DQBarge: Improving Data-Quality Tradeoffs in Large-Scale Internet Services

Michael Chow, *University of Michigan;* Kaushik Veeraraghavan, *Facebook, Inc.;*
Michael Cafarella and Jason Flinn, *University of Michigan*

https://www.usenix.org/conference/osdi16/technical-sessions/presentation/chow

This paper is included in the Proceedings of the
12th USENIX Symposium on Operating Systems Design
and Implementation (OSDI '16).

November 2–4, 2016 • Savannah, GA, USA

ISBN 978-1-931971-33-1

DQBarge: Improving data-quality tradeoffs in large-scale Internet services

Michael Chow[*], *Kaushik Veeraraghavan*[†], *Michael Cafarella*[*], *and Jason Flinn*[*],
University of Michigan[*] *Facebook, Inc.*[†]

Abstract

Modern Internet services often involve hundreds of distinct software components cooperating to handle a single user request. Each component must balance the competing goals of minimizing service response time and maximizing the quality of the service provided. This leads to low-level components making *data-quality tradeoffs*, which we define to be explicit decisions to return lower-fidelity data in order to improve response time or minimize resource usage.

We first perform a comprehensive study of low-level data-quality tradeoffs at Facebook. We find that such tradeoffs are widespread. We also find that existing data-quality tradeoffs are often suboptimal because the low-level components making the tradeoffs lack global knowledge that could enable better decisions. Finally, we find that most tradeoffs are reactive, rather than proactive, and so waste resources and fail to mitigate system overload.

Next, we develop DQBarge, a system that enables better data-quality tradeoffs by propagating critical information along the causal path of request processing. This information includes data provenance, load metrics, and critical path predictions. DQBarge generates performance and quality models that help low-level components make better, more proactive, tradeoffs. Our evaluation shows that DQBarge helps Internet services mitigate load spikes, improve utilization of spare resources, and implement dynamic capacity planning.

1 Introduction

A *data-quality tradeoff* is an explicit decision by a software component to return lower-fidelity data in order to improve response time or minimize resource usage. Data-quality tradeoffs are often found in Internet services due to the need to balance the competing goals of minimizing the service response time perceived by the end user and maximizing the quality of the service provided. Tradeoffs in large-scale services are pervasive since hundreds or thousands of distinct software components may be invoked to service a single request and each component may make individual data-quality tradeoffs.

Data-quality tradeoffs in low-level software components often arise from defensive programming. A programmer or team responsible for a specific component wishes to bound the response time of their component even when the resource usage or latency of a sub-service is unpredictable. For instance, a common practice is to time out when a sub-service is slow to respond and supply a default value in lieu of the requested data.

To quantify the prevalence of data-quality tradeoffs, we undertake a systematic study of software components at Facebook. We find that over 90% of components perform data-quality tradeoffs instead of failing. Some tradeoffs we observe are using default values, calculating aggregates from a subset of input values, and retrieving alternate values from a stale or lower-quality data source. Further, we observe that the vast majority of data-quality tradeoffs are reactive rather than proactive, e.g., components typically set timeouts and make data-quality tradeoffs when timers expires rather than predict which actions can be performed within a desired time bound.

These existing data-quality tradeoffs are suboptimal for three reasons. First, they consider only local knowledge available to the low-level software component because of the difficulty in accessing higher-level knowledge such as the provenance of data, system load, and whether the component is on the critical request path. Second, the tradeoffs are usually reactive (e.g., happening only after a timeout) rather than proactive (e.g., issuing only the amount of sub-service requests that can be expected to complete within a time bound); reactive tradeoffs waste resources and exacerbate system overload. Finally, there is no mechanism to trace the set of data-quality tradeoffs made during a request, and this makes understanding the quality and performance impact of such tradeoffs on actual requests difficult.

DQBarge addresses these problems by propagating critical information along the causal path of request processing. The propagated data includes load metrics, as well as the expected critical path and slack for individual software components. It also includes provenance for request data such as the data sources queried and the software components that have transformed the data. Finally, it includes the specific data-quality tradeoffs that have been made for each request; e.g., which data values were left out of aggregations.

In an offline stage, DQBarge uses this data to generate performance and quality models for low-level tradeoffs in the service pipeline. Later, while handling production traffic, it consults the models to proactively determine which tradeoffs to make.

DQBarge generates performance and quality models by sampling a small percentage of the total requests processed by the service and redundantly executing them to compare the performance and quality when different tradeoffs are employed. Redundant execution minimizes interference with production traffic; duplicated requests run offline on execution pipelines dedicated to model generation. Performance models capture how throughput and latency are affected by specific data-quality tradeoffs as a factor of overall system load and provenance. Quality models capture how the fidelity of the final response is affected by specific tradeoffs as a function of input data provenance.

These models enable better tradeoffs during the processing of subsequent production requests. For each production request, DQBarge passes extra data along the causal path of request processing. It predicts the critical path for each request and which software components will have substantial slack in processing time. It also measures current system load. This global and request-specific state is attached to the request at ingress. As the request propagates through software components, DQBarge annotates data objects with provenance. This information and the generated models are propagated to the low-level components, enabling them to make better tradeoffs.

We investigate three scenarios in which better data-quality tradeoffs can help. First, during unanticipated load spikes, making better data quality tradeoffs can maintain end-to-end latency goals while minimizing the loss in fidelity perceived by users. Second, when load levels permit, components with slack in their completion time can improve the fidelity of the response without impacting end-to-end latency. Finally, understanding the potential effects of low-level data-quality tradeoffs can inform dynamic capacity planning and maximize utility as a function of the resources required to produce output.

One way to frame this work is that data-quality tradeoffs are a specific type of quality-of-service tradeoff [7, 25, 29], akin to recent work in approximate computing [4, 8, 19, 18, 28, 30]. The distinguishing feature of data-quality tradeoffs is that they are embedded in low-level software components within complex Internet pipelines. This leads to a lack of global knowledge and makes it difficult for individual components to determine how making specific tradeoffs will impact overall service latency and quality. DQBarge addresses this issue by incorporating principles from the literature on causal tracing [5, 9, 10, 13, 23, 26, 27, 31] to propagate needed knowledge along the path of request processing, enabling better tradeoffs by providing the ability to assess the impact of tradeoffs.

Thus, this work makes the following contributions. First, we provide the first comprehensive study of low-level data-quality tradeoffs in a large-scale Internet service. Second, we observe that causal propagation of request statistics and provenance enables better and more proactive data-quality tradeoffs. Finally, we demonstrate the feasibility of this approach by designing, implementing, and evaluating DQBarge, an end-to-end approach for tracing, modeling, and actuating data-quality tradeoffs in Internet service pipelines.

We have added a complete, end-to-end implementation of DQBarge to Sirius [15], an open-source, personal digital assistant service. We have also implemented and evaluated the main components of the DQBarge architecture at Facebook and validated them with production data. Our results show that DQBarge can meet latency goals during load spikes, utilize spare resources without impacting end-to-end latency, and maximize utility by dynamically adjusting capacity for a service.

2 Study of data-quality tradeoffs

In this section, we quantify the prevalence and type of data-quality tradeoffs in production software at Facebook. We perform a comprehensive study of Facebook client services that use an internal key-value store called *Laser*. *Laser* enables online accessing of the results of a batch offline computation such as a Hive [33] query.

We chose to study clients of *Laser* for several reasons. First, *Laser* had 463 client services, giving us a broad base of software to examine. We systematically include all 463 services in our study to gain a representative picture of how often data-quality tradeoffs are employed at Facebook. Second, many details about timeouts and tradeoffs are specified in client-specific RPC configuration files for this store. We processed these files automatically, which reduced the amount of manual code inspection required for the study. Finally, we believe a key-value store is representative of the low-level components employed by most large-scale Internet companies.

Table 1 shows the results of our study for the 50 client services that invoke *Laser* most frequently, and Table 2 shows results for all 463 client services. We categorize how clients make data-quality decisions along two dimensions: proactivity and resultant action. Each entry shows the number of clients that make at least one data quality decision with a specific proactivity/action combination. For most clients, all decisions fall into a single category. A few clients use different strategies at different points in their code. We list these clients in multiple categories, so the total number of values in each table is slightly more than the number of client services.

