MONITORING NETWORK DATA STREAMS

RUI ZHANG

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2005

Acknowledgement

I would like to thank my supervisor Professor Beng Chin Ooi for his guidance on all my work during my PhD candidature, his guidance on how to be a better researcher, and his suggestions on how to be a better person. I would like to thank Dr. Divesh Srivastava and Dr. Nick Koudas for their guidance and contribution to the work on multiple aggregations over data streams. I would like to thank Associate Professor Kian-Lee Tan for his suggestions and comments on the work on nearest neighbor search over data streams.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

Summary

The data input of a new class of applications such as network monitoring, web contents analysis and sensor networks takes the form of a stream, called *data stream*. This type of data is characterized by an extremely high data arrival rate and a very large data volume. Network monitoring may be the most compelling application that deals with data streams. The backbone of a large Internet service provider (ISP) can generate 500 gigabytes data per day, even with a high degree of sampling. Data stream algorithms are essential to the efficient processing of such data. Major network tasks such as traffic management and security exploit two basic operations: aggregation and nearest neighbor search. This thesis addresses the problem of efficient processing of these two types of queries. We base our study on Gigascope, a real data stream management system (DSMS) deployed in AT&T, which is specially designed for processing network data streams.

Aggregation is a primitive operation needed for network performance analysis and statistics collection. The need for exploratory IP traffic data analysis naturally leads to related aggregation queries on data streams that differ only in the choice of grouping attributes. One problem we address in this thesis is to efficiently com-

pute multiple aggregations over high speed data streams, based on the two-level (LFTA/HFTA[1]) query processing architecture of Gigascope. On this problem, our first contribution is the insight that in such a scenario, additionally computing and maintaining fine-granularity aggregation queries (phantoms) at the LFTA has the benefit of supporting shared computation. Our second contribution is an investigation into the problem of identifying beneficial LFTA configurations of phantoms and user queries. We formulate this problem as a cost optimization problem, which consists of two sub-optimization problems: how to choose phantoms and how to allocate space for them in the LFTA. We formally show the hardness of determining the optimal configuration, and propose cost greedy heuristics for these independent sub-problems based on detailed analyses. Our third contribution is a thorough experimental study using synthetic data and real IP traffic data to demonstrate the effectiveness of our techniques for identifying beneficial configurations.

Another problem we address in this thesis is similarity search over data streams, which we model as the nearest neighbor queries. This type of queries can be useful for security and stream mining, where TCP/IP packets that are similar to a certain pattern need to be found. We use approximation techniques to achieve low memory usage and high performance for this problem. Our first contribution on this problem is the introduction of a new type of approximate nearest neighbor queries, called the *e-approximate kNN (ekNN) query*, which considers the class of applications where errors are expressed as an absolute value. Our second contribution is the proposal of a framework that could reduce the information that has to be maintained to guarantee the error bound. Our third contribution is the proposal for a technique called aDaptive Indexing on Streams by space-filling Curves (DISC) to realize the proposed framework. DISC can adapt to different data distributions to either:

[1]LFTA/HFTA stands for Low/High-level Filter, Transform and Aggregate. These are the two query processing components in the Gigascope system.

(a) optimize memory utilization to answer ekNN queries under certain accuracy requirements, or (b) achieve the best accuracy under a given memory constraint. At the same time, DISC provides efficient updates and query processing, which are important requirements in data stream applications. Our fourth contribution is an extensive experimental study, still using both synthetic and real data sets to demonstrate the effectiveness and efficiency of DISC.

CHAPTER 1

Introduction

The phenomenon of data streams has emerged in recent years, and the wide use of the Internet is the driving force. Internet service providers possess large networks, which generate data, mainly in the form of TCP/IP packets, at an extremely high speed. Much of web site activities, such as online ordering systems, bulletin board systems and stock price reporting systems, exhibit stream characteristics. Another new application, the sensor network, also generates data in a streamed fashion. In this chapter, we describe the phenomenon of data streams in detail and identify two important query types for monitoring network data streams: the aggregate query and the nearest neighbor query. These two query types are the focus of the study presented in this thesis.

The rest of the chapter is organized as follows. We first show some real life data stream examples such as network monitoring, network security, financial tickers, sensor network and web contents monitoring in Section 1.1. Then in Section 1.2, we take a closer look at network data streams, which are of central interest in this

thesis. We articulate the problems we are trying to solve, give an overview of the work we have done, and summarize the contributions of our work in Section 1.3. Finally, we present an outline of the thesis in Section 1.4.

1.1 Phenomenon of data streams

Over the past few years, we have witnessed the emergence of a new class of applications where the data input is of a very large volume (possibly infinite) and arrives at the system at a very high speed. Due to the high data volume, we cannot afford to store the data on hard disk and issue queries on it offline as in the traditional database. Typically, we can read the data records only once as they stream by and then discard the data. A small portion may be retained for some time in order to answer certain queries, but generally, the data would not be stored onto the hard disk. Moreover, the queries in these applications are usually continuously evaluated, and the query answers change as new data arrive. In other words, query processing is driven by data input instead of, traditionally, the human who issues the query. The input in these applications, characterized by a large volume and a high speed, is called ***data stream*** (subsequently, we may simply say "stream"). Representative data stream applications are as follow:

- **Network traffic management**

  Large Internet service providers (ISPs) need to monitor and analyze the network traffic flowing through their system to obtain link utilization, compute traffic matrices, detect denial-of-service attacks, etc. For example, even with a high degree of sampling and aggregation in Netflow records (traffic summaries produced by routers) the AT&T IP backbone alone generates 500 gigabytes of data per day (about 10 billion fifty-byte records) [46]. Monitoring and

analyzing such a large network system are typical data stream problems.

- **Network security**

  Network security systems apply sophisticated rules over the network or compare the traffic against *signatures* that describe network intrusion patterns to support firewall or detect intrusions [125, 124]. For example, *iPolicy Networks* [95] provide these services through an integrated security platform that performs complex stream processing including URL-filtering based on table lookups, and correlation across multiple network traffic flows.

- **Sensor networks**

  Recent advances in integrated circuit technology have enabled the mass production of very capable sensor motes (e.g., [89]), which are actually full-fledged computer systems with a CPU, main memory, operating system and a suite of sensors, and the communication between sensors is wireless. These sensors are to be used for a wide range of monitoring and data collection tasks in industries such as transportation, manufacturing, health care, environmental oversight, safety and security. For example, in the US, a sensor infrastructure is deployed on San Francisco Bay Area freeways to monitor traffic conditions [113]. Thousands of primitive sensors have been embedded on the freeways. These sensors consist of inductive loops that register whenever a vehicle passes over them, and can be used to determine aggregate flow and volume information on a stretch of road as well as provide gross estimates of vehicle speed and length. The readings of the large number of sensors arrive at a central system continuously, therefore stream algorithms are crucial for processing the data.

- **Financial tickers**

Real-time stock price analysis tools need to discover correlations, trends and arbitrage opportunities, and forecast future values in an online fashion as the stock market changes. For example, the Traderbot web site [88] is a web-based financial ticker that allows users to pose complex continuous queries over the streaming financial data as follows: find all stocks priced between $10 and $100, where the spread between the high tick and the low tick over the past 20 minutes is greater than five percent of the last price.

- **Web log/content monitoring and analysis**

  Web sites monitor logs to discover interesting customer behavior patterns and identify suspicious spending behavior for applications such as personalization and crime detection and for performance considerations. Some researchers also envision a "World-Wide Database" in which continuous queries can be posed over the large amount of XML data on the Internet [39].

1.2 Network data streams

In this section, we focus on the two fundamental network management tasks: traffic management and network security, and present more details on real-life applications. Then we show what kinds of data management problems are posed by these applications and why data stream approaches are needed to solve them.

1.2.1 Traffic management

Managing a large communication network is a complex task handled by a group of human operators called *network analysts*. The analysts perform tasks such as performance analysis and conformance testing to detect equipment failure and shifts in traffic load. If a failure or unbalanced load is detected, the operator may change

the configuration of the equipment to improve the utilization of network resources and the performance experienced by users. At the same time, statistics such as link utilizations are collected for use in functions such as billing clients. Performance analysis and statistics collection are done through ***aggregate queries***. For example, the operator may query the number of packets sent from every source to every destination during a particular time period, which translates into a sum-group-by query, to see if there is unbalanced load between links. To process such a query, network operators usually use a combination of hardware and software tools. High speed (gigabit and higher) network hardware or software tools such as NetFlow built into the routers are available. These tools operate directly on the live network, but they all have the problem of inflexibility. For complex operations, network analysts have to use TCPdump to save network traces and then write ad-hoc programs to analyze the data. These ad-hoc programs are highly tuned to perform well on the dumped data, which could not be achieved if a conventional database management system (DBMS) were used. When diagnosing potential performance problems, analysts benefit from having a timely view of the traffic across the network. However, this requirement cannot be satisfied by dumping network traces and examining them offline. A stream approach for the aggregation operation is compelling. Efficiently aggregating network data is one of the topics in this thesis, and we study it in Chapter 3.

1.2.2 Security

Network intrusions are common these days and have been a major concern of ISPs. A typical intrusion scenario is as follows. An intruder first finds out as much as possible publicly available information about a target machine such as its domain name through a "whois" lookup. The intruder may also use more invasive

techniques to scan for information. For example, he/she may walk through the web pages and look for CGI scripts (CGI scripts are often easily hacked) and do "ping" sweeps to see which machines are available. Until now, the intruder has not done anything harmful. Next, the intruder crosses the line and starts exploiting holes such as software buffer overflows and TCP/IP protocol flaws in the target machines to get access to the machines. Once the intruder gains access to the machines, he/she can do whatever to them at will. More information about network intrusion may be found in [141]. There are mainly three intrusion detection techniques: *signature*, *anomaly* and *misbehavior* [119].

- **Intrusion detection by signature**

  The *signature* approach defines a set of policies (or rules), and then filters the network packets according to the policies. Usually, the policies use *signatures* which define or describe a traffic pattern of interest. These signatures are extracted from known intrusions and need to be updated if new ones appear. For example, the Land attack (a type of denial-of-service attack) sends packets whose source IP address and source port are the same as their destination IP address and port, which causes some TCP/IP implementations to crash. Since no legitimate application would send these kinds of packets, we can use a filter to check the equality of source and destination IP addresses/ports. So the signature for the Land attack is that the source IP address and port are the same as the destination IP address and port, respectively [125]. This signature is relatively simple since we only need to do an equality check, but many signatures are complex and may be fuzzy. For example, intrusions that exploit buffer overflow usually use a command sequence that contains a large number of hex 90's followed by some machine code, some ASCII strings, and a literal command "/bin/sh -C". This buffer overflow is designed to break out

to a shell and execute code that will break out. One of the intrusion's characteristics is to create a directory called ADMROCKS. Therefore, we may have a signature looking for some 90's, "/bin/sh -C" and "ADMROCKS" in different parts of the traffic flow. But if we make the signature too strict, an intruder can modify the exploit code slightly (e.g. change some 90's to other numbers or change ADMROCKS to ADMROXXS) to slip under the highly tuned radar [124]. Therefore, we should allow approximate pattern matching between the signatures and actual traffic, which translates to approximate similarity search over the network packets. Similarity search on multi-attribute data is usually modelled as the ***nearest neighbor query*** in multi-dimensional space.

- **Intrusion detection by anomaly**

  Anomaly detection takes the opposite approach from signature detection. It admits the fact that malicious behavior evolves, and that a defense system cannot predict and model all of them. Instead, anomaly detection tries to model legitimate traffic and raises an alert if the observed traffic violates the model. Legitimate traffic are defined from past traffics that have been shown to be of no harm to the system. The advantage of this approach is that attacks unknown previously can be discovered if they differ sufficiently from the legitimate traffic. However, there is a big challenge in anomaly detection. Legitimate traffic is diverse with new applications arising from time to time. It is difficult to model legitimate traffic as its patterns change over time. A model that is too rigid would generate many false positives from legitimate traffic, but a model that is too flexible may overlook real intrusions (false negatives). Identifying the right set of features and model to tackle the balance between false positives and false negatives is a real challenge.

Operations needed by anomaly detection is again pattern matching over the packets, which translates into the ***nearest neighbor query***.

- **Intrusion detection by misbehavior**

  In contrast to anomaly detection, which models legitimate behavior, misbehavior detection tries to model misbehavior in the traffic. At the extreme, misbehavior detection is similar to signature-based detection, that is, receiving packets that match certain pattern of a particular attack toolkit. However, misbehavior can be more generally defined than signatures. For example, when the machine is receiving high traffic and is not able to keep up, there is probably a denial-of-service (DoS) attack and an alarm is triggered. This phenomenon can be defined as a misbehavior but not a signature. Misbehavior detection faces a challenge similar to that of the anomaly detection approach. It needs to model misbehavior properly so that the false positive and false negative rates are kept low. Operations needed by misbehavior detection are ***aggregate queries*** (to obtain the number of packets arriving at a machine) and ***nearest neighbor queries*** (to detect certain traffic patterns) over the packets.

Irrespective of which of the above approaches is used, a common requirement is that intrusions be detected promptly. To this end, online monitoring of network traffic is necessary, and hence, stream algorithms are required. The operations needed frequently are aggregate queries, and similarity search which translates into nearest neighbor queries. Aggregate queries (e.g. "how many packets are sent from every sender to the backbone server?", more specific examples in Section 3.1) are needed in network performance analysis, statistics collection and intrusion detection by misbehavior. Nearest neighbor queries are needed in most intrusion detection techniques, and may also be useful for virus detection.

We note that *malware* (virus, worm, trojan, etc.), is generally viewed as *virus*, and poses a significant security problem for computers. However, most anti-virus software monitors files which are contents reassembled from packets instead of the raw packets themselves. The files delivered to the computers of end users usually arrive at a much lower speed, and therefore, a stream algorithm may not be necessary. A few anti-virus software companies, such as Trend Macro and McAfee, also provide *gateway virus scanners* which perform virus protection at the gateway of a network. A gateway virus scanner must check for virus at a very high speed, and it usually exploits hardware of high performance. Stream algorithms for pattern matching may also be useful in this case.

1.3 Contributions of this thesis

As discussed in the previous section, many network monitoring tasks comprise aggregate queries and nearest neighbor queries as their basic operations. We therefore focus on these two types of queries in this thesis. We have based our research on the two-level (LFTA/HFTA[1]) query processing architecture of the data stream system, Gigascope, developed at AT&T.

1.3.1 Contributions on aggregate queries over data streams

For aggregate queries, we study how to achieve optimal overall processing cost when a set of aggregate queries are given. We propose maintaining additional information, called *phantoms*, which are fine-granularity aggregation queries not defined by users but maintained for sharing of computation among the queries. There are many choices of phantoms. The problem is which phantoms to maintain,

[1]LFTA/HFTA stands for Low/High-level Filter, Transform and Aggregate. These are the two query processing components in the Gigascope system.

and for the chosen phantoms, how to allocate limited resources (memory in our case) to them. We model this multiple aggregation problem as an optimization problem consisting of two subproblems: phantom choosing and space allocation. Specifically, we make the following contributions on the multiple aggregate query processing problem:

- We generate the insight on the benefit of computing and maintaining *phantoms* at the LFTA when computing multiple aggregate queries that differ only in their grouping attributes. Phantoms are fine-granularity aggregate queries that, while not of interest to the user, allow for shared computation between multiple aggregate queries over a high speed data stream.

- We investigate the problem of identifying beneficial configurations of phantoms and user-queries in the LFTA of Gigascope. We formulate this problem as a cost optimization problem which consists of two sub-optimization problems: how to choose phantoms and how to allocate space for hash tables in the LFTA amongst a set of phantoms and user queries. Specifically, among many choices of phantoms, the phantom choosing sub-problem needs to find out the set of phantoms to maintain so as to minimize the cost. However, just finding out the right set of phantoms is not enough to achieve the minimum cost. We still need to allocate space correctly to the hash tables of the phantoms to reach the goal. This is the second sub-problem, space allocation.

  We formally show the hardness of determining the optimal configuration (the set of phantoms to be maintained), and propose a greedy algorithm to identify phantoms which can help reduce the cost. We have a detailed analysis on the space allocation problem and the analysis results in optimal space allocation for some configurations. For those untractable configurations, we propose some heuristics based on our analysis.

- We carry out a thorough experimental study using synthetic data and real IP traffic data to understand the effectiveness of our techniques for identifying beneficial configurations. We demonstrate that the heuristics result in near optimal configurations (within 15-20% most of the time) for processing multiple aggregations over high speed streams. Further, choosing a configuration is extremely fast, taking only a few milliseconds. This permits adaptive modification of the configuration to changes in data stream distributions.

1.3.2 Contributions on nearest neighbor queries over data streams

For nearest neighbor queries, we study what information should be maintained in order to answer queries approximately with an error bound guarantee. The information maintained should be only what is necessary to satisfy the error bound requirement so that either memory usage can be minimized, or conversely, when a memory constraint is given, errors are minimized. We make the following contributions on this problem:

- We introduce of a new type of approximate nearest neighbor queries, called the *e-approximate k nearest neighbor (ekNN) query*, which specifies the error bound as an absolute value instead of a relative one.

- We propose a framework that makes it possible to reduce the information needed to answer ekNN queries with a guaranteed error bound. Specifically, we divide the data space in to cells and only need to maintain at most G records in each cell in order to guarantee some error bound, where G is a user defined parameter.

- We propose a technique called aDaptive Indexing on Streams by space-filling

Curves (DISC), under our proposed framework, to efficiently maintain data and process queries from the maintained data. DISC has efficient insertion, deletion and kNN search operations. We also propose an efficient merge-cell algorithm for DISC, which is essential to adjust DISC to the data distribution of the data stream. By DISC, we attain two optimization goals: *memory optimization for a given error bound*, and *error minimization for a given memory size*.

- We carry out a thorough experimental study using synthetic data and real IP traffic data to study the memory and error behavior of DISC. The results show that DISC achieves the optimization goals, outperforming competitors with very efficient query processing which meets the real-time response requirement of data stream applications.

