Ubora: Measuring and Managing Answer Quality for Online Data-Intensive Services

Jaimie Kelley & Christopher Stewart
The Ohio State University

Devesh Tiwari
Oak Ridge National Labs

Yuxiong He & Sameh Elnikety
Microsoft Research

***Abstract*— Online data-intensive services parallelize query execution across distributed software components. Interactive response time is a priority, so online query executions return answers without waiting for slow running components to finish. However, data from these slow components could lead to better answers. We propose Ubora, an approach to measure the effect of slow running components on the quality of answers. Ubora randomly samples online queries and executes them twice: the first execution elides data from slow components and provides fast online answers, the second execution waits for all components to complete. 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 platforms, including Hadoop/Yarn, Apache Lucene, the EasyRec Recommendation Engine, and the OpenEphyra question answering system. Ubora computes answer quality much faster than competing approaches that do not use memoization. With Ubora, we show that answer quality can and should be used to guide online admission control. Our adaptive controller processed 37% more queries than a competing controller guided by the rate of timeouts instead of answer quality.**

I. Introduction

Online data-intensive (OLDI) services, such as search engines, product recommendation, sentiment analysis and Deep QA power many popular websites 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 [12]. Similarly, IBM Watson would have lost to elite Jeopardy contestants if it waited too long to answer [10], [17]. Unlike traditional services, OLDI query executions use complicated data analysis to extract answers. Consider the OpenEphyra question answering system. Each query execution reduces text documents to a few phrases by finding noun-verb answer templates within sentences.

OLDI services use large and growing data to improve the quality of their answers, but large data also increases processing demands. To keep response time low, OLDI query executions are parallelized across distributed software components. At Microsoft Bing, query execution invokes over 2,000 components in parallel. Each component contributes data analysis 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 can not wait for slow components. Instead, they compute answers with whatever data is available within response time constraints.

OLDI services answer queries quickly even though performance anomalies, failed hardware and skewed partitioning schemes slow down some parallel components. However, eliding data from slow components could degrade answer quality [9], [20]. In this paper, answer quality is the similarity between answers produced with and without data from slow components. Queries achieve high answer quality when their execution does not suffer from slow components or when slow components do not affect answers. Queries have low answer quality when slow components affect answers significantly. Prior work has shown the virtue of adaptive resource management with regard to response time. Adaptive management could also help OLDI services manage answer quality. For example, adaptive admission control could stabilize answer quality in the face of time-varying arrival rates.

Answer quality is hard to measure online because it requires 2 query executions. Figure 1 depicts the process of computing answer quality. First, an online execution provides answers within response time constraints by eliding data from slow components. Second, a *mature execution* provides mature answers by waiting for all components before producing answers. Finally, a service-specific similarity function computes answer quality. In this paper, our similarity function is the true positive rate, i.e., the percentage of mature answers represented in online answers. However, any function that compares mature and online answers is permissible. For example, normalized discounted cumulative gain measures answer quality in search [19].

We present Ubora[1]*, an 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, i.e., a mature execution can complete faster by reusing data from its corresponding online execution instead of re-invoking components.

[1]Ubora means *quality* in Swahili.

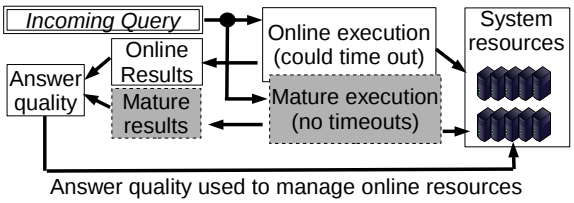

Fig. 1: Steps to measure answer quality online. Mature and online executions may overlap.

When a query arrives, Ubora conducts a normal online query execution except it records intermediate data provided by each component, including data from slow components that were elided from online answers. After the slow components finish, Ubora computes *mature answers* using data recorded during and after the online exeuction. Implementing memoization for multi-component OLDI services presents systems challenges. First, OLDI components span multiple platforms. It is challenging to coordinate mature and online executions across 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 stored in a shared key-value store. Second, memoization speeds up computationally intensive components but its increased bandwidth usage can also cause slowsdown for some components. Ubora profiles components to assess the benefits of memoization.