	Failure	Data-quality tradeoff		
		Default	Omit	Alternate
Reactive	5 (10%)	14 (28%)	30 (60%)	1 (2%)
Proactive	0 (0%)	0 (0%)	2 (4%)	1 (2%)

Table 1: Data-quality decisions of the top 50 *Laser* clients. Each box shows the number of clients that make decisions according to the specified combination of reactive/proactive determination and resultant action. The total number of values is greater than 50 since a few clients use more than one strategy.

	Failure	Data-quality tradeoff		
		Default	Omit	Alternate
Reactive	40 (9%)	250 (54%)	174 (38%)	4 (1%)
Proactive	0 (0%)	3 (1%)	7 (2%)	1 (0%)

Table 2: Data-quality decisions made by all *Laser* clients. Each box shows the number of clients that make tradeoffs according to the specified combination of reactive/proactive determination and resultant action. The total number of values is greater than 463 since a few clients use more than one strategy.

2.1 Proactivity

We consider a tradeoff to be *reactive* if the client service always initiates the request and then uses a timeout or return code to determine if the request is taking too long or consuming too many resources. For instance, we observed many latency-sensitive clients that set a strict timeout for how long to wait for a response. If *Laser* takes longer than the timeout, such clients make a data-quality tradeoff or return a failure.

A *proactive* check predicts whether the expected latency or resource cost of processing the request will exceed a threshold. If so, a data-quality tradeoff is made immediately without issuing the request. For example, we observed a client that determines whether or not a query will require cross-data-center communication because such communication would cause it to exceed its latency bound. If there are no hosts that can service the query in its data center, it makes a data-quality tradeoff.

2.2 Resultant actions

We also examine the actions taken in response to latency or resource usage exceeding a threshold. *Failure* shows the number of clients that require a response from *Laser*. If the store responds with an error or timeout, the client fails. Such instances mean a programmer has chosen to not make a data-quality tradeoff.

The remaining categories represent different types of data-quality tradeoffs. *Default* shows the number of clients that return a pre-defined default answer when a tradeoff is made. For instance, we observed a client service that ranks chat threads according to their activity level. The set of most active chat groups are retrieved from *Laser* and boosted to the top of a chat bar. If retrieving this set fails or times out, chat groups and contacts are listed alphabetically.

The *Omit* category is common in clients that aggregate hundreds of values from different sources; e.g., to generate a model. If an error or timeout occurs retrieving values from one of these sources, those values are left out and the aggregation is performed over the values that were retrieved successfully.

One example we observed is a recommendation engine that aggregates candidates and features from several data sources. It is resilient to missing candidates and features. Although missing candidates are excluded from the final recommendation and missing features negatively affect candidate scores in calculating the recommendation, the exclusion of a portion of these values allows a usable but slightly lower-fidelity recommendation to be returned in a timely manner in the event of failure or unexpected system load.

The *Alternate* category denotes clients that make a tradeoff by retrieving an alternate, reduced quality, value from a different data source. For example, we observed a client that requests a pre-computed list of top videos for a given user. If a timeout or failure occurs retrieving this list, the client retrieves a more generic set of videos for that user. As a further example, we observed a client that chooses among a pre-ranked list of optimal data sources. On error or timeout, the client retrieves the data from the next best data source. This process continues until a response is received.

Before performing our study, we hypothesized that client services might try to retrieve data of equal fidelity from an alternate data store in response to a failure. However, we did not observe any instance of this behavior in our study (all alternate sources had lower-fidelity data).

2.3 Discussion of results

Tables 1 and 2 show that data quality tradeoffs are pervasive in the client services we study. 90% of the top 50 *Laser* clients and 91% of all 463 clients perform a data-quality tradeoff in response to a failure or timeout; the remaining 9-10% of clients consider the failure to retrieve data in a timely manner to be a fatal error. Thus, in the Facebook environment, making data-quality tradeoffs is normal behavior, and failures are the exception.

For the top 50 clients, the most common action when faced with a failure or timeout is to omit the requested value from the calculation of an aggregate (60%). The next most common action (28%) is to use a default value in lieu of the requested data. These trends are reversed when considering all clients. Only 36% of all 463 clients omit the requested values from an aggregation, whereas 52% use a default value.

We were surprised that only a few clients react to failure or timeout by attempting to retrieve the requested data from an alternate source (4% of the top 50 clients and 1% of all clients). This may be due to tight time or resource constraints; e.g., if the original query takes too long, there may be no time left to initiate another query.

Only 6% of the top 50 clients and 2% of all clients are proactive. The lack of proactivity represents a significant lost opportunity for optimization because requests that timeout or fail consume resources but produce no benefit. This effect can be especially prominent when requests are failing due to excessive load; a proactive strategy would decrease the overall stress on the system. When a proactive check fails, the service performing that check always makes a data-quality tradeoff (as opposed to terminating its processing with a failure); it would be very pessimistic for a client to return a failure without at least attempting to fetch the needed data.

In our inspection of source code, we observed that low-level data-quality decisions are almost always encapsulated within clients and not reported to higher-level components or attached to the response data. Thus, there is no easy way for operators to check how the quality of the response being sent to the user has been impacted by low-level quality tradeoffs during request processing.

3 Design and implementation

Motivated by our study results, we designed DQBarge to help developers understand the impact of data-quality tradeoffs and make better, more proactive tradeoffs to improve quality and performance. Our hypothesis is that propagating additional information along the causal path of request processing will provide the additional context necessary to reach these goals.

DQBarge has two stages of operation. During the offline stage, it samples a small percentage of production requests, and it runs a copy of each sampled request on a duplicate execution pipeline. It perturbs these requests by making specific data quality tradeoffs and measuring the request latency and result quality. DQBarge generates performance and quality models by systematically sweeping through the space of varying request load and data provenance dimensions specified by the developer and using multidimension linear regression over the data to predict performance and quality as a factor of load and provenance. Note that because requests are duplicated, end users are not affected by the perturbations required to gather model data. Further, DQBarge minimizes interference with production traffic by using dedicated resources for running duplicate requests as much as possible.

During the online stage, DQBarge uses the quality and performance models to decide when to make data-quality tradeoffs for production traffic in order to realize a configured goal such as maximizing quality subject to a specified latency constraint. It gathers the inputs to the models (load levels, critical path predictions, and provenance of data) and propagates them along the critical path of request execution by embedding the data in RPC objects associated with the request. At each potential tradeoff site, the low-level component calls DQBarge. DQBarge performs a model lookup to determine whether to make a data-quality tradeoff, and, if so, the specific tradeoff to make (e.g., which values to leave out of an aggregation). The software service then makes these tradeoffs proactively. DQBarge can optionally log the decisions that are made so that developers can understand how they are affecting production results.

The separation of work into online and offline stages is designed to minimize overhead for production traffic. These stages can run simultaneously; DQBarge can generate a new model offline by duplicating requests while simultaneously using an older model to determine what tradeoffs to make for production traffic. The downside of this design is that DQBarge will not react immediately to environmental changes outside the model parameters such as a code update that modifies resource usage. Instead, such changes will be reflected only after a new model is generated. We therefore envision that models are regenerated regularly (e.g., every day) or after significant environmental changes occur (e.g., after a major push of new code).

Section 3.1 describes how DQBarge gathers and propagates data about request processing, including system load, critical path and slack predictions, data provenance, and a history of the tradeoffs made during request processing. This data gathering and propagation is used by both the online and offline stages. Section 3.2 relates how DQBarge duplicates the execution of a small sample of requests for the offline stage and builds models of performance and quality for potential data-quality tradeoffs. As described in Section 3.3, DQBarge uses these models during the online stage to make better tradeoffs for subsequent requests: it makes proactive tradeoffs to reduce resource wastage, and it uses provenance to choose tradeoffs that lead to better quality at a reduced performance cost. Finally, Section 3.4 describes how DQBarge logs all tradeoffs made during request processing so that operators can review how system performance and request quality have been impacted.

3.1 Data gathering and propagation

DQBarge provides a library for developers to specify the information that should be propagated along the critical path. The library is implemented in 3268 lines of C++ code, plus Java bindings for services implemented in that language. Developers use the library interface to annotate objects during request processing and query those annotations at later stages of the pipeline. Table 3 shows selected functions from the DQBarge library API to which we will refer in the following discussion.

