Obtaining and Managing Answer Quality for Online Data-Intensive Services

JAIMIE KELLEY, CHRISTOPHER STEWART, and NATHANIEL MORRIS,
Ohio State University
DEVESH TIWARI, Northeastern University
YUXIONG HE and SAMEH ELNIKETY, Microsoft Research

Online data-intensive (OLDI) services use anytime algorithms to compute over large amounts of data and respond quickly. Interactive response times are a priority, so OLDI services parallelize query execution across distributed software components and return best effort answers based on the data so far processed. Omitted data from slow components could lead to better answers, but tracing online how much better the answers could be is difficult. We propose Ubora, a design approach to measure the effect of slow-running components on the quality of answers. Ubora randomly samples online queries and executes them a second time. The first online execution omits data from slow components and provides interactive answers. The second execution uses mature results from intermediate components completed after the online execution finishes. Ubora uses memoization to speed up mature executions by replaying network messages exchanged between components. Our systems-level implementation works for a wide range of services, including Hadoop/Yarn, Apache Lucene, the EasyRec Recommendation Engine, and the OpenEphyra question-answering system. Ubora computes answer quality with more mature executions per second than competing approaches that do not use memoization. With Ubora, we show that answer quality is effective at guiding online admission control. While achieving the same answer quality on high-priority queries, our adaptive controller had 55% higher peak throughput on low-priority queries than a competing controller guided by the rate of timeouts.

11

Categories and Subject Descriptors: H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Performance Evaluation (Efficiency and Effectiveness)*

General Terms: Management, Measurement

Additional Key Words and Phrases: Answer quality, big data, services

ACM Reference Format:
Jaimie Kelley, Christopher Stewart, Nathaniel Morris, Devesh Tiwari, Yuxiong He, and Sameh Elnikety. 2017. Obtaining and managing answer quality for online data-intensive services. ACM Trans. Model. Perform. Eval. Comput. Syst. 2, 2, Article 11 (April 2017), 31 pages.
DOI: http://dx.doi.org/10.1145/3055280

1. INTRODUCTION

Online data-intensive (OLDI) services, such as product recommendation, sentiment analysis, question answering, and search engines, power many popular Web sites

---

This work was supported by NSF grants CAREER CNS-1350941 and CNS-1320071, and also by the Oak Ridge Leadership Computing Facility managed by UT Battelle, LLC for the U.S. DOE (contract DE-AC05-00OR22725).
Authors' addresses: J. Kelley, C. Stewart, and N. Morris, The Ohio State University, Department of Computer Science and Engineering, 395 Dreese Laboratories, 2015 Neil Avenue, Columbus, OH 43210-1277; emails: kelley.530@osu.edu, cstewart@cse.ohio-state.edu, morris.743@osu.edu; D. Tiwari, Northeastern University, Electrical and Computer Engineering, 409 Dana, 360 Huntington Avenue, Boston, MA 02115; email: Tiwari@northeastern.edu; Y. He and S. Elnikety, Microsoft Research Redmond, 1 Microsoft Way, Redmond, WA 98052; emails: {yuxhe, samehe}@microsoft.com.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2017 ACM 2376-3639/2017/04-ART11 $15.00
DOI: http://dx.doi.org/10.1145/3055280

and enterprise products. Like traditional Internet services, OLDI services must answer queries quickly. For example, Microsoft Bing's revenue would decrease by $316M if it answered search queries 500ms slower [Forrest 2009]. Similarly, IBM's deep question-answering (DeepQA) implementation, Watson, would have lost to elite Jeopardy contestants if it waited too long to answer [Lenchner 2011; Ferrucci 2010]. However, OLDI and traditional services differ during query execution. Traditional services use structured databases to retrieve provably correct answers, but OLDI services use loosely structured or unstructured data. Extracting answers from loosely structured data can be complicated. Consider the OpenEphyra question-answering system [Schlaefer 2013]. Each query execution reduces text documents to potentially relevant phrases by finding noun-verb answer templates within sentences.

OLDI services use large quantities of data to improve the quality of their answers. For example, IBM Watson parsed 4TB of data for its Jeopardy competition [Ferrucci 2010]. The amount of data used by these services is growing; Wikipedia alone grew 116X from 2004 to 2011 [Wikipedia 2014]. However, large data also increases processing demands. To keep response time low, OLDI query executions are parallelized across distributed software components. These software components run in virtual machines distributed across cloud infrastructure. At Microsoft Bing, query execution invokes hundreds to thousands of components in parallel [Jalaparti et al. 2013]. Each component contributes intermediate data that could improve answers. However, some query executions suffer from slow-running components that take too long to complete. Since fast response time is essential, OLDI query executions cannot wait for slow components. Instead, they use anytime algorithms to compute answers with whatever data is available within response time constraints [Zilberstein 1996].

OLDI services use incremental computation over whatever data is available, returning the best available answers within response time constraints [He et al. 2012a; Jalaparti et al. 2013; Meisner et al. 2011]. Parallel components can have different processing requirements based on skew in partitioned data, but a hardware failure or software anomaly could also cause severe performance degradation [Attariyan et al. 2012]. Returning these best available answers prevents slow parallel components from slowing down interactive queries. However, omitting data from slow components could degrade answer quality [Ren et al. 2013; Falsett et al. 2004]. In this article, answer quality is the similarity between answers produced online and without omitting data from slow components [Kelley et al. 2015]. Queries achieve high answer quality when their execution does not suffer from slow components or when data omitted from slow components do not affect answers. Low answer quality means that omitted data from slow components have important contributions that would affect final answers significantly. Prior work has shown the virtue of adaptive resource management with regard to response time and quality of service [Spinner et al. 2014; Gandhi et al. 2014; Lama and Zhou 2012]. Adaptive management could also help OLDI services manage answer quality. For example, admission control traditionally performs a check before accepting a query to determining if the system has resources available; in this context, admission control could check recent high priority queries for low quality before admitting low-priority queries to the queue. In this way, admission control could stabilize answer quality for high-priority queries even under time-varying arrival rates by increasing shed of low-priority queries when answer quality drops.

Answer quality is hard to measure online because it requires two query executions. Figure 1 depicts the process of computing answer quality. First, an online execution provides answers within response time constraints by omitting data from slow components. Second, we define a mature execution to use all available data relevant to a query by waiting for all components to complete before producing mature answers. Finally, a service-specific similarity function computes answer quality. This work uses

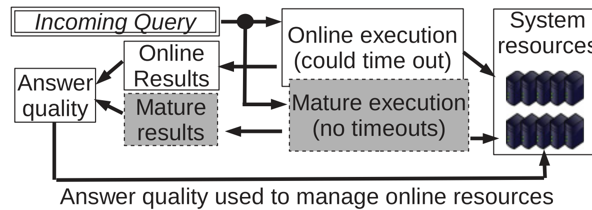

Fig. 1. Steps to measure answer quality online. Mature and online executions may overlap.

the true positive rate as the similarity function, but other functions are permissible, such as normalized discounted cumulative gain [Manning et al. 2008].

Prior research ran mature executions on dedicated offline testbeds [He et al. 2011; Kelley et al. 2013], but storage costs for offline testbeds grow as OLDI services ingest data. Further, a service's expected answer quality per query depends on its query mix. Changing query mixes and data can affect answer quality online, which is difficult to measure in offline testbeds [Kelley et al. 2013].

We present Ubora,[1] *a design approach to speed up mature executions.* Our key insight is that mature and online executions invoke many components with the same parameters. Memoization can speed up mature executions—for instance, a mature execution can complete faster by reusing data from its corresponding online execution instead of reinvoking components.

When a query arrives, Ubora conducts a normal online query execution, except it records intermediate data provided by each software component, including data not reflected in the online answers because they were omitted from slow components. After the slow components finish, Ubora computes mature answers using data recorded during and after the online execution. Implementing memoization for multicomponent OLDI services presents systems challenges. First, OLDI services span multiple software components. It is challenging to coordinate mature and online executions across software components without changing application-level source code. Ubora manages mature and online operating context. During mature executions, it uses network redirection to replay intermediate data from in-memory storage. Second, memoization speeds up computationally intensive components, but its increased bandwidth usage can also cause slowdown for some components. Ubora provides flexible settings for memoization, allowing each component to turn off memoization. We use offline profiling to determine which components benefit from memoization.

We have evaluated Ubora on four different open source OLDI services with varying degrees of complexity and data size. To be clear, Ubora's systems-level implementation is able to support these applications without modification to their source code. We compared Ubora to query tagging, which changes application source code to resume mature execution at the point when an online execution returned a response. We also compared timeout toggling, an approach that transparently applies the same context across all currently executing queries. Ubora completes mature executions nearly as quickly as query tagging with slowdown ranging from 8% to 16%. Ubora finishes mature executions 7X faster than timeout toggling. Finally, Ubora slows down normal, online query executions by less than 7%. We used Ubora to guide adaptive admission control. Ubora responded quickly to changing arrival rates, keeping answer quality above 90% during most of the trace.

This article is organized as follows. In Section 2, we put our contributions in the context of related work. We describe the structure of OLDI services in Section 3. Section 4 explains the motivation for our work. We present Ubora in Section 5. Section 6

[1]Ubora means "quality" in Swahili.

presents our implementation of query context tracking and profiling for memoization. In Section 7, we measure Ubora's performance using a wide range of OLDI benchmarks. In Section 8, we show that Ubora computes answer quality quickly enough to guide online admission control. Section 9 concludes, and Section 10 gives details regarding the availability of our open source Ubora implementation.

2. RELATED WORK

Ubora focuses on online systems, which trade answer quality for fast response times. Zilberstein [1996] first characterized similar applications as anytime algorithms. Like the OLDI workloads used with Ubora, anytime algorithms increase in result quality as they increase in computation time. The metric Zilberstein uses that is closest to our answer quality metric is accuracy, but he does not indicate how the exact answer is to be reached for comparison. His work indicates that anytime algorithms should have measurable quality, monotonically increase in quality as computation time increases, provide diminishing returns, and produce a correct answer when interrupted. Ubora broadens the category of applications that can use an answer quality metric beyond anytime algorithms, not requiring that applications can suspend and resume at any time, nor requiring that the optimal answer be determined in constant time.

2.1. Approximation for Performance

Recent work has focused on introducing approximation into existing systems to increase performance [Goiri et al. 2015; Jeon et al. 2013; Jalaparti et al. 2013].

ApproxHadoop [Goiri et al. 2015] integrates sampling of input data, user-defined approximate code versions, and execution of a subset of tasks into Hadoop. ApproxHadoop allows the user to set error bounds within a confidence interval, set a specific data sampling ratio, or specify the percentage of tasks to drop to increase performance. Ubora enables users to similarly manage resources based on the online answer quality trace.

Sequential search may terminate early on a server if the processing of the ranked documents goes below a certain relevance. Since parallel search over the same index will generally result in more processing per query, Jeon et al. [2013] reduces this wasted work by keeping the order in which documents are processed sequential. Although this is not necessary under low load, higher loads are more impacted by wasted work. The authors adaptively change the amount of parallelism per query based on the current system load.

Kwiken is an optimization framework for lowering tail latency in Bing [Jalaparti et al. 2013]. Kwiken uses techniques that include allowing the return of incomplete results, reissuing queries that lag on different servers, and increasing apportioned resources. It calculates incompleteness as utility loss based on whether the answer returned contains the highest-ranked document for certain stages, and in other stages this is the percentage of parallel components that had not responded. Our work differs from their solution in that we focus on speeding up the mature execution with which to produce answer quality. Additionally, our framework provides for provisioning of other resources based on answer quality.

