

# Data Purchase Advisor: Cost-Effective Data Acquisition for Shared Data Analysis

by

Martina Unutzer

Supervised by Magdalena Balazinska

A senior thesis submitted in partial fulfillment of  
the requirements for the degree of


Bachelor of Science  
With Departmental Honors

Computer Science & Engineering

University of Washington

June 2014

Presentation of work given on 6/3/14

Thesis and presentation approved by 

Date 6/3/14

# Data Purchase Advisor: Cost-Effective Data Acquisition for Shared Data Analysis

Martina Unutzer

Prasang Upadhyaya

## ABSTRACT

A data purchase advisor suggests data purchases to support a data-driven application. The advisor takes into account available sources and prices of data to purchase and the data requirements of the application in suggesting purchases which satisfy the application's requirements for data while minimizing purchase cost. We explore possible roles and uses for such an advisor, both as an offline tool and as an online service. This includes an examination of how data can be purchased online and possible formats to specify data quality and freshness metrics important to the data sellers and application. The data purchase advisor is essentially a decision-maker, and we explore several strategies for choosing data purchases when budget is limited. The advisor can prioritize data purchases with an application model that supports tiers of customers and priorities within the application. Finally, we examine approximate solutions in which we purchase data samples or noisy or approximate values at lower cost to find a balance between data quality and cost which is useful for the application.

## 1. INTRODUCTION

Data has always been a valuable commodity, and recently, it is increasingly available for purchase online. Online data purchase offers new opportunities for developers, researchers, and other analysts to create new analysis from an existing data set, often combining subsets of many different data sets. With online data sales, data can be bought at a more fine-grained level rather than a single purchase of the entire relation. Pricing can become increasingly sophisticated as we learn how to determine the value of smaller subsets of the original relation, and purchases can be automated and made real-time to support applications or online analysis.

Application developers are using this wide availability of data to build data-intensive applications which purchase data from several online sources and update it frequently to provide nearly real-time information and analysis to application users. Developers must purchase updated data smartly to

provide appropriate information while staying within a budget; a dramatic change in application use or a poor purchasing strategy can make an application either unusable from a user perspective (lack of data) or too costly for the developer (too many data purchases).

To help developers leverage the online data market and manage data purchases to support their application, we envision a data purchase advisor. The purchase advisor supports a data-driven application by accepting constraints about what data must be purchased and updated and outputting a cost-efficient purchase schedule. In this report, we make three contributions to the emerging definition of a purchase advisor: (1) We survey the existing data marketplaces in which a purchase advisor would operate. (2) We explore the role and architecture of the data purchase advisor and how to formalize the data requirements of an application. (3) We explore the impact of sharing and approximate data purchases and examine how a purchase advisor can use these relaxed data requirements to purchase data for an application at a lower cost.

## 2. SURVEY OF ONLINE DATA MARKET

This section presents a survey of the current data market. Information is presented from the perspective of an example application which performs sentiment analysis on tweets from the Twitter API. The application must choose from several datasources based on price, availability, completeness of the dataset, and latency.

### 2.1 Survey of data sellers

Data is sold in a variety of formats online [1, 3]. Companies can sell their own data directly, often by providing an API for developers. An API typically comes with usage restrictions. It may limit the frequency of queries or the amount downloaded in a window of time, particularly for a smaller free tier of usage. A company may also limit what can be done with the data it provides; for example, Yelp makes a limited amount of its data available to application developers for free, but stipulates that if Yelp data is used in an application, it must be clearly credited to Yelp and cannot be aggregated or combined with ratings from other sites[2].

Larger data markets such as the Microsoft Azure Marketplace offer a single place for developers to purchase datasets from a variety of companies and a unified method of getting the data[6]. Again, limited amounts of data are sometimes available for free, but large quantities must be pur-

chased. The Azure Marketplace provides the purchased data through an API to query; some other data marketplaces also provide the data for bulk download. APIs which are queried directly for small amounts of data allow finely grained pricing based on the amount of data requested; for analytics which require a large volume of data bulk purchases may be cheaper and more useful than querying for and purchasing specific information from a dataset.

The same data set can also be sold under different purchase models. Twitter data is a good example of this, as it is updated constantly and covers a wide range of topics (based on keywords present in the tweets). Twitter data is available in small quantities directly through Twitter APIs, which allow querying for specific tweets. For larger-scale purchases required to support a larger application, Twitter uses data reseller Gnip[5]. Gnip processes the full Twitter firehose of all public tweets and packages it into different products. Data can be purchased as a subscription to a real-time stream of tweets or a set of saved past tweets that match some criteria. In each case, customers can purchase a broad sample, such as 10% of the full Twitter firehose, or a more specific set of tweets, such as all tweets involving a particular username or originating from a certain geographic area. Each of these purchase options is fitting for a different application. When choosing what data to purchase for an application which uses Twitter data, a developer or data purchase advisor must evaluate each option to determine how much data the application would have to purchase (for example, is it worth purchasing a 10% sample of the full firehose, or specifying only a few topics?) and at what price.

