UNIVERSITY OF CALIFORNIA,
IRVINE

Supporting Interactive Analytics and Visualization on Large Data

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Jianfeng Jia

2017

DEDICATION

To my family.

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF TABLES

ACKNOWLEDGMENTS

I am deeply grateful to my advisers Professor Chen Li and Professor Michael J. Carey for their continuous support and guidance during the past five years. Professor Li guided every aspect of my Ph.D. study from choosing my research topics, developing systems, to writing technical articles. His insightful vision made the Cloudberry project happen. His distinctive perspective and serious research procedure fundamentally changed my mindset and shaped me to be an independent researcher. Professor Carey is a knowledgeable system researcher who has been a walking encyclopedia and led me to a more enlightened research path. His carefulness on every detail of experiment analysis as well as the enthusiasm in every potential performance improvement educated me how to become a serious system builder. Without them, I could not have finished this thesis.

I would like to thank Professor Sharad Mehrotra for joining my doctoral committee. His knowledge and insightful comments have always strengthened my work.

I would like to thank Ian Maxon for proof-reading the draft version of the thesis and providing invaluable suggestions from structure to wordings.

I would like to thank Dr. Simon Su, Micheal An, and Vincent Perry from the Army Research Lab for building their multi-display visualization application on top of the Cloudberry system and authorizing us to use their demo picture in the thesis.

I would like to thank Chen Li (the student version) and Zhang Xi for contributing the early prototype of Cloudberry. With their help, we won the Best Visualization Award in the UCI Data Hackathon, which gave a high confidence on the direction of the system at the early stage.

I would like to thank Yingyi Bu for his invaluable suggestions and discussions during my research. His idea directly inspired the work in Chapter 5.

I would like to thank Mengfan Tang for his great help on the mathematical solutions in Chapter 4.

I would like to thank all other Cloudberry team members and alumnus: Taewoo Kim, Chen Luo, Te-Yu Chen, Dharini Sreenivasan, Nishad Gurav, Aishwarya Kapse, Kaiyi Ma, Zongheng Ma, Monique Moledo, Victor Phung, Hao Chen, Vignesh Sankar, Shengjie Xu, Liangju Chu, Sicong Liu, Vidhyasagar Thirumaraiselvan, Haohan Zhang, Qiancheng Wu, Qiushi Bai, and Tejia Zhang. I would like to thank the Professor Wenhai Li and Professor Hui Zhang who tried Cloudberry and gave us feedback.

I would like to thank the AsterixDB team including Dr. Till Westmann, Dr. Alex Behm, Dr. Raman Grover, Dr. Sattam Alsubaiee, Pouria Pirzadeh, Ian Maxon, Abdullah Alamoudi, Inci Cetindil, Taewoo Kim, Zach Heilbron, Preston Carman, Ildar Absalyamov, Steven Jacobs, Madhusudan C.S, Khurram Faraaz, Kereno Ouaknine, Xikui Wang, Chen Luo, Professor Heri Ramampiaro, and Professor Wenhai Li for working hard together to make the system stable and reliable.

I would like to thank my wife Hangyan for every day we have spent together, for her love, support, care, and understanding through these years. I thank my daughter Jasmine for filling my life with love, joy, and happiness. I thank my parents Zhongqi and Faping for their love and continuous encouragement. Their sacrifices allowed me to pursue this path and come this far.

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

CURRICULUM VITAE

Jianfeng Jia

EDUCATION

Doctor of Philosophy in Information and Computer Science **2017**
University of California, Irvine *Irvine, California*

Master of Science in Computer Science **2008**
Xiamen University *Xiamen, China*

Bachelor of Science in Computer Science **2005**
Xiamen University *Xiamen, China*

SELECTED HONORS AND AWARDS

Google Graduate Student Award in ICS **2017**
University of California, Irvine

Best Visualization Demo Award in UCI Data Science Hackathon **2016**
University of California, Irvine

PUBLICATIONS

Drum: A Rhythmic Approach to Interactive Analytics on Large Data **2017**
IEEE International Conference on Big Data (IEEE Big Data)

Caching Geospatial Objects in Web Browsers **2017**
International Conference on Advances in Geographic Information Systems

Visual Analytics Ecology for Complex System Testing **2017**
Visualization in Practice at IEEE VIS

Twitter Coverage of Climate Change and Health before and after the 2016 US Presidential Election **2017**
APHA Annual Meeting

Towards Interactive Analytics and Visualization on One Billion Tweets **2016**
International Conference on Advances in Geographic Information Systems

Pregelix: Big(ger) Graph Analytics on A Dataflow Engine **2014**
Proceedings of the Very Large Database Endowment (PVLDB)

ABSTRACT OF THE DISSERTATION

Supporting Interactive Analytics and Visualization on Large Data

By

Jianfeng Jia

Doctor of Philosophy in Computer Science

University of California, Irvine, 2017

Professor Chen Li, Chair

There is an increasing demand to visualize large datasets as human observable reports in order to quickly draw insights and gain timely awareness from the data. An interactive user interface is an indispensable tool that allows users to analyze the data from different perspectives and to inspect the result from the global overview to the finest granularity. To enable this type of interactive user experience, the frontend needs to generate new requests on the fly, and the results must be computed and delivered within seconds. Big Data platforms can take tens or hundreds of seconds to complete an OLAP-style query, so there is a need for a solution that can meet the stringent latency requirement of interactive visualization frontends.

In this thesis, we address the interactivity challenges from a middleware perspective to provide a generic solution that can utilize existing database systems as a "black box" to support various interactive visualization applications efficiently.

We first present Cloudberry, an open-source general-purpose middleware system to support interactive analytics and visualization on big data with various attributes. It can automatically create, maintain, and delete materialized views by analyzing each request and its results. By utilizing materialized views stored in the backend database, it can reduce the query response time remarkably. We build an application called "TwitterMap" using Cloudberry to demonstrate its suitability to support interactive data analytics and visualization on more than one billion tweets (about 2TB).

We then present a query-slicing technique in Cloudberry, called Drum, that can "slice" a query into small pieces (called "mini-queries") so that the middleware can send these mini-queries to the DBMS one by one and compute results progressively. The Drum framework can collect run-time behavioral statistics for the database system to decide the predicate for the next mini-query so that it can appropriately provide a smooth user experience. Moreover, Drum is a general technique in Cloudberry that requires no changes to the underlying database system. Our experiments on a large, real dataset show that the Drum technique can reduce the delay of delivering intermediate results to the user without much reduction of the overall speed for the complete query answer.

To further speed up Big Data visualization, we present a method of using LSM filters to accelerate secondary-to-primary index search in an LSM storage setting. This technique can propagate the filter values from the secondary index to speed up the primary index search. We implemented it in AsterixDB, and our experiments show that the new approach can reduce the query time by 20% to 70% for different queries with various selectivities. With this improvement, Cloudberry can generate mini-queries that contain much wider range predicates, which can reduce the cost of the progressive query evaluation.

Chapter 1

Introduction

We are living in an era where a significant amount of digital information is generated on a daily or even hourly basis through social networks, blogs, online communities, news sources, and mobile applications. This massive amount of data holds valuable information that can be used for various analytical purposes, e.g., to track social opinions on different topics, trends of a specific event, etc. There are increasing demands to visualize the large datasets as human observable reports in order to quickly draw insights and gain timely awareness from the data. Most importantly, an interactive user interface is an indispensable tool that allows users to analyze the data from multiple perspectives and to inspect the results from the global overview to the finest granularity.

To achieve the interactivity, the frontend needs to generate new requests on the fly, and the results must be computed and delivered in seconds or even in milliseconds. Many analytics systems (e.g., Superset [9], Tableau [87]) are connecting to databases directly and send new database queries once users have new actions in the frontend. Consequently, the responsiveness depends on the query execution time of the underlying database. Although nowadays Big Data systems (e.g., Apache AsterixDB [13]) can manage large amounts of data, it can still take tens or hundreds of seconds to answer an OLAP-style query. There is a need for a backend service that can meet the stringent latency requirements of interactive visualization in frontend applications.

In this thesis, we address the interactivity challenges from a middleware perspective to provide a general solution that can utilize existing database systems to support various interactive visualization applications efficiently. In particular, we study three different problems and provide an efficient solution for each of them.

Design a General-Purpose Middleware System

An analytical application often issues a series of queries where the results of the previous queries affect the formulation of the next ones. For example, an analyst may want to find how social networks respond to events such as the Zika virus. The analyst may first want to get a quick overview of the distribution of the tweets that mention Zika. Once they have found that Zika was discussed more frequently in a particular region, e.g., Florida, the analyst may want to drill down to the Florida region to further explore the time and spatial patterns. The later queries share the same interest in Zika as the first one. This pattern presents a chance to speed up new queries by using the previous query results. For example, if we can store all the Zika-related tweets in a materialized view after the first query, the future zoom-in queries will only need to visit the view without touching many irrelevant records in the main dataset. It is desirable to have a system that can automatically collect the previous results and reuse them efficiently for solving related queries.

Chapter 3 presents Cloudberry, an open-source general-purpose middleware system to support interactive analytics and visualization on large data with various attributes. It can automatically create, maintain, and delete views by analyzing each request and its results. By utilizing materialized views stored in the database, it can reduce the query-response time significantly. Furthermore, by implementing the main logic in the middleware layer, it is a generic solution that can connect to different types of frontend applications as well as various database systems with the corresponding connectors.

Progressively Solving Queries in Middleware

Some queries cannot be answered using materialized views, in which case executing the query over the entire dataset is inevitable. In order to guarantee the responsiveness of the frontend, we present query-slicing techniques in Cloudberry in Chapter 4, called Drum[1], which can slice a query into small pieces (called "mini-queries") so that the middleware can send them efficiently one by one and compute results progressively. For example, suppose the first request that the user sends to Cloudberry is "show the number of tweets mentioning the keyword `zika` for each state". Instead of directly sending one query to the database system, Cloudberry can issue a sequence of cheaper "mini-queries" by adding a predicate on the time period, e.g., "show the number of tweets mentioning the keyword `zika` for each state from January 2017 to February 2017", so that each of them can access a small amount of data and could finish faster. Cloudberry can merge the results as needed and send them to the frontend progressively. Then the frontend interface can be updated more quickly, which leads to a more responsive user experience. 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, the key goal being to provide the incremental results at a rhythmic pace to improve the user experience. The Drum framework can collect run-time behavioral statistics for the database system to decide the predicate for the next mini-query appropriately. Moreover, Drum is a middleware solution that requires no changes to the underlying database system.

Using Filters to Improve Secondary-to-Primary Index Search

Besides the enormous size, data can be created and ingested at a high rate in real-time.

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

To accommodate this kind of insert-intensive workload, popular NoSQL systems such as HBase [1], Cassandra [22], LevelDB [54], and BigTable [23] have adopted LSM-trees (log-structured merge-trees) [63]. LSM-trees amortize the cost of writes by batching updates in-memory before writing them to disk, thus avoiding random writes. This benefit comes at the cost of possibly sacrificing read efficiency. To improve read performance, AsterixDB implemented LSM filters [15] as a way to add additional metadata to LSM components and to exploit that metadata during query processing to skip irrelevant components. The filter feature plays an important role in optimizing mini-query performance. If the original query is split on the same dimension as the filter field, it can significantly improve query performance since most of the irrelevant components can be skipped. In order to efficiently utilize the power of the filter, the mini-query needs to have a highly selective filter related predicate. This limitation largely restricts the scenarios that can utilize the pruning power of the filter, as many analytical queries do not match this pattern. For example, a mini-query can have a wide time range predicate, and then many index components will be matched based on their filters. In Chapter 5, we visit the filter idea, explore the aligned structure property between secondary index components and primary index components, and develop an improved version of the secondary-to-primary search. The main idea is to propagate the filter attached to the secondary index components along with their output primary keys to the primary-index search. Then the primary-index search can use the filter hints to improve the search speed, even though the query itself does not contain any filter-related predicates.

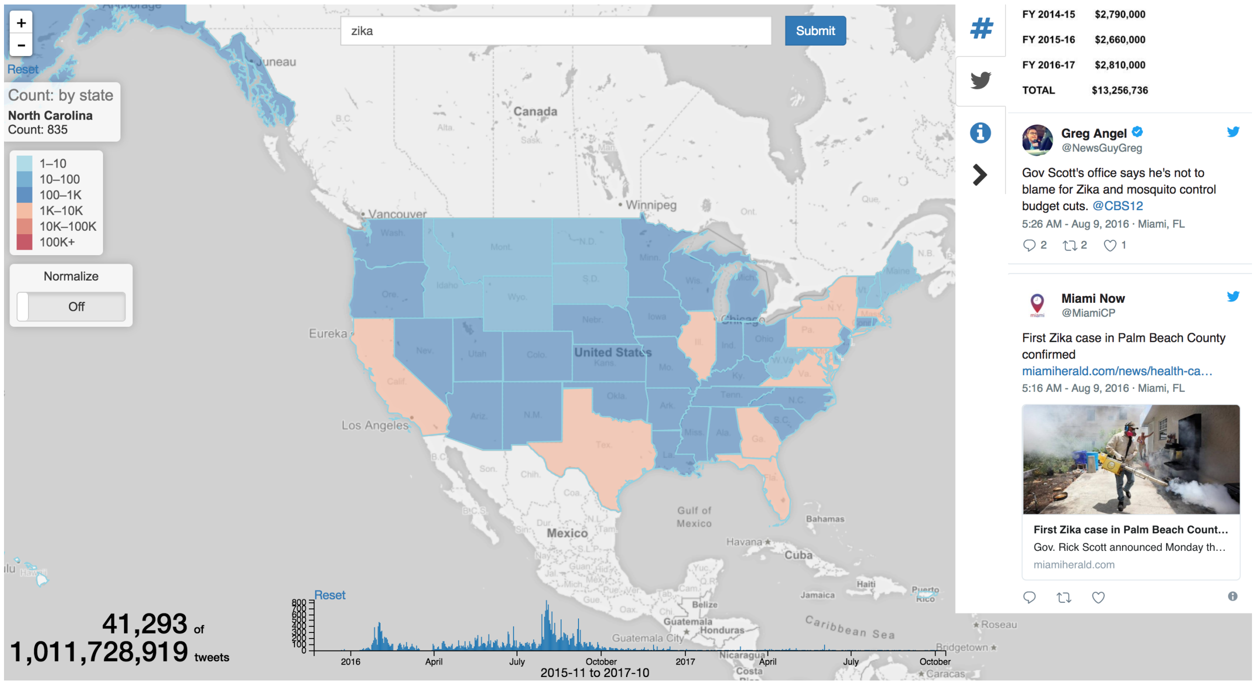
The thesis is organized as follows. We first present an analytics web application example of Cloudberry, called TwitterMap, that allows users to interactively query, analyze, and visualize more than one billion tweets in Chapter 2. Then we introduce the design and implementation of Cloudberry in Chapter 3. In Chapter 4 we present the Drum framework to show how to progressively answer a time-consuming query on a large dataset by generating a sequence of mini-queries. In Chapter 5 we visit the filter idea and present a method of using LSM filters to accelerate secondary-to-primary LSM index search in AsterixDB to reduce the response time of a mini-query. Chapter 6 concludes the thesis and outlines a few directions for future work.

Chapter 2

TwitterMap: Interactive Visualization on One Billion Tweets

We have developed a live visualization system called “TwitterMap” [84] that allows users to interactively explore and visualize more than one billion tweets by specifying temporal, spatial, and textual conditions in real time. In this chapter, we will introduce how TwitterMap supports interactive analysis and visualization to illustrate the requirements of the visualization system.

Fig. 2.1 shows the user interface of TwitterMap. The system follows the “overview first, zoom and filter, then details on demand” [74] design principle. A user can type keywords (e.g., “zika”) to see the spatial, temporal, and hashtag distributions of the related tweets. After identifying a hot area, time range, or hashtags of interest, the user can further drill down or zoom out to different levels, including states, counties, and cities. The system also lists relevant tweets in a sidebar in their reverse chronological order so that the user can review the finest details of the tweets.



zika

41,293 of
1,011,728,919 tweets

FY 2014-15	$2,790,000
FY 2015-16	$2,660,000
FY 2016-17	$2,810,000
TOTAL	$13,256,736

Greg Angel
@NewsGuyGreg
Gov Scott's office says he's not to blame for Zika and mosquito control budget cuts. @CBS12
5:26 AM - Aug 9, 2016 · Miami, FL
2 2 1

Miami Now
@MiamiCP
First Zika case in Palm Beach County confirmed
miamiherald.com/news/health-ca…
5:16 AM - Aug 9, 2016 · Miami, FL
First Zika case in Palm Beach Count…
Gov. Rick Scott announced Monday th…
miamiherald.com

Figure 2.1: TwitterMap interface when a user searches "Zika" on tweets.

2.1 Architecture

Fig. 2.2 shows the four-tier architecture of the application as a full-stack analytics and visualization solution. It includes a modularized web interface, a web server to interpret the user action description into the corresponding Cloudberry JSON request, the Cloudberry middleware to rewrite the request into backend queries, and an AsterixDB cluster to continuously ingest, store, and query data in parallel. The four parts together enable an interactive, real-time visualization UI on a large-scale Twitter dataset.

2.2 Frontend Interface

The frontend of TwitterMap is a single web page developed in HTML, Angular JS, and CSS. It displays the spatial and temporal distribution of tweets that contain user-specified keywords on both a map and a timeline dashboard. The interface allows users to zoom into

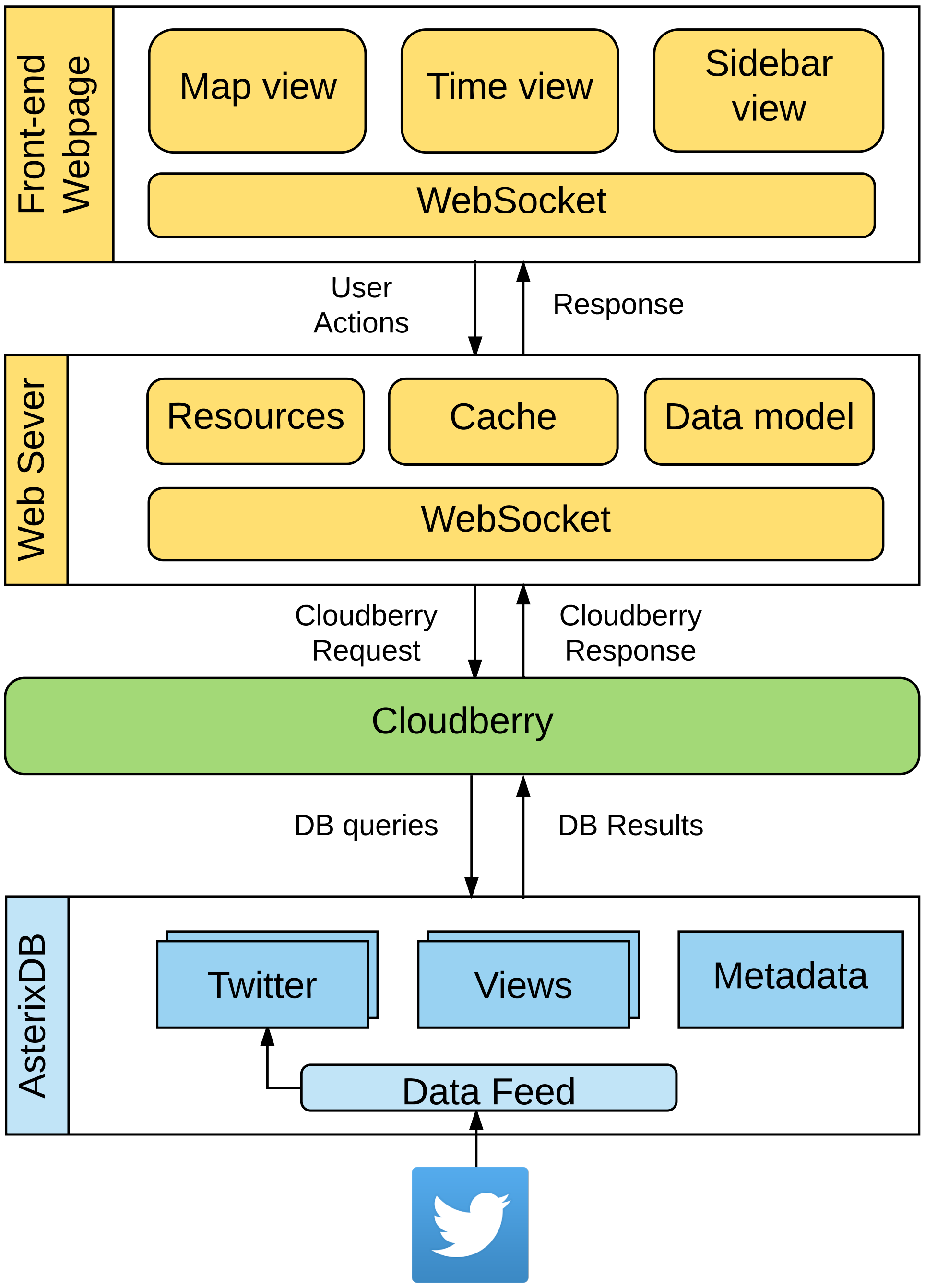


Figure 2.2: TwitterMap architecture.

```
{
 "action": "move",
 "from": [[-117.826505,33.684567], [-115.139830,36.169941]],
 "to": " [[-117.396156,33.953349], [-115.112997,36.106965]],
 "mapLevel": "county",
 "keywords": ["zika"],
 "time": ["2015-10-00T00:00:00", "2017-10-06T17:02:18"]
}
```

Figure 2.3: A frontend web JSON record to indicate that the user has a “move” action.

finer details on both spatial and temporal dimensions. Whenever the zoom-in level changes on either dimension, the frontend will issue a new request to the web server and render the results on the UI. By using a WebSocket connection, the UI is continuously updated as new results are received from the web server. Fig. 2.3 shows an example JSON record that describes that a “move” event has happened on the web page. It contains the geographic bounding box (left-bottom and right-top coordinates) of the map canvas, the keyword that was input, and the time range that was selected on the UI.

2.3 Frontend Web Server

The web server launches the frontend web page and provides essential resources such as a GeoJSON file of the state, county, and city polygons. It also receives the frontend actions then interprets them as corresponding data request(s) to the Cloudberry system.

Take the “move” action in Fig. 2.3 as an example. The web server will first check the overlap relationship between the stored county shapes with the requested boundary to generate a list of matched county ids. Then it will generate four Cloudberry requests to ask for per-county and per-day count aggregations, top-50 hashtags, and the latest sample tweets.

We will introduce how to compose the Cloudberry requests later in Section 3.3.3.

2.4 AsterixDB Cluster

Apache AsterixDB is used to store a large number of tweets and views, and provide a fast query engine to run queries efficiently. It is a scalable big data management system that supports a variety of indexes such as B-tree, R-tree, and inverted index to support filtering operations without scanning entire datasets. It has a built-in parallel runtime query execution engine, Hyracks, to scale up to hundreds of machines so that an aggregation query can finish with a low latency. The view manager in Cloudberry communicates with AsterixDB server via SQL++ queries.

In addition to query processing, AsterixDB provides a "data feed" feature that accepts continuous data into the database. The TwitterMap system uses AsterixDB's built-in socket feed adapter to get streaming tweets while applying a geo-tagging function to annotate the named locations of each tweet. In this way, users can query the newest tweets in real-time.

We set up the whole system on a cluster of five Intel NUC machines (shown in Fig. 2.4). Each machine has four cores, 16 GB of memory, and a 500GB SSD disk, and each runs an Apache AsterixDB node. The cluster costs less than $4,000. The data set is collected using the Twitter streaming API for the North American area. We have already collected data for about two years since November 2015, yielding more than one billion tweet records (about 2TB in size), and are still receiving new data at a rate of about 20 tweets per second.

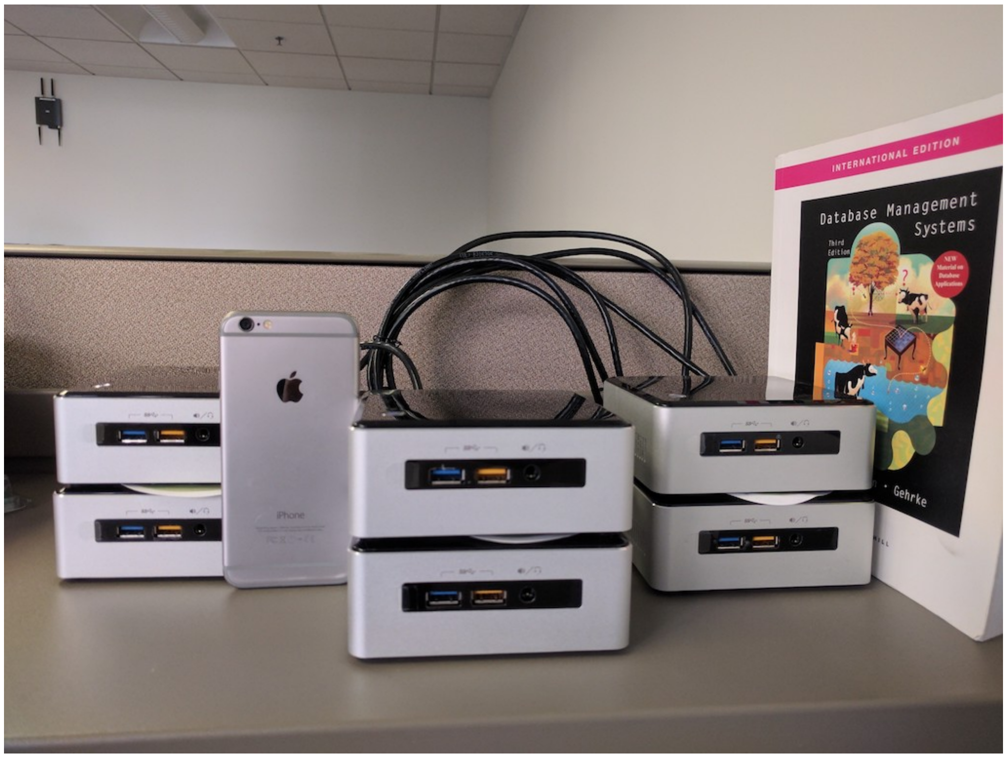


Figure 2.4: The $4,000 TwitterMap cluster supporting interactive visualization on 2TB data.

2.5 Tweet Analytics Demonstration

2.5.1 Initial Map View

Fig. 2.1 shows the primary Web interface, which consists of a U.S. geographic map, a time chart, and a sidebar with rich contextual information. The user types keywords into the search box, which are used as a filter condition in the frontend action request sent to the web server. The web server translates the user's actions into a request in the Cloudberry JSON format and sends it to the Cloudberry middleware system. After the response comes back, the map displays the geographic tweet-count distribution for different states. The time chart shows the number of tweets per day, and the sidebar shows the top-50 frequent hashtags in the tweets and also the details of the latest sample tweets. For instance, Fig. 2.1 shows that there were over 41 thousand tweets mentioning "zika" out of one billion tweets.

The map figure illustrates how "zika" is distributed in different states. The color of each region illustrates the "hotness" of the topic in the region. The time chart shows how "zika" is distributed at different days. We can see there is a spike in August 2016 when the Olympic Games in Rio de Janerio, Brazil took place and there was a lot of global concern about the Zika epidemic.

2.5.2 Normalized Count

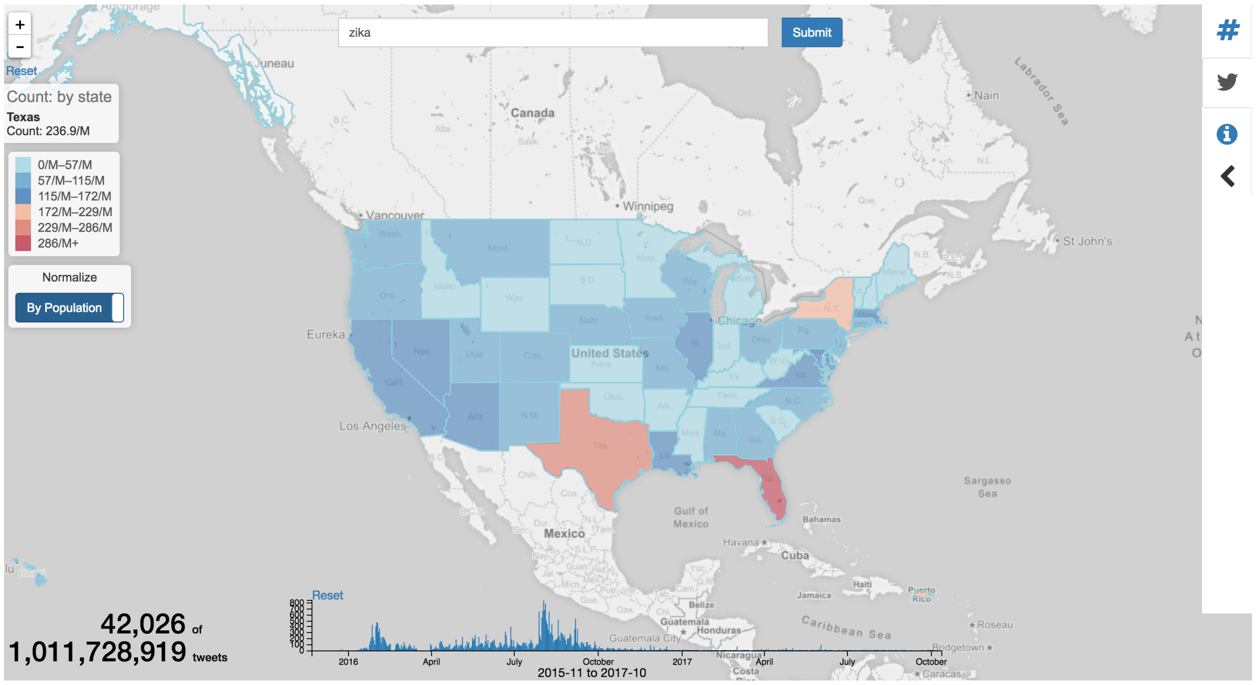


Figure 2.5: Normalized distribution.

One limitation of using the absolute count to show the map result as shown in Fig. 2.1 is that the "hotness" states are often the populous ones, e.g., California, Texas, New York, etc. We provide a "normalization" view feature that can show the number of tweets per capita in the region to show the relative "hotness" among different places.