In between executing queries, DICE uses wait time to speculatively execute the queries most likely to be asked next and cache these results [Kamat et al. 2014]. DICE also implements timeouts on total query execution so that even if only some of the data is assembled in postprocessing, an answer will be available. DICE sampled data proportionately to the most likely speculative queries found. DICE is very similar in two ways to Ubora, in that we also use cached data from queries hidden from the user. However, DICE uses this cached data to improve the latencies of further queries within a user session rather than for mature executions. DICE also uses sampling to reduce the resources spent running queries not sent by an end user.

2.2. Query Tagging

One of the approaches we used to increase the maturity of answers as an alternative to Ubora is specially tagging each query with context clues. Several recent works have illustrated the use of query tagging for approximated workloads.

A proxy-based approach can dynamically scale quality of Web results across different end platforms [Fox et al. 1998]. Fox et al. [1998] use lossy compression to distill specifically typed information down to the portions with semantic value. Their proxy adapts on demand to fit the needs of a client. The authors access mature execution from a Web server and approximate this data to meet the needs of a range of client platforms. We instead focus on services that provide online results and measure the amount of approximation present.

SocialTrove tags queries with data regarding the minimum diversity expected among the returned samples from data-intensive applications. Instead of measuring answer quality, SocialTrove uses application-specific similarity metrics to automatically cluster and summarize social media data [Amin et al. 2015].

He et al. [2012b] use a budget consisting of total execution time for current queries to determine whether and how long to schedule a query. Each query is tagged with an amount of processing time based on this budget. Their work uses a feedback mechanism to help ensure that the desired response times are being met and an optimization procedure to schedule based on request service demands and response quality profiles. Their algorithm takes advantage of prior knowledge regarding the overall concave quality profile of Microsoft Bing to estimate the individual request quality profile rather than attempting to measure request quality with a mature execution.

Similarly, Zeta was designed to better schedule requests in online servers for high response quality and low response quality variance [He et al. 2012a]. Zeta focuses on online services that produce partial results under a deadline, where trading additional computation time produces diminishing returns in additional response quality. Their response quality, like our answer quality, uses an application-specific metric to compare a partially executed request to a full execution. They measure their response quality offline.

Ren et al. [2013] tag queries with deadline and arrival time to implement their fast old and first (FOF) algorithm, which schedules incoming, unknown requests on the fastest core available in a heterogeneous processor, then migrates requests from slower cores to faster cores as jobs finish. They explain how heterogeneous processors can execute long requests on faster cores and shorter requests on slow cores to achieve high throughput and high quality. Their algorithm can improve answer quality and throughput in heterogeneous processors as compared to homogeneous processors with the same power budget. The authors used Bing without deadlines in a controlled setting to produce mature executions and then used this data in their simulation study.

2.3. Timeout Toggling: Adaptive Configuration

A second approach that can be used to achieve mature executions in an online setting is to dynamically change the configuration at the application level via argument specification. Later in this article, we compare Ubora to timeout toggling, which uses this approach to extend query processing time.

As in our work, Kephart and Lenchner [2015] focus on OLDI computations occurring across multiple components working together. Their system changes configuration to maximize a utility function, then displays interpreted system responses to the user and corrects computations when the user indicates incorrect analysis.

Our work focuses on data-intensive applications and uses application-specific similarity metrics to study answer quality. Previous work has used answer quality to reduce

costs in cache provisioning for online, natural language applications [Kelley et al. 2013]. However, the work of Kelley et al. [2013] adapted cache configuration offline based on answer quality.

ISPEED uses a deadline-agnostic scheduler to explore anytime algorithm workloads without information regarding individual queries [Zheng et al. 2015]. ISPEED focuses on maximizing total utility over all jobs in the cluster and ignores concerns regarding individual query deadlines. In addition to the utility functions used in He et al. [2012a, 2012b], ISPEED also facilitated a user study for the Google search engine to find its average utility function [Zheng et al. 2015].

SkewTune mitigates skew for user-defined MapReduce programs by reconfiguring the amount of data per task online [Kwon et al. 2012]. SkewTune has similar goals to Ubora with regard to transparency and minimal overhead for untuned queries but dynamically redistributes data from the task expected to take the longest to complete instead of using approximation. Our work shows that data skew is one of the motivations for approximation in OLDI services.

2.4. Adaptive Resource Allocation

Also highly related to Ubora is the area of adaptive resource allocation [Spinner et al. 2014; Gandhi et al. 2014; Lama and Zhou 2012]. Spinner et al. [2014] presented a library for estimating resource demands with seven different approaches. Using their library, it is possible to control how often the resource use is sampled, and when and for how long to perform the estimate.

DC2, an autoscaling cloud service, can learn an application's system parameters and scale based on its understanding of resource requirements [Gandhi et al. 2014]. Without direct knowledge of the application's needs, DC2 relies on user-specified SLA information, virtual CPU statistics, and knowledge of request URLs to autoscale. Like Ubora, DC2 is mostly transparent, with key information provided by the user. However, DC2 focuses directly on autoscaling.

AROMA is an automated resource provisioning system that uses Hadoop parameter configuration and resource allocation in a heterogeneous cloud environment to target quality of service while minimizing cost [Lama and Zhou 2012]. Instead of directly profiling each workload and regulating resources based on answer quality, AROMA profiles each workload for a short time on a staging cluster before matching the workload's signature to a cluster of workloads with a set of associated resources.

3. BACKGROUND ON OLDI SERVICES

Query executions differ fundamentally between OLDI and traditional Internet services. Traditional Internet services have query executions that process all data retrieved from well-structured databases, often via SQL (i.e., LAMP services) [Lawton 2005]. Correct query executions produce answers with well-defined structure—for instance, answers are provably right or wrong. In contrast, OLDI queries execute algorithms that increase in accuracy as time allows, including anytime algorithms [Zilberstein 1996]. They produce answers by discovering correlations within large quantities of data. OLDI services produce good answers if they process data relevant to query parameters.

OLDI answers improve in quality with larger datasets. For example, IBM Watson competed at Jeopardy using 4TB of mostly public-domain data in distributed memory [Ferrucci 2010]. One of Watson's data sources, Wikipedia, grew 116X from 2004 to 2011 [Wikipedia 2014]. However, it is challenging to analyze an entire large dataset within strict response time limits. This section provides background on the software structure of OLDI services that enables the following:

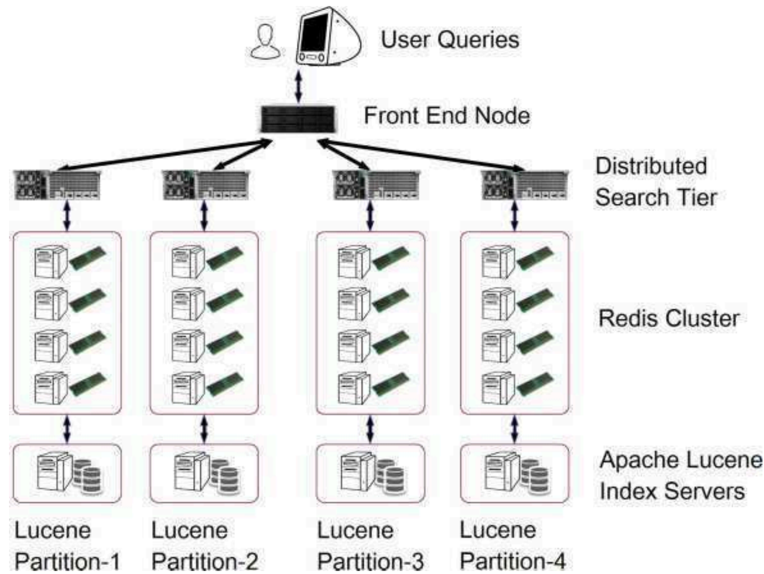

Fig. 2. Execution of a single query in Apache Lucene. Adjacent paths reflect parallel execution across data partitions.

(1) Parallelized query executions for high throughput
(2) Returning best-effort, online answers based on partial data to prevent slow software components from delaying response time.

Parallelized query execution. Figure 2 depicts the query execution path in a service based on Apache Lucene, a widely used open source information retrieval library [Lucid Imagination 2010]. This query execution path invokes 25 software components. Components in adjacent columns can execute in parallel. A front-end software component manages network connections with clients, sorts results from components running distributed search logic, and produces a running list of answers from results so far received. This list is returned to the user. Distributed search software components parse the query, request a wide range of relevant data from partitioned storage components, and collect data returned within a given timeout. Data is retrieved from either (1) an in-memory Redis cluster that caches a subset of index entries and documents for a Lucene index server or (2) the Lucene index server itself, which stores the entire index and data on relatively slow disks.

The Lucene service in Figure 2 indexes 23.4 million Wikipedia and New York Times documents (pages + revisions) produced between 2001 and 2013. Queries access in parallel Lucene indexes partitioned across multiple virtual machines. A partition is a subset of data. Each parallel subexecution (i.e., a vertical column) computes intermediate data based on its underlying partition. When intermediate data from parallel executions over each partition are combined, they compose a query response. This intermediate data is combined at a software component layer that may execute many queries in parallel, but processing for each query is done in sequential order (i.e., the front-end node).

OLDI services also parallelize query executions via partial redundancy. In this approach, subexecutions compute intermediate data from overlapping partitions. The query execution weights answers based on the degree of overlap and aggregate

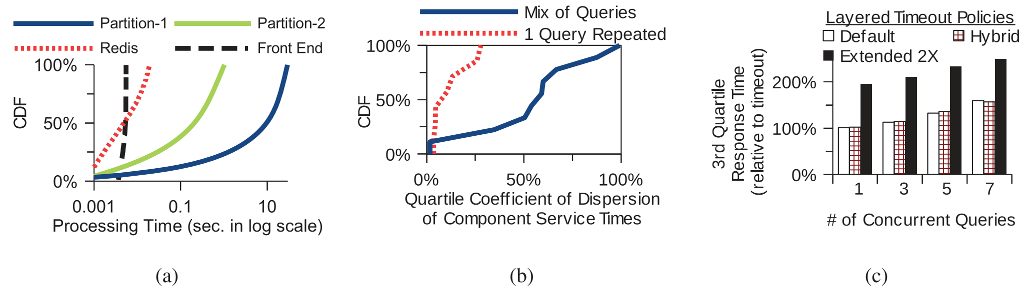

Fig. 3. Experimental results with an Apache Lucene cluster. (a) OLDI components exhibit diverse processing times. (b) Query mix increases variability. (c) Timeout policies mask variation in favor of fast response times.

data processing per partition. Consider a product recommendation service. Its query execution may spawn two parallel subexecutions. The first finds relevant products from orders completed in the past hour. The second considers the past 3 days. The service prefers the product recommended by the larger (3-day) subexecution. However, if the preferred recommendation is unavailable or otherwise degraded, the results from the smaller parallel subexecution help.

Online answers are best effort. In traditional Internet services, query execution invokes software components sequentially. The query response time depends on aggregate processing times of all components. In contrast, OLDI query executions invoke components in parallel. The processing time of the slowest component determines response time. Figure 3(a) quantifies component processing times in our Apache Lucene service. The query workload from Google Trends and hardware details are provided in Section 7. Processing times vary significantly from query to query. Note that the x-axis is shown in log scale. Lucene index servers can take several seconds on some queries even though their typical processing times are much faster. Further, processing time is not uniform across partitions. For example, a query for "William Shakespeare" transferred 138KB from the partition 4 execution path but only 1KB from the partition 1 execution path. This is an instance of data skew increasing the time to process partitions disproportionately. Partition 4 hosted more content related to this query even though the data was partitioned randomly.

4. MOTIVATION

Extending some timeouts is not enough to achieve mature executions. Many OLDI services prevent slow components from delaying response time by returning answers before slow components finish. Specifically, query executions trigger timeouts on slow components and produce answers that exclude some intermediate data. Timeouts effectively control response time. In our Apache system, we set a 2-second and a 4-second timeout in our front-end component. Average response time fell. In addition, third quartile response times were consistently close to median times, showing that timeouts also reduced variance. Unfortunately, query executions that trigger timeouts use less data to compute answers. This degrades answer quality. For data-parallel queries, answer quality degrades if the omitted data is relevant to query parameters.