1.4 Outline of the thesis

The rest of the thesis is organized as follows:

- Chapter 2 gives a more precise description of the data stream model, reviews commonly used techniques in data stream algorithms, surveys state-of-the-art data stream management systems built in different institutes and organizations, with an emphasis on the Gigascope data stream management system (DSMS) since we will use this system's architecture as the infrastructure of our network stream monitoring algorithms. Finally we discuss related work to the problems we study in this thesis.

- Chapter 3 presents our proposed technique for optimizing the processing of multiple aggregate queries based on a two-level query processing architecture of the Gigascope DSMS. This optimization problem consists of two sub-

problems: phantom choosing and space allocation, which are studied in depth in the chapter.

- Chapter 4 presents a technique called aDaptive Indexing on Streams by space-filling Curves (DISC) to process approximate nearest neighbor queries over data streams. We focus on the set of approximate nearest neighbor queries guaranteed by an absolute error bound, which is a new query type we introduce, called the *e-approximate kNN (ekNN) query*. While the DISC technique is originally proposed for general data stream applications, we show that it fits into the two-level query processing architecture of Gigascope fairly well.

- Chapter 5 concludes our work and discusses directions for future work.

Two papers have been published from the work reported in this thesis. The work on multiple aggregations over data streams, presented in Chapter 3, has been published in [155]. The work on approximate nearest neighbor processing over data streams, presented in Chapter 4, has been published in [103].

CHAPTER 2

The Data Streams

Data streams have the nature of extremely high speed and large volume. The traditional database model for relatively static data is no longer capable of processing the streams. In this chapter, we present the data stream model and the way stream queries are specified. We give an overview of stream algorithms and systems with an emphasis on Gigascope, the system our study is based on. Finally we discuss related work.

The rest of the chapter is organized as follows. We first discuss the data stream model and queries over data streams in Section 2.1. Then we summarize commonly used techniques in data stream algorithms such as approximation, window queries and sharing in Section 2.2. Next, we review a number of existing data stream management systems (DSMSs) in Section 2.3. We describe the Gigascope DSMS in detail in Section 2.4, since we would use this system's architecture as the infrastructure of our network stream monitoring algorithms. Finally, in Section 2.5, we investigate existing work related to the two problems we study. Some topics such

as query language for DSMSs are not covered in this section because they are not our focus. Interested readers would be directed to the survey paper [14], which provides a more comprehensive view of models and issues in data streams.

2.1 The data stream model and queries

2.1.1 The data stream model

In the data stream model, the input is a sequence of data records. Each record is of the same record type. The records can be of fixed length or of variable lengths. The particular attributes depend on the application. For example, in network data streams, the typical attributes are *source IP*, *source port*, *destination IP*, *destination port*, etc. The length of the sequence may be infinite. The input arrives at the system or the query processing unit continuously. The arrival rate, time and the order of the records depend on the nature of the input; they could not be controlled by the system. Each record is read only once, processed immediately, and then discarded; it cannot be accessed again unless it is explicitly stored in main memory, which is very small relative to the size of the input stream. In very rare cases, the data having streamed by could be archived, but the archived data is hard to be retrieved due to the very large size.

Given the above discussion of the data stream model, a stream algorithm typically should satisfy the following requirements:

- The algorithm reads each data record only once as the record streams by.
- The algorithm can only use a limited amount of memory.
- The algorithm should be very efficient to answer the queries, that is, having almost real-time response.

The third requirement is due to the reason that many data stream applications need real-time response such as network traffic monitoring, sensor network monitoring, etc.

The data stream model was first formalized in [85]. Their model allows multiple passes over the data streams. However, more realistic data stream applications fit into the model that allows only one pass over the streams, and most of the existing work on data streams have assumed this model. In this thesis, we also focus on this model, which allows only one pass over the data streams.

2.1.2 Queries over data streams

Many traditional query types find their applications in data streams, but their semantics differ slightly from the traditional ones in the data stream setting. One class of the queries include those common operators found in a DBMS such as selection, aggregations (SUM, COUNT, MIN, MAX and AVG), join, etc. Another important class of queries maintain "miniature" representations of the original stream data, such as sketches, sampling, histograms and wavelets to facilitate other queries or query optimization. Finally, we also have some ad hoc query types over streams such as nearest neighbor queries. Note that all these queries must change their requirements a little to comply with the data stream model. That is, these queries require that each input record be read only once and the record typically gets discarded except maybe a few ones maintained in a memory with size constraint. The queries may also be modified in another way. Many applications such as monitoring tasks require the system to provide answers continuously, therefore we can also have a *continuous* version of the above queries, for example, "report the IP address that sends the maximum number of packets every second". Here, "continuously" means every time unit, or at a user-defined frequency.

While most of the above mentioned query types have counterparts in previous database research, the term "sketch" may seem new as it just started to appear in the past couple of years in the data stream literature. we would explain it a little bit here. Sketches are a small amount of data maintained based on the data stream in order to compute some characteristics (such as frequency moments as explained below) of the data stream approximately. For example, Morris [120] showed that a register of $O(\log \log m)$ bits can be used to count up to m elements approximately (usually we need $\log m$ bits to count to m accurately). Then the data maintained in the register (which has a small size, $O(\log \log m)$) is called a sketch of the data stream. Frequency moments [5] provide useful statistics on the data sequence. They are defined as follows. Let $\mathcal{S} = (r_1, r_2, ..., r_n)$ be a sequence of elements, where each r_i is a member of the set $\mathcal{N} = \{1, 2, ..., n\}$. Let $m_i = |\{j : r_j = i\}|$ denote the number of occurrences of i in the sequence $\mathcal{S}$. Then for each non-negative integer k,

$$F_k = \sum_{i=1}^{n} m_i^k$$

is the k-th frequency moment of $\mathcal{S}$. Frequency moments represent important demographic information about the data sequence. For example, F_0 is the number of distinct elements in the sequence, $F_1 (= m)$ is the length of the sequence, and F_2 is the self-join size (also called *Gini's index of homogeneity*), that is, when a relation is joined with itself, F_2 is the output size of the join. F_∞ is defined as $max_{1 \le i \le n} m_i$ [5], which is the most frequent element's number of occurrence. Consider the sequence, {A,A,A,B,B}, which has 5 records (please note that a data stream can be of limited length). For this sequence, the frequency of A is 3 and the frequency of B is 2. Therefore, $F_0 = 3^0 + 2^0 = 2$, $F_1 = 3^1 + 2^1 = 5$, $F_2 = 3^2 + 2^2 = 13$ and $F_\infty = max\{3, 2\} = 3$. N. Alon et al. [5] proved memory upper bounds needed to approximate the F_k's through randomized algorithms. They also proposed ran-

domized algorithms to improve previous ones to calculate some F_k's. The word "sketch" was first used in [72] to mean the structure and data needed to calculate the frequency moments through randomized algorithms, and was then widely used in other papers [54, 14, 43]. It was called "sketch" probably because the structure and the data maintained to calculate the frequency moments give a sketch (approximate representation) of the data stream.

2.2 Stream algorithms

As discussed in Section 2.1.1, stream algorithm can read the record once and store only a small portion of it in memory. Since we cannot access all the data, whenever we need to process a query that refers to data not explicitly stored, the answer must be inaccurate. Therefore, a prominent characteristic of stream algorithm is *approximation*. Many queries evolve to an approximate version. The research community has been very active in developing algorithms to provide approximate answers to queries over streams. We would discuss techniques to obtain approximation of either the original data stream or the query answers in Section 2.2.1. In many applications, the user is only or more interested in recent data, say, the total network traffic in the last 5 minutes instead of that in the last two years ever since the server started working. In these cases, a *window query* that returns merely the answers for the query in a recent time window is appropriate. More generally, a window query can also be viewed as an algorithmic strategy for approximation, that is, approximating the whole history by the recent status. However, different from the other techniques that approximate by reducing the data, the window query approach approximates by reducing the time range. Therefore we discuss the window queries separately in Section 2.2.2.

2.2.1 Approximation techniques

In this section, we summarize the state of the art of several approximation techniques commonly used in stream algorithms. They are sketches, sampling, histograms and wavelets.

In our work on nearest neighbor search over data streams presented in Chapter 4, we also use an approximation technique. We partition the space into cells and use some representative points in the space to approximate all the points and provide guarantee that the query answers are within certain error bound. This is a spatial approximation technique, which is different from the approximation techniques commonly used in the literature of data stream research.

Sketches

We have presented the definition of sketch in Section 2.1.2. Sketches are used to efficiently calculate the frequency moments, F_k's, by using a small amount of space (usually less than $O(\log m)$, where m is the length of the stream). Probabilistic counting may be the earliest form of sketch technique (recall that F_1 is the length of the sequence, namely the count of the elements). R. Morris [120] showed how to count approximately (that is, to approximate F_1) using $O(\log \log m)$ bits of memory (see [66] for a detailed analysis). The basic idea is to use a randomized algorithm to determine whether to increase the counter when there is an occurrence of an event. Then one can estimate the actual counts from the number in the counter using statistics. An algorithm approximating F_0 using $O(\log n)$ (n is the cardinality of the domain of the elements) bits of memory was proposed by P. Flajolet and G. Martin [67]. This algorithm hashes the data values to a bit string. Then the number of distinct values can be estimated statistically from the 0's and 1's in the bit string. Using a similar approach, that is, statistical estimation from a hashed

bit string, K.-Y. Whang et al. [147] proposed an algorithm to approximate F_0 in $O(\log n)$ time while allowing duplicates in the data set. A key contribution of [5] is an algorithm to approximate F_2 using $O(\log n + \log m)$ and providing arbitrarily small approximation factors. The basic idea of the F_2-sketch technique is as follows. Every element i in the domain $\mathcal{N}$ hashes randomly to a value $v_i \in \{-1, +1\}$. Then the random variable $X = \sum_i m_i v_i$ is defined and X^2 is returned as the estimator of F_2. The estimator can be computed in a single pass over the stream as long as the v_i values can be efficiently computed. It can be proven that X^2 has expectation equal to F_2 and variance less than $2F_2^2$ if the hash functions have four-wise independence[1]. We can combine several independent estimators to achieve an accurate estimation of F_2 with high probability. This sketch technique for F_2 has many applications in database, including join size estimation [4], estimation L1-distance of vectors [62], and processing complex aggregate queries over multiple streams [54, 70].

Sampling

Sampling has been widely used in statistics and databases. When a small sample is expected to capture the essential characteristics of the whole data set, we can use sampling as a summary structure. Here we focus on sampling over data streams. If we simply want to compute a random sample of the stream, we can use the *reservoir sampling* [145] which makes one pass over a sequence of data with unknown length. The first step of any reservoir sampling is to put the first n records of the file into a "reservoir". The rest of the records are processed sequentially; records can be selected for the reservoir only as they are processed. The algorithm maintains the invariant that after each record is processed a true random sample of size n can be

[1] A family H of hash functions is called "k-wise independent" if a random hash function from H maps each set of k elements in the universe U to uniformly random and independent values. There are standard techniques (e.g. see [138]) to construct a family of k-wise independent hash functions.

extracted from the current state of the reservoir. Sometimes it may be more efficient to use specially designed sampling methods for particular problems. Chaudhuri et al. proposed a sampling technique for join queries [37]. They devised a variety of sampling schemes based on the observation that that given some partial statistics (e.g., histograms) on the first operand relation, they can use the statistics to bias the sampling from the second relation in such a way that it becomes possible to produce the sample of the join. Manku and Motwani proposed a sampling based algorithm to approximate frequency counts [115]. They use a changing sampling rate as the data elements arrive so that they can use a user-defined space to obtain the frequency counts within a good error bound. Gibbons proposed a sampling based algorithm to estimate the number of distinct values [128]. In particular, the algorithm collects a specially tailored sample over the data which estimates the number of distinct values with high accuracy. Duffield et al. [56] proposed a sampling algorithm to estimate the size of a subset of objects in a data stream . The algorithm continuously stratifies the sampling scheme so the probability that a record in a subset is selected depends on the size of the subset. This attaches more weight to larger subsets whose omission could skew the total size estimate, and so reduce the impact of heavy tails on variance. Therefore the algorithm obtains smaller variance and hence better accuracy. Datar and Muthukrishnan [49] proposed a sampling algorithm to attack two problems: rarity of elements in a data stream and similarity between two data streams. The basic idea is to use a hash function in the sampling. The hash values are nearly random and therefore they are able to derive an unbiased estimator. Hershberger and Suri [86] proposed an adaptive sampling algorithm to approximate the convex hull of objects in a data stream. The algorithm first uses an uniform sampling and then adapt the sampling according to the distribution of the data objects. By this

means, both the error bound and the computation time are reduced. Recently stratified sampling was proposed in place of uniform sampling to reduce error caused by the variance in the data and also reduce error for group-by queries [3, 35]. Johnson et al. [98] reported the implementation of a *stream sample operator* in the Gigascope [46] DSMS. This stream sample operator is actually a framework, which can accommodate a wide variety of sampling algorithms over streams which are better than traditional random sampling algorithms.

Histograms

Histograms are summary structures to succinctly capture the distribution of values in a data set. They approximate the data by grouping attribute values into "buckets"(subsets) and then maintain certain summary statistics (e.g., the average) in each bucket to approximate the true values and frequencies. For many applications, there exist histograms that produce low-error estimates while using reasonably small space. They have been used in selectivity estimation and approximate query answering. There are different types of histograms (see [130] for a classification of the histograms). We describe as follows some popular ones and recent work on computing them over data streams.

- **Equi-width histograms:** These histograms partition the domain into ranges of the same length. Suppose we have β buckets in total, then the sum of the spreads in each bucket (i.e., the maximum minus the minimum in the bucket) is approximately $1/\beta$ times the range of all the values that appear (i.e., the maximum of all the values minus the minimum of all the values). Equi-width histograms are used in many commercial systems. To compute an equi-width histogram in one pass, we can simply maintain an array of β counters which count the number of elements that fall in each bucket. Building equi-width

histograms needs to know the minimum and maximum of the values a priori. This may not be possible sometimes. Also, data skew would result in equi-width histograms with poor quality. Recently Fu and Rajasekaran proposed to use a tree structure to organize equi-width histograms [68]. These histograms partition dense buckets into subbuckets to adapt to the distribution, so that the unknown maximum/minimum problem and data skew problem are alleviated.

- **Equi-depth histograms:** These histograms (also called *equi-height histograms*) partition the domain into ranges so that the number of records in each range is the same. Equi-depth histograms are less sensitive to the skew of the data distribution. The β (still assuming β buckets in total for the histogram) boundaries of the equi-depth histograms are also called quantiles [130]. Determination of these quantiles are expensive, therefore the use of equi-depth histograms are limited in commercial systems. Chaudhuri et al. [36] studied the problem of how much sampling is enough for computing approximate histograms. Specifically, they introduced a conservative metric to capture the errors of histograms and established optimal bound on sampling required for pre-specified error bounds. Then they can build histograms based on the sampling. Their algorithms require multiple passes over the data streams. Manku et al. [116] proposed algorithms to compute approximate quantiles with explicit error bounds in one pass of the data. They further proposed methods to exploit sampling with the algorithm to reduce memory requirement. But these algorithms must know the length of the input sequence in advance, which may not be possible in many stream applications. The same authors proposed algorithms that released this requirement by giving up deterministic guarantee on accuracy in [117]. In this paper, they also

presented a more efficient algorithm for quantile that is an extreme value, e.g., within the top 1% of the elements. More recently, Greenwald and Khanna [75] proposed an algorithm to improve the worst-case space requirement of previous work [116], which is $O(\frac{1}{\epsilon}\log^2(\epsilon N))$, to $O(\frac{1}{\epsilon}\log(\epsilon N))$, where ϵ is the approximation factor. This new algorithm gives deterministic error bound while not requiring a priori knowledge of the length of the input sequence.

- **V-optimal histograms:** These histograms have the least *variance* among all histograms using the same number of buckets. The variance of a histogram is the sum of the squared errors between the histogram values and the actual attribute values in each bucket. Let $v_1, v_2, ..., v_n$ be the set of values we want to approximate by histograms. The V-optimal histogram of this set is a piecewise-constant function $\hat{v}(i)$ that minimizes the sum $\sum_i (v_i - \hat{v}(i))^2$. Jagadish et al. [96] used dynamic programming to compute optimal V-optimal histograms for a given data set. The algorithm requires $O(N)$ space and $O(N^2\beta)$ time, where N is the data set size and β is the number of buckets. These requirements are too expensive for data streams. Guha et al. [79] adapted this algorithm to sorted data streams but with approximate answers. Their result is an arbitrarily close V-optimal histogram (i.e., with the error bound arbitrarily close to that of the optimal histogram), which requires $O(\beta^2 \log N)$ space and $O(\beta^2 \log N)$ time per element. The authors further adapted their algorithms over sorted data streams for two types of time windows, namely *agglomerative window* and *fixed window* in [78]. Their algorithm's update time per element is amortized to $O((\beta^3/\epsilon^2)\log^3 N)$, but the space requirement is linear with respect to the time window size. Subsequently, Gilbert et al. [71] removed the restriction that the data stream must be sorted and provided algorithms based on sketch techniques. They

view the data sequence as a vector of length N and each data record as an update to the sequence. The time to process a single update, time to reconstruct the histogram, and size of the sketch are each bounded by $\text{poly}(B, \log(N), \log||A||, 1/\epsilon)$. They first obtain a *robust* histogram (a histogram such that adding a few buckets does not change the approximation quality significantly) approximation for the data sequence, and then select a histogram of desired accuracy with β buckets.