The feature is implemented by using the "lookup" request in the Cloudberry, which can augment the results with additional fields from another dataset. Specifically, the aggregation records for each region contain additional "population" fields whose values are obtained from a "US population" dataset.

Fig. 2.5 shows the normalized distribution. In the figure, we can see that "zika" is discussed more frequently per capita in Florida than other states. This result may due to the fact that there were many reported "zika" cases in Florida, so people there were tweeting more about "zika" than in other regions.

2.5.3 Zooming In

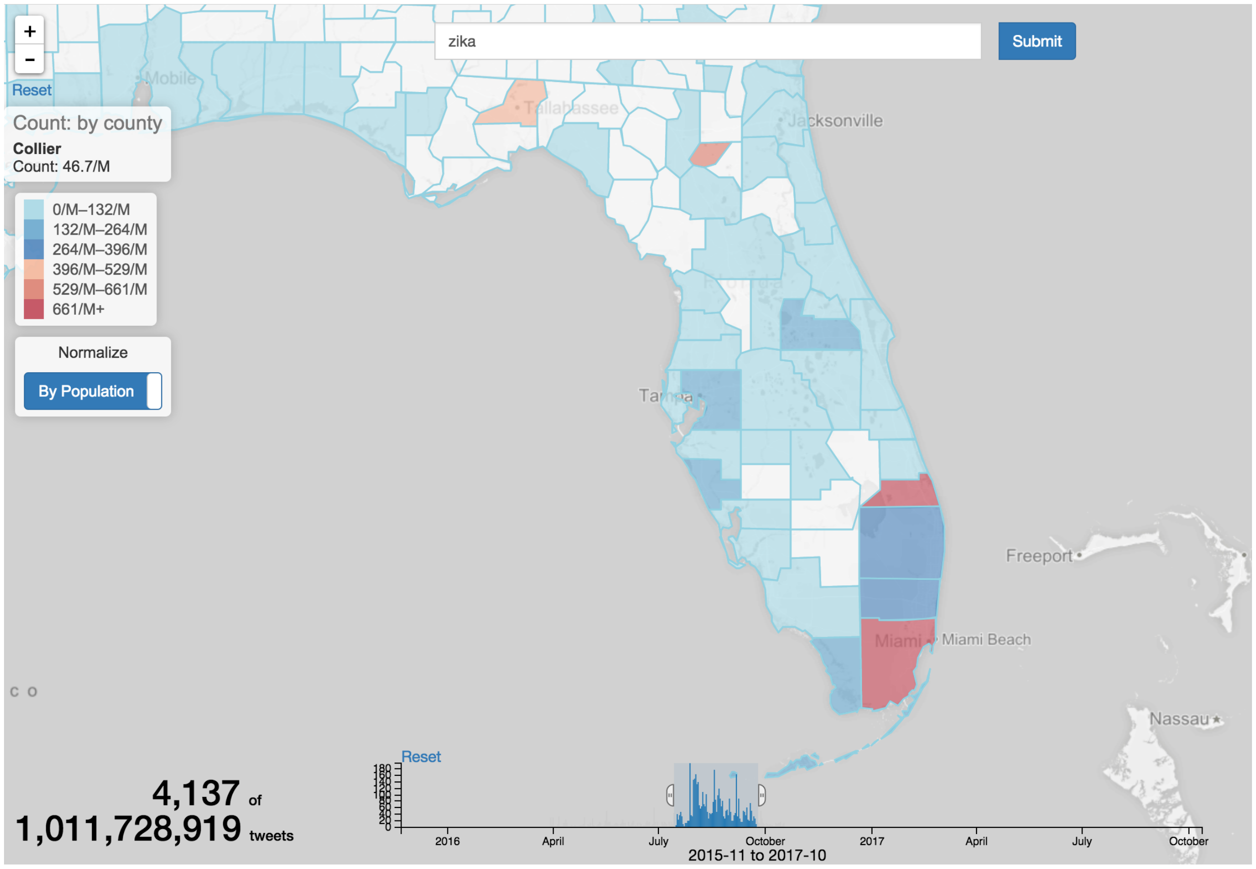
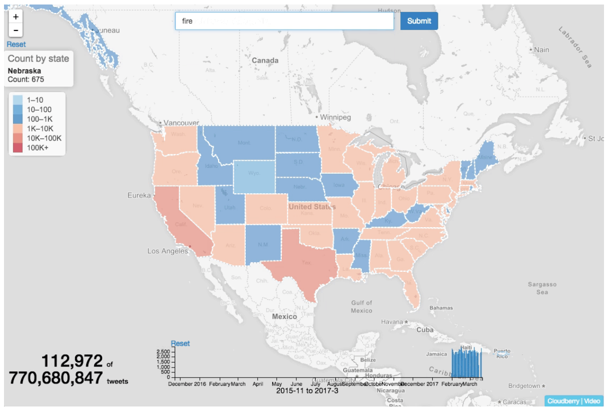


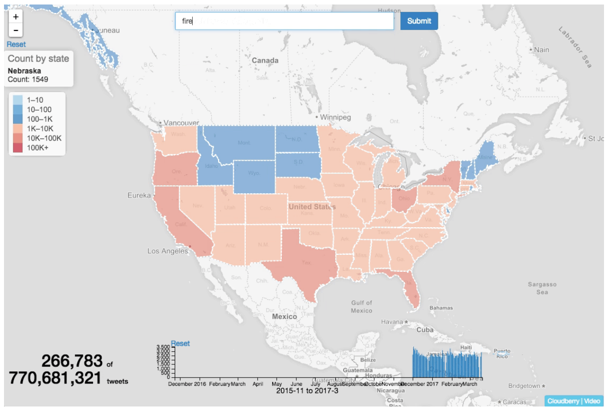
Figure 2.6: Zoom into county-level details in Florida.

TwitterMap allows users to explore multi-granularity data. Fig. 2.6 shows a user zooming into the Florida area after the finding that it is the hottest state talking about "zika". The map section automatically switches to the county-level distribution. There are two hot-spots, in Miami and Martin counties. The time chart is updated to show the corresponding patterns of tweets published in the zoomed-in area. The user can further select an interesting period to add another predicate in the time dimension to focus on tweets issued within that period. Under the hood of the Web interface, every frontend action triggers an HTTP request to the web sever and finally to Cloudberry to get the required data. Thus, the response time is a critical factor to achieving an interactive user experience.

2.5.4 Query Slicing



(a) The first response returns the result of the query on the dataset since February 2017



(b) The second response returns the result from last December to now

Figure 2.7: Example of using query slicing

When there is no view to be utilized, scanning the entire dataset is inevitable, which may result in a long waiting time in the frontend application. To improve the interactiveness of the user experience, Cloudberry provides a *query slicing* feature to return a stream of quick responses. The query-slicing strategy decomposes a single large query into a sequence of *mini-queries* by adding a series of range predicates on the time dimension to the original query. In this way, each of the mini-queries can deliver results sooner so that it can provide a better interactive user experience. Inside the middleware system, the partial results will be merged and sent to the client as a stream of progressive results. TwitterMap uses this feature by specifying the option "`slicingMills:2000`" in the request to Cloudberry to indicate that it accepts partial results and that the expected interval per result is 2,000 milliseconds. We will present the details of this feature in Chapter 4.

From the perspective of the frontend web page, there is a stream of results from the server. TwitterMap uses a feature called "watch" in the AngularJS library [17] to continuously update the UI when the new results are coming. Fig. 2.7 shows the effect of the UI changes. A user is asking for the distribution of the tweets that mention "fire". Since there is no view available, the query will be answered progressively. The first response shown in Fig 2.7a returns the distribution of the tweets published since February. After two seconds, another response that contains the aggregation results from last December is returned, and the frontend updates the interface with the new results. (Though the results are not complete, they already convey much more information, e.g., how many tweets per day, which states discuss more about "fire", etc.) In this way, users can receive progressive results as soon as possible without waiting for query completion.

2.6 Summary

In this chapter, we presented the TwitterMap application that can allow users to interactively explore and visualize more than one billion tweets by specifying temporal, spatial, and textual conditions in real time. Besides its powerful and friendly interface for users to visualize information on a map, it has several unique capabilities that are enabled by Cloudberry and the underlying AsterixDB system: (1) Scalability: it can store, index, and query large amounts of information (e.g., billions of tweets). Besides the primary dataset of tweets, it can also access other datasets to augment the visualization results. (2) Interactivity: it can answer an analytical query efficiently (e.g., in sub-seconds). In the case where computing the complete result for the original query takes a long time, it can provide progressive results to give timely feedback on the user interface. (3) Real-time analysis: the tweets are ingested in real-time, and the system allows users to query the latest information as well as historical data.

In the following chapters, we will introduce the details of the underlying system to explain how it helps the TwitterMap implement those features.

Chapter 3

Cloudberry: A Middleware System to Support Interactive Analytics

3.1 Introduction

As discussed earlier, we are living in an era where a tremendous amount of digital information is being generated on a continuous basis through social networks, blogs, online communities, news sources, and mobile applications. This massive amount of data holds valuable information that can be used for various analytical purposes, e.g., to track social opinions on different topics, trends of a specific event, etc.

To understand what is in a dataset and the characteristics of the data, an analyst often uses data visualization or tabular reports to have an initial view of the data and a basic understanding of key characteristics. It is usually followed by drill-down or filtering of the data to identify anomalies or patterns recognized through a serial of visualization results [48, 49].

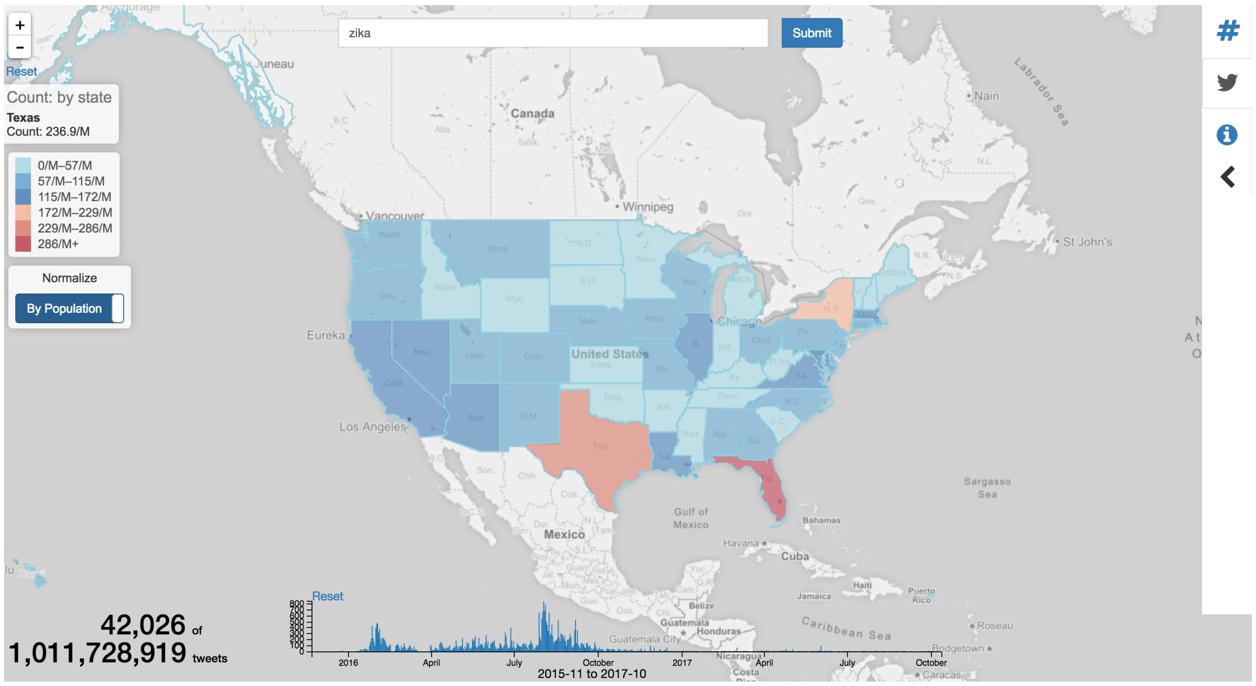


Figure 3.1: The spatial and temporal distribution of tweets that mention "Zika"

In the TwitterMap example, an analyst may want to find how social networks respond to events like epidemics, such as the Zika virus. The analyst may want to get a quick overview of the distribution of the tweets that mention "Zika". Fig. 3.1 shows a by-state and by-day distribution of those tweets on a web page. In the figure, we can see that Zika was discussed more frequently in Florida than in other states. Also, on the time bar chart, there is a high spike of tweets in August 2016. The analyst can drill down to the Florida region or the period of August 2016 to further explore the time and spatial patterns. Then the analyst may make some hypotheses about the pattern, and examine the detailed tweet content afterward to verify the hypotheses.

This kind of iterative analysis often defined as an *exploration session* [48], which includes several queries where the results of the previous queries trigger the formulation of the next ones. There is thus a chance to speed up the queries by using the previous query results. For interactive visualization applications, the query-response time greatly affects the user experience. It is therefore desirable to have a system that can automatically collect the previous results and reuse them efficiently for solving the later queries.

Connecting a visualization frontend with a database system directly (e.g., Superset [9], Polaris [78], Tableau [87]) does not suffice for this end, mainly due to two reasons. First, the database system usually processes each incoming query independently. Therefore, it lacks optimization for a sequence of queries that share the similar interests. Second, even though some database systems support materialized views, it is difficult for the system to create or drop such views automatically. They instead rely on the administrator to define the views by evaluating a predictable workload. Moreover, in the exploration case, users may not know what they are looking for, and they will know that something is interesting only after they find it. Thus, it is not feasible to define views in advance. Those difficulties often lead to a complex design in the application's logic, which burdens visualization developers with the task of optimizing query speed.

OLAP datacubes [39] support fast aggregation queries, but only over the dimensions that were included during the construction of the cubes. With large datasets involving 20 or more dimensions, constructing a "full" cube with all dimensions is often not feasible [56]. For the same reason, it is not efficient to build a cube with a field that holds a large number of values, such as a textual field that contains tens of thousands of words.

There are also end-to-end solutions for building specialized visualization data management systems (e.g., SeeDB [86], DVMS [89]) that improve the visualization performance at every layer of the overall software stack. Since the outputs of such a system are visualization figures instead of data, they may not be easily integrated into existing frontend user interfaces. In addition, these kinds of systems often need to store another copy of the data in their own managed space. From the developer's perspective, it could be an unacceptable additional overhead to maintain an entire copy of a large dataset just for visualization.

In this chapter, we introduce a general-purpose middleware system, called Cloudberry, sitting between a visualization application and a database system to support interactive analytics and visualization on large datasets with various attributes. Cloudberry can automatically create, maintain, and delete views by analyzing each request and its results. By utilizing materialized views stored in the database, it can reduce the query response time remarkably. The system has several unique capabilities:

- *Visualization friendly interface*: we developed a generic API for the frontend to define and query data. The API allows the frontend to specify rich semantic information about the dataset, functional operations (e.g., filtering, aggregations, and lookups), query execution and result delivery requirements (e.g., ship the entire result as a whole or as a sequence of streaming partial results), and other visualization-related parameters;
- *Interactivity*: it is able to answer an analytical query efficiently (e.g., in sub-seconds) by using view-materialization techniques, which allows a user to analyze and explore data interactively;
- *Real-time Analysis*: by asking the query to visit both the base dataset and the already materialized view dataset, it allows users to query the latest information as well as historical data;
- *Generality*: by implementing the main logic in the middleware layer, it can connect to different types of frontend applications. It can also connect to various database systems with the corresponding connectors.

We focus on datasets with temporal, spatial, and textual attributes, which are common in domains such as social media and mobile phone application usage. We make two assumptions about the datasets.

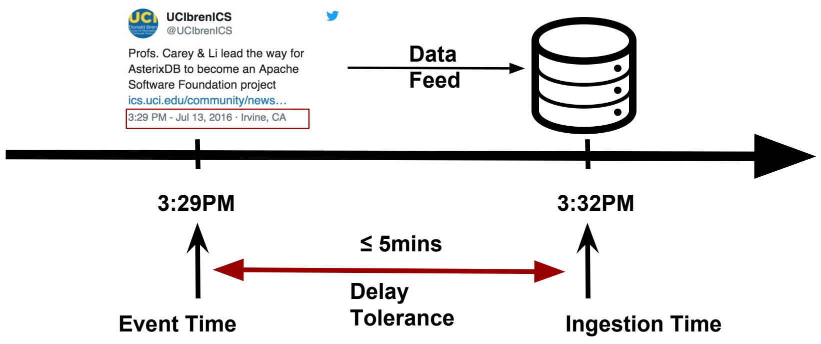

UCIbrenICS
@UCIbrenICS
Profs. Carey & Li lead the way for AsterixDB to become an Apache Software Foundation project
ics.uci.edu/community/news...
3:29 PM - Jul 13, 2016 · Irvine, CA
Data Feed
3:29PM
3:32PM
≤ 5mins
Delay Tolerance
Event Time
Ingestion Time

Figure 3.2: The relation between the *event time* and the *ingestion time*.

- *Append-only*: Many systems, including logging or monitoring, Internet of Things (IoT), social-media, etc., generate a huge amount of data and keep it for a long time for future analysis. Our target dataset is *append-only* in the sense that there is no update or deletion on historical records.

- *No "too late" records*: Often the time when a record is ingested into the database (called the "*ingestion time*") is later than the time when the event in the record occurred (called "*event time*"). Fig. 3.2 shows an example of TwitterMap that is using the AsterixDB data feed to collect the tweets from the Twitter API. The actual tweet was published at 3:29pm (event time). Due to delay in the Twitter service and network processing, it was not inserted into the database until 3:32pm (ingestion time). Cloudberry allows the developer to specify a maximum *delay tolerance* (noted as D, e.g., 5 minutes) to define the maximum time between the event time and the ingestion time.

In the following sections, we will provide an overview of the Cloudberry system (Section 3.2), its request interface (Secion 3.3), and its internal implementation details (Section 3.4). We then show several use cases of Cloudberry to demonstrate its generality (Section 3.5).

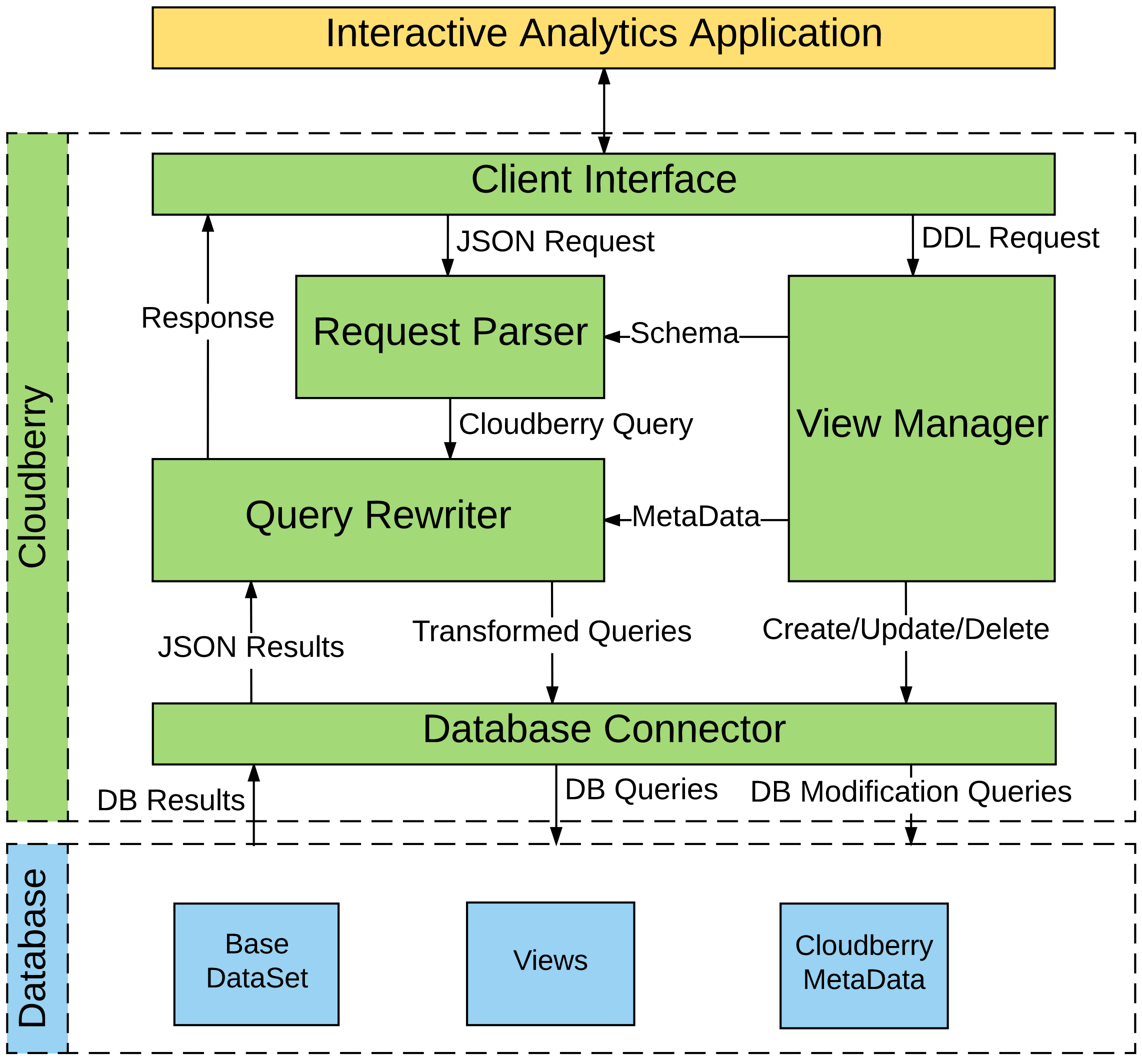


Figure 3.3: Cloudberry architecture.

3.2 Clouberry Overview

Fig.. 3.3 shows an overview of the main software components of Cloudberry.

The clients of Cloudberry are web applications. It provides HTTP as an interface to receive JSON requests. After receiving a JSON request, the *Request Parser* translates it into an internal Cloudberry query object. Each query object contains the dataset name, a sequence of data transformation expressions, e.g., an array of filter conditions, a series of unnesting and lookup parameters, group-by and aggregation functions, etc. To validate the request, the parser asks for the schema information of the dataset from the *View Manager*. In case the request contains a lookup request, the lookup dataset schema will also be retrieved from the *View Manager*. If the request is valid, it will be forwarded to the *Query Rewriter* to process.

The main optimization logic inside the *Query Rewriter* is utilizing materialized views [40] to answer the query. Based on the available view information provided by the *View Manager*, it may rewrite the given query into one or multiple queries on both the base dataset and view dataset.

The finalized query (or series of queries) is forwarded to the underlying database via the *Database Connector* that translates a query object into the corresponding database statements and then runs them in the connected database. The connector will also transform the database results of various formats into a unified JSON object.

Finally, the results are returned back to the *Query Rewriter*, which packages the results directly or combines them with the previous results into one response and then returns it to the client.

The *View Manager* controls the life-cycle of views. It stores the metadata of the base datasets and their associated views. The materialized views and the meta-dataset of Cloudberry are also stored in the backend database. In this way, we can rely on the database to do the computations both on views and base datasets so that the middleware is very lightweight.

3.3 API Design

The frontend application can communicate with Cloudberry using a JSON format request via an HTTP connection. Similar to SQL, there are two types of requests, including a *Data Declaration Request* that describes the data model of a dataset in a database and a *Data Querying Request* that specifies how to filter and transform the records of the dataset. In this section, we explain the details of these two request types.

3.3.1 Data Declaration Request

A *Data Declaration Request* defines the data model of a dataset in the underlying database of Cloudberry. In addition to the structure of a dataset, i.e., its field names and types, we require the application to specify additional semantic information that may be useful to analytics and visualizations. One main requirement is that the fields of a dataset should be classified as being either a *dimension* or a *measurement*. For example, a product name could be a dimension, and a product price or size would be a measurement. The definition is composed of five elements:

- *dataset*: it specifies the name of the dataset in the database;
- *primaryKey*: the name of the primary key in the dataset;
- *dimension*: it is a field considered as an independent variable whose data type is *ordinal* (e.g., the id of a record) or *categorical*. It is a field that can be filtered on or grouped by. From the perspective of visualizations, dimension fields are usually used as an

axis of a figure. Internally, the middleware may keep statistical information on certain dimensions for the purpose of optimized query rewriting;

- *measurement*: a *measurement* field is a dependent variable; that is, its value is a function of one or more dimensions. It is often of a quantitative type and is stored as numbers (integers or floats) or as complex objects like text or bag. A measurement is usually used to apply an aggregation function, such as count(), min(), max(), or topK(). It can also be used to filter the data, but should not be used as a "group by" key in a request;

- *timeField*: the name of the timestamp field in the dataset. This field is mandatory for declaring a more "active" dataset that new records are being inserted into. The time field is critical to guarantee the correctness of the response if it is composed partially by the results from a previous view. If the `timeField` is not declared, then Cloudberry will treat the dataset in question as a "static" dataset that will be used for the *lookup* request to augment the record from another dataset. We will introduce the *lookup* request in Section 3.3.3.

We will explain the *Data Declaration Request* of Cloudberry by using a dataset named `tweet`. Each `tweet` record contains a mix of date, string, and integer attributes. In addition, it also has nested data types. For example, the `hashtags` field contains a bag (unordered list) of string values; the `user` field itself is a structured record that contains user-related information such as id, name, image, etc. Before being registered with Cloudberry, the dataset should already be created and possibly ingested in the underlying database system. As an example, Fig. 3.4 shows the definition of the `tweet` dataset stored in the AsterixDB system in SQL++.

A frontend application can register the `tweet` dataset with the Cloudberry system by using the JSON specification in Fig. 3.5. The fields that are more likely to be grouped on, like `id`

```
create dataverse twitter;
use twitter;
create type typeUser if not exists as open {
   id: int64,
   name: string,
   screen_name : string,
   lang : string
};

create type typeGeoTag if not exists as open {
   stateID: int32,
   countyID: int32,
   cityID: int32
};

create type typeTweet if not exists as open{
   create_at : datetime,
   id: int64,
   "text": string,
   lang : string,
   favorite_count : int64,
   retweet_count : int64,
   hashtags : {{string }},
   user_mentions : {{ int64 }} ,
   user : typeUser,
   geo_tag: typeGeoTag
};

create dataset tweet(typeTweet) if not exists primary key id;
```

Figure 3.4: An example of the dataset `tweet` in AsterixDB

and `lang`, are declared as dimensions, while the fields that are less likely to be grouped on, like `text` and `retweet_count`, are specified as measurements. The primary key is the `id` field. The `create_at` field is the time field, indicating that there will be new records appended to this dataset in the database.

```
{
  "dataset":"twitter.tweet",
  "dimension":[
    {"name":"create_at", "datatype":"Time"},
    {"name":"id", "datatype":"Number"},
    {"name":"lang","datatype":"String"},
    {"name":"user.id","datatype":"Number"},
    {"name":"geo_tag.stateID","datatype":"Number"},
    {"name":"geo_tag.countyID","datatype":"Number"},
    {"name":"geo_tag.cityID","datatype":"Number"},
    {"name":"geo","datatype":"Hierarchy","innerType":"Number",
      "levels":[
        {"level":"state","field":"geo_tag.stateID"},
        {"level":"county","field":"geo_tag.countyID"},
        {"level":"city","field":"geo_tag.cityID"}]}
  ],
  "measurement":[
    {"name":"text","datatype":"Text"},
    {"name":"user_mentions","datatype":"Bag","innerType":"Number"},
    {"name":"hashtags","datatype":"Bag","innerType":"String"},
    {"name":"favorite_count","datatype":"Number"},
    {"name":"retweet_count","datatype":"Number"},
  ],
  "primaryKey":["id"],
  "timeField":"create_at"
}
```

Figure 3.5: DDL in Cloudberry to register the `tweet` dataset of the underlying database.

3.3.2 Data Model

The Cloudberry system can support the flat relational data model that has been widely adopted in visualization applications. It is straightforward to use the fields that are defined in the underlying relational database (e.g., MySQL and PostgreSQL). If the dataset is stored in a nested format, the full path of a field is required to be specified in the DDL. For example, AsterixDB supports the semi-structured data model. The `user.id` field name in Fig. 3.5 is a nested field stored at one level deeper in the `tweet` record. A client can use `dot` (".") to declare the `user.id` field in the Cloudberry DDL. When a request needs to access the nested field, the Cloudberry system will generate the appropriate underlying query syntax to obtain the value of the nested field. Unlike the Druid system [95], we do not need users to manually flatten their nested data records before the analysis can be.

Type	Filter	Group	Aggregation
Boolean	isTrue, isFalse	self	-
Number	<, <=, >, >=, ==, in, inRange	self,bin	count, sum, min, max, avg
Time	<, <=, >, >=, ==, inRange	self,interval	-
String	contains, matches, in	self	distinct-count
Text	contains	-	-
Bag	contains	-	-

Table 3.1: Cloudberry data types and functions. "-" means unsupported

Each field needs to have a data type. The Cloudberry system supports primitive JSON types, such as `String`, `Number`, and `Boolean`, as well as other frequently-used data types, such as `Datetime` and `Text`. Compared to the `String` type that specifies a short attribute (e.g., the language field `lang`), the `Text` type specifies a field with much larger content. Users may care more about its inside strings than using the entire bag of words as a single dimension. For example, the `text` field is declared to be of type `Text` so that the user can filter on the `text` field, but they should not group on the field.

In addition, Cloudberry supports the nested datatype `Bag` that declares a collection of values that have the same primitive type. For example, the `hashtags` field declares a bag of words of the `String` type. Table 3.1 lists the supported types as well as some of their corresponding functions. A complete list and more details can be found in the Cloudberry documentation [26].

Cloudberry allows users to define a synthetic field consisting of the existing fields. For example, the `geo` field in Fig. 3.5 declares a *hierarchical* field that consists of the `stateID`, `countyID`, and `cityID`. By specifying the level, Cloudberry knows the hierarchical relation between the three fields. When a client request a `group-by` operation on the `geo` field on the "county" level, the actual query will group on both `stateID` and `countyID` fields. In the future, the Cloudberry system could apply a `roll-up` function on such a field to generate nested group-by results in a predefined hierarchical order.