We have 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. Collectively, these services comprise 9 widely used open-source platforms. Ubora finishes mature executions 9X faster than naively re-executing queries. It slows down online queries by less than 15% for most workloads.

We also used Ubora to guide adaptive admission control. 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. Naively re-executing queries degraded answer quality by 12X. We also used component timeouts as a proxy for answer quality [15]. 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 processed 37% more low priority queries than admission control powered by component timeouts.

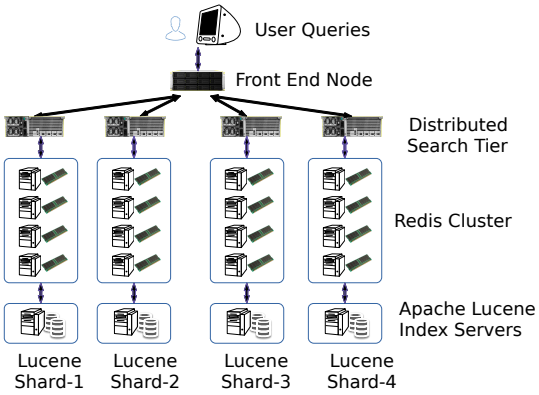

Fig. 2: Execution of a single query in Apache Lucene. Adjacent paths reflect parallel execution across data partitions.

This paper is organized as follows. We overview the structure of OLDI services in Section II. We present Ubora in Section III. Section IV presents our implementation of query context tracking and profiling for memoization. In Section V, we measure Ubora's performance using a wide range of OLDI benchmarks. In Section VI, we show that Ubora computes answer quality quickly enough to guide online load shedding.

II. Background on OLDI Services

Query executions differ fundamentally between online data-intensive (OLDI) and traditional Internet services. In traditional services, query executions process data retrieved from well structured databases, often via SQL. Correct query executions produce answers with well defined structure, i.e., answers are provably right or wrong. In contrast, OLDI queries execute on unstructured data. They produce answers by discovering correlations within data. OLDI services produce good answers if they process data relevant to query parameters.

Large datasets improve the quality of OLDI answers. For example, IBM Watson parsed 4TB of mostly public-domain data [10]. One of Watson's data sources, Wikipedia, grew 116X from 2004–2011 [4]. However, it is challenging to analyze a large dataset within strict response time limits. This section provides background on the software structure of OLDI services that enables the following:

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

Parallelized Query Execution: Figure 2 depicts a query execution in an Apache Lucene system, a widely used open-source information retrieval library [18]. Query execution invokes 25 software components. Components

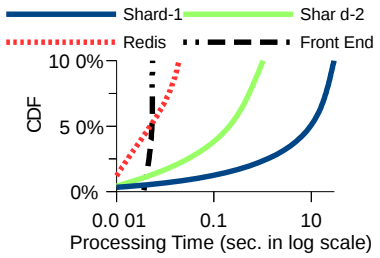

Fig. 3: OLDI components exhibit diverse processing times

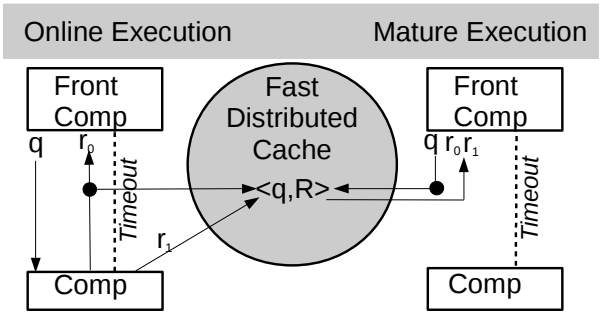

Fig. 4: Memoization in Ubora. Arrows reflect messages in execution order (left to right).

in adjacent columns can execute in parallel. A front-end node manages network connections with clients, sorts results from nodes running Distributed Search logic and produces final answers. Distributed Search parses the query, requests a wide range of relevant data from storage nodes, and collects 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 system in Figure 2 indexes 23.4 million Wikipedia and NY Times documents (pages + revisions) produced between 2001 and 2013. It parallelizes query execution via data parallelism, i.e., the Lucene Index Servers partition the index across multiple nodes. Each parallel sub-execution (i.e., a vertical column) computes intermediate data based on its underlying partition. Intermediate data is combined to produce a query response.

