Drum: A Rhythmic Approach to Interactive Analytics on Large Data

Jianfeng Jia, Chen Li, Michael J. Carey
Department of Computer Science, University of California, Irvine
{jianfenj,chenli,mjcarey}@ics.uci.edu

***Abstract*—In this paper, we study how to progressively answer a time-consuming query on a large data set by generating a sequence of mini-queries. We formulate an optimization problem to produce the predicates of mini-queries by considering both their total running time as well as the smoothness of result delivery in order to show the incremental results at a rhythmic pace to improve the user experience. We develop an adaptive framework called Drum that can collect the runtime behavioral statistics of the database system to decide the predicate of the next mini-query appropriately. The framework is a general middleware solution without any changes to the underlying database system. We have conducted extensive experiments on a large, real data set, and the results show that Drum can reduce the delay of delivering intermediate results to the user without sacrificing much total time.**

***Keywords*-progressive computing; online aggregation; analytics; big data; response time; Cloudberry; AsterixDB;**

I. Introduction

Data exploration and analytics are becoming increasingly important in applications to help users gain insights from their data and make time-critical decisions. Data analysis can become even more powerful and desirable by being *interactive*, so that users can see the results quickly, ideally in sub-seconds after submitting a request. At the same time, achieving such a user experience is technically challenging on large data sets due to the high computational cost.

As a motivating scenario, suppose we want to build a system that allows users to explore and visualize a large collection of social media tweets. For illustration purposes, we use Table I (Section II) as a running example, which is a relation with sample tweets. Its simplified schema contains attributes including the time, location, and message of a tweet. Suppose that this relation has a large number of records, e.g., hundreds of millions or billions of records. An example analytical request is "show the number of tweets mentioning the keyword `zika` per state." Fig. 1 presents an interface for showing the results, where the map uses colors to indicate the aggregation count of a state.

To serve this request, we can send the following query to the database:

Q:
```
SELECT state, COUNT(*)
FROM twitter
WHERE ftcontains(message, "zika")
GROUP BY state;
```

The predicate `ftcontains(message, 'zika')` checks if the textual *message* value contains the keyword `zika`. Due to the large number of records in the relation, the query Q can take a long time to finish, and the user has to wait during this period. One way to solve this long-waiting-time problem is to issue a sequence of cheaper "mini-queries" by adding a range predicate on the `create_at` attribute so that each of them can be answered by the database system efficiently. The following is an example mini-query:

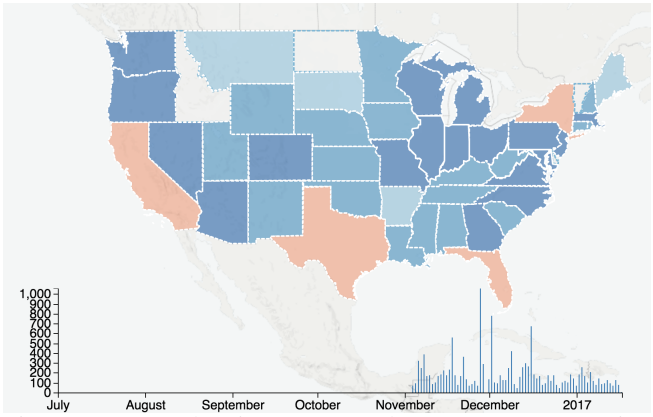


Figure 1: An UI showing aggregated tweet counts per region

Q_m:
```
SELECT state, COUNT(*)
FROM twitter
WHERE ftcontains(message, "zika")
WHERE 2016-01-01 <= create_at AND
      create_at < 2016-03-01
GROUP BY state;
```

One naive solution is to divide the space of the `create_at` attribute into multiple *fixed-length intervals* and generate a mini-query for each of them. For instance, we could divide the attribute into multiple months and generate a mini-query for each month. While this slicing method is easy to implement, it has two major limitations. First, the interval is difficult to decide, especially for queries with different query times. For instance, a tweet query for a popular keyword such as `water` can take a much longer time than a query with a rare keyword such as `authoritarianism`. Second, the user experience can be very poor due to the large variance of running times of the min-queries, mainly due to two factors. (1) The distribution of the relation on the slicing attribute can be skewed. Fig. 2 shows the distribution of the number of tweets mentioning the keyword `zika`, which has a large variance. There was a peak in the summer of 2016 due to the Olympic Games in Rio de Janeiro, Brazil and to the global concern about this potential epidemic, but public attention quickly diminished after that. If we divide the time range equally into 14 months

(as shown in the figure), the mini-queries will have different running times because they need to access different amounts of data. (2) The performance of the backend database system can fluctuate over time, especially when there are multiple queries running simultaneously. As a result, some of the mini-queries can be fast, and others can be slow, causing the incremental results to be sent at an irregular pace.

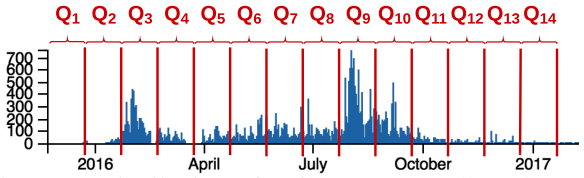


Figure 2: Distribution of "Zika" tweets collected over 1.5 years from November 2015 to May 2017.

In this paper we study how to progressively answer a time-consuming query on a large data set by generating a sequence of mini-queries. We focus on how to deliver the results of mini-queries smoothly by following an "expected rhythm" for the user so that the user sees regular updates of the incremental results. We address two main challenges, as discussed above: (1) The data distribution of the slicing attribute is hard to model using offline statistics, especially when the query can have various selection predicates. In the example, the user can type in arbitrary keywords, and it would be difficult to know a-priori the distributions for the keywords. (2) The performance of the database system can be very dynamic and thus hard to model offline due to other running queries that compete for the limited computing resources. We make the following contributions here:

- We formulate an optimization problem to generate the predicates of mini-queries by considering both their total running time as well as the smoothness of result delivery to deliver incremental results at a rhythmic pace to improve the user experience (Section II).
- We develop an adaptive framework, called "Drum"[1], in which we collect run-time behavioral statistics of the mini-queries and the backend database system. It includes a regression function for the relationship between the predicate and the running time of a mini-query and an uncertainty model for the estimation error of the regression function. It also includes a greedy algorithm that can automatically use the observed performance information to generate the predicate for the next mini-query (Section III).
- We develop a technique that considers both the total running time and the penalty of missing the next expected milestone to deliver the incremental results. Based on a rigorous analysis of the benefit and cost of varying the predicate, this technique can select a predicate that can maximize the gain of the next mini-query (Section IV).