## 2.2 Service-level agreements

When data is purchased as access to an API or a real-time stream, some data sellers provide SLAs which guarantee some minimum percentage of uptime and maximum percentage of requests resulting in error. If the data seller does not meet their own guarantees, they may compensate the customer with credits or refund. APIs also include requirements on the customer side describing what the purchaser may do with the data, such as the Yelp restriction on combining ratings with other sources. Developers must take SLAs into account when the same data is available for multiple sources. An advisor should accordingly be able to consider the availability and latency guaranteed by a data seller, and the developer's needs and willing to pay for availability and low latency in any purchase of streaming real-time data.

## 3. ROLE OF PURCHASE ADVISOR

A simple purchase advisor would be an analytic tool run by the developer. As input, it accepts an application's data use requirements and possible data purchase sources. These demands can be formalized using metrics such as latency, statistical accuracy, completeness or representative sample guarantees, and are described in more detail in the following section. The advisor would then output a suggested set of purchases from these sources which meets, to the extent possible, the data requirement constraints specified by the application.

### 3.1 Static purchase advisor

In the case where data must be purchased only once or where data can be updated and application use (customer demands for data) will occur in a predictable, cyclic fashion, a purchase advisor which runs once may be sufficient.

#### 3.1.1 Local prioritization: balancing data demands within application logic

To provide acceptable service (enough recent data) to customers when budget is limited, the advisor must take into account different priorities within the application. For example, consider a mobile application which uses data from different sources to provide information about restaurants near the application user. An outdated menu entry or even a missing restaurant is less likely to cause a problem for users than outdated hours of operation when referenced by a customer on their way to a restaurant. The application developer must specify which information is most important to keep updated, and the purchase advisor is responsible for prioritizing frequent update purchases for these attributes when there is not enough budget to purchase all attributes used in the application.

#### 3.1.2 Global prioritization: balancing competing customer demands

An application may also assign different priorities to different customers. Many applications use a tiered subscription model, with some customers using a limited-feature free version of the application and other customers paying for a premium version of the same application. These different groups of customers will typically require overlapping but not identical data. If providing complete and up-to-date information to premium customers is an application's priority (to maintain a satisfied paying customer base), the purchase advisor needs to know what data is required by premium customers and the extent to which it should prioritize these purchases given a limited budget for data purchases.

## 3.2 Dynamic and online advisor

While a static advisor may still be a beneficial tool for developers, the purchase advisor becomes much more useful if it is run as a service which runs online with the application and offers near real-time purchase suggestions based on current application use and data requirements and potentially changing data purchase options.

This online purchase advisor might require an analysis period, in which it tracks inputs and data use for a while, before it begins outputting purchase recommendations. This would allow the advisor to use repetition and patterns in data use or updates to make smarter predictions and learn over time. In a case where data purchases are relatively stable and consistent, the purchase advisor could even make purchases on behalf of the app, assuming the existence of an online data purchase mechanism that the advisor can make use of. If the purchase advisor assumed the role of purchaser as well, with the developer performing only periodic oversight and checks, the advisor would have to be given additional constraints. Strict budget constraints would be used to avoid disastrous overpurchasing, and on the other hand, constraints on minimum acceptable service for users

(in terms of availability of recent data) would keep service at a minimum. If these upper and lower limits conflict and the advisor is unable to choose a set of purchases which satisfy both, the developer must step in to revise the requirements or available purchases.

### 3.3 Devloper role using purchase advisor

The usefulness of the advisor depends largely on the accuracy of the inputs it is given. This requires that the application developer find effective functions to express the value of customers, the breakdown of required data by keyword, and the value of each keyword. For this reason, use of the advisor could be an iterative process in which the application developer tries different configurations of inputs to best communicate the requirements of the application.

Finally, all this information required by the purchase advisor must be somehow communicated from the application developer, who understands the purpose and priority of various data uses in the application, to the purchase advisor, which must assign a value to each relevant data purchase it could make. The developer must somehow translate the application logic into data requirements that the purchase advisor can understand. For any reasonably large application, inputting data requirements and potential purchase sources becomes a complicated and tedious task. A purchase advisor that could be widely used in practice will therefore require a standardized format for communicating data requirements and should eventually provide the ability to automate some of this work, such as enumerating and evaluating possible online purchase sources.

### 3.4 Output

Given constraints about the desired budget, queries, and data quality, the advisor will explore a large space of possible solutions. It should output only a few solutions to the application developer which differ significantly in terms of queries answered, or quality metrics satisfied[8]. For example, within a tier of purchase schedules for the same approximate cost, one purchase schedule may provide good recency on a small set of queries, while another may provide wide coverage of queries or larger data sets with better statistical significance, but with a higher probability of stale data.

## 4. FORMALIZING DATA PURCHASE REQUIREMENTS: THE DATA SLA

Right now, we do not know of a way to collect information about an application's data requirements and options for online data purchases in a standardized format from sellers and specify it to the advisor. The advisor will be more useful if it can cover a wide range of applications and data sources. Information about the data being purchased must be specified in a format that the purchase advisor can understand automatically. We suggest a data SLA (dSLA) which is similar to an SLA used for purchasing computing services online, but with a focus on the quality and information content of the data instead of computing performance.

The dSLA will require a set of standardized metrics which enable the advisor to compare sources and make decisions on what data the application developer should purchase auto-

matically. The question of which metrics will prove useful is still under exploration, but we envision metrics to cover the following dimensions along which we measure data quality.