OLDI services also parallelize query executions via partial redundancy. In this approach, sub-executions compute intermediate data from overlapping data partitions. The query execution weights answers based on the degree of overlap and aggregate data processing per partition. Consider a product recommendation service. Its query execution may spawn two parallel sub-executions. The first finds relevant products from orders completed in the last hour. The second considers the last 3 days. The service prefers the product recommended by the larger (3-day) sub-execution. However, if the recommendation is unavailable or otherwise degraded, the smaller parallel sub-execution helps.

Online Answers Are Best Effort: In traditional Internet services, query execution invokes software components sequentially. Their response time depends on aggregate processing times of all components. In contrast, online data-intensive query executions invoke components in parallel. The processing time of the slowest components determines response time. Figure 3 quantifies component processing times in our Apache Lucene system. The query workload from Google Trends and hardware details are provided in Section V. Pocessing times vary significantly from query to query. Note, 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 shards. For example, a query for "William Shakespeare" transferred 138KB from the shard 4 execution path but only 1KB from shard 1. Shard 4 hosted more content related to this query even though the data was partitioned randomly.

Many OLDI services prevent slow components from delaying response time by returning answers prematurely—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 4-second timeout in our Front-end component. Average response time fell. Also, 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 elided data is relevant to query paramerts. For partially redundant sub-executions, answer quality depends on the representativeness of small data.

III. Design

Ubora measures the answer quality of online query executions. By design, it directly computes answer quality by comparing answers produced with and without timeouts. It uses existing online resources and employs memoization to speed up query executions.

Figure 4 depicts memoization in Ubora. During online query execution, Ubora records inter-component communication. It allows only front-end components to time out. Components invoked by parallel sub-executions complete in the background. As shown on the left side of Figure 4, without Ubora, the front-end component

invokes a component with query q, receives message r_0 and then times out. The front-end component then triggers a timeout for the invoked component, stopping its execution prematurely. Ubora blocks 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 stores.

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 sub-executions for a query complete, Ubora re-executes the front-end, as if a new query arrived. However, during this re-execution, Ubora intercepts messages to other components and serves from 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*.

IV. Implementation

This section discusses the implementation of Ubora. First, we describe axiomatic choices, e.g., the user interface, target users and prerequisite infrastructure. Second, we discuss the impact of operating system support on the implementation of memoization. Finally, we provide details about our implementation, including our approach to determine which components constitute a front-end.

A. Interface and Users

Ubora targets system managers. It runs on a cluster of compute nodes. Each node runs a networked operating system. Many software components can map to each node, but each component runs on just 1 node. To be sure, a software component is a running binary that accepts invocations over the network. Software components have unique network addresses, e.g., IP address and TCP port.

System managers understand the query execution paths in their system (e.g., as depicted in Figure 2). They classify each component as front- or back-end. Front components receive queries, record inter-component 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 a number of parameters before starting Ubora , shown in Listing 1. The number of mature executions to produce per unit time controls the query sampling rate. 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 True Positive Rate.

B. Impact of Operating System Support

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

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, queries executed have access to the same data under replay mode as they do under normal and record modes.
2. *Back-end components use more resources during record mode than they use during normal online execution because timeouts are disabled.*

Design with OS Managed Contexts: Some operating systems track execution context by annotating network messages and thread-local memory with context and ID. Dapper [25] instruments Google's threading libraries, Power Containers [24] tracks context switches between Linux processes and annotates TCP messages and Xtrace [11] instruments networked middleware.

OS-managed execution context simplifies our design. Ubora intercepts messages between components, acting as a middle box. Before delivering messages that initiate

Listing 1: YAML Configuration.

```
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
```

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 the query context and ID are matched. We split the value and form a new key when the invoking component sends another message.

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 to transparently provide the illusion of RPC. On misses, the message is delivered to the destination component as described above.

Design without OS Support: Most vanilla operating systems do not track execution context. Without such support, it is challenging to distinguish remote procedure calls between concurrent queries. However, 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. First, we describe a simple but broken idea that is highly transparent, 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. Also, 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, inter-component 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 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 2 (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 V-D.

Next we describe how to propagate request context, which is necessary when the operating system does not support execution contexts. On a front component, we wait for queries to arrive on a designated TCP port. If a query is selected for mature execution, we change 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, we send a UDP control message that tells that component to enter record mode and forwards the proper context id. Then we send the original message. Note, 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 roundtrips. 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 anymore. 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 the system has returned to normal.