[1]Drum stands for "Data Retrieval Using Milestones."

- We have conducted an extensive experimental study based on a real, large data set of social media tweets to evaluate Drum and these techniques and to demonstrate their efficiency (Section V).

A. Related Work

Progressive computing: Many approaches have been proposed in the literature to support interactive queries on large data sets residing in a backend database system [1], [2], [3], [4]. For example, [1], [2] developed a solution called "online aggregation" that progressively shows approximate aggregation answers with a confidence interval based on partial results and a statistical model. The solution relies on random sampling, which requires significant changes to the underlying database system. [5] introduced another method for improving the user experience by showing partial results, which also requires changes to the database system. [6], [4] studied online aggregation in a MapReduce-like framework. [7], [8] studied how to provide aggregation results with different levels of detail incrementally by using an auxiliary data structure specially designed for spatio-temporal data; the data structure needs to be synchronized with the underlying database system. Our study is different since we consider the case where the underlying database system is used as a "black box" without any changes.

[9] also addressed how a middleware layer can decompose a query into mini-queries to improve responsiveness. Their focus was mainly on how to select an attribute to do the decomposition, and the generated mini-queries have the same-size predicate intervals. (See the analysis of fixed-length intervals above.) Our focus is on how to generate *variable-range* predicates for the mini-queries to maintain the rhythm of delivering results.

Waiting time in progressive computation: Since the user encounters a series of waiting times, the quality of the experience heavily depends on the timing of those updates on the frontend [10], [11]. Similar to the case of streaming online videos, user-perceived quality can suffer from freezes during the delivery of results. A sudden, different waiting time can be different from the user's expectation, causing a negative experience [11]. Thus, the rhythm of result delivery is a key determinant towards a user satisfaction in progressive computing. Deciding on a good amount of waiting time for the next incremental update has been studied in [10], [12], [13]. [14] suggested 10 seconds as an upper time limit for keeping the user's attention focused on the interface. Instead of returning a single response after several minutes, progressive computing provides a flow of partial results that helps users "lose their sense of time", resulting in a good experience of distorted time perception [11].

Query time estimation: Estimating the running time of a query is a fundamental topic extensively studied (e.g., [15], [16]). Many of the existing techniques can be adopted in our Drum framework as part of the regression function.

[17] considered an uncertainty model that relied on internal information about a database system that may not be available in many applications. [18] implemented a system called BlinkDB that allows users to choose a trade-off between query accuracy and response time. [19] considered an uncertainty model for admission control of multiuser workloads, one which needs to be trained in advance on sample data. Our proposed Drum solution is different; it does not rely on internal information about the database system nor a pre-trained model, as it collects statistical information on the fly during the execution of mini-queries.

II. Problem Formulation

A. Architecture and Query Slicing

We consider a widely-used three-tier Web architecture that includes a backend database system, a middleware server, and a frontend interface that runs in a browser, as shown in Fig. 3. A frontend user sends requests to the middleware, which in turn generates queries to the backend system to retrieve results and sends a response to the frontend.

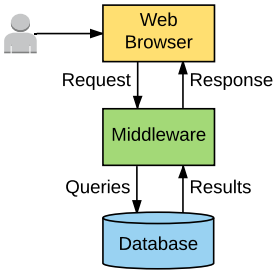


Figure 3: Architecture

Consider a relational table $R(A_1, A_2, \ldots, A_n)$ in the backend database. The middleware posts a query Q to the table. Due to a large number of records in R, the query Q can be computationally expensive, and the user needs to wait for a long time to see the results. We assume that the relation R has an attribute ρ, called its *slicing attribute*, using which the middleware can generate a sequence of mini-queries $Q_1, Q_2, \ldots$, and send them to the backend. Each mini-query Q_i is Q with an additional filtering condition on the slicing attribute ρ. After receiving the results of each mini-query Q_i, the middleware uses these results to update the frontend interface. We make the following two assumptions:

1) Each mini-query is much faster than the original query Q, e.g., due to the smaller amount of data accessed by Q and an available index on this attribute.
2) The results of these mini-queries can be incrementally combined to compute the results of the query Q.

Table I shows an example relation with sample tweets. Section I showed an original tweet query Q and a mini-query Q_m on this table. Notice that after receiving the results of each mini-query, the frontend can decide how to combine them with earlier results and update the interface. It can either show the results as they are or it could show estimated results for the whole data set based on a statistical model. Our work mainly focuses on the timing of the generated mini-queries, not on the way their results are displayed. We also focus on the case where ρ is a numerical attribute.

B. Slicing Schedules and User Satisfaction

There can be many ways to slice a query into mini-queries. Fig. 4 shows three ways to answer our example query Q. (We call each of them a *slicing schedule*, or just *schedule* for short.) The first schedule S_1 uses a single query (Q) that takes 6 seconds to finish. The second schedule S_2 uses 4 mini-queries that take 2 seconds each, with a total time of 8 seconds. The third schedule S_3 also uses 4 mini-queries, but they take 1, 3, 1, and 3 seconds, respectively.

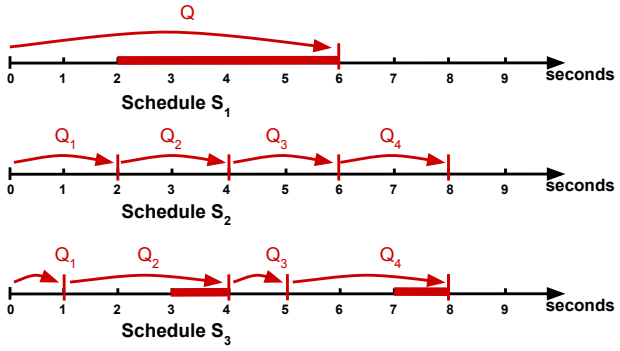


Figure 4: Three query-slicing schedules, where each thick line indicates a delay for an expected 2-second pace.

When deciding a good strategy to slice a query, we need to consider how the corresponding mini-query schedule affects the user experience. In the three schedules above, schedule S_1 takes the least amount of total time (6 seconds), but the user cannot get any intermediate results before that. Schedule S_2 takes a longer time (8 seconds) to finish, but it gives the user an earlier response and updates the interface periodically every 2 seconds. Schedule S_3 takes the same total amount of time as S_2, but its update frequency is not regular. From the user's perspective, the way the interface is updated in S_3 is not very "smooth," as the next time the new results are shown is not predictable. Schedule S_2 updates the interface with a regular pace and has a better predictability and rhythm in terms of the next time the user expects the interface to be refreshed, so it can provide a better user experience.

