution of each operator with which to build our models. In what follows, we describe how we collected the operator data, and then we explain several approaches we employed for building operator models.

4.1.1 Operator Data Collection

In order to gather operator data, we created entities, namespaces, and an index using a generic PIQL specification, or spec (Figure 3). Our motivation for using a generic PIQL spec was to avoid producing operator models that are tied to a particular application; instead, we want to create models that can be used to predict high-quantile latency for queries of various applications. We created tables that would allow us to exercise each operator. For example, the `prefixJoin` operator receives a stream of entities; for each of the entities in its input, it returns a list of objects that possess the entity as a foreign key. Thus, we included this relationship in our PIQL spec (note the relationship between entities A and B). We do likewise for each operator.

Table 1: The 9 PIQL operators and their descriptions.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>I/O Primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>singleGet</td>
<td>Access one item</td>
<td>get</td>
</tr>
<tr>
<td>prefixGet</td>
<td>Access one range</td>
<td>get_range</td>
</tr>
<tr>
<td>sequentialDereferenceIndex</td>
<td>Lookup (n) items from index</td>
<td>(n) get</td>
</tr>
<tr>
<td>prefixJoin</td>
<td>Access one range per input entity</td>
<td>(n) get_range</td>
</tr>
<tr>
<td>pointerJoin</td>
<td>Access one item per input entity</td>
<td>(n) get</td>
</tr>
<tr>
<td>materialize</td>
<td>Convert (n) tuples to (n) entities</td>
<td>-</td>
</tr>
<tr>
<td>selection</td>
<td>Given (n) entities, return those that match</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>given criteria</td>
<td></td>
</tr>
<tr>
<td>sort</td>
<td>Given (n) entities, sort</td>
<td>-</td>
</tr>
<tr>
<td>topK</td>
<td>Given (n) entities, return top (K)</td>
<td>-</td>
</tr>
</tbody>
</table>

\(^5\)http://twitter.com/
Once we created our PIQL spec, the next step was to execute the operators against the spec’s tables. We deployed a mini SCADS cluster for each of the nine operators. Each SCADS cluster consists of a ZooKeeper node\(^6\), which maintains information about the data layout, as well as a storage node and a client node. The client node represents an app server, since queries to the storage tier originate at the app servers in a three-tier web application. Each node is run on a small Amazon EC2 instance, which consists of 1.7 GB of memory and 1 EC2 Compute Unit\(^7\). The storage node stores all of the data we generated with our generic PIQL spec. The client node hosts five threads, each of which continually calls the operator to which the cluster has been assigned. Recall that these threads correspond to threads at the app tier rather than users of the system. Since cloud computing allows us to scale out the app tier in order to maintain low contention there, we deem that five threads at an app server is a reasonable amount. We also tried using fifty threads, but this resulted in a significant amount of thrashing. The threads wait for one millisecond in between operator calls.

Though query operators are internal to the compiler and are therefore privileged, we are able to call them directly since this is a research system. The goal is that this modeling process would not need to be re-run by application developers; rather, the PIQL development team would gather operator data and produce operator models and would then integrate these models into the PIQL compiler. Then, the compiler would use the models to produce feedback to the developer when she compiled her app’s PIQL spec.

In Figure 4, we have provided a smoothed version of each operator’s empirical latency distribution which we obtained using the density command in \(^8\texttt{R}\). We note that the per-operator latency ranges vary widely among the operators. Some can be quite expensive (for example, operator 5), while others are trivial (for example, operator 9). Clearly, it is especially critical to produce good models for the more expensive operators, which have greater impact on the overall query latency.

### 4.1.2 Building the Models

To model the operator latency distributions, we explored several techniques. The techniques fall under two main categories: empirical and analytical. A key criterion for our modeling process is the match between the operator’s actual 99th-percentile latency value and the value predicted by the model. For the sake of space, many of our example fits will only be provided for one of the operators, operator 3, because its distribution was interesting, spread over a wide range of latencies, and challenging to fit. For the operators with non-trivial tail latency, the empirical models produce the most accurate fit of the distribution’s tail since they maintain more information about the data than the analytical techniques do. Among the analytical techniques, mixture models produce a much better fit than single distributions. For operators whose latency is negligible even at the tail, even single distributions suffice. In what follows, we present a detailed explanation of our modeling process.