Notice that the fields in a request may not correspond to all the fields of a dataset in the backend database. If some fields are not relevant to the frontend analysis, they need not be mentioned in the request. For example, the `user.name` and `user.screen_name` fields in AsterixDB do not appear in the Cloudberry DDL in Fig. 3.5.

Compared to database DDL, which focuses more on the physical nature of the record's fields, the Cloudberry system is designed to capture a higher level of information to better understand the semantic nature of fields and their relationships. This kind of information is helpful to infer the relationship between previous requests and the current request, so that the common computation between them can potentially be saved by using previously stored query results.

3.3.3 Data Query Request

After declaring a dataset, clients can send a request to Cloudberry to query the underlying dataset. A Cloudberry's data-querying request is also expressed in a JSON format and can be submitted via an HTTP POST method or via a WebSocket. Such requests specify which dataset to query on and what records should be filtered and transformed. The request language supports common SQL-like functionalities, e.g., filter, group, sort, and lookup. In addition, it can also define how to deliver the query results. For instance, a query can be sliced into *mini-queries* and then there will be a stream of partial results returned to the client instead of one result (see Chapter 4). The result is also a JSON object, and contains the filtered or aggregated values. We will describe the request format by presenting a series of illustrative examples based on the previous declared `tweet` schema.

We begin the introduction with the request shown in Fig 3.6, which contains the basic filtering and group-by specifications. The request first filters the tweets whose `text` field contains "zika" and "virus", then groups the filtered tweets by both the `geo_tag.stateID`

```
{
  "dataset": "twitter.tweet",
  "filter": [{
      "field": "text",
      "relation": "contains",
      "values": [ "zika", "virus" ]
  }],
  "group": {
    "by": [
      { "field": "geo_tag.stateID", "as": "state"},
      { "field": "create_at",
        "apply": { "name": "interval",  "args": {"unit": "hour"} },
        "as": "hour"}
    ],
    "aggregate": [
      { "field": "*", "apply": { "name": "count" }, "as": "count"}
    ]
  }
}
```

Figure 3.6: Get the per-state and per-hour count of tweets that contain “zika” and “virus”

and hour fields, generated by applying the interval function to the create_at field. By using the “as” renaming operation, the result will contain the grouped keys in the “state” and “hour” field, and the aggregated value in the “count” field.

Fig 3.7 illustrates an example of the unnest and the order operations. The unnest operation flattens a Bag field into individual items, producing multiple records, each of which is one of the original input records augmented with a flattened item from its original collection. For example, after unnesting the “hashtags” field, there will be more records produced and each record will contain a single hashtag word in the newly added “tag” field. After that, the request groups the record on the new “tag” field and uses “count” as the aggregation function to get the frequency of each hashtag. Lastly, the select segment asks to sort the hashtags by its count in the descending order. The “-” mark indicates descending order, while a field name without “-” defines ascending order. The “limit” property specifies that only the top 10 most frequent hashtag should be returned.

Except for the aggregation result, analytics applications often have the need to examine

```
{
  "dataset": "twitter.tweet",
  "filter": [
    { "field": "text", "relation": "contains", "values": ["zika"]}
  ],
  "unnest" : [{ "hashtags": "tag"}],
  "group": {
    "by": [{ "field": "tag" }],
    "aggregate": [
      { "field" : "*", "apply" : { "name": "count" },"as" : "count"}
    ]
  },
  "select" : {
    "order" : [ "-count"], "limit": 10, "offset" : 0
  }
}
```

Figure 3.7: Get the top-10 related hashtags for tweets that mention “zika”

```
{
  "dataset": "twitter.tweet",
  "filter": [
    { "field": "text", "relation": "contains", "values": ["zika"]}
  ],
  "select" : {
    "order" : [ "-create_at"],
    "limit": 100, "offset" : 0,
    "field": ["text" ]
  }
}
```

Figure 3.8: Get 100 latest sample tweets that mention “zika”

```
{
  "dataset": "twitter.tweet",
  "filter": [
    { "field": "text", "relation": "contains", "values": ["zika"]}
  ],
  "group": {
    "by": [{ "field": "geo_tag.stateID", "as": "state"}],
    "aggregate": [
      { "field" : "*", "apply" : { "name": "count" },"as" : "count"}
    ]
    "lookup" : {
      "joinKey": ["state"],
      "dataset": "twitter.population",
      "lookupKey": ["stateID"],
      "select": ["population"],
      "as": ["population"]
    }
  }
}
```

Figure 3.9: Lookup the population value from the "population" dataset and append it to the current record. The "state" field (`joinKey`) in the result record will be used to join with the `population.stateID` (`lookupKey`) from the population dataset.

the details of the interesting records that satisfy certain conditions. Fig. 3.8 illustrates a sampling request that fetches the latest 100 tweets that mention "`zika`". Users can specify the `limit` and `offset` properties to return fewer results. In addition, one can specify certain fields in `field` properties to output only the selected fields of the results.

Fig. 3.9 shows an example of specifying the `lookup` operation. It expresses the semantics of a left-outer join that augments the records from the main dataset or the aggregation results with additional fields from another dataset. The `joinKey` specifies the fields from the original record; the `lookupKey` specifies the fields from another dataset to join with. In the example, the request first generates the per-state aggregation result of the tweets mentioning "zika", then augments the aggregated result by adding a new "population" field obtained from the matching records in the "population" dataset whose `stateID` is equal to the grouped `state` value. This is a useful feature to combine fields from other datasets. Particularly, TweetMap uses this `lookup` request to get the population of each administrative region to support the normalization feature that shows the per-capita count of matched tweets.

```
{
  "dataset": "twitter.tweet",
  "global": {
      "aggregate": {
        "field": "*", "apply": { "name": "count"}, "as": "count"},
  }
  "estimate" : true,
}
```

Figure 3.10: An estimate request that asks for the total number of tweets and it can accept an approximate answer.

Often in analytics applications, the most precise result may not always be needed. A frontend application may only want to get a rough idea of some simple facts (e.g., the cardinality of the dataset or the distinct count of a dimension) about the dataset quickly. Cloudberry supports this kind of *approximate* request by declaring the "estimate" property in the JSON request. The request in Fig. 3.10 illustrates an "estimate" example that asks for the cardinality of the `tweet` dataset. Recall that the attributes of a dataset are separated into *dimensions* and *measurement*. Cloudberry considers *dimensions* and *measurements* differently. Internally, the system can periodically collect the metrics (e.g., min, max, cardinality, etc) of some dimensions and store the values in a separate metadata dataset. In this way, without asking the underlying database, Cloudberry itself can answer some simple questions if an estimation is acceptable. Currently, we store the cardinality of the dataset and the `min` and `max` values of the `time` dimension in Cloudberry's metadata. Thus, the "`count *`" request in Fig. 3.10 can be estimated simply by checking the metadata at the middleware layer. In the future, we could keep track of the metrics of all dimension fields to be able to answer more estimate requests. If the `estimate` option is not specified or set to "`false`", the request will be solved by sending the rewritten queries to the database for their precise results.

Fig. 3.11 shows another distinct feature called "*slicing*" in Cloudberry. Analytical queries are often expensive due to the large data size. Thus, the frontend application may need to wait for a long time to update its user interface. To improve the interactiveness of the application, Cloudberry supports progressive computing by sending partial results sequentially to the

```
{
  "dataset": "twitter.tweet",
  "group": {
    "by": [{ "field": "geo_tag.stateID", "as": "state"}],
    "aggregate": [
      { "field" : "*", "apply" : { "name": "count" },"as" : "count"}
    ]
  }
  "options" : {
    "sliceMillis": 2000
    "continueMillis": 2000
  }
}
```

Figure 3.11: Using the `option` field to specify that the partial result should be delivered every 2,000 milliseconds and the result should be updated every 2,000 milliseconds for the newest data.

client, rather than waiting for query completion. As a result, the frontend application can refresh its UI at a much faster rate, and hence the interactiveness is much improved. The example in Fig. 3.11 asks for the per-state count for the entire `tweet` dataset. It has an “`option`” property indicating that it accepts the slicing of result and that the expected updating rate is 2,000 milliseconds. (Currently, we only support slicing the dataset on the time dimension from the latest data to the oldest one.) Then, instead of a single result, there will be a stream of partial results returned in reverse-chronological order in 2,000 millisecond intervals. Each partial result will contain the following attributes:

- result: It includes the requested aggregation results. In the case of the request in Fig. 3.11, it will be a collection of records containing the count of each state;
- percentage: It shows the percentage of progress of answering the current request. The `result` value is complete once the percentage reaches 100%;
- time interval: It indicates the time interval during which the `result` is valid.

In the future, we can also support *continuous* queries in the same manner by specifying the “`continueMillis`” option in the request.

3.4 System Implementation

We have described how a frontend application can issue requests to the Cloudberry system. Now we introduce the architecture and implementation details of the system.

3.4.1 Data Registration

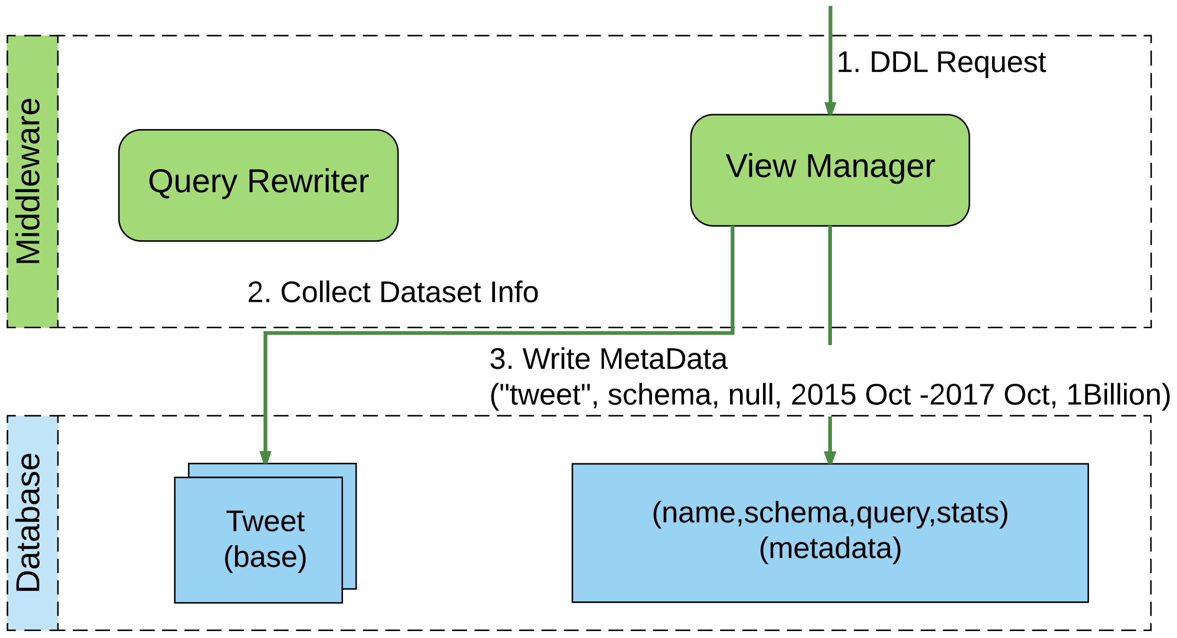


Figure 3.12: Data registration workflow

The first step of using Cloudberry is to register the dataset in the database by using the DDL JSON request described in Section 3.3.1. Fig. 3.12 shows the registration process. When the *View Manager* receives the data declaration request as described in Section 3.3.1 for registering the `twitter` dataset, a batch of statistical queries will be sent to the declared dataset. For example, the *View Manager* will send the query to get the range of the time dimension, the distinct count of the `user.id` dimension, the cardinality of the dataset, etc.

After the statistical information is collected, the *View Manager* will generate a metadata record including:

- name: the name of the dataset in the database;
- schema: the schema information of the dataset;
- query: the query object that defines the view if the dataset is a view. Views and their creating queries are automatically generated by the system. For the registered base dataset it will be always empty.
- time interval: the range of the time dimension. This field is mainly used to define the valid interval of the view, which is critical for the purpose of correct query rewriting using a view.
- stats: the statistical information about this dataset, including "create time", "last modify time", "last read time", "cardinality", etc. This information is mainly used to maintain a view. It can also be used to estimate the simple query result if the client accepts approximate answers.

The record is stored as a nested JSON record. Fig. 3.13 shows an example metadata record after registering the `tweet` dataset. The "`schema`" field stores the DDL JSON object as shown in Fig. 3.5. The "`query`" field is empty because it is a base dataset. The "`timeInterval`" field stores the `min` and `max` values of the time dimension at the registration time. The "`stats`" field shows that the dataset was registered at "`2017-10-03T12:40:12`" and that there are about one billion records in the `tweet` dataset.

The nested JSON format may not be fully supported in a traditional database system (e.g., MySQL). In such a case, we store each meta record as a string in the database and load the metadata dataset entirely into the middleware layer. The *Database Connector* parses the string record and sends the transformed JSON record to the *View Manager* for later use.

```
{ "name": "twitter.tweet",
  "schema": {
    "dimension":[
      {"name":"create_at", "datatype":"Time"},
      {"name":"id", "datatype":"Number"},
      {"name":"lang","datatype":"String"},
      {"name":"user.id","datatype":"Number"},
      ...
    ],
    "measurement":[
      {"name":"text","datatype":"Text"},
      {"name":"user_mentions","datatype":"Bag","innerType":"Number"},
      {"name":"hashtags","datatype":"Bag","innerType":"String"},
      ...
    ],
    "primaryKey":["id"],
    "timeField":"create_at"
  },
  "query": null,
  "timeInterval": {
    "start": "2015-01-01T13:33:18",
    "end": "2017-10-03T12:19:49"
  },
  "stats": {
    "createTime": "2017-10-03T12:40:12",
    "lastModifyTime": "2017-10-03T12:40:12",
    "lastReadTime": "2017-10-03T12:40:12",
    "cardinality": 1010826330
  }
}
```

Figure 3.13: An example metadata record of the `tweet` dataset in Cloudberry.

Once the data about a dataset is written into the database, the dataset is ready to be queried using Cloudberry.

3.4.2 Query Rewriting Using Views

Now we describe how views are created, utilized, and maintained in Cloudberry.

```
{
  "dataset": "twitter.tweet",
  "filter": [
    {"field": "text", "relation": "contains","values": [ "zika", "virus" ]}
    {"field": "retweet_count", "relation": ">","values": 5}
  ],
  "global": {
    "aggregate": {
      "field": "*", "apply": { "name": "count"}, "as": "count"},
    }
  }
}
```

Request 3.1: Get the count of tweets that contain “zika” and “virus” and retweeted more than five times.

3.4.2.1 View Creation

A *view* is a form of derived data as the result of applying some structural or computational transformations to the base data [25]. It is said to be materialized if its results are stored in the database. Studies [35, 94] show that precomputing views can significantly reduce query response time. In Cloudberry, views are always materialized.

In Cloudberry, we focus on append-only datasets that keep ingesting new data at a fast rate, in which case the time dimension plays an important role for correctly rewriting a query. For this reason, we include the time range as part of a view definition to specify that the result of a view is consistent with the result of the original query applied on the records from the base dataset within a specific time interval.

Formally, we define a view V of a dataset S as a query Q that applies on S with an additional time range predicate P_t. A view can be represented as $V(S, Q, P_t)$.

Cloudberry currently supports *subset view*s. A subset view is created by applying one filter predicate (in addition to the time predicate) onto the base table. Since such a view keeps every detail of the filtered records, it will be more likely to be utilized for answering new queries.

When the *Query Rewriter* receives a query, the filter predicates of the query expressions are transformed into *conjunctive normal form* (CNF). The filter expression is defined as $F_q = P_t \wedge P_1 \wedge P_2 \wedge \ldots \wedge P_m$, where P_t is the time condition. If it does not appear in the query, we will use the current time ($now()$) as an upper bound for the time filter condition. For example, if the query issuing time is "`2017-10-10T08:03:00`", then the filter predicates in Fig. 3.1 can be expressed as follows:

$$\begin{aligned} F_{tweet} = \{&create\_at < \text{2017-10-10T08:03:00}\} \\ &\wedge \{contains(text, \text{“}zika\text{”})\} \\ &\wedge \{contains(text, \text{“}virus\text{”})\} \\ &\wedge \{retweet\_count > 5\}. \end{aligned} \tag{3.1}$$

When there are no views that can be utilized, the query will be run as it is. By the time the query finishes processing, the query rewriter will suggest the views that may be useful to speed up such queries in the future. One principle it follows is that the size of the generated view should be as small as possible so that the future computation on the view could be cheaper. However, as a middleware component, the rewriter may not have the statistics for every filter condition. As a heuristic approach, we assume that a point-search predicate (e.g. "==" comparison or `contains()`) is more selective than a range search predicate. Under this assumption, we use the following rules:

1. If a point search predicate exists, only the point search predicate will be used to define the view;

2. If there are multiple point search predicates, multiple views will be generated such that each has a point search predicate;

3. If no point search is available, only one of the range predicates will be used to define the view. In the case where the predicate is on a field without statistical information,

the first predicate will be used.

As an example, in case the request in Fig. 3.1 has been processed and no views are available, the query rewriter will suggest two views:

1. view `zika` with the definition predicate as $P_{zika} = \{contains(text, zika)\}$;
2. view `virus` with the definition query as $P_{virus} = \{contains(text, virus)\}$.

The $\{$`retweet_count` $>$ `5`$\}$ condition will not be used to generate a view since there are already two point search predicates.

The query rewriter generates the view-creation requests with the predicate P_v of the view definition, but it does not create any views. The query will be packaged into a "createView" request and sent to the *View Manager* that will conduct the actual view creation task. The *View Manager* manages the lifecycle of views and their metadata information.

When the *View Manager* receives the view-creation query, it tries to create a view as latest as possible to catch up with the base dataset. Ideally, the time predicate of the view definition should be from the start time of the dataset (T_s) to the current timestamp (T_{now}). The issue is that as studied in [36, 10], the *ingestion time* when a record is ingested into the database is often later than the *event time* when the event in the record occurred. Moreover, the dataset usually only contains the event time. Thus, if we directly use the current time T_{now} as the view ending time to match the event time of the records, there could be some records that have not yet arrived in the system, and hence will not be included in the view. The view-based computation would then be incorrect. To solve this problem, Cloudberry allows the developer to specify a maximum *Delay Tolerance* (noted as D) to define the maximum time (e.g., 3 minutes) between the event time and the ingestion time to accommodate the lag between them.

When the *View Manager* receives the creation query, it will complete the view definition by filling the time period predicate that is from the start time of the dataset (T_s) that was collected during the dataset registration, and the timestamp of the current second minus the delay tolerance ($T_{now} - D$), i.e., $P_t = \{T_s <= timeField < T_{now} - D\}$. Then the view definition $V(S, \sigma_{P_v}, \sigma_{P_t})$ is complete and the data-creation query that contains two conjunctive predicates will be sent to the underlying database to execute.

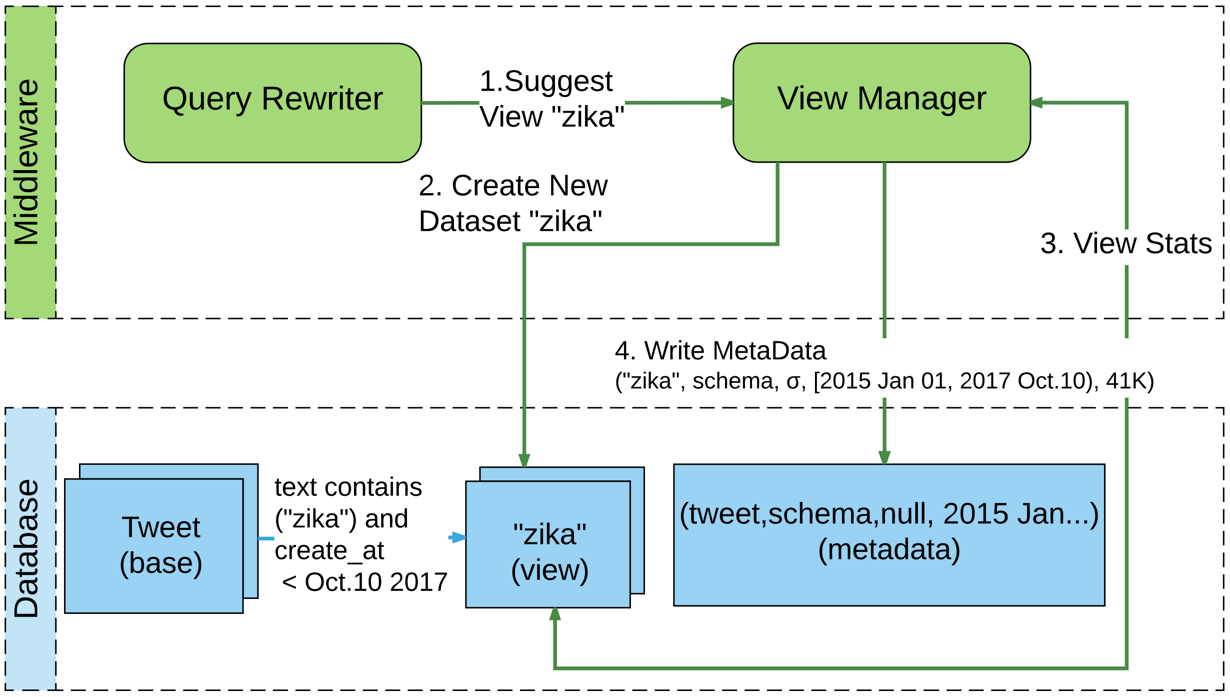

Middleware
Query Rewriter
1.Suggest View "zika"
View Manager
2. Create New Dataset "zika"
3. View Stats
4. Write MetaData
("zika", schema, σ, [2015 Jan 01, 2017 Oct.10), 41K)
Database
Tweet (base)
text contains ("zika") and create_at < Oct.10 2017
"zika" (view)
(tweet,schema,null, 2015 Jan...) (metadata)

Figure 3.14: Create view "`zika`".

Fig. 3.14 shows the process of creating the "zika" view (the request parser and the database connector are not shown for ease of illustration). First, the definition query $\sigma_{contains(text,zika)}$ is sent to the *View Manager*. Suppose that the time when the *View Manager* receives the query is 2017-10-10T08:03:00, the delay tolerance D is 3 minutes, and the starting date of the base dataset is 2015-01-01T13:33:18 (collected when registering the `tweet` dataset). Then the view definition is

$$V(tweet, \sigma_{contains(text,zika)}, \sigma_{\text{2015-01-01T13:33:18} <= create\_at < \text{2017-10-10T08:00:00}}).$$

Then the "zika" view will be generated by pulling the matching records from the `tweet` dataset to the newly created `zika` dataset. The generated SQL++ statement is shown in Fig. 3.15.

```
USE twitter;
CREATE dataset zika(typeTweet) IF NOT EXISTS primary key id;
INSERT INTO zika (
  SELECT VALUE t
  FROM twitter.tweet t
  WHERE t.`create_at` < datetime('2017-10-10T08:00:00')
    AND ftcontains(t.`text`, ['zika'], {'mode':'all'})
)
```

Figure 3.15: Create the "zika" view in AsterixDB.

When the view is created, the *View Manager* will again issue statistical queries on the newly generated view to collect necessary information, e.g., the cardinality of the view dataset. Once the statistical data is returned, a new record about the "zika" view is stored in the metadata. Fig. 3.16 shows an example meta record about the "zika" view after it has been created. It has the same schema as the base dataset. The "query" field stores the query with the predicate P_v. Its time range is from the start date of the base dataset (`2015-01-01T13:33:18`) to the end of the creating time (`2017-10-10T08:00:00`). The "stats" fields indicate that the view was successfully built at `2017-10-10T08:05:35`, which is the creation time, the last update time, and also the last read time of the view. This time information is used by the *View Manager* to control the lifecycle of the view. The view can be used by the query rewriter in the future.

Multiple view creation requests will be executed sequentially since the dataset creation could be an expensive operation due to a significant amount of disk reads from the base dataset and also the disk writes to the view dataset. For example, the "virus" view as suggested by the query rewriter will not be processed until the "zika" view has been built successfully.

```
{ "name": "twitter.zika",
  "schema": {...},
  "query": {
     "dataset": "twitter.ds_tweet",
     "filter": [
       { "field": "text", "relation": "contains", "values": [ "zika" ] }
     ]
  },
  "timeInterval": {
    "start": "2015-01-01T13:33:18",
    "end": "2017-10-10T08:00:00""
  },
  "stats": {
    "createTime": "2017-10-10T08:05:35",
    "lastModifyTime": "2017-10-10T08:05:35",
    "lastReadTime": "2017-10-10T08:05:35",
    "cardinality": 41150
  }
}
```

Figure 3.16: The "zika" view metadata record

3.4.2.2 View Selection

Having discussed how to create views, we now turn to their use. After receiving a data request, the query rewriter will ask the *View Manager* to get all the views created on the requested base dataset. From all the views of the requested dataset, the rewriter needs to find a view whose definition *contains* the original query. This is a classic problem that has been studied extensively in the literature [43, 67, 38]. Based on the context of the subset views, in which case the details of the record are well kept in the view dataset, the question we wish to answer is "*can the required rows be selected using the view?*".

We will use the request example in Fig. 3.1 again to explain the view-selection process. Suppose that request is sent to the system at `2017-10-10T08:30:00`. Then the predicate

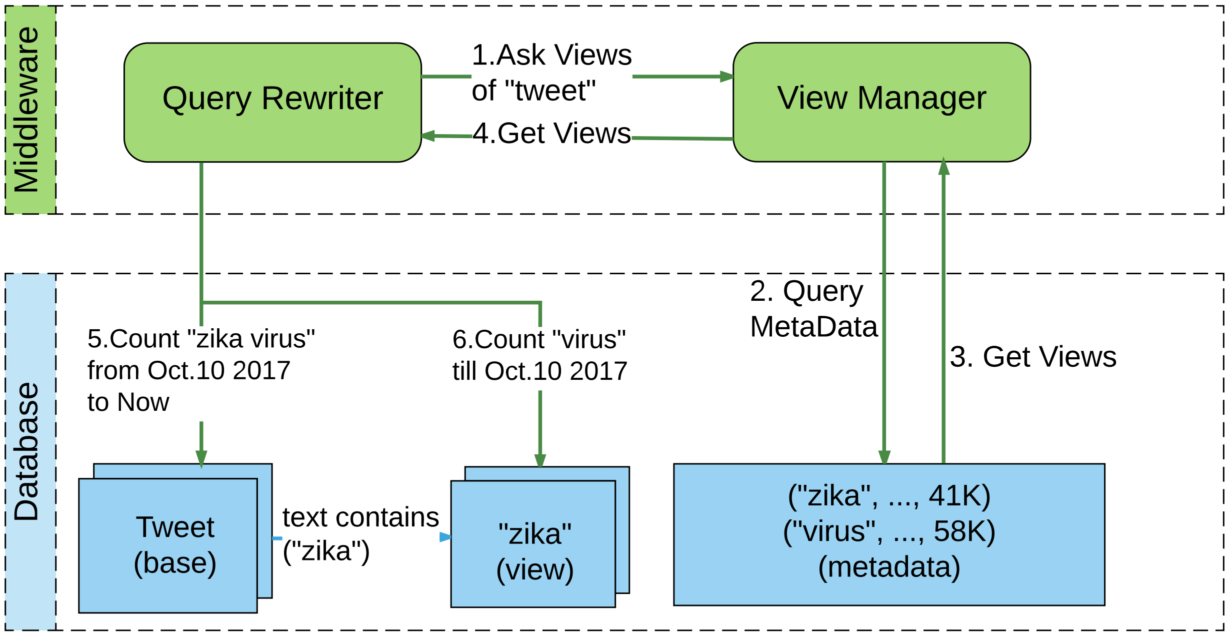


Figure 3.17: The process of rewriting the request using "`zika`" view.

expression of the request is as below:

$$\begin{aligned} F_{tweet} = &\{create\_at < \text{“2017-10-10T08:30:00"}\} \\ &\wedge \{contains(text, \text{“}zika\text{”})\} \\ &\wedge \{contains(text, \text{“}virus\text{”})\} \\ &\wedge \{retweet\_count > 5\}. \end{aligned} \tag{3.2}$$

Similarly, the view definition can be expressed as two predicates $F_v = P_{v,t} \wedge P_v$. The predicates of the "zika" view and the "virus" view are given as:

$$F_{zika} = \{\text{2015-01-01T13:33:18} <= create\_at < \text{2017-10-10T08:00:00}\}$$
$$\wedge \{contains(text, \text{“}zika\text{”})\}$$

and

$$F_{virus} = \{\text{2015-01-01T13:33:18} <= create\_at < \text{2017-10-10T08:00:00}\}$$
$$\wedge \{contains(text, \text{“}virus\text{”})\}$$

respectively.

The first task for the query rewriter now is to find a *matching* view. We say a view is *matching* if it satisfies both of the following conditions:

1. the predicate of the view $V.P_v$ *covers* one of the predicates $Q.P_i$ from the query Q,
2. the time range predicate of the view $V.P_t$ overlaps with the time predicate $Q.P_t$ of the original query.

The coverage test between $V.P_v$ and $Q.P_i$ is conducted in a semantic way so that it will check the logical meaning of the expression rather than a simple string match. For example, $\{contains(text, \text{“}zika\text{”})\}$ and $\{contains(text, \text{“}ZIKA\text{”})\}$ will be treated as a covered case because `text.contains` is a case-insensitive keyword search. Similarly, the set predicate $\{in(state.id, [23, 55]\}$ is covered by $\{in(state.id, [55, 23])\}$ since the order of values does not matter for the “*in*” relation. The time overlap test is based on checking interval overlaps. For example, given the predicate in Equation 3.2 of the request in Fig. 3.15, both views “zika” and “virus” will be matched because both time ranges overlap with the request's range and both view definitions cover the query predicates $\{contains(tweet.text, \text{“}zika\text{”})$ and $\{contains(tweet.text, \text{“}virus\text{”})\}$, respectively.

Once a view is matched, the rewriter will generate a new equivalent predicate expression on the view by applying the uncovered predicates from the original query. In case the view's time range does not entirely cover the request range, the time predicate on the base table query will also be updated to compensate for uncovered query results.