Our Apache Lucene service answers mature queries too slowly to support interactive response times. First, since query mix changes often, the resource demands of parallel execution paths in OLDI services will vary significantly. Without excessive overprovisioning, some queries will inevitably have paths that require more processing time than interactive services permit. To effect interactive response times, these paths will execute only for a preset time before halting, which causes answer quality to vary

significantly. A second consequence arises because data growth affects data skew [Ahmad et al. 2008].

Impact of timeout policies. To examine the impact of timeout policies on the maturity of online executions, we set a 3-second end-to-end timeout in our workload generator. The front-end software component timed out connections to the distributed search software components in 2 seconds. The distributed search software components timed out connections to Redis and Lucene index servers in 0.8 seconds. Such layered timeouts allow for some bounded response-time variation from components but prevent large variation and failures from degrading end-to-end response time. We studied the impact of layered timeouts on the maturity of online executions. Figure 3(c) plots concurrency versus response time under this timeout policy. We compared the policy to default timeouts, doubled timeouts, and a hybrid approach. This hybrid approach used a layered timeout, where two Lucene software components and a Redis software component had 10X timeouts, but all others followed the default policy. Timeouts controlled response times well, triggering best-effort results, which sped up some queries by 88X. Third quartile response times were consistently close to median times, showing that timeouts also reduced variance.

The hybrid approach is promising because it targets specific components and keeps response times low, but it fails to affect the results maturity. Even though some components had long timeouts, their executions were truncated by timeouts at higher layers. In contrast, extending all timeouts produced mature results, but response time increased proportionally to the timeout extension.

Taken all together, our results show timeouts control response time for online executions effectively. Timeouts prematurely halt components affected by data skew, especially components replicated across data partitions. However, the exact components that trigger timeouts vary at runtime. Naively extending all timeouts may produce mature results, but this also increases average response time linearly. These lessons influenced our system design.

Variation of mature executions. Given a query's parameters and data partitioning scheme, some components will be used more heavily than others. Data skew persists despite random hashing.

Processing times on partition 4 were 2.6X slower than partition 1. For example, queries related to the people who participated in the 2008 U.S. presidential election pulled much more heavily from Lucene partition 2 than Lucene partition 1, reflecting our partitioning strategy based on creation date.

We measure variation using the same query issued repeatedly and capturing the quartile coefficient of dispersion (QCoD; $QCoD = \frac{Q3-Q1}{Q3+Q1}$) for the response times of each component in our system. The QCoD is a nonparametric metric like the coefficient of variation, but it is more robust to skew caused by outliers. Smaller numbers indicate that data is less spread out. Figure 3(b) shows that the quartile coefficient of variation increased significantly under a mix of queries compared to reissuing the same query. This illustrates that per-component processing times vary from query to query and across execution paths.

The variation of mature executions has practical consequences. First, since query mix changes often, the resource demands of parallel execution paths in OLDI services will vary significantly. Without excessive overprovisioning, some queries will inevitably have paths that require more processing time than interactive services permit. Those paths will halt prematurely, causing answer quality to vary significantly. A second consequence arises because data growth affects data skew [Ahmad et al. 2008]. Answer quality may differ significantly after an OLDI service ingests new data, even if query mix is stable. Both query mixes and data growth change online, so answer quality also

changes online. Query mixes are nonstationary within 5-minute and 1-hour windows [Stewart et al. 2007]. Data growth is ubiquitous. Facebook's Scuba ingests 1M events per minute [Barykin et al. 2013]. TripAdvisor ingests 86K user reviews per hour [Gelfond 2011]. These observations support our observation that answer quality changes dynamically.

We showed that response time variations across components and queries are present in modern OLDI services. These variations may be caused because of query mixes and data skew. These results indicate that it is challenging to achieve mature results online by simply controlling a few components or workflow paths.

Ubora reduces resource needs by measuring answer quality for only randomly sampled queries, reusing computation from online executions and scheduling mature executions during low concurrency periods.

5. DESIGN

By design, Ubora measures the answer quality of online query executions by comparing answers produced with and without timeouts. It reduces cost compared to other approaches by using existing online resources and employs memoization to speed up mature query executions. Memoization also reduces the overhead of executing mature queries online by allowing reuse of previous intermediate results from targeted software components. Ubora further uses sampling to reduce overhead.

5.1. Design Goals

We designed Ubora around the following goals:

Q1

- *Timeliness*: The primary goal of Ubora is to measure answer quality quickly enough to enable resource management based on the results. For our purposes, the challenge was to acquire mature executions quickly from an online environment. Because query mix changes over time, it is necessary to replay queries issued to the online service.
- *Transparency*: Require no code changes to software components. The secondary goal of Ubora is to support a broad range of OLDI services composed of multiple different software components. For this to succeed, by design we require no changes to the code of software components used by the service. Instead of changing the services to require additional contextual information (i.e., online or mature), we use a middleware framework that tracks query context.
- *Low overhead*: Online queries need to execute quickly, so we do not want slowdown. To this end, we introduce two optimizations that reduce slowdown in online queries. Sampling only a percentage of incoming online queries greatly reduces the overhead, as does delaying replay when needed to avoid queuing delay with online queries.
- *Low cost*: Although it is possible to compute mature results offline using an online query trace, this requires an increase in resources allotted to the service. In this section, we present an analysis of the cost of these additional resources to motivate why our design instead only uses resources currently available to the service.

5.2. Timeliness

Our design is mostly motivated by timeliness and transparency. We aim to overlap the mature execution as much as possible with the online execution. Once the online execution has completed, we keep targeted components executing in the background until they have fully processed the online query and cache the results in distributed in-memory storage. When we then replay the online query, we use these cached results instead of accessing the targeted component.

Figure 4 depicts memoization in Ubora. During online query execution, Ubora records intercomponent communication. It allows only front-end components to time out. Components invoked by parallel subexecutions complete in the background. As shown on

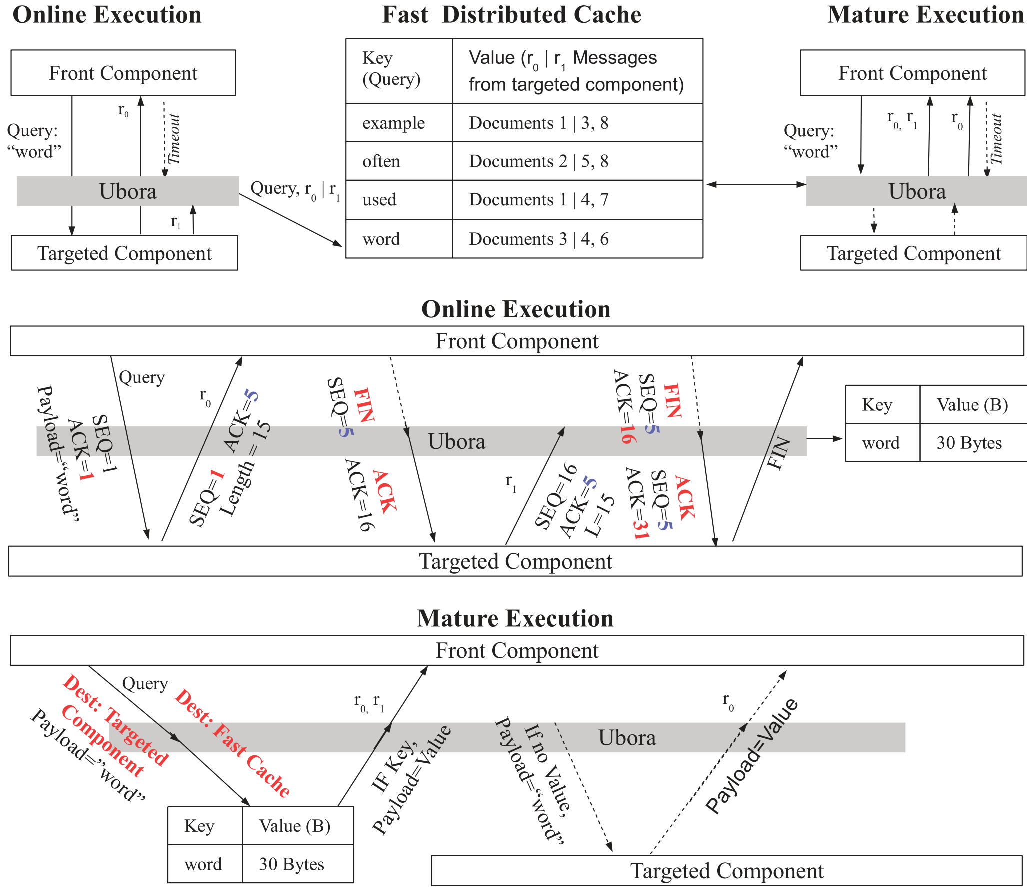

Fig. 4. Memoization in Ubora. Arrows reflect messages in execution order (left to right). Dotted lines in the online execution indicate communications that are transformed from their original purpose. Dotted lines in the mature execution indicate communications that happen on occasion, as needed for correctness.

the left side of Figure 4, without Ubora, this example front-end component invokes a component with a TCP payload containing a query, receives message r_0, and then times out. The front-end component then uses a FIN packet to trigger a timeout for the invoked component, stopping its execution before completion. A dotted line in Figure 4 shows Ubora blocking the trigger from the front-end component, allowing the invoked component to complete a mature execution. It records output messages before and after the front-end times out, in this case $r_0 + r_1$. These messages are cached in fast in-memory storage distributed across the least utilized machines.

With Ubora, front-end components still answer online queries within strict response time limits. As shown in Figure 4, the front-end component uses r_0 to produce an online answer. After all subexecutions for a query complete, Ubora re-executes the front end, as if a new query arrived. However, during this mature execution, Ubora intercepts messages to other components and serves response messages r_0 and r_1 from the cache (i.e., memoization). The cache delivers messages with minimal processing or disk delays. During this mature execution, the front end uses both $r_0 + r_1$ to produce a mature answer. For correctness in mature executions, Ubora may connect to the targeted component if data is not in the fast cache. This connection is shown as dotted lines in Figure 4.

Mature and online results are stored in the cache. After replay completes, answer quality is computed by callback functions or scheduled jobs. Each service may define answer quality differently by providing its own function. Example functions include the weighted top K, normalized discounted cumulative gain, or true positive rate. In our experience, these functions complete quickly relative to query execution time.

5.3. Transparency

Queries execute under different contexts tracked by Ubora, depending on whether they are online executions being recorded, normal online executions, or mature executions being replayed. Such execution context requires coordination across distributed nodes and concurrent queries. Memoization should be implemented differently depending on available systems support for execution context tracking. Further, replayed executions may steal resources from high-priority online executions. The challenge is to minimize queuing interference.

5.4. Low Overhead

Queries issued by users or other external sources must complete quickly. For sampled queries, a query's online execution completes during record mode. Record mode adds light overhead by sending messages to the cache, but importantly, most components execute under normal timeout settings. This approach is similar to the hybrid approach in Section 4. The query completes quickly due to layered timeouts, whereas mature component-level executions happen asynchronously. In contrast, naively extending all timeouts increases response times. We introduce two optimizations to keep overhead low.

Replay mode can be postponed to reduce interference. Replays can be temporarily postponed to avoid queuing delays with online executions, but they must finish in a timely fashion to impact online management. Fortunately, components operate under normal timeout settings during replay mode. Replay mode completes quickly and predictably.

Sampling frequency versus slowdown and representativeness. Ubora uses sampling to reduce the aggregate overhead of mature executions. Recall that mature executions use a lot more resources than online executions. Services do not have enough idle resources to complete a mature execution for every online execution. However, sampling too infrequently can inhibit online management because answer quality is produced too late or is not representative. The sampling rate allows managers to trade throughput on mature executions for processing overhead.