#### Empirical

We investigated two empirical methods: a histogram representation of the distribution and a kernel density estimate of the distribution. When we use a histogram of the operator’s latency values, we must select the number of bins into which the data is grouped. For computational and storage efficiency and to avoid overfitting our dataset, we would like to choose the smallest number of bins that will result in a good match to the tail of the actual data distribution. In Figure 5, we show how the 99th percentile of the histogram for op3 approaches that of the actual data as the number of bins is increased. With 1000 bins, the 99th percentile of the histogram is already within 0.12 ms of the actual value, so we use 1000 bins. This value also worked well for the rest of the operators. An explanation of how kernel density estimates are obtained, including how to choose the bandwidth of the kernels, is given in [16]. In Table 2, we provide the actual 99th-percentile latency values for each of the nine operators as well as the values predicted by the 1000-bin histogram and the kernel density estimate.

\(^6\)http://hadoop.apache.org/zookeeper/
\(^7\)http://aws.amazon.com/ec2/instance-types/
\(^8\)http://www.r-project.org/
Figure 4: Operator latency distributions. Note that the latency of some operators is highly variable, while the latency of other operators is trivial even at the distributions's tail.

Analytical

For the analytical models, we attempted to fit both single distributions and mixture models to the operator data. For both single distribution models and mixture models, we used normal, exponential, gamma, and Weibull distributions. We selected these distributions for the following reasons: the normal is a very familiar distribution; the exponential is often used to describe service times; the gamma and Weibull distributions are similar to the exponential and are also suitable for representing time data, but they have an additional degree of freedom and are therefore more extensible. As we shall see, we had to use mixture models because single distributions did not provide satisfactory fits.

Analytical: Single Distributions

To fit the single distributions to our data, we employed maximum likelihood estimation, a well-known technique for choosing parameters that maximize the value of the likelihood function when it is applied to the data [20]. In Figure 6, we show the results of fitting each of these distributions to operator 3’s data. Clearly, the normal distribution is a poor choice for our data; it is symmetrical, while our data is not, and it
Figure 5: Predicted op3 99th-percentile latency using histogram of op3 latency as a function of the number of bins used to create the histogram. Predicted 99th-percentile value improves as the bin count increases. We chose 1000 bins, as this achieved the best tradeoff between storage space and accuracy of the 99th-percentile value.

allocates non-zero density to negative latency values. The other distributions, which are asymmetrical and allocate all their mass to positive latency values, are a better match for the actual op3 data. Comparing the key quantiles indicated on the plot, we notice that the mean, median, and 90th percentile values of the fitted distributions are close to the actual values; however, the fits perform poorly for predicting the 99th percentile value. The fits also fail to capture the multimodality in the tail of the actual distribution, most visible in the plot between 10 and 50 ms. Thus, as we see in the CDF of the actual distribution and the fits, the fits are all a poor match for the actual latency distribution. However, single gamma fits are sufficient for operators 6-9 since their latency range is so small and so narrow.

We conclude that single distributions produce unsatisfactory fits to our operators with non-trivial tail latencies. Therefore, we decided to try mixture models, which are more powerful.

Analytical: Mixture Models

The idea of a mixture model is to fit a weighted sum of several distributions of a certain type, such as normal, to a given dataset. Mixture models are well-suited for addressing the challenges we faced in attempting to model our data with single distributional models. First, mixture models can handle multimodal data. Less obviously, mixture models can produce fits whose tails are heavier than any single distribution of the same type. Thus, even if a distribution is unimodal, mixture models can be a good choice if the data is heavy-tailed. This is a significant advantage in our case, since our operator data is heavy-tailed and since we care most about the quality of the fit at the tail. We note that a difference between our problem and many common applications of density estimation, where the fit of the distribution to the bulk of the data is at least as important as the fit to the tail, is that we are most concerned with the fit to the tail. Another advantage of using mixture models to represent our data is that this is the first step towards using a more powerful technique called mixture of experts [13]. This technique would allow us to extend our modeling to account for variations in data stream length or storage node workload. We hope to explore the use of this technique in future work.

We used maximum likelihood estimation to fit our mixture models to the operator data, which is the
Table 2: Operator 99th-percentile latency values from empirical models. Since empirical models lose little information about the data’s distribution, the 99th-percentile values obtained from the models are highly accurate.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Raw Data (ms)</th>
<th>Histogram (ms)</th>
<th>Density (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>op1</td>
<td>20.98</td>
<td>20.90</td>
<td>21.02</td>
</tr>
<tr>
<td>op2</td>
<td>25.18</td>
<td>24.75</td>
<td>24.65</td>
</tr>
<tr>
<td>op3</td>
<td>60.87</td>
<td>60.75</td>
<td>60.73</td>
</tr>
<tr>
<td>op4</td>
<td>16.53</td>
<td>16.50</td>
<td>16.41</td>
</tr>
<tr>
<td>op5</td>
<td>65.78</td>
<td>65.70</td>
<td>65.64</td>
</tr>
<tr>
<td>op6</td>
<td>0.45</td>
<td>0.45</td>
<td>0.43</td>
</tr>
<tr>
<td>op7</td>
<td>0.40</td>
<td>0.40</td>
<td>0.40</td>
</tr>
<tr>
<td>op8</td>
<td>0.34</td>
<td>0.33</td>
<td>0.31</td>
</tr>
<tr>
<td>op9</td>
<td>0.001</td>
<td>0.0008</td>
<td>0.002</td>
</tr>
</tbody>
</table>