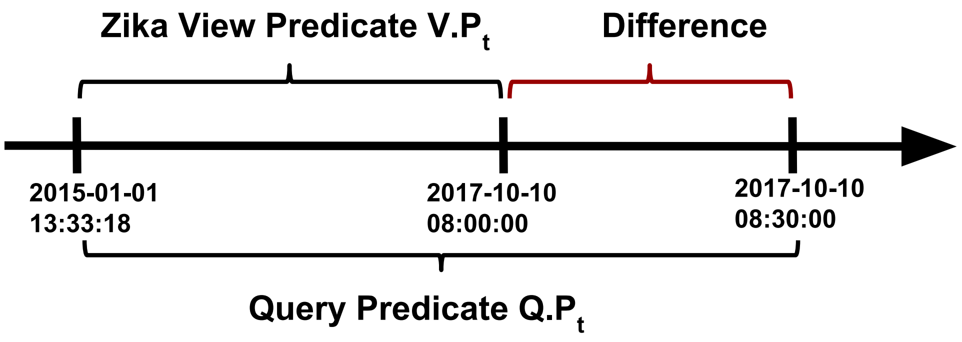


Figure 3.18: The time relationship between the "zika" view and the query.

For instance, as illustrated in Fig. 3.18, the "zika" view can be used to answer the query partially. To compute all the answers, the base dataset is also accessed to retrieve the newer records between the view ending time and the query ending time. The predicate on the view is

$$\begin{aligned} F'_{zika} = \{&\text{2015-01-01T13:33:18} <= create\_at < \text{2017-10-10T08:00:00}\} \\ &\wedge \{contains(text, \text{“}virus\text{”})\} \\ &\wedge \{retweet\_count > 5\} \end{aligned}$$

and the predicate on the base dataset is

$$\begin{aligned}F'_{tweet} = &\{\text{2017-10-10T08:00:00} <= create\_at < \text{2017-10-10T08:30:00}\}\\ &\wedge \{contains(text, \text{“}zika\text{”})\}\\ &\wedge \{contains(text, \text{“}virus\text{”})\}\\ &\wedge \{retweet\_count > 5\}.\end{aligned}$$

The equivalent predicate from $Q.P_i$ will be eliminated in the rewritten filter expression on the view. However, the time predicate is not dropped because there might be an ongoing maintenance process on the view, which appends the records that happen after the original view's time predicate. For example, when the query Q was issued, there could be already a maintenance operation to append the records that occur during the "difference" interval after the view ending time (`2017-10-10T08:00:00` as shown in Fig. 3.18). If we do not include the ending time predicate to the query on the "zika" view, the records that happened after `2017-10-10T08:00:00` may be included twice.

There could be multiple views that match the query, in which case we need to decide which views are more likely to speed up query processing. Currently, we only pick one of the views to use. We use a heuristic approach to pick the view with the smallest cardinality. In our current example, both the "zika" view and "virus" view are matched based on the coverage test. Since the cardinality of "zika" view (`41,000`) is less than that of the "virus" view (`58,000`) as shown in Fig. 3.17, the "zika" view will be chosen.

At last, the records from the selected view and the base dataset will be combined, and then the rest of the operation expressions (e.g., group by, order, lookup, etc.) will be appended to form a new query. The newly composed query will be translated into the appropriate database language and executed. As an example, the rewritten query using "zika" view can be expressed in SQL++ as shown in Fig. 3.19.

```
SELECT count(u) AS count FROM (
  SELECT * FROM twitter.zika t
  WHERE t.`create_at` >= datetime('2015-01-01T13:33:18')
    AND t.`create_at` <  datetime('2017-10-10T08:00:00')
    AND ftcontains(t.`text`, ['virus'])
    AND t.`retweet_count` > 5
  UNION ALL
  SELECT * FROM twitter.tweet t
  WHERE t.`create_at` >= datetime('2017-10-10T08:00:00')
    AND t.`create_at` <  datetime('2017-10-10T08:30:00')
    AND ftcontains(t.`text`, ['zika'])
    AND ftcontains(t.`text`, ['virus'])
    AND t.`retweet_count` > 5
) AS u;
```

Figure 3.19: The SQL++ query to combine the records from the "zika" view and the base dataset "tweet".

The size of the "zika" view (41,000) is significantly smaller compared to the size of the base `tweet` dataset (one billion). Thus, the time spent on scanning on view "zika" is much less than querying on the base dataset. On the other hand, the time range predicate applied on the base `tweet` dataset is only asking for records from the last eight hours. Compared to the entire range of two years, it is again a small portion of data to process, if there are some storage optimizations based on the time dimension. For example, there could be an index on the time dimension to speed up the data access. It is also not uncommon for a database system to use the time dimension to partition the dataset so that visiting a certain small range of the data will be even faster than using an index. Particularly, AsterixDB uses filters to accelerate data access by skipping many unrelated LSM components [15]. Moreover, a database system often optimizes the access speed on the "hot" (recent) data, e.g., keeping the most recent data in memory. Thus, the total time that is spent on the base data for a small portion of the latest data on the base dataset should be small. In this way, by routing the data access path from the large base dataset to a small query on a materialized view plus a small residual base query, we can significantly reduce the total query time.

3.4.3 View Maintenance

After a view has been created, we need to keep it up to date with changes to the base data (a.k.a view maintenance) to keep the content consistent with its definition. Cloudberry focuses on the domain where the dataset is append-only (no updates and deletes on the historical records) and new records are continually inserted. Though there will not be any correctness issue if a view is not updated, it is desirable in order to cover a larger range of the base dataset so that more pre-computed results could be used to answer future queries.

3.4.3.1 Incremental Updates of Views

We employ a *periodic* incremental view maintenance strategy (e.g., every 1 hour) [41] so that the views are updated gradually over time. A naive implementation, for example, is to trigger an update process every 1 hour, which retrieves the records that satisfy the predicate from the base table between the last update time to the current system time T_{now} and stores them in the view. As we described in Section 3.4.2.1, however, we also need to consider the maximum *Delay Tolerance* (e.g., 3 minutes) between the event time and the ingestion time to accommodate the lag between them.

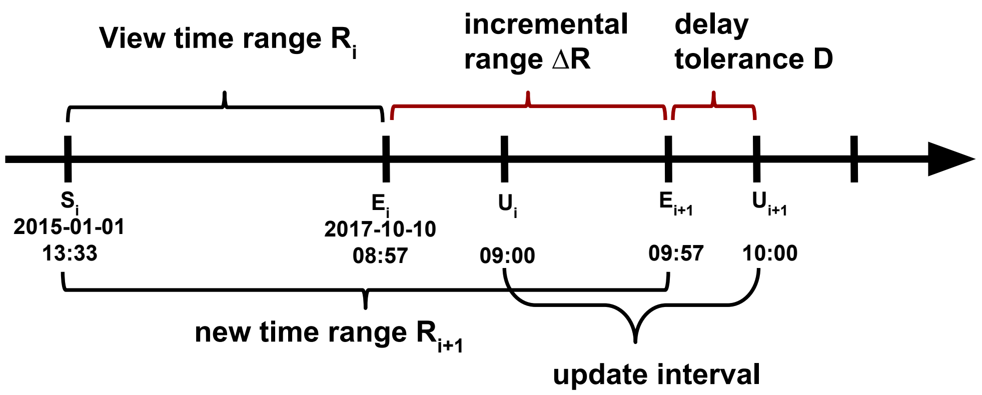


Figure 3.20: Update the view based on its time range.

Fig. 3.20 illustrates the timestamps that are used to append a view. Recall that each view contains the "stats" information about when it has been created or updated as described in Section 3.4.2.1. Take the "zika" view as an example. Based on the metadata in Fig. 3.16, the view ending time (E_i) is `2017-10-10-08:00:00`, meaning that all of the "zika" related records that happened before that time have been stored in the view. The last updating time (U_i) is at `09:00am`. Assuming the update interval is 1 hour, by the time `10:00am` (U_{i+1}), the *View Manager* is reminded to update the "zika" view. If the delay tolerance is 3 minutes, then the ending time of the updated view is `09:57am` (E_{i+1}, 3 minutes to U_{i+1}). The *View Manager* then applies the range predicate `08:57am` $<=$ *create_at* $<$ `09:57am` (ΔR) together with the filter predicates of the view definition $\{contains(text, \text{“}zika\text{”})\}$ to the base dataset to incrementally update the view. The corresponding SQL++ request is shown in Fig. 3.21.

```
INSERT INTO twitter.zika (
  SELECT value t
  FROM twitter.tweet t
  WHERE t.`create_at` >= datetime('2017-10-10T08:57:00')
    and t.`create_at` < datetime('2017-10-10T09:57:00')
    and ftcontains(t.`text`, ['zika'], {'mode':'all'})
)
```

Figure 3.21: The SQL++ example of appending more records from the base dataset `tweet` to the view "zika".

Once the update request has finished, the *View Manager* collects the new cardinality of the view and updates the meta record of the view by changing the time range as well as the statistical information including the last update time and cardinality of the view. Fig. 3.22 shows an example of the updated view meta record after the maintenance process was finished.

3.4.3.2 View Deletion

The advantage of having a subset view, compared to an aggregation view, is that it keeps the finest details of the original records from the base dataset. As a result, it allows more

```
{ "name": "twitter.zika",
  "schema": {...},
  "query": {
     "dataset": "twitter.ds_tweet",
     "filter": [
       { "field": "text", "relation": "contains", "values": [ "zika" ] }
     ]
  },
  "timeInterval": {
    "start": "2015-01-01T13:33:18",
    "end": "2017-10-10T09:57:00"
  },
  "stats": {
    "createTime": "2017-10-10T08:05:35",
    "lastModifyTime": "2017-10-10T10:01:00",
    "lastReadTime": "2017-10-10T01:03:35",
    "cardinality": 41154
  }
}
```

Figure 3.22: The “zika” view’s metadata record after the update finished at 2017-10-10T10:01:00.

operations (e.g., group, lookup, sort, project) on the view data, which can increase the chance of utilizing a view. On the other hand, the disadvantage of keeping the complete copy is that more resources will be needed and the update cost will be increased proportionally with the number of views.

In Cloudberry, we tackle this problem by removing less-frequently used views. Recall that each view meta record has a “last read time” property. Each time a view gets used, the “last read time” in the meta record will be updated. The *View Manager* will check all the views periodically to remove the views that have not been actively used for a certain amount of time (e.g., 24 hours).

3.4.4 Concurrency Management

Cloudberry follows the actor model [5] to handle concurrency. The actor model provides a high-level abstraction for writing concurrent and distributed systems. An actor is a conceptual entity to deal with computation within its context thread. Actors are completely isolated from each other, and they will never share memory. Actors communicate with each other by sending asynchronous messages. Those messages are stored in actors' mailboxes until they are processed. We use the Akka [11] toolkit to implement the actor model for Cloudberry. The library natively supports mailbox and message passing. Thus, it alleviates efforts of dealing with explicit locking and thread management, making it easier to implement concurrent and parallel systems.

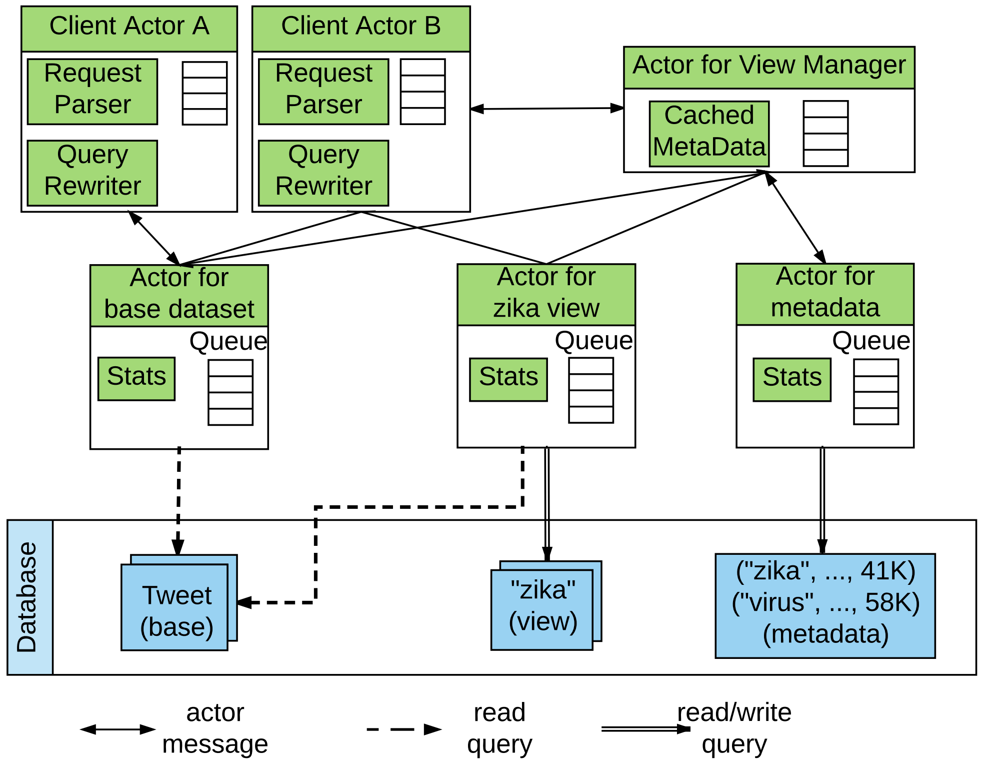


Figure 3.23: Actor architecture of Cloudberry.

Fig. 3.23 shows how actors are organized in Cloudberry. Internally, each actor has a queue to store its incoming messages. Each HTTP connection is handled by a *Client Actor*. The actor has its own *Request Parser* and *Query Rewriter* to serve its queries independently. There is only one *View Manager Actor* in the system that provides the functionality of the view maintenance. It also has an in-memory copy of the metadata to speed up the frequent metadata search requests. Each dataset in the database is assigned to one specific actor to control the query and maintain against the dataset. The dataset actor can read other datasets, but it can only write to its assigned one. The actor assigned to the base data is unique in the sense that it can only read but not write the base dataset.

A client actor asks for the data actor through the view router; then the client will interact with the data actor directly to send the query. Different client actors may interact with the same data actor. In this way, the data actor can precisely know the statistical information, such as how many times different clients have used the view. The read requests sent to the data actor will be issued asynchronously, while the write requests (e.g., updating view, updating metadata) will be run sequentially so that multiple maintenance processes will not interleave with each other. Whenever a data actor needs to write to another dataset, for example, if the "zika" view wants to update its time range, the request will be sent to the centralized router to reconcile the multiple writes.

Another interesting aspect of the actor model is that actors can live remotely on different machines. An actor is capable of sending a message to a remote actor that lives on another machine. Thus, this architecture can easily scale up to multiple machines, which is an important step towards a scalable middleware.

3.5 Use Cases

3.5.1 Paraview Web

The Army Research Lab [18] has developed a high-resolution data visualization framework [79] on top of the ParaViewWeb [66] supporting both scientific and non-scientific data visualization within a single framework. Fig. 3.24 shows their data visualization application running on a high-resolution tiled display within the SAGE2 [59] framework. It shows an

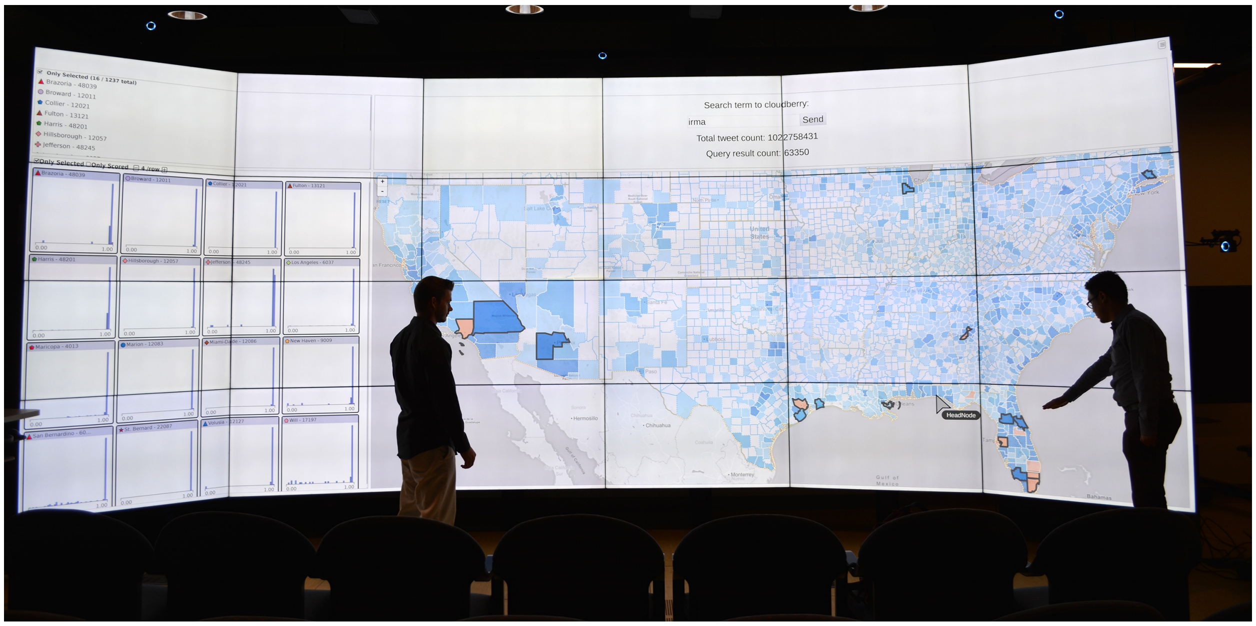

Search term to cloudberry:
irma
Send
Total tweet count: 1022758431
Query result count: 63350

Figure 3.24: The large display visualization system at the Army Research Lab.

example of using Paraview web to visualize one billion tweets on a map interface similar to TwitterMap. Additionally, their interface can show temporal patterns for the selected regions for a more in-depth analysis. Cloudberry is used as a middleware layer between a data management system and the visualization side of the ecology. Based on their experience, it is simple for the frontend application to interface with Cloudberry. The frontend passes a JSON request to Cloudberry and listens for messages and responses. As the Cloudberry middleware supports query optimization, there is no more complexity on the client side as compared to sending a query directly to a database. Therefore, the frontend can focus on the visualization tasks without worrying about performance issues.

3.5.2 Twitter Analytics

The TwitterMap application is used by the UCI public health department to conduct research on social media's reactions to certain public health concerns. [47] studied the content of climate change in Twitter messages before, during, and after the November 8th, 2016 US

Presidential election. TwitterMap was used to generate the by-time trend of the related topics. [60] employed TwitterMap to investigate the feasibility of using Twitter data for Zika virus epidemic tracking and forecasting at a national and state (Florida) level. They demonstrated the value of utilizing Twitter data for the purposes of disease surveillance and early disease signaling. TwitterMap was used to quickly generate geographic and time patterns of the "zika" related tweets to compare with the official Zika cases reported by U.S. Centers for Disease Control and Prevention (CDC).

3.5.3 Connecting to Different Databases

Cloudberry is a general-purpose middleware system that can not only allow multiple different applications to request various datasets, but also connect to different databases.

We have implemented an AsterixDB connector to communicate with an AsterixDB cluster by using SQL++ statements. We also implemented a MySQL connector and a PostgreSQL connector to interact with a MySQL database or a PostgreSQL database by using their SQL dialects. With minimal schema changes (due to their flattened relational model), one can successfully build the TwitterMap application with the data stored in MySQL or PostgreSQL database separately. More details can be found on the Cloudberry Wiki page [27].

3.6 Comparison with Related Work

Analytics and Visualization Systems

There have been many research studies on data analytics and visualization as surveyed in [37]. Frontend visualization solutions, like Superset [80] and Polaris [78], can generate precise sets of relational queries from the visualization specification in the user interface. Such solutions could be clients of Cloudberry that could be used to quickly retrieve their required data.

There are in-memory visualization solutions, like Nanocubes [55], ImMens [56] , that rely on in-memory data structures to reduce query processing time. These solutions would not scale easily if the data size is larger than memory.

There are specialized storage systems supporting interactive analytics on big data. Druid [95] is a high-performance, column-oriented, distributed data store. Dremel [61] combines multi-level execution trees and a columnar data layout to store read-only nested data, and it can support interactive aggregation queries over trillion-row tables. MapD [58] provides an analytical solution by relying on hardware GPU support. Our solution is extensible in the sense that by building the corresponding database connectors, Cloudberry can sit on top of these systems to explore the shared context between queries to further reduce the query time, which could then save computation resources.

Some visualization systems use pre-processing to reduce their query latency in the case where the workload is known in a-priori. Battle et al [20] developed middleware based on a memory cache to serve data tiles from a backend database based on users' recent requests. Hippo [96] pre-partitions the data on a pre-defined attribute and builds a lightweight index structure to skip the irrelevant pages if they do not contain the queried partitions. Since their results are calculated beforehand, this approach may not be feasible if each query is not known a-priori. XmdvTool [72] provides speculative prefetching strategies where the interaction options in the visualization frontend are limited.

There are also domain-specialized visualization systems. KITE [57] is an end-to-end system capable of managing microblog data at a large scale. HadoopViz [33] uses a MapReduce-based framework to generate high-quality geographic images with giga-pixel resolution for spatial data. Our work targets more general domains and more diverse workload.

There are also end-to-end systems, such as Tableau [83] and DVMS [90], that are designed to explore cross-layer optimization opportunities from the bottom data management layer to the top visualization layer. Since the outputs of the system are visualization figures, the solution may not be easily integrated with other existing frontend user interfaces. Besides, this kind of systems need to store another copy of the data in their own managed space. From the developer's perspective, this additional overhead to maintain an entire copy of the dataset just for visualization may not be acceptable.

BlinkDB [4] and INCVISAGE [68] answer queries interactively by delivering approximate results using sampling techniques. The approximate answering technique could also be used in our system.

Materialized Views

Materialized views have been studied intensively in the literature [40, 2, 41, 43, 38, 67]. Given a set of predefined views, the problems are divided into two categories: 1) maintaining materialized views efficiently when the base dataset changes, and 2) answering queries using views effectively to improve performance and availability. Given a general-purpose database environment, the maintenance workload must consider deletions and updates, which greatly increase the overhead of maintenance compared to the unique scenario of append-only datasets. Answering queries using views to find the least expensive plan is computationally hard even for restricted query classes [2]. The work in [43, 38] has explored various rewriting techniques that can apply to both select-project-join view and aggregation views. These techniques are being used in our work to implement materialized views in the middleware layer instead of in the database layer. In this way the visualization frontend can connect to the dataset in a database system that does not have its own support for materialized views. Moreover, Cloudberry's views are automatically managed by the middleware layer based on the query logic. The visualization frontend then does not need to handle query performance issues.

Data Cube

Data Cube [39] is another pre-computing technique to generate aggregation results for pre-defined multidimensional groups. There are many studies of efficiently computing data cubes [3, 44, 71] and organizing them for query answering [44, 75, 75]. There are also systems [52, 77] that generate a data cube for a large dataset stored in Hadoop or HBase. Different from the materialized view approaches, data cube systems usually enumerate all combinations of every dimension value so that users can quickly slice and dice on the specific range of dimensions to get the pre-calculated results. However, the exploration path is largely limited by the pre-defined dimensions and aggregation functions during cube construction. For example, this approach cannot support slicing on measurement fields. Especially for the case of keyword matching, it is infeasible to enumerate all keywords in a text field to build the cube. Our system adopts the notion of *dimension* and *measurement* only for the purpose of supporting better query rewriting. The views in Cloudberry can be built by applying any filter conditions, either on dimensions or measurements.

Semantic Caching

Semantic caching [30, 16] studies how to reuse subsets of input tables that are stored in a client-side cache. Each entry is modeled semantically as a set of queries, rather than at the physical level as a set of data pages or tuples. When a query is submitted to the client and can be answered (partially) using the cache, only a "remainder query" will be sent to the server to fetch the residual results. The technique needs to split the result on an attribute with bounded value space, thus its usage is limited and more suitable for a predefined query workload on the client side. We have implemented this technique in the TwitterMap frontend in [50] to cache the geographic aggregation results in order to skip unnecessary Cloudberry requests. Our work in the middleware layer focuses on using much richer context information between requests to speed up a broader range of requests.

Automated Physical Design

Automated tuning [24] is a rich field including topics such as partitioning [65, 70], index selection [12, 73], and materialized view selection [6, 7, 19, 31]. Adaptive index selection creates and drops indexes on-the-fly [12, 73]. Adaptive materialized view selection [6, 7, 19, 31] shares the same philosophy by selecting the most promising views based on the observed queries. Most of the index and materialized view selection techniques use a DBMS optimizer's cost model to evaluate the benefits of an index or view, and they have to be implemented inside a database. Cloudberry implemented the automatic view maintenance in the middleware layer, which may save the frontend development efforts for managing the views if the application connects to a database systems without the automated physical design support.

3.7 Conclusions

In this chapter, we have presented the design, implementation, and use cases of Cloudberry, a general-purpose middleware system specially designed for interactive analytic and visualization applications for large-scale data. It can reduce the query response time remarkably by utilizing materialized views stored in the database. By using the example TwitterMap, we demonstrated that it is a suitable solution to support interactive data analytics and visualization on one billion tweets. By using an example Paraview web application and building connectors to different databases, we have demonstrated that Cloudberry is general-purpose and can be used by various frontend applications on different backend database systems. We have open-sourced the Cloudberry system (http://cloudberry.ics.uci.edu) since 2015, and we invite others to download and try the system.

Chapter 4

Drum: A Rhythmic Approach to Interactive Analytics on Large Data

4.1 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 latency 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.

One way to solve the problem is to materialize earlier query results as views, with which we can answer future related queries, as discussed earlier in Chapter 3. This view-based approach does not solve the problem completely, since there can always be queries that cannot be answered using materialized views, thus resulting in a high latency.

As a motivating scenario, suppose the first request that the TwitterMap application sends to Cloudberry is "show the number of tweets mentioning the keyword `zika` per state". Since there are no views available, Cloudberry has to send the query to the database as it is:

Q:
```
SELECT state, COUNT(*)
FROM twitter t
WHERE ftcontains(t.text, "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 may 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. Then Cloudberry can merge the results as needed and send them back to TwitterMap progressively. Finally, the frontend user interface can be updated more quickly, which in turn leads to a more responsive user experience as demonstrated in Section 2.5.4.

The following is an example mini-query:

Q_m:
```
SELECT state, COUNT(*)
FROM twitter t
WHERE ftcontains(t.text, "zika") AND
      2016-01-01 <= t.create_at AND t.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

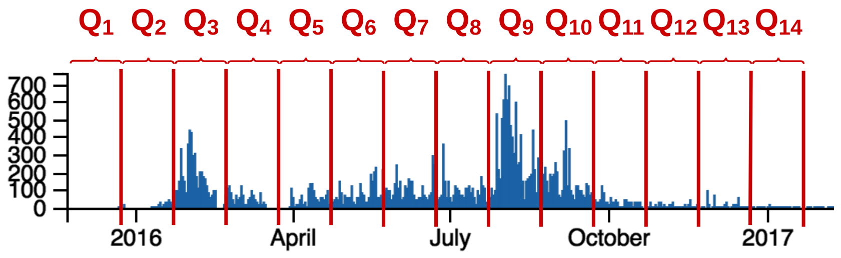


Figure 4.1: Distribution of "Zika" tweets collected over 1.5 years from November 2015 to May 2017.

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 data distribution along the slicing attribute can be skewed. Fig. 4.1 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.

In this chapter 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 4.2).

- 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 4.3).

- 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 expected gain of the next mini-query (Section 4.4).

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

- We have conducted an extensive experimental study on a real, large data set of social media tweets to evaluate Drum and these techniques and to demonstrate their efficiency (Section 4.5).

4.2 Problem Formulation

4.2.1 Architecture and Query Slicing

We consider a widely-used multi-tier Web architecture that includes a backend database system, a Cloudberry middleware server, and a frontend application, such as the web service as described in Section 2.1. The frontend application sends requests to Cloudberry, which in turn generates queries to the backend system to retrieve results and sends a response to the frontend.

Consider a dataset R that contains a set of fields in the backend database. The middleware posts a query Q to the dataset. 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 dataset R has an attribute ρ, called a *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.

Section 4.1 showed an original tweet query Q and a mini-query Q_m on this dataset. 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 the ρ is a numerical attribute.

4.2.2 Slicing Schedules and User Satisfaction

There can be many ways to slice a query into mini-queries. Fig. 4.2 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.

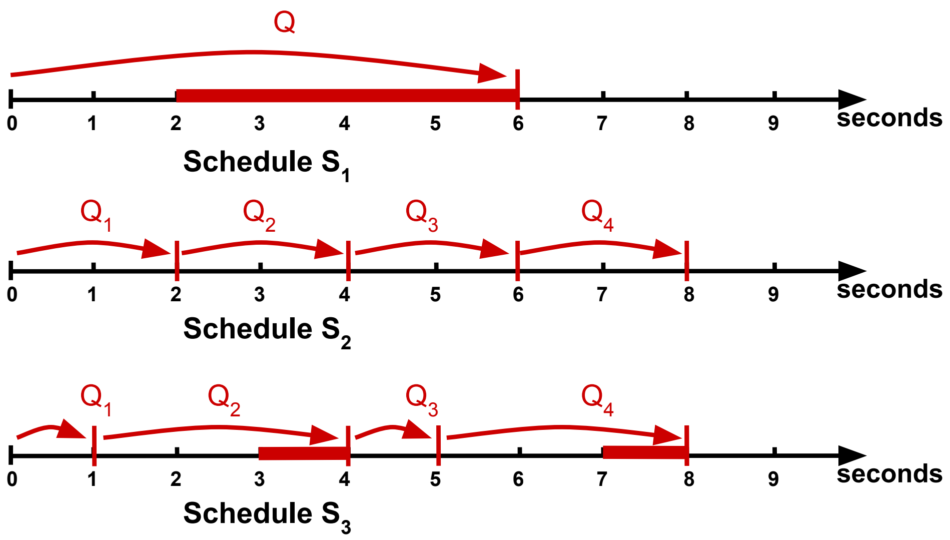


Figure 4.2: Three query-slicing schedules, where each thick line indicates a delay for an expected 2-second pace.

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 provides a better user experience.