### 4.1 Recency

For data which changes over time, such as traffic data or listings of open restaurants, having a recent version of the data is important to the application users and developer. This is one measure of data quality that we will focus on in the advisor. The advisor can approximate updates as periodic, and can output a schedule of advised purchases in a time window to match the periodic updates of the data.

Different data sources are updated with different frequencies. Updates may be on a regular schedule or may be irregular, and they can affect the entire database or only a single tuple. To understand how quickly the purchased data will become stale and inaccurate, the advisor must have information about how the data is likely to be updated. Some of this information can be specified by sellers easily (such as a daily update to weather data), but a potentially irregular update schedule may have to be learned by the advisor by purchasing small samples of data over a long period of time to build an approximate update model.

Possible ways to measure recency: [9]

1. **Binary:** The data is marked as current or not current depending on whether it matches the version in the seller's database.
2. **Time elapsed:** For data which is updated somewhat regularly and with a known update frequency, the time elapsed since purchase of the data can provide information about how out of date the purchased copy of the data is.
3. **Version number:** If updates are tracked by the source database, each update can be stamped with an increasing version number. The difference between the version number of the data in the source and the purchased copy indicates how many changes have been made to that data since purchase.

We believe measuring recency with version numbers will be useful. In our current concept of the advisor, we maintain the requirement that all data in the dataset maintained by the advisor must come from the same version of the database: the view maintained by the advisor is a snapshot that can be constructed from the original database at some point in time. Version numbers allow us to determine how many updates old a database is; depending on the application, this distance (number of missed updates) may be meaningful in telling us how inaccurate the view is compared to the current version of the database, or it may simply be a binary signal. Version numbers are also independent of update rate, allowing comparison between datasets with frequent updates and infrequent updates. However, the best measure of recency is dependent on the dataset, its update frequency and patterns, and how it is used by an application.

Recency metrics will also depend on whether data updates

are provided via a *push* or *pull* model. When purchasing a data set or sample online, the transaction is a 'pull' by the purchaser, who must request the data. Updates typically work in a similar fashion: it is up to the purchaser to request and purchase an updated copy of data purchased. Data sources could support queries for what version number of a specific data item is available. This information may be free, or it may be available only for a cost comparable to other queries.

Some subscription-based models may provide a push service, in which updates are provided when they are available instead of when they are requested. This better describes a streaming service such as a firehose of tweets that continually provides new information. These pushed updates may provide the new data, or they may simply be notifications that updated versions of the data are available for purchase if desired.

## 4.2 Sample Quality

In many cases, application developers can only purchase a small sample of a very large dataset. A small sample may be sufficient to provide good enough information to application users, but this depends on the sample that is provided by the data seller. A data quality specification (SLA) should include some statistical properties guaranteed about the sample, such as a maximum standard deviation or a guarantee that all groups present in a dataset will be represented in the sample. A data seller could guarantee a sample large enough to meet certain agreed upon thresholds, or the seller can use the full database to provide guarantees (with some certainty level) about the properties of the dataset given the sample size requested by the purchaser. Different statistics will be applicable to different data sets and use cases, but the format for specifying any of these properties over any data sample offered for purchase can be standardized.

## 4.3 Granularity

Quality metrics such as recency and statistical properties of the provided sample can be measured at different granularities, just as data can be purchased at the granularity of individual tuples, query responses, views, or full datasets. A data quality SLA must choose an appropriate and meaningful granularity at which to specify this information, depending on the nature of the data being sold.

## 5. PRIORITIZING CUSTOMERS WITHIN AN APPLICATION

We first considered a purchase advisor which purchases data as individual keywords (with a set price to purchase all the tuples corresponding to that keyword) from a single source. The purchase advisor must satisfy application/customer data requirements as much as possible while staying under a specified budget. In our somewhat simplified model, each customer belongs to either the free (low priority) or premium (higher priority) group of customers. To provide the requested information to each customer using the app, the application requires a set of keywords specific to that customer. Because all customers are using the same application, their requests are similar: each customer requests the same number of keywords, and each keyword is purchased independently from the same data source. However, not all customers request the same keywords: each customer

has slightly different requirements, and some keywords are more popular than other. This corresponds to an application which provides information based on some user-specific parameter, such as location. For example, keywords could be user locations in an application which searches for restaurants near the user. Every user could request restaurants from a different location, but there will be overlap (two nearby users requesting restaurants can use the same data) and some locations, such as major cities, will be requested more frequently than other locations.

While this case is clearly simplistic in its approach to data purchase, considering only a single homogeneous source from which all data can be purchased, it allows us to explore how a purchase advisor should prioritize customers with competing and overlapping data requests on a limited budget.

### 5.1 Offline greedy advisor implementation

We implemented a purchase advisor like the one described above for an offline case which takes a set of customer requests, a set of data available for purchase, and a budget and chooses a subset of the requested keywords to purchase. The advisor must define a purchase function which chooses which keyword to purchase next, given the remaining budget available and the demand for each keyword. For this basic offline advisor, purchases are output as a single computation as requested by the application developer using the advisor. Such an advisor could be periodically re-run by an application developer as the customer base and customer requests, or the availability of new data sources, changes.