The DQBarge library has a RPC-package-specific back-end that modifies and queries existing RPC objects

DQBarge API
`putMetric(scope, key, type, value)`
`getMetric(key) → (type, value)`
`addProvenance(data object, key, type, value)`
`removeProvenance(data object, key)`
`getProvenance(data object) → list <key, type, value>`
`makeAggregationTradeoff(performance model, quality model, list<key>, list<object>) → list<object>`

Table 3: Selected functions from the DQBarge API

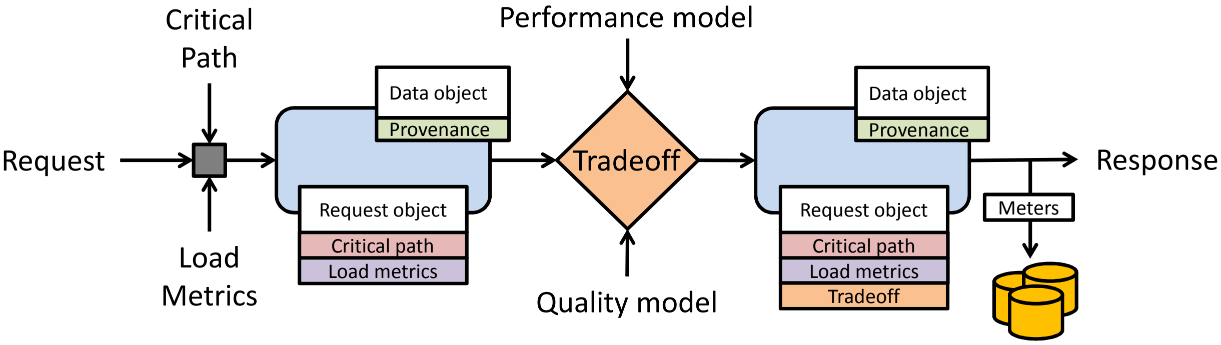


Figure 1: DQBarge overview.

to propagate the information. It modifies RPC objects by adding additional fields that contain data to be propagated along the causal path. It supports three object scopes: request-level, component-level, and data-level.

Request-level data are passed through all components involved in processing the request, following the causal path of request execution; such data includes system-wide load metrics, slack predictions, and a list of actual data-quality tradeoffs made during execution of the particular request. Services call `putMetric` to add this data to the request, specifying `request` as the scope and a typed key-value pair to track. Later, they may retrieve the data by calling `getMetric` with the specified key. The services in our case studies both have a global object containing a unique request identifier; DQBarge appends request-level information to this object. This technique for passing and propagating information is widely used in other tracing systems that follow the causal path of execution [13, 23].

Component-level objects persist from the beginning to end of processing for a specific software component within the request pipeline. Such objects are passed to all sub-components that are called during the execution of the higher-level component. DQBarge appends component-specific data to these objects, so such data will be automatically deallocated when execution passes beyond the specified component. Component-specific load metrics are one example of such data. To add this data, services call `putMetric` and specify a component-level RPC object as the scope.

Data-level objects are the specific data items being propagated as a result of request execution. `addProvenance` associates a typed key-value pair with a specific data object, since the provenance is meaningful only as long as the data object exists. A data object may have multiple provenance values.

Our library provide a useful interface for manipulating RPC objects, but developers must still make domain-specific decisions, e.g., what metrics and provenance values to add, what objects to associate with those values, and what rules to use to model the propagation of provenance. For instance, to reflect the flow of provenance in a component, developers should call `getProvenance` to retrieve the provenance of the inputs and `addProvenance` and `removeProvenance` to show causal propagation to outputs. Figure 1 shows an overview of how this data propagates through the system.

Load metrics may be relevant to the entire request or only to certain components. Each load metric is a typed key-value pair (e.g., a floating point value associated with the key “requests/second”). Currently supported load metrics are throughput, CPU load, and memory usage.

Critical path and slack predictions are specified as directed acyclic graphs. Each software component in the graph has a weight that corresponds to its predicted slack (the amount of additional time it could take to process a request without affecting the end-to-end latency of the request). Components on the critical path of request execution have zero slack. DQBarge relies on an external component, the Mystery Machine, to make critical path and slack predictions; [11] describes the details of that system. Currently, slack predictions are made at request ingress; such predictions may cover the entire request or only specific components of the request. The graphs for our two case studies in Section 4 are relatively small, so we transmit this data by value along the request processing path (using `PutMetric` and a graph type). If we were to deploy DQBarge along the entire Facebook request processing path, then the graphs would be much larger, and we would likely need to transmit them by reference or only send relevant subgraphs to components.

DQBarge associates provenance with the data objects it describes. Provenance can be a data source or the algorithm employed to generate a particular object. Provenance is represented as an unordered collection of typed key-value pairs. DQBarge supports both discrete and continuous types. DQBarge extracts a schema for the quality model from the data objects passed to tradeoff functions such as `makeAggregationTradeoff` by iterating through all provenance entries attached to each object to read the provenance keys and their associated types. Components are treated as black boxes, so developers must specify how provenance is propagated when a component modifies existing data objects or creates new ones.

Finally, DQBarge stores the tradeoffs that were made during request processing in a request-level object. As described in Section 3.4, this information may be logged and used for reporting the effect of tradeoffs on quality and performance.

3.2 Model generation

For each potential tradeoff, DQBarge creates a performance model and a quality model that capture how the tradeoff affects request execution. Performance models predict how throughput and latency are affected by specific data-quality tradeoffs as a factor of overall system load and the provenance of input data. Quality models capture how the fidelity of the final response is affected by specific tradeoffs as a function of provenance.

DQBarge uses *request duplication* to generate models from production traffic without adversely affecting the user experience. At the RPC layer, it randomly samples incoming requests from production traffic, and it routes a copy of the selected requests to one or more request duplication pipelines. Such pipelines execute isolated, redundant copies of the request for which DQBarge can make different data-quality tradeoffs. These pipelines do not return results to the end user and they are prevented from making modifications to persistent stores in the production environment; in all other respects, request execution is identical to production systems. Many production systems, including those at Facebook, already have similar functionality for testing purposes, so adding support for model generation required minimal code changes.

DQBarge controls the rate at which requests enter the duplication pipeline by changing the sampling frequency. At each potential tradeoff site, services query DQBarge to determine which tradeoffs to make; DQBarge uses these hooks to systematically explore different tradeoff combinations and generate models. For instance, `makeAggregationTradeoff` specifies a point where values can be omitted from an aggregation; this function returns a list of values to omit (an empty list means no tradeoff). DQBarge has similar functions for each type of tradeoff identified in Section 2.

To generate a performance model, DQBarge uses load testing [20, 24]. Each data-quality tradeoff offers multiple fidelities. A default value may be used or not. Different types or percentages of values can be left out of an aggregation. Multiple alternate data stores may be used. For each fidelity, DQBarge starts with a low request rate and increases the request rate until the latency exceeds a threshold. Thus, the resulting model shows request processing latency as a function of request rate and tradeoffs made (i.e., the fidelity of the tradeoff selected). DQBarge also records the provenance of the input data for making the tradeoff; the distribution of provenance is representative of production traffic since the requests in the duplication pipeline are a random sampling of that traffic. DQBarge determines whether the resulting latency distribution varies as a result of the input provenance; if so, it generates separate models for each provenance category. However, in the systems we study in Section 4, provenance does not have a statistically significant effect on performance (though it does significantly affect quality).

Quality models capture how the fidelity of the final response is affected by data-quality tradeoffs during request processing. To generate a quality model, DQBarge sends each request to two duplication pipelines. The first pipeline makes no tradeoffs, and so produces a full-fidelity response. The second pipeline makes a specified tradeoff, and so produces a potentially lower-fidelity response. DQBarge measures the quality impact of the tradeoff by comparing the two responses and applying a service-specific quality ranking specified by the developer. For example, if the output of the request is a ranked list of Web pages, then a service-specific quality metric might be the distance between where pages appear in the two rankings.