4.2.3 Schedule Quality

The example above suggests the following factors that affect the user experience in the way 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:

$$Cost(S) = Cost_t(S) + Cost_m(S). \tag{4.1}$$

In the formula, $Cost_t(S)$, measures the overall time cost of the mini-queries, which can be quantified by their total running time. $Cost_m(S)$, where "m" means "smoothness," measures

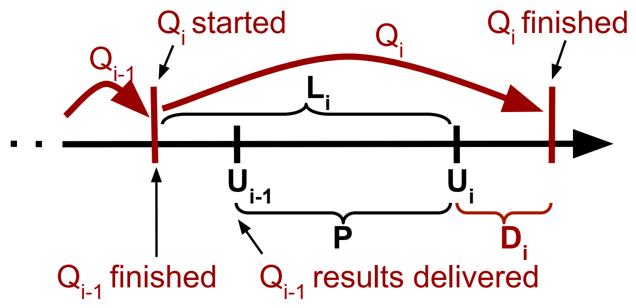


Figure 4.3: Missing a delivery deadline.

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 seconds after the previous update. Every time the middleware does not update the interface after this time P, this schedule needs to pay a "missing-the-deadline" penalty. The smoothness cost can be quantified as:

$$Cost_m(S) = \alpha \sum_i D_i. \tag{4.2}$$

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:

$$D_i = max(0, U_i - (U_{i-1} + P)), \tag{4.3}$$

where U_i and U_{i-1} are the update times when the actual results of mini-queries Q_i and Q_{i-1} are delivered to the frontend, respectively. The general idea is shown in Fig. 4.3. 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).

Table 4.1 shows the formula costs for the three schedules in Fig. 4.2 when the 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 provides.

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

Table 4.1: The costs of the schedules of Fig. 4.2

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

4.3 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. 4.4 shows the framework. Each incoming request from the frontend is submitted to the mini-query generator. The generator utilizes estimation information provided by the

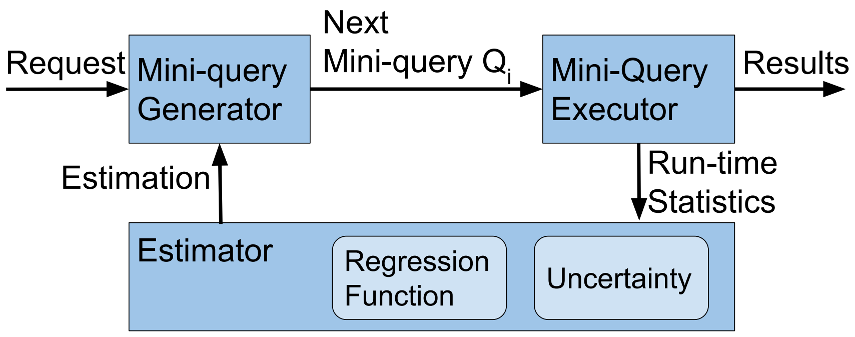


Figure 4.4: Drum framework for adaptive query slicing.

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.

4.3.1 Regression Function

The regression function in the estimator is used to capture the relationship between the slicing predicate range size and the corresponding mini-query running time. As more statistics are collected, the regression function can become more accurate. 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:

$$f(r_i) = a_1 r_i + a_0. \tag{4.4}$$

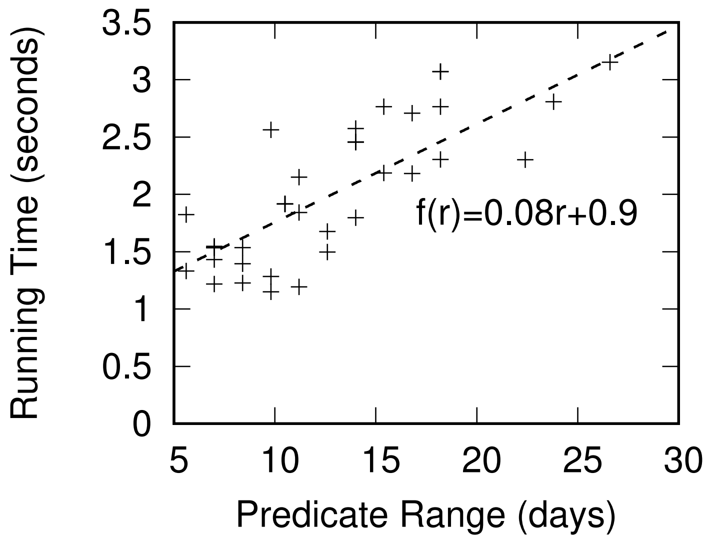


Figure 4.5: Linear regression for tweet mini-queries with different predicates on the `create_at` attribute. (110 million tweets, keyword=`zika`)

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 a `Trump` query. We can use a standard curve-fitting algorithm with the least squares fitting method to train the linear models on the collected pairs of range size and running time. Each newly added 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 4.5 shows an actual linear model learned from a collection of pairs of range size r_i and running time t_i:

4.3.2 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

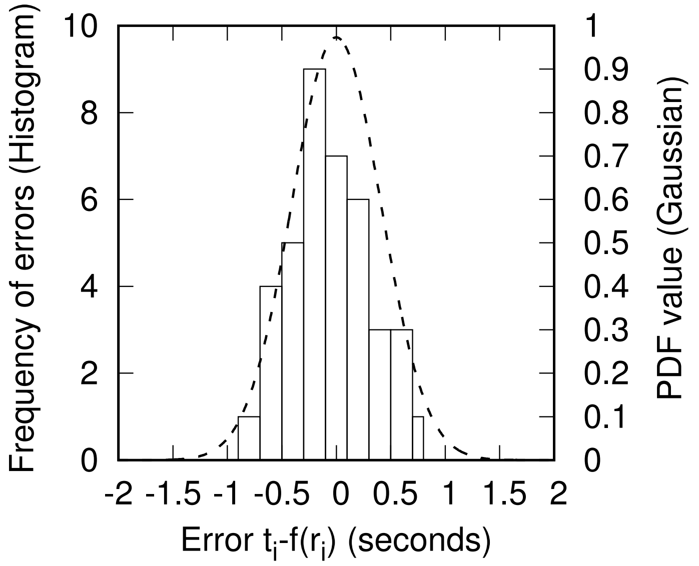


Figure 4.6: The Histogram and Gaussian models. The errors are collected from the observation $t_i - f(r_i)$ in Fig. 4.5.

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. 4.6 shows both the histogram distribution and Gaussian distribution for the errors $t_i - f(r_i)$ collected in Fig. 4.5. 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.

4.3.3 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:

$$score(Q_i) = \begin{cases} \frac{r_i}{I} & \text{if } t_i <= L_i \\ \frac{r_i}{I} - \alpha\frac{t_i - L_i}{C_i L_i} & \text{otherwise.} \end{cases} \tag{4.5}$$

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. 4.3, 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 the constant weight in the penalty function in Equation 4.2.

- 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 on 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 is a constant.

Using the linear function in Fig. 4.5 and the error PDF from Fig. 4.6, Fig. 4.7 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. 4.7(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 have to pay a penalty for missing the L_i deadline. Fig. 4.7(b) shows the PDF of the running time t_i that is of the same shape as shown in Fig. 4.6, but the mean of the distribution is $f(r_i) = 2.2$ instead of 0.

Fig. 4.7(c) shows the score function for another strategy A_2 that picks a range $r'_i = 12$ days and the target running time $f(r'_i) = 1.85$ seconds, which is more conservative by using a value smaller than L_i. Similar to the previous case, when the real time t_i is less than L_i (2.2 seconds), the score is a constant (which is smaller than the value when we had $f(r_i) = 2.2$). When the real time t_i is larger than L_i, the score value constantly decreases. Fig. 4.7(d) shows the PDF of the running time t_i when r'_i is 12 days. As a consequence, compared to the previous $f(r_i)$, the PDF "shifts" to the left, providing a larger probability of having results to deliver to the user by the L_i deadline. When comparing these two strategies, we notice that strategy A_1 is more aggressive by choosing a target running time $f(r_i) = 2.2s$ that is equal to the time limit L_i, while A_2 is more conservative by using a smaller time $f(r'_i) = 1.85s$ (with a smaller slicing range r'_i). As a consequence, A_2 has a lower progress r'_i/I, but it also has a lower probability of missing the deadline.

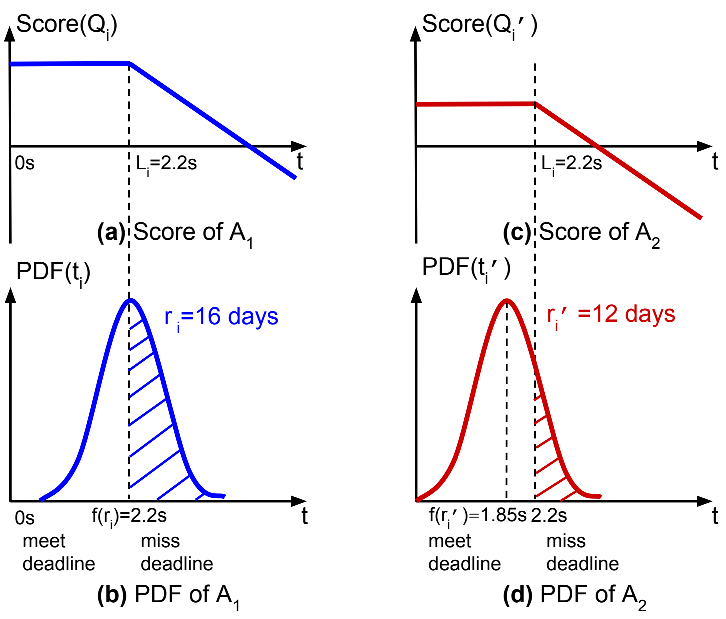


Figure 4.7: The score and error PDF for two target running times, $r_i = 2.2$ seconds and $r_i' = 1.85$ seconds, with time limit $L_i = 2.2$ seconds, $f(r_i) = 0.08r_i + 0.9$, and $PDF(t_i) = N(f(r_i), 0.4)$.

4.3.4 Choosing the 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_is 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:

$$\begin{aligned} E(score(Q_i)) &= \int_0^{\infty} score(Q_i)P(t|f(r_i))dt \\ &= \int_0^{L_i} \frac{r_i}{I}P\big(t|f(r_i)\big)dt + \int_{L_i}^{\infty}\Big(\frac{r_i}{I} - \alpha\frac{(t-L_i)}{C_iL_i}\Big)P\big(t|f(r_i)\big)dt \\ &= \frac{r_i}{I} - \alpha\int_{L_i}^{\infty}\frac{(t-L_i)}{C_iL_i}P\big(t|f(r_i)\big)dt. \end{aligned} \tag{4.6}$$

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

4.4 Choosing an Optimal Predicate Range

We now study how to choose a predicate range r_i for the next mini-query Q_i in order to maximize the expected score $E(score(Q_i))$. 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.

4.4.1 Histogram

Construction

A histogram of the estimated error distribution can be constructed as follows. Take an equi-width histogram as an example. We first choose its error bin size b. After running each mini-query Q_j, we measure its real execution time t_j and compute the difference between t_j and the estimated time $f(r_j)$. We identify the bin corresponding to this error, $t_j - f(r_j)$, and increment the frequency of the corresponding bin.

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 4.6 to compute the integral of the score using the error distribution. In particular, before the time limit L_i there is only

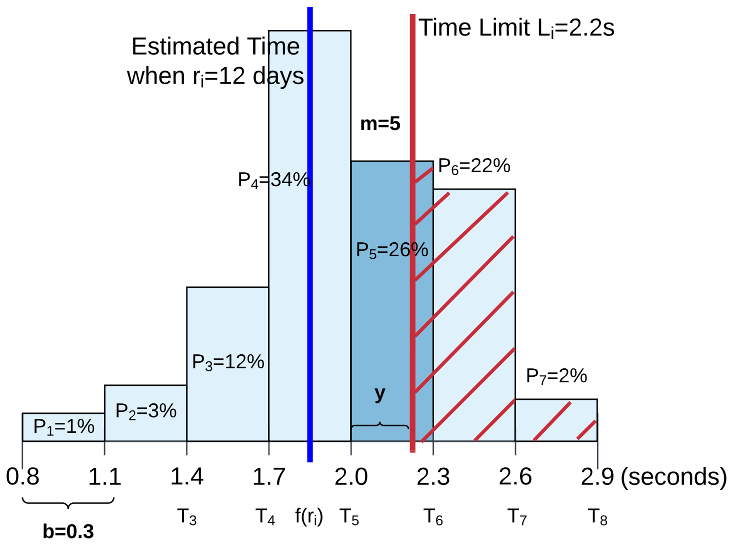


Figure 4.8: Use an error histogram to compute the expected score for targeting running time $f(r_i)$.

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 $f(r_i)$ with L_i, the L_i could fall into different bins with different probabilities, which results in different score computations.

Using the previous example of $L_i = 2.2$ seconds and $f(r_i) = 0.08r_i + 0.9$, Fig. 4.8 shows the histogram distribution of the running time t_i when $r_i = 12$ days and $f(r_i) = 1.85$. The limit L_i falls into the 5-th bin in the histogram, with error interval $[2.0, 2.3]$. The left four bins are all less than L_i, so they will not be contribute to the penalty. The two rightmost bins $[2.3, 2.6]$ and $[2.6, 2.9]$ are both larger than L_i. We thus need to multiply their probability with the linear penalty function to compute the integral. Within the interval bin $[2.0, 2.3]$, we assume the error is uniformly distributed, and can deal with the two pieces divided by the L_i limit separately to compute the integral as the expected score. At the end, we take the summation of these three cases as the overall expected score.

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:

$$E(r_i) = \frac{r_i}{I} - \frac{\alpha}{C_i L_i}\Big(\int_{L_i}^{T_{m+1}} (1-\frac{y}{b})\frac{Pr_m}{b}(t-L_i)dt + \sum_{k=m+1}^{n} \big(\int_{T_k}^{T_{k+1}} \frac{Pr_k}{b}(t-L_i)dt\big)\Big), \quad (4.7)$$

where

- 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 that time limit L_i falls into, i.e., $T_m \le L_i < T_{m+1}$;
- y: the time difference $L_i - T_m$.

For a specific range of r_i that makes L_i fall into the m-th bin, Equation 4.7 can be transformed to:

$$E(r_i) = \frac{r_i}{I} - \frac{\alpha}{C_i L_i}\Big(\frac{Pr_m}{2b^2}(b-y)^3 - \sum_{k=m+1}^{n} Pr_k y + b\sum_{k=m+1}^{n} Pr_k(\frac{1}{2}+k-m)\Big). \quad (4.8)$$

Computing r_i to maximize the expected score

Fig. 4.9 shows how the expected score changes as we increase the range r_i for the histogram distribution in Fig. 4.8 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

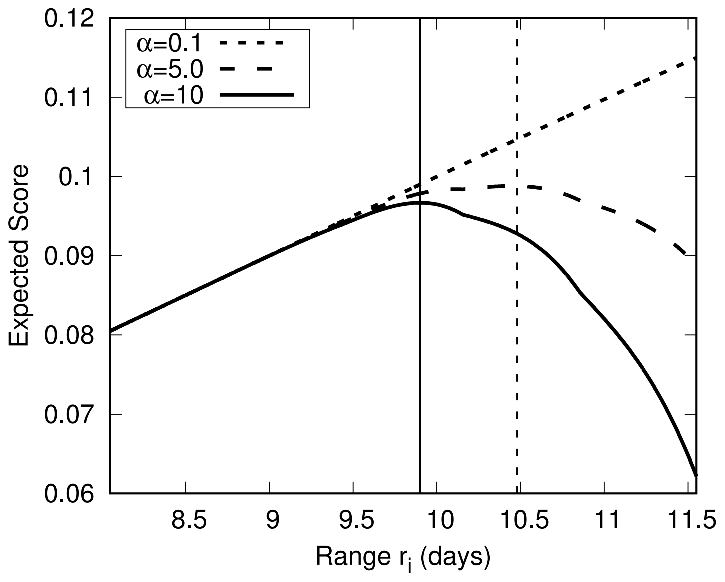


Figure 4.9: Expected score as r_i changes (histogram model).

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.

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 4.8 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 (see the Appendix A for more details):

$$(b - y_{max})^2 = \frac{2b^2}{3Pr_m}\left(\frac{C_i L_i}{\alpha a_1 I} - \sum_{k=m+1}^{n} Pr_k\right). \tag{4.9}$$

Let g be the id of the bin that covers $f(r_i)$. Using Equation 4.9 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 4.4.

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

Table 4.2: Maximal expected score for each interval bin

Notice that the expectation score in Equation 4.8 will change based on which bin the 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. 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 4.2 shows the maximum expected score for each interval bin in the example in Fig. 4.8.

To terminate the search for the maximum progress earlier, we can start by being aggressive and choosing the target time $f(r_i)$ close to the time limit L_i. We can gradually decrease the $f(r_i)$ value by the bin size b. As a consequence, the progress will decrease, while the penalty also decreases, and the overall expected score increases. We can keep decreasing $f(r_i)$ until we reach an interval bin where the expected score starts to decrease. Then we choose the best $f(r_i)$ in this bin as the final value.

4.4.2 Gaussian Distribution

Assuming that the estimated time error $t_i - f(r_i)$ follows a Gaussian distribution $N(0, \sigma)$, 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 4.6 with $N(f(r_i), \sigma)$ and get the following function:

$$E(r_i) = \frac{r_i}{I} - \alpha \int_{L_i}^{\infty} \frac{t - L_i}{C_i L_i} \frac{1}{\sqrt{2\pi}\sigma} e^{-(\frac{t - f(r_i)}{\sqrt{2}\sigma})^2} dt. \tag{4.10}$$

This equation can be transformed to

$$E(r_i) = \frac{r_i}{I} - \frac{\alpha\sigma}{C_i L_i \sqrt{2}} \Big(\frac{1}{\sqrt{\pi}} e^{-z^2} + z\big(1 + \mathrm{erf}(z)\big)\Big), \tag{4.11}$$

where

$$z = \frac{f(r_i) - L_i}{\sqrt{2}\sigma}. \tag{4.12}$$

To compute an r_i value to maximize the expected score, we need to find a value that makes the derivative of Equation 4.11 be 0, i.e., $E'(r_i) = 0$. The derivative can be transformed to

$$E'(r_i) = \frac{1}{I} - f'(r_i) \frac{\alpha}{2 C_i L_i} \big(1 + \mathrm{erf}(z)\big). \tag{4.13}$$

Using the estimation model in Equation 4.4, we can compute r_i as

$$r_i = \frac{\sqrt{2}\sigma z_{max} + L_i - a_0}{a_1}. \tag{4.14}$$

When

$$\alpha > \frac{2 C_i L_i}{a_1 R}, \tag{4.15}$$

we have

$$z_{max} = \text{erf}^{-1}(\frac{2C_iL_i}{a_1I\alpha} - 1). \tag{4.16}$$

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 Equation 4.14 and Equation 4.16 to compute r_i to achieve the maximal expected score.

4.4.3 An Example Sequence of Mini-queries

In this section, we use an example to illustrate how the Drum framework decides the sequence of mini-queries by using the adaptive regression function and the uncertainty model.

Mini-query	L_i	r_i	$f(r_i)$	t_i	error	σ	new $f(r)$
Q_1	2.0s	1.0h	0.74s	0.5s	-0.24s	0.2623	-
Q_2	3.5s	2.0h	1.13s	1.5s	0.37s	0.2623	-
Q_3	4.0s	4.0h	1.92s	1.8s	-0.12s	0.2623	$0.392r + 0.350$
Q_4	4.2s	9.2h	3.96s	3.0s	-0.96s	0.5294	$0.269r + 0.611$
Q_5	3.2s	7.3h	2.57s	3.2s	0.63s	0.5503	$0.302r + 0.578$
Q_6	2.0s	1.8h	1.12s	1.5s	0.38s	0.5255	$0.286r + 0.710$
Q_7	2.5s	...	...	...	...	...	...
...	...	...	...	...	...	...	...

Table 4.3: A sequence of mini-queries generated adaptively based on the latest statistical information. ($\alpha = 25$)

All Drum variants start with three mini-queries with fixed time ranges of one, two, and four times the minimal range, respectively, to accumulate the initial statistics, and then begin the adaptive process. An example of possible running times t_i for the first three mini-queries is shown in Table 4.4.

Based on the first three observations of range size r_i and running time t_i, we can learn the linear function $f(r_i)$, and use it to calculate errors $(t_i - f(r_i))$ to build the uncertainty

Mini-queries	r_i	t_i (seconds)
Q_1	1h	0.5
Q_2	2h	1.5
Q_3	4h	1.8

Table 4.4: The first three mini-query results

model. For simplicity, we use the Gaussian distribution model that only requires the standard deviation (σ) of the errors to estimate the uncertainty. Table 4.3 shows the chosen range size r_i of each mini-query and how the model is adaptively updated based on newly arriving statistical information. Taking Q_4 as an example, after the first three mini-queries, we got the linear function $f(r) = 0.392r + 0.350$ and the uncertainty model $N(f(r), 0.2623)$. Given the time limit of 4.2 seconds, using Equation 4.14 we select the range size $r_4 = 9.2h$ for Q_4, which is predicated by the model to finish in 3.96 seconds. When Q_4 actually finished in 3.0 seconds, we calculate the error ($3.0s - 3.96s = -0.96s$), then update the standard deviation of the Gaussian model to $\sigma = 0.5294$. Meanwhile, based on the new observation of range size r_4 and running time t_4, we update the linear function as $f(r) = 0.269r + 0.611$, then use the new $f(r)$ and σ to choose the next range size for Q_5. This online learning process continues until the mini-queries cover the entire value space of the slicing attribute.

4.5 Experiments

In this section we report the results of an empirical evaluation of the proposed Drum framework and its adaptive estimation techniques.

4.5.1 Setting

We collected 114 million tweets using the Twitter streaming API from November 14, 2016 to January 17 2017. We used Apache AsterixDB [13] (0.9.2 version) as the backend database to store the data. Its schema included the following attributes: *(**id**: int, **create_ at**: datetime, **text**: 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 the 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 in the following format:

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

In the query, the predicate `ftcontains()` checks if the *message* attribute includes a given keyword. We considered different types of queries regarding their keyword condition:

1. NoKeyword: The query did not have an `ftcontains` predicate;
2. Uniform keyword: The keyword in the query was distributed uniformly over the whole time range. 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 *text* field to speed up these keyword-search queries. We used the "`create_at`" field as the slicing attribute, which was the creation timestamp of the tweet. We also used this field to do filtering for the AsterixDB indexes [15], 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.

4.5.2 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 4.7) to decide the next predicate range;
5. Drum with a Gaussian error model ("DRUM-GA"): we used a Gaussian error model (Equation 4.10) to decide the next predicate range.

All Drum methods started with 3 mini-queries with a fixed creation timestamp 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 desired rhythmic response pace to be 2 seconds.

As explained in Section 4.1, a good fixed length is hard to decide for the FL method. To give it a fair (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 evaluate 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. 4.10a. 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 depending on the observed performance numbers of the database system. For example, keywords `election` and `rain` were very selective, and their corresponding original queries would access fewer records than the query of the popular keyword `happy`. The Drum framework utilized the collected performance information and picked a bigger predicate range for these two queries. As a result, their numbers of mini-queries were much less than that of the `happy` query and that of the query without keywords. Among the three Drum methods, the DRUM-HS and DRUM-GA methods considered the uncertainty of the regression, thus were more conservative when generating the next predicate range and issued more mini-queries than DRUM-BL.

Total running time

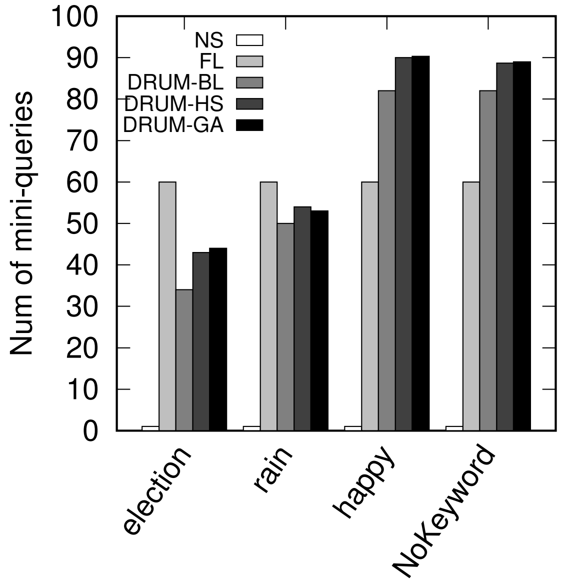
Fig. 4.10b shows the total running time of the schedules generated by different models. The NS method had the least total running time because it did not do any slicing and thus did not pay any overhead. The FL method sent 60 mini-queries and had a slightly longer total time. The times of all the DRUM methods were comparable, and they did not increase the running time much.

Total delay of missing milestones

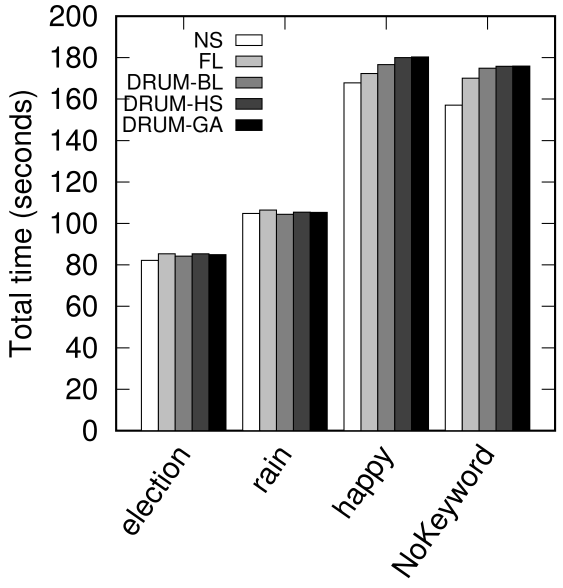
We next evaluated the different methods in terms of their total delay time. The results are shown in Fig. 4.10c. The NS method had the longest delay because it did not give the user any updates until the whole query was finished. The delay time of FL was always 0 for the `election` and `rain` requests, 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 DRUM-GA and DRUM-HS were much smaller.

Total cost

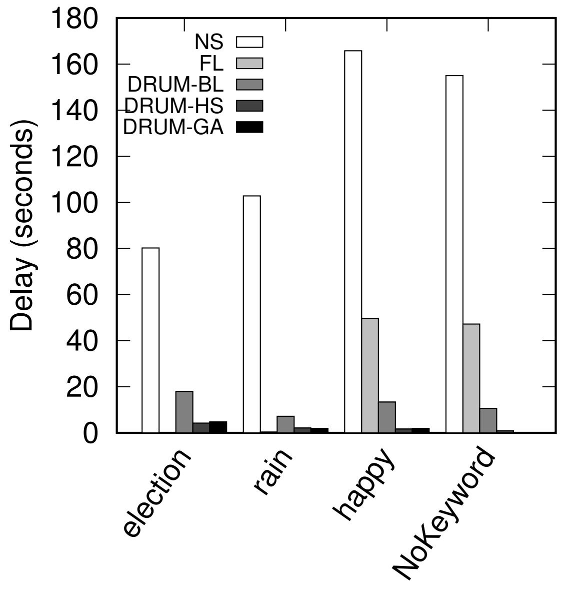
Fig. 4.10d shows the overall cost of a schedule generated by each method as defined in Equation 4.1, where the weight was set to $\alpha = 25$ to reflect a case where the client is concerned more about the pace of result delivery than the total running time. The NL method had the highest cost due to 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



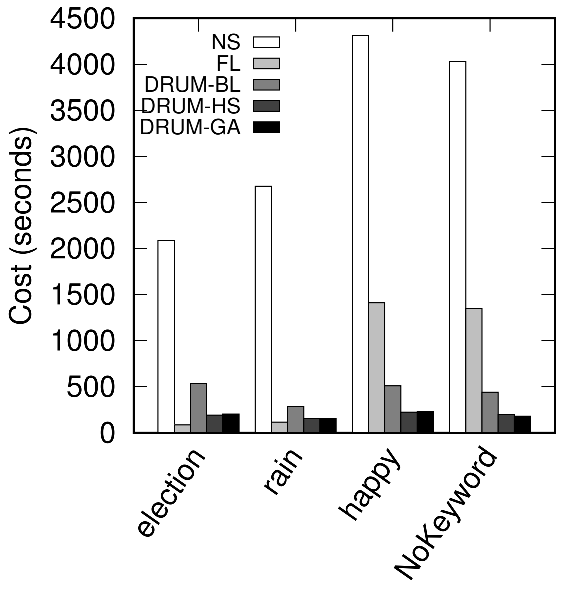
(a) Number of mini-queries



(b) Total time

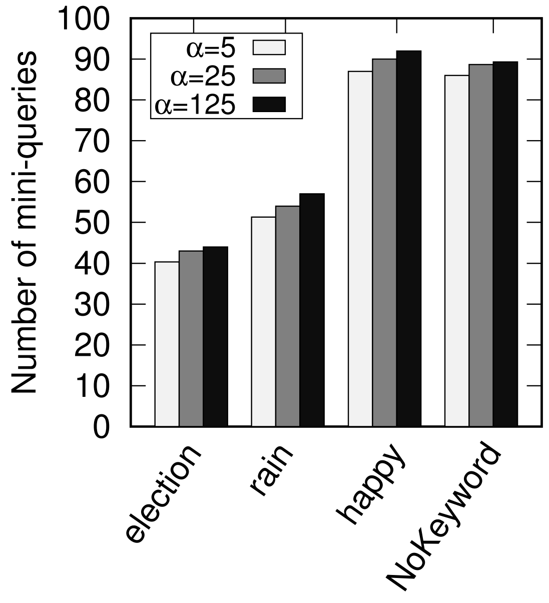


(c) Total delay (FL is 0 in `election` and `rain` cases)

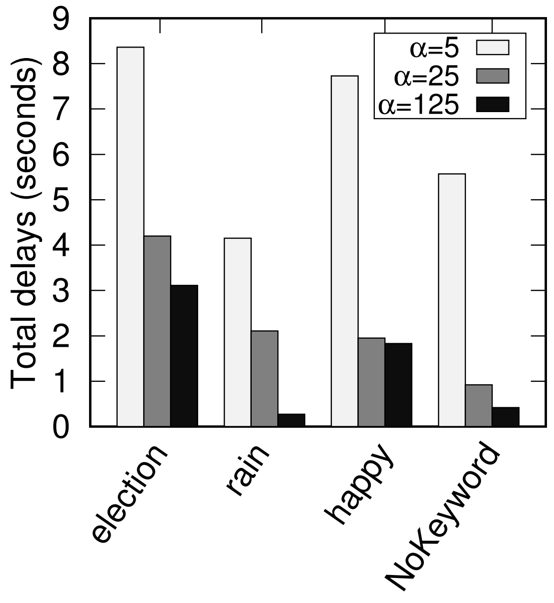


(d) Total cost

Figure 4.10: Evaluating slicing methods



(a) Number of mini-queries



(b) Delay

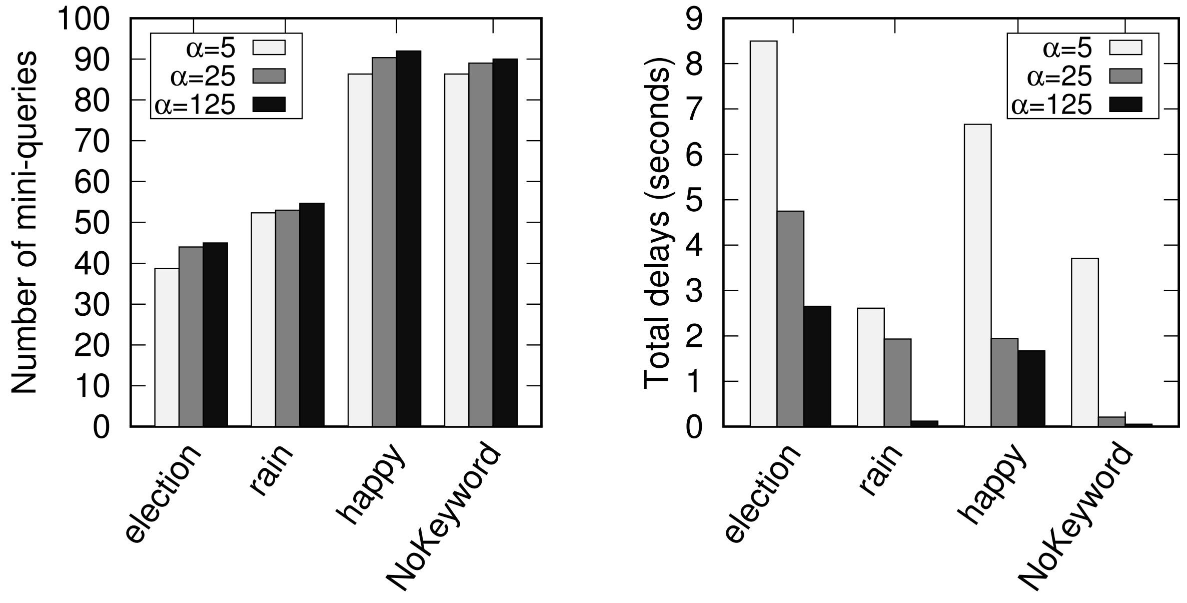
Figure 4.11: Effect of penalty weight α (DRUM-HS)

milestones, so their costs were the lowest.