### 5.2 Purchase criteria

The list of all requested keywords is ranked according to utility, where utility = value - cost of the keyword and the value of a keyword  $k$  is computed as  $\sum_{c \in \text{Requesters}(k)} \text{cust\_val}(c) * \text{cust\_key\_val}(c, k)$ , where  $\text{Requesters}(k)$  are all customers who request a keyword  $k$ ,  $\text{cust\_val}(c)$  is the value assigned to customer  $c$  by the application, and  $\text{cust\_key\_val}(c, k)$  is the value assigned to that keyword by the customer requesting it. The advisor tries to add each keyword to the purchased set in order of decreasing utility, skipping any keywords it cannot afford to buy with the budget remaining.

### 5.3 Data generation

Several sample datasets were generated to test and evaluate the advisor. These datasets each consist of 1000 customers, each with a customer value of either 0.8 or 0.2 to represent premium and free-subscription customers. There are twice as many free-subscription customers as premium customers in the sample data. Each customer requests the same number (20) of the available 100 keywords and assigns each keyword a uniform, then normalizes the values assigned to the keywords so each customer assigns a total value of 1.0 (in this case, a value of 0.5 per keyword). There are 30 data-sources, each of which sell some subset of the 100 keywords for a price randomly chosen in the range[0.1, 1.0].

In each dataset, the customer - keyword requests are determined by constructing a bipartite graph matching customer nodes to keyword nodes to satisfy the specified degree sequences for each group of nodes. The degrees of the customer nodes are all uniform, because each customer requests the same number of keywords. The degrees of the

keyword nodes, however, varies. In the first dataset used, the number of customers requesting each keyword is chosen randomly from the triangular distribution with minimum 1, maximum 1000, and mode 200 (number of total customer requests / number of keywords). In a second dataset, the number of customers requesting each keyword varies according to a power law distribution with exponent 2. This models a scenario in which a few keywords are interesting to most customers, and there is a long tail of keywords which are relevant to only a few customers.

#### 5.4 Baseline comparison algorithms

To compare the advisor to a simple decision process without an advisor, we implemented several simple comparison policies to run over the same sample data. These differ from the advisor in the algorithm they use to choose the next keyword to purchase.

- The *naive* policy randomly shuffles the complete list of customer requests, then proceeds down the list in order, purchasing whatever keywords it can afford until it runs out of budget. This is equivalent to randomly choosing which customer requests to satisfy, or, if requests are sent to an application over a period of time, satisfying all requests that arrive in the order of arrival until the budget is depleted. Because the list of requests is used, rather than the de-duplicated list of requested keywords, keywords requested by more customers are still more likely to be purchased.
- The *customer importance* policy is similar to the naive policy, but always selects requests from the higher-value customers first (prioritizes premium customers and purchases keywords requested by free customers only if there is enough budget left). This strategy does not differ dramatically from the naive strategy in this dataset because free and premium customers request keywords from the same relatively small set of keywords.
- The *keyword popularity* policy purchases as many keywords as it can afford ordered by the number of customers requesting the keyword. This strategy performs well when customer values and the values assigned to different keywords by the customers are close to uniform, but fails to account for the values assigned to different keywords.

#### 5.5 Performance of greedy algorithm

We tested the advisor and each of the baseline policies over the two sample datasets (uniform and power law keyword popularities) and evaluated the resulting purchase lists by computing the sum of the customer-request values for each purchased keyword, weighted by the value of each customer. The same measures were performed for the datasets with uniform and power-law keyword request distributions:

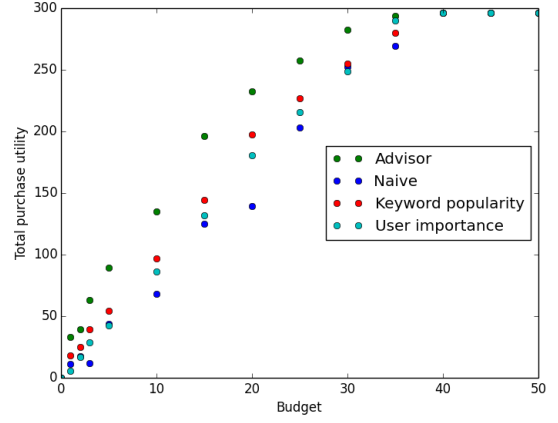


Figure 1: Performance of purchase policies on generated dataset with uniform keyword request distribution