- **End-biased histograms:** These histograms maintain exact counts on some of the highest frequencies and some of the lowest frequencies in separate individual buckets. The remaining frequencies (those in the middle) are all approximated by a single bucket. The task of computing end-biased histograms is actually to find out the most frequent items. In some literature, finding frequent items are also called *iceberg queries* [60] or finding *hot* items [44]. Demaine et al. [51] proposed an algorithm that processes each record in expected $O(1)$ time using $O(k)$ space, where k is the number of the most frequent items we want to find out. Manku and Motwani [115] provided stronger guarantee of finding all items that occur more than n/k times and not reporting any items that occur less than $n(1/k - \epsilon)$ times, where n is the number of records in the input sequence and ϵ is the approximation factor. Their algorithm makes use of a sampling that changes the sampling rate as the data elements arrive so that they can use a user-defined space to obtain the frequency counts within a good error bound. This algorithm uses $O(1/\epsilon \log \epsilon n)$ space. Cormode and Muthukrishnan [44] studied the case where records can be both deleted as well as inserted through a randomized algorithm. The algorithm monitors the changes to data distribution and maintains some summary data structure using $O(k \log k \log m)$ space, where

k is the number of hot items and m is the maximum possible value of the data items. When queries, the hot items can be found from the summary structure in $O(k \log k \log m)$ time . Babcock and Olston [18] studied the top-k monitoring problem in a distributed environment. In their approach, arithmetic constraints are maintained at remote stream sources to ensure that the most recently provided top-k answer remains valid to within a user-specified error tolerance. Distributed communication is only necessary when constraints are violated, therefore the overall communication cost is greatly reduced.

We summarize the above described histograms in Table 2.1.

Table 2.1: Histograms

Histogram type	Error	Computation cost	Use
Equi-width	Largest	Smallest	Implemented in many commercial systems
Equi-depth	Large	Medium	Not widely implemented in commercial systems due to higher computation cost compared to equi-width histograms
V-optimal	Smallest	Largest	Recently proposed, not widely implemented
End-biased	Small	Small	Recently proposed, not widely implemented

Some notes about these histograms are as follow. Equi-depth histograms were first proposed by G. Piatetsky-Shapiro and C. Connell [129] (called *distribution steps* in this paper). G. Piatetsky-Shapiro and C. Connell [129] also showed that equi-width histograms have a much higher worst-case and average error for a variety of selection queries than equi-depth histograms. V-optimal histograms were introduced in [93]. It is proved that optimal histograms must be *serial* [92]. A serial histogram means that the frequencies of the attribute values associated with each bucket are either all greater or all less than the frequencies of the attribute values associated with any other bucket. That is, the buckets of a serial histogram

group frequencies that are close to each other with no interleaving [94]. Therefore the V-optimal histogram are actually the V-optimal serial histogram. We can also have the V-optimal end-biased histogram, which minimizes the sum squared errors of the histogram among all end-biased histograms. The end-biased histogram is a subclass of the serial histogram, but with the restrictions specified in the previous paragraph. It is shown in [93] that the errors of the end-biased histograms are not far from the (V-optimal) general serial histograms, but the (V-optimal) end-biased histograms use much less number of buckets and have much smaller storage and usage complexity than the (V-optimal) general histograms.

Wavelets

Wavelets, ***wavelet analysis*** or ***wavelet transform*** is a commonly used signal processing technique like other transforms such as *Fourier transform*. It appeared just a few decades ago, but has been used widely in many areas such as data compression, computer graphics, databases as well as signal processing. Wavelet transform becomes so widely accepted is because, through a multi-resolution decomposition of the original signal, it overcomes Fourier transform's (or more accurately, *short time Fourier transform*'s) deficiency of not being able to achieve good frequency resolution and time resolution at the same time. After applying the wavelet transform over a signal, we obtain a number of wavelet coefficients (the number is the same as the length of the original signal), analogous to the amplitudes we get after the Fourier transform. The wavelet coefficients are projections of the signal onto a set of orthogonal basis vectors. The choice of the basis vectors determines the types of wavelets. The most popular one may be the Haar wavelets, which is easy to implement and fast to compute. Some of the wavelet coefficients we obtained may be small, therefore we can replace these ones

by zeros and have the data reduced. We can apply inverse wavelet transform on the reduced wavelet coefficients and get an approximation of the original signal. This is basically how wavelet transform is used for compression. It is shown that discrete wavelet transform based compression provides better image quality than discrete cosine transform based compression [150], and the new JPEG digital image standard, JPEG-2000, uses wavelets for all its codecs [139].

If we view the data stream sequence as a discrete signal, we can also use the wavelet transform to compress the stream and get a synopsis structure about the stream. Y. Matias et al. [118] introduced an efficient method for dynamic maintenance of wavelet-based histograms using probabilistic counting and sampling. A. C. Gilbert et al. [72] uses sketch techniques to maintain the coefficients of the wavelet transform of the input sequence. Later, the same authors improved their algorithm in construction time, error bound, and generality of the technique [71]. S. Guha et al. [77] proposed algorithms to maintain *extended wavelets* (a flexible storage method wavelet coefficients when multiple measures are present [50]) over streams based on dynamic programming and a near optimal approximate greedy algorithm. S. Guha and B. Harb [76] proposed algorithm to construct wavelet synopses over data streams that minimize non-Euclidean error measures. The authors also released some restriction such as synopsis having to be wavelet coefficients, or how the synopsis must be arrived at. Some other work also takes advantage of other virtues of wavelets. S. Papadimitriou et al. [127] proposed a modelling method to discover interesting patterns and trends in data streams. They use wavelet coefficients as data representation so that: i) redundancies, seasonalities and long-range behavior are eliminated; ii) arbitrary periodicities can be discovered.

2.2.2 Window queries[2]

When answering queries over the whole data stream accurately is impossible, we may choose to answer queries based on the data stream within a recent time window, during which the appeared data can fit in the limited memory and computation resources.

We can view window queries as a way of approximation, that is, we approximate the whole stream by the recent part of the stream, which may be the most interesting part to the users. However, this approximation approach is different from previous approaches discussed in Section 2.2.1. The previous approaches are independent of time. They may choose some data records during a time period as representatives of all the data records in that period or, if multiple records can have a same timestamp, they choose some of them as the representatives of all the data records at that timestamp. The window query approach approximates with regard to time. It chooses the data records in the current time window (data records at all the timestamps in the window) as the representatives of all the data records (data records at all timestamps). Therefore, *window* is an approximation technique that is orthogonal to the previous ones, and it is possible that both approximations coexist, for example, histograms over sliding windows [78].

From another perspective, evaluating queries in a windowed fashion may be requirements of certain applications. In [12], window queries were introduced to adaptively react over time to the changes in performance and data characteristics. In many real-time monitoring applications, the window query requirement is even obvious, for example, "report the number of IP packets sent by each IP source in every minute" (more explanation on this query is given in Section 2.4.1).

[2]Note that here "window" is a range in the time domain or in the data sequence domain. It is not a hyperrectangle in a multi-dimensional space as in spatial databases.

Next we would discuss how the "window" in window queries is defined. A window at any timestamp is determined by the beginning and end. A window for a continuous query is determined by how the beginning and the end change over time. Usually the users are interested in a window of fixed length, that is, the end of the window is always the beginning of the window plus the window length. This type of window is called the *sliding window*. The sliding window is actually determined by how many time units it proceeds (called the *pace*) after one window finishes processing. Two kinds of paces are often used, one time unit or the length of the time window. Using pace of one time unit results in the finest granularity to slide the window. Using pace of the window length results in disjoint partitions of the time domain, or called the *tumbling window*. Although tumbling window is a special case of sliding window, we usually mean a non-tumbling window when we talk about a sliding window, because algorithms to process tumbling window queries can be every different from those to process general sliding window queries. Figure 2.1 shows the difference between a general sliding window and a tumbling window. B. Babcock et al. [15] also considered moving the window over the data record domain (called *sequence-based window*), instead of the time domain (called the time-based window). An example of the sequence-based window query would be:"report the highest temperature in every five continuous temperature readings". We do not elaborate the exact definition of sequence-based window here since the extension from the time-based window is straightforward.

2.2.3 Sharing among queries

Resource sharing has been an important technique for reducing cost when multiple queries exist. In data stream applications, quite often the queries are continuous and sustain for a period of time. The chances are that a number of monitoring

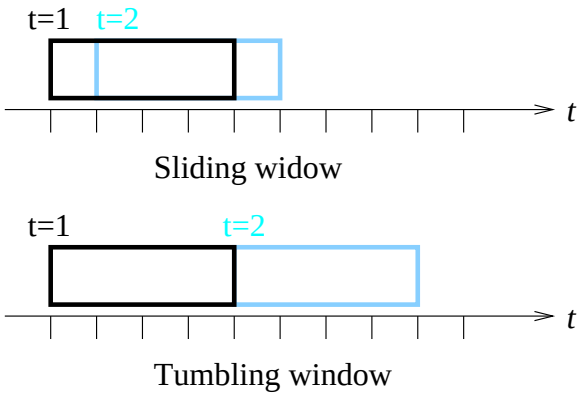


Figure 2.1: Sliding window and tumbling window

tasks coexist at the same time. Therefore, quite a few papers have been published regarding how to optimize the resource allocation. S. Madden et al. [114] showed how to share work and space across queries aggressively in the *eddy* query processing framework [12]. S. Chandrasekaran and M. J. Franklin [31] discussed sharing both computation and storage of different query results. J. Chen et al. [39] address the problem of system scalability by grouping web queries that have similar structures so as to share the computation and reduce I/O cost. A. Arasu and J. Widom [9] considered resource sharing among large number of sliding-window aggregates and presented a suite of sharing techniques that cover a wide range of possible scenarios: different classes of aggregation functions (algebraic, distributive, holistic aggregation functions as defined in [74]), different window types (time-based, tuple-based, suffix, historical), and different input models (single stream, multiple substreams). In Chapter 3 of this thesis, we study sharing the computation among aggregations queries with different group-by relations.

2.3 Data stream management systems

In this section, we review a number of existing data stream management systems (DSMSs). These systems may be specially designed for certain applications or may aim at a general stream system. We summarize their characteristics in Table 2.3 and subsequently discuss each of them. As assume the Gigascope system architecture in our work, we would discuss Gigascope in detail in the next section.

Table 2.2: Data Stream Management Systems

Name	Organization	Target application	Web stie
Aurora	Brandeis Univ. , Brown Univ. and M.I.T.	General	www.cs.brown.edu /research/aurora
COUGAR	Cornell Univ.	Sensor networks	www.cs.cornell.edu /database/cougar
CQ	Georgia Institute of Technology	Web content	disl.cc.gatech.edu/CQ
Gigascope	AT&T Labs-research	Network traffic	
NiagaraCQ	Univ. of Wisconsin and Portland State Univ.	Web content	www.cs.wisc.edu/niagara
STREAM	Stanford Univ.	General	www-db.stanford.edu /stream
TelegraphCQ	U.C. Berkeley	General	telegraph.cs.berkeley.edu
Tribeca	Bell Communications Research	Network traffic	

- **Aurora:** Aurora [40, 28, 2] is a data-flow system that uses the *boxes* (operators) and *arrows* (workflow) paradigm as in workflow systems. The structure is shown in Figure 2.2. Users can build continuous queries out of a small set of well-defined operators: *Filter*, *Map*, *Aggregate* and *Join*. *Filter* is like the relational operator *select*. It applies any number of predicates to each incoming tuple, routing the tuples according to which predicates they satisfy. *Map* is a generalized projection operator which can take multiple streams as

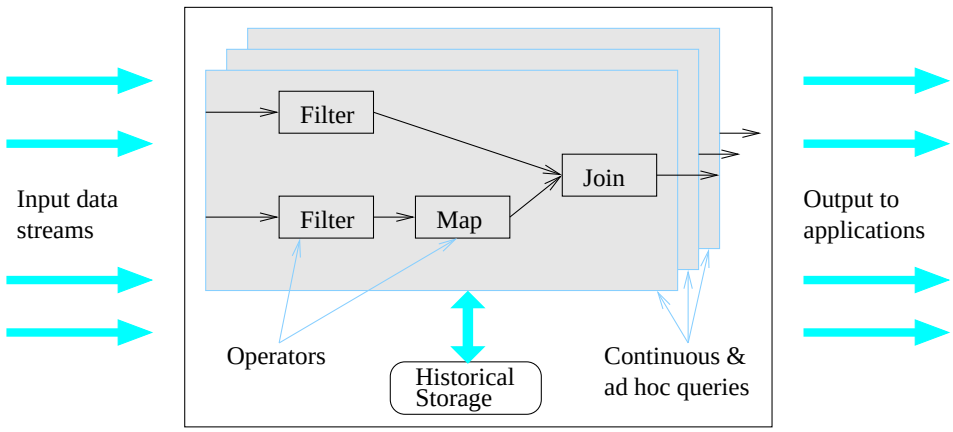


Figure 2.2: Structure of Aurora

inputs. *Aggregate* computes stream aggregates in a way that addresses the fundamental push-based nature of streams, applying a function across a window of values in a stream (e.g., a moving average). *Join* is an operator that performs join over two streams in a "banded" fashion, that is, a data record in one stream can only be joined with a record in another stream within a time range from its own timestamp. For example, in Figure 2.2, the query first applies a Filter operation on both data streams, then applies a Map operation on the second stream and finally performs a join over the two streams.

Each Aurora application defines one or more *Quality of Service* (QoS) functions/graphs in terms of some quality metrics over the query answers such as latency, value produced and loss-tolerance. Figure 2.3 shows some QoS graphs (the curves of QoS functions). For example, the first graph means the quality is good as long as the latency of query processing is within a certain time. The quality decreases when the latency goes beyond that time and becomes even larger. A load shedder is responsible for detecting overload situations and in these cases, adding "drop" operators into the processing workflow to filter out some messages based on the value of the records or

in a randomized fashion. The load shedder will get feedback from the QoS functions/graphs to achieve better overall QoS.

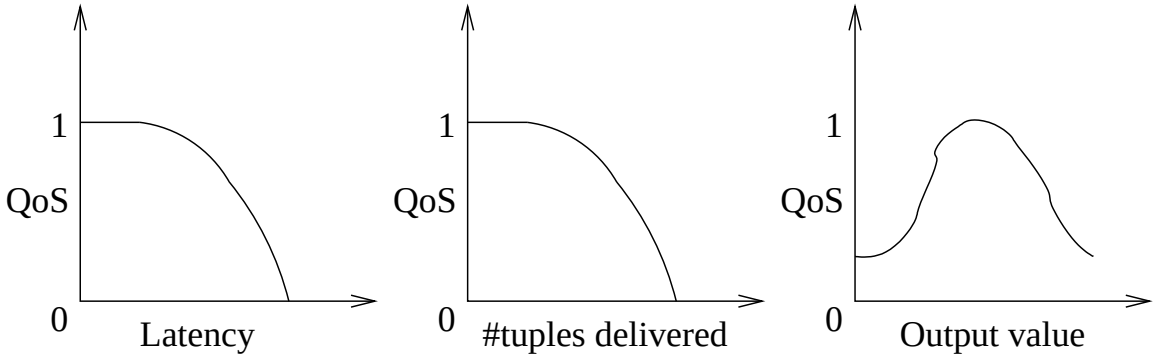


Figure 2.3: QoS graphs

Research focus of the Aurora project includes real-time data processing issues, such as QoS- and memory-aware operator scheduling, semantic load shedding for coping with transient spikes in incoming data rates, as well as novel hybrid data storage organizations that would seamlessly and efficiently combine pull- and push-based data processing [28]. Specifically, D. Carney et al. [29] studied scheduling issues such as which operators to schedule, in what order to schedule them, how many records to process at each execution step as well as various scheduling techniques such as batching of operators and records. N. Tatbul et al. [142] addressed the problem of determining when shedding is needed, where to insert drop operators and how much load to shed.

The Aurora group also proposed a distributed system, called *Borealis* [1] to achieve high scalability and availability. Based on this Borealis system, Y. Xing et al.[151] proposed a load distribution algorithm that aims at avoiding overload and minimizing end-to-end latency by minimizing load variance and maximizing load correlation. J.-H. Hwang et al. [90] studies various

recovery guarantees and pertinent recovery techniques in a distributed steam environment where the applications can tolerate some impreciseness.

- **COUGAR:** COUGAR [27] is a sensor database that models each type of sensors as a new Abstract Data Type (ADT). This sensor database has both stored data represented as relations and sensor data represented as time series. Signal processing functions are modelled as ADT functions that return sensor data. Like other data stream data, each record in the sensor data stream is of the same record type. The records can be of fixed length or of variable lengths. The particular attributes depend on the application. For example, the data from temperature sensors may have attributes *temperature* and *timestamp*. COUGAR's query execution engine is extended with a new mechanism to execute the sensor ADT functions so that evaluation of long-running queries are supported. A. Faradjian [61] proposed a new object-relational data type, the Gaussian ADT (GADT), that models physical data as Gaussian probability distribution functions (pdf's). Y. Yao and J. Gehrke [153] presented the design of a *query layer* for sensor networks based on the COUGAR system. The query layer accepts queries in a declarative language that are then optimized to generate efficient query execution plans with in-network processing. Y. Yao and J. Gehrke [152] also discussed some research problems to address in the sensor network, such as aggregation, query optimization and catalog management.

- **CQ:** CQ [112, 111] is a distributed event-driven continuous query system. The targeted applications of CQ is monitoring the contents on the large scale World Wide Web. The framework of CQ is organized around five abstract models: an object model, an event model, an observation model, a notification model and a resource model. All hardware and software components

of the system, such as the information producers and consumers, a clock, a file, a program and a process, are characterized by the *object model*. The *event model* characterizes distributed events, which may be defined in the time dimension (e.g., every 10 minutes or 10am everyday) or in the information space (e.g., some company's stock price drop by 10%). The events can be *predefiend*, which would be automatically detected by the system, or *user-defined*, which should be registered in the system by the user. The *observation model* defines the mechanisms by which event occurrences and patterns of event occurrences are observed. The *notification model* defines the mechanisms that users use to express queries on events and receive notifications. In the CQ system, the continuous query specification language is used to express the event monitoring requests. For example, the query shown

```
Creat CQ Savannah_weather_watch as
Query:    SELECT * FROM www.wns.nova.gov
          WHERE location like% 'Savannah' AND state = 'Georgia';
             OR location like% 'Fort Stewart';
Trigger:  20 minutes;
Stop:     1 year (default).
```

Figure 2.4: A Query example in CQ

in Figure 2.4 specifies the request for monitoring updates on weather conditions at the region from port of Savannah to Fort Stewart every 20 minutes, and detects the update on weather conditions at this region using a temporal event detector. Whenever an update event is signaled, the system takes the action of notifying some relevant person by email and delivering the updated result using a specific web URL pointer. Notifications are treated as independent communications between event detectors (observers) and recipients. The *resource model* defines where in the Internet the observation and notifi-

cation components are located, and how resources for the computations are allocated and accounted. Typically each information source is wrapped by a "wrapper" that establishes the correspondence between the resource entities and objects in the CQ system.