These experiments showed that there was not 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.

4.5.3 Effect of Penalty Weight α

We evaluated the effect of the penalty weight α in the scoring function of Equation 4.2. Intuitively, a higher α value means a higher penalty for missed 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. 4.11a shows the number of mini-queries when using the DRUM-HS method. When α increases, the number of mini-queries also increases since each mini-query became



(a) Number of mini-queries (b) Delay

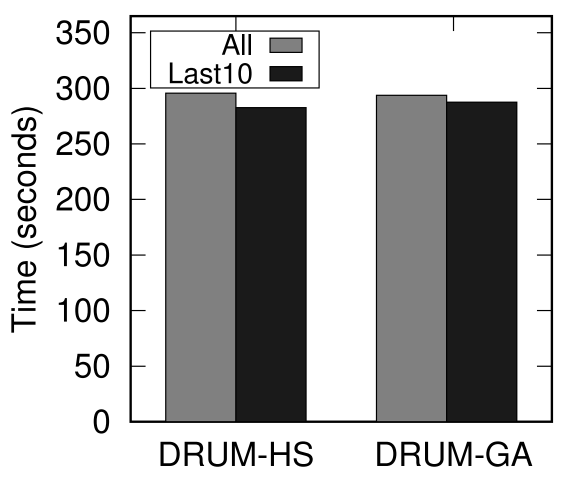
Figure 4.12: Effect of penalty weight α (DRUM-GA)

more conservative (with a smaller range). Meanwhile, the total delay decreased due to the smaller predicate ranges, as shown in Fig. 4.11b. The DRUM-GA results for different α values are shown in Fig. 4.12.

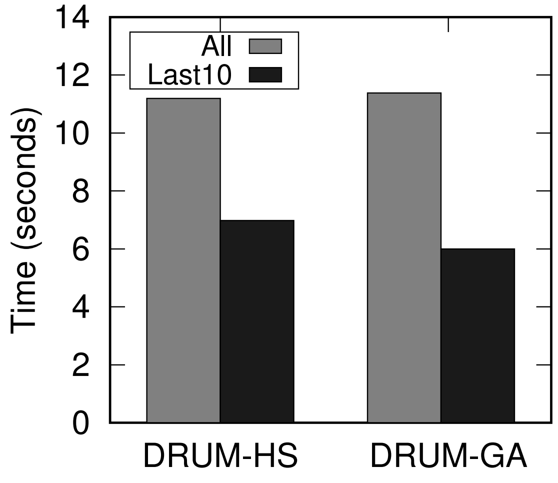
4.5.4 Adaptiveness of Regression Function

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 well the Drum framework can adapt to changes in the performance of the underlying database system. We considered the *NoKeyword* query, whose results were distributed evenly over the entire time range.

We first simulated a situation with a dynamic database workload. The first half of the mini-queries were run normally, but after some time we added a 1-second sleep to the database



(a) Total running time



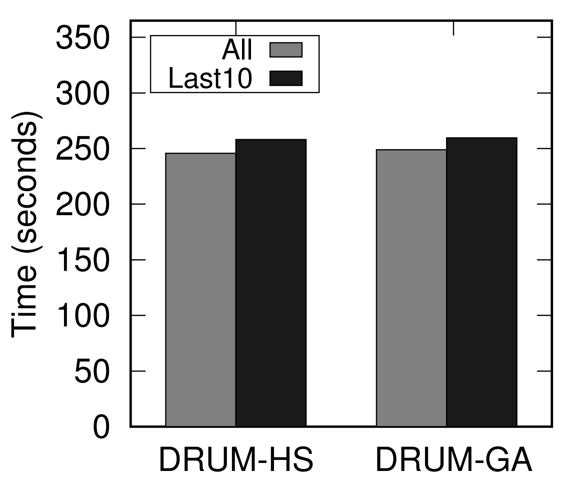
(b) Total delay

Figure 4.13: Adaptiveness of the linear regression (adding a 1-second sleep for the second half of the mini-queries

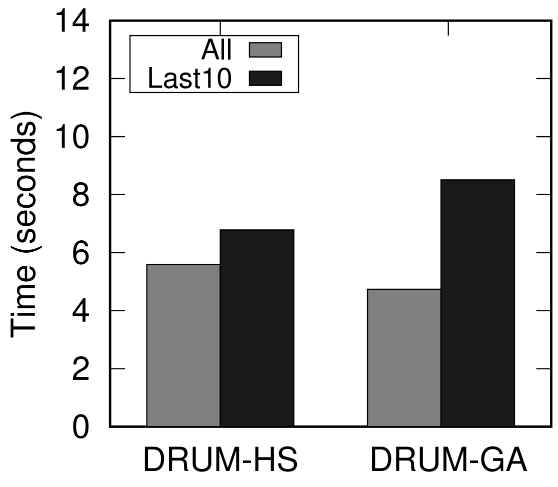
connection before sending each result back to the mini-query executor in order to simulate a case where the second half of the mini-queries took longer than the mini-queries with the same time ranges in the first half. Fig. 4.13 shows the results. The Last10 method was able to adapt to the new performance of the database, since its obtained linear regression better described the relationship between the predicate range and the running time.

We then simulated a situation where the database system is less stable. In this case we added a 1-second sleep randomly (with probability of 0.5) in the database connector for each mini-query. Fig. 4.14 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 help Drum to adapt to new backend performance changes as expected. 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 larger penalties).



(a) Total running time



(b) Total delay

Figure 4.14: 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.

4.6 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 [46, 42, 45, 64]. For example, [46, 42] 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. [69] introduced another method for improving the user experience by showing partial results, which also requires changes to the database system. [28, 64] studied online aggregation in a MapReduce-like framework. [53, 82] studied how to provide aggregation results with different levels of details 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.

[81] 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 [32, 76]. 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 [76, 88]. Thus, the rhythm of result delivery is a key determinant related to user satisfaction in progressive computing. Deciding a good amount of waiting time for the next incremental update has been studied in [32, 21, 85]. [34] 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" [29], resulting in a good experience of distorted time perception [76].

Query time estimation: Estimating the running time of a query is a fundamental topic that has been extensively studied (e.g., [8, 91]). Many of the existing techniques can be adopted in our Drum framework as part of the regression function. [92] considered an uncertainty model that relied on internal information about a database system that may not be available in many applications. [4] implemented a system called BlinkDB that allows users to choose a trade-off between query accuracy and response time. [93] considered an uncertainty model for admission control of multiuser workloads, 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.

4.7 Conclusions

In this chapter, 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. Its goal key is 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.

Chapter 5

Using Filters to Improve Secondary-to-Primary Index Search

5.1 Introduction

Fast query response time is critical for a good user experience. In the previous query slicing work, Cloudberry slices a query into a sequence of mini-queries by adding time-range predicates to the original query so that the mini-queries can run much faster to support a responsive user experience. It is always desirable to improve the query processing speed so that there will be fewer mini-queries needed, and hence less time for the frontend to receive the complete results.

To improve mini-query performance, the TwitterMap application benefits from the LSM filter feature [15] provided by AsterixDB to speed up the time-related queries. AsterixDB's LSM filters provides a way to add additional metadata to LSM components and to exploit the metadata during query processing to skip irrelevant components. In order to efficiently utilize the power of the filters, the query needs to have a highly selective filter-related predicate.

This limitation largely restricts the scenarios that can utilize the pruning power of the filters. Unfortunately, many analytical queries do not match this pattern. For example, a mini-query can have a wide time-range predicate, and many index components will be matched by checking their filters. In such cases, the query time can not be reduced much.

In this chapter, we revisit the filter idea, explore the aligned structure property between secondary index components and primary index components, and propose an improvement for accelerating the secondary-to-primary search. The main idea is to propagate the filter information attached to a searched secondary index component along with its output primary keys to the primary index search. The primary index search can then still use filter hints to improve the speed, even though the query itself does not contain any filter-related predicates.

The rest of the chapter is organized as follows. First, we revisit the background of LSM-trees and the filter concept in Section 5.2. Then we introduce the idea of acceleration using secondary components' filters in Section 5.3. We present the evaluation of the improvement in Section 5.4. Finally, we provide conclusions in Section 5.6.

5.2 Background

5.2.1 LSM-Tree

An LSM-tree [63] is an ordered, persistent index structure that supports typical operations such as insert, delete, and search. It is optimized for frequent or high-volume updates. By first batching updates in memory, the LSM-tree amortizes the cost of an update by converting what would have been several disk seeks into some portion of a sequential I/O. Entries being inserted into an LSM-tree are initially placed into a component of the index that resides in main memory, called an in-memory component. As shown in Fig. 5.1, when the space

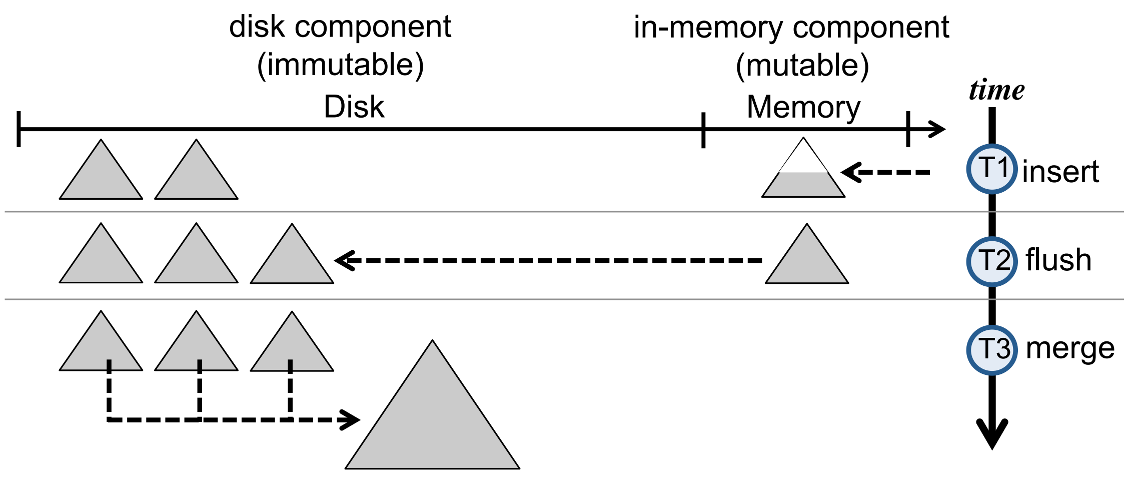


Figure 5.1: Life cycle of a component-based LSM-tree [51]

.

occupancy of the in-memory component exceeds a specified threshold, entries are sequentially flushed to disk. As entries accumulate on disk, the entries are periodically merged together subject to a policy that decides when and what to merge.

5.2.2 Filter-based LSM Storage in AsterixDB

5.2.2.1 LSM Storage in AsterixDB

The AsterixDB's storage layer [14] provides a general framework for converting a class of indexes (including conventional B-trees, R-trees, and inverted indexes) into LSM-based indexes.

Fig. 5.2 shows a snapshot of the LSM storage after inserting the records from Table 5.1. Each record contains the `id` field that stores the primary key, the `time` field that collects the timestamp, and the `text` field that holds the tweet content. After ingestion, there is one primary LSM B-tree index and one secondary inverted LSM index built on field `text`. The in-memory component of the primary LSM B-tree stores entries of $\langle pk, record\rangle$ pairs and orders them by the primary key (noted as "*pk*"). There are multiple on-disk components of

id	time	text
1	25	`Zika virus symptoms and risks`
2	34	`Rio Olympics`
3	35	`Flu virus`
4	47	`Zika cases`
5	48	`Zika in mosquitos`
6	67	`News about Olympics`
7	69	`Ebola virus`

Table 5.1: A sample dataset `tweet`. The `id` is the primary key. The value of the `time` field has a timestamp number. The `text` field contains a bag of words.

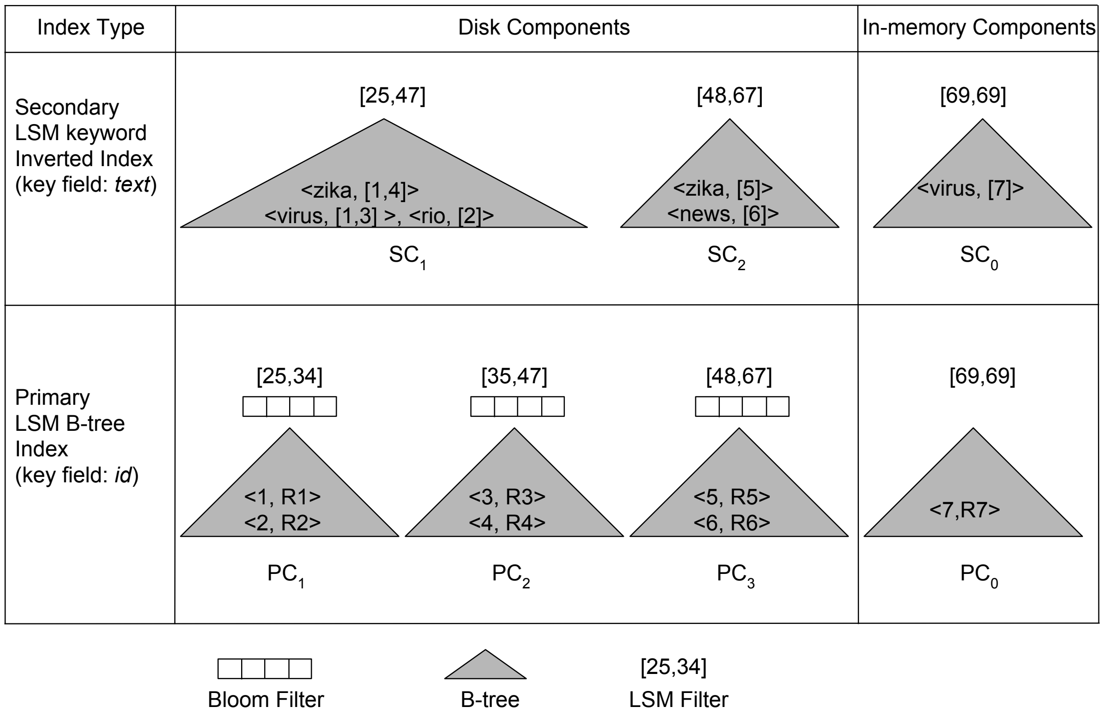


Figure 5.2: LSM storage example in AsterixDB.

the primary LSM B-tree that are formed by flushing an in-memory component or by merging multiple smaller on-disk components. Particularly for the case of the primary index, a disk component consists of a B-tree with an associated Bloom filter that is built by the primary keys stored in the specific component. The Bloom filter is used to skip irrelevant B-tree components when looking for a specific primary-key.

Similarly, the secondary index consists of an in-memory index structure and several on-disk index structures. Different from the primary index, a secondary disk component does not have Bloom filters. Logically, the secondary index stores entries of $\langle sk, pk \rangle$, where sk represents a secondary key. Depending on the type of the index, the entries may be stored in different formats. For example, in the case of inverted index, all *pk*'s associated with sk are physically stored as a list in a separate inverted list file. Then the B-tree structure stores $\langle sk, pointer \rangle$, where the *pointer* stores the offset of the list address in the inverted list file.

5.2.2.2 LSM Filter

The LSM filter is an additional piece of metadata added to each LSM component that stores the minimum and maximum values of a selected attribute (called the filter key) among the records in the component [15]. If a query has an predicate on the filter key, the query executor can exploit the metadata to skip irrelevant components.

Users of filters are advised to use time-correlated fields or a field that is monotonically increasing over the order of record arrivals as their filter field. In this case, the filters on the disk components are likely to have disjoint filter ranges, making them very effective for pruning. Fig. 5.3 shows an example of the SQL++ DDL to declare the `time` field filter of the `tweet` dataset in AsterixDB.

As shown in Fig. 5.2, after declaring the filter field, both primary and secondary index components contain an LSM filter. The in-memory component incrementally maintains the

```
CREATE DATASET tweet(tweetType) PRIMARY KEY id WITH FILTER ON time;
```

Figure 5.3: SQL++ DDL statement to declare the filter field.

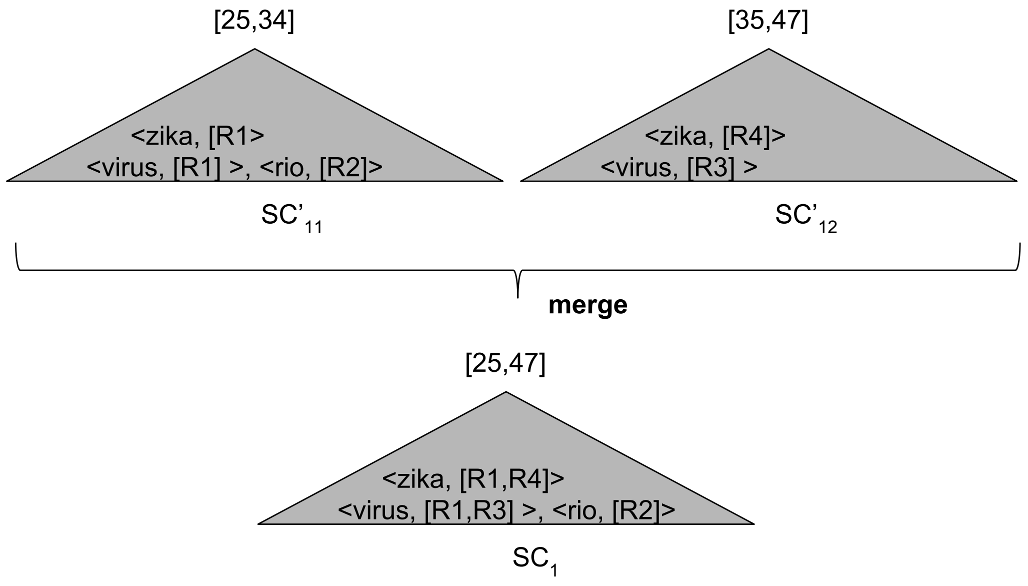


Figure 5.4: Update the filter range when merging two components.

minimum and maximum filter key values based on newly inserted values. When the in-memory component is flushed to disk, its filter record is also flushed to disk. During the merge phase, the minimum and maximum values for the resulting component's filter record are taken from the smallest and largest values of all components participating in the merge. For example, as shown in Fig. 5.4, component SC_1 is created by merging the previous two on-disk components SC'_{11} and SC'_{12}. The filter range of the merged component is set to cover both filter ranges of the previous components.

5.2.2.3 Filter-based Secondary-to-Primary Search

When a query comes to the system, the AsterixDB query optimizer first checks if the dataset has a filter. If so, then it analyzes the query to find if there are any filter-related predicates. Once the optimizer finds applicable predicates, it will pass them to all the index search

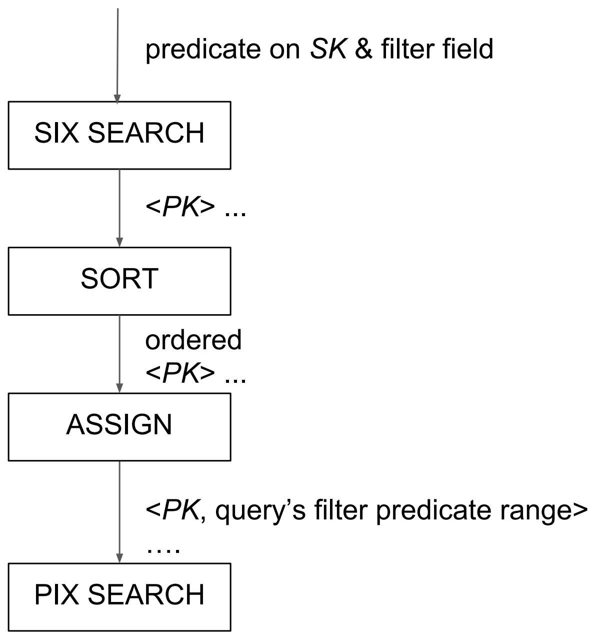

predicate on *SK* & filter field
SIX SEARCH
<*PK*> ...
SORT
ordered <*PK*> ...
ASSIGN
<*PK*, query's filter predicate range> ....
PIX SEARCH

Figure 5.5: The query plan of secondary-to-primary index search using a filter. "SIX" stands for secondary index search. "PIX" stands for primary index search.

```
SELECT t FROM tweet t
WHERE t.time > 20 AND t.time < 30
AND ftcontains(t.text, "zika");
```

Figure 5.6: SQL++ query with a filter condition.

operators. At runtime, the index search operator uses the filter predicate to prune irrelevant components before searching them. Then a regular search on the index is performed only for the components that match the query's filter predicates. Both the primary and secondary index search operators use the filter in the same way to skip unnecessary searches.

Fig. 5.5 shows the query plan for filter-based secondary-to-primary search. The secondary index is searched first to return all the matching primary keys. Then the keys are sorted and sent to the assign operator to attach the filter predicate ranges from the query. Finally, the primary index search operator searches for all primary keys one by one, and the associated filter predicate can be used to prune the irrelevant primary components during this process. Fig. 5.7 shows an example of the corresponding data flow when the query in Fig. 5.6 is issued.

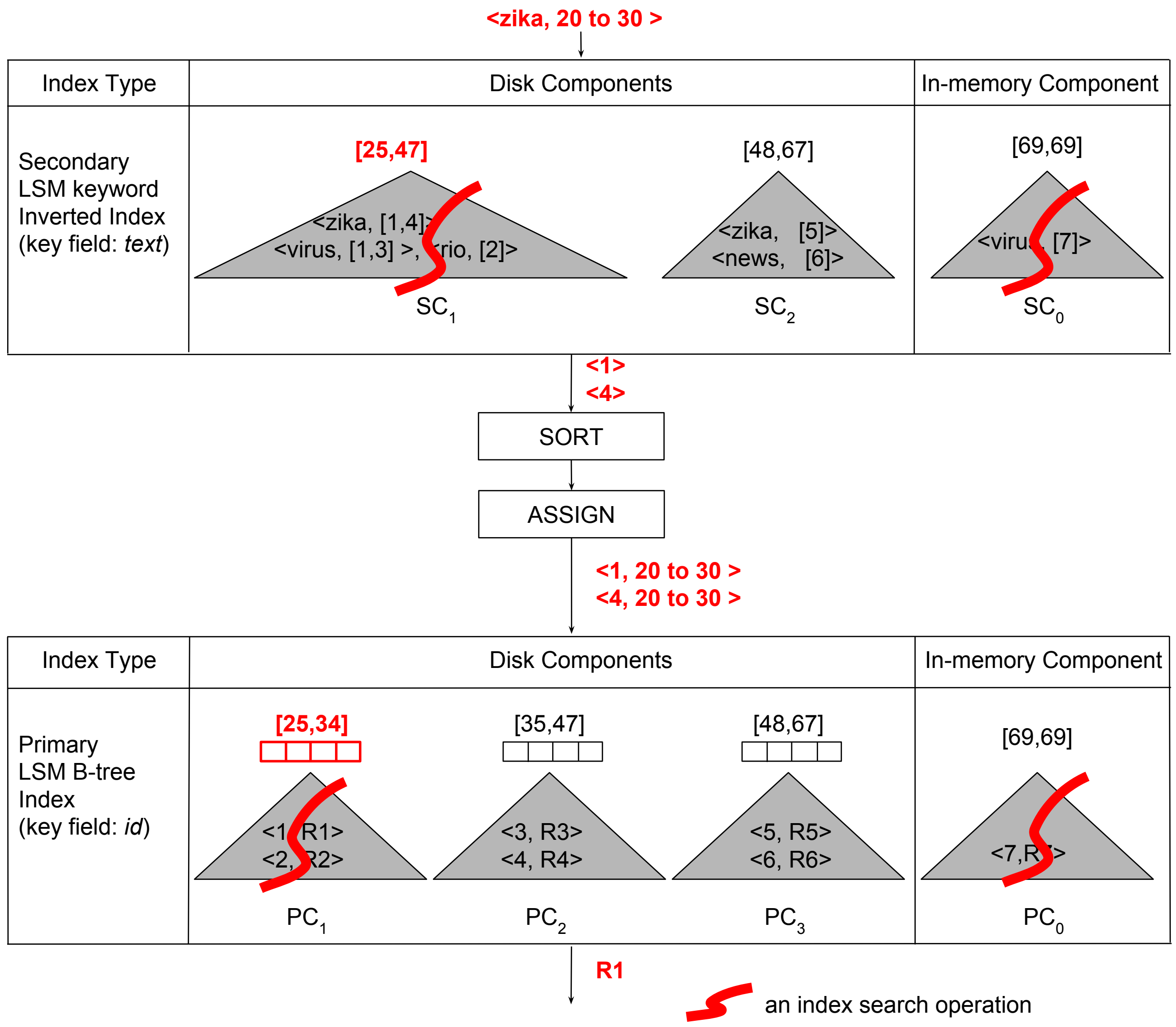


Figure 5.7: The physical data flow of filter-based secondary-to-primary search

The query optimizer finds a full-text search predicate `ftcontains(t.text, "zika")` that can utilize the inverted index on the `text` field and another predicate (`[20,30]`) that can utilize the LSM filter on `time` field of the `tweet` dataset. At runtime, both the keyword and the time range predicates are sent to the inverted index to find the matching primary keys. Given the filter predicate, SC_2 will be filtered out because its range `[48,67]` does not overlap with the query predicate `[20, 30]`. The in-memory component is always selected because it can include update and delete operations. After the secondary index search, only the primary keys 1 and 4 are returned after searching SC_1. The output records are sent to the sort operator, then through the "assign" operator to attach the same filter predicate range from the query again. Finally, the primary search operator receives a collection of primary key *pk*'s. The collection will be "unnested" in the sense that each *pk* will be searched independently by using a point lookup in the primary B-tree index. Hence, there will be a sequence of key searches instead of one search as in the secondary index search case. For each *pk* search, the filter predicate from the query is again used to prune irrelevant disk components. For example, there is no need to search PC_2 and PC_3 since their component filter ranges do not overlap with the query predicate. Note that since each *pk* search is a point lookup, the Bloom filter will also be used to eliminate unnecessary B-tree component accesses in the primary index search. Finally, $pk = 1$ will be found in PC_1, and $R1$ will be returned from the index and pushed to downstream operators.

5.2.2.4 Merge Policies

The flushed on-disk components are periodically merged together to improve the query performance. During the process, the merge policy plays a critical role to decide when and what to merge.

The *prefix merge* policy relies on component sizes and the number of components to decide which components to merge. When the size of a component exceeds a certain limit, which

```
SELECT t FROM tweet t
WHERE ftcontains(t.text, "zika");
```

Figure 5.8: SQL++ query to count the number of tweets mentioning zika.

can be configured by users, the component is not included in the merging process. The size limit parameter is shared by all indexes that are built on a dataset, and each index is merged independently. Since the size of the secondary index is often comparatively smaller than the primary index, the number of secondary components can be much less than the number of primary components.

The *correlated-prefix* policy delegates the decision of merging the disk components of *all* the indexes in a dataset to its primary index. When the policy decides that the primary index needs to be merged, it will issue successive merge requests to the I/O scheduler on behalf of all other indexes associated with the same dataset. The end result is that secondary indexes will always have the same number of disk components as their primary index and that they will be aligned in terms of their filter ranges.

5.3 Using Filters to Improve Secondary-to-Primary Index Search

5.3.1 Basic Idea[1]

LSM filters can improve the performance of a query if the query contains a highly selective predicate on the LSM filter field [15]. Otherwise, if the query does not contain a filter-related predicate or the predicate range is not highly selective, the filter is inapplicable or of limited use.