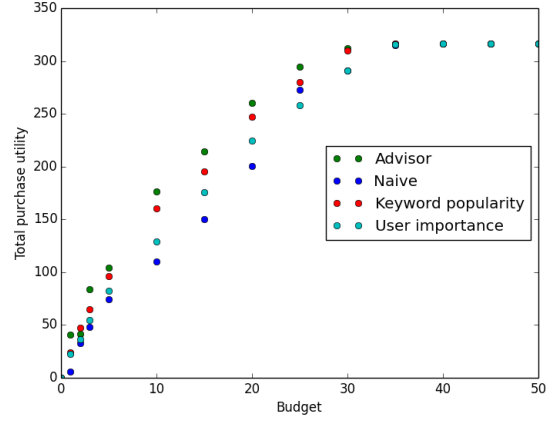
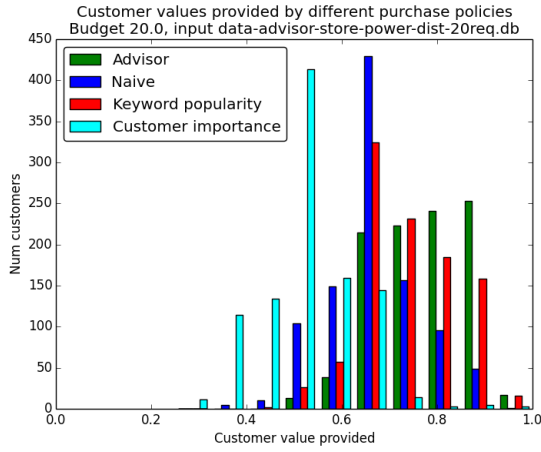


Figure 2: Performance of purchase policies on generated dataset with power-law keyword request distribution

For very small or very large budgets, all algorithms performed similarly, because either very few customers could be satisfied or nearly all keywords could be purchased, making the decision of which keywords to purchase trivial. For budgets which covered the minimum cost of approximately half the keywords, the different policies showed the greatest difference in performance, suggesting that the usefulness of a purchase advisor is constrained by the budget available to the application relative to the data prices.

The purchase set for each policy can be examined in more detail by measuring the value (out of a possible 1.0 total) provided to each customer. The following histogram shows the number of customers provided with each value over the power law keyword request distributions for a budget of 20:



**Figure 3: Histogram of values provided to customers. The shape of each policy’s histogram provides an impression of its fairness in providing reasonably high value to all customers.**

This graph provides an approximate sense of how the keywords purchased are distributed between customers. In comparison to the other three policies, the shape of the advisor’s histogram is skewed toward the high-value end of the graph (ignoring the 0 and 1.0 buckets, the advisor peaks at a higher value than most other policies), suggesting that the advisor prioritizes purchasing keywords that will increase value drastically for a few customers.

## 6. FAIRNESS IN THE PURCHASE ADVISOR

The advisor described in the previous section uses a very simplistic model of utility which determines the utility provided by each keyword in isolation and does not consider what keywords have already been purchased for different customers. We developed a second, discounted advisor which can handle submodular utility functions. This discount advisor takes into account the amount of value a customer has already received when determining the utility gained by purchasing an additional keyword, and therefore can prioritize keywords which are requested by customers who do not have many of their keyword purchased yet. Fairness may be important to application developers interested in retaining a large customer base and providing a baseline level of acceptable service to all application users. Adjusting purchases to prioritize providing similar values to all customers does have some cost, in terms of the total value purchased, and each application will prioritize fairness differently depending on the specific function, data requirements, and customers of the application. To address this, we implemented a version of the advisor that considers fairness and prioritizes keywords requested by customers who have not received many of their requested keywords yet.

### 6.1 Measuring discounted value

In computing the total discounted value of a purchase set, we assign value to each customer based on the number of keywords purchased. We take the logarithm of the number of keywords, rather than using the number of keywords directly, to indicate that, while early increases in the number

of keywords purchased (from 1 keyword to 2) cause significant increase in value, the same increase for a customer with many keywords (from 19 keywords to 20) causes a less significant increase in value. The formula used to compute the total discounted value is:

$$\sum_{c \in C} \log_2(\text{num\_purchased}(c) + 1) * \text{cust\_val}(c)$$

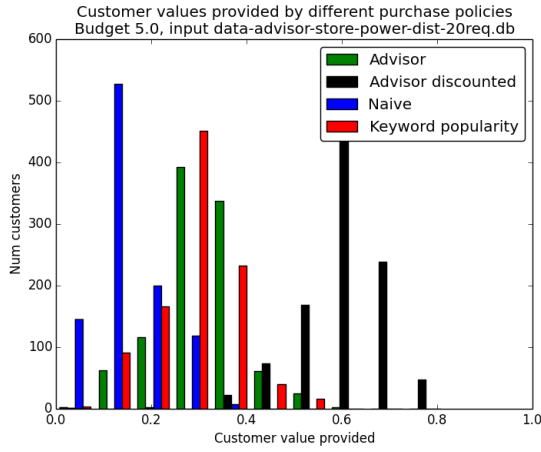
where  $C$  is the set of all customers,  $\text{cust\_val}(c)$  is the value assigned to the customer by the application and  $\text{num\_purchased}(c)$  is the number of keywords requested by customer  $c$  that are in the purchased set. This particular submodular utility function is straightforward to compute directly in our advisor for any given set of keywords. However, if a more complex utility function is specified, convex optimization solvers may be used.

### 6.2 Discounted value advisor

We implemented a purchase advisor which chooses which keyword to purchase next from the set of all possible requested keywords by selecting the keyword that provides the largest possible marginal utility (discounted value - cost). This advisor is more computationally intensive to run than the earlier greedy advisor, because for each keyword it purchases, it must compute the discounted value of each set containing already purchased keywords plus one potential keyword to purchase. Storing the discounted value of different keyword sets as they are computed helps somewhat, but this advisor is still significantly slower to run.