Our design allows each service to set a sampling rate that matches its hardware and query mixes. Online executions selected for record and replay suffer longer service times, but unsampled queries execute normally.

5.5. Low Cost

Although it would be possible to compute mature executions on an offline testbed using the online queries, this is a costly proposition, requiring 100% additional infrastructure cost in the worst case. To keep the cost of measuring answer quality low, we opt to share the resources already allocated to the service.

Memoization and replay modes redirect network traffic from nodes used to process online queries. The approach does not require an offline testbed. Offline testbeds require costly data replication for accurate mature executions. As datasets grow, testbed costs grow as well. Next we show that offline testbeds are expensive by analyzing the growth of Wikipedia data.

Figure 5 shows the expected cost of our Apache Lucene setup hosting the Wikipedia data. We studied the dataset from 2004 to 2009. In those years, the dataset size (d_i)

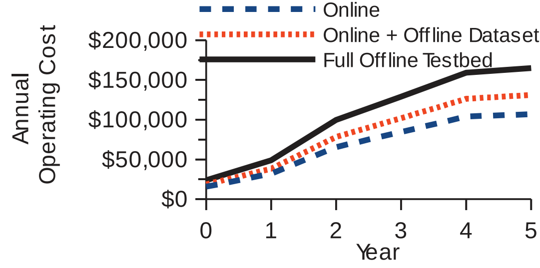

Fig. 5. Annual operating costs for Apache Lucene on EC2 with Wikipedia growth rates.

grew by 130%, 150%, 110%, 70%, 40%, and 23% respectively, ending at 4TB [Wikipedia 2014]. Hard drive capacity (h_i) grew at 20% annually, starting at 250MB. We used Amazon EC2 pricing for reserved nodes ($2,400/year) and EBS storage pricing ($1.20 GB/year). We set annual inflation to 2%. The number of partitions is captured by Equation (1). The cost for online resources (i.e., Lucene partitions (p_i), four cache instances per partition and one distributed search component per partition) is captured by Equation (2). The least expensive cost of an offline testbed that fully replicates data is modeled in Equation (3). It adds additional instances and doubles EBS costs but does not include offline resources for processing. A full offline testbed with one distributed search component and one cache cluster per partition is modeled by Equation (4).

$$p_i = \frac{d_i}{h_i} \tag{1}$$

$$C_{Online} = (6p \times \$2400) \times 1.02^i + (d_i \times \$1.2) \times 1.02^i \tag{2}$$

$$C_{Data} = (7p \times \$2400) \times 1.02^i + (2 \times d_i \times \$1.2) \times 1.02^i \tag{3}$$

$$C_{Offline} = (9p \times \$2400) \times 1.02^i + (2 \times d_i \times \$1.2) \times 1.02^i \tag{4}$$

Figure 5 depicts the cost of data growth and offline testbeds. Data growth alone would increase costs for our Apache Lucene setup by 3X between 2005 and 2009. We note that Wikipedia's reported hosting costs grew by 4.3X during the same period. A full offline testbed would increase costs by 50%. An offline dataset alone would increase costs by 18%. Further with inflation, the relative cost of an offline dataset compared to the cost of online nodes would grow from 18% in 2004 to 22% in 2009. Record, cache, and replay reduces operating costs by 35% compared to approaches that maintain a full offline testbed.

Prior work proposed subsampling data to reduce costs [Kelley et al. 2013]. A sampled dataset can only approximate the answer quality of online results. Still, an offline testbed that samples 20% of data would increase costs by $46,000 during the period studied.

5.6. Limitations

Timeliness of results and low overhead required our solution to occur in an online environment. Transparency is the requirement that motivates our approach as a networking solution. Although a new kernel module would also have sufficed for the other requirements, we wanted our solution to be usable out of the box for many services and systems. It is our hope that every service that communicates between components using a networking connection can find use in our design.

Our design approach is not fully automatic. System managers choose components to target. We leave identifying which components to target automatically for future work. In addition, record mode assumes that the targeted component responds to the

component that invoked it [Song et al. 2009]. In graph and event processing systems, the targeted component forwards data to the next node in the dataflow [Murray et al. 2013; Vlachos et al. 2010]. We also assume that request execution is read-only or otherwise idempotent, because workloads that allow writes may have incorrect output on replay. Finally, cached output can be used to replay only one component in the parallel execution path of each query. Specifically, our approach produces inaccurate results when two or more components that execute in the same sequential execution path time out, because replay can speed up only one. Section 4 suggests that picking components that interact with data partitions and are most affected by data skew suffices in many cases.

6. IMPLEMENTATION

This section discusses the implementation of Ubora. First, we describe axiomatic choices: the user interface, target users, and prerequisite infrastructure. Second, we discuss how operating system support for transparent context tracking impacted the implementation of memoization. Third, we provide details about our implementation and optimizations made to keep overhead low. Finally, we discuss our approach to determine which components constitute a front end.

6.1. Interface and Users

Ubora is designed for use by system managers. It runs on a cluster of compute nodes. Each node runs a networked operating system with many virtual machines. Each virtual machine runs one software component on its associated resources. To be clear, a software component is a running binary that accepts invocations over the network. Each software component has unique network addresses (e.g., IP address and TCP port) assigned through its virtual machine. A cluster of nodes may run one or more services. Each service comprises a set of software components logically arranged in query execution flow paths yet physically distributed across one or more nodes.

System managers understand the query execution paths in their service (e.g., as depicted in Figure 2). They classify each component as front- or back end. Front components receive queries, record intercomponent messages, and produce online and mature answers. They are re-executed to get mature answers. Back-end components propagate query context, record messages, and do not time out for sampled queries. Figure 2 labels the front-end component. The search tier, Redis and/or Lucene, could be front-end or back-end components.

Ubora is started from the command line. Two shell scripts, *startOnBack and startOnFront*, are run from a front component. Managers can configure several parameters before starting Ubora, shown in Listing 1. The query sampling rate is given in terms of the number of mature executions to initiate per unit time. When new queries arrive at front-end TCP ports, a query sampler randomly decides how to execute the query. Sampled queries are executed under the record mode context shown on the left side of Figure 4. Queries not sampled are executed normally without intervention from Ubora. Record timeout duration sets the upper bound on processing time for a back-end component's mature execution. Propagate timeout is used to set the upper bound on time to scan for newly contacted components to propagate the execution context. To get mature answers, the query execution context is called *replay mode*. Finally, the callback function used to compute answer quality is service specific. The default is the *true positive rate*.

6.2. Transparent Context Tracking

A key implementation goal was to make Ubora as transparent as possible. Here, transparent means that (1) it should work with existing middleware and operating systems

```
IPAddresses
-  front: 10.243.2.*:80
-  back: 10.244.2.*; 10.245.2.*:1064

samples: 8 per minute
recordTimeout: 15 seconds
propagateTimeout: 0.1 seconds
answerQualityFunction: default
```

Listing 1. Ubora's YAML Configuration.

without changing them and (2) it should have small effects on response times for online queries. Transparency is hard to achieve because Ubora must manage record and replay modes without changing the interaction between software components. In other words, the execution context of a query must pass between components unobtrusively. Some operating systems already support execution contexts. Therefore, we present two designs. The first design targets these operating systems. The second design targets commodity operating systems. Our designs exploit the following features of memoization:

1. *Queries produce valid output under record, replay, and normal modes*: This property is achieved by maintaining a shadow connection to the invoked component during replay. Cache misses trigger direct communication with invoked components. As a result, replay, normal, and record modes have access to full data.
2. *Back-end components use more resources during record mode than they use during normal online execution because timeouts are disabled.*
3. *Replay mode produces mature results within normal online timeout settings since the output of invoked components are replayed from the fast cache*: Our design schedules replay executions to avoid queuing delay.

Transparency using operating system–managed contexts. Some operating systems track execution context by annotating network messages and thread-local memory with context and ID. Dapper [Sigelman et al. 2010] instruments Google's threading libraries, Power Containers [Shen et al. 2012] tracks context switches between Linux processes and annotates TCP messages, and Xtrace [Fonseca et al. 2007] instruments networked middleware.

Operating system–managed execution context simplifies our design. Ubora intercepts messages between components, acting as a middle box. Before delivering messages that initiate remote procedures, Ubora checks query ID and context and configures memoization-related context (i.e., record or replay mode). The same checks are performed on context switches. During record mode, when a component initiates a remote invocation, we use the message and query ID as a key in the cache. Subsequent component interactions comprise the value associated with the key—provided that the query context and ID are matched. We split the value and form a new key when the invoking component sends another message. Subsequent messages from the target would provide values for the new key.

In replay mode, when an invocation message is intercepted, the message is used to look up values in the cache. On hits, the cache returns all values that are associated with the message. The cache results are turned into properly formatted messages (e.g., TCP packets) to transparently provide the illusion of a remote procedure call (RPC). On misses, the message is delivered to the destination component as described previously.

Transparency without operating system support. Most vanilla operating systems do not track execution context. Without such support, it is challenging to distinguish RPCs between concurrent queries. A universal feature of OLDI services is the use of distributed communication between software components. Although other context-tracking solutions are available, this network traffic is sufficient to allow transparency with regard to both workload and underlying operating system. Ubora's memoization permits imperfect context management because record, replay, and normal modes yield valid output. This feature allows us to execute concurrent queries under the same context, but we still must ensure correctness. First, we describe a simple but broken idea that is highly transparent, and then we present an empirical insight that allows us to improve this design without sacrificing transparency.

In this simple idea, each component manages its current global execution context that is applied to all concurrent queries. In addition, it manages a context ID that distinguishes concurrent record contexts. Ubora intercepts messages between components. When a component initiates a remote invocation in record mode, the message and context ID are used to create a key. For the duration of record mode, intercomponent messages are recorded as values for the key. If the context indicates replay mode, the message and context ID are used to retrieve values from the cache.

This simple idea is broken because all messages from the invoked component are recorded and cached, including concurrent messages from different queries. In replay mode, those messages can cause wrong output. Our key insight is that record mode should use replies from the invoked component only if they are from the same TCP connection as the initiating TCP connection. The approach works well as long as TCP connections are not shared by concurrent queries. Widely used paradigms like TCP connection pooling and thread pooling are ideal for use with Ubora. We studied the source code of 15 widely used open source components, including JBoss, LDAP, Terracotta, Thrift, and Apache Solr. Only two (13%) of these platforms multiplexed concurrent queries across the same connection. This suggests that our transparent design can be applied across a wide range of services. We confirm this in Section 7.4.

Next we describe how to propagate request context, which is necessary when the operating system does not support execution contexts. On a front component, the Ubora controller waits for queries to arrive on a designated TCP port. If a query is selected for mature execution, the Ubora controller changes the front component context from normal to record and create a context ID. Before sending any TCP message, we extract the destination component. If the destination has not been contacted since record mode was set, the Ubora controller sends a UDP message to tell the Ubora daemon running on that component to enter record mode and forwards the proper context ID. Then we send the original message. Note that UDP messages can fail or arrive out of order. This causes the mature execution to fail. However, we accept lower throughput (i.e., mature executions per query) when this happens to avoid increased latency from TCP round-trips. Middle components propagate state in the same way. Each component maintains its own local timers. After a propagation timeout is reached, the context ID is not forwarded any more. After the record timeout is reached, each component reverts back to normal mode independently. We require front components to wait slightly longer than record timeout to ensure that the system has returned to normal.

6.3. Prototype

We implemented transparent context tracking as described earlier for the Linux 3.1 operating system. The implementation is installed as a user-level package written in C and requires the Linux Netfilter library to intercept and reroute TCP messages. It uses IPQueue to trigger context management processes. It assumes that components

communicate through RPCs implemented on TCP and that an IP address and TCP port uniquely identify each component. It also assumes that timeouts are triggered by the RPC caller externally—not internally by the called component.