DQBarge next learns a model of how provenance affects request quality. As described in the previous section, input data objects to the component making the tradeoff are annotated with provenance in the form of typed key-value pairs. These pairs are the features in the quality model. DQBarge generates observations by making tradeoffs for objects with different provenance; e.g., systematically using default values for different types of objects. DQBarge uses multidimension linear regression to model the importance of each provenance feature in the quality of the request result. For example, if a data-quality tradeoff omits values from an aggregation, then omitting values from one data source may have less impact than omitting values from a different source.

Provenance can substantially reduce the number of observations needed to generate a quality model. Recall that all RPC data objects are annotated with provenance; thus, the objects in the final request result have provenance data. In many cases, the provenance relationship is direct; an output object depends only on a specific input provenance. In such cases, we can infer that the effect of a data-quality tradeoff would be to omit the specified output object, replace it with a default value, etc. Thus, given a specific output annotated with provenance, we can infer what the quality would be if further tradeoffs were made (e.g., a specific set of provenance features were used to omit objects from an aggregation). In such cases, the processing of one request can generate many data points for the quality model. If the provenance relationship is not direct, DQBarge generates these data points by sampling more requests and making different tradeoffs.

3.3 Using the models

DQBarge uses its performance and quality models to make better, more proactive data-quality tradeoffs. System operators specify a high-level goal such as maximizing quality given a latency cap on request processing. Components call functions such as `makeAggregationTradeoff` at each potential tradeoff point during request processing; DQBarge returns a decision as to whether a tradeoff should be made and, if appropriate, what fidelity should be employed (e.g., which data source to use or which values to leave out of an aggregation). Services provide a reference to the performance and quality models, as well as a list of load metrics (identified by key) and identifiers for objects with provenance. The service then implements the tradeoff decision proactively; i.e., it makes the tradeoff immediately. This design does not preclude reactive tradeoffs. An unexpectedly delayed response may still lead to a timeout and result in a data-quality tradeoff.

DQBarge currently supports three high-level goals: maximizing quality subject to a latency constraint, maximizing quality using slack execution time available during request processing, and maximizing utility as a function of quality and performance. These goals are useful for mitigating load spikes, efficiently using spare resources, and implementing dynamic capacity planning, respectively. We next describe these three goals.

3.3.1 Load Spikes

Services are provisioned to handle peak request loads. However, changes in usage or traffic are unpredictable; e.g., the launch of a new feature may introduce additional traffic. Thus, systems are designed to handle unexpected load spikes; the reactive data-quality tradeoffs we saw in Section 2 are one such mechanism. DQBarge improves on existing practice by letting an operator specify a maximum latency for a request or a component of request processing. It maximizes quality subject to this constraint by making data-quality tradeoffs.

At each tradeoff site, there may be many potential tradeoffs that can be made (e.g., sets of values with different provenance may be left out of an aggregation or distinct alternate data stores may be queried). DQBarge orders possible tradeoffs by “bang for the buck” and greedily selects tradeoffs until the latency goal is reached. It ranks each potential tradeoff by the ratio of the projected improvement in latency (given by the performance model) to the decrease in request fidelity (given by the quality model). The independent parameters of the models are the current system load and the provenance of the input data. DQBarge selects tradeoffs in descending order of this ratio until the performance model predicts that the latency limit will be met.

3.3.2 Utilizing spare resources

DQBarge obtains a prediction of which components are on the critical path and which components have slack available from the Mystery Machine [11]. If a component has slack, DQBarge can make tradeoffs that improve quality without negatively impacting the end-to-end request latency observed by the user. Similar to the previous scenario, DQBarge calculates the ratio of quality improvement to latency decrease for each potential tradeoff (the difference is that this goal involves improving quality rather than performance). It greedily selects tradeoffs according to this order until the additional latency would exceed the projected slack time.

3.3.3 Dynamic capacity planning

DQBarge allows operators to specify the utility (e.g., the dollar value) of reducing latency and improving quality. It then selects the tradeoffs that improve utility until no more such tradeoffs are available. DQBarge also allows operators to specify the impact of adding or removing resources (e.g., compute nodes) as a utility function parameter. DQBarge compares the value of the maximum utility function with more and less resources and generates a callback if adding or removing resources would improve the current utility. Such callbacks allow dynamic re-provisioning. Since DQBarge uses multidimension linear regression, it will not model significantly non-linear relationships in quality or performance; more sophisticated learning methods could be used in such cases.

3.4 Logging data-quality decisions

DQBarge optionally logs all data-quality decisions and includes them in the provenance of the request data objects. The information logged includes the software component, the point in the execution where a tradeoff decision was made, and the specific decision that was made (e.g., which values were left out of an aggregation). To reduce the amount of data that is logged, only instances where a tradeoff was made are recorded. Timeouts and error return codes are also logged if they result in a reactive data-quality tradeoff. This information helps system administrators and developers understand how low-level data-quality tradeoffs are affecting the performance and quality of production request processing.

3.5 Discussion

DQBarge does not guarantee an optimal solution since it employs greedy algorithms to search through potential tradeoffs. However, an optimal solution is likely unnecessary given the inevitable noise that arises from predicting traffic and from errors in modeling. For the last use case, DQBarge assumes that developers can quantify the impact of changes to service response times, quality, and the utilization of additional resources in order to set appropriate goals. DQBarge also assumes that tradeoffs are independent, since calculating models over joint distributions would be difficult. Finally, because DQBarge compares quality across different executions of the same request with different tradeoffs, it assumes that request processing is mostly deterministic.

Using DQBarge requires a reasonably-detailed understanding of the service being modified. Developers must identify points in the code where data-quality tradeoffs should be made. They must specify what performance and quality metrics are important to their service. Finally, they must select which provenance values to track and specify how these values are propagated through black-box components. For both of the case studies in Section 4, a single developer who was initially unfamiliar with the service being modified was able to add all needed modifications, and these modifications comprised less than 450 lines of code in each case.

DQBarge works best for large-scale services. Although it generates models offline to reduce interference with production traffic, model generation does consume extra resources through duplication of request processing. For large Internet services like Facebook, the extra resource usage is a tiny percentage of that consumed by production traffic. However, for a small service that sees only a few requests per minute, the extra resources needed to generate the model may not be justified by the improvement in production traffic processing.

4 Case studies

We have implemented the main components of DQBarge in a portion of the Facebook request processing pipeline, and we have evaluated the results using Facebook production traffic. Our current Facebook implementation allows us to track provenance, generate performance and quality models and measure the efficacy of the data-quality tradeoffs available through these models. This implementation thus allows us to understand the feasibility and potential benefit of applying these ideas to current production code.

We have also implemented the complete DQBarge system in Sirius [15], an open-source personal digital assistant akin to Siri. Our Sirius implementation enables end-to-end evaluation of DQBarge, such as observing how data-quality tradeoffs can be used to react to traffic spikes and the availability of slack in the request pipeline.

4.1 Facebook

Our implementation of DQBarge at Facebook focuses on a page ranking service, which we will call *Ranker* in this paper. When a user loads the Facebook home page, *Ranker* uses various parameters of the request, such as the identity of the requester, to generate a ranked list of page recommendations. *Ranker* first generates candidate recommendations. It has a flexible architecture that allows the creation and use of multiple candidate generators; each generator is a specific algorithm for identifying possible recommendations. At the time of our study, there were over 30 generators that collectively produced hundreds of possible recommendations for each request.

Ranker retrieves feature vectors for each candidate from *Laser*, the key-value store we studied in Section 2. *Ranker* is a service that makes reactive data-quality tradeoffs. If an error or timeout occurs when retrieving features, *Ranker* omits the candidate(s) associated with those features from the aggregation of candidates and features considered by the rest of the *Ranker* pipeline.

Ranker uses the features to calculate a score for each candidate. The algorithm for calculating the score was opaque to us (it is based on a machine learning model regenerated daily). It then orders candidate by score and returns the top N candidates.

DQBarge leverages existing tracing and monitoring infrastructure at Facebook. It uses a production version of the Mystery Machine tracing and performance analysis infrastructure [11]. This tool discovers and reports performance characteristics of the processing of Facebook requests, including which components are on the critical path. From this data, we can calculate the slack available for each component of request processing; prior results have shown that, given an observation of past requests by the same user, slack for future requests can be predicted with high accuracy. Existing Facebook systems monitor load at each component in the pipeline.