same principle that guided our parameter estimation in the single distribution case. To perform maximum likelihood estimation for mixture models, we used our own implementation of the well-known Expectation-Maximization (EM) algorithm [12]. The EM algorithm is a hill-climbing algorithm that attempts to find values of the parameters that will maximize the likelihood of the data. It iteratively alternates between determining the conditional probabilities that points belong to each of the mixture components and computing the parameters of each mixture component. A disadvantage of the EM algorithm is that it is not guaranteed to find the global maximum of the likelihood function; rather, it may get stuck and return parameters that achieve a local optimum of the likelihood. Thus, it is sensitive to the initialization. A technique that is used to ameliorate this issue is random restarts, where we use several random initializations and choose the parameters from the run which achieved the highest likelihood value. We did not employ this technique because we noticed that since our data is so skewed, with one large mode and one or more lesser modes, random initializations resulted in most of the mixtures being concentrated at the large mode. Instead, we used a deterministic scheme in which we initially assigned a certain quantile range of the data to each mixture. This technique resulted in a better spread among the mixture components.

An important task when using mixture models is to choose the number of mixtures, as this is a necessary input to the EM algorithm. One way to do so is to use cross validation, which allows us to choose a value that results in the maximum likelihood value yet avoids overfitting our data. We used the standard technique of $k$-fold cross validation with $k = 10$. We employed this process for each operator with each of our four types of distributions. Our use of cross validation allowed us to choose the number of mixtures for each operator under each type of mixture model. In Figure 7, we provide the cross validation results when fitting a mixture of Weibulls to the op3 data. In this case, we chose to use five mixtures, since the median log likelihood is maximized at this value. We prefer to go by the median since it is less vulnerable to outlying values.

In Figure 8, we present the results from fitting mixture models of each type to our operator 3 data. In comparison to the fits shown in Figure 6, which were achieved using single distributions, we notice that mixture models were better able to approximate the asymmetrical shape of our distribution. We also notice that the tails of the mixture models are much heavier than the tails of their corresponding single distribution fits, even though the fits’ multimodality is not obvious in the figure. The CDF, shown on a log scale, allows us to compare all the fits with the actual data. We see that while the mixture of normals and mixture of exponentials were able to predict the actual quantiles well in some cases, they are a poor match for the distribution overall. The mixture of gammas and mixture of Weibulls each produced a fit that matches the actual distribution throughout its latency domain. We note that the 99th-percentile latency predicted by the mixture of Weibulls is more accurate than that predicted by the mixture of gammas; thus, for this operator data, we chose to use a mixture of Weibulls model. We employed a similar process for operators 1, 2, 4, and 5.

**Discussion**

We may conclude from our modeling process that a desire to fit the tail of a distribution makes a dramatic impact on whether or not a model is deemed acceptable. We observed that it is straightforward to find a
Figure 6: Single distribution fits to data for operator 3. All of these models are quite different from the data’s distribution and produce highly inaccurate estimates of the 99th-percentile latency. These models are therefore unsatisfactory.

model that will achieve a good prediction of the mean and other low quantiles of the distribution; even the single distributions allowed us to do that. However, much effort must be expended to find a model that will fit the tail well, such as the empirical or mixture models. We also discovered that the assumption from queueing theory that service times follow an exponential distribution do not hold even in this relatively simple setup.

4.2 Query Modeling

Our modeling target is SCADr, a clone of Twitter that was written by some of the students in the RAD Lab\(^9\) at UC Berkeley. We provide a partial PIQL spec for SCADr in Figure 9. This spec includes several queries that provide SCADr’s functionality. SCADr allows users to generate “thoughts”. Users may subscribe to other users’ thought feeds. The \texttt{needsApproval} query returns a list of the users to whom a given user has submitted subscription requests that have not yet approved the user’s requests, as subscriptions must

\(^9\)http://radlab.cs.berkeley.edu/
Figure 7: Log likelihood values obtained from 10-fold cross validation used to choose number of mixtures for fitting mixture of Weibulls to op3 data. We chose to use 5 mixtures since this resulted in the highest median log likelihood.
three model types. The most obvious observation from this figure is that the exponential performs relatively poorly for all four queries. We note that the empirical and mixture models produced very similar results; thus, we may conclude that the models we chose allowed us to capture the empirical operator distributions well.