Ubora also implements a user-level controller that changes a node's operating mode to record, replay, or normal. A single-writer but globally readable file holds the current operating mode on each node. In normal mode, rules regarding targeted components and query detection are disabled.

Recording network payloads. Our approach records messages sent from targeted components during live execution. First, headers are matched against rules about (1) identifying targeted components, (2) new queries, and (3) Ubora control. Packets that do not match these rules are accepted (more precisely, they are not redirected). Second, we keep TCP connections to targeted components open after timeouts to obtain messages excluded from premature results. Ubora extends timeouts transparently by blocking FIN packets sent to the targeted component and spoofing ACKs from the caller. Messages from the targeted component that arrive after a blocked FIN are cached but not delivered to the caller. This continues until record times out or the targeted component sends a message to end the connection. Then, the entire recorded payload from the targeted component is stored as a value in Ubora's cache, with the query payload from the caller that invoked this output as its key. Whereas we delineate between key-value pairs by default using either FIN packets or the next inbound message to a component, service managers can specify alternate, application-specific data payloads to use for this purpose. This is especially important for workloads that use connection pooling with collated requests separated by special characters. We then separate messages using these application-specific termination payloads. Service managers can specify this in the configuration file. Packets are parsed in two stages.

Distributed cache. Naively limiting our storage usage to a single node increases average response time by increasing that storage node's network bandwidth and decreasing the amount of data used by the host application that can be stored. Instead, we allow system administrators to volunteer a list of nodes on which to store recorded message pairs. Recorded message pairs can be matched to nodes by a random hash, decreasing network bandwidth for bandwidth-intensive applications.

We use a distributed Redis cache for in-memory key-value storage. Redis allows us to set a maximum memory footprint per node. The aggregate memory across all nodes must exceed the footprint of a query. By using only a small percentage of the cache on each of allowed nodes, we minimize the overall cache miss rate overhead. Our default setting is an aggregate 1GB. In addition, Redis can run as a user-level process even if another Redis instance runs concurrently, providing high transparency.

We want to minimize the overhead in terms of response time and cache miss rate. Each key-value pair expires after a set amount of time. Assuming a set request rate, cache capacity will stabilize over time. A small amount of state is kept in local in-memory storage on the Ubora control unit node (a front node). Such state includes sampled queries, online and mature results, and answer quality computations.

Replay. When we rerun the cached query, we spoof the targeted components using recorded messages. Replayed queries bypass processing and I/O delays on targeted components, returning results at network speed. If a packet is headed for the targeted component, the packet is intercepted, and Ubora accesses its cache for the requested key-value pair associated with the data payload. The value is sent back to the source address if the key is found in the cache. If nothing is found, the data payload is sent to the targeted component for correctness. For correctness, when a key is not found in

the Ubora cache or the Ubora cache develops a fault, we also open a real connection to the targeted component and send the data payload.

Ubora's memoization approach meets the criteria outlined earlier. Other designs that we studied do not. For example, a system could measure answer quality by replaying queries on an offline testbed. This would violate our goal of using only online resources [Kelley et al. 2013]. A system could measure answer quality online by rewriting system components to support query-specific timeouts as in Bing [Jalaparti et al. 2013; He et al. 2012a]. However, this would require recoding software platforms. Finally, a system could simply change timeouts for each component dynamically. However, layered timeouts would force such changes to cover whole workflows (i.e., no incremental deployment). This would hurt response times for all live queries.

6.4. Optimizations for Low Overhead

Context tracking. Ubora reduces bandwidth required for context propagation. First, Ubora propagates context only to components used during online execution. Section 7.4 demonstrates that the increase in packets touched is more than compensated for in the decrease of Ubora network traffic during mode changes.

Second, Ubora does not use bandwidth to return components to normal mode, only sending UDP messages when necessary to enable record or replay mode. The naive context propagation sends messages for all context switches, including return to normal mode. Timeouts local to each component ensure that the system fully returns to normal mode, regardless of any lost UDP messages. Timeouts therefore increase robustness to network partitions and congestion. Front components time out after other components so that the entire system will have returned to normal mode before another mode change is issued.

Reducing replay overhead. To keep response times low, online services underutilize systems resources [Meisner et al. 2009, 2011]. Replay executions increase utilization on middle components. After record completes, Ubora queues the context ID on the front component for replay. The expected time to complete queued replays is the product of queue size and average online execution time. Naively, Ubora replays queries for mature execution as soon as possible after the record mode completes.

However, we have reduced replay overhead by using two further factors to decide when to replay queries. First, replay queries must execute within a short window after online queries finish to be useful to online management. This expiration window is set by system managers. If the time to clear the queue exceeds the remaining expiration window, Ubora initiates replays. Otherwise, replays are initiated if (1) there are no outstanding online queries and (2) the average interarrival time for online queries exceeds the time to replay. If these conditions are met, we replay the first query in the queue.

Second, by caching results without expiration, Ubora can run replay executions over a window of time after initial live executions. The value of delaying replay is that replay can be done when queuing delay is low, reducing the impact of replay on response times of other live executions. If services have frequent idle periods between queries, Ubora can schedule replays during such time. Meisner et al. [2009] found that such services idle about 70% but for less than 5ms at a time. Ubora can be set to read queue lengths at front-end nodes and schedule replays when queue length is below the 25th percentile.

Sampling. Mature results do not directly contribute to end-user satisfaction. Naively collecting mature results for each query would reduce an OLDI service's throughput by more than 50%. Ubora allows managers to specify stochastic sampling rates to determine when to compute a mature result.

We also use a node sampling optimization for applications with intensive data reuse. When this is used, recorded message pairs are stored on the node with the lowest Ubora storage footprint.

6.5. Determining Front-End Components

Thus far, we have described the front end as the software component at which queries initiate. Its internal timeout ensures fast response time, even as components that it invokes continue to execute in the background. To produce an online answer, the front end must complete its execution. Ubora re-executes the front end to get mature answers. Ubora cannot apply memoization to the front-end component.

At first glance, re-execution seems slower than memoization. However, as shown in Figure 3(a), many components execute quickly. In some cases, execution is faster than transferring intermediate data to the key-value store. Our implementation allows for a third class of component: middle components. Like front-end components, middle components are allowed to time out. They are re-executed in replay mode without memoization. Unlike front-end components, middle components do not initiate queries, but they can invoke targeted components and can be the target of memoization. In Figure 2, distributed search or Redis components could be labeled as middle components.

Given a trace of representative queries, Ubora determines which components to memoize by systematically measuring throughput with different combinations of front-, middle-, and back-end components. We do the same to determine the best sampling rate.

7. EXPERIMENTAL EVALUATION

In this section, we compare Ubora to alternative designs and implementations across a wide range of OLDI workloads. First, we discuss the chosen metrics of merit. Then, we describe the competing designs and implementations. Next, we present the software and hardware architecture for the OLDI services used. Finally, we present experimental results.

7.1. Metrics of Merit

Ubora speeds up mature query executions needed to compute answer quality. The research challenge is to complete mature query executions while processing other queries online at the same time. The primary metric used to evaluate Ubora's performance (*throughput*) is mature executions completed per 100 online executions.

Mature executions use resources otherwise reserved for online query executions, slowing down response times. Online queries that Ubora does not select for mature execution (i.e., unsampled queries) are slowed down by queuing delays. We report *slowdown* as the relative increase in response time. In addition to queuing delay, online queries sampled for mature execution are also slowed down by context tracking and memoization.

Finally, we used true positive rate (i.e., the percentage of mature results represented in online results) to compute answer quality.

7.2. Competing Designs and Implementations

Ubora achieves several axiomatic design goals. Specifically, it (1) speeds up mature executions via memoization, (2) uses a systems approach that works for a wide range of OLDI services, (3) supports adjustable query sampling rate, and (4) implements optimizations that reduce network bandwidth usage. Collectively, these goals make Ubora usable. Our evaluation compares competing designs that sacrifice one or more of these goals. They are listed in the following with an associated codename that will be used to reference them in the rest of the article:

Table I. OLDI Workloads Used to Evaluate Ubora Supported Diverse Data Sizes and Processing Demands

Code-name	Platform	Parallel Paths	Data (GB)	Nodes	Maturity	Utilization	QCoD
YN.bdb	Apache Yarn	2	1	8	96%	46%	8%
LC.news	Lucene	1	4	4	82%	73%	13%
LC.wik	Lucene	4	128	31	20%	23%	53%
LC.big	Lucene	4	4,096	31	10%	40%	55%
ER.fst	EasyRec	2	2	7	75%	15%	89%
OE.jep	OpenEphyra	4	4	8	5%	20%	56%

- *Ubora* implements our full design and implementation. The sampling rate is set to maximize mature query executions per online query.
- *Ubora-LowSamples* implements our full design and implementation but lowers the sampling rate to reduce slowdown.
- *Ubora-NoOpt* disables Ubora's optimizations. Specifically, this implementation disables node-local timeouts that reduce network bandwidth usage.
- *Query tagging and caching* essentially implements Ubora at the application level. Here, we implement context tracking by changing the OLDI service's source code so that each query accepts a timeout parameter. Further, we set up a query cache to reuse computation from online execution. This approach is efficient but requires invasive changes.
- *Query tagging* implements context tracking at the application level but disables memoization.
- *Timeout toggling* eschews both context tracking and memoization. This implementation increases each component's global timeout settings by 4X for mature executions. All concurrent query executions also have increased timeout settings. This is noninvasive because most OLDI components support configurable timeout policies. However, extending timeouts for all queries is costly.

7.3. OLDI Services

Table I describes each OLDI service used in our evaluation. In the rest of this article, we refer to these services using their codename. The setup shown in Figure 2 depicts LC.big, a 31-node cluster that supports 16GB DRAM cache per terabyte stored on disk. Each component runs on a dedicated node comparable to an EC2 medium instance, providing access to an Intel Xeon E5-2670 VCPU, 4GB DRAM, 30GB local storage, and (up to) 2TB block storage:

- *YN.bdb* uses Hadoop/Yarn for sentiment analysis. Specifically, it runs query 18 in BigBench, a data analytics benchmark [Ghazal et al. 2013]. Each query spawns two parallel executions. The first subexecution extracts sentiments from customer reviews over 2 months. The second covers 9 months. The 9-month execution returns the correct answer, but the 1-month answer is used after a 3-minute timeout. Each subexecution flushes prior cached data in HDFS, restores a directory structure, and compresses its output. As a result, query execution takes minutes, even though customer reviews are smaller than 1GB. The average response time without timeout is 3 minutes; 44% of queries get the 9-month answer within timeout. We mainly include YN.bdb to show that Ubora can capture answer quality for longer-running services as well.
- *LC.big*, *LC.wik*, and *LC.news* use Apache Lucene for bag-of-words search. All of these workloads replay popular query traces taken from Google Trends.
  LC.news hosts 4GB of news articles and books on a Redis cluster with 4GB DRAM. LC.news implements one of the four parallel paths shown in Figure 2. It returns the

best answer produced within 1 second. Without timeouts, the average response time is 1.22 seconds. More than 83% of LC.news queries complete within the timeout.

LC.wik hosts 128GB of data from Wikipedia, New York Times, and TREC NLP [Technology Laboratory's (ITL) Retrieval Group 2014]. After executing warm-up queries, the data mostly fits in memory. We set timeout at 3 seconds. Without the timeout, response time was 8.9 seconds; 39% of the LC.wik queries complete within the timeout.

LC.big hosts 4TB. Most queries access disk. Average response time without timeout is 23 seconds. The timeout is 5 seconds.