In evaluating the discounted value advisor, we used the generated datasets similar to those created for the last advisor. Some simplifications were made, however: all customers requested the same number of keywords, and the value assigned to each customer-keyword request was fixed at  $(1 / \text{num\_keywords-requested})$  for each customer. This was done to reflect the simplification in the discount value advisor which takes into account only the number of requested keywords that were purchased, not the value assigned to these keywords or the total number of keywords requested by a customer.

We ran the same tests for the total value generated and include a histogram of value per customer. However, in this case, for the discounted advisor, the customer value measures were computed using the log formula described above and scaled back to a  $[0, 1]$  range. The graph below shows the distribution of customer values provided by different policies for one budget and dataset:



**Figure 4: Distribution of customer values provided with a budget of 5 on the power-law keyword distribution dataset.**

The distribution for each policy follows a somewhat smooth curve, but peaks occur at different values for each of the distributions. The naive strategy performs poorly, and the entire histogram is shifted to the left, indicating uniformly lower values. The discounted policy histogram is a smooth, symmetrical curve shifted to the right of the earlier greedy advisor, demonstrating the discounted advisor’s ability to maintain the desired curve peaking near the middle of the distribution while using discounted values instead of the simpler value measure used in the previous section.

## 7. SAMPLING ADVISOR

The purchase advisor above, which buys individual keywords in their entirety, is not a good fit for all data purchase by applications. In many cases, applications can purchase only a small subset of the data which could be used to answer a particular question. The sampling advisor uses a different problem statement to describe a situation in which result tuples for a particular query could be purchased from several data sources, and an application specifies a number of tuples matching that query to purchase rather than purchasing the (potentially very large) result set in its entirety.

A sample of the requested data may be good enough depending on how the application uses the data: if a user queries for restaurants near him, it is reasonable to show only 10 well-rated nearby restaurants, which does not require an exhaustive list of every single restaurant in their area. This is an example of an application that requires only the top  $k$  results, or even any  $k$  of the possible results. If an approximation (any  $k$  of the results near the top of the full list) is good enough, we can purchase samples rather than the full data to provide the requested data without purchasing too much data.

Purchasing only a selected sample can also be good enough in aggregates and analytics use cases. For example, when analyzing data from Twitter, many individual tweets could mention a particular keyword, but if the data purchased is used as an aggregate for an application such as sentiment

analysis, it is not necessary or feasible to purchase all tweets for that keyword. Instead, an application can purchase a 10% sample of tweets matching that keyword or a fixed number of tweets which match the keyword.

## 8. PURCHASING APPROXIMATE DATA

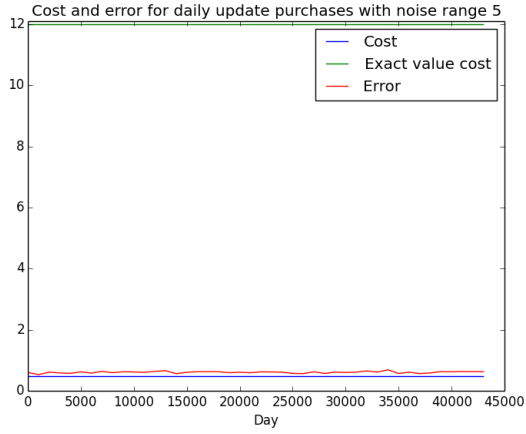
In addition to purchasing a sample of the total requested data to provide a “good enough” answer, we consider purchasing approximate data, which contains slight inaccuracies and is correspondingly less expensive. This is a potentially good fit for aggregate computation and analysis in which no individual point is visible in the final application or otherwise crucial to the application; over a large amount of data, the application may be able to tolerate some inaccuracies. Similarly, if the user is asking for top-rated restaurants, we must purchase the data for nearby restaurants and their ratings, but if we are using the rating solely to sort the restaurant list, it may be reasonable to purchase only approximate (for example, slightly old and stale) ratings. Under the assumption that the current true ratings will not be very different, most of the same restaurants will end up in the top 20 positions of the list, producing a reasonably good list to show to an application user.

### 8.1 Sample Application with Low-Cost Approximate Purchases

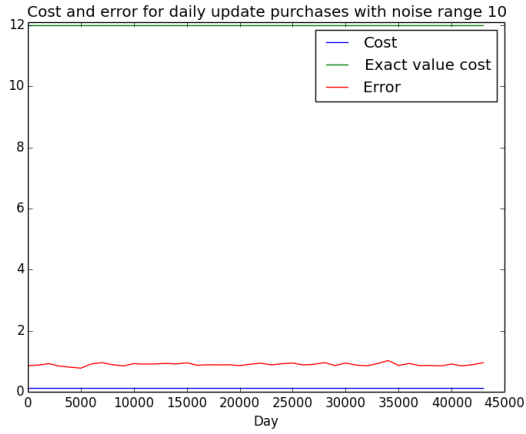
We use a simplified weather application, which reports the average high temperature over the last week, as an example of an application which must balance cost and quality while updating a data set frequently. Acquiring these updates by purchasing each new daily value as it appears is expensive, and, depending on the accuracy requirements of the application, potentially unnecessary. Using historical weather data over the past century for Washington state, we demonstrate the potential for a tiered model of data sales and purchase, with noisy lower-quality values sold for a lower price and incorporated successfully into our model application.