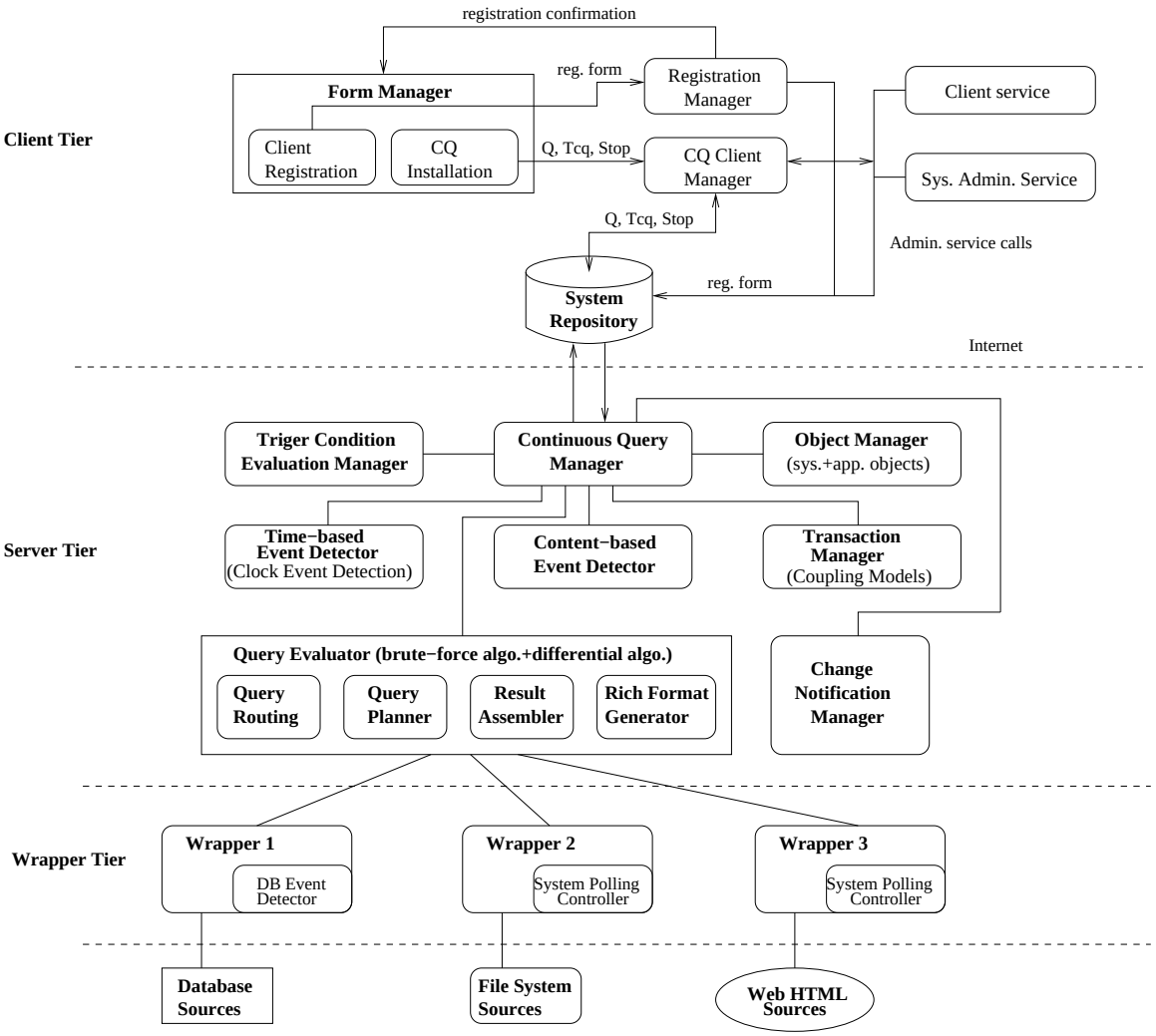


Figure 2.5: Architecture of CQ

CQ has a three-tier architecture, client, server, and wrapper/adapter as shown in Figure 2.5. The *client* tier has four components: (1) The *form manager* which provides the CQ clients with fill-in forms to register their continu-

ous queries; (2) The *registration manager* which manages user ID and passwords; (3) The *client system administration services* which provide utilities for browsing and updating installed continuous queries, for testing query triggers and tracing performance. (4) The *client manager* which coordinates different client requests and invokes different external devices. CQ's second tier, the *server* has three major components: (1) The *continuous query manager*, which is responsible for coordinating with the trigger condition evaluator and event detectors to monitor updates of interest, and coordinating with wrappers/adapters to track the new updates to the source data; (2) The *trigger condition evaluation manager*, which is in charge of evaluating the trigger condition for each registered continuous query whenever the update events of interest are detected by the event detectors; (3) The *query evaluator*, which executes a query Q whenever the trigger condition T_{cq} for the query is evaluated to be true. It also provides a guard for the *Stop* condition to guarantee the semantic consistency of the continuous query $(Q, T_{cq}, Stop)$. CQ's third tier, the *wrapper/adapters* tier serves as a "translator" between event detectors/the query evaluator and the information sources. One wrapper is needed for each data source due to the difference in how the data is accessed and represented. It translates the query into the format understood by the remote site and as result comes back, translates the response from the data source site into the CQ object format, which can be understood by the system. The three tiers could all be located on a single machine or distributed in different combinations among multiple computers connected through networks. Based on the system, further work has been done on wrapper construction over XML data [109, 110].

- **NiagaraCQ:** NiagaraCQ [39, 123] is a distributed database system for query-

ing distributed XML data sets on the Internet using a query language like XML-QL [53]. Due to the large volume of data on the Internet, scalability of the system is a significant issue. J. Chen et al. [39] addressed this problem by grouping continuous queries based on their query plan structures. J. Chen et al. [38] showed that among different incremental group optimization strategies, a *PullUp* strategy, in which selections are pulled above joins, is often better than other alternatives. A cost model for choosing grouping strategies is also proposed in this paper. J. F. Naughton et al. [123] studied the interaction between the search engine and the query engine.

- **STREAM:** STREAM [20, 58, 6, 122] is a general-purpose relational data stream management system. It supports a declarative query language using CQL [7] and flexible query execution plans which can be entered directly through a graphical interface or using XML. The architecture of STREAM is shown in Figure 2.6. The system has three store components: the stored relations, the archive and the scratch store. The *stored relations* are data kept as in the traditional relational DBMS. The *scratch store* keeps intermediate state of the streams, typically synopses on the streams. Finally, if some records need to be preserved or processed offline, they are copied to the *archive*. In the figure, the *data streams* coming in on the left produce the input infinitely and drive the query processing. The users can register *continuous queries* to the system. Query results are either in the form of data streams or relational results that are updated over time (similar to materialized views). A continuous query is registered using the declarative query language CQL and is compiled into a *query plan*. A separate plan is generated for each query, then the system would optimize the processing by trying to merge similar plans. The query plans can also be entered and merged manually for

experimenting different optimization strategies.

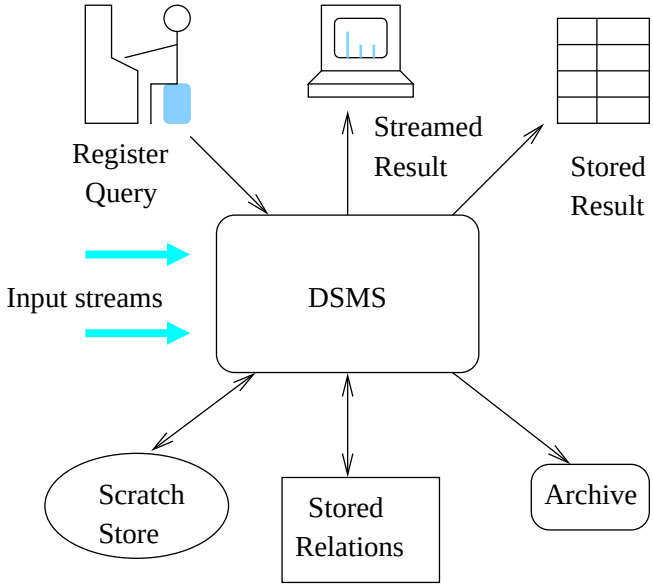


Figure 2.6: Architecture of STREAM

The query plan consists of three types of components: query operators, inter-operator queues and synopses. Inter-operator queues and synopses comprise the scratch store. Query operators are similar to those in a traditional DBMS, but enhanced by stream-oriented operators such as *window* and *sample*. Inter-operator queues are also similar to the approach taken by a traditional DBMS, which connect operators along *query paths* (query path is explained below). Synopses are used to maintain state associated with operators through techniques as discussed in Section 2.2.1. They summarizes the records seen so far at some intermediate operator in a running query plan, as needed for future evaluation of the operator. Figure 2.7 shows the example of a plan for two queries Q_1 and Q_2. This plan contains three operators O_1 (join), O_2 (selection), O_3 (join), four synopses $s_1 \sim s_4$ (two per joins, which keep summaries of the tuples of some intermediate stream) and four queues $q_1 \sim q_4$. Query Q_1 is a selection over a join of two streams R and S. Query Q_2 is a join of

three streams, R, S and T. The two plans share a subplan joining streams R and S by sharing its output queue q_3. Finally, the answers to the queries are given to the users (along the arrows pointing to Q_1 and Q_2 in the figure). For each query Q_i, there is a corresponding path in the data flow diagram from some input data stream though a set of query operators O_{i1}, O_{i2}, ... O_{ip} to node Q_i. This path represents the processing necessary to compute the answer to query Q_i, and it is called the ***query path*** for query Q_i.

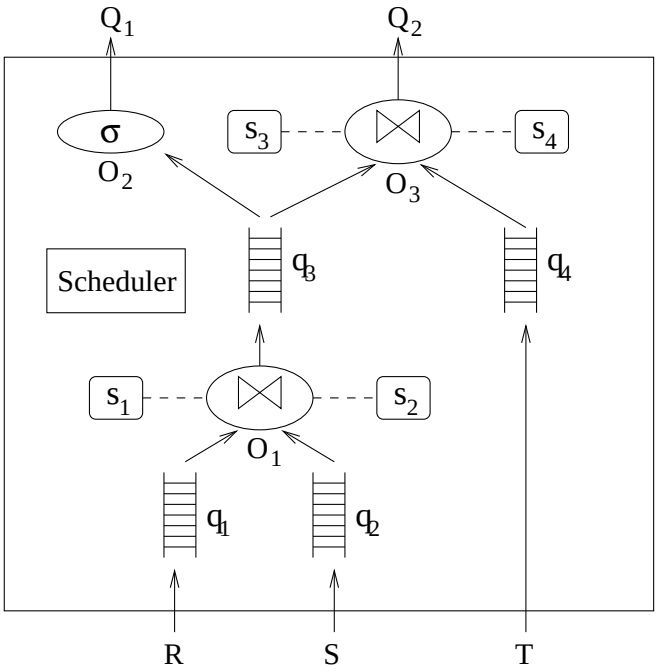


Figure 2.7: STREAM query plans

Research problems over the STREAM system include resource allocation and sharing [9], operator scheduling [13], caching [121, 19], load shedding [16] and general approximation techniques [15, 115, 17, 8].

- **TelegraphCQ:** TelegraphCQ [30, 104] is a stream system designed to process large number of continuous queries over high-volume, highly-variable streams. Figure 2.8 shows the architecture of TelegraphCQ. Its server consists of three

parts: (1) The *Telegraph Front End* contains the Listener, Catalog, Parser, Planner and "mini-Executor"; (2) The *TelegraphCQ Back End* is a separate process that does the actual query processing; (3) The *TelegraphCQ Wrapper ClearingHouse* is used to host the data ingress operators which make fresh tuples available for processing, archiving them if required. The query process runs as follows. The Postmaster listens on a well-known port and forks a Front End (FE) process for each fresh connection it receives. The listener accepts commands from a client and based on the command, chooses where to execute it. Data definition language (DDL) statements and queries over tables are executed in the FE process. Continuous queries that involve streams are "preplanned" and sent via the Query Plan Queue to the Back End (BE) process. The BE executor continually dequeues fresh queries and dynamically folds them into the current running query. Query results are placed into the Query Result queues. Once the FE has handled a query off to the BE, it produces an FE-local minimal query plan the Mini-Executor runs to continually dequeue results from the Query Result queue and send back to the user.

TelegraphCQ is a new generation system of the Telegraph project [84, 12], which targets at a global-scale, plug-and-play shared-nothing parallel query engine that can execute complex continuous queries over all the available data online. Therefore, many techniques proposed for the Telegraph project are applied in the TelegraphCQ system. Research in the Telegraph project has focused on shared query evaluation and adaptive query processing. *eddy* [12] is an online query reoptimization mechanism which can reorder the pipelined query processing operators to adapt to changing parameters such as operator costs, selectivities and data arrival rates. S. Madden et al. [114] showed the

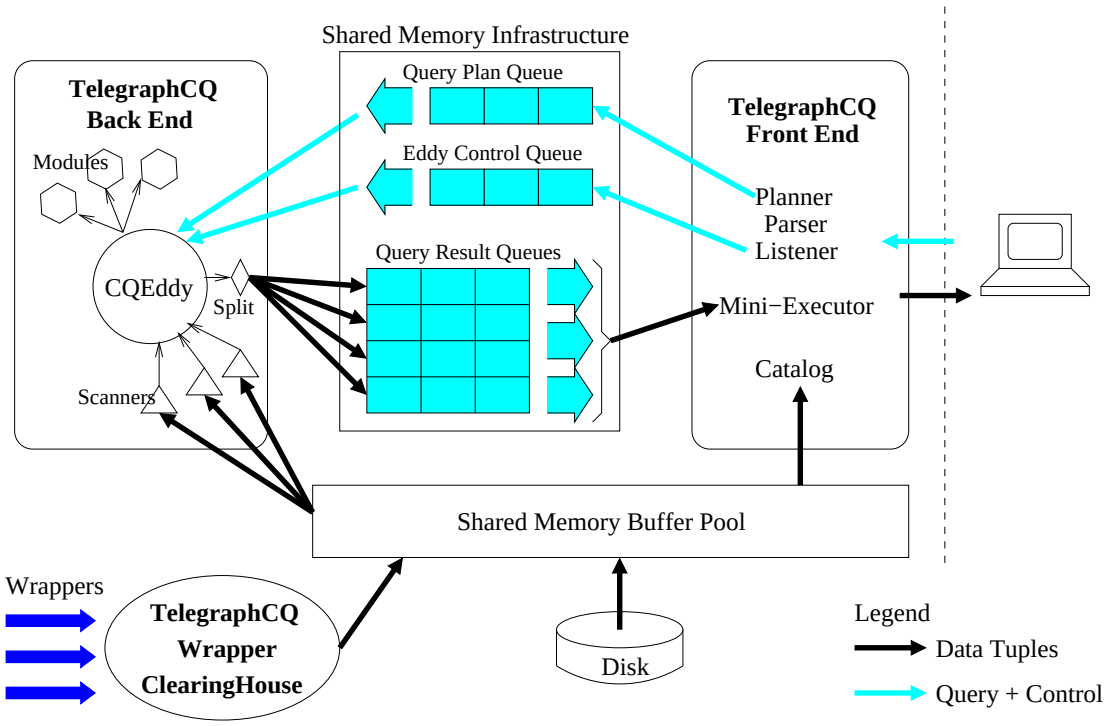


Figure 2.8: Architecture of TelegraphCQ

continuously adaptive continuous query implementation, which exploits not only adaptivity but also aggressive cross query sharing of work and space, over the Telegraph data flow engine [84]. S. Madden and M. J. Franklin [113] proposed the architecture *Fjord* as part of the Telegraph dataflow engine, for managing multiple queries over many sensors and show how it can be used to limit sensor resource demands while maintaining high query throughput. S. Chandrasekaran and M.J. Franklin [31] presented *PSoup*, a system built on Telegraph that combines the processing of ad-hoc and continuous queries by treating data and queries symmetrically, allowing new queries to be applied to old data and new data to be applied to old queries. M. A. Shah et al. [137] proposed a dataflow operator *Flux* that can repartition stateful operators online so as to adapt to changing workload and runtime conditions. V. Raman

et al.[131] proposed to decompose join operators into their constituent data structures (State Modules, or *SteMs*), and then dataflow among these SteMs is managed adaptively by an eddy routing operator. S. Krishnamurthy et al. [105] proposed the approach of *precision sharing*, aiming to share common work aggressively without unnecessary work. S. Chandrasekaran and M. J. Franklin [32] studied DSMS that process queries on both historical and live data. This paper proposed a framework in which multiple resolutions of summarization/sampling can be generated efficiently so that the query engine can select only a reduced version (an appropriate level of summarization depending on available resources) of the historical data. A. Deshpande and J. M. Hellerstein [52] proposed a mechanism *STAIRSs* which improves both adaptivity and performance of eddies by lifting the historical effects on the state of operators in the query.

- **Tribeca:** Tribeca [140] is a software system for monitoring and analyzing either a live network or recorded network traffic on tape. M. Sullivan and A. Heybey [140] discussed the query language and optimization issues. The query language contains the Tribeca type system, basic stream operators (such as AND, OR, NOT), *demultiplexing* operators used to partition streams, *multiplexing* operators used to combine streams and *window* operators. Optimization issues include minimizing query execution time and trying to fit the intermediate state associated with the query into memory.

2.4 Gigascope: a network stream system

In this section, we have a detailed look at the Gigascope since we have adopted it as the basic architecture of our work. Gigascope [47, 48, 46] is a DSMS specialized

for high speed network traffic monitoring, developed at AT&T Labs-research. We would discuss its query language, architecture and related research, respectively as follows.

2.4.1 Query language and query model

There are two options for the query language: a standard database language such as SQL, or a special purpose language designed to succinctly and efficiently express network monitoring queries such as the procedural ones used in Tribeca [140] and Hancock [45]. Gigascope took the standard language approach (with some adaptation to the system) considering that the advantage of using a well-known and well-researched query language outweighed the advantages to be had from a special purpose language. The Gigascope query language, *GSQL*, is a pure stream query language with SQL-like syntax but restricted version of SQL due to the functionalities supported in Gigascope.