Reducing Bandwidth Needs: Ubora reduces bandwidth required for context propagation. First, Ubora propagates context to only components used during online execution. Second, Ubora does not use bandwidth to return components to normal mode, only sending UDP messages when necessary to enable record or replay mode. Timeouts local to each component also increase robustness to network partitions and congestion, ensuring that components always return to normal execution context.

C. 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 can not apply memoization to the front-end component.

At first glance, re-execution seems slower than memoization. However, as shown in Figure 3, many components execute quickly. In some cases, execution is faster than transfering 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, the do not initiate queries. In Figure 2, Distributed Search or Redis components could be labeled middle components.

Given a trace of representative queries, Ubora determines which components to memoize by systematically measuring throughput with all different combinations of front-, middle- and back-end components.

D. Prototype

We implemented transparent context tracking as described above for the Linux 3.1 operating system. The implementation is installed as user-level package and requires the Linux Netfilter library to intercept and reroute TCP messages. It uses IPQueue to trigger context management processes. It assumes components communicate through remote procedure calls (RPC) implemented on TCP and that an IP address and TCP port uniquely identify each component. It also assumes timeouts are triggered by the RPC caller externally—not internally by the callee. It extends timeouts transparently by blocking FIN packets sent to the callee and spoofing ACKs to the caller. Messages from the callee that arrive after a blocked FIN are cached but not delivered to the caller. For workloads that use connection pooling, we block application-specific termination payloads. Service managers can specify this in the configuration file.

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. Our default setting is 1 GB. Also, 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.

V. Experimental Evaluation

In this section, we evaluate Ubora's throughput, slowdown, and accuracy using micro benchmarks and real multi-component OLDI services. We also show that Ubora can guide online management based on answer quality. First, we discuss the chosen metrics of merit. Then, we present the software and hardware architecture for the OLDI services used. Finally, we present experimental results that assess Ubora's design and implementation.

A. Metrics of Merit

To compute answer quality, one needs mature and premature results. The latter can be acquired during online execution easily but the former demands extra resources. The challenge is to acquire mature results while processing other queries online at the same time. Our primary metric used to evaluate Ubora's performance (throughput) is mature results per online query.

Getting mature results uses resources that otherwise could have been used for online queries, slowing the system down. Queries that were not sampled for mature execution (i.e., unsampled queries) are slowed because of queuing delays. Our scheduling techniques reduce slowdown for these queries. In addition to queuing delay, sampled queries also face delays caused by record and replay modes. We report *slowdown* as the relative increase in response time.

Ubora's design is very different from a state of the art approach based on extending timeouts. Using micro benchmarks, we can ensure that the mature results produced by Ubora are accurate, i.e., they produce the same mature results as extended timeouts.

We used *true positive rate*, i.e., the percentage mature results represented in online results, for answer quality.

B. OLDI Services

Table I describes each OLDI service used in our evaluation, highlighting diverse features and workload demands. In the rest of this paper, we will refer to these services using their codename. The setup shown in Figure 2 depicts LC.all, a 31 node cluster that supports 16 GB DRAM cache per TB 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, 30 GB local storage and (up to) 2 TB block storage.

- *LC.all* and *LC.wik* implement bag-of-words search based on the widely used Apache Lucene indexing platform. Figure 2 depicts query execution for these services. We evaluate with a 31-node cluster that has 4 Redis nodes to each Lucene node.
- *OE.jep* uses OpenEphyra, a question answering framework [1]. OpenEphyra uses Lucene and Redis

OLDI Service Features						Workload Demands		
Code	**Platform**	**Data**	**Size**	**# Nodes**	**Timeout**	**Avg. Maturity**	**Util**	**QCoD**
LC.all	Lucene(4)	Wikipedia+NYT	4 TB	31	5.0 Sec.	10%	40%	55%
LC.wik	Lucene(4)	Wikipedia	128 GB	31	2.0 Sec.	20%	23%	53%
OE.jep	OpenEphyra(5)	NYT	4 GB	8	2.0 Sec.	5%	20%	56%
ER.cpu	EasyRec(4)	Netflix Ratings	2 GB	3	3.0 Sec.	50%	44%	89%
ER.fst	EasyRec(4)	Netflix Ratings	2 GB	3	1.5 Sec.	75%	15%	89%
Micro	NanoWeb(1)	Syntheti c	NA	3	0.1 Sec.	NA	NA	NA

TABLE I: OLDI services tested. Salient platform is listed. Total platforms is shown in parenthesis.

for indexing and data caching. A front end answers a trace of 1K questions related to recent events by distributing each query and aggregates results from parallel invocations of OpenEphyra.