- *ER.fst* uses the EasyRec platform to recommend Netflix movies. It compiles two recommendation databases from Netflix movie ratings [Netflix 2009]: a 256MB version and a 2GB version. Each query provides a set of movie IDs that seed the recommendation engine. The engine with more ratings normally takes longer to respond but provides better recommendations. Query execution times out after 500ms; 80% of query executions produce the 2GB answer.
- *OE.jep* uses OpenEphyra, a question-answering framework [Schlaefer 2013]. OpenEphyra uses bag-of-words search to extract sentences in the AQUAINT-2 NLP dataset related to queries from the TREC trace [Technology Laboratory's (ITL) Retrieval Group 2014]. It then compares each sentence to a large catalog of noun-verb templates, looking for specific answers. The workload is computationally intensive. The average response time in our setup was 23 seconds. Motivated by the responses times for IBM Watson, we set a timeout of 3 seconds [Ferrucci 2010]; fewer than 15% of queries completed within timeout.

We set up a workload generator that replayed trace workloads at a set arrival rate for each workload. The goal was to keep a concurrency level of 1 at the top-level node. Based on averaging the CPU utilization for all of the nodes used in a workload, our workload generator kept CPU utilization between 15% and 35% overall.

Table I also displays numerical characteristics that illustrate the diversity of our tested workloads, including utilization, the quartile coefficient of distribution, and maturity. The utilization shown for each workload is defined as $\frac{Arrival\,Rate}{Processing\,Rate}$. The arrival rate and processing rate used in this calculation were measured for each workload without turning on Ubora. As utilization increases, Ubora is challenged to achieve memoization and replay without creating too much queuing delay. Table I also shows the quartile coefficient of distribution for the response times of target components in each workload. Finally, we define *maturity* as the ratio between average online query execution time (affected by premature timeouts) and mature execution time. Greater maturity allows less time for mature executions to differ from online executions. These values are computed offline and are used here to characterize the workload. Our services have diverse CPU% and IO% (not shown). This stems from the wide range of data and cluster sizes covered. Taken together, our services represent many OLDI workloads.

7.4. Results

Microbenchmark tests. Our first test studied the effect of data skew and component selection. For this, we used a microbenchmark consisting of three software components, a front component that accepts queries, and two auxiliary components. Each query randomly selected one auxiliary component to have a running time X% longer than the other. Here, X% approximates data skew. The output of each auxiliary component is its running time, and the microbenchmark's output is the largest observed running time.

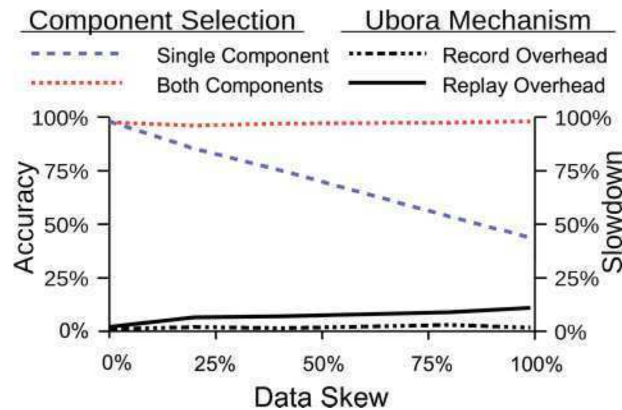

Fig. 6. Microbenchmark study on the effects of component selection on accuracy and Ubora mechanisms on overhead under changing data skew. Data skew represents the difference in running times between two auxiliary components.

The front component times out after the shortest component completes (100ms). We issued 10,000 queries to this microbenchmark one after another (e.g., 10 queries/second).

The left axis of Figure 6 shows the accuracy of mature results in this test—that is, the relative error between the time given by our mature results and the running time of the slowest component. We report average error. The top dotted line, labeled *Both Components*, shows results when both auxiliary components are targets. The dashed line shows results when only one auxiliary component is a target. When both components are targets, accuracy ranges between 96% and 99%. However, the *Single Component* line warns about the perils of selecting targets poorly. Consider the extreme case where the shortest component runs for 100ms and the longest runs for 200ms. If the wrong component (in this case, the shorter-running component) is selected, the best possible accuracy is 50%. Record and replay overheads cause further degradation. On the right axis of Figure 6, we report slowdown (i.e., increase in response time) caused by record mode and replay mode, respectively, in the *Both Components* experiment. Record mode includes the cost of redirecting network messages to cache. Its overhead is around 1%. Replay mode includes the cost of extending timeout for the long-running component and the cost of queuing delays to replay executions. The slowdown grew by 1.8% per 10% increase in data skew. Effectively, this means that we reduced the amount of time needed to perform a mature execution by 5.5X. These tests show that our record and replay mechanism are implemented efficiently.

Comparison to competing approaches. Figure 7 compares competing approaches in terms of mature executions completed per 100 online queries. For these experiments, we set the sampling rate to record approximately 40 queries out of every 100. Ubora-NoOpt reveals that node-local timeouts and just-in-time query propagation collectively reduce the overhead of sampling, improving the throughput of mature execution completions by 1.6X, 1.3X. and 2.1X, respectively. ER.fst has relatively fast response times that require messages to turn off record and replay modes. Node-local timeouts reduce these costs. Internal component communications in LC.big and LC.wik also benefit from node-local timeouts.

Excluding Ubora, the other competing approach that can be implemented for a wide range of services is toggling timeouts. Unfortunately, this approach performs poorly, lowering throughput by 7 to 8 X. To explain this result, we use a concrete example of

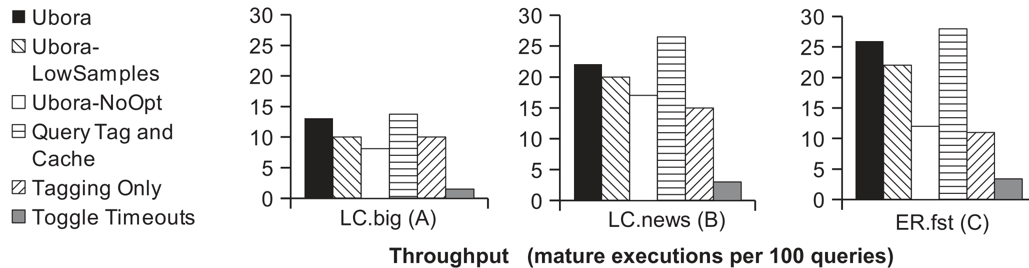

Fig. 7. Experimental results. Ubora achieves greater throughput than competing systems-level approaches. It performs nearly as well as invasive application-level approaches (within 16%).

a search for "Mandy Moore" in LC.big. First, we confirm that both Ubora and toggling timeouts produce the same results. They produce the same top 5 results, and 90% of the top 20 results overlap. Under 5-second timeout, the query times out prematurely, outputting only 60% of top 20 results. Ubora completes mature executions faster because it maintains execution context. This allows concurrent queries to use different timeout settings. Queries operating under normal timeouts free resources for the mature execution. Further, per-component processing times vary within mature executions (recall Figure 3(a)). By maintaining execution context, Ubora avoids overusing system resources. For the "Mandy Moore" query, Ubora's mature execution took 21 seconds in record mode and 4 seconds in replay mode. Conversely, under the toggle timeouts approach, service times for all concurrent queries increased by 4X, exceeding system capacity and taking 589 seconds.

We also compared our systems-level implementation of Ubora, which strives to transparently support a wide range of services, to application-level approaches. For these experiments, we maximize throughput for Ubora on ER.fst. Based on this curve, in Figure 9(a) (shown later), we set the sampling rate to 20% for all approaches. Application-level approaches can track query context efficiently by tagging queries as they traverse the service [Fonseca et al. 2007]. Specifically, we modified LC.big, LC.news, and ER.fst to pass query context on each component interaction. Further, we implemented a query cache for targeted query interactions [Amza et al. 2005; Guo et al. 2013; Paiva et al. 2013]. Our cache uses Ubora's mechanism for memoization but tailors it to specific intercomponent interactions and context IDs. As such, our application-level approach is expected to outperform Ubora, and it does. However, Ubora is competitive, achieving performance within 16% on all applications. We also compared to a simple application-level approach that disables query caching. This approach shows that memoization improves throughput by 1.3X on LC.big, 1.7X on LC.news, and 2.5X on ER.fst. The benefit provided by memoization is correlated with the ratio of mature execution times to online execution times. In ER.fst, mature executions are mostly repeating online executions.

Impact on response time. Ubora allows system managers to control the query sampling rate. Figure 8(a) compares the throughput rate (mature executions per 100 queries) for Ubora with and without optimizations for different sampling rates. Our optimizations improve throughput for LC.big by 2X at the 40% sampling rate.

In contrast, Ubora-LowSamples only replays queries when the interarrival time is high. This slight reduction in the sampling rate can still achieve high throughput. However, this approach significantly reduces Ubora's effect on response time. Figure 8(b) shows the slowdown caused by the Ubora-LowSamples approach across all tested workloads. By executing mature executions in the background and staying within processing capacity, we achieve slowdown below 13% on all workloads for unsampled queries and

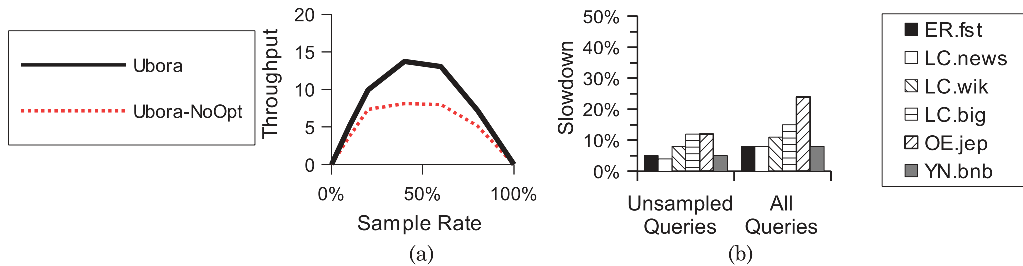

Fig. 8. Impact on response time. (a) Throughput under varying sampling rate for Ubora and Ubora-NoOpt. (b) Ubora delayed unsampled queries by 7% on average. Sampled queries were slowed by 10% on average.

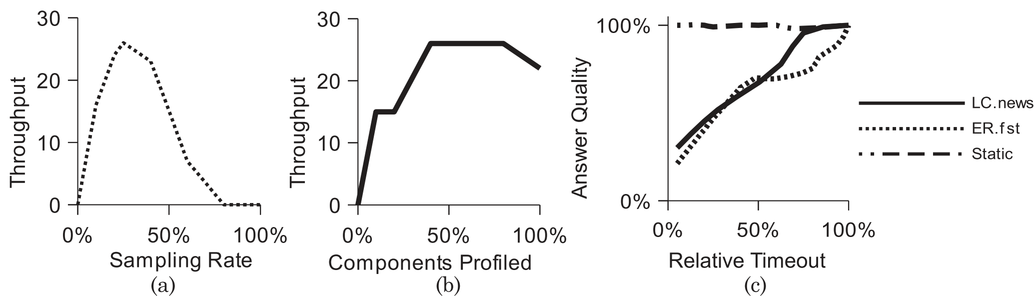

Fig. 9. Experimental results for maximized throughput with ER.fst. (a) We profiled sampling options. (b) We profiled memoization options. (c) Timeout settings have complex, application-specific affects on answer quality.

below 10% on four of six workloads for sampled queries. OpenEphyra and LC.big incur the largest overhead because just-in-time context interposes on many intercomponent interactions due to cluster size. For such workloads, operating system–level context tracking would improve response time for sampled queries.

Collectively, the five workloads shown use nine platforms, including the widely used Apache Lucene, EasyRec Recommendation Engine, OpenEphyra, and NanoWeb PHP server. In general, the variance of mature execution times (i.e., QCoD) correlates positively to the throughput achieved by each workload. The target components in EasyRec workloads in particular have the greatest QCoD. EasyRec workloads yield throughput about 50% relative to other workloads. Higher utilization levels were associated with greater slowdown on unsampled queries, reflecting queuing delay. We also observed less slowdown on ER.fst. This workload had higher maturity and low utilization, which limits the potential for slowdown.