C. Schedule Quality

The example above suggests the following factors that affect the user experience in the way that the mini-query results are delivered:

1) **Total running time**: For a frontend request, we want to minimize the total time of these mini-queries to compute the complete results.
2) **Smoothness of result delivery**: Users want the frontend to be updated at a *regular* pace so that the next time of seeing new results is predictable.

Based on this analysis, we define the cost of a schedule S as follows:

$$Cost(S) = Cost_t(S) + Cost_m(S). \quad (1)$$

Id	Create_at	Message	City	State	Coordinate
799794320	2016-11-18	`out state is on fire lol`	Greensboro	NC	[-80.029518, 35.962623]
810674471	2016-12-18	`House Fire (Frisco)`	Frisco	TX	[-96.937783, 33.081206]
819372472	2017-01-01	`When's the Knicks fire sale?`	Memphis	TN	[-90.135782, 34.994192]
819734325	2017-01-12	`his music is actually fire`	Buffalo	NY	[-78.79485, 42.966451]

Table I: A sample table of tweets

In the formula, $Cost_t(S)$ measures the time cost of the mini-queries, which can be quantified by their total running time. $Cost_m(S)$, where "m" means "smoothness," measures the smoothness of result delivery. To quantify this cost, we assume a given desired *pace* parameter P (an interval time), and the user expects an update of the frontend P time after the previous update. Every time the middleware does not update the interface after this P time, this schedule needs to pay a "missing-the-deadline" penalty. The smoothness cost can be quantified as:

$$Cost_m(S) = \alpha \sum_i D_i. \quad (2)$$

In the formula, α is a weight used to quantify the penalty of missing the next deadline. D_i is the delay time of mini-query Q_i based on the pace P, defined as:

$$D_i = max(0, U_i - (U_{i-1} + P)).$$

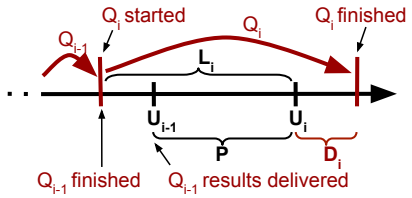


Figure 5: Missing a delivery deadline.

U_i and U_{i-1} are the update times when the results of mini-queries Q_i and Q_{i-1} are delivered, respectively. The general idea is shown in Figure 5. Notice that the middleware starts the new mini-query Q_i right after the previous mini-query Q_{i-1} finishes, but the middleware can choose to wait for some time before delivering the results to the frontend in order to provide a smooth experience since the user is expecting an update at time U_{i-1} based on the pace P. (Our developed results are orthogonal to whether or not the middleware decides to wait on the mini-query results before the next milestone.)

Schedule	Cost	Cost ($\alpha = 2$)
S_1	$6 + 4\alpha$	14
S_2	8	8
S_3	$8 + 2\alpha$	12

Table II: The costs of the schedules.

Table II shows the formula's costs for the three schedules in Fig. 4 when pace $P = 2$ seconds. Take schedule S_3 as an example. It finished in 8 seconds, so $Cost_t(S_3) = 8$ seconds. Its Q_1 and Q_3 did not miss their deadlines, while Q_2 and Q_4 missed their deadlines by 1 second each, so its smoothness cost is $Cost_m(S_3) = \alpha * 2$ seconds. If $\alpha = 2$, for example, the total cost of S_3 is 12 seconds. In general, the lower the cost a schedule has, the better a user experience it should provide.

Next we study the following problem: *given a query Q, generate a sequence of mini-queries whose execution schedule has a minimal cost.*

III. Drum: An Adaptive Framework for Generating Mini-Queries

In this section, we present an adaptive middleware-based framework, called Drum, to dynamically decide a condition on the slicing attribute for the next mini-query based on statistics collected from earlier mini-queries. A main advantage of this framework is that it does not require any apriori knowledge about the performance characteristics of the backend database system, and it can be adaptive with respect to skewed data distributions and performance fluctuations of the backend.

Fig. 6 shows the framework. Each incoming request from the frontend is submitted to the mini-query generator. The generator utilizes estimation information provided by the estimator module to choose a range of size r_i of the slicing attribute for the next mini-query Q_i. It then generates Q_i and sends it to the backend database to execute. After the intermediate results return, the middleware sends updated results to the frontend, possibly by combining them with earlier results. Meanwhile, the run-time statistics of executing the mini-queries are collected and stored in the estimator module. The estimator utilizes the collected information to help the generator create the next mini-query. Next we give the details of the framework.

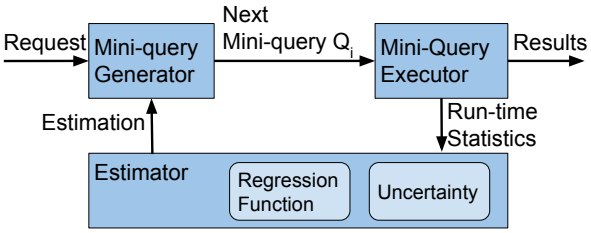


Figure 6: Drum framework for adaptive query slicing.

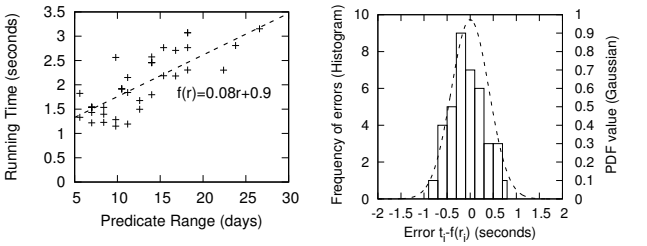
A. Regression Function

The regression function in the estimator is used to capture the relationship between a slicing-predicate range size and the corresponding mini-query running time. As more statistics are collected, the more accurate the regression function can become. Since each mini-query is obtained by adding a predicate on the slicing attribute, we need to know how this predicate affects the execution time of the mini-query. As an example, if the mini-query time is closely related to the number of records satisfying the predicate, we can use

a linear function between a predicate range size r_i and the running time $f(r_i)$ for the next mini-query Q_i:

$$f(r_i) = a_1 r_i + a_0. \quad (3)$$

Notice that the values a_1 and a_0 in the regression can be request-dependent, so we build a linear regression for each request. That is, we would have one linear function for a `Zika` tweet query and another for an `authoritarianism` query. We can use a standard curve-fitting algorithm with the least-square fitting method to train the linear models on the collected pairs of range size and running time. Each new observation from running a mini-query can update the linear model. Note that due to data skew and fluctuation of system performance, the relationship between the predicate range size and running time can also change over time. To enable the regression function to quickly adapt to such changes, we can adjust the length of the history to train the model. As an illustration, Fig 7a shows a linear model learned from a collection of pairs of range size r_i and running time t_i:



(a) Linear regression for tweet mini-queries with different predicates on the `create_at` attribute. (110 million tweets, keyword=`zika`)

(b) The Histogram and Gaussian models. The errors are collected from the observation $t_i - f(r_i)$ in Fig. 7a.