- *ER.cpu* and *ER.fst* feed Netflix movie ratings [2] into the EasyRec recommendation engine [3]. Our 20K-query trace submits randomly selected movie IDs and returns a set of related (recommended) movies for each. The front end contacts two EasyRec engines simultaneously; one uses 10X more ratings than the other. The engine with more ratings normally takes longer to respond but provides better recommendations. In ER.fst, the front end responds fast by forwarding output from the best engine that completes within timeout. In ER.cpu, the front end does a CPU-intensive merge by eliding recent recommendations. Unlike the other services, Netflix data fits in main memory.
- *Micro* is a micro benchmark written in PHP. The front component reads a sequence of integers from the two targeted components. All components run in NanoWeb. The target components write a value every millisecond. The output is largest value observed. Each query specifies the running time of each component, allowing us to know the expected output in advance.

The services above support interactive queries. We set up a workload generator that replayed trace workloads at a set arrival rate. Table I also shows the utilization defined as $\frac{ArrivalRate}{ProcessingRate}$. As utilization increases, Ubora is challenged to achieve record, cache and replay without creating too much queuing delay. Table I also shows the quartile coefficient of distribution for the 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.

C. Competing Approaches

We evaluate Ubora by comparing the following competing designs:

- The *extended-TO* design extends timeouts by 4X for 40 seconds. Queries that arrive when timeouts are extended return mature results. Then it returns to normal timeouts until the sample rate is exceeded. Queries run under extended timeouts are rerun under normal timeouts to get online results needed for answer quality.
- We implemented *naive* Ubora. It broadcasts execution context from the front component for record, replay and normal mode without local timeouts.
- Our tests with full Ubora set the parameters in Listing 1 to maximize throughput. We also test *low-overhead* Ubora, where the sampling rate is manually lowered in the configuration file to reduce overhead.

D. Results

Microbenchmark Tests: Our first test studied the effect of data skew and component selection. We issued 10,000 queries to Micro one after another. Each query randomly selected 1 component to have a running time X% longer than the other. Here, X% reflects data skew. Recall, the output of each component is its running time and the benchmark's output is the largest observed running time. The front component times out after the shortest component completes (100 milliseconds).

The left axis of Figure 5(A) shows the accuracy of mature results in this test, i.e., the relative error between our mature results and the running time of the slowest component. We report average error. The top dotted line shows results when both components are targets. The dashed line shows results when only one component is a target. When both components are targets, accuracy ranges between 96-99%. However, the *Partial Record* line warns about the perils of selecting targets poorly. Consider the extreme case where the shortest component runs for 100 milliseconds and the longest runs for 200 milliseconds. If the wrong component was selected,