We examine the effect of purchasing lower-quality, cheaper update values instead of the exact values. The cheaper values have noise added from a uniform random distribution with range  $[-r/2, r/2]$ . This lower-quality data is priced according to its variance: the price of the data is  $1/v$ , where  $v$  is the variance of the noise. For the exact data purchases, we estimate the variance from the significant digits in the data: all numbers are rounded to integers, so the noise is akin to a uniform distribution with range 1. As expected, purchasing values with a larger noise range decreases cost dramatically, but in the case of our simple average application, it does not increase error significantly, measured as the absolute value of the difference between the average computed with purchased data and the average computed with the most accurate values available. When buying data with noise added uniformly in the range  $[-10, 10]$  (range 20), the average high temperature computation is typically about 2 degrees off.

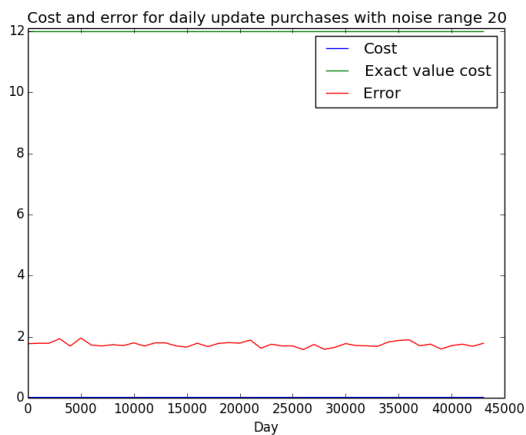




**Figure 5: Daily cost and error of purchasing values with range 5 noise vs exact values.**



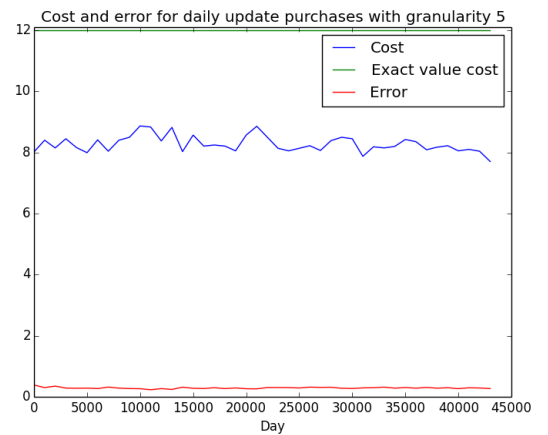
**Figure 6: Daily cost and error of purchasing values with range 10 noise vs exact values.**



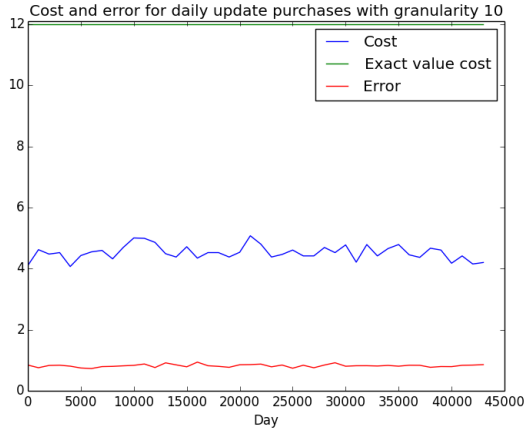
**Figure 7: Daily cost and error of purchasing values with range 20 noise vs exact values.**

Adopting a blanket policy of purchasing noisier data is very cost-effective, but in a data analytics application, we may be able to use other purchased data to improve our estimate. For example, if we have already purchased additional weather metrics such as precipitation and minimum temperature, we can build a simple linear regression model which predicts the high temperature from the past three days of weather metrics purchased and train it on the data we already have. However, our model involves few inputs and produces predictions with high errors. A simple purchase advisor is used to determine which data updates to buy each day. Each day, we produce our prediction value based on the last few days' worth of data. We then purchase an inexpensive noisy value with some range  $r$ . If our prediction is within the range of values that could have produced the noisy value, we deem the prediction good enough and use it to compute our average. Otherwise, we identify this day's update as a case when we have insufficient information to make a good prediction, and proceed to purchase the true value at full price. The noisy value serves only as an inexpensive check on our prediction. This purchase advisor is a simplified example which demonstrates the role of an advisor in identifying data updates to purchase while maintaining a balance of cost and accuracy suitable for the application.

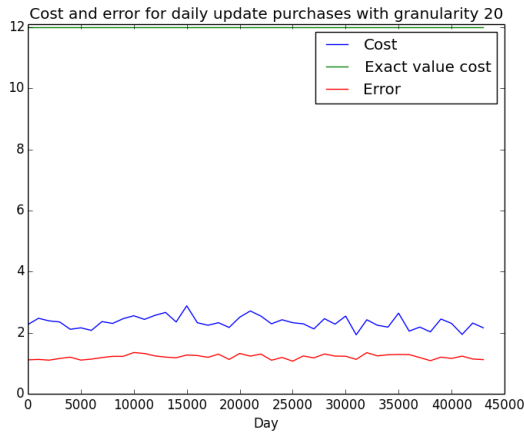
The purchase advisor which incorporates predictions incurs significantly higher costs than simply purchasing only noisy values, but it offers a slight increase in accuracy which may be worthwhile depending on the application. It is also largely dependent on the quality of the regression model used for predictions: in an existing data analytics system with a large store of historical data to train a predictor, predictions may prove accurate enough that most values will only require an inexpensive noisy purchase as a check.