[1]The idea is primarily credited to Yingyi Bu.

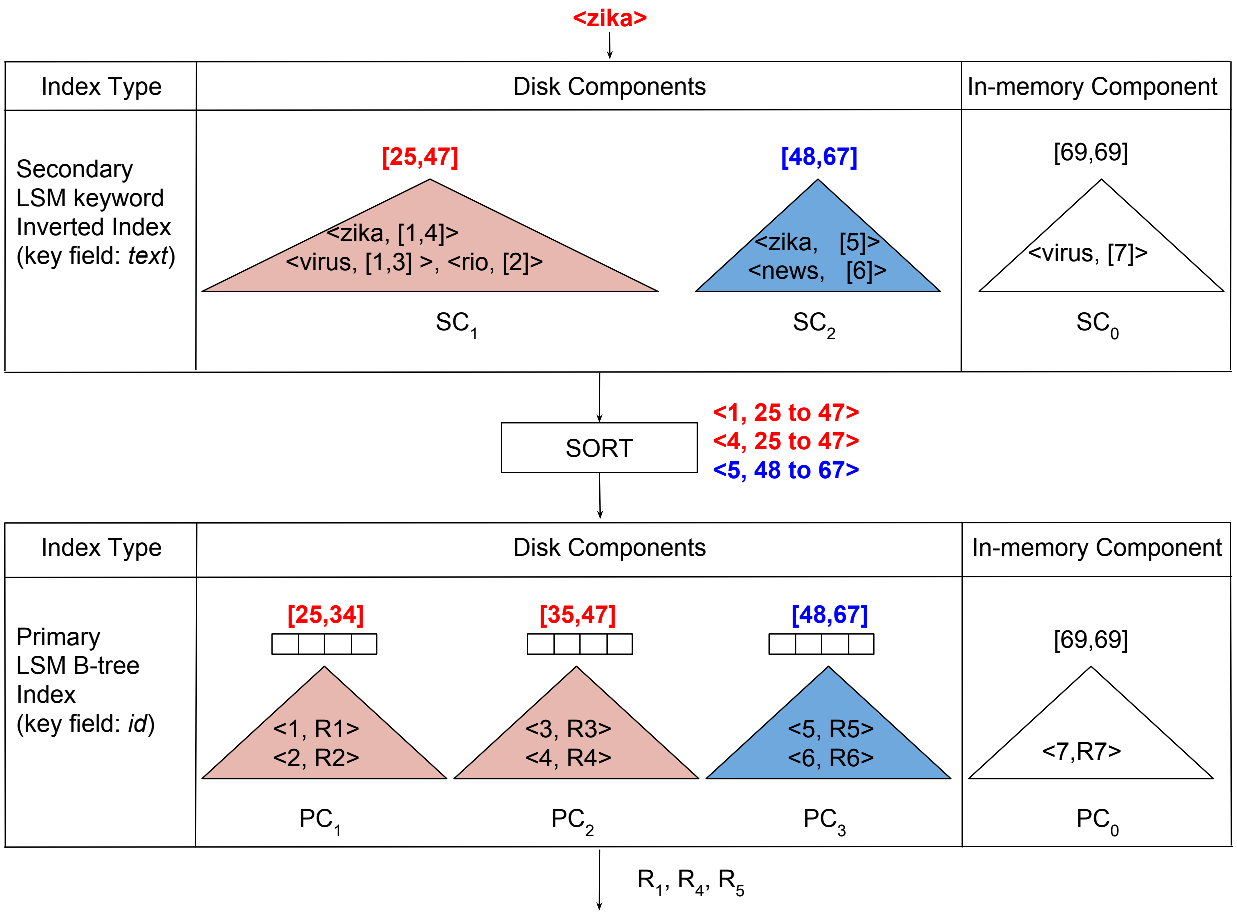


Figure 5.9: The relationship between secondary-index components with primary-index components based on their filter value. The primary key found in a secondary component does not need to be probed on the primary components whose filter values are not overlapping with the secondary component's filter.

However, in the secondary-to-primary search case, since both secondary and primary index components contain the filter metadata, the filter ranges of the secondary components can still be used to improve the primary index search regardless of whether the query has a filter related predicate or not. We illustrate the idea with an example.

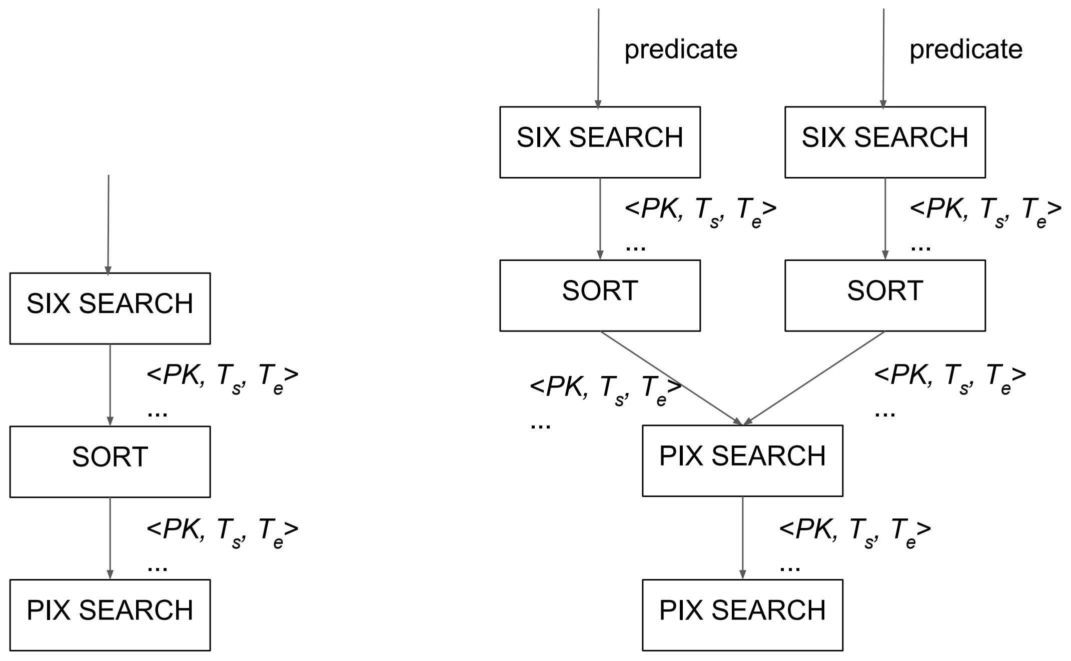
Suppose the LSM storage layout is the same as shown in Fig. 5.2. The inverted index has two on-disk components, and the primary B-tree has three on-disk components. All components have associated filters that are formed from the minimum and maximum values of the time attribute of the records within each component. The issued SQL++ query is the one shown in Fig. 5.8, which computes the number of tweets mentioning zika.

Since the query does not contain a filter predicate, all secondary components will be searched to get a list of "zika"-related primary keys (*pk*'s). After the *pk*'s arrive at the primary index search, again each *pk* will search (possibly) all the disk components in the reverse-chronical order to obtain the matching record. (The older components will not be accessed once the *pk* has been found in a more recent component).

On more careful consideration, we can observe that some of the primary components actually do not need to be searched. As illustrated in Fig. 5.9, the LSM filter ranges of the secondary and primary components reveal some relationships between them. For example, the secondary component SC_2 stores the `text` field of the tweets that were created between 48 and 67, and the filter of primary component PC_3 indicates that PC_3 only stores the records from 48 to 67. Logically, to search the matched primary key 5 from SC_2, only PC_3 needs to be searched. Similarly, to check the primary keys 1 and 4 obtained from SC_1, only PC_1 and PC_2 need to be accessed.

To utilize this relationship between the secondary and primary components, we can choose to carry the filter value from the secondary index along the data flow pipeline to the primary index search operator. The primary index search can then reuse the original filter-based pruning logic to skip the irrelevant components and thus improve the performance.

5.3.2 Implementation in AsterixDB



(a) Single secondary index search plan. (b) Multiple-secondary-index search plan.

Figure 5.10: The filter-based secondary-to-primary index search plan.

To implement this search improvement when filters are available, we modified the existing filter rewrite rule in the AsterixDB optimizer. Fig. 5.10 shows the logical query plan after applying the rewrite rule. The optimizer analyzes the query to see if there is any secondary-to-primary index search. If so, it sets the parameter in the secondary search to have it output the pk and also append a pair of filter values (denoted as T_s and T_e) of the component where pk come from. Each resultant record can be represented as a triple $\langle pk, T_s, T_e \rangle$. The entire list of records will be sorted on the pk field and sent to the primary index to use in its search. The primary index search operator can then utilize the filter predicate in the record for every pk search to prune components for different pk's independently. If multiple indexes are found in the path, all indexes will append the filter values of their own. The records will then be sorted by *pk* and sent to the intersect operator. Only the records whose *pk*'s appear in all the index paths will be selected, and only one of the index's records will be output. (Ideally, the output record could have a narrower filter range computed by getting the overlapped range by comparing the multiple filter values. However, the intersect operator supported in Hyracks is a general-purpose operator without an interface to specify such merging logic.) Finally, the intersected records are sent to the primary-index-search operator to execute the filter-based index search.

5.4 Experiments

In this section, we present the result of an experimental evaluation of the proposed filter-based secondary-to-primary index search implemented in AsterixDB. We focus on the queries that can trigger the secondary-to-primary index search but without any filter related predicates. We show the effect of using the improved filter-based index searches with different secondary index access paths and merge policy settings.

5.4.1 Dataset

We used one month of real Twitter data collected via the Twitter streaming API from September 13, 2016 to October 14, 2016. The dataset had 54 million records with a total size of 50GB. The schema of the dataset is shown in Fig. 5.11.

The size of each tweet was about 1KB. Some of the fields of a tweet that can have an impact on the storage layout and as well as query performance are as follows:

- *id*: the primary key of a tweet;

```
create type typeUser if not exists as open {
    id: int64,
    name: string,
    screen_name : string,
    lang : string,
    location: string,
    create_at: date,
    description: string,
    followers_count: int32,
    friends_count: int32,
    statues_count: int64
}
create type typePlace if not exists as open {
    country : string,
    country_code : string,
    full_name : string,
    id : string,
    name : string,
    place_type : string,
    bounding_box : rectangle
}
create type typeGeoTag if not exists as open {
    stateID: int32,
    stateName: string,
    countyID: int32,
    countyName: string,
    cityID: int32,
    cityName: string
}
create type typeTweet if not exists as open {
    create_at : datetime,
    id: int64,
    "text": string,
    in_reply_to_status : int64,
    in_reply_to_user : int64,
    favorite_count : int64,
    coordinate: point?,
    retweet_count : int64,
    lang : string,
    is_retweet: boolean,
    hashtags : {{ string }} ?,
    user_mentions : {{ int64 }} ? ,
    user : typeUser,
    place : typePlace?,
    geo_tag: typeGeoTag
}
```

Figure 5.11: Tweet type definition used in the experiments.

- *create_at*: the publishing time of a tweet. We used this field as the LSM filter field and ingested the data in the same order as their creating time so that the storage components could have disjoint minimum and maximum values, making them very effective for pruning using the filter rules;
- *text*: the content of the tweet. We built a full-text inverted index on this field for the purpose of quickly finding the tweets that mentioned a specific keyword;
- *geo_tag.cityID*: the `geo_tag` field describes the region where the tweet was published. We built a B-tree index on this field to find the tweets published in a certain city.
- *coordinate*: the latitude and longitude values for the place where the tweet was published. It was a null-able field, meaning not every tweet contained this field. In our test dataset, 8 million out of 54 million tweets had a non-null value for this field. We built an R-tree on this field to quickly find the tweets published within a certain geographic area.

5.4.2 Machine and Parameter Configuration

We used a single machine with a dual-core CPU, 16GB memory, and a 500GB M.2 SSD disk to host an AsterixDB instance. We used the parameters shown in Table 5.2 to configure the datasets.

Parameter	Value
Data page size	128KB
Disk buffer cache size	3GB
Bloom filter target false-positive rate	1%
Max component size for merge policy	128MB

Table 5.2: AsterixDB settings for the experiment.

We ingested two copies of the 54 million-tweet dataset separately with the LSM *prefix* and *correlated* merge policies as described in Section 5.2.2.4 to test the performance of the secondary-to-primary search filter rule under different merge policies. Table 5.3 shows the number of on-disk components that resulted from using these different policies after the same data file was ingested. Under the prefix merge policy, each secondary index was merged independently and the adjacent components were merged until the merged size reached the given size limit, i.e., 128MB. Since the secondary index was smaller than the primary index, there were fewer secondary index components than primary index components. For example, the text index had 43 components, which means that one inverted index component roughly mapped to 6 primary index components. The R-tree index had the fewest components because only 8/54 of the tweets had this attribute, hence the size was the smallest among all secondary indexes. The correlated merge policy aligned each secondary component with the corresponding primary component. Therefore, all secondary and primary indexes had the same number of components, i.e., 240.

Index type	Prefix merge policy	Correlated merge policy
Primary	231	240
text (inverted index)	43	240
cityID (B-tree)	11	240
coordinate (R-tree)	5	240

Table 5.3: The number of components of each index after ingestion.

5.4.3 Query Performance

5.4.3.1 Test Queries

We experimented with different types of secondary-to-primary index searches to see the performance of using the proposed optimization. The queries were simple count queries that calculated the number of tweets satisfying specific predicates. The database system utilized different secondary indexes to find the matching records depending on the predicates. Fig. 5.12 shows three examples that led the system to use the inverted index on the `text` field, B-tree index on the `cityID` field, R-tree index on the `coordinate` field, respectively. Each query was run three times and we report the average running time.

```
SELECT count(t) FROM twitter.tweet t
WHERE ftcontains(t.text, ['zika'], {'mode':'all'});
```

(a) Count the number of tweets mentioning `zika`.

```
SELECT count(t) FROM twitter.tweet t
WHERE t.geo_tag.cityID = 100820;
```

(b) Find the number of tweets published in Alabaster, Alabama ($cityID = 100820$).

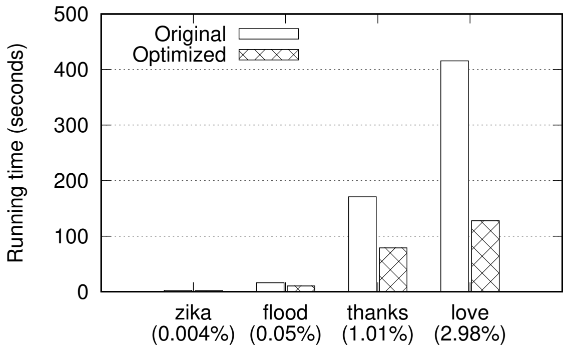
```
SELECT count(t) FROM twitter.tweet t
WHERE spatial_intersect(t.coordinate,
      create_circle(create_point(-118.3238,34.1347), 0.01));
```

(c) Find the number of tweets published around the "Hollywood" sign in Los Angeles, CA by specifying a circle range around its coordinates.

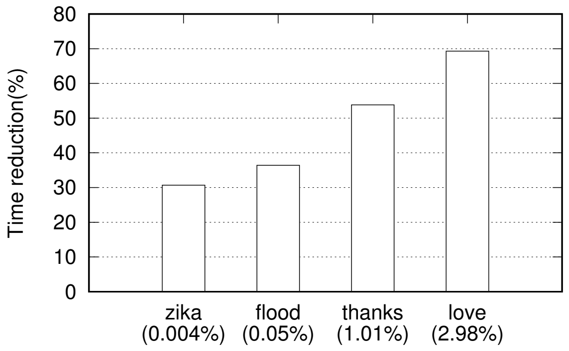
Figure 5.12: Three types of test queries. We changed the keywords, city ids, and radii of the queries to change their selectivity.

5.4.3.2 Using the Prefix Merge Policy

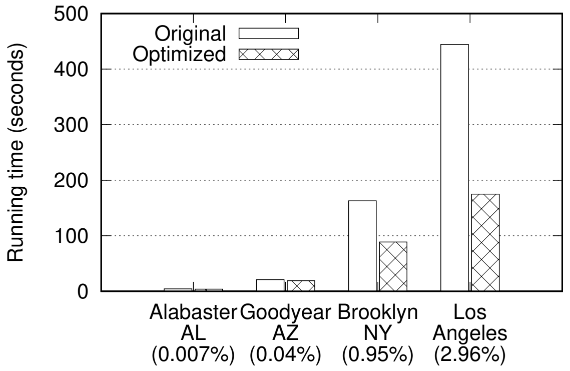
We first evaluated the performance of using the secondary filter optimization on the dataset ingested using the prefix merge policy. Fig. 5.13 shows the query performance for the keyword search, B-tree search, and R-tree search queries against the dataset. Fig. 5.13a shows the query time using different keyword search predicates with different selectivities. Fig. 5.13b shows the speedup from using the secondary filters. The results show that the secondary filters optimization can reduce the query time for all the keyword search predicates of various selectivities. As we increased the selectivity of the search predicate, the speedup became more significant. For the "`love`" query with a 2.98% selectivity, the filter search optimization can reduce the query time by 70%, which corresponds to a three-times speedup as compared to



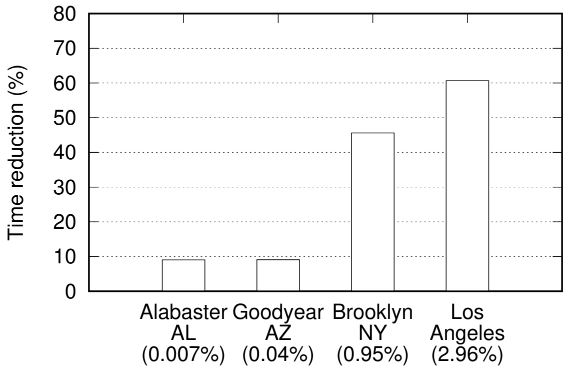
(a) Query time of counting the records with different matching keywords.



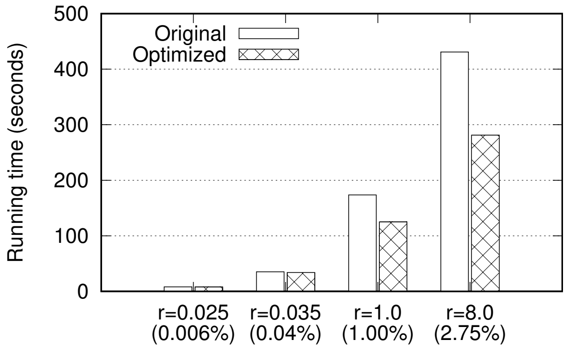
(b) Time reduction of using the filter from the inverted index.



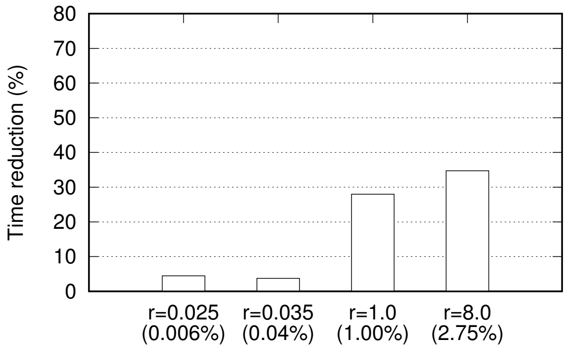
(c) Query time of counting the records with different matching cities .



(d) Time reduction of using the filter from secondary B-tree.

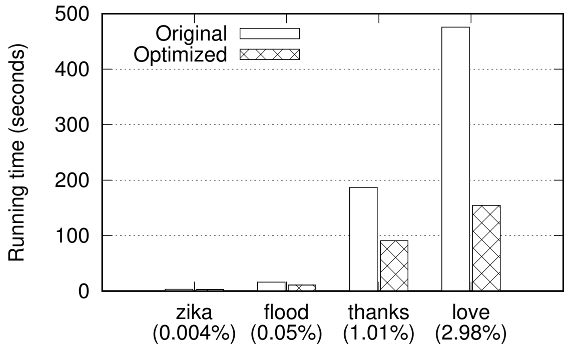


(e) Query time of counting the records within different spatial circle regions with different radii r.

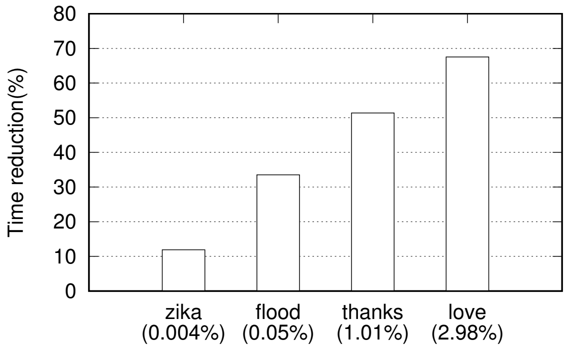


(f) Time reduction using the filter from secondary R-tree.

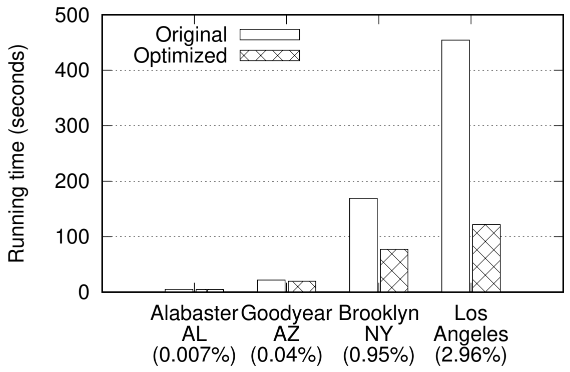
Figure 5.13: Comparison of the query performance between the original filter plan to using the secondary filter values for the primary index search. (The percentages below the predicate values on the x-axes show their selectivities.)



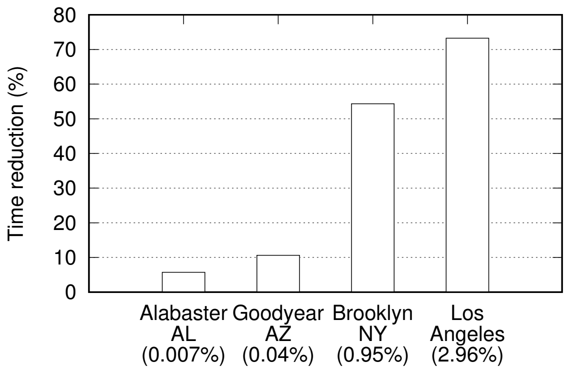
(a) Query time of counting the records with different matching keywords.



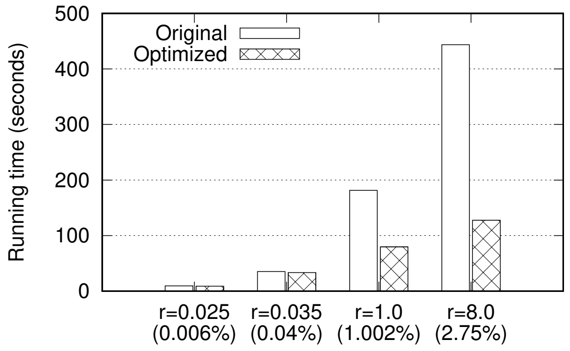
(b) Time reduction using the filter from the inverted index.



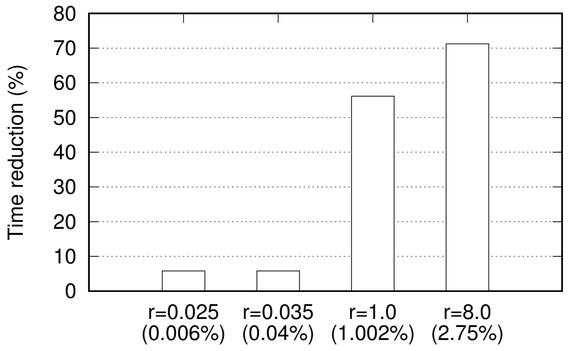
(c) Query time of counting the records with different matching city ids.



(d) Time reduction using the filter from secondary B-tree.



(e) Query time of counting the records within different spatial circle regions with different radii r.



(f) Time reduction using the filter from secondary R-tree.

Figure 5.14: Comparison of the query performance between using and not using the secondary filters for the primary index search under the correlated merge policy.

the original query plan. The reason behind this is as expected, using the filter hints reduces the individual primary index search time, as it then only needs to check a few primary components. As a result, the more primary keys that the secondary index returns, the more time the optimization can save.

Fig. 5.13c and Fig. 5.13d show the query performance when using the secondary filters from the secondary B-tree index built on the `cityID` field. We can see a similar improvement of using the secondary filters from the inverted index. Compared to the speedup values of the inverted index case, the speedup from the `cityID` index was relatively smaller for queries with similar selectivities. Since the number of `cityID` index components was small (11), the filter values attached to the secondary component was 4 times ($\frac{onemonth}{11}$ v.s. $\frac{onemonth}{43}$) wider than the range of the inverted index filter. Thus, there were more matched primary index components to search in this case.

Fig. 5.13e and Fig. 5.13f show the performance when using the filters from the R-tree index built on the `coordinate` field. Due to missing locations, the R-tree index was only built on 15% of the records. Since the total size of index was small, when using the prefix merge policy the index only consisted of 5 components, which means on average each R-tree component aligned with 46 (231/5) primary index components. Thus, the pruning power of the secondary filter from this `coordinate` index is even less than the other indexes.

5.4.3.3 Using the Correlated Merge Policy

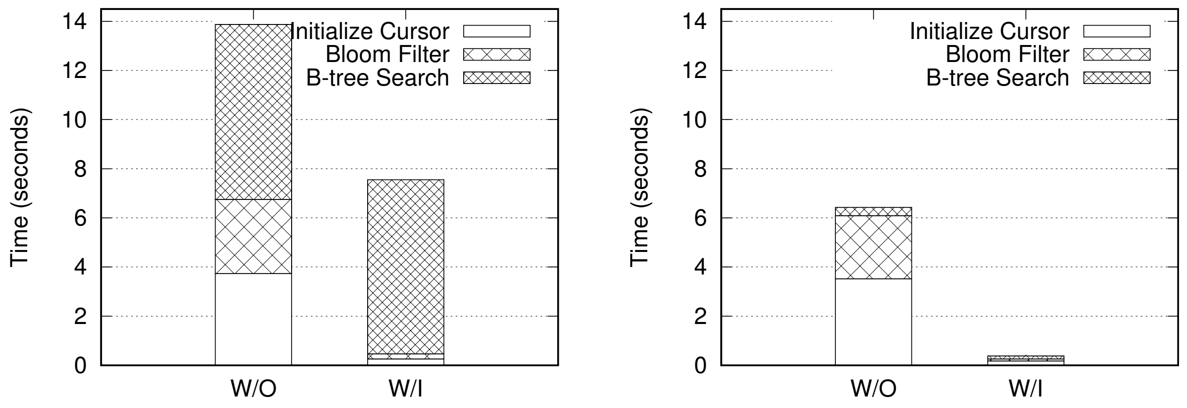
In contrast with the prefix merge policy, where each index is merged independently, the correlated merge policy coordinates all secondary indexes with the associated primary index so that each secondary index component aligns with precisely one primary index component. As Table 5.3 shows, all of the indexes, including the text index, city index, and coordinate index, had 240 components, which was identical to the number of components of the primary index. Fig. 5.14 shows the response time of queries against the dataset ingested using the correlated merge policy. As with the prefix merge policy, we can see that using the secondary filters optimization can significantly improve performance. Moreover, since the secondary index components are perfectly aligned with the primary index, the filter values coming from the secondary index can overlap with at most 3 primary index components (the aligned one and its left and right neighbors). Thus, the speedup here was higher than the experiment on the dataset with the prefix merge policy. Especially for the case of R-tree index in Fig. 5.14f, the time reduction can be up to 70% for the query with a 2.75% selectivity, while the same query can only have a 35% reduction in the prefix merge case.

5.4.4 Analysis of the Improvement

In this section, we drill down further to analyze the internal access details to identify where the speed improvement came from. The cost of secondary-to-primary search is composed of the secondary index search, sort, and primary index search. Since using secondary filter optimization does not change the secondary search or sort operation much, we focus on how the cost of the primary index search has changed.

To search for a specific pk on n primary index components, the search operator checks for pk in the latest component to the earliest one until pk is found. For each component, it first checks the Bloom filter to avoid unnecessary B-tree searches. If the Bloom filter matches pk, the B-tree search will look for pk. Note that there could be a false positive match from the Bloom filter, which results in extra B-tree lookups. By using the secondary filter optimization, we can reduce this cost by searching fewer primary components instead of all the n components. Correspondingly, there are fewer Bloom filter searches and also less false-positive B-tree searches.

We can use one of the keyword search queries matching the keyword "`flood`" as an example



(a) The Bloom filter time and B-tree search time comparison when cache is clean.

(b) The Bloom filter time and B-tree search time comparison when records are cached.

Figure 5.15: Break down the performance gain between not using (W/O) and using (W/I) the secondary filter plan.

to verify the analysis. Internally, we added code to collect the numbers of Bloom filter searches and B-tree searches as well as their corresponding timing. The experiment was conducted on the dataset built using the prefix merge policy.

Count	Original	Secondary filter optimization	Reduction
Bloom filter checks	4,261,002	108,433	97.5%
B-tree searches	66,920	27,623	58.7%
False Positive B-tree searches	40,046	749	98.1%

Table 5.4: The number of Bloom filter checks and primary B-tree searches to find "`flood`" related records.

Table 5.4 shows the number of Bloom filter searches and B-tree searches. As expected, both numbers were greatly reduced by using the secondary index filter. The Bloom filter searches were reduced by about 97.5%. Correspondingly, the false-positive B-tree searches were also reduced by a similar factor (98.1%). Therefore, the total number of B-tree searches was greatly decreased.

Fig. 5.15a shows the time spent on the Bloom filter search and B-tree search when the buffer cache of the database was initially empty. We can see the Bloom filter search time was greatly reduced when the secondary filter optimization was used. Surprisingly, however, the

B-tree search time did not change much, so the performance gain was mainly due to the 4 million fewer Bloom filter searches. The reason for this result is that the B-tree search time was mainly spent on fetching cold pages. Since the primary keys were ordered, false positive searches may not need to load new pages, so the entire B-tree time did not reduce much. To verify this, we conducted another experiment to check the numbers when the B-tree pages were already cached (by running the same query multiple times). The result is shown in Fig. 5.15b. In this in-memory setting, the B-tree search time reduced by more than half (from 0.33 seconds to 0.12 seconds), which was consistent with our previous analysis. However, compared to the 2.5-second difference that was saved by reducing Bloom filter searches, the absolute time benefit from saved B-tree searches was small.