In Table 3, we provide the actual and predicted values for the median, 90th-percentile, and 99th-percentile latency for each of the four queries. We observe that the median and 90th-percentile values are separated by a factor of 2 or 3; thus, predicting the median would be of limited usefulness for reasoning about the tail of the latency distribution. This problem becomes more severe when considering the difference between the median and the 99th-percentile values, which are sometimes separated by an order of magnitude.

Due to the challenging nature of predicting the 99th percentile, we focus our attention on this case. We observe that the empirical approach was usually the best, where we consider the measure of predictive accuracy to be the percentage of the actual value by which the prediction differs from the actual. An exception to this is in the prediction for myFollowing, where the mixture approach had a much higher accuracy. For all four queries, the empirical prediction of the 99th percentile was within 20% of the actual.
Figure 9: An excerpt of the PIQL spec for SCADr, a Twitter clone.

```
ENTITY user
{
    string name,
    string password,
    string email,
    string hometown,
    string profileData
    PRIMARY(name)
}

ENTITY thought
{
    int timestamp,
    string thought
    PRIMARY(owner, timestamp)
}

ENTITY subscription
{
    bool approved
    PRIMARY(owner, target)
}

QUERY needsApproval
FETCH subscription
    OF user BY target
    WHERE user = [this] AND subscription.approved = false
    LIMIT [1: count] MAX 100

QUERY userByHometown
FETCH user
    WHERE hometown = [1:hometown]
    LIMIT 10 MAX 10

QUERY myFollowing
FETCH user
    OF subscription BY target
    OF user me BY owner
    WHERE me=[this]
    LIMIT [1: numPage] MAX 30

QUERY myThoughts
FETCH thought
    OF user BY owner
    WHERE user=[this]
    LIMIT [1: numPage] MAX 10
```

Figure 10: Physical query plan for myFollowing, generated by PIQL compiler.

The query for which our predictions were worst, myThoughts, is a particularly difficult query to model due to its extremely skewed shape.

In Table 4, we provide the average error for each method at each quantile of interest. The average error is computed as the average of the absolute value of the percent difference between the method’s prediction and the corresponding actual value for each query. As expected, the empirical method provides the best overall results, while the mixture model is a few percent worse. The exponential method provides results that are much worse overall.

We believe that the limiting factor of the accuracy of our modeling is the empirical operator data gathered as described in Section 4.1.1. In future work, we hope to improve this by compiling operator data at different times of day to lessen our exposure to noise from the EC2 environment. The new version of SCADS, which has recently become available, may present another opportunity to reduce the noise in our data.

Overall, we determine that the methodology we have presented is a flexible way to combine component models to predict the latency of higher-level structured requests. Without basing our models on query data but rather on operator data, which is available at compile time, we have achieved prediction within 20% for the challenging tail. We recall that the overall goal of this work was to help developers determine whether or not their queries would result in SLO violations. Predicting the 99th-percentile query latency accurately is a means to this end. Our predictions would prove very useful for developers desiring to know more about the behavior of their services, as we could include a ±20% error bar with each prediction. Developers would then assess the relationship between the prediction range and the SLO; if the upper bound of the prediction range is less than the SLO, the developer may conclude that the query latency will be acceptable.
Figure 11: CDFs for several queries, demonstrating fits of several types. The query latency distributions predicted using the empirical and mixture models of the operator latency distributions are similar to each other and good matches to the actual query latency distribution. However, the prediction obtained via modeling the operator distributions with the exponential was poor.

5 Conclusions and Future Work

As SaaS providers obsessively tend to the latency of their services, they desire to detect when their applications’ queries will cause SLO violations. The increasing desirability of key-value stores for backend storage in web applications combined with PIQL leads to an appealing setting for query latency prediction. We have applied to this setting a well-known concept from the query optimization community wherein estimates of the total work performed by each operator involved in executing a query are combined to reason about the total work done by a query. In our setting, we can use this idea to actually predict even the tails of our queries’ latency distributions, which is vital to reasoning about whether or not they meet their SLOs. We have estimated the latency distributions of several queries for SCADr, a clone of Twitter, so that our estimates even for the 99th percentile of the distributions are within 10 ms, or 20%, of the actual values.