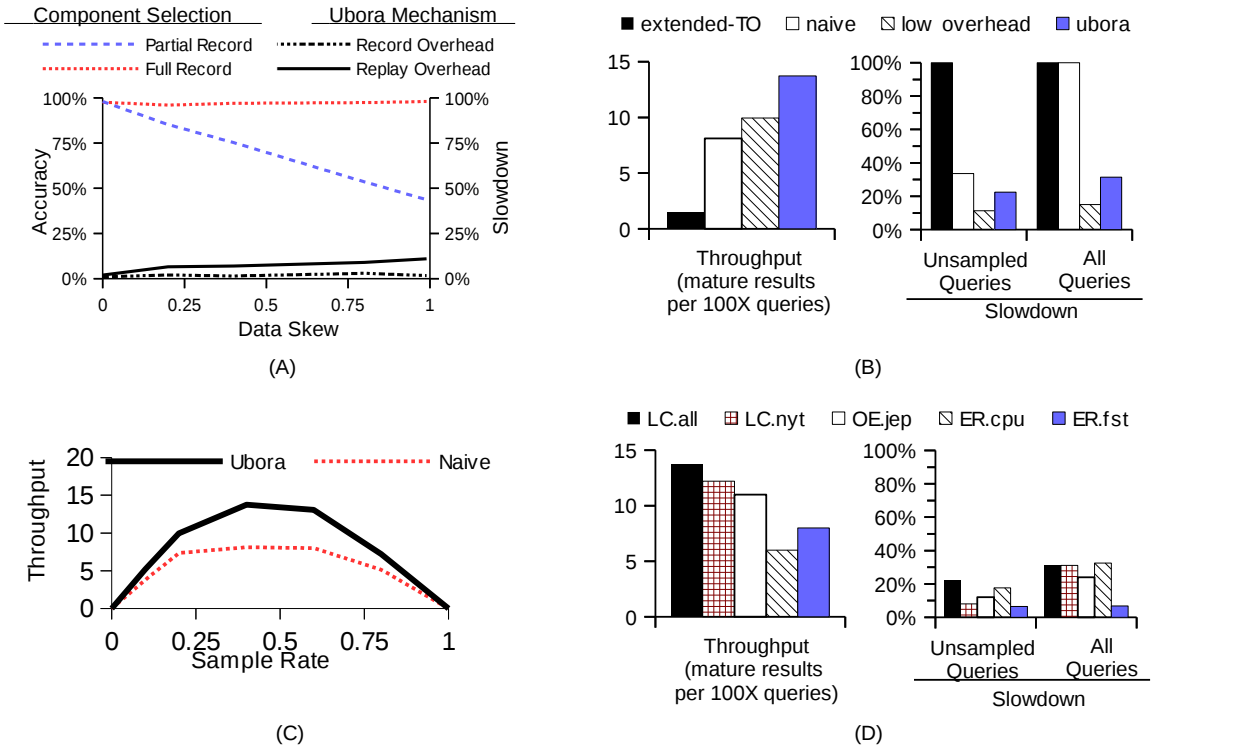

Fig. 5: **Experimental results:** (a) Micro benchmarks highlight Ubora's accuracy and efficiency, (b) Throughput and slowdown of competing approaches, (c) The effect of sampling rate, and (d) Throughput and slowdown across multiple, widely used services.

the best possible accuracy is 50%. Record and replay overheads cause further degradation. On the right axis of Figure 5(A), we report slowdown, i.e., increase in response time, caused by record mode and replay mode in the experiments respectively. 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 unit of data skew. These tests show that our record and replay mechanism are implemented efficiently.

Comparison to Extended Timeouts: We used LC.all to compare the performance and mature execution throughput of competing designs. Figure 5(B) shows that record, cache and replay approaches achieve greater throughput than extended timeouts. To explain this result, we use a concrete example of a search for "Mandy Moore." Both approaches are accurate. 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. The mature execution took 21 seconds in record mode and 4 seconds in replay mode. Under extended timeouts, service times for all concurrent queries increase by 4X. Utilization exceeded system capacity, causing execution time to take 589 seconds.

Second, Ubora uses online executions to speed up mature executions. Replay mode completes quickly under normal timeout settings. In this case, replay completed in 4.5 seconds. Replay and record mode complete this mature execution within 60 seconds.

Comparing Ubora to Low-overhead Ubora: Low-overhead and full versions present interesting tradeoffs. Full Ubora can achieve 38% greater throughput but the cost is 2X slowdown across all queries. Ubora prioritizes online executions by leaving normal timeouts unchanged. It uses idle periods to schedule mature executions. However, as the sampling rate increases, idle periods are less available and mature executions cause queuing delay. Figure 5(C) studies the effect of sampling rate more closely. Along the x-axis, we vary the sampling rate, i.e., mature executions sampled per normal query. The y-axis shows the achieved throughput of mature

executions. The peak sampling rate corresponds to 12 queries per minute. First, we observe that the failure rate increases with the sampling rate. This is due to expired cache entries and slow state propagation. Under 20% sampling rate, 17% of mature execution fail to yield mature results. This rises to 84% at 80% 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.

Evaluation Across Workloads: Figure 5(D) shows the diverse workloads supported by the transparent system design. Collectively, the 5 workloads shown use 9 platforms including 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.