Figure 7: Regression function and error models

B. Uncertainty Model

The real running time t_i for a mini-query Q_i will differ from the predicted time $f(r_i)$ from the regression function. The uncertainty model in Drum is used to measure the distribution of such per-query errors for use in selecting r_i. We consider two types of models: (1) *Histogram*: We split the space of the errors into equal-size bins and maintain a frequency for each bin. (2) *Gaussian function*: We use a function to model the error distribution. For example, we can assume errors follow the Gaussian distribution function $N(0, \sigma)$. Fig. 7b shows both the histogram distribution and Gaussian distribution for the errors $t_i - f(r_i)$ collected in Fig. 7a. For the same collection of observations, using different models can produce different probability values and thus lead to different mini-query choices.

In reality, the error distribution may depend on the target running time $f(r_i)$. Intuitively, the larger the time $f(r_i)$, the harder it may be to decide a predicate range r_i to achieve the desired time. In addition, the middleware has a limited number of performance data points collected during the execution of past mini-queries. For simplicity, our model will assume that the error distribution is independent of the target time $f(r_i)$. That is, if the error probability distribution function (PDF) is $PDF(t_i - f(r_i))$, then for a specific given $f(r_i)$ the distribution of t_i is $PDF\big(t_i|f(r_i)\big) = PDF\big(t_i - f(r_i)\big)$. For instance, when using the Gaussian distribution to model the error distribution as $N(0, \sigma)$, we assume the real running time t_i follows the distribution of $N(f(r_i), \sigma)$, where $f(r_i)$ is the target running time and σ is the standard derivation of all the observations.

C. Tradeoff of Running Time and Penalty

When deciding the predicate range r_i for the next mini-query Q_i, if we choose a large range, the total running time can be reduced, since the middleware will send fewer mini-queries to the database. At the same time, a mini-query with a large range r_i will have a longer execution time, which can cause a large penalty of missing the next deadline. To consider the tradeoff between the two factors, we define the following scoring function for mini-query Q_i:

$$score(Q_i) = \begin{cases} \frac{r_i}{I} & \text{if } t_i <= L_i \\ \frac{r_i}{I} - \alpha \frac{t_i - L_i}{C_i L_i} & \text{otherwise.} \end{cases} \quad (4)$$

In the function,

- I is the entire interval range of Q_i's slicing attribute;
- r_i is the predicate range on the slicing attribute of Q_i;
- $\frac{r_i}{I}$ quantifies the progress that Q_i will contribute to the overall task;
- L_i is the time limit (i.e., time to deadline) for the next mini-query Q_i. As shown in Fig. 5, the limit L_i can be bigger than the pace P if the previous query Q_{i-1} completes before its own deadline;
- t_i is the real running time of query Q_i;
- α is a constant weight in the penalty function in Equation 2;
- C_i is the estimated total number of mini-queries needed. It can be estimated from the number of mini-queries issued so far and the summation of the relative progress that has been made. $C_i L_i$ is thus a current estimate of the total running time, making the penalty be of the same scale as the progress $\frac{r_i}{I}$. Notice that C_i is specific to the mini-query Q_i, but in our analysis step for deciding range r_i, it acts as a constant.

Using the linear function in Fig. 7a and the error PDF from Fig. 7b, Fig. 8 shows the score values for two r_i values for Q_i, where we have $L_i = 2.2$ seconds from the time Q_{i-1} was finished until the time we need to have the results of Q_i in order to update the frontend. Fig. 8(a) shows the score for a strategy A_1 that picks a range $r_i = 16$ days for a target running time of $f(r_i) = 2.2$ seconds. When the real time t_i is less than L_i, the score is equal to the progress. As the real time t_i becomes larger than L_i, the score constantly decreases since we incur a penalty for missing the L_i deadline. Fig. 8(b) shows the PDF of the running time t_i that is of the same shape as shown in Fig. 7b, but the mean of the distribution is set to $f(r_i) = 2.2$.

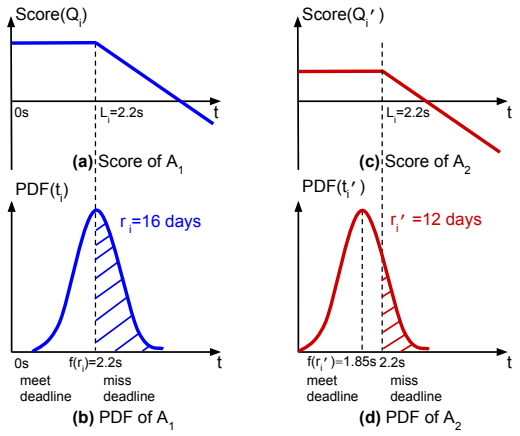


Figure 8: The score and error PDF for two target running times, $r_i = 2.2$ seconds and $r_i' = 1.85$ seconds, with time limit $L_i = 2.2$ seconds, $f(r_i) = 0.08r_i + 0.9$, and $PDF(t_i) = N(f(r_i), 0.4)$.

Fig. 8(c) shows the score function for another strategy A_2 that picks a range $r_i' = 12$ days and the target running time $f(r_i') = 1.85s$, which is more conservative, by using a value smaller than L_i. Consequently, its maximum score is smaller than the value when we had $f(r_i) = 2.2s$. Fig. 8(d) shows the PDF of the running time t_i when r_i' is 12 days. Compared to the previous $f(r_i)$, the PDF "shifts" to the left, providing a larger probability of having results to be delivered by the L_i deadline. When comparing the two strategies, we notice that strategy A_1 is more aggressive, choosing a target running time $f(r_i) = 2.2s$, while A_2 is more conservative, using a smaller time $f(r_i') = 1.85s$ (with a smaller slicing range r_i'). As a result, A_2 has a lower progress r_i'/I, but also a lower probability of missing the deadline.