Impact of profiling. Figure 9(a) and (b) study our approach to determine sampling rate and front-end components (i.e., memoization). We studied the ER.fst workload. Figure 9(a) shows the achieved throughput (mature executions per 100 queries) as the percentage of mature executions initiated increases. Figure 9(b) shows the achieved throughput as the percentage of components included as the front end of middle components increases. For the ER.fst workload, it is better to apply memoization to many components. The 20% sampling rate for Ubora-LowSamples on ER.fst maximized throughput.

The peak sampling rate corresponds to 12 queries per minute. Because the requests for LC.news took longer, the peak sampling rate for Figure 8(a) corresponded to 1 query per minute. First, we observe that under Ubora-LowSamples, the failure rate increases with the sampling rate. This is due to expired cache entries and the potential for

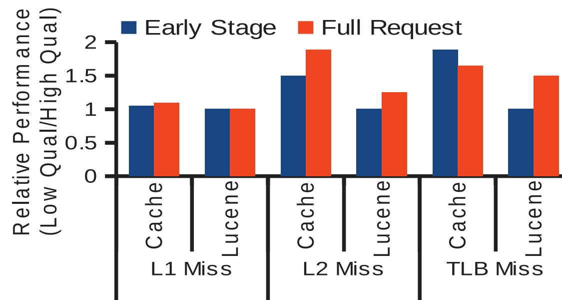

Fig. 10. Some hardware counters predict answer quality.

additional time between memoization and replay. Under a 20% sampling rate, 17% of mature execution fail to yield mature results. This rises to 84% at an 80% sampling rate. Figure 8(a) agrees with this rise, with 30% of mature executions failing to yield mature results at 60% of the sampling rate. Peak throughput is achieved at the cost of efficiency. We also observe that Ubora's optimizations collectively lead to significant throughput gains across sampling rates.

Studying answer quality: Figure 9(c) shows answer quality (i.e., the true positive rate) as we increase timeout settings. For LC.news and ER.fst, we increase timeouts at front-end components. We also validate our results by increasing timeouts in an unrelated component in ER.fst (static). We observe that answer quality is stable in the static setting. Further, answer quality curves differ between applications. After timeout settings reach 600ms for LC.news and 300ms for ER.fst, the curves diverge and answer quality increases slowly for ER.fst. Finally, answer quality curves have two phases in LC.news and three phases in ER.fst. It is challenging to use timeouts to predict answer quality.

Using hardware counters to improve sampling. Online executions that complete without triggering timeouts make mature executions unnecessary. Ubora may further reduce its overhead by turning off memoization and replay when it predicts that online executions will complete fully. Prior work has shown that hardware counters are useful predictors of query outcomes. We studied whether hardware counters collected early in a query execution can be used to predict answer quality in LC.big. For this test, we used the Google trace and issued queries one at a time under a tighter timeout (3 seconds). We collected level-1 (L1) cache misses, level-2 (L2) cache misses, and translation look-aside buffer (TLB) misses every second. Figure 10 shows hardware counters after the first third of query execution and across the whole query execution. The figure shows the results of low-quality queries relative to the results for high-quality queries. We observed that the ratio of L2 misses and TLB misses on cache nodes were markedly higher in cache nodes. These predictors detect high-quality queries quickly enough to prevent mature executions (if the query had been sampled). In LC.big, this approach has the capacity to reduce mature executions by a further 60%, doubling Ubora's throughput.

For our Apache Lucene search engine, this makes sense as TLB misses often mean that a slow Lucene disk will have more lookups and that the query will likely time out.

8. ONLINE MANAGEMENT

OLDI services use anytime algorithms, returning valid results even when provisioned to provide target response times. In addition to classic metrics like response time, these services could use answer quality to manage resources. We show here that Ubora

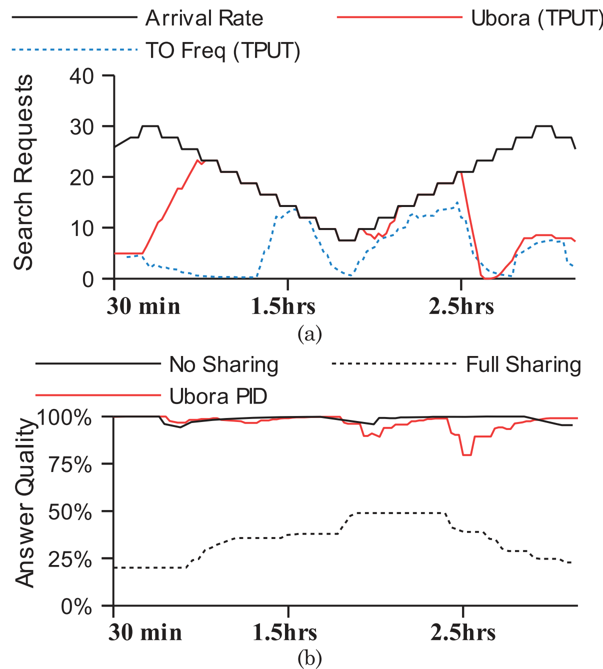

Fig. 11. Ubora enables online admission control. Arrival rate refers only to low-priority requests. High-priority requests arrive at a fixed rate.

enables better resource management through answer quality. In this case study, we use Ubora to improve admission control, a classic system management challenge.

Control theory with answer quality. We studied admission control on the LC.big workload. We issued two classes of queries that arrived at different TCP ports, indicating high and low priority. High-priority requests arrived at a fixed rate in terms of requests per second. We used diurnal traces from previous studies [Stewart et al. 2007, 2013] to issue low-priority requests. At the peak workload, low- and high-priority arrival rates saturate system resources (i.e., utilization is 90%). Figure 11(a) shows the *Arrival Rate* of low-priority queries over time as well as the number of low-priority search requests admitted. We used Ubora to track answer quality for high-priority queries. Here, answer quality is the true positive rate for the top 20 results. At the 45-minute and 2-hour marks, the query mix shifts toward multiple word queries that take longer to process fully. This accounts for the drops in answer quality for the *No Sharing* line in Figure 11(b). When quality dipped, we decreased the admission control rate on low-priority queries. Specifically, we used a proportional-integral-derivative (PID) controller. Every 100 requests, we computed answer quality from 20 sampled queries (20% sample rate). The PID controller used 10-minute sliding windows to average out spikes in answer quality and timeout frequency. The PID controller weighted current reading at 40% (i.e., proportional portion).

The y-axis of Figure 11(b) shows answer quality of competing admission control approaches. When no low-priority queries are admitted, the *No Sharing* approach maintains answer quality above 90% throughout the trace, even during periods with complex queries. When admission control is disabled, the *Full Sharing* approach sees answer quality as low as 20%, corresponding with peak arrival rates. The PID controller powered by Ubora manages the admission rate well, keeping answer quality above

Table II. Adaptive Management Degrades Under Low Sampling Rates

Sampling Rate	Measurement Error for Answer Quality		Rate of Quality Violations
	Average	95th Percentile	
10%	0%	0%	4%
5%	20%	45%	9%
3%	30%	50%	13%
2%	51%	78%	29%

Note: A *quality violation* is a window where answer quality falls below 90%. Error is relative to the 10% rate.

90% in more than 90% of the trace. There is about a 20% drop in answer quality for the UBORA PIDcontroller approximately at the point in time where the query mix increases in complexity. The drop in UBORA TPUT occurs concurrently with this, indicating that the amount of low-priority queries shed to counter this drop increased. It allows almost 60% of low-priority queries to complete (*Ubora (TPUT)*).

The state of the art for online management in OLDI services is to use proxies for the answer quality metric. Metrics like the frequency of timeouts provide a rough indication of answer quality and are easier to compute online [Jalaparti et al. 2013]. For comparison, we implemented a PID controller that used frequency of timeouts instead of answer quality. We tuned the controller to achieve answer quality similar to the controller based on answer quality. However, timeout frequency is a conservative indicator of answer quality for Lucene workloads. It assumes that partial results caused by timeouts are dissimilar to mature results. Figure 11(a) also shows that the controller based on timeout frequency (*TO Freq (TPUT)*) drops requests too aggressively. Queries can only be dropped explicitly in our system, so both TO Frequency and Ubora PID controllers achieve full throughput on high-priority requests. For most of the trace, the Ubora PID controller has a higher throughput on low-priority requests than *TO Freq*. When arrival rate increases, both Ubora and TO Frequency controllers admit fewer low-priority queries. The *TO Freq* PID controller is consistently more conservative than *Ubora PID*. The TO Frequency PID controller only allowed 25% of low-priority queries to complete. Compared to the TO Frequency PID controller's peak throughput over the whole trace, our Ubora PID controller improved peak throughput on low-priority queries by 55%.

Sampling rate and representativeness. Ubora allows reducing the overhead of mature executions by sampling online executions. This lowers mature results per query, but how many mature results are needed for online management? Table II shows the effect of lower sampling rates on the accuracy of answer quality measurements and on the outcome of adaptive admission control. We observed that sampling 5% of online queries significantly increased outlier errors on answer quality, but our adaptive admission control remained effective—it still achieved greater than 90% quality over 90% of the trace. In contrast, a 2% sampling rate produced many quality violations.

9. CONCLUSION

OLDI queries have complex and data-parallel execution paths that must produce results quickly. Data used by each query is skewed across data partitions, causing some queries to time out and return premature results. Answer quality is a metric that assesses the impact of timeouts on the quality of results. It is challenging to compute online because it require results from mature executions that are unaffected by timeouts.

This article describes Ubora, a design approach to speed up mature executions by reusing intermediate computations from online queries (i.e., memoization). Ubora adopts a challenging systems-level approach that allows us to measure answer quality

for a wide range of services. Our implementation includes novel context tracking for commodity operating systems and bandwidth optimizations. The evaluation shows that Ubora produces mature results faster than competing transparent approaches and nearly as fast as a less flexible, application-specific approach.

We evaluated Ubora on Apache Lucene with Wikipedia data, OpenEphyra with New York Times data, EasyRec recommendation engine with Netflix data, and Hadoop/Yarn with BigBench data [The Apache Software Foundation 2015, 2016; Wikimedia Foundation 2014; Schlaefer 2013; Technology Laboratory's (ITL) Retrieval Group 2014; Research Studios Austria Forschungsgesellschaft mbH 2014; Netflix 2009; Intel Corporation 2016]. Ubora slows down normal query executions by less than 7% on average. Ubora completes mature executions almost as quickly as query tagging, which eschews transparency for efficiency, with slowdown ranging from 8% to 16%. We also compared Ubora to timeout toggling, an alternative approach that does not require changing application source code if the allowed processing time is a configuration setting for the application. However, under this approach, all currently executing queries operate under the same context. Ubora exhibited a 7X speedup in finishing mature executions over timeout toggling.

Most importantly, Ubora produces answer quality quickly enough to enhance online system management. We used Ubora to guide online management, increasing throughput compared to offline approaches. We adaptively shed low-priority queries to our Apache Lucene and EasyRec systems. The goal was to maintain high answer quality for high-priority queries. Ubora provided answer quality measurements quickly enough to detect shifts in the arrival rate and query mix. The other transparent approach to measure answer quality (i.e., toggling timeouts) produced mature executions too slowly. This approach allowed answer quality to fall below 90% 12X much more often than Ubora. We also used component timeouts as a proxy for answer quality [Jalaparti et al. 2013]. This metric is available after online executions without conducting additional mature executions. As a result, it has much lower overhead. However, component timeouts are a conservative approximation of answer quality because they do not assess the effect of timeouts on answers. While achieving the same answer quality on high-priority queries, Ubora-driven admission control improved peak throughput on low-priority queries by 55% compared to admission control powered by component timeouts.