Our modeling process was guided by our desire to achieve accurate fits to the distributions’ tails. Though it was relatively easy to find models that predicted the mean and low quantiles of the target distributions well, as even the single distributions allowed us to do, more flexible models such as empirical and mixture...
Table 3: Actual and predicted values for median, 90th-percentile, and 99th-percentile latency for each query. The empirical approach is the best overall, with all 99th-percentile predictions within 20% of the actual values. Percentages given are a value’s percentage of the actual value.

<table>
<thead>
<tr>
<th>Query</th>
<th>Method</th>
<th>Median</th>
<th>90th Percentile</th>
<th>99th Percentile</th>
</tr>
</thead>
<tbody>
<tr>
<td>needsApproval</td>
<td>actual</td>
<td>9.98 ms</td>
<td>100%</td>
<td>34.26 ms</td>
</tr>
<tr>
<td></td>
<td>empirical</td>
<td>9.49 ms</td>
<td>95.14%</td>
<td>32.26 ms</td>
</tr>
<tr>
<td></td>
<td>exponential</td>
<td>12.52 ms</td>
<td>125.48%</td>
<td>28.60 ms</td>
</tr>
<tr>
<td></td>
<td>mixture</td>
<td>9.55 ms</td>
<td>95.72%</td>
<td>36.27 ms</td>
</tr>
<tr>
<td>userByHometown</td>
<td>actual</td>
<td>8.26 ms</td>
<td>100%</td>
<td>29.62 ms</td>
</tr>
<tr>
<td></td>
<td>empirical</td>
<td>7.64 ms</td>
<td>92.46%</td>
<td>30.08 ms</td>
</tr>
<tr>
<td></td>
<td>exponential</td>
<td>10.15 ms</td>
<td>122.92%</td>
<td>26.32 ms</td>
</tr>
<tr>
<td></td>
<td>mixture</td>
<td>7.67 ms</td>
<td>92.81%</td>
<td>33.15 ms</td>
</tr>
<tr>
<td>myFollowing</td>
<td>actual</td>
<td>12.41 ms</td>
<td>100%</td>
<td>39.90 ms</td>
</tr>
<tr>
<td></td>
<td>empirical</td>
<td>10.07 ms</td>
<td>81.13%</td>
<td>34.37 ms</td>
</tr>
<tr>
<td></td>
<td>exponential</td>
<td>13.69 ms</td>
<td>110.34%</td>
<td>31.97 ms</td>
</tr>
<tr>
<td></td>
<td>mixture</td>
<td>10.08 ms</td>
<td>81.22%</td>
<td>40.14 ms</td>
</tr>
<tr>
<td>myThoughts</td>
<td>actual</td>
<td>3.59 ms</td>
<td>100%</td>
<td>6.30 ms</td>
</tr>
<tr>
<td></td>
<td>empirical</td>
<td>3.73 ms</td>
<td>103.88%</td>
<td>5.74 ms</td>
</tr>
<tr>
<td></td>
<td>exponential</td>
<td>4.78 ms</td>
<td>133.26%</td>
<td>9.13 ms</td>
</tr>
<tr>
<td></td>
<td>mixture</td>
<td>3.75 ms</td>
<td>104.33%</td>
<td>8.03 ms</td>
</tr>
</tbody>
</table>

Table 4: Average error for our three methods considered at each quantile of interest. The empirical approach yields the lowest average error at the 99th percentile, with the mixture model approach a few percent worse. The exponential approach yields significantly larger average error than the other two methods at each quantile of interest.

<table>
<thead>
<tr>
<th>Method</th>
<th>Median</th>
<th>90th Percentile</th>
<th>99th Percentile</th>
</tr>
</thead>
<tbody>
<tr>
<td>empirical</td>
<td>8.79%</td>
<td>7.55%</td>
<td>11.62%</td>
</tr>
<tr>
<td>exponential</td>
<td>23.0%</td>
<td>23.11%</td>
<td>28.21%</td>
</tr>
<tr>
<td>mixture</td>
<td>8.64%</td>
<td>11.47%</td>
<td>16.73%</td>
</tr>
</tbody>
</table>

models were required to predict the tails. In addition, we learned that queueing theory’s assumption that service times follow an exponential distribution does not hold for our system.

Going forward, we plan to use the mixture of experts technique [13] in order to predict query latency when the selectivity varies. Also, we will relax our assumption about the stability of the storage tier. In particular, we are interested in modeling query performance in the presence of data hotspots. We will also explore how our methodology of composing models of low-level components to predict the latency distribution of higher-level requests may extend to other domains, such as MapReduce or three-tier web applications.

References