VI. Online Management

OLDI services are 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 enables better resource management through answer quality. In this case study, we use Ubora to improve load shedding.

Control Theory with Answer Quality:

We used LC.All for our load shedding study. Using diurnal traces from previous studies [26], we issued two classes of queries: high and low priority. The queries were directed to two different TCP ports. At the peak workload, low and high priority arrival rates saturate system resources (i.e., utilization is 90%). Figure 6 shows the *Arrival Rate* over time (on the right axis). At the 45 minute and 2 hour mark, the query mix shifts toward multiple word queries that take longer to process fully.

We used Ubora to track answer quality for high priority queries. Here, answer quality is the true positive rate for the top 20 results. When quality dipped, we increased the load shedding rate on low priority queries. Specifically, we used a proportional-integral-derivative (PID) controller. Every 100 requests, we computed average answer quality.

The left axis of Figure 6 shows answer quality of competing load shedding approaches. When all low priority queries are shed, the *No Sharing* approach maintains answer quality above 90% throughout the trace.

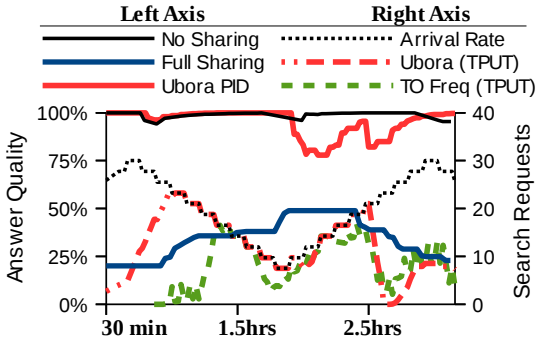

Fig. 6: Answer quality measured online is used to control load shedding.

When shedding is disabled, the *Full Sharing* approach sees answer quality drop as low as 20%, corresponding with peak arrival rates. The PID controller powered by Ubora manages the shed rate well, keeping answer quality above 90% in over 90% of the trace. It maintains throughput (*Ubora TPUT*) of almost 60% of low priority queries (shown on the right axis).

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. For comparison, we implemented a PID controller that used frequency of timeouts instead 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. It assumes that partial results caused by timeouts are dissimilar to mature results. Figure 6 also shows that the controller based on timeout frequency (*TO Freq*) sheds requests too aggressively. Our approach improved throughput on low priority queries by 37%.

Sampling Rate and Representativeness: The record, cache and replay design allows managers to reduce the aggregate overhead of mature executions by sampling fewer 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 load shedding. We observed that sampling 5% of online queries significantly increased outlier errors on answer quality, but our adaptive load shedding remained effective—it still achieved over 90% quality over 90% of the trace. In contrast, a 2% sampling produced many quality violations.

Sampling Rate	Absolute Relative Error of Measured Answer Quality		Rate of Quality Violations
	Avg.	95th %tile	
10%	0%	0%	4%
5%	20%	45%	9%
3%	30%	50%	13%
2%	51%	78%	29%

TABLE II: Adaptive management degrades under low sampling rates. A *quality violation* is a window where answer quality falls below 90%. Reported error is relative to the 10% sampling rate.

VII. Related Work

Ubora provides runtime support toward approximate computing [21], [23]. It directly measures imprecision (i.e., answer quality) by comparing approximate and mature results for online data-intensive services. It works quickly enough to help manage system resources online.

Paraprox is a compiler and runtime manager that detects workloads amenable to approximate computing and tunes them for quality-aware performance. It supports map, scatter-gather and reduce workloads and does not require re-programming. Paraprox does not compare approximate and mature results. Instead, it uses proxies, e.g., sampling rate, to both measure and tune quality [22]. Jalaparti et al. also use a proxy for answer quality, based on timeout frequencies [15]. Prior work used offline comparisons of partial and full results as a proxy for online measurements [14], [16], [20].

Ubora samples queries to produce answer quality. One sample may differ from the expected value of the underlying quality distribution. Uncertain<T> allows programmers to reason about such random variables [6], allowing statistical confidence to affect variable comparisons. EneryJ also helps programmers to differentiate possibly approximate from must-be-correct computation [23]. Stabilizer creates statistical noise by changing memory layouts. This helps programmers conduct experiments on statistical metrics [8].