DQBarge annotates data passed along the pipeline with provenance. The data object for each candidate is annotated with the generator that produced the data.

Similarly, features and other data retrieved for each candidate are associated with their data source.

We implemented meters at the end of the *Ranker* pipeline that measure the latency and quality of the final response. To measure quality, we compare the difference in ranking of the top N pages returned from the full-quality response (with no data-quality tradeoffs made) and the lower-fidelity response (that includes some tradeoffs). For example, if the highest-ranked page in the lower-fidelity response is the third-ranked page in the full-quality response, the *quality drop* is two.

4.2 Sirius

We also applied DQBarge to Sirius [15], an open-source personal assistant similar to Apple's Siri or Google Now. Sirius answers fact-based questions based on a set of configurable data sources. The default source is an indexed Wikipedia database; an operator may add other sources such as online search engines.

Sirius generates several queries from a question; each query represents a unique method of parsing the question. For each query, it generates a list of documents that are relevant to answering the query. Each document is passed through a natural language processing pipeline to derive possible answers. Sirius assigns each answer a numerical score and returns the top-ranked answer.

Data-quality tradeoffs in Sirius occur when aggregating values from multiple sub-service queries. Our DQBarge implementation makes these tradeoffs proactively by using quality and performance models to decide which documents to leave out of the aggregation when the system is under load.

Initially, Sirius did not have request tracing or load monitoring infrastructure. We therefore added the ability to trace requests and predict slack by adding the Mystery Machine to Sirius. For load, we added counters at each pipeline stage to measure request rates. Additionally, we track the CPU load and memory usage of the entire service. The performance data, predicted slack, and load information are all propagated by DQBarge as each request flows through the Sirius pipeline.

In each stage of the Sirius pipeline, provenance is propagated along with data objects. For example, when queries are formed from the original question, the algorithm used to generate the query is associated with the query object. Sirius provenance also includes the data used to generate the list of candidate documents.

Since Sirius did not have a request duplication mechanism, we added the ability to sample requests and send the same request through multiple instances of the Sirius pipeline. User requests are read-only with respect to Sirius data stores, so we did not have to isolate any modifications to service state from duplicated requests.

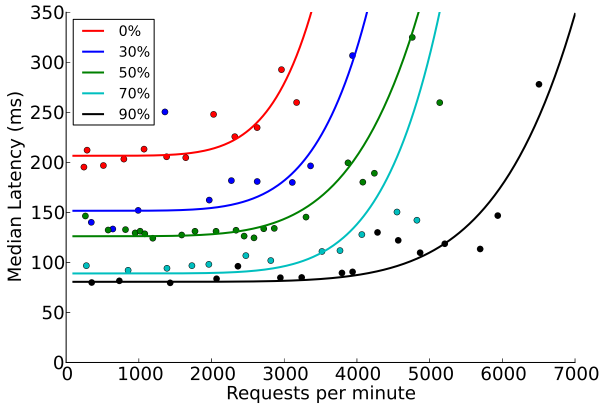


Figure 2: *Ranker* performance model This graph shows the effect of varying the frequency of data-quality tradeoffs on *Ranker* request latency. We varied the request rate by sampling different percentages of live production traffic at Facebook.

5 Evaluation

Our evaluation answers the following questions:

- Do data-quality tradeoffs improve performance?
- How much does provenance improve tradeoffs?
- How much does proactivity improve tradeoffs?
- How well does DQBarge meet end-to-end performance and quality goals?

5.1 Experimental setup

For *Ranker*, we perform our evaluation on Facebook servers using live Facebook traffic by sampling and duplicating *Ranker* requests. Our entire implementation uses duplicate pipelines, so as to not affect the results returned to Facebook users. Each pipeline duplicates traffic to a single isolated front-end server that is identical to those used in production. The duplicate pipelines share services from production back-end servers, e.g., those hosting key-value stores, but they are a small percentage of the total load seen by such servers. We change the load within a pipeline by sampling a larger or smaller number of *Ranker* requests and redirecting the sampled requests to a single front-end server for the pipeline.

For Sirius, we evaluated our end-to-end implementation of DQBarge on 16-core 3.1 GHz Xeon servers with 96 GB of memory. We send Sirius questions sampled from an archive from previous TREC conferences [32].

5.2 Performance benefits

We first measure the effect of data-quality tradeoffs on throughput and latency by generating performance models for *Ranker* and Sirius; Section 5.3 considers the effect of these tradeoffs on quality. DQBarge performs a full parameter sweep through the dimensions of request rate, tradeoff frequency, and provenance of the data being considered for each tradeoff, sampling at regular intervals. For brevity, we report a portion of these results.

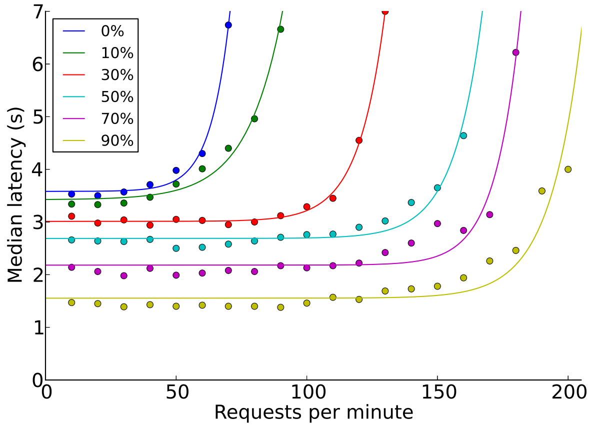


Figure 3: Sirius performance model. This graph shows the effect of varying the frequency of data-quality tradeoffs on Sirius request latency. Each curve shows a different tradeoff rate.

We show the median response time calculated over the sampling period at a specified request rate and tradeoff rate. For Sirius, 900 requests were sent over the sampling period. Median response time is shown because it is used for the remainder of the evaluation.

5.2.1 *Ranker*

Figure 2 shows the latency-response curve for *Ranker* when DQBarge varies the incoming request rate. Each curve shows the best fit for samples taken at a different *tradeoff rate*, which we define to be the object-level frequency at which data tradeoffs are actually made. When making tradeoffs, *Ranker* omits objects from aggregations; thus, to achieve a target tradeoff rate of x% during model generation, DQBarge will instruct *Ranker* to drop x% of the specific candidates. At a tradeoff rate of 0%, no candidates are dropped.

These results show that data-quality tradeoffs substantially improve *Ranker* latency at low loads (less than 2500 requests/minute); e.g., at a 30% tradeoff rate, median latency decreases by 28% and latency of requests in the 99th percentile decreases by 30%. Prior work has shown that server slack at Facebook is predictable on a per-request basis [11]. Thus, *Ranker* could make more tradeoffs to reduce end-to-end response time when *Ranker* is on the critical path of request processing, yet it could still provide full-fidelity responses when it has slack time for further processing.

Data-quality tradeoffs also improve scalability under load. Taking 250 ms as a reasonable knee in the latency-response curve, *Ranker* can process approximately 2500 requests per minute without making tradeoffs, but it can handle 4300 requests per minute when the tradeoff rate is 50% (a 72% increase). This allows *Ranker* to run at a lower fidelity during a load spike.

DQBarge found that the provenance of the data values selected for tradeoffs does not significantly affect performance. In other words, while the number of tradeoffs made has the effect shown in Figure 2, the specific candidates that are proactively omitted from an aggregation do not matter. Thus, we only show the effect of the request rate and tradeoff rate.

5.2.2 Sirius

Figure 3 shows results for Sirius. Like *Ranker*, the provenance of the data items selected for tradeoffs did not affect performance, so we show latency-response curves that vary both request rate and tradeoff rate.

The results for Sirius are similar to those for *Ranker*. A tradeoff rate of 50% reduces median end-to-end request latency by 26% and the latency of requests in the 99th percentile by 38%. Under load, a 50% tradeoff rate increases Sirius throughput by approximately 200%.