D. Choosing Range r_i for Next Mini-Query Q_i

At each step, Drum picks a range r_i to maximize the expected score for mini-query Q_i so that Q_i can finish as close to the deadline L_i as possible but have less risk of running longer than L_i. In this heuristic way, the schedule composed by all Q_i's can deliver the results at a regular pace. For a specific range r_i, the running time $f(r_i)$ can be estimated using the regression function. Let $P(t|f(r_i))$ be the PDF of the real running time t_i when the targeted time is $f(r_i)$. Then the expected score $E(score)$ of the mini-query Q_i can be calculated as:

$$\begin{aligned} E(score(Q_i)) &= \int_0^{\infty} score(Q_i)P(t|f(r_i))dt \\ &= \frac{r_i}{I} - \alpha \int_{L_i}^{\infty} \frac{(t-L_i)}{C_i L_i} P\big(t|f(r_i)\big)dt. \end{aligned} \tag{5}$$

Our goal is to compute a value r_i to maximize this expected score $E(score(Q_i))$.

IV. Choosing an Optimal Predicate Range

We consider two different error models for the time distribution $P(t|f(r_i))$, namely histogram and Gaussian, and develop a technique for deciding the range r_i for each of them to maximize the expected score $E(score(Q_i))$.

A. Histogram

Calculating expected score for a given range r_i: For a given predicate range r_i and the corresponding estimated running time $f(r_i)$ from the regression function, we want to calculate the expected score for the generated mini-query Q_i. The intuition of our method is the following. The obtained histogram will give an error distribution centered at $f(r_i)$. We apply Equation 5 to compute the integral of the score using the error distribution. In particular, before the time limit L_i there is only progress without any penalty, and after L_i the penalty increases linearly. The integral can be computed based on the probability accumulated in each of the bins whose ending time is larger than L_i. Depending on the distance between $f(r_i)$ and L_i, the L_i value could fall into different bins with different probabilities, resulting in different score computations.

Using the previous example of $L_i = 2.2$ seconds and $f(r_i) = 0.08r_i + 0.9$, Fig. 9 shows the histogram distribution of the running time t_i when $r_i = 12$ days and $f(r_i) = 1.85s$. The limit L_i falls into the 5th bin in the histogram. The bins that are larger than L_i will contribute to the penalty. Within the interval bin $[2.0, 2.3]$, we assume that the error is uniformly distributed. The shallow part of the bin will also contribute to the penalty.

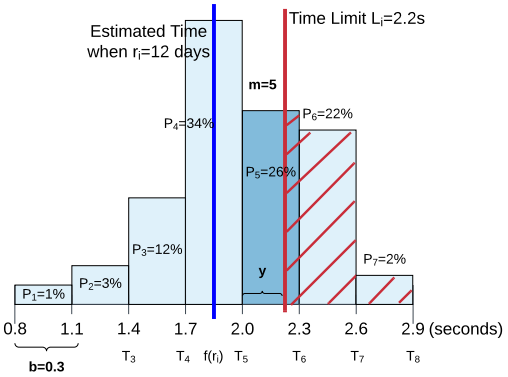


Figure 9: Use an error histogram to compute the expected score for targeting running time $f(r_i)$.

Formally, we assume the error distributes evenly within each interval bin of id k. Hence the PDF for the error within the k-th interval bin is $PDF_k = Pr_k/b$, where Pr_k is the aggregated probability in the bin. Then the expected score can be computed as:

$$\begin{aligned} E(r_i) = \frac{r_i}{I} - \frac{\alpha}{C_i L_i}\Big(\int_{L_i}^{T_{m+1}} (1 - \frac{y}{b}) \frac{Pr_m}{b}(t - L_i)dt + \\ \sum_{k=m+1}^{n} \big(\int_{T_k}^{T_{k+1}} \frac{Pr_k}{b}(t - L_i)dt \big) \Big). \end{aligned} \tag{6}$$

- n: total number of bins in the histogram;
- b: the width of each bin;
- k: the sequence id of each bin;

- T_k: the start time of the k-th bin;
- m: the id of the bin which time limit L_i falls into, i.e., $T_m \leq L_i < T_{m+1}$;
- y: the time difference $L_i - T_m$.

For a specific range r_i that makes L_i fall into the m-th bin, Equation 6 can be transformed to:

$$E(r_i) = \frac{r_i}{I} - \frac{\alpha}{C_i L_i}\Big(\frac{Pr_m}{2b^2}(b-y)^3 - \sum_{k=m+1}^{n} Pr_k y + b\sum_{k=m+1}^{n} Pr_k(\frac{1}{2}+k-m)\Big). \quad (7)$$

Computing r_i to maximize the expected score: Fig. 10 shows how the expected score changes as we increase the range r_i for the histogram distribution in Fig. 9 for different values of the penalty weight α. As r_i increases, initially the expected score also increases since the progress increases without much penalty. After a certain time, the expected score starts declining since the penalty of missing the time limit also increases. There is an optimal r_i that can achieve the best expected score, and the optimal value depends on the penalty weight α.

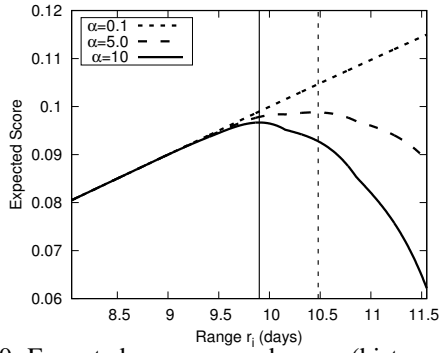


Figure 10: Expected score as r_i changes (histogram model).

In general, we can find an optimal r_i to maximize the expected score as follows. For each interval bin B_m that the time limit L_i falls into, we can use Equation 7 to compute a best y value, namely y_{max}, that maximizes the expected score. The value y_{max}, if it exists, must satisfy the following equation (more details can be found in the appendix in [20]):

$$(b - y_{max})^2 = \frac{2b^2}{3Pr_m}(\frac{C_i L_i}{\alpha a_1 I} - \sum_{k=m+1}^{n} Pr_k). \quad (8)$$

Let g be the id of the bin that covers $f(r_i)$. Using Equation 8 we can compute the corresponding y_{max} value. We can then compute the best $f(r_i)$ as:

$$f(r_i) = L_i - y_{max} - (m - g - 1/2)b.$$

Then we can compute the best r_i value based on the regression function in Equation 3.