On the issue of query semantics, many systems have adopted the model of continuous queries over a sliding window (recall the discussion on window queries in Section 2.2.2). This approach has the advantage of precise presentation of the results at very fine time granularity. However, sliding window queries have several drawbacks. First, it is hard to compose complex queries. Specifically, we cannot directly use the result of a sliding window query as the input of another continuous query since these results contain overlapping states. We have to reverse-interpret the result as a stream before feeding it to another query. Second, query evaluation of sliding window queries is less efficient. They may not satisfy the requirement of the extremely high-speed network traffic data. Gigascope takes the tumbling window query model as the tumbling window query can be evaluated much more efficiently than the sliding window query. The tumbling window query gives coarse result

compared to the sliding window query, but this may not be a significant problem for network traffic analysis since usually approximate answers suffice. Another effect of using the tumbling window query model is that, result of one query is a stream and can be used as input to other queries directly, therefore we can easily compose stream queries. To implement the tumbling window query, a "timestamp" must be enforced. Network data usually contains several types of timestamps and also sequence numbers; they can be used as the "timestamp" for the tumbling window queries. Gigascope provides a mechanism for schema annotations to indicate which stream fields act as the timestamp. For example, consider the following query (Figure 2.9). IPPackets is a relation defined by the user to represent IP packets.

```
select tb, SourceIP, count(*) as cnt
from IPPackets
group by time/60 as tb, SourceIP
```

Figure 2.9: A Query example in Gigascope

Its attributes are extracted from raw network data, usually containing SourceIP, SourcePort, DestinationIP, DestinationPort, time, and maybe other attributes such as packet length etc, depending on its exact definition given by the user. The meaning of SourceIP, SourcePort, DestinationIP and DestinationPort are clear. The attribute time is the timestamp when the packet arrives. Here we assume the unit for time is second. What the above query does is actually to obtain the number of packets sent from every sender per minute (that is, show each one's IP address with the number of packets sent from it). This query is meaningful as long as `time` has been annotated as the timestamp.

2.4.2 Architecture of Gigascope

Gigascope has two major components, a *stream manager* and a *registry*. The stream manager tracks queries that can be activated. Each query is a process. When the queries are started, they register themselves with the registry. When a user or a query needs to subscribe the output of a query (a query can use the output as input), it submits the query name to the registry and receives a query handle in turn. The process then contacts the query to set up communication through shared memory.

Gigascope is featured by a two-level query processing architecture. Any query is decomposed to a low-level subquery, called *LFTA*[3] and a high-level subquery, called *HFTA*. Usually LFTAs work on the network traffic directly, either through libpcap or in the Network Interface Card (NIC). The LFTAs provide data streams to the stream manager, which can be used by HFTAs or the user applications. HFTAs also provide streams to the stream manager, which can be used by user applications. All the FTAs provide the *schema* of their output to the registry, including attribute names, data types, the query that FTA executes, and temporal properties of the attributes.

When a query is submitted, it is analyzed by the system to determine which parts should be executed as LFTAs and which parts as HFTAs. After the query is decomposed, it is translated into executable code. HFTAs are implemented as separate processes using template operators written in C++. LFTAs are translated into C for linking and execution in a run time system (RTS) discussed below.

In Gigascope, a replace run time system (RTS) is written for the Tigon Gigabit Ethernet NIC. When the RTS receives a packet from the network, it presents the packet to a set of processing modules. These modules can perform arbitrary pro-

[3]FTA stands for "Filter, Transform, Aggregate".

cessing (within resource constraints) and produce output streams for transmission to the host computer. The LFTAs are translated into C modules that follow the API expected by the RTS. The packets are linked into an executable and loaded onto the NIC. A collection of templatized push-based operators have been implemented for evaluating higher-level queries such as selection, projection, aggregation and stream merge. These operators make use of the temporal properties of the attributes in a stream to optimize query processing (e.g., emit aggregates as soon as possible). Given the schema of a stream and the query evaluated on it, temporal properties of the output stream can be deduced. Thus, both LFTAs and HFTAs produce data streams with temporal properties and the users can compose complex queries using stream operators.

The design of the two-level architecture has several advantages. First, the LFTAs are lightweight queries which perform preliminary filtering, projection and aggregation. They are linked to the RTS of the NIC (which would be discussed more below), therefore the preliminary queries can be evaluated without additional data transfers, which makes them very fast. Second, it is observed that data analysis is best done close to the data source to reduce the data volume as soon as possible. After the LFTA subqueries, the data traffic passed to the HFTAs is greatly reduced, enabling the HFTAs to take their time to perform more complex operations. Take aggregation as an example. The LFTA of the aggregation operator uses a small hash table to direct packet counts. A hash table collision results in a tuple's count ejected from the hash table and written to the output stream, which is passed on to HFTA. HFTA will complete the aggregation as in the *subaggregates* and *superaggregates* algorithm used in data cube computation. Due to the temporal clustering of network stream data (all packets in a flow have the same source/destination IP/port), even with a small hash table has a low col-

lision rate and the LFTA is effective in early data reduction. Third, the LFTAs usually use a small size of memory and the operations can fit L2 cache of the CPU, which makes the execution rather efficient. Because of these advantages, Gigascope meets its goal of being a lightweight stream query processing system which works as fast as hand-written specially tuned programs while also having the flexibility of composing complex queries.

The Gigascope design also has some limitations. Because LFTAs are linked into the RTS which is possibly into the NIC, all queries which generate LFTAs must be submitted in a batch. Changing the set of LFTAs requires the system to be stopped, the RTS changed and restarted. HFTAs can be submitted at any point. To increase this flexibility, the queries can accept query parameters, which are similar to constants and which are specified at query installation time and can be changed on-the-fly. Therefore the small inflexibility can be alleviated and is tolerable since our major goal is to achieve extremely high speed stream processing.

2.4.3 Research based on Gigascope

Several research directions are pointed out in [46]. One issue is that the choice of operator implementation affects the attribute ordering properties of the output, which in turn affects the performance of downstream operators. Another issue is the proper use of sampling and approximation techniques and the integration of them into the query language under the users' control. Other issues include how to integrate complex group definition mechanisms as described in [34] into GSQL and a scalable scheme for expressing stream query sources. G. Cormode et al. [43] studied how to use user-defined aggregate functions (UDAFs) to implement selection-based and sketch-based algorithms for holistic aggregate (quantiles and heavy hitters) operators in Gigascope. Key performance bottlenecks are identified

and novel techniques to improve efficiency are proposed. T. Johnson et al. [98] reported the implementation of a *stream sample operator* in the Gigascope [46] DSMS. This stream sample operator can accommodate a wide variety of sampling algorithms such as reservoir sampling [145], subset-sum sampling [56], min-wise hash sampling [49] and heavy hitter algorithm [115] over streams which are better than traditional random sampling algorithms. In Chapter 3, we study how to optimize processing in case of multiple aggregations. The idea is to maintain additional hash tables in LFTAs, called *phantoms*, to share computation among queries. The technique optimize the phantom choosing and space allocation processes. T. Johnson et al. [99] introduced a system for punctuation-carrying heartbeat generation and showed how heartbeats can be generated in query execution plans to unblock stream operators. Note that the DISC method for approximate nearest neighbor search presented in Chapter 4 of this thesis is generally applicable. We would show how it can be implemented in the Gigascope two-level query processing architecture.

2.5 Related work

2.5.1 Work related to aggregations over data streams

The focus of our work is how to optimize the computation when processing *multiple* aggregate queries over the network data streams. This work is closely related to the problem of multi-query optimization, i.e., optimizing multiple queries for concurrent evaluation. This problem has been around for a long time, and several techniques for this problem have been proposed in the context of a conventional DBMS. P. A. V. Hall [82] uses operator trees to represent the queries and a bottom-up traversal algorithm to find common subexpressions. N. Roussopoulos [133]

uses query graphs [149] to find views for indexing to minimize the total cost of answering all queries. U. Charkravarthy and J. Minker [33] extended query graphs to integrated query graphs and proposed an algorithm based on integrated query graphs. S. Finkelstein [65] studied how query processing can be improved using answers produced from early queries when no common expressions are found for multiple query optimization. Per-Ake Larson and H. Z. Yang [107] showed when and how can a query be computed from derived relations. T. Sellis [136] analyzed this problem from a systematic point of view and proposed heuristics under different system architectures. The basic idea of all these techniques is the identification of common query subexpressions, whose evaluation can be shared among the query execution plans produced. This is also the basis for sharing of filters in pub-sub systems (see, e.g., [59]). Our technique of sharing computation common to multiple aggregate queries is based on the same idea, but our technique is specially devised for data stream applications where the data inputs stream by the system at extremely high speed.

Our problem also has similarities to the view materialization problem, which is described below. View materialization is a common way to reduce computation when processing queries in data warehouses, where a data set is viewed as a multidimensional data cube. Cells of the data cube, namely *views*, are chosen to be materialized so that aggregate queries can be answered from materialized views instead of computed from the raw data. As materializing all views may be too expensive (in terms of both computation and space), one needs to decide which views to materialize, which is the view materialization problem. This problem has been studied extensively in the literature (a survey can be found in [80]). K. A. Ross et al. [132] has studied the problem of identifying, in a cost-based manner, what additional views to materialize, in order to reduce the total cost of view main-

tenance. V. Harinarayan et al. [83] proposed a lattice framework to represent the dependencies among views and a greedy algorithm to determine which views to materialize. Our idea of additionally maintaining phantoms, and choosing which phantoms to maintain, to efficiently process multiple aggregations is based on the same conceptual idea. However, due to two differences: i) maintaining a view uses fixed space while maintaining a phantom can use flexible space; ii) maintaining a view is always beneficial while maintaining a phantom is not, the greedy algorithm proposed by V. Harinarayan et al. [83] is inapplicable to our problem. Although we can somehow adapt the view materialization algorithm to our problem, the adapted algorithm is very cumbersome and shown to be very inefficient by our experiments. We devise a novel greedy algorithm for our problem which optimizes the cost.

Many papers (see, e.g., [39, 28, 30]) have highlighted the importance of resource sharing in continuous queries. [39, 114] use variants of predicate indexes for resource sharing in filters in continuous query processing systems. In the context of query processing over data streams, A. Dobra et al. [55] considers the problem of sharing sketches for approximate join-based processing.

On computing aggregates over data streams, J. Gehrke et al. [70] proposed single-pass techniques for approximate computation of correlated aggregates such as "compute the percentage of international phone calls that are longer than the average duration of a domestic phone call", whose exact answer requires multiple passes over the data. S. Chaudhuri et al. [35] uses sampling-based technique to provide approximate answers to aggregate queries, but different from previous studies, S. Chaudhuri et al. [35] treats the problem as an optimization problem whose goal is to minimize the error in answering queries in the given workload. A. Dobra et al. [54] uses randomized techniques that compute "small" summaries of the streams to provide approximate answers for general aggregate SQL queries

with provable guarantees on the approximation error. A. C. Gilbert et al. [72] uses sketching techniques to approximate wavelet projections that represent the original data stream. However, none of these papers considered query optimization through sharing among different queries.

S. Chandrasekaran and M. J. Franklin [31] and A. Arasu and J. Widom [9] studied sharing data structures among sliding window queries posed on the same input stream, aggregate function and aggregated attributes. The sharing queries only differ in the window specifications. The approach proposed in [31] is to maintain a ranked n-ary tree on the recent data that can cover all the sliding windows of the queries. Each leaf of the tree is a data record of the stream and the leaves are sorted in the order of their insertion times. Each internal node stores the value of the aggregate function computed over the descendant leaves of that node. This algorithm has an update and search cost of $O(\log n)$, where n is the number of elements maintained in the tree, which equals the number of elements needed to cover all the query windows. The other work, A. Arasu and J. Widom [9], proposes to exploit the *distributive* or *algebraic* properties of the aggregate functions. These properties ensure that the aggregation over the union of two sets of elements can be computed from the aggregation over each set (for example, $SUM(S_1 \bigcup S_2) = SUM(S_1) + SUM(S_2)$). The idea of A. Arasu and J. Widom [9] is to precompute the aggregation over some intervals and store them. When an aggregate query is issued, the window of the query is decomposed into intervals whose aggregations have been precomputed. Then the asked aggregation can be computed from the precomputed aggregations. This approach uses more space (to store the precomputed aggregations) and may have more update costs. However, it has better search performance and works well especially when the number of queries is large. Two algorithms are proposed in [9], B-Int and L-Int, for general

aggregate queries. B-Int requires $O(n)$ space, has an amortized update time complexity of $O(1)$ and a worst-case search time complexity of $O(logW)$, where W is the size of the query window. L-Int requires $O(\gamma n)$ space, has an amortized update time complexity of $O(\gamma)$ and a worst case search time of $O(1)$, where γ is a parameter used in partitioning the stream sequence. As mentioned, the algorithms proposed in [31] and [9] use a sliding window model and share data structures for aggregates differing only in the query window specifications. Our work, based on the Gigascope architecture, assumes a tumbling window model. Moreover, our approach shares the computation among different aggregated attributes, or relations in general.

2.5.2 Work related to approximate nearest neighbor search over data streams

There have been extensive studies on the Nearest Neighbor (NN) problem in traditional databases. Early algorithms are based on R-tree-like structures [81]. The R-tree is a hierarchical structure consisting of nodes, each of which corresponds to the minimum bounding rectangle (MBR) of the objects in or under this node. For example, Figure 2.10(a) shows a set a points {A, B, C, D, F, G, H, I, J, K} and Figure 2.10(b) shows how these points are organized in an R-tree with node capacity of three entries. This R-tree has three levels, where *level* 0 contains the data points and the other levels contain entries $(E_1, E_2, ...)$, each consisting of an MBR and pointers to the nodes in the next level of the tree. In Figure 2.10(a), the dark gray regions are the MBRs of the *level* 0 nodes and the light gray regions are the MBRs of the *level* 1 nodes.

The NN algorithms based on the R-tree follow a branch-and-bound approach, that is, they traverse the tree in certain order and obtain some candidate NN from

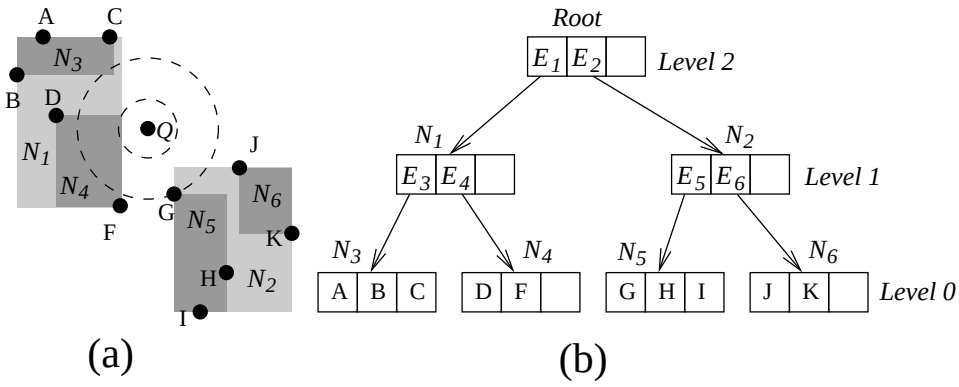


Figure 2.10: An R-tree example

the nodes visited. On the other hand, some distance bounds can be derived for the points in the rest of the nodes. If the current candidate NN distance is smaller than the minimum distance between an unvisited node and the query point, the algorithm will skip this node. These algorithms mainly differ in the order they traverse the tree. For example, a depth-first algorithm is proposed in [134]. Starting from the root, it proceeds down the tree until a path is exhausted and then moves back one level and proceeds down another path. For each node, the entries are visited in the order of the minimum distance between their MBRs and the query point. For query point Q in Figure 2.10(a), the algorithm first visits the nearest node to the root, that is, N_1. Among the entries of N_1, E_4 has the nearest MBR, therefore the algorithm visits N_4 and gets a NN candidate F. Then it moves one level up and visits the second nearest node in N_1, that is, N_3, but find no points nearer than F, so it moves one level up and begins to visit N_2. Following similar steps, the algorithm stops when it gets the NN candidate G in N_5, which has a smaller distance to Q than the minimum distance between N_6 and Q. Therefore, G is the nearest neighbor.

Another algorithm traverses the tree in a closest-first style [87]. It accesses the nodes in the order of increasing distance to the query point. After N_4, the algorithm

goes to N_2, instead of N_3, and then N_5. At this moment, the NN candidate G has a distance smaller than the minimum distance of the remaining nodes, N_3 and N_6. Hence they need not be accessed. However, this closest-first traversal is achieved by using a priority queue which maintains the information of the nodes that may contain the nearest neighbor. It may cause problems when memory is stringent.

The above R-tree based algorithms work well in low-dimensional space. It was observed later that, in medium- and high-dimensional spaces, the overlaps in the R-tree become significant and its performance deteriorates rapidly. Although variations of the R-tree have been proposed [148, 100, 25, 135], the intrinsic difficulty of (k)NN search in high-dimensional space is still unresolved. A quantitative analysis in [146] shows that these structures are outperformed by a sequential scan of the data when dimensionality is high. Therefore a technique based on data quantization called the vector approximation-file (VA-file) was proposed [146] to accelerate sequential scan.