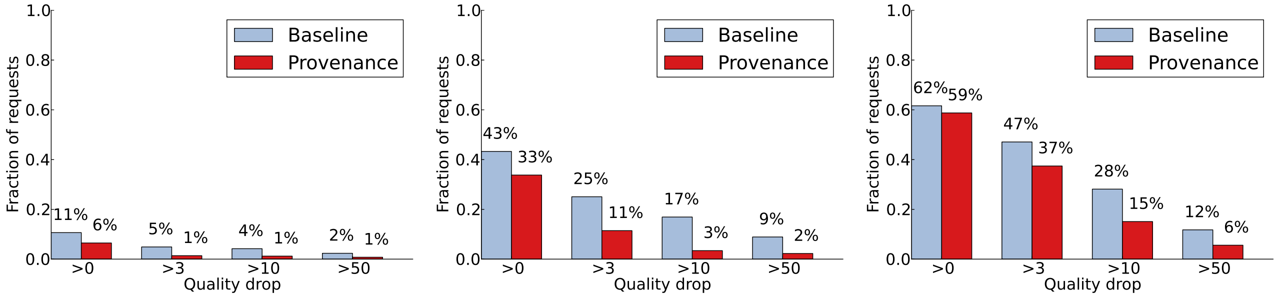
5.3 Effect of provenance

We next consider how much provenance improves the tradeoffs made by DQBarge. We consider a baseline quality model that does not take into account any provenance; e.g., given a target tradeoff rate, it randomly omits data values from an aggregation. This is essentially the policy in existing systems like *Ranker* and Sirius because there is no inherent order in requests from lower-level services to data stores; thus, timeouts affect a random sampling of the values returned. In contrast, DQBarge uses its quality model to select which values to omit, with the objective of choosing those that affect the final output the least.

5.3.1 *Ranker*

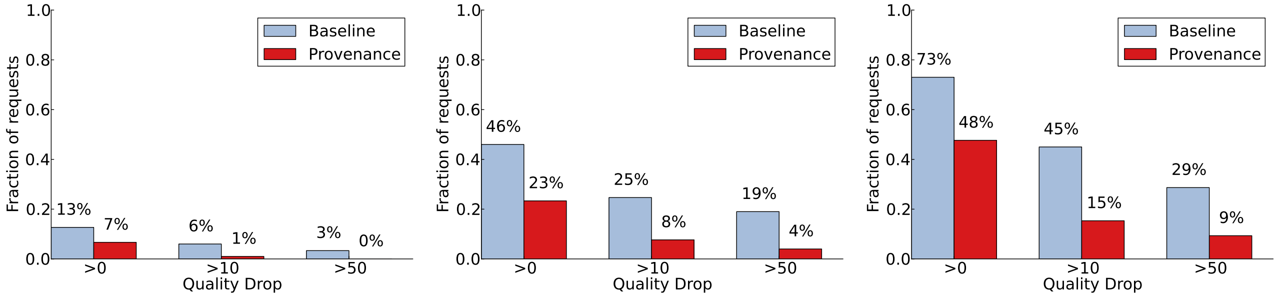
We first used DQBarge to sample production traffic at Facebook and construct a quality model for *Ranker*. DQBarge determined that, by far, the most important provenance parameter affecting quality is the generator used to produce a candidate. For example, one particular generator produces approximately 17% of the top-ranked pages but only 1% of the candidates. Another generator produces only 1% of the top-ranked pages but accounts for 3% of the candidates.

Figure 4 compares the quality of request results for DQBarge with a baseline that makes tradeoffs without using provenance. We sample live Facebook traffic, so the requests in this experiment are different from those used to generate the quality model. We vary the tradeoff rate and measure the *quality drop* of the top ranked page; this is the difference between where the page appears in the request that makes a data-quality tradeoff and where it would appear if no data-quality tradeoffs were made. The ideal quality drop is zero. While Sirius returns a single result, *Ranker* may return up to 3 results. We examined quality drops for the second and third *Ranker* results and found that they are similar to that of the top-ranked result; thus, we only show the top-ranked result for both services.



(a) Tradeoff rate 10% **(b)** Tradeoff rate 50% **(c)** Tradeoff rate 80%

Figure 4: Impact of provenance on *Ranker* quality. We compare response quality using provenance with a baseline that does not consider provenance. Each graph shows the quality drop of the top ranked page, which is the difference between where it appears in the *Ranker* rankings with and without data-quality tradeoffs. A quality drop of 0 is ideal.



(a) Tradeoff rate 10% **(b)** Tradeoff rate 50% **(c)** Tradeoff rate 80%

Figure 5: Impact of provenance on Sirius quality. We compare response quality using provenance with a baseline that does not consider provenance. Each graph shows the quality drop of the Sirius answer, which is the difference between where it appears in the Sirius rankings with and without data-quality tradeoffs. A quality drop of 0 is ideal.

As shown in Figure 4a, at a low tradeoff rate of 10%, using provenance reduces the percentage of requests that experience any quality drop at all from 11% to 6%. With provenance, only 1% of requests experienced a quality drop of more than three, compared to 5% without provenance. Figure 4b shows a higher tradeoff rate of 50%. Using provenance decreases the percentage of requests that experience any quality drop at all from 43% to 33%. Only 3% of requests experienced a quality drop of 10 or more, compared to a baseline result of 17%. Figure 4c compares quality at a high tradeoff rate of 80%. Use of provenance still provides a modest benefit: 59% of requests experience a quality drop, compared to 62% for the baseline. Further, with provenance, the quality drop is 10 or more for only 15% of requests compared with 28% for the baseline.

5.3.2 Sirius

For Sirius, we used k-fold cross validation to separate our benchmark set of questions into training and test data. The training data was used to generate a quality model based on provenance features, which included the language parsing algorithm used, the number of occurrences of key words derived from the question, the length of the data source document considered, and a weighted score relating the query words to the source document.

Figure 5 compares the quality drop for the result returned by Sirius for DQBarge using provenance with a baseline that does not use provenance. As shown in Figure 5a, at a tradeoff rate of 10%, provenance decreases the percentage of requests that see any quality drop at all from 13% to 7%. Only 1% of requests see a quality drop of 10 or more using provenance, compared to 6% for the basline. Figure 5b shows that, for a higher tradeoff rate of 50%, provenance decreases the percentage of requests that see any quality drop from 46% to 23%. Further, only 8% of requests see a quality drop of 10 or more using provenance, compared to 25% for the baseline. Figure 5c shows a tradeoff rate of 80%; provenance decreases the percentage of requests that see any quality drop from 73% to 48%.

5.4 Effect of proactivity

We next examine how proactivity affects data-quality tradeoffs. In this experiment, we send requests to Sirius at a high rate of 120 requests per minute. Without DQBarge, this rate occasionally triggers a 1.5 second timeout for retrieving documents, causing some docu-

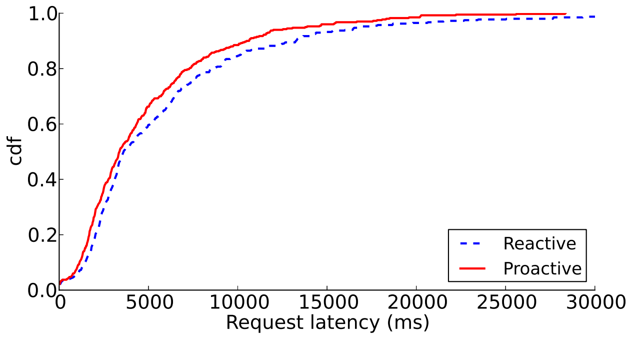


Figure 6: Performance of reactive tradeoffs. This graph compares the distribution of request latencies for Sirius when tradeoffs are made reactively via timeouts and when they are made proactively via DQBarge.

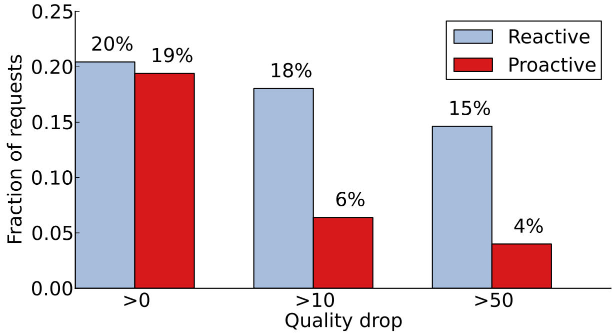


Figure 7: This graph shows that using proactive tradeoffs at a tradeoff rate of 40% can achieve higher quality tradeoffs than using reactive tradeoffs with a timeout of 1.5 s in Sirius.