Prior work also highlights other workloads amenable to approximation. Web content adaptation [7], [13] degrades image quality and webpage features to meet response time goals using image resolution as a proxy for quality. Baek and Chilimbi [5] present a general framework to support approximate computation. Anytime algorithms [27] define a class of problems that can be solved incrementally.

VIII. 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. Record, cache and replay is an approach to produce mature results unaffected by timeouts. Mature results are required to assess the impact of timeouts—i.e., answer quality. Our approach avoids offline resources and keeps response times low. Ubora implements our approach with a focus on transparency and efficiency. The evaluation shows that Ubora produces mature results faster than competing approaches. Ubora produces answer quality quickly enough to enhance online system management.

References

[1] The ephyra question answering system. http://www.ephyra.info/.

[2] Netflix prize. http://www.netflixprize.com//index.

[3] Easyrec-open source recommendation engine. http://easyrec.org/, 2014.

[4] Wikipedia:modelling wikipedia's growth. http://en.wikipedia.org/wiki/Wikipedia:Modelling_Wikipedias_growth, 2014.

[5] W. Baek and T. M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, 2010.

[6] J. Bornholt and K. McKinley. Uncertain ¡t¿: A first-order type for uncertain data. In *ACM ASPLOS*, 2014.

[7] Y. Chen. Detecting web page structure for adaptive viewing on small form factor devices. In *WWW*, 2003.

[8] C. Curtsinger and E. Berger. Stabilizer: Statistically sound performance evaluation. In *ACM ASPLOS*, 2013.

[9] R. Falsett, R. Seyer, and C. Siemers. Limitation of the response time of a software process, Dec. 29 2004. WO Patent App. PCT/EP2003/000,721.

[10] D. Ferrucci, E. Brown, J. Chu-Carroll, J. Fan, D. Gondek, A. Kalyanpur, A. Lally, J. Murdock, E. Hyberg, J. Prager, N. Schlaerfer, and C. Welty. The ai behind watson—the technical article. In *The AI Magazine*, 2010.

[11] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *USENIX NSDI*, 2007.

[12] B. Forrest. Bing and google agree: Slow pages lose users. radar.oreilly.com, 2009.

[13] A. Fox, S. D. Gribble, Y. Chawathe, and E. A. Brewer. Adapting to network and client variation using infrastructural process proxies: lessons and perspectives. *Personal Communications*, 5:10–19, 1998.

[14] Y. He, S. Elnikety, J. Larus, and C. Yan. Zeta: scheduling interactive services with partial execution. In *SOCC*, 2012.

[15] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up distributed request-response workflows. In *SIGCOMM*, 2013.

[16] J. Kelley, C. Stewart, S. Elnikety, and Y. He. Cache provisioning for interactive nlp services. In *Workshop on Large-Scale Distributed Systems and Middleware*, 2013.

[17] J. Lenchner. Knowing what it knows: selected nuances of watson's strategy. http://ibmresearchnews.blogspot.com.

[18] Lucid Imagination. The case for lucene/solr: Real world search applications. White Paper, 2008.

[19] C. Manning, P. Raghavan, and H. Schtze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[20] S. Ren, Y. He, S. Elnikety, and K. McKinley. Exploiting processor heterogeneity in interactive services. In *IEEE ICAC*, 2013.

[21] L. Renganarayana, V. Srinivasan, R. Nair, and D. Prener. Context-aware adaptive approximation. In *First Workshop on Approximate Computing Across the Systems Stack* , 2014.

[22] M. Samadi and S. Mahilke. Paraprox: Pattern-based approximation for data parallel applications. In *ACM ASPLOS*, 2014.

[23] A. Sampson, L. Ceze, and D. Grossman. Enerj, the language of good-enough computing, 2013.

[24] K. Shen, A. Shriraman, S. Dwarkadas, X. Zhang, and Z. Chen. Power containers: An os facility for fine-grained power and energy management on multicore servers. In *ACM ASPLOS*, 2012.
[25] B. Sigelman, L. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbag. Dapper, a large-scale distributed systems tracing infrastructure. In *Google Technical Report*, 2010.
[26] C. Stewart, T. Kelly, and A. Zhang. Exploiting nonstationarity for performance prediction. In *EuroSys Conf.*, Mar. 2007.
[27] S. Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3), 1996.