Besides the Bloom filter and B-tree search, for each pk, the system also needs to initialize the search cursor for every filtered component. Though one such initialization operation is as cheap as a Bloom filter search, as the number of pk's and the number of components n increases, the overall time for the cursor initialization adds up. Based on the result shown in Fig. 5.15, by using the information from the secondary filter, the initialization time was also significantly reduced, this is because many components were filtered and thus there was no need to initialize their search cursors.

In summary, by leveraging the secondary filter, we are able to reduce the query time by skipping many unnecessary searches of irrelevant components. The main contributor to this savings comes from reducing many "cheap" in-memory operations.

5.5 Related Work

The work in [62] introduced an index structure called LHAM for transaction-time temporal data. Instead of using an attribute from records, their solution uses the insertion timestamp as the record version number, and the data is partitioned into successive components based on the timestamps of the record versions. Queries with temporal predicates only need to access those components that satisfy the predicates, and the remaining components can be skipped. HBase uses an idea similar to LHAM by using LSM-tree to store incoming records and maintaining a timestamp for each record for versioning purposes. Each LSM component is then tagged with the minimum and maximum timestamps of the records contained in the component. When answering a query with temporal predicates, HBase can leverage the minimum and maximum timestamp filters to only access those relevant components, which reduces query time.

The main difference between our work and the previous work, including the original use of filter in AsterixDB, is that we do not require the query to have filter-related predicates. Our secondary filter optimization can be applied to leverage the presence of filters whenever there is a secondary-to-primary index search, which is a much more general query setting.

5.6 Conclusion

In this chapter, we presented a method of using LSM filters to accelerate the secondary-to-primary LSM index search process in AsterixDB. This technique automatically propagates filter range values from secondary index searches to speed up the primary index search. Importantly, the technique does not rely on the predicate of the query. Thus, it can benefit a broader range of queries.

We have designed and implemented this technique in AsterixDB. Our experiments showed that the new approach can reduce query execution times by 20% to 70% for different queries with various selectivities. The lower the selectivity of the predicate, thus the more keys involved, the more significant the improvement becomes. With this improvement, Cloudberry can choose to generate mini-queries that have much wider time range predicates, which will introduce less overhead for progressive query answering. We also analyzed the reasons for the improvement in detail. We found that a large portion of the observed speedup is from saving many in-memory operations, which reveals the fact that many individual “cheap” in-memory operations are no longer cheap when their number becomes large.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, we have presented a general-purpose middleware system called Cloudberry that sits between a frontend visualization application and a backend database system to support interactive analytics and visualization on large datasets.

In Chapter 3 we presented the design, implementation, and use cases of Cloudberry. We showed that it can reduce the query response time significantly by utilizing materialized views stored in the database. By using an example application, TwitterMap, we demonstrated its suitability to support interactive data analytics and visualization on more than one billion tweets. By using an example Paraview web application and connectors to different databases, we demonstrated that Cloudberry is a general-purpose middleware system.

In Chapter 4 we 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 of the database system to decide on the predicate for the next mini-query appropriately. Drum is a general-purpose technique that requires no changes to the underlying database system. Our experiments on a large, real data set showed that Drum can reduce the delay of delivering intermediate results to the user without increasing the total query time much. Drum enables users to see smooth result updates at the expected rhythm.

In Chapter 5 we presented a way to use LSM filters to accelerate secondary-to-primary LSM index searches in Apache AsterixDB. This technique automatically propagates filter values from the secondary index to speed up the primary index search. It does not rely on having a filter predicate in the user query, so a broader range of queries can benefit from the pruning power of the filter. We have designed and implemented the technique in AsterixDB. Experiments showed that it can reduce the query time by 20% to 70% for different queries with various selectivities; the lower the selectivity of the predicate, the more significant the improvement can have. With this improvement, Cloudberry can generate mini-queries that contain much wider time range predicates, resulting in less overhead for progressive query evaluation. We analyzed the reason for this improvement and found that a large portion of the speedup was from saving many in-memory operations - many individual "cheap" in-memory operations are no longer cheap when their number becomes large.

6.2 Future Work

Our work on Cloudberry has identified a number of interesting opportunities for follow-on work.

Aggregation Views

In Chapter 3 we have shown the general logic of using views to answer queries. We mainly focused on subset views that contain records matching certain filter conditions. One possible future direction is to exploit aggregation view techniques to extend the answering-query-using-view logic to support aggregation queries. For example, it would be natural to ask for the distribution of the total number of tweets for each state or different days. We can store the by-state and by-day counts in a separate materialized view in order to skip the time-consuming scan and aggregation steps on many historical records by fetching the aggregated results from a much smaller view. Moreover, we can exploit the hierarchical information provided by the Cloudberry DDL to store the finer granularity aggregation results to maximize the utilization of views. For example, since we know there are hierarchical relationships between the `stateID`, `countyID`, and `cityID` fields, then instead of storing the by-state aggregation results, we can store the by-city results. In this case, the view can also be useful if a query asks for by-county or by-state aggregation results.

Updates in the Underlying Database

We made two assumptions about the datasets: 1) the dataset should be append-only; 2) there are no too late records whose difference between the event time and ingestion time is more than a delay tolerant time. We want to support the occasional update or delete operations, and late records. One possible way could be to "watch" the database system logs, if it has such feature. Then Cloudberry will not be blind to the underlying updates and can maintain the views accordingly to make sure that the content of a view remains consistent with the base datasets.

Approximate Answering by Sampling in Middleware

In Fig. 3.10 we introduced the "estimate" feature, allowing some simple requests to be answered in the middleware layer without accessing the database. To support this feature, the middleware needs to periodically collect the metrics (e.g., min, max, cardinality, etc) of certain dimensions and to store the values in a separate metadata dataset. This statistical information can also be used to support a sampling approach to approximate answering. For example, instead of query slicing, a user may want to get a rough but fast estimation of the by-state distribution of tweets that mention certain keywords, e.g., "hurricane". One estimation approach can be fetching a certain number of samples, e.g., 1,000 tweets, from the tweets in each state to get the percentages of tweets contain "hurricane" for each of them. We can then give an estimated count of each state by multiplying the total number of tweets in each state with each sampled percentage. To enable such an estimation, we need to know the total number of tweets for all the states in advance. We also need to implement the corresponding logic of calculating the error bounds of the estimation in the middleware system accordingly.

Slicing on More Attributes

In Chapter 4 we studied the query-slicing technique and mainly focused on slicing on a time dimension. One future direction could be to extend the techniques to do slicing on other dimensions. For example, it might be reasonable to slice a query by geographical region so that the frontend user interface can show the results progressively in the spatial dimension. This feature still relies on the underlying database's support to speed up the mini-queries' performance. Compared to the time dimension, the values of other attributes, e.g., spatial coordinates, are less likely to be consistent with the ingestion order of records. Consequently, this could result in highly overlapping filter ranges between LSM components, making them ineffective for pruning. One possible direction could be to periodically re-partition the components based on the declared filter values so that most of the components can have disjoint filter ranges afterward. The mini-query with a geographic boundary predicate can then still skip many irrelevant components so that the overall time of query slicing can be reduced.

More Mergeable Queries

Another challenge of query-slicing is that the mini-query results may not always be mergeable in a straightforward way. For a query with distributive functions (e.g., sum(), count(), min(), max()), the results can be merged by applying the function on the partial result of the same groups. For the algebraic function case (e.g., avg()), the middleware needs to rewrite the query to get all algebraic elements and then apply the algebraic function similarly to the result of the same group. In contrast, a sort query is not easily mergeable in general. However, if the ordering attribute is same as the slicing attribute, the result is still mergeable by a simple union operation if the whole value range of the sorting attribute is known a priori. For example, if a request wants to sort tweets by the number of “likes” they have received and the middleware knows the distribution of the “like”s, it can compose a series of mini-queries by attaching disjoint range predicates to the number of “like”s. Thus, the results can be concatenated directly without extra processing. For an aggregation query with holistic aggregation functions, e.g., median() or topk(), it cannot be simply merged because it requires global information. One possible solution could be to rewrite the mini-query to ask for a richer context of the result. For example, we can request the top 20 results in the mini-query in order to return the top 10 results in the merging case. We could even re-issue the previous mini-query if the previous top 20 results are not adequate to produce the final results. In this way, we can make query slicing more general to support a larger set of progressive display options.

Batch Primary Index Lookup

In Chapter 5 we investigated filter-based acceleration for the secondary-to-primary index search process. We found that a significant speedup is from saving many in-memory operations. This observation suggests improving the secondary-to-primary index search path. For example, there could be a specialized primary index set-based search operator that opens the cursor only once for all primary key probes within the set. By doing so, we could save many repeated open operations for each primary key search.

Bibliography

[1] Apache HBase Website. `http://hbase.apache.org/`.

[2] S. Abiteboul and O. M. Duschka. Complexity of answering queries using materialized views. In A. O. Mendelzon and J. Paredaens, editors, *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*, pages 254–263. ACM Press, 1998.

[3] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 506–521. Morgan Kaufmann, 1996.

[4] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In Z. Hanzálek, H. Härtig, M. Castro, and M. F. Kaashoek, editors, *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, pages 29–42. ACM, 2013.

[5] G. A. Agha. *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence. MIT Press, 1990.

[6] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K. Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 496–505. Morgan Kaufmann, 2000.

[7] S. Agrawal, E. Chu, and V. R. Narasayya. Automatic physical design tuning: workload as a sequence. In S. Chaudhuri, V. Hristidis, and N. Polyzotis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 683–694. ACM, 2006.

[8] M. Ahmad, S. Duan, A. Aboulnaga, and S. Babu. Predicting completion times of batch query workloads using interaction-aware models and simulation. In *ACM EDBT*, pages 449–460, 2011.

[9] Airbnb superset, https://github.com/airbnb/superset.

[10] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB*, 8(12):1792–1803, 2015.

[11] Akka Website, https://akka.io/.

[12] I. Alagiannis, D. Dash, K. Schnaitter, A. Ailamaki, and N. Polyzotis. An automated, yet interactive and portable DB designer. In A. K. Elmagarmid and D. Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 1183–1186. ACM, 2010.

[13] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. R. Borkar, Y. Bu, M. J. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. J. Tsotras, R. Vernica, J. Wen, and T. Westmann. AsterixDB: A scalable, open source BDMS. *PVLDB*, 7(14):1905–1916, 2014.

[14] S. Alsubaiee, A. Behm, V. R. Borkar, Z. Heilbron, Y.-S. Kim, M. J. Carey, M. Dreseler, and C. Li. Storage management in asterixdb. *PVLDB*, 7(10):841–852, 2014.

[15] S. Alsubaiee, M. J. Carey, and C. Li. Lsm-based storage and indexing: An old idea with timely benefits. In *Second International ACM Workshop on Managing and Mining Enriched Geo-Spatial Data, GeoRich at SIGMOD 2015, Melbourne, VIC, Australia, May 31, 2015*, pages 1–6, 2015.

[16] M. Altinel, C. Bornhövd, S. Krishnamurthy, C. Mohan, H. Pirahesh, and B. Reinwald. Cache tables: Paving the way for an adaptive database cache. In J. C. Freytag, P. C. Lockemann, S. Abiteboul, M. J. Carey, P. G. Selinger, and A. Heuer, editors, *VLDB 2003, Proceedings of 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany*, pages 718–729. Morgan Kaufmann, 2003.

[17] Angular JavaScript, https://angularjs.org/.

[18] Army Research Lab, https://www.arl.army.mil/.

[19] E. Baralis, S. Paraboschi, and E. Teniente. Materialized views selection in a multidimensional database. In M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 156–165. Morgan Kaufmann, 1997.

[20] L. Battle, R. Chang, and M. Stonebraker. Dynamic prefetching of data tiles for interactive visualization. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1363–1375, 2016.

[21] N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating user-perceived quality into web server design. *Computer Networks*, 33(1):1–16, 2000.

[22] Apache Cassandra, http://cassandra.apache.org/.

[23] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, 2008.

[24] S. Chaudhuri and V. R. Narasayya. Self-tuning database systems: A decade of progress. In C. Koch, J. Gehrke, M. N. Garofalakis, D. Srivastava, K. Aberer, A. Deshpande, D. Florescu, C. Y. Chan, V. Ganti, C. Kanne, W. Klas, and E. J. Neuhold, editors, *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 3–14. ACM, 2007.

[25] R. Chirkova and J. Yang. Materialized views. *Foundations and Trends in Databases*, 4(4):295–405, 2012.

[26] Cloudberry Project Website, http://cloudberry.ics.uci.edu/.

[27] Cloudberry Wiki Page, https://github.com/ISG-ICS/cloudberry/wiki/Documentation-for-Cloudberry-(Using-SQL-Database).

[28] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. In *Proc. USENIX Symposium on NSDI*, pages 313–328, 2010.

[29] M. Csikszentmihalyi. *Flow and the psychology of discovery and invention.* New York: Harper Collins, 1996.

[30] S. Dar, M. J. Franklin, B. Þ. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 330–341. Morgan Kaufmann, 1996.

[31] J. Du, R. J. Miller, B. Glavic, and W. Tan. Deepsea: Progressive workload-aware partitioning of materialized views in scalable data analytics. In V. Markl, S. Orlando, B. Mitschang, P. Andritsos, K. Sattler, and S. Breß, editors, *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017.*, pages 198–209. OpenProceedings.org, 2017.

[32] S. Egger, T. Hoßfeld, R. Schatz, and M. Fiedler. Waiting times in quality of experience for web based services. In *QoMEX*, pages 86–96, 2012.

[33] A. Eldawy, M. F. Mokbel, and C. Jonathan. Hadoopviz: A mapreduce framework for extensible visualization of big spatial data. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 601–612. IEEE Computer Society, 2016.

[34] 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.

[35] D. Florescu, A. Y. Levy, D. Suciu, and K. Yagoub. Optimization of run-time management of data intensive web-sites. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 627–638, 1999.

[36] A. U. Frank, I. Campari, and U. Formentini, editors. *Theories and Methods of Spatio-Temporal Reasoning in Geographic Space, International Conference GIS - From Space to Territory: Theories and Methods of Spatio-Temporal Reasoning, Pisa, Italy, September 21-23, 1992, Proceedings*, volume 639 of *Lecture Notes in Computer Science*. Springer, 1992.

[37] P. Godfrey, J. Gryz, and P. Lasek. Interactive visualization of large data sets. *IEEE Trans. Knowl. Data Eng.*, 28(8):2142–2157, 2016.

[38] J. Goldstein and P. Larson. Optimizing queries using materialized views: A practical, scalable solution. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 331–342, 2001.

[39] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Min. Knowl. Discov.*, 1(1):29–53, Jan. 1997.

[40] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[41] A. Gupta and I. S. Mumick. *Materialized views: techniques, implementations, and applications*. MIT press, 1999.

[42] P. J. Haas and J. M. Hellerstein. Ripple Joins for Online Aggregation. In *Proc. ACM SIGMOD*, pages 287–298, 1999.

[43] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.

[44] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In H. V. Jagadish and I. S. Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996.*, pages 205–216. ACM Press, 1996.

[45] J. M. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, and P. J. Haas. Interactive data analysis: The Control Project. *IEEE Computer*, 32(8):51–59, 1999.

[46] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *Proc. ACM SIGMOD*, pages 171–182, 1997.

[47] S. Hopfer, M. Runnerstrom, J. Jia, T. Kim, and C. Li. Twitter coverage of climate change and health before and after the 2016 us presidential election. In *Proceedings of the 2017 American Public Health Association Annual Meeting, Nov. 4-8, 2017, Atlanta, Georgia, USA.*, 2017.

[48] S. Idreos, O. Papaemmanouil, and S. Chaudhuri. Overview of data exploration techniques. In T. K. Sellis, S. B. Davidson, and Z. G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 277–281. ACM, 2015.

[49] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. Enterprise data analysis and visualization: An interview study. *IEEE Trans. Vis. Comput. Graph.*, 18(12):2917–2926, 2012.

[50] T. Kim, V. Thirumaraiselvan, J. Jia, and C. Li. Caching geospatial objects in web browsers. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2017, Redondo Beach, California, USA, November 7 - November 10, 2017*, 2017.

[51] Y.-S. Kim. *Transactional and Spatial Query Processing in the Big Data Era.* PhD thesis, UNIVERSITY OF CALIFORNIA, IRVINE, 2016.

[52] Apache Kylin, http://kylin.apache.org/.

[53] I. Lazaridis and S. Mehrotra. Progressive Approximate Aggregate Queries with a Multi-Resolution Tree Structure. In *Proc. ACM SIGMOD*, pages 401–412, 2001.

[54] LevelDB, URL: https://github. com/google/leveldb.

[55] L. D. Lins, J. T. Klosowski, and C. E. Scheidegger. Nanocubes for real-time exploration of spatiotemporal datasets. *IEEE Trans. Vis. Comput. Graph.*, 19(12):2456–2465, 2013.

[56] Z. Liu, B. Jiang, and J. Heer. *imMens*: Real-time visual querying of big data. *Comput. Graph. Forum*, 32(3):421–430, 2013.

[57] A. Magdy and M. F. Mokbel. Demonstration of kite: A scalable system for microblogs data management. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 1383–1384. IEEE Computer Society, 2017.

[58] Mapd demo. https://www.mapd.com/demos/taxis.

[59] T. Marrinan, J. Aurisano, A. Nishimoto, K. Bharadwaj, V. A. Mateevitsi, L. Renambot, L. Long, A. E. Johnson, and J. Leigh. SAGE2: A new approach for data intensive collaboration using scalable resolution shared displays. In E. Bertino, S. Chen, K. Aberer, P. Krishnamurthy, and M. Kantarcioglu, editors, *10th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing, CollaborateCom 2014, Miami, Florida, USA, October 22-25, 2014*, pages 177–186. ICST / IEEE, 2014.

[60] S. Masri, J. Jia, C. Li, G. Zhou, M.-C. Lee, G. Yan, and J. Wu. Use of twitter data to predict zika virus cases in the united states during the 2016 epidemic. 2017.

[61] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *Commun. ACM*, 54(6):114–123, 2011.

[62] P. Muth, P. E. O'Neil, A. Pick, and G. Weikum. Design, implementation, and performance of the LHAM log-structured history data access method. In A. Gupta, O. Shmueli, and J. Widom, editors, *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 452–463. Morgan Kaufmann, 1998.

[63] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, 1996.

[64] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online Aggregation for Large MapReduce Jobs. *PVLDB*, 4(11):1135–1145, 2011.

[65] S. Papadomanolakis and A. Ailamaki. Autopart: Automating schema design for large scientific databases using data partitioning. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM 2004), 21-23 June 2004, Santorini Island, Greece*, pages 383–392. IEEE Computer Society, 2004.

[66] ParaViewWeb Website, https://www.paraview.org/web/.

[67] R. Pottinger and A. Y. Halevy. Minicon: A scalable algorithm for answering queries using views. *VLDB J.*, 10(2-3):182–198, 2001.

[68] S. Rahman, M. Aliakbarpour, H. Kong, E. Blais, K. Karahalios, A. G. Parameswaran, and R. Rubinfeld. I've seen "enough": Incrementally improving visualizations to support rapid decision making. *PVLDB*, 10(11):1262–1273, 2017.

[69] V. Raman and J. M. Hellerstein. Partial results for online query processing. In M. J. Franklin, B. Moon, and A. Ailamaki, editors, *Proc.ACM SIGMOD*, pages 275–286, 2002.

[70] J. Rao, C. Zhang, N. Megiddo, and G. M. Lohman. Automating physical database design in a parallel database. In M. J. Franklin, B. Moon, and A. Ailamaki, editors, *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*, pages 558–569. ACM, 2002.

[71] K. A. Ross and D. Srivastava. Fast computation of sparse datacubes. In M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 116–125. Morgan Kaufmann, 1997.

[72] E. A. Rundensteiner, M. O. Ward, Z. Xie, Q. Cui, C. V. Wad, D. Yang, and S. Huang. Xmdvtool^q: : quality-aware interactive data exploration. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 1109–1112, 2007.

[73] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. On-line index selection for shifting workloads. In *Proceedings of the 23rd International Conference on Data Engineering Workshops, ICDE 2007, 15-20 April 2007, Istanbul, Turkey*, pages 459–468. IEEE Computer Society, 2007.

[74] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings of the 1996 IEEE Symposium on Visual Languages, Boulder*, pages 336–343, 1996.

[75] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. Dwarf: shrinking the petacube. In M. J. Franklin, B. Moon, and A. Ailamaki, editors, *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*, pages 464–475. ACM, 2002.

[76] Y. Skadberg and J. R. Kimmel. Visitors' flow experience while browsing a Web site: its measurement, contributing factors and consequences. *Computers in Human Behavior*, 20(3):403–422, 2004.

[77] J. Song, C. Guo, Z. Wang, Y. Zhang, G. Yu, and J. Pierson. Haolap: A hadoop based OLAP system for big data. *Journal of Systems and Software*, 102:167–181, 2015.

[78] C. Stolte and P. Hanrahan. Polaris: A system for query, analysis and visualization of multi-dimensional relational databases. In J. D. Mackinlay, S. F. Roth, and D. A. Keim, editors, *IEEE Symposium on Information Visualization 2000 (INFOVIS'00), Salt Lake City, Utah, USA, October 9-10, 2000.*, pages 5–14. IEEE Computer Society, 2000.

[79] S. Su, V. Perry, N. Cantner, D. Kobayashi, and J. Leigh. High-resolution interactive and collaborative data visualization framework for large-scale data analysis. In W. W. Smari and J. Natarian, editors, *2016 International Conference on Collaboration Technologies and Systems, CTS 2016, Orlando, FL, USA, October 31 - November 4, 2016*, pages 275–280. IEEE Computer Society, 2016.

[80] Apache Superset, https://superset.incubator.apache.org/.

[81] K.-L. Tan, C. H. Goh, and B. C. Ooi. Query rewriting for SWIFT (first) answers. *IEEE Trans. Knowl. Data Eng.*, 12(5):694–714, 2000.

[82] Y. Tao, G. Kollios, J. Considine, F. Li, and D. Papadias. Spatio-Temporal Aggregation Using Sketches. In *Proc. ICDE*, pages 214–225, 2004.

[83] P. Terlecki, F. Xu, M. Shaw, V. Kim, and R. M. G. Wesley. On improving user response times in tableau. In T. K. Sellis, S. B. Davidson, and Z. G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1695–1706. ACM, 2015.

[84] TwitterMap demo, http://cloudberry.ics.uci.edu/demos/twittermap/.

[85] A. Van Moorsel. Metrics for the internet age: Quality of experience and quality of business. In *Fifth International Workshop on Performability Modeling of Computer and Communication Systems*, volume 34, pages 26–31, 2001.

[86] M. Vartak, S. Madden, A. G. Parameswaran, and N. Polyzotis. SEEDB: automatically generating query visualizations. *PVLDB*, 7(13):1581–1584, 2014.

[87] R. M. G. Wesley, M. Eldridge, and P. Terlecki. An analytic data engine for visualization in tableau. In T. K. Sellis, R. J. Miller, A. Kementsietsidis, and Y. Velegrakis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 1185–1194. ACM, 2011.

[88] A. K. Wong. A literature review of the impact of flow on human-computer interactions (hci)–the study of a fundamental ingredient in the effective use of computers. In *Proc. IAMB*, 2006.

[89] E. Wu, L. Battle, and S. R. Madden. The case for data visualization management systems. *PVLDB*, 7(10):903–906, 2014.

[90] E. Wu, F. Psallidas, Z. Miao, H. Zhang, and L. Rettig. Combining design and performance in a data visualization management system. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, 2017.

[91] W. Wu, Y. Chi, H. Hacigümüs, and J. F. Naughton. Towards Predicting Query Execution Time for Concurrent and Dynamic Database Workloads. *PVLDB*, 6(10):925–936, 2013.

[92] W. Wu, X. Wu, H. Hacigümüs, and J. F. Naughton. Uncertainty Aware Query Execution Time Prediction. *PVLDB*, 7(14):1857–1868, 2014.

[93] P. Xiong, Y. Chi, S. Zhu, J. Tatemura, C. Pu, and H. Hacigümüs. ActiveSLA: a profit-oriented admission control framework for database-as-a-service providers. In *ACM SOSP*, page 15, 2011.

[94] K. Yagoub, D. Florescu, V. Issarny, and P. Valduriez. Caching strategies for data-intensive web sites. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 188–199, 2000.

[95] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli. Druid: a real-time analytical data store. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 157–168, 2014.

[96] J. Yu, R. Moraffah, and M. Sarwat. Hippo in action: Scalable indexing of a billion new york city taxi trips and beyond. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 1413–1414. IEEE Computer Society, 2017.

Appendix A

Solve the Optimal r_i in the Drum Framework

A.1 The Optimal r_i using Histogram Distribution

The expectation score of using histogram is

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

It can be solved as

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

Let $T_{m+1} = L_i - y + b$, then

$$\frac{Pr_m}{b} (1 - \frac{y}{b}) (\frac{t^2}{2} - L_i t) \mid_{L_i}^{T_{m+1}} = \frac{Pr_m}{2b^2} (b - y)^3. \quad (A.3)$$

Let $T_k = L_i - y + (k-m)b$, then

$$\frac{1}{b}\sum_{k=m+1}^{n}\Big(Pr_k(\frac{t^2}{2} - L_i t)\ |_{T_k}^{T_{k+1}}\Big) = \sum_{k=m+1}^{n} Pr_k\Big((\frac{1}{2}+i-k)b - y\Big). \tag{A.4}$$

Then, for a given m, the expectation function is as following:

$$E(r_i) = \frac{r_i}{I} - \frac{\alpha}{C_i L_i}\Big(\frac{Pr_m}{2b^2}(b-y)^3 - \sum_{k=m+1}^{n} Pr_k y + b\sum_{k=m+1}^{n} Pr_k(\frac{1}{2}+k-m)\Big). \tag{A.5}$$

To get the critical value, by solving the $E'(r_i) = 0$, we can get the following form:

$$(b - y_{max})^2 = \frac{2b^2}{3Pr_m}(\frac{C_i L_i}{\alpha a_1 I} - \sum_{k=m+1}^{n} Pr_k). \tag{A.6}$$

Using this equation we can compute the corresponding y_{max} value. We compute the best $f(r_i)$ as:

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

A.2 The Optimal r_i using Gaussian Distribution

The expectation score function is

$$E(score(q_i)) = \frac{r_i}{I} - \alpha\int_{L_i}^{\infty}\frac{t_i - L_i}{C_i L_i}P(t|f(r_i))dt = \frac{r_i}{I} - \frac{\alpha}{C_i L_i}\int_{L_i}^{\infty}(t - L_i)\frac{1}{\sqrt{2\pi}\sigma}e^{-(\frac{t-f(r_i)}{\sqrt{2}\sigma})^2})dt \tag{A.7}$$

Let

$$y = \frac{t - f(r_i)}{\sqrt{2}\sigma}, \tag{A.8}$$

the penalty part of the function can be written as following

$$\begin{aligned} Penalty &= \int_{L_i}^{\infty} (t - L_i) \frac{1}{\sqrt{2\pi}\sigma} e^{-(\frac{t-f(r_i)}{\sqrt{2}\sigma})^2} dt \\ &= \int_{\frac{L_i - f(r_i)}{\sqrt{2}\sigma}}^{\infty} (\sqrt{2}\sigma y + f(r_i) - L_i) \frac{1}{\sqrt{\pi}} e^{-y^2} dy \\ &= \frac{\sqrt{2}\sigma}{\sqrt{\pi}} \int_{\frac{L_i - f(r_i)}{\sqrt{2}\sigma}}^{\infty} y e^{-y^2} dy \\ &+ \frac{f(r_i) - L_i}{\sqrt{\pi}} \int_{\frac{L_i - f(r_i)}{\sqrt{2}\sigma}}^{\infty} e^{-y^2} dy \end{aligned} \tag{A.9}$$

Let

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

then

$$Penalty = \frac{\sigma}{\sqrt{2\pi}} e^{-z^2} + \frac{\sigma z}{\sqrt{2}} (1 + \mathrm{erf}(z)). \tag{A.11}$$

Then the score function is

$$E(score) = \frac{r_x}{I} - \frac{\alpha\sigma}{C_i L_i \sqrt{2}} \Big(\frac{1}{\sqrt{\pi}} e^{-z^2} + z\big(1 + \mathrm{erf}(z)\big) \Big). \tag{A.12}$$

The critical point of r_i is solved by let $E'(r_i) = 0$.

$$E'(r_i) = \frac{1}{I} \frac{a_1}{\sqrt{2}\sigma} \frac{\alpha\sigma}{C_i L_i \sqrt{2}} \frac{d\Big(\frac{1}{\sqrt{\pi}} e^{-z^2} + z\big(1 + \mathrm{erf}(z)\big)\Big)}{dz}. \tag{A.13}$$

Using

$$\frac{d(\frac{1}{\sqrt{\pi}}e^{-z^2})}{dz} = -\frac{2e^{-z^2}z}{\sqrt{\pi}} \tag{A.14}$$

and

$$\frac{d\Big(z\big(1+\mathrm{erf}(z)\big)\Big)}{dz} = \mathrm{erf}(z) + 1 + \frac{2e^{-z^2}z}{\sqrt{\pi}} \tag{A.15}$$

Let equation $A.13 = 0$ will have

$$\frac{2C_iL_i}{a_1I\alpha} = 1 + \mathrm{erf}(z) \tag{A.16}$$

Based on the constraints of erf(z) $\in (-1, 1)$ and $f(r_i) < L_i$, we will have the limit value when the erf(z) $\in (-1, 0)$, when

$$\alpha > \frac{2C_iL_i}{a_1I}, \tag{A.17}$$

the critical point is

$$z_{max} = \mathrm{erf}^{-1}(\frac{2C_iL_i}{a_1I\alpha} - 1) \tag{A.18}$$

Finally we can get the r_i by

$$r_i = \frac{\sqrt{2}\sigma z_{max} + L_i - a_0}{a_1} \tag{A.19}$$