Notice that the expected score in Equation 7 will change based on which bin L_i falls into, so we need to examine all the possible bins to find the r_i that yields the global maximum expectation. A simple r_i optimization method is to find such a y_{max} for each of the bins and choose the one with the largest value.

Bin id k	$E_{max}(score)$	r_i (days)
4	0.085	10.85
5	0.095	10.15
6	0.096	9.9
7	0.087	9.45

Table III: Maximal expected scores.

Then we use the y_{max} value for this chosen bin to compute the corresponding r_i as the final predicate range on the slicing attribute. Table III shows the maximum expected score for each interval bin in our running example from Fig. 9.

B. Gaussian Distribution

Assume that the estimated time error $t_i - f(r_i)$ follows a Gaussian distribution $N(0, \sigma)$. With this assumption, the value of t_i follows the distribution of $N(f(r_i), \sigma)$. The parameter σ is the standard deviation of the observation data. We keep updating a single standard deviation estimate after seeing each new data point of the performance of the database system. One advantage of the Gaussian distribution is that it has a closed-form PDF. We can replace the PDF in Equation 5 with $N(f(r_i), \sigma)$ and get the following function (more details can be found in the Appendix in [20]):

$$E(r_i) = \frac{r_i}{I} - \alpha \int_{L_i}^{\infty} \frac{t - L_i}{C_i L_i} \frac{1}{\sqrt{2\pi}\sigma} e^{-(\frac{t - f(r_i)}{\sqrt{2}\sigma})^2} dt. \quad (9)$$

Let

$$z = \frac{f(r_i) - L_i}{\sqrt{2}\sigma}. \quad (10)$$

We compute z_{max} that yields the maximum expectation as:

$$z_{max} = \text{erf}^{-1}(\frac{2C_i L_i}{a_1 I \alpha} - 1). \quad (11)$$

Using the model in Equation 3, we compute r_i as:

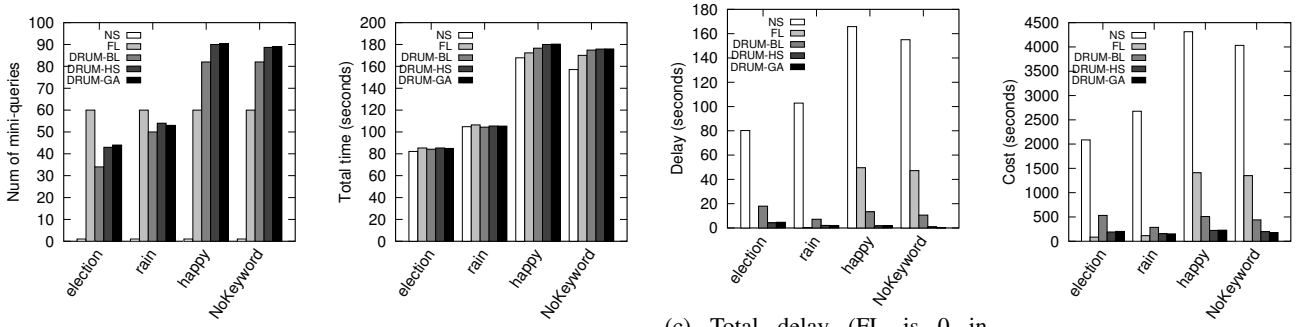
$$r_i = \frac{\sqrt{2}\sigma z_{max} + L_i - a_0}{a_1}, \quad (12)$$

To summarize, for a given time limit L_i, penalty weight α, range space I, coefficients a_1 and a_0 of the linear function, and variance σ of the Gaussian model, we can use Equations 11 and 12 to compute r_i to have the maximal expected score.

V. Experiments

A. Setting

We collected 114 million tweets using the Twitter streaming API from November 14, 2016 to January 17 2017. We used Apache AsterixDB [21] (0.9.2 version) as the backend database to store the data. Its schema included the following attributes: *(**id**: int, **create_at**: datetime, **message**: string, **lang**:string, **stateID**: int, **countyID**:int, **cityID**:int)*. The total size of the records in the database was 90GB. The Drum middleware was written in Scala 2.11.7 and ran on a 64-bit JVM (Oracle 1.8 version). Both the middleware and database ran on a machine with a CPU of 2 cores, 16GB memory, and a 500GB M.2 SSD disk, running the Centos 7 operating system. We evaluated queries to count the number of tweets mentioning a particular keyword. The original, unsliced queries to AsterixDB were as following:



(a) Number of mini-queries (b) Total time (c) Total delay (FL is 0 in `election` and `rain` cases) (d) Total cost ($\alpha = 25$)

Figure 11: Evaluating slicing methods

```
SELECT COUNT(t)
FROM twitter t
WHERE ftcontains(t.message, $keyword);
```

We considered different types of queries regarding their keyword condition. (a) NoKeyword: The query did not have an `ftcontains` predicate; (2) Uniform keyword: The keyword in the query was distributed uniformly over time. An example was `rain`, whose daily tweet number did not change very much over time. (3) Non-uniform keyword: The keyword had a skewed distribution in the time range. Example keywords were `election` (popular around November 2016) and `happy` (popular around January 2017).

We built a full-text index on the *message* field to speed up these keyword-search queries. We used the "*create_at*" field as the slicing attribute, which was the creation timestamp of a tweet. We also used this field to do filtering for the AsterixDB indexes [22], which can prune irrelevant disk components if a query has a range predicate on the attribute. Therefore, a mini-query with a short range condition on the *create_at* field ran faster than one with a bigger range. We assume the user was more interested in the latest results, so the slicing was moving from the latest time to the earliest.

B. Effect of Different Slicing Methods

For comparison purposes, we implemented the following methods to run a query: (1) No slicing ("NS"): we ran the original query as it was; (2) Fixed-length slicing ("FL"): we used a fixed interval size for the slicing attribute in each mini-query; (3) Baseline Drum ("DRUM-BL"): we used the Drum framework without an error model, and used a linear regression function to directly compute a predicate range for the next mini-query without considering the penalty of missing the next milestone; (4) Drum with a histogram error model ("DRUM-HS"): we used a histogram error model (Equation 6) to decide the next predicate range; and (5) Drum with a Gaussian error model ("DRUM-GA"): we used a Gaussian error model (equation 9) to decide the next predicate range. All Drum methods started with 3 mini-queries with a fixed range of one, two, and four hours respectively to accumulate the initial statistics and then began the adaptive process. For each query, we ran each slicing method three times and computed their average performance numbers. We set the required response rhythmic pace to be 2 seconds. As explained in Section I, a good fixed length is hard to decide for the FL method. To give it a fair and comparable setting, we used the average number of mini-queries obtained from the DRUM-BL method for our different keywords.