The VA-file uses bit strings, each of which is typically 4 to 8 bits long, to compress the original data, which are usually floating point variables, therefore the size of the VA-file is 1/8 to 1/4 of the original data. Figure 2.11(b) shows how the set of points {A, B, C, D} shown in Figure 2.11(a) are represented in a VA-file. In this example, the data space is divided into four partitions in each dimension and each partition is assigned a two-bit string. A data point is represented by the bit strings of all the dimensions, which is called the *vector approximation* (VA) of the point. Vector approximations of all the points are stored sequentially resulting in a VA-file. The search algorithm first scans the VA-file and maintains a kNN candidate list. According to the distance bounds derived from the VAs, most of the points can be pruned in the VA-file scan. Then the few points in the kNN candidate list are retrieved from the original data file directly for further check. The IQ-tree [24]

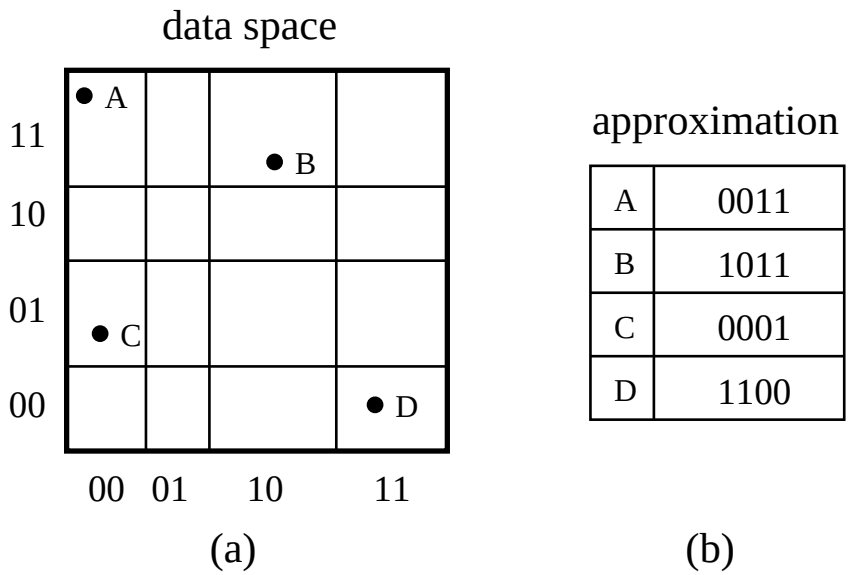


Figure 2.11: A VA-file example

combines the R-tree-like structure and the approximation technique, and exploits a cost model to decide which approach to adopt.

Some methods are proposed for kNN search in a metric space such as the M-tree[41], Omni-family indexes[64] and the iDistance [97]. In [97], it is reported that the iDistance performs better than sequential scan on uniform data up to 30 dimensions and beats existing techniques on clustered data sets in even higher dimensions.

All the above methods assume that the data are disk-resident and can be scanned multiple times. As such, they are not suitable for processing data streams that typically require one-pass algorithms as the data are not stored on disk and are too large to fit into memory.

Due to the intrinsic difficulty of the kNN problem, some researchers have looked into approximate solutions. A structure based on quadtrees for answering kNN queries approximately was proposed in [26]. The relative error is dependent on the dimensionality d so that the larger the value of d, the greater the relative error.

Then the $(1+\epsilon)$-approximate nearest neighbors problem was studied [10, 11, 106], in which the relative error ϵ is a constant specified by the user. An algorithm requiring exponential time in d and linear space was proposed in [10] and follow-up studies improved its time/space requirements [91, 106, 108]. These studies share the common feature of a relative error bound, while in our work, we specify the error as an absolute value. Moveover, these studies assume the whole data set is in memory, therefore they are inapplicable to the data stream model, where only a small portion of the data can be maintained in memory.

The ND P-sphere tree [73] also accelerates kNN search by providing non-exact answers. The algorithm guarantees that for a user specified percentage of time, the returned answers are correct, but it cannot distinguish between the correct and incorrect answers. Again, this technique is proposed for traditional database where data are disk-resident and hence inapplicable for data stream queries.

A most closely related work may be [69], which proposes using the Fast Fourier Transform to solve the problem of pattern similarity search over data streams. However, it uses values from the incoming stream (time series) as queries to identify the nearest neighbors from an existing pattern database. In our problem setting, queries are specified by users on demand and we seek to locate nearest neighbors in the incoming data stream. The idea of [69] is to take advantage of batch processing through prediction of the data streams. When the actual time series arrives, prediction error lower bounds and upper bounds are calculated and used together with the predicted distances to filter candidate patterns. Their error bounds are used for prediction but the answers are accurate. In our case, the error bounds are for the answers, which means that the answers are approximate. In [42], hamming norms are used to measure the similarity between two streams, and in [143], a regression-based algorithm was proposed to mine frequent temporal patterns for

data streams. Reverse nearest neighbor aggregate queries over streams have also been investigated in [102].

2.6 Summary

In this chapter, we first discussed the data stream model and queries over data streams. We have seen the differences of data stream applications from traditional databases and the necessity of algorithms specially devised for data streams. Then we examined commonly used techniques in data stream algorithms: approximation, window queries and sharing. Next, we reviewed a number of existing data stream management systems (DSMSs), and had a close look at Gigascope, a DSMS especially designed for network data streams. Finally, we explored existing work related to the two problems we study.

CHAPTER 3

Efficient Aggregation Over Data Streams

Aggregation is a basic operation for both traffic management and some security applications as discussed in Section 1.2. The need for exploratory IP traffic data analysis naturally leads to posing related aggregation queries on data streams, that differ only in the choice of grouping attributes. In this chapter, we address this problem of efficiently computing multiple aggregations over high speed data streams, based on the two-level LFTA/HFTA query processing architecture of Gigascope.

The rest of the chapter is organized as follows. We first show how a single aggregation is processed in the Gigascope DSMS in Section 3.1. Then we examine the problem of multiple aggregations and introduce an approach, that is, maintaining phantoms, to optimize the multi-query processing in Section 3.2. Section 3.3 gives the cost model for optimization and formalize our problem which consists of two sub-problems, phantom choosing and space allocation. Hardness result of the problem is also given in this section. A synopsis of our proposed solution follows in Section 3.4. Section 3.5 presents the phantom choosing algorithms. Section 3.6

presents our collision rate model, a key component in the space allocation scheme, which is in turn used by our proposed greedy algorithm. Section 3.7 analyzes space allocation schemes. Section 3.8 presents our experimental study.

3.1 Single aggregation

Let us first see how a single aggregate query is processed in Gigascope. Consider the following query Q0, which is the same as the one shown in Figure 2.9.

```
Q0:  select tb, SourceIP, count(*) as cnt
     from IPPackets
     group by time/60 as tb, SourceIP
```

As explained in Section 2.4.1, IPPackets is a relation defined by the user to represent IP packets. Its attributes are extracted from raw network data, usually containing SourceIP, SourcePort, DestinationIP, DestinationPort, time, and maybe other attributes such as packet length etc, depending on its exact definition by the user. The meaning of SourceIP, SourcePort, DestinationIP and DestinationPort are clear. The attribute time is the timestamp when the packet arrives. Here we assume the unit for time is second. What the above query does is actually to obtain the number of packets sent from every sender per minute (that is, show each one's IP address with the number of packets sent from it).

Figure 3.1 is an abstracted model of Gigascope. M_L corresponds to the LFTA, and M_H corresponds to the HFTA. Q0 is processed in Gigascope as follows. When a network data stream record arrives, it is observed at M_L. M_L maintains a hash table consisting of a specified number of entries, and each entry is a {*group-by-attributes*, *count*} pair. The item *group-by-attributes* stores the value of the attributes grouped by in the query, and this item always records the most recently observed values that

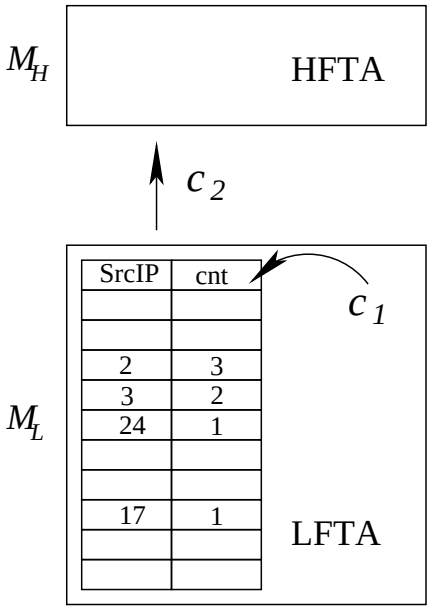


Figure 3.1: Single aggregation in Gigascope

hash to this entry. In query Q0, we group the results by SourceIP every minute, so the first item of an entry stores the most recent source IP hashed to this entry. The second item of an entry, *count*, keeps track of the number of times the value stored in the first item of the entry has been observed without any other record hashed to this entry.

When a new record r hashes to entry bk, Gigascope checks if r belongs to the same group as the existing group in bk (that is, to check if the value of r's group-by attributes is the same as the value stored in bk). If yes, the *count* of bk is incremented by 1. Otherwise, we say a *collision* occurs. In this case, first the current entry in bk is evicted to M_H. Then, a new group corresponding to r is created in bk in the hash table in M_L and the *count* of the group corresponding to r is set to 1. Because M_H is much larger than M_L, we can store all the groups in a big table with their counts. When the group in the evicted entry already exists in the table in M_H, we just add the new count in the evicted entry to the existing

count of the group in the table.

Consider an example stream with the following (simplified) source IP's: 2, 24, 2, 2, 3, 17, 3, 4. Suppose we use a simple hash function which is the remainder modulo 10 and maintain a hash table of 10 entries at M_L. The first item in the stream, 2, hashes to the third entry. We check the entry in the hash table, which is empty in the beginning, so we set the third entry of the hash table as $(2, 1)$. Then we see 24, which hashes to the fifth entry of the hash table. Similarly $(24, 1)$ is put in the fifth entry of the hash table. The third item is 2, which hashes to the third entry of the hash table. We check the hash table and find that the existing source IP in the third entry is also 2, which means they are in the same group, so we just increment the count of the entry by 1, resulting in $(2, 2)$. The rest of the items are similarly processed one by one. After we processed the $7th$ item, which is 3, the status of the hash table is shown in Figure 3.1. The next item 4 hashes to fifth entry of the hash table. The existing group value in the fifth entry is 24, different from the new value 4, therefore a collision happens. In this case, we evict the existing entry $(24, 1)$ to M_H and set the fifth entry to $(4, 1)$.

Query Q0 is processed by Gigascope in an epoch by epoch fashion (that is, tumbling window query), for an epoch of length 1 minute (i.e., time/60). This means that at the end of every minute, all the entries in M_L would be evicted to M_H to compute the aggregation results for this epoch at the M_H. At M_H, multiple tuples for the same group in the same epoch may be seen because of evictions, and these are combined to compute the desired query answer.

Gigascope is especially designed for processing network level packet data. Usually this data exhibits a lot of clusteredness, that is, all packets in a flow have the same source/destination IP/port. Therefore, the likelihood of a collision is very low until many packets have been observed. In this fashion, the data volume fed

to M_H is greatly reduced.

3.1.1 Cost of processing a single aggregation

In Gigascope, LFTAs are run on a Network Interface Card (NIC), which has a memory constraint. Also LFTAs use a small size of memory which fits into the L2 cache of the CPU so that LFTAs are run very efficiently. Therefore, we have a memory size constraint for M_L, depending on the hardware (typically several hundred KB). M_H has much more space and a much reduced volume of data to process, so the processing at M_H does not dominate the total cost. The overall bottlenecks are:

- The cost of looking up the hash table in M_L, and possible update in case of a collision. This whole operation, called a *probe*, has a nearly constant cost c_1.
- The cost of transferring an entry from M_L to M_H. This operation, called an *eviction*, has a nearly constant cost c_2.

Usually, c_2 is much higher than c_1 because the transfer from M_L to M_H is more expensive than a probe in M_L.

The total cost of query processing thus depends on the number of collisions incurred, which is determined by the number of groups of the data and collision rate of the hash table. The number of groups depends on the nature of the data. The collision rate depends on the hash function, size of the hash table, and the data distribution. Therefore, generally, what we can do is to devise a good hash function and allocate more space (within space and peak load constraints, as we will discuss more later) to the hash table in order to minimize the total cost.

3.2 Multiple aggregations

3.2.1 Processing multiple aggregations naively

Given the method to process single aggregate queries, and its cost model, based on the Gigascope architecture, we now examine the problem of evaluating multiple aggregate queries. Suppose the user is interested in the following three aggregate queries:

```
Q1:  select tb, SourceIP, count(*) as cnt
     from IPPackets
     group by time/60 as tb, SourceIP

Q2:  select tb, DestinationIP, count(*) as cnt
     from IPPackets
     group by time/60 as tb, DestinationIP

Q3:  select tb, DestinationPort, count(*) as cnt
     from IPPackets
     group by time/60 as tb, DestinationPort
```

In the sequel, for the sake of simplicity, we will use R to represent the relation of the data stream and A, B, C, D, etc as R's attributes. Then the above three queries are re-written as follow:

```
Q1:  select tb, A, count(*) as cnt
     from R
     group by time/60 as tb, A
```

```
Q2:  select tb, B, count(*) as cnt
     from R
     group by time/60 as tb, B

Q3:  select tb, C, count(*) as cnt
     from R
     group by time/60 as tb, C
```

A straightforward method is to process each query separately using the above single aggregate query processing algorithm, so we maintain, in M_L, three hash tables for A, B, and C separately as shown in Figure 3.2. For each incoming record, we need to probe each hash table, and if there is a collision, some entry gets evicted to M_H.

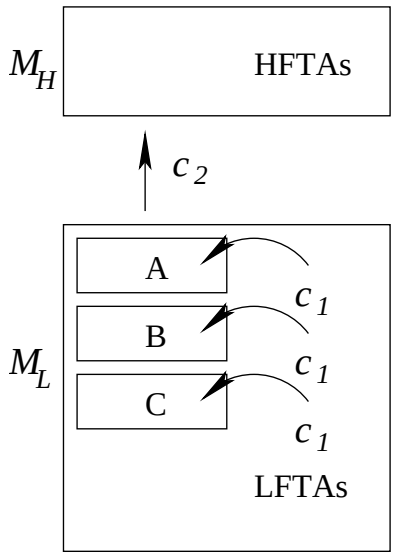


Figure 3.2: Multiple aggregations in Gigascope

3.2.2 Processing multiple aggregations using phantoms

Since we are processing multiple aggregate queries, we may be able to share the computation that is common to each one and thereby reduce the overall processing cost. For example, we can additionally maintain a hash table for the relation ABC in M_L as shown in Figure 3.3. If we have the counts of each group in ABC, we can derive the counts of each group of A, B and C, respectively, from it. The processing of the group-by queries on A, B and C runs as follows. When a new record arrives, we first hash on the combined attributes, ABC, and maintain the groups of ABC in its hash table. Suppose the new record hashes to entry bk_1 in the hash table ABC. If the new record belongs to the same group as the existing group in bk_1, then we simply increase the count by one in bk_1; otherwise, we evict the existing entry to the three hash tables for A, B and C and put the new group in bk_1 with count one. At this moment, the entry evicted from the hash table ABC is like a new record arriving at the hash tables A, B and C, respectively. Then we do similar things on each of hash tables A, B and C as we do in the single query case. Suppose the evicted entry from hash table ABC is $(v_a v_b v_c, cnt)$, where v_a, v_b, v_c are the values of A, B, and C in this entry respectively and cnt is the count of this entry. For hash table A, we hash v_a and suppose it's hashed to the entry bk_2. If the existing entry in bk_2 is also in group v_a, then we add the count of bk_2 by cnt; otherwise, we evict the existing entry in bk_2 to M_H and put the entry (v_a, cnt) in bk_2. For hash tables B and C, the similar process happens.

The intuition behind the processing with phantoms is that, when a new record arrives, instead of probing three hash tables A, B and C, we only probe the hash table ABC. We would delay the probes on A, B and C (we omit "hash tables" when the context is clear) until the point when an entry is evicted from ABC (that is, a collision happens in ABC). Again, because network data show much clusteredness,

that is, all packets in a flow have the same source/destination IP/port. Therefore, the likelihood of a collision is very low until many packets have been observed. Therefore, we may reduce the probing cost by maintaining ABC.

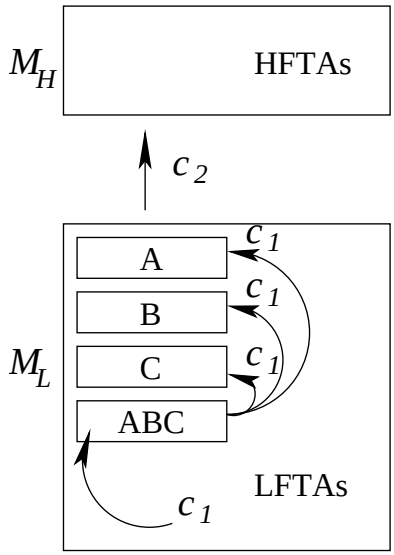


Figure 3.3: Multiple aggregations using phantoms

Since the aggregate queries of A, B and C are derived from ABC, we say that ABC *feeds* A, B and C. Although ABC is not of interest to the user, its maintenance could help reduce the overall cost. We call such a relation as a *phantom*. While for A, B and C, whose aggregate information is of users' interest, we call each of them as a *query*. Both *queries* and *phantoms* are called *relations*.

Now we examine Figure 3.3 to illustrate how the instantiation ("maintenance" and "instantiation" are used interchangeably throughout this chapter) of a phantom can benefit the total evaluation cost. To be fair, the total space used for the hash tables should be the same with or without the phantoms. So when we add the phantom ABC, the size of the hash tables for A, B and C need to be reduced. Suppose the total space allocated for the three queries is M. Assume that ABC has twice the number of groups as the number of groups of A (A has the same

number of groups as B or C). We should allocate more space to ABC so that ABC does not have a too high collision rate. Simply, we can allocate the space to their hash tables proportional to their number of groups (but actually this is not the optimal way; we will show the optimal scheme in Section 3.7). Then ABC has the space $2M/5$ and each of A, B, C has $M/5$. Without phantoms, we allocate $M/3$ to each hash table. In both cases, A, B and C have the same collision rate. Without the phantom, their collision rate is denoted x_1; with the phantom, their collision rate is denoted x'_1. Since the hash table size of A, B and C is smaller in the presence of the phantom, x'_1 should be larger than x_1. Let the collision rate of phantom ABC be x_2.