We also studied the predictive power of hardware counters to answer quality on Redis, a key-value store we used with the Lucene workload [Redislabs 2016; The Apache Software Foundation 2016]. Predictive hardware counters enable preemptive actions, such as extending timeouts before they are triggered. We counted L1 cache misses, L2 cache misses, and TLB misses during periods with high (>90%) and low answer quality. After executing 10% of a query, L2 misses for Redis were good predictors of low-quality answers. However, their predictive power varied across components.

We believe that the transparent design of Ubora can be of use to future frameworks aiming to share context among a cluster of machines. Custom, hand-coded approaches could possibly achieve similar gains, but Ubora can help a wide range of multicomponent services, including outreach efforts, as in Muhammad et al. [2016]. In the future, we intend to use Ubora to dynamically tune cache size in the OpenEphyra question-answering system to support science, technology, engineering, and mathematics outreach. We are developing a unit to teach big data and natural language processing using Ubora to facilitate a classroom game where students compete against an online question-answering service. By dynamically allocating or reducing cache size to match its competitors knowledge base, it is our hope that the OpenEphyra question-answering system will be able to adequately compete with people of multiple age ranges across a broad range of knowledge categories. Our conclusion is that Ubora democratizes

answer quality, allowing many services to provide high-quality results and fast response times.

10. CODE AVAILABILITY

The source code for Ubora is available at https://github.com/JaimieKelley/ubora/. We will also share file system images of our OLDI workloads.

REFERENCES

Mumtaz Ahmad, Ashraf Aboulnaga, Shivnath Babu, and Kamesh Munagala. 2008. Modeling and exploiting query interactions in database systems. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management*.

M. T. A. Amin, S. Li, M. R. Rahman, P. T. Seetharamu, S. Wang, T. Abdelzaher, I. Gupta, M. Srivatsa, R. Ganti, R. Ahmed, and H. Le. 2015. SocialTrove: A self-summarizing storage service for social sensing. In *Proceedings of the 2015 International Conference on Autonomic Computing*.

Cristiana Amza, Gokul Soundararajan, and Emmanuel Cecchet. 2005. Transparent caching with strong consistency in dynamic content Web sites. In *Proceedings of the 19th Annual International Conference on Supercomputing*. ACM, New York, NY, 264–273.

Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. 307–320.

Oleksandr Barykin, Bhuwan Chopra, Ciprian Gerea, Josh Metzler, Subbu Subramanian, Janet Wiener, David Reiss, and Daniel Merl. 2013. Scuba: Diving into data at Facebook. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'13)*.

R. Falsett, R. Seyer, and C. Siemers. 2004. Limitation of the response time of a software process. Retrieved March 29, 2017, from http://www.google.com/patents/WO2003069424A3?cl=en

D. Ferrucci, E. Brown, J. Chu-Carroll, J. Fan, D. Gondek, A. Kalyanpur, A. Lally, et al. 2010. The AI behind Watson-the technical article. AI Magazine. Retrieved March 29, 2017, from http://www.aaai.org/Magazine/Watson/watson.php.

Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. 2007. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation (NSDI'07)*. 20.

B. Forrest. 2009. Bing and Google Agree: Slow Pages Lose Users. Retrieved March 29, 2017, from http://radar.oreilly.com/2009/06/bing-and-google-agree-slow-pag.html.

Armando Fox, Steven D. Gribble, Yatin Chawathe, and Eric A. Brewer. 1998. Adapting to network and client variation using infrastructural process proxies: Lessons and perspectives. *Personal Communications* 5, 10–19.

Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, and Li Zhang. 2014. Adaptive, model-driven autoscaling for cloud applications. In *Proceedings of the 2014 International Conference on Autonomic Computing*.

A. Gelfond. 2011. TripAdvisor Architecture—40M Visitors, 200M Dynamic Page Views, 30TB Data. Retrieved March 29, 2017, from http://highscalability.com/blog/2011/6/27/tripadvisor-architecture-40m-visitors-200m-dynamic-page-view.html.

A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H. Jacobsen. 2013. BigBench: Towards an industry standard benchmark for big data analytics. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. 1197–1208.

I. Goiri, R. Bianchini, S. Nagarakatte, and T. Nguyen. 2015. ApproxHadoop: Bringing approximations to MapReduce frameworks. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. 383–397.

Y. Guo, P. Lama, J. Rao, and X. Zhou. 2013. V-Cache: Towards flexible resource provisioning for multi-tier applications in iaas clouds. In *Proceedings of the International Symposium on Parallel and Distributed Processing*.

Yuxiong He, Sameh Elnikety, James Larus, and Chenyu Yan. 2012a. Zeta: Scheduling interactive services with partial execution. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC'12)*. Article No. 12.

Yuxiong He, Sameh Elnikety, and Hongyang Sun. 2011. Tians scheduling: Using partial processing in best-effort applications. In *Proceedings of the 2011 31st International Conference on Distributed Computing Systems (ICDCS'11)*.

Yuxiong He, Zihao Ye, Qiang Fu, and Sameh Elnikety. 2012b. Budget-based control for interactive services with adaptive execution. In *Proceedings of the 2012 International Conference on Autonomic Computing*.

Intel Corporation. 2016. GitHub: intel-hadoop/Big-Data-Benchmark-for-Big-Bench: Big Bench Workload Development. Retrieved March 29, 2017, from https://github.com/intel-hadoop/Big-Data-Benchmark-for-Big-Bench.

V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. 2013. Speeding up distributed request-response workflows. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM'13)*. 219–230.

M. Jeon, Y. He, S. Elnikety, A. Cox, and S. Rixner. 2013. Adaptive parallelization of Web search. In *Proceedings of the 2013 EuroSys Conference.*

Niranjan Kamat, Prasanth Jayachandran, Kathik Tunga, and Arnab Nandi. 2014. Distributed interactive cube exploration. In *Proceedings of the 30th IEEE International Conference on Data Engineering (ICDE'14)*.

J. Kelley, C. Stewart, S. Elnikety, and Y. He. 2013. Cache provisioning for interactive NLP services. In *Proceedings of the Workshop on Large-Scale Distributed Systems and Middleware*.

Jaimie Kelley, Christopher Stewart, Nathaniel Morris, Devesh Tiwari, Yuxiong He, and Sameh Elnikety. 2015. Measuring and managing answer quality for online data-intensive services. In *Proceedings of the 2015 International Conference on Autonomic Computing*.

J. Kephart and J. Lenchner. 2015. A symbiotic cognitive computing perspective on autonomic computing. In *Proceedings of the 2015 International Conference on Autonomic Computing*.

YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. 2012. SkewTune: Mitigating skew in MapReduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD'12)*. 25–36.

Palden Lama and Xiaobo Zhou. 2012. AROMA: Automated resource allocation and configuration of MapReduce environment in the cloud. In *Proceedings of the 9th International Conference on Autonomic Computing.*

George Lawton. 2005. LAMP lights enterprise development efforts. *Computer* 9, 18–20.

Jon Lenchner. 2011. Knowing What It Knows: Selected Nuances of Watson's Strategy. Retrieved March 29, 2017, from https://www.ibm.com/blogs/research/2011/02/knowing-what-it-knows-selected-nuances-of-watsons-strategy/.

Lucid Imagination. 2010. *The Case for Lucene/Solr: Real World Search Applications*. White Paper. Lucid Imagination.

Christopher Manning, Prabhakar Raghavan, and Hinrich Schtze. 2008. *Introduction to Information Retrieval*. Cambridge University Press, Cambridge, MA.

D. Meisner, B. Gold, and T. Wenisch. 2009. PowerNap: Eliminating server idle power. In *ACM ASPLOS*.

D. Meisner, C. Sadler, L. Barroso, W.-D. Weber, and T. F. Wenisch. 2011. Power management of on-line data intensive services. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. 205–216.

Stephanie Muhammad, Jaimie Kelley, and Christopher Stewart. 2016. Ed Watson: Teaching big data to K-12 students. In *Proceedings of the 2016 Spring Undergraduate Research Expo.*

Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. 2013. Naiad: A timely dataflow system. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. ACM, New York, NY, 439–455.

Netflix. 2009. Netflix Prize. Retrieved March 29, 2017, from http://www.netflixprize.com/

Joao Paiva, Pedro Ruivo, Paolo Romano, and Luís Rodrigues. 2013. AUTOPLACER: Scalable self-tuning data placement in distributed key-value stores.. In *Proceedings of the 2013 International Conference on Autonomic Computing*.

Redislabs. 2016. Redis. Retrieved March 29, 2017, from http://redis.io/

Shaolei Ren, Yuxiong He, Sameh Elnikety, and Kathryn McKinley. 2013. Exploiting processor heterogeneity in interactive services. In *Proceedings of the 2013 International Conference on Autonomic Computing*.

Research Studios Austria Forschungsgesellschaft mbH. 2014. Easyrec: Open Source Recommendation Engine. Retrieved March 29, 2017, from http://easyrec.org/.

Nico Schlaefer. 2013. The Ephyra Question Answering System. Retrieved March 29, 2017, from https://sourceforge.net/projects/openephyra/.

Kai Shen, Arrvindh Shriraman, Sandhya Dwarkadas, Xiao Zhang, and Zhuan Chen. 2012. Power containers: An OS facility for fine-grained power and energy management on multicore servers. In *AProceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*.

B. Sigelman, L. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Google Technical Report. Retrieved March 29, 2017, from https://research.google.com/pubs/pub36356.html.

Yee Jiun Song, Marcos K. Aguilera, Ramakrishna Kotla, and Dahlia Malkhi. 2009. RPC chains: Efficient client-server communication in geodistributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*. 277–290.

Simon Spinner, Giuliano Casale, Xiaoyun Zhu, and Samuel Kounev. 2014. LibReDE: A library for resource demand estimation. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*.

Christopher Stewart, Aniket Chakrabarti, and Rean Griffith. 2013. Zoolander: Efficiently meeting very strict, low-latency SLOs. In *Proceedings of the 2013 International Conference on Autonomic Computing*.

C. Stewart, T. Kelly, and A. Zhang. 2007. Exploiting nonstationarity for performance prediction. In *Proceedings of the 2007 EuroSys Conference.*

Technology Laboratory's (ITL) Retrieval Group. 2014. Text Retrieval Conference Data. Retrieved March 29, 2017, from http://trec.nist.gov/data.html.

The Apache Software Foundation. 2015. Apache Hadoop 2.7.1—Apache Hadoop NextGen MapReduce (YARN). Retrieved March 29, 2017, from https://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/YARN.html.

The Apache Software Foundation. 2016. Apache Lucene Core. Retrieved March 29, 2017, from http://lucene.apache.org/core/.

Evangelos Vlachos, Michelle L. Goodstein, Michael A. Kozuch, Shimin Chen, Babak Falsafi, Phillip B. Gibbons, and Todd C. Mowry. 2010. ParaLog: Enabling and accelerating online parallel monitoring of multithreaded applications. *ACM SIGARCH Computer Architecture News* 38, 1, 271–284.

Wikimedia Foundation. 2014. Wikimedia Downloads. Retrieved March 29, 2017, from https://dumps.wikimedia.org.

Wikipedia 2014. Wikipedia:Modelling Wikipedia's Growth. Retrieved March 29, 2017, from https://en.wikipedia.org/wiki/Wikipedia:Modelling_Wikipedia's_growth.

Y. Zheng, B. Ji, N. Shroff, and P. Sinha. 2015. Forget the deadline: Scheduling interactive applications in data centers. In *Proceedings of the 2015 IEEE 8th International Conference on Cloud Computing (CLOUD'15)*.

Shlomo Zilberstein. 1996. Using anytime algorithms in intelligent systems. *AI Magazine* 17, 73–83.

Received April 2016; revised January 2017; accepted February 2017