Number of mini-queries: We first tested the different methods to explore their number of mini-queries. We considered four queries with different keyword conditions, namely keyword `rain` (uniform), keyword `election` (non-uniform), keyword `happy` (non-uniform and expensive), and *NoKeyword* (uniform and expensive). The results are shown in Fig. 11a. The NS method sent one big query directly to the database, so its number of mini-queries was always 1. The FL method used a fixed interval of 29 hours and issued 60 mini-queries regardless of the keyword condition. In contrast, the Drum framework dynamically adjusted the number of mini-queries based on the observed performance numbers of the database system. For example, keywords `election` and `rain` were very selective, and the corresponding original queries would access fewer records than the query of the popular keyword `happy`. The Drum framework utilized the collected information and picked a bigger predicate range for the two queries. As a result, their numbers of mini-queries were much less than that of the `happy` query and of the query without keywords. Among the three Drum methods, the DRUM-HS and DRUM-GA methods considered the uncertainty of the regression and thus were more conservative when generating the next predicate range and issued more mini-queries than DRUM-BL.

Total running time: Fig. 11b shows the total running time of the schedules generated by different models. The NS method had the least total running time because it did not do any slicing and thus did not pay any overhead. The FL method sent 60 mini-queries and had a slightly longer total time. The time of each DRUM method was comparable; these methods did not increase the running time much.

Total delay of missing milestones: We next evaluated the different methods regarding their total delay time. The results are shown in Fig. 11c. The NS method had the longest delay because it did not give the user any updates until the whole query was finished. The delay time of FL was always 0 for the `election` and `rain` requests, as each

of their mini-queries used a small fixed range and could finish before the next milestone. However, the same fixed range setting caused mini-queries of the expensive requests happy and *NoKeyword* to miss their deadlines. This result illustrates the difficulty of choosing a single range that works well for different requests. Compared to the DRUM-BL method, DRUM-HS and DRUM-GA used the histogram and Gaussian distribution uncertainty models to measure the penalty of missing a deadline, and hence they were more conservative regarding the range of each mini-query. As a result, the delays of these two methods were much smaller.

Total cost: Fig. 11d shows the overall cost of a schedule generated by each method as defined in Equation 1, where the weight was set to $\alpha = 25$ to reflect a case where the client is concerned more about the pace of result delivery than the total running time. The NL method had the highest cost due to its long delay. The cost of FL was also significant for the two expensive queries happy and *NoKeyword*. The Drum methods had lower costs for all the queries because of the low penalty of their generated schedules. The DRUM-HS and DRUM-GA methods were able to balance the number of mini-queries and the risk of missing milestones, so their costs were the lowest.

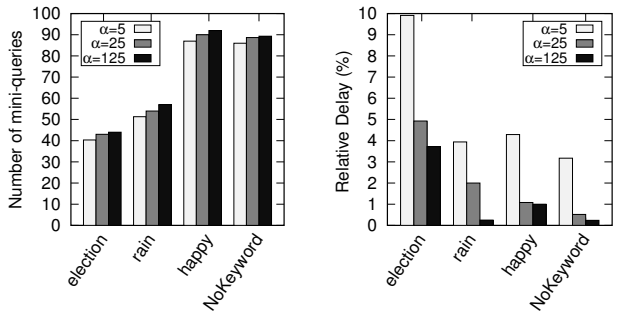
These experiments showed that there was no much difference between the schedules generated by DRUM-HS and DRUM-GA. The main reason was that Drum is an adaptive framework, and it can automatically adjust the regression function based on observed performance numbers, so the estimation error distribution was similar to Gaussian.

C. Effect of Penalty Weight α

We evaluated the effect of the penalty weight α in the scoring function of Equation 2. Intuitively, a higher α value means a higher penalty for missing deadlines, and the slicing method should become more conservative in generating the next predicate range. We did experiments using three different values for α, namely 5, 25, and 125, for both DRUM-HS and DRUM-GA. Fig. 12a shows the number of mini-queries when using the DRUM-HS method. When α increased, the number of mini-queries also increased since each mini-query became more conservative (with a smaller range). Meanwhile, the percentage of delay in the overall time decreased due to the smaller predicate ranges, as shown in Fig. 12b. The DRUM-GA method has similar results.

D. Adaptiveness of Regression Function

To evaluate the adaptiveness of the regression function, we considered two ways to derive the linear regression function: (1) All, which used the observed performance numbers of all previous mini-queries; and (2) Last10, which used the latest 10 performance numbers. The purpose here was to see how the Drum framework can adapt to changes in the performance of the underlying database system. We

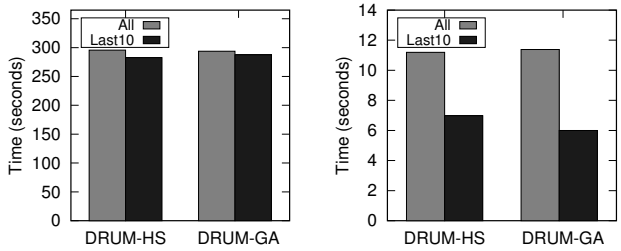


(a) Number of mini-queries (b) The percentage of delay

Figure 12: Effect of penalty weight α (DRUM-HS)

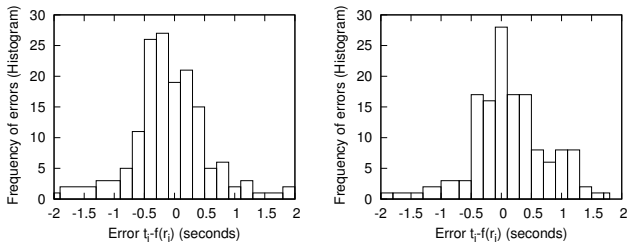
considered the *NoKeyword* query, whose query results were distributed evenly over the entire time range.

We first simulated a situation with a single sudden change of the database performance. The first half of the mini-queries were run normally, but at that point in time we added a 1-second sleep to the database connection before sending the result back to the mini-query executor in order to simulate a case where the second half of the mini-queries took longer than the mini-queries with the same time ranges in the first half. Fig. 13 shows the results. The Last10 method was able to adapt to the new database performance much faster since its obtained linear regression better described the relationship between the predicate range and running time. The All method did not adapt to the changes well and there were more underestimated predictions as shown in Fig. 14.