Consider the cost for processing n records. Without the phantom, we need to probe 3 hash tables for each incoming record, and there are $x_1 n$ evictions from each table. Therefore the total cost is:

$$E_1 = 3nc_1 + 3x_1nc_2 \tag{3.1}$$

With the phantom, we probe only ABC for each incoming record and there would be $x_2 n$ evictions. For each of these evictions, we probe A, B and C, and hence there are $x'_1 x_2 n$ evictions from each of them. The total cost is:

$$E_2 = nc_1 + 3x_2nc_1 + 3x'_1x_2nc_2 \tag{3.2}$$

Comparing Equations 3.1 and 3.2, we can get the difference of E_1 and E_2 as follows

$$E_1 - E_2 = [(2 - 3x_2)c_1 + 3(x_1 - x'_1x_2)c_2]n \tag{3.3}$$

If x_2 is small enough so that both $(2 - 3x_2)$ and $(x_1 - x'_1x_2)$ are larger than 0, then

E_2 will be smaller than E_1, and therefore instantiation of the phantom benefits the total cost. If x_2 is not small enough so that one of $(2 - 3x_2)$ and $(x_1 - x_1'x_2)$ is larger than 0 but the other is less than 0, then $E_1 - E_2$ depends on the relationship of c_1 and c_2. If x_2 is so large that both $(2 - 3x_2)$ and $(x_1 - x_1'x_2)$ are less than 0, then the instantiation of the phantom increases the cost and therefore we should not instantiate it.

3.2.3 Choice of phantoms

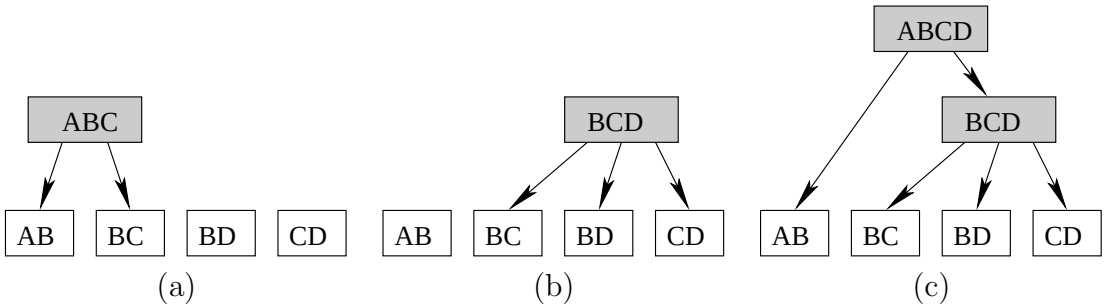


Figure 3.4: Choices of phantoms

In the above example, we have only considered one phantom. In fact, we can have many phantoms and multiple levels of phantoms. Again, consider stream relation R with four attributes A, B, C, D. Suppose the queries are AB, BC, BD and CD. We could instantiate phantom ABC, which feeds AB and BC as shown in Figure 3.4(a) (a shaded box is a phantom and a non-shaded box is a query); or we could instantiate phantom BCD, which feeds BC, BD and CD as shown in Figure 3.4(b); or we could instantiate BCD and ABCD, where ABCD feeds AB and BCD as shown in Figure 3.4(c). We only list three choices here, although there are many other possibilities.

It is easy to prove that a phantom that feeds less than two relations is never beneficial. So by combining two or more queries, we can obtain all possible phan-

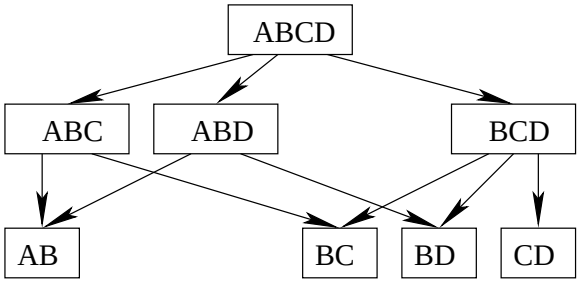


Figure 3.5: Feeding graph for the relations

toms and plot them in a *relation feeding graph* as in Figure 3.5. Each node in the graph is a relation and each directed edge shows a *feed* relationship between two nodes, that is, the parent feeds the child. Note that this feed relationship can be "short circuited", that is, a node can be directly fed by any of its ancestors in the graph. For example, AB could be fed directly by ABCD without having ABC or ABD maintained.

Given the relation feeding graph, and a set of user queries that are maintained in the LFTA, one optimization problem is to identify the phantoms that we should instantiate to minimize the cost. The exhaustive method is obvious, that is, we try all possible combinations of the phantoms and calculate the cost (the same way as in Section 3.2.2) of each combination. Then we choose the one with the minimum cost. However, the exhaustive method is too expensive, especially for data stream systems where a fast response is essential.

In our example in Section 3.2.2, we assumed that each hash table has the same size for simplicity of exposition. However, given a set of phantoms and queries to instantiate in M_L, how does the allocation of space to each hash table affect the collision rate and hence the cost? Therefore, another optimization problem is that, given a set of relations to instantiate, how to allocate space to them so that the cost is minimized.

In summary, our cost optimization problem consists of two sub-optimization problems: how to choose phantoms and how to allocate space. We formulate the cost model for the multiple aggregation problem and propose a *cost greedy* algorithm to choose phantoms. The algorithm is "cost greedy" in the sense that it always chooses the phantom that reduces the cost most among the candidate phantoms. In addition, for a given set of relations to instantiate, we analyze which space allocation gives the minimum cost; in case the optimal space allocation cannot be calculated, we propose heuristics which can approximate the optimal solution very well.

3.3 Problem formulation

In this section, we formulate our cost model, and give a formal definition of our optimization problem. We present hardness results, motivating the greedy heuristic algorithms for identifying optimal configurations.

3.3.1 Terminology and notation

When we have chosen a set of phantoms to instantiate in the M_L, we call the set of maintained relations (i.e., the chosen phantoms and user queries) as a *configuration*. For example, Figure 3.4 shows three possible configurations for the example query set, Q1, Q2 and Q3 presented in Section 3.2.1. While the feeding graph is a DAG, a configuration is always a tree, consistent with the path structure of the feeding graph. If a relation in a configuration is directly fed by the stream, we call it a *raw relation*. For example, ABC, BD, CD are raw relations in Figure 3.4(a); and ABCD is the only raw relation in Figure 3.4(c). If a relation in a configuration has no child, then it is called a *leaf relation* or just a *leaf*. Due to two reasons: i) early

data reduction is most efficient in exploiting the clusteredness of the network data stream; ii) local probes in LFTA are much cheaper than data transfers between LFTA and HFTA, user queries are always maintained in the LFTA, therefore only queries are leaves. For all the configurations in Figure 3.4, queries AB, BC, BD and CD are the leaves. Note that raw relations and leaf relations need not be mutually exclusive. For example, BD and CD are both raw and leaf relations in Figure 3.4(a).

We next develop our cost model, which determines the total cost incurred during data stream processing of a configuration. We then formalize the optimization problem studied in this chapter. For ease of reference, we summarize the symbols used in this chapter in Table 3.1.

3.3.2 Cost model

Recall that aggregate queries usually include a specification of temporal epochs of interest. For example, in the query "for every destination IP, destination port and 5 minute interval, report the average packet length", the "5 minute interval" is the epoch of interest (a window in the tumbling window query model). During stream processing within an epoch (e.g., a specific 5 minute interval), the aggregate query hash tables need to be maintained, for each record in the stream. At the end of an epoch, all the hash tables of the user queries at the LFTA need to be evicted to the HFTA to complete the user query computations. Thus, there are two components to the cost: intra-epoch cost, and end-of-epoch cost. We discuss each of these next.

Intra-epoch cost

Let E_m be the maintenance cost of all the hash tables during an epoch T. It includes updating all hash tables for the raw relations when a new record in the stream is

Table 3.1: Symbols

Symbol	Meaning
A_R	The set of ancestors of relation R
bk	A bucket in a hash table
b	The number of bucket of the hash table
B_k	The number of buckets that k groups hash to
$\mathcal{B}_i$	Occupancy numbers
c	Some constant denoting certain cost
e	Per record cost
E_m	Intra-epoch cost
E_u	End-of-epoch cost
f	Number of relations a phantom feed (the fanout)
$\mathcal{F}_R$	The number of tuples fed to relation R
g_i	The number of groups of relation i appearing in an epoch
I	A configuration, which is a set of relations to be maintained
k	The number of groups hashing to a hash table bucket
l_a	Average length of network flows
M	Memory constraint in LFTA
M_R	The size of the hash table for relation R
n_T	The number of tuples observed in an epoch
n_Q	The number of queries
Q	A query
R	A relation
S	A set
T	The time range of an epoch
W	The set of all raw relations
x	Collision rate
ϕ	A user defined ratio used in space allocation of the GS algorithm (see Section 3.5.1)

processed. If (and only if) there is collision in hash tables for the raw relations, the hash tables of the relations they feed are updated. This process recurses until the hash tables for the leaf level. A *probe* is defined as the process of checking a {group, count} pair against a hash table, and then either increasing the count of the entry or setting a new entry in the hash table due to collision. Every probe has the cost of c_1.

If there are collisions in the hash tables for the leaf (user) queries, evictions to the HFTAs are incurred, each with the cost of c_2. Therefore, the total maintenance

cost is

$$E_m = \sum_{R \in I} \mathcal{F}_R c_1 + \sum_{R \in L} \mathcal{F}_R x_R c_2 \tag{3.4}$$

where I is a configuration, L is the set of all leaves in I, $\mathcal{F}_R$ is the number of tuples fed to relation R during epoch T, and x_R is the collision rate of the hash table for R. $\mathcal{F}_R$ is derived as follows.

$$\mathcal{F}_R = \begin{cases} n_T & \text{if } R \in W \\ \mathcal{F}_p x_a & \text{else} \end{cases} \tag{3.5}$$

where W is the set of all raw relations, n_T is the number of tuples observed in T, $\mathcal{F}_p$ is the number of tuples fed to the parent of R in I, and x_a is the collision rate of the hash table for the parent of R in I. If we further define $\mathcal{F}_p = n_T$ and $x_a = 1$, when R is a raw relation, Equation 3.4 can be rewritten as follows.

$$E_m = \left[\sum_{R \in I} \Big(\prod_{R' \in A_R} x_{R'} \Big) c_1 + \sum_{R \in L} \Big(\prod_{R' \in A_R} x_{R'} \Big) x_R c_2 \right] n_T \tag{3.6}$$

where A_R is the set of all ancestors of R in I.

n_T is determined by the data stream and is not affected by the configuration. Hence, the per record cost is:

$$e_m = \sum_{R \in I} \Big(\prod_{R' \in A_R} x_{R'} \Big) c_1 + \sum_{R \in L} \Big(\prod_{R' \in A_R} x_{R'} \Big) x_R c_2 \tag{3.7}$$

where c_1 and c_2 are constants determined by the LFTA/HFTA architecture of the DSMS. Therefore, the cost is only affected by the feeding relationship and collision rates of the hash tables.

End-of-epoch cost

Denote the update cost at the end of epoch T as E_u. It includes the cost of the following operations. From the raw level to the leaf level of the feeding graph of the configuration, we scan each hash table and propagate each item in the hash table to hash tables of the lower level relations they feed. Finally, we scan the leaf level hash table and evict each item in it to the HFTA, M_H. Using an analysis similar to the one for intra-epoch costs, taking the possibilities of collisions during this phase into account, the end-of-epoch cost E_u can be expressed as follows.

$$\begin{aligned} E_u = &\textstyle\sum_{R\in I, R\notin W}[\sum_{R'\in A_R}(M_{R'} * \prod_{R''\in A_{R'}\cup R', R''\notin W} x_{R''})]c_1 + \\ &\textstyle\sum_{R\in L}[M_R + \sum_{R'\in A_R}(M_{R'} * \prod_{R''\in A_{R'}\cup R', R''\notin W} x_{R''})]c_2 \end{aligned} \tag{3.8}$$

where M_R is the size of the hash table of relation R, and W is the set of all raw relations.

3.3.3 Our problem

Intuitively, the lower the average per-record intra-epoch cost, the lower is the load at the LFTA, increasing the likelihood that records in the stream are not dropped during query processing. We also want to ensure that the total cost of the end-of-epoch processing is manageable. This leads to the *multiple aggregation* (MA) optimization problem studied for the two-level LFTA/HFTA architecture in this chapter.

Consider a set of aggregate queries over a data stream that differ only in their group-by attributes[1], $S_Q = \{Q_1, Q_2, ..., Q_{n_Q}\}$, and memory limit M in M_L. Determine the configuration I, of relations in the feeding graph of S_Q to instantiate

[1]Our current work only considers group-by queries that differ in their group-by attributes, that is, they must have the same time window size. Optimizing group-by queries with different group-by attributes with different time window sizes is one direction of our future work.

in M_L and also the allocation of the available memory M to the hash tables or the relations so that the per-record intra-epoch cost (Equation 3.7) for answering all the queries is minimized, subject to the end-of-epoch cost being less than peak load cost E_p.

Our cost optimization problem consists of two sub-problems: how to choose phantoms and how to allocate space. For any given configuration, there exists a space allocation that has the lowest per-record intra-epoch cost (we simply say *cost* in the sequel when the meaning is clear from the context). In Section 3.7, we would show how to allocate the space in order to achieve the minimum cost. For now, we first assume that we have a function to output the minimum cost given a configuration as the input, and we give the hardness result of the phantom choosing problem below.

Hardness of the problem

Phantom choosing is a minimization problem. To study its hardness, we first recast it as a decision problem as follows:

PHANTOMCHOOSING=[*Given a query set* $S_Q = \{Q_1, Q_2, ..., Q_{n_Q}\}$, *a function that returns the cost of any configuration, does there exist a configuration of cost less than or equal to* c_Q ?]

The hardness of the problem is given by the following theorem.

***Theorem* 1**

The phantom choosing problem is NP-complete.

Proof We first show that PHANTOMCHOOSING $\in$ NP. For a given configuration, we simply calculate its cost and compare the cost with c_Q, which takes constant time. Therefore PHANTOMCHOOSING $\in$ NP.

Next we show that PHANTOMCHOOSING is NP-hard by reducing the set cover problem to it. The set cover problem, defined below, has been proven to be NP-complete [144].

Given a universe U of n_Q elements, a collection of subsets of U, $S = \{S_1, S_2, ..., S_{n_S}\}$, and a cost function that turns the cost (a positive number) of any subset in S, find a minimum cost subcollection of S that covers all elements of U.

The decision problem version of set cover is:

SETCOVER=[*Given U, S and the cost function as above, does there exist a subcollection of S that covers all elements of U and has the cost less than or equal to c_Q ?*].

In the following, we construct a PHANTOMCHOOSING instance *pc* and form a mapping from SETCOVER to *pc*. Given any U of SETCOVER, we define the queries of *pc* as the elements of U. A subcollection of S consists of a number of elements of S, denoted by $S' = \{S_{i1}, S_{i2}, ..., S_{ij}\}$, where $i1, i2, ..., ij \in \{1, 2, ..., n_S\}$. If we view each subset as a relation in PHANTOMCHOOSING, then each subcollection corresponds to configuration in PHANTOMCHOOSING. We map each subcollection S' that covers all elements of U to its corresponding configuration and define the cost of the configuration as the sum of the cost of all the subsets in S'. The cost of the configuration corresponding to any other set in the power set of U than those in S is defined as $+\infty$. There are at most 2^{n_S} subcollections of S. Since n_S is a constant, the mapping algorithm can be run in constant time. For any instance of SETCOVER, if there exists a subcollection of S that covers all elements of U and has the cost of c_Q, then its corresponding configuration for PHANTOMCHOOSING also has the cost of c_Q. Conversely, if there is a configuration for PHANTOMCHOOSING having the cost of c_Q, this configuration must correspond to a subcollection of S, since all other sets in the power set of U has the

cost of $+\infty$. Therefore SETCOVER is reduced to PHANTOMCHOOSING, and PHANTOMCHOOSING is NP-hard.

PHANTOMCHOOSING $\in$ NP and $\in$ NP-hard, therefore, PHANTOMCHOOSING is NP-complete. $\square$

To help understand the mapping algorithm, an example is given as follows. In SETCOVER, let U be $\{A, B, C, D, E\}$, S be $\{\{A\}, \{B, C\}, \{C, D\}, \{A, D, E\}, \{E\}\}$ and the costs of the elements be $\{4, 8, 6, 10, 3\}$, each corresponding to the subset at the same position in S. One subcollection that covers all elements of U is $\{\{A\}, \{B, C\}, \{C, D\}, \{E\}\}$, hence we define the cost of the configuration $\{A, BC, CD, E\}$ as $(4+8+6+3) = 21$. Obviously this configuration does not have the minimum cost since both BC and CD are feeding C, but this is still a valid configuration for PHANTOMCHOOSING. Another subcollection that covers all elements of U is $\{\{B, C\}, \{A, D, E\}\}$, hence we define the cost of the configuration $\{BC, ADE\}$ as $(8+10) = 18$. This is actually the configuration with minimum cost. We can still find other subcollections that cover all elements of U by adding some elements in S to the above two subcollections, such as $\{\{A\}, \{B, C\}, \{A, D, E\}\}$, but they have larger cost. Except the above mentioned configurations, any other configuration that does not correspond to a subcollection of S, such as $\{AB, CDE\}$, would be defined to have the cost of $+\infty$.

In our problem setting, we assume that user queries are always maintained in the LFTA due to two reasons: i) early data reduction is most efficient in exploiting the clusteredness of the network data stream; ii) local probes in LFTA are much cheaper than data transfers between LFTA and HFTA. This variation of the phantom choosing problem is still NP-complete. The proof is almost the same as above except a small change in the mapping algorithm. In every PHANTOMCHOOSING configuration we map to, all the elements of U must appear. We would change the

mapping algorithm as follows. We first map a subcollection of S to a configuration for PHANTOMCHOOSING as described in the proof of theorem 1. If any element of U is absent from the resultant configuration, we would add it to the configuration while keeping the cost unchanged. Still use the example in the previous paragraph, when we obtain the configuration $\{BC, ADE\}$, we would add all the queries (which correspond to all the elements of U) to it and get the configuration $\{A, B, C, D, E, BC, ADE\}$. The cost of this configuration is $(8 + 10) = 18$. However, a problem arises here because $\{\{A\}, \{B, C\}, \{A, D, E\}\}$ is also mapped to $\{A, B, C, D, E, BC, ADE\}$, which means we should define the cost of this configuration as $(4 + 8 + 10) = 22$. To solve this problem, we define the cost of a configuration according to the subcollection of S with the least elements of U in it (which is actually the one with the minimum cost among those mapped to the same configuration for PHANTOMCHOOSING). Therefore in the above example, both $\{BC, ADE\}$ and $\{\{A\}, \{B, C\}, \{A, D, E\}\}$ are mapped to the configuration $\{A, B, C, D, E, BC, ADE\}$, which has the cost of 18. Note that multiple inputs in SETCOVER being mapped to one input in PHANTOMCHOOSING still makes a valid reduction, as long as the mapped input produces the same answer. It is easy to see that the modified mapping algorithm still run in polynomial time.