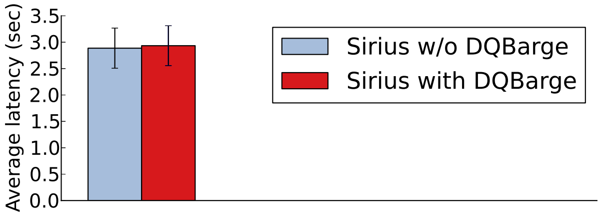


Figure 8: DQBarge Overhead. This graph compares time to process 140 Sirius questions with and without DQBarge; error bars are 95% confidence intervals.

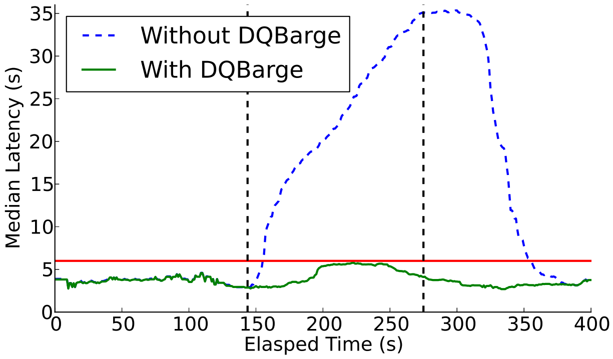


Figure 9: Response to a load spike. DQBarge makes data-quality tradeoffs to meet a median latency goal of 6 seconds.

ments to be left out of the aggregation. These tradeoffs are reactive in that they occur only after a timeout expires. In contrast, with DQBarge, tradeoffs are made proactively at a rate of 40%, a value selected to meet the latency goal of not exceeding the mean latency without DQBarge.

Figure 6 shows request latency as a CDF for both the reactive and proactive methods of making data-quality tradeoffs and Figure 7 shows the quality drop for both methods. The results show that DQBarge proactivity simultaneously improves *both* performance and quality when making tradeoffs. Comparing the two distributions in Figure 6 shows that DQBarge improves performance across the board; e.g., the median request latency is 3.4 seconds for proactive tradeoffs and 3.6 seconds for reactive tradeoffs. For quality, DQBarge proactivity slightly decreases the number of requests that have any quality drop from 20% to 19%. More significantly, it reduces the number of requests that have a quality drop of more than 10 from 18% to 6%.

Under high loads, reactive tradeoffs hurt performance because they waste resources (e.g., trying to retrieve documents that are not used in the aggregation). Further, their impact on quality is greater than with DQBarge because timeouts affect a random sampling of the values returned, whereas proactive tradeoffs omit retrieving those documents that are least likely to impact the reply.

5.5 Overhead

We measured the online overhead of DQBarge by comparing the mean latency of a set of 140 Sirius requests with and without DQBarge. Figure 8 shows that DQBarge added a 1.6% latency overhead; the difference is within the experimental error of the measurements.

DQBarge incurs additional space overhead in message payloads for propagating load metrics, critical path and slack predictions, and provenance features. For Sirius, DQBarge adds up to 176 bytes per request for data such as load metrics and slack predictions. Tracking provenance appends an extra 32 bytes per object; on average, this added 14% more bytes per provenance-annotated object.

5.6 End-to-end case studies

We next evaluate DQBarge with three end-to-end case studies on our Sirius testbed.

5.6.1 Load spikes

In this scenario, we introduce a load spike to see if DQBarge can maintain end-to-end latency and throughput goals by making data-quality tradeoffs. We set a target median response rate of 6 seconds. Normally, Sirius receives 50 requests/minute, but it experiences a two-minute load spike of 150 requests/minute in the

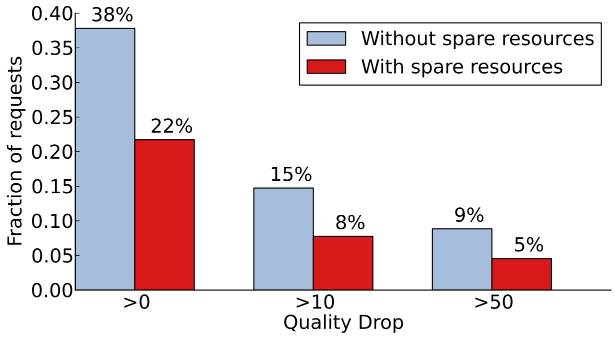


Figure 10: Quality improvement using spare resources. DQBarge uses slack in request pipeline stages to improve response quality.

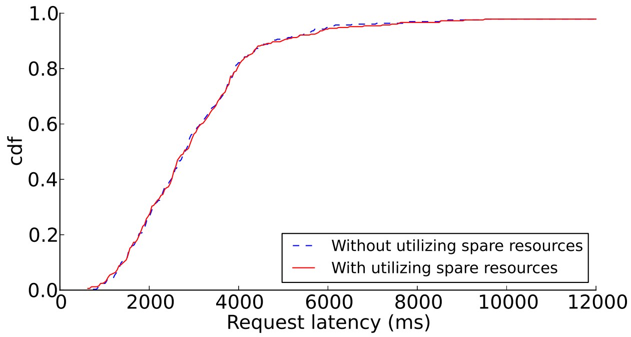


Figure 11: Performance impact of using spare resources. When DQBarge uses slack in request pipeline stages, it does not impact end-to-end latency.

middle of the experiment. Figure 9 shows that without DQBarge, the end-to-end latency increases significantly due to the load spike. The median latency within the load spike region averages 25.2 seconds across 5 trials.

In comparison, DQBarge keeps median request latency below the 6 second goal throughout the experiment. Across 5 runs, the median end-to-end latency during the spike region is 5.4 seconds. In order to meet the desired latency goal, DQBarge generally selects a tradeoff rate of 50%, resulting in a mean quality drop of 6.7.

5.6.2 Utilizing spare resources

Next, DQBarge tries to use spare capacity and slack in the request processing pipeline to increase quality without affecting end-to-end latency. Sirius is configured to use both its default Wikipedia database and the Bing Search API [6] to answer queries. Each source has a separate pipeline that executes in parallel before results from all sources are compared at the end. The Bing pipeline tends to take longer than the default pipeline, so slack typically exists in the default pipeline stages.

As described in Section 4.2, DQBarge predicts the critical path for each request and the slack for pipeline stages not on the critical path. If DQBarge predicts there is slack available for a processing pipeline, it reduces the tradeoff frequency to increase quality until the predicted added latency would exceed the predicted slack. To give DQBarge room to increase quality, we set the default tradeoff rate to 50% for this experiment; note that this simply represents a specific choice between quality and latency made by the operator of the system.

Figure 10 shows that DQBarge increases quality for this experiment by using spare resources; the percentage of requests that exprience any quality drop decreases from 38% to 22% (as compared to a full-fidelity response with no data-quality tradeoffs). Figure 11 shows a CDF of request response times; because the extra processing occurs off the critical path, the end-to-end request latency is unchanged when DQBarge attempts to employ only spare resources to increase quality.

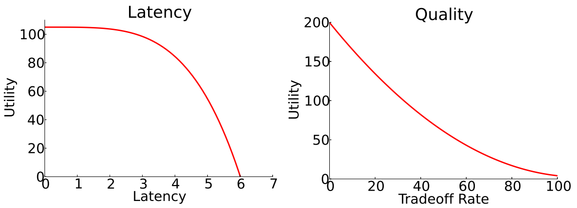


Figure 12: Utility parameters for dynamic capacity planning. These values are added together to calculate final utility.

5.6.3 Dynamic capacity planning

Finally, we show how DQBarge can be used in dynamic capacity planning. We specify a utility function that provides a dollar value for reducing latency, improving quality, and provisioning additional servers. The utility of latency and quality are shown in Figure 12. DQBarge makes data-quality tradeoffs that maximize the utility function at the incoming request rate.