(a) Total running time (b) Total delay

Figure 13: Adaptiveness of the linear regression (adding a 1-second sleep for the second half of the mini-queries)



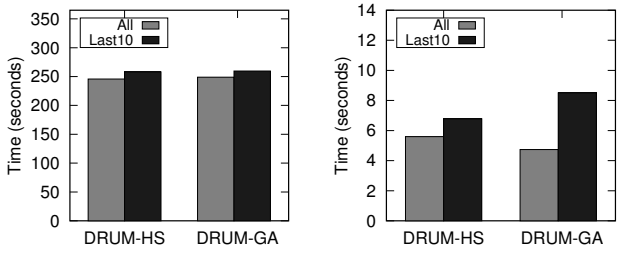
(a) Last10 method (b) All method

Figure 14: Error distribution of the predication model by changing the length of learning history.

We then simulated a situation where the database system was constantly not stable, where we added a 1-second sleep randomly (with probability of 0.5) in the database connector for each mini-query. Fig. 15 shows the results. In this case,

the linear function trained using the All method was more precise, and both its total running time and penalty were less than those for the Last10 method.

These two experiments showed that using fewer recent observations to train a linear function can more quickly adapt to new backend performance changes. However, if the underlying database is unstable, using fewer observations may more likely be affected by "noise," making the trained model less precise, which could lead to worse performance (e.g., a longer running time and bigger penalty).



(a) Total running time (b) Total delay

Figure 15: Adaptiveness of the linear regression (adding a 1-second sleep for each mini-query randomly)

Remarks: In summary, we have seen that (1) the Drum framework can adaptively adjust the size of each mini-query to successfully reduce the amount of delay for different queries without any prior knowledge, and it does not increase the total running time by much. (2) The uncertainty models can automatically balance the number of mini-queries and the penalty of missing milestones depending on the weight on the penalties. (3) The Gaussian model and histogram model gave similar results. Since the Gaussian model is easier to implement, it is arguably the better choice to adopt.

VI. Conclusions

In this paper, we have studied how to progressively answer a time-consuming query on a large data set by generating a sequence of mini-queries. We formulated an optimization problem to produce the predicates of mini-queries by considering both their total running time as well as the smoothness of result delivery, the key goal being to provide the incremental results at a rhythmic pace to improve the user experience. We developed an adaptive framework called Drum that can collect run-time behavioral statistics for the database system to decide the predicate for the next mini-query appropriately. Drum is a general middleware solution that requires no changes to the underlying database system. Our experiments on a large, real data set showed that Drum and its techniques can reduce the delay of delivering intermediate results to the user without sacrificing much total time. Therefore, the user can see smooth result updates at the expected rhythm. In the future, we will extend the techniques to do slicing on multi-dimensional and categorical fields to support more progressive display options.

Acknowledgments

This work has been supported by NIH award 1U01HG008488-01, NSF CNS award 1305430, and the Army Research Laboratory under Cooperative Agreement No. W911NF-16-2-0110.

References

[1] J. M. Hellerstein *et al.*, "Online Aggregation," in *ACM SIGMOD*, 1997.

[2] P. J. Haas *et al.*, "Ripple Joins for Online Aggregation," in *ACM SIGMOD*, 1999, pp. 287–298.

[3] J. M. Hellerstein *et al.*, "Interactive data analysis: The Control Project," *IEEE Computer*, vol. 32, no. 8, pp. 51–59, 1999.

[4] N. Pansare *et al.*, "Online Aggregation for Large MapReduce Jobs," *PVLDB*, vol. 4, no. 11, pp. 1135–1145, 2011.

[5] V. Raman *et al.*, "Partial results for online query processing," in *ACM SIGMOD*, 2002, pp. 275–286.

[6] T. Condie *et al.*, "MapReduce Online," in *USENIX Symposium on NSDI*, 2010, pp. 313–328.

[7] I. Lazaridis *et al.*, "Progressive Approximate Aggregate Queries with a Multi-Resolution Tree Structure," in *ACM SIGMOD*, 2001, pp. 401–412.

[8] Y. Tao *et al.*, "Spatio-Temporal Aggregation Using Sketches," in *ICDE*, 2004, pp. 214–225.

[9] K.-L. Tan *et al.*, "Query rewriting for SWIFT (first) answers," *IEEE Trans. Knowl. Data Eng.*, vol. 12, no. 5, 2000.

[10] S. Egger *et al.*, "Waiting times in quality of experience for web based services," in *QoMEX*, 2012, pp. 86–96.

[11] Y. Skadberg *et al.*, "Visitors' flow experience while browsing a Web site: its measurement, contributing factors and consequences," *Computers in Human Behavior*, vol. 20, 2004.

[12] N. Bhatti *et al.*, "Integrating user-perceived quality into web server design," *Computer Networks*, vol. 33, no. 1, 2000.

[13] J. Nielsen, "Usability Engineering," in *The Computer Science and Engineering Handbook*, 1997, pp. 1440–1460.

[14] M. Fiedler *et al.*, "State-of-the-art with regards to user-perceived Quality of Service and quality feedback," *Euro-NGI Deliverable D. WP. JRA. 6. 1. 1*, 2004.

[15] M. Ahmad *et al.*, "Predicting completion times of batch query workloads using interaction-aware models and simulation," in *ACM EDBT*, 2011, pp. 449–460.

[16] W. Wu *et al.*, "Towards Predicting Query Execution Time for Concurrent and Dynamic Database Workloads," *PVLDB*, vol. 6, no. 10, pp. 925–936, 2013.

[17] W. Wu *et al.*, "Uncertainty Aware Query Execution Time Prediction," *PVLDB*, vol. 7, no. 14, pp. 1857–1868, 2014.

[18] S. Agarwal *et al.*, "BlinkDB: queries with bounded errors and bounded response times on very large data," in *ACM EuroSys*, 2013, pp. 29–42.

[19] P. Xiong *et al.*, "ActiveSLA: a profit-oriented admission control framework for database-as-a-service providers," in *ACM SOSP*, 2011, p. 15.

[20] J. Jia *et al.*, "Drum: A rhythmic approach to interactive analytics on large data (full)," http://cloudberry.ics.uci.edu/img/query-slicing-long.pdf.

[21] S. Alsubaiee *et al.*, "AsterixDB: A scalable, open source BDMS," *PVLDB*, vol. 7, no. 14, pp. 1905–1916, 2014.

[22] S. Alsubaiee *et al.*, "LSM-Based Storage and Indexing: An old idea with timely benefits," in *ACM Workshop on Managing and Mining Enriched Geo-Spatial Data*, 2015, pp. 1–6.