**Figure 8: Daily cost and error of purchasing values with range 5 noise vs exact values, using predicted values as guide.**



**Figure 9: Daily cost and error of purchasing values with range 10 noise vs exact values, using predicted values as guide.**



**Figure 10: Daily cost and error of purchasing values with range 20 noise vs exact values, using predicted values as guide.**

## 9. CONCLUSION

As data-rich applications continue to make use of a large and growing online data market, a data purchase advisor can play an important role assisting developers with managing increasingly complex and real-time data purchase decisions. The data advisor we propose here attempts to take into account a wide range of applications, data sets, and priorities and measures of success, balancing cost with harder-to-specify measures of the performance and quality of information provided by an application. These measures are very application-dependent, leaving a number of open questions in the advisor about how and when to purchase data. Instead of specifying a single algorithm for a purchase advisor, we have developed a framework and set of questions to consider, and implemented several simple advisors with different priorities, decision rules, and supported applications as a demonstration of the possible roles an advisor can play in application development and long-term support.

We have examined the problem of developing a data advisor from several different viewpoints: offline and online advisors, advisors which accommodate a variety of data and customer priorities, and, perhaps most useful, advisors which work in less ideal situations to provide a practical balance of cost and accurate information. For applications with a focus on analytics and using large quantities of data, this approximate advisor may prove most useful in managing large data consumption at reasonable cost. In each case examined in the work above, we have outlined some of the tradeoffs and choices that must be made depending on the application which the advisor is to support. We have then proceeded to demonstrate the performance of a sample advisor constructed from this scenario. Our advisors are not developed to a point of supporting real use yet, but they provide a proof of concept for a variety of relatively simple purchase advisors which handle purchase decisions in a simple manner while attempting to maximize a developer-defined success metric for the application.

Regardless of how it is implemented, use of a purchase advisor is very much dependent on how data is purchased. In this area, we have surveyed the current data market and popular data sources, sellers, and APIs, but also make some assumptions about data purchase which are not yet a reality in the online data market. Despite the popularity of online data sales, there remains a gap between current publicly available sales channels and the granularity and automation required for intelligent, real-time purchases by a purchase advisor. To fill this gap, we have proposed a data SLA as a standardized method of specifying data quality and freshness metrics. This concept is somewhat more general than the advisor, allowing for the comparison and combination of different online sources to build a more comprehensive and useful data set.

The data purchase advisor as we have described it is a flexible concept which can be adapted to a variety of applications and data markets. In defining its interaction with the developer and data sellers, its inputs and outputs, and its decision mechanisms, we have raised several questions about how data-rich applications and data sellers interact with or without a purchase advisor. Beyond the single-purpose advisors we have implemented here, built to support a single simple application, there remains a gap between current data purchase mechanisms and strategies and a more general-purpose purchase advisor which can be widely used by developers independently of the application development itself. What a more general data purchase advisor looks like will depend on future work in online data sales, data purchase metrics, and development of a standardized language for data use in increasingly data-heavy applications.

## 10. REFERENCES

- [1] Schomm, Stahl, Vossen. Marketplaces for Data: An Initial Survey.
- [2] "API Overview." Yelp for Developers. Yelp. [http://www.yelp.com/developers/getting\\_started](http://www.yelp.com/developers/getting_started).
- [3] Dumbill, Edd. "Data Markets Compared." Strata. O'Reilly, 7 Mar. 2012. <http://strata.oreilly.com/2012/03/data-markets-survey.html>.
- [4] "Google Prediction API." Google Developers. Google.

- <http://developers.google.com/prediction/sla>.
- [5] Products. Gnip. <http://gnip.com/products/>.
  - [6] Windows Azure Marketplace. Microsoft.  
<http://datamarket.azure.com/>.
  - [7] "Jigsaw API." Data.com.[http:](http://www.data.com/export/sites/data/common/assets/pdf/DS_Datadotcom_Connect_API_Docs.pdf)  
[//www.data.com/export/sites/data/common/](http://www.data.com/export/sites/data/common/assets/pdf/DS_Datadotcom_Connect_API_Docs.pdf)  
[assets/pdf/DS\\_Datadotcom\\_Connect\\_API\\_Docs.pdf](http://www.data.com/export/sites/data/common/assets/pdf/DS_Datadotcom_Connect_API_Docs.pdf).
  - [8] Ortiz, Almeida, Balazinska. A Vision for Personalized Service Level Agreements in the Cloud.
  - [9] Bouzeghoub, Peralta. A Framework for Analysis of Data Freshness.
  - [10] Agarwal et al. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data.
  - [11] M.J. Menne, C.N. Williams, Jr., and R.S. Vose. United States Historical Climatology Network Daily Dataset. National Climatic Data Center, National Oceanic and Atmospheric Administration. [http://cdiac.ornl.gov/ftp/ushcn\\_daily/](http://cdiac.ornl.gov/ftp/ushcn_daily/)