In this scenario, we examine the benefit of using DQBarge to decide when to provision additional resources. We compare DQBarge with dynamic capacity planning against DQBarge without dynamic capacity planning. Figure 13 shows the total utility of the system over time. When the request rate increases to 160 requests per minute, DQBarge reports that provisioning another server would provide a net positive utility. Using this server increases utility by an average of 58% compared to a system without dynamic capacity planning.

DQBarge is also able to reduce the number of servers in use. Figure 13 shows that when the request rate subsides, DQBarge reports that taking away a server maximizes utility. In other words, the request rate is low enough that using only one server maximizes utility.

6 Related work

Although there is an extremely rich history of quality-of-service tradeoffs [7, 25, 29] and approximate computing [4, 8, 19, 18, 28, 30] in software systems, our work focuses specifically on using the causal propagation of request information and data provenance to make better

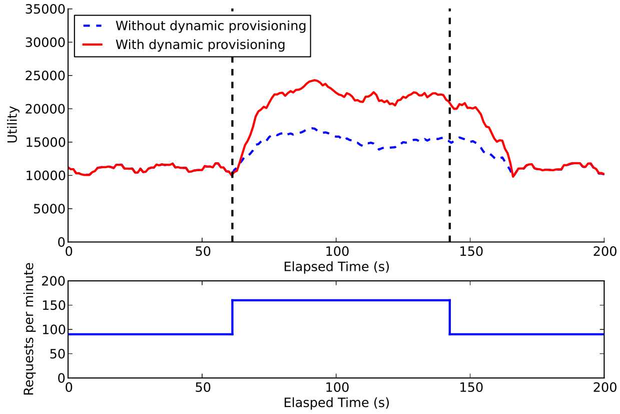


Figure 13: Benefit of dynamic capacity planning. With dynamic capacity planning, DQBarge improves utility by provisioning an additional server. When it is no longer needed, it removes the additional server.

data-quality tradeoffs in low-level software components. Our study revealed the need for such an approach: existing Facebook services make mostly reactive tradeoffs that are suboptimal due to limited information. Our evaluation of DQBarge showed that causal propagation can substantially improve both request performance and response quality.

Many systems have used causal propagation of information through distributed systems to trace related events [5, 9, 10, 13, 23, 26, 27, 31]. For example, Pivot Tracing [23] propagates generic key-value metadata, called baggage, along the causal path of request processing. DQBarge uses a similar approach to propagate specific data such as provenance, critical path predictions, and load metrics.

DQBarge focuses on data-quality tradeoffs in Internet service pipelines. Approximate Query Processing systems trade accuracy for performance during analytic queries over large data sets [1, 2, 3, 17, 22]. These systems use different methods to sample data and return a representative answer within a time bound. BlinkDB [2] uses an error-latency profile to make tradeoffs during query processing. Similarly, ApproxHadoop [14] uses input data sampling, task dropping, and user-defined approximation to sample the number of inputs and bound errors introduced from approximation. These techniques are similar to DQBarge's performance and quality models, and DQBarge could potentially leverage quality data from ApproxHadoop in lieu of generating its own model.

LazyBase [12] is a NoSQL database that supports trading off data freshness for performance in data analytic queries. It is able to provide faster read queries to stale-but-consistent versions of the data by omitting newer updates. It batches and pipelines updates so that intermediate values of data freshness can be queried. Similar to how LazyBase uses data freshness to make a tradeoff, DQBarge uses its quality model to determine the best tradeoff that minimizes the effect on the quality.

Some Internet services have been adapted to provide partial responses after a latency deadline [16, 21, 22]. They rely on timeouts to make tradeoffs, whereas the tradeoffs DQBarge makes are proactive. PowerDial [19] adds knobs to server applications to trade performance for energy. These systems do not employ provenance to make better tradeoffs.

7 Conclusion

In this paper, we showed that data-quality tradeoffs are prevalent in Internet service pipelines through a survey of existing software at Facebook. We found that such tradeoffs are often suboptimal because they are reactive and because they fail to consider global information. DQBarge enables better tradeoffs by propagating data along the causal path of request processing and generating models of performance and quality for potential tradeoffs. Our evaluation shows that this improves responses to load spikes, utilization of spare resources, and dynamic capacity planning.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Rebecca Isaacs, for their thoughtful comments. We also thank Qi Hu and Jason Brewer for their help with the Facebook infrastructure. This work has been supported by the National Science Foundation under grant CNS-1421441. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. The aqua approximate query answering system. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, 1999.

[2] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013.

[3] Brian Babcock, Surajit Chaudhuri, and Gautam Das. Dynamic sample selection for approximate query processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, 2003.

[4] Woongki Baek and Trishul M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation*, 2010.

[5] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 259–272, San Francisco, CA, December 2004.

[6] `https://datamarket.azure.com/dataset/bing/search`.

[7] Josep M. Blanquer, Antoni Batchelli, Klaus Schauser, and Rich Wolski. Quorum: Flexible quality of service for internet services. Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation, 2005.

[8] Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C. Rinard. Proving acceptablility properties of relaxed nondeterministic approximate programs. In *Proceedings of the ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation*.

[9] Anupam Chanda, Alan L. Cox, and Willy Zwanepoel. Whodunit: Transactional profiling for multi-tier applications. In *Proceedings of the 2nd ACM European Conference on Computer Systems*, Lisboa, Portugal, March 2007.

[10] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic Internet services. In *Proceedings of the 32nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 595–604, Bethesda, MD, June 2002.

[11] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. The Mystery Machine: End-to-end performance analysis of large-scale internet services. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation*, October 2014.

[12] James Cipar, Greg Ganger, Kimberly Keeton, Charles B. Morrey III, Craig A.N. Soules, and Alistair Veitch. Lazybase: Trading freshness for performance in a scalable database. In *Proceedings of the 7th ACM European Conference on Computer Systems*.

[13] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*, pages 271–284, Cambridge, MA, April 2007.

[14] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D. Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, Istanbul, Turkey, March 2015.

[15] Johann Hauswald, Michael A. Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ron Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, and Jason Mars. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[16] Yuxiong He, Sameh Elnikety, James Larus, and Chenyu Yan. Zeta: Scheduling interactive services with partial execution. In *Proceedings of the Third ACM Symposium on Cloud Computing (SOCC '12)*, 2012.

[17] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, 1997.

[18] Henry Hoffmann. Jouleguard: Energy guarantees for approximate applications. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles*, 2015.

[19] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, California, March 2011.

[20] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1991.

[21] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. Speeding up distributed request-response

workflows. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM '13)*, 2013.

[22] Gautam Kumar, Ganesh Ananthanarayanan, Sylvia Ratnasamy, and Ion Stoica. Hold 'em or fold 'em? aggregation queries under performance variations. In *Proceedings of the 11th ACM European Conference on Computer Systems*, 2016.

[23] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles*, 2015.

[24] Justin Meza, Dmitri Perelman, Wonho Kim, Sonia Margulis, Daniel Peek, Kaushik Veeraraghavan, and Yee Jiun Song. Kraken: A framework for identifying and alleviating resource utilization bottlenecks in large scale web services. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation*, Savannah, GA, November 2016.

[25] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, J. Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 276–287, Saint-Malo, France, October 1997.

[26] Lenin Ravindranath, Jitendra Padjye, Sharad Agrawal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. AppInsight: Mobile app performance monitoring in the wild. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation*, Hollywood, CA, October 2012.

[27] Lenin Ravindranath, Jitendra Pahye, Ratul Mahajan, and Hari Balakrishnan. Timecard: Controlling user-perceived delays in server-based mobile applications. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, Farmington, PA, October 2013.

[28] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power consumption. In *Proceedings of the ACM SIGPLAN 2011 Conference on Programming Language Design and Implementation*, 2011.

[29] Kai Shen, Hong Tang, Tao Yang, and Lingkun Chu. Integrated resource management for cluster-based internet services. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, Massachusetts, December 2002.

[30] Stelios Sidiroglou, Sasa Misailovic, Henry Hoffmann, and Martin Ricard. Managing performance vs accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011.

[31] Benjamin H. Sigelman, Luiz Andr Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[32] `http://trec.nist.gov/`.

[33] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive – a warehousing solution over a map-reduce framework. In *35th International Conference on Very Large Data Bases (VLDB)*, Lyon, France, August 2009.