Therefore, this variation of the phantom choosing problem that must instantiate all queries in the LFTA (which is the one we study in this thesis) is also NP-complete.

Since the phantom choosing problem is NP-complete, we would use heuristics, specifically a greedy algorithm in this thesis, to solve it. We would describe cost greedy algorithms to choose phantoms, based on the cost model presented in Section 3.5. The cost model critically depends on the collision rate model, which we discuss in detail in Section 3.6. For a given configuration, we analyze which space allocation

gives the minimum cost, in Section 3.7.

3.4 Synopsis of our proposal

The multi-aggregation (MA) problem has similarities to the view materialization (VM) problem [83]. They both have a feeding graph consisting of nodes some of which can feed some others, and we need to choose some of them to instantiate. So one possibility is to adapt the greedy algorithm developed for VM to MA. However, there are two differences between these two problems. First, instantiation of any of the views in VM will add to the benefit; while in MA, instantiation of a phantom is not always beneficial. Second, the space needed for instantiation of a view is fixed but the hash table size is flexible. Therefore, in order to adapt the VM greedy algorithm, we need to have a space allocation scheme that fixes the hash table size and at the same time guarantees a low collision rate of the hash table to make each maintained phantom beneficial. This adapted approach, called *greedy by increasing space*, is discuss in Section 3.5.1. The greedy-by-increasing-space approach has several drawbacks. First, it cannot apply the optimal space allocation scheme as we would investigate in Section 3.6. Second, it depends on a good choice of a parameter (ϕ as defined below) to yield good performance, but a good value for the parameter is hard to determine in practice. Therefore we propose a new approach, called *greedy by increasing collision rates*, in Section 3.5.2. In this approach we always use the whole available space and add the phantoms one by one in a greedy fashion to the configuration. Initially, when there is a small number of relations in the configuration, the collision rates are low and hence adding phantoms reduce the cost. As more and more phantoms are added, the hash table sizes become smaller and collision rates become higher. Until the point that

no phantom candidate can be added with a reduction to the cost, the algorithm stops. This strategy does not need any tuning parameter, and also an optimal (or heuristic approximating the optimal) space allocation scheme can be applied. However, the greedy-by-increasing-collision-rates approach requires an accurate model to estimate the collision rate, which is investigated closely in Section 3.6.

3.5 Phantom choosing

We present the two phantom choosing strategies, both of which are greedy algorithms in this section. The greedy-by-increasing-space approach is adapted from the algorithm for the view materialization (VM) problem [83]. The greedy-by-increasing-collision-rates approach is our proposed algorithm.

3.5.1 Greedy by increasing space

To adapt the greedy algorithm used in the view selection problem [83], we need to have a space allocation scheme that fixes the hash table size and at the same time guarantees a low collision rate of the hash table to make each maintained phantom beneficial. Generally, the more space allocated to a hash table, the lower the collision rate. On the other hand, the more groups a relation has for a fixed sized hash table, the higher is the collision rate. Let g be the number of groups of a relation and b be the number of buckets in a hash table. A straightforward way of allocating hash table space to a relation is in proportion to the number of groups in the table so as to make all the hash tables have similar collision rates. Specifically, we can allocate space ϕg for a relation with g groups, where ϕ is a constant chosen by the user and can be tuned. We set it large so that the hash table is guaranteed to have a low collision rate. We will develop a model to estimate collision rate in

Section 3.6. We can then have a better sense of what value of ϕ might be good according to the analysis there.

In order to allocate space to the hash tables of different relations, we need to know the number of groups of the relations. As we process the queries in tumbling time windows, and the time window size is small (usually one minute). We assume that the data distribution does not change greatly from the last time window to the current one. Therefore, we maintain the number of groups of all the relations in the feeding graph (we can use sampling or sketch techniques described in Section 2.2.1 to efficiently maintain them) for every time window and use this information at the beginning of the next time window to decide which phantoms to maintain and how to allocate space.

Algorithm **GS**
1 choose a ϕ value;
2 $I = S_Q$; $S_C = \{R \in relations\ in\ the\ feeding\ graph \wedge R \notin S_Q\}$;
3 $\mathcal{M} = M - \sum_{R \in S_Q} \phi R.g$
4 $R_m = 1$
5 while $\mathcal{M} > 0$ && $S_C \neq \emptyset$ && $R_m \neq NULL$
6 $\beta = -\infty$, $R_m = NULL$
7 for each $R \in S_C$
8 if $\mathcal{M} > \phi R.g$ && $(cost(I) - cost(R \cup I))/(\phi R.g) > \beta$
9 $\beta = (cost(I) - cost(R \cup I))/(\phi R.g)$
10 $R_m = R$
11 if $R_m \neq NULL$
12 $I = R_m \cup I$
13 $S_C = S_C - R_m$
14 $\mathcal{M} = \mathcal{M} - \phi R.g$
15 return I
End GS

Figure 3.6: Algorithm **GS**

Figure 3.6 shows the greedy algorithm by increasing space (Algorithm GS) which goes as follows. Initially, the configuration I only contains the queries,

which must be maintained in the LFTA by our problem assumption. Then, for each candidate phantom in the candidate set S_C, we calculate its benefit (which is the difference between the costs of the configurations with and with out this phantom) according to the cost model. Then we divide the benefit by the relation R's number of groups $R.g$, to get the benefit per unit space for R. In the algorithm, $\mathcal{M}$ denotes the size of the remaining memory and β denotes the benefit per unit space. We choose the phantom with the largest benefit per unit space as the first phantom to be added to I. For the other phantoms, this process is iterated. The process ends when the space is exhausted, or all phantoms are maintained. When the algorithm terminates, I contains the relations we should instantiate in the LFTA.

This approach has two drawbacks: (1) ϕ needs to be tuned to find the best performance. A bad choice can result in suboptimal performance. In practice, a good choice is very hard to achieve. (2) By allocating space to a relation proportional to the number of its groups, we make the collision rates of all the relations the same. As we shall show later, this is not a good strategy.

3.5.2 Greedy by increasing collision rates

Here, we propose a different greedy algorithm for allocating space to hash tables of the relations in the LFTA. Instead of allocating a fixed amount of space to each phantom progressively, we always allocate *all available space* to the current configuration (how to allocate space among relations in a configuration is analyzed in Section 3.7). So as each new phantom is added to a configuration, what changes is not the total space used, but the collision rate of each table. Since the more the number of groups mapped to a fixed space, the higher the collision rate, the collision rate would increase as new phantoms are maintained.

```
Algorithm GC
1   I = S_Q; S_C = {R ∈ relations in the feeding graph ∧ R ∉ S_Q}
2   β = 1
3   while β > 0 && S_C ≠ ∅
4       β = 0, R_m = NULL
5       for each R ∈ S_C
6           if cost(I) − cost(R ∪ I) > β
7               β = cost(I) − cost(R ∪ I)
8               R_m = R
9       if β > 0
10          I = R_m ∪ I
11          S_C = S_C − R_m
12  return I
End GC
```

Figure 3.7: Algorithm **GC**

Figure 3.7 shows the greedy algorithm by increasing collision rates (Algorithm GC) which goes as follows. At first, the configuration I only includes all the queries. Then, for each candidate phantom in the candidate set S_C, we calculate its benefit (which is the difference between the costs of the configurations with and with out this phantom) according to the cost model, but note that different from algorithm GS which allocate $\phi R.g$ space, here we allocate space according to the scheme devised in Section 3.7. Among all the candidate phantoms, we choose the one with the largest benefit to be added to I. Note that here we are not using benefit per unit space as the phantom choosing criteria because the effect of the space used by the phantom is already reflected by the cost. (If the phantom needs too much space, it would reduce the space other relations have and therefore has a negative effect on the overall cost.) After we choose the first phantom to instantiate (that is, added to I), we iterate the same process with the remaining candidate phantoms. The process ends when no remaining candidate phantom produce a positive benefit, or all phantoms are maintained. When the algorithm terminates, I contains the

relations we should instantiate in the LFTA.

A prerequisite of this algorithm is an accurate model to estimate the collision rates. We derive such a model in Section 3.6.

3.5.3 An Example

We use an example to show the difference of the two algorithms GS and GC. Suppose we have three queries, A, B and C. Possible phantoms for these queries are AB, AC, BC and ABC. The feeding graph for these relations are shown in Figure 3.8. The number of groups of each relation is written beside the relation. Suppose the total space we have is 1000 hash table buckets (we simply use buckets

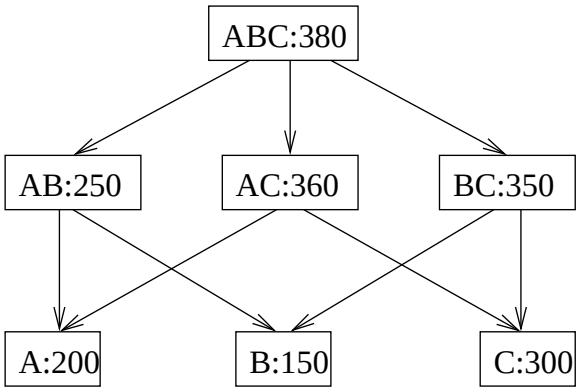


Figure 3.8: Feeding graph of the example

as the unit of space here) and we choose the ϕ value 1 for algorithm GS. The GS algorithm runs as follows. First all the queries are maintained, therefore 650 buckets are allocated to A, B and C. Then for all the phantoms, we check which ones may be maintained in the remaining space. The remaining space, 350 buckets, is only enough for relations AB or BC. For these two candidates, we compare their per unit space benefit. Specifically, we calculate the cost of the configuration which has only the queries, A, B and C according to Equation 3.7. Denote this cost by $cost_1$. We calculate the cost of the configuration which has the queries A, B, C and

phantom AB feeding A and B. Denote this cost by $cost_2$. Then we obtain the per unit space benefit of AB, $\beta_1 = (cost_2 - cost_1)/250$. Similarly we can obtain the per unit space benefit of BC, β_2. Suppose $\beta_1 > \beta_2$, so we add AB to our configuration and allocate 250 buckets to AB. The remaining space is 100 buckets, which is not enough for any of the rest relations in the feeding graph, AC, BC or ABC, therefore the algorithm GS stops. We allocate the rest 100 buckets to the relations in our current configuration proportionally to the space already allocated to them. A gets 22 more buckets; B gets 17 more buckets; C gets 33 more buckets; AB gets 28 more buckets. So the final configuration we obtain by the GS algorithm is A with 222 buckets, B with 167 buckets, C with 333 buckets and AB with 278 buckets.

The GC algorithm runs as follows. First all the queries are maintained. We allocate all the 1000 buckets to the three queries A, B and C using our space allocation scheme devised in Section 3.7 and we calculate the cost of the configuration by Equation 3.7. Denote this cost by $cost'_1$. For the rest relations in the feeding graph, AB, AC, BC and ABC, we will try to add each of them to the configuration and see how much cost is reduced. For example, suppose we add AB to the configuration and get A, B, C and AB feeding A and B. We allocate all the buckets to these relations using our space allocation scheme and calculate the cost of the configuration, $cost'_2$. Therefore we get the benefit of the phantom AB, which is $\beta'_1 = cost'_2 - cost'_1$. Suppose we add ABC to the configuration and get A, B, C and ABC feeding A, B and C. Then we calculate the benefit of ABC as the way described above. Similarly we calculate the benefit of AC and BC, respectively. Suppose ABC has the largest benefit among all the current candidates to be added to the current configuration, and the benefit is positive, then we actually add ABC to our configuration, which results the cost of $cost'_3$. Now the remaining relations of the feeding graph are AB, AC and BC. Again, we try to add them to the cur-

rent configuration and compare their benefits. Suppose AB has the largest benefit, which is positive, therefore we actually add AB to the configuration, which results in the cost of $cost'_4$. The current configuration becomes A, B, C, AB feeding A and B, and ABC feeding AB and C. Now the remaining relations of the feeding graph are AC and BC. Again, we try to add them to the current configuration and find out their benefits. This time, we find that their benefits are both negative, therefore the GC algorithm stops. The final configuration we obtain by the GS algorithm is A, B, C, AB feeding A and B, and ABC feeding AB and C. The space is allocated according to our space allocation scheme described in Section 3.7.

3.6 The collision rate model

In this section, we develop a model to estimate the collision rate. We assume that the hash function randomly hashes the data, so each hash value is equally possible for every record. We first consider uniformly distributed data, and subsequently consider when the data exhibits clusteredness.

3.6.1 Randomly distributed data

Let g be the number of groups of a relation and b the number of buckets in the hash table. If k groups hash to a bucket, we say that this bucket has k groups. Let B_k be the number of buckets having k groups. If the records in the stream are uniformly distributed, each group has the same expected number of records, denoted by n_{rg}. So $n_{rg}k$ records will go to a bucket having k groups. Under the random hash assumption, the collision rate in this bucket is $(1 - 1/k)$. Therefore $n_{rg}k(1 - 1/k)$ collisions happen in this bucket. The overall collision rate is obtained by summing all the collisions and then dividing by the total number of records. Therefore, we

have collision rate

$$x = \frac{\sum_{k=2}^{g} B_k n_{rg} k(1-1/k)}{g n_{rg}} = \frac{\sum_{k=2}^{g} B_k (k-1)}{g} \tag{3.9}$$

k begins from 2 because when 0 or 1 group hashes to a bucket, no collision happens. In order to calculate it, we still need to know B_k. This problem belongs to a class of problems called the *occupancy problem*.

As we know, the expectation of k for each bucket is g/b [57]. A rough estimation of B_k based on expectation would be

$$B_k = \begin{cases} b & \text{k=g/b} \\ 0 & \text{k}\neq \text{ g/b} \end{cases}$$

Substituting this for B_k in Equation 3.9, we get

$$x = 1 - b/g \tag{3.10}$$

However, in a real random process, the probability of each bucket having the same number of groups is small. In [63] (Chapter II.5), an example when $g = b = 7$ is given to calculate the probability of different distributions of groups. It is shown that probability of each of the 7 buckets having exactly 1 group is 0.006120, which makes it extremely unlikely. Therefore, we need a more accurate estimation of B_k based on probability.

B_k can be estimated more accurately by the "multinomial allocations" model [21] (Chapter 6.2). We sketch the derivation below. The probability of k groups

out of g hashed to a given bucket is

$$\binom{g}{k}(1/b)^k(1-1/b)^{g-k} \tag{3.11}$$

Note this holds for any bucket, which means each bucket has the chance of Equation 3.11 to have k groups. If we assume that all b buckets are independent of each other, then statistically there are

$$b\binom{g}{k}(1/b)^k(1-1/b)^{g-k} \tag{3.12}$$

buckets each of which has k groups. Substitute Equation 3.12 for B_k in Equation 3.9 we have

$$x = \frac{b\sum_{k=2}^{g}\binom{g}{k}(1/b)^k(1-1/b)^{g-k}(k-1)}{g} \tag{3.13}$$

Our experiments on both synthetic and real data show that the actual distribution of B_k matches Equation 3.13 well, even though the buckets are not completely independent (they satisfy the equation $\sum_{k=1}^{b} B_k = b$).

Thanks to Ted Johnson[2], who points out that Equation 3.13 may be further simplified. We show the simplification of the equation as follows.

$$\begin{aligned} x &= \frac{b\sum_{k=2}^{g}\binom{g}{k}(1/b)^k(1-1/b)^{g-k}(k-1)}{g} \\ &= \frac{b}{g}[\sum_{k=2}^{g}\binom{g}{k}(1/b)^k(1-1/b)^{g-k}k - \sum_{k=2}^{g}\binom{g}{k}(1/b)^k(1-1/b)^{g-k}] \\ &= \frac{b}{g}(Y-Z) \end{aligned} \tag{3.14}$$

[2]AT&T Labs-research

where

$$Y = \sum_{k=2}^{g} \binom{g}{k} (1/b)^k (1-1/b)^{g-k} k \tag{3.15}$$

$$Z = \sum_{k=2}^{g} \binom{g}{k} (1/b)^k (1-1/b)^{g-k} \tag{3.16}$$

let $q = 1/b$, then

$$\begin{aligned}
Y &= \sum_{k=2}^{g} \frac{g!}{(g-k)!k!} q^k (1-q)^{g-k} k \\
&= \sum_{k=2}^{g} \frac{(g-1)!g}{(g-k)!(k-1)!} q^{k-1} (1-q)^{g-1-(k-1)} q \\
&\overset{j=k-1}{=} \sum_{j=1}^{g-1} g \binom{g-1}{j} q^j (1-q)^{g-1-j} q \\
&= gq[\sum_{j=0}^{g-1} \binom{g-1}{j} q^j (1-q)^{g-1-j} - (1-q)^{g-1}] \\
&= gq[(q+1-q)^{g-1} - (1-q)^{g-1}] \\
&= gq[1-(1-q)^{g-1}]
\end{aligned} \tag{3.17}$$

$$\begin{aligned}
Z &= \sum_{k=2}^{g} \binom{g}{k} (q)^k (1-q)^{g-k} \\
&= \sum_{k=0}^{g} \binom{g}{k} (q)^k (1-q)^{g-k} - (1-q)^g - \binom{g}{1} q(1-q)^{g-1} \\
&= (q+1-q)^g - (1-q)^g - gq(1-q)^{g-1} \\
&= 1-(1-q)^g - gq(1-q)^{g-1}
\end{aligned} \tag{3.18}$$

therefore,

$$\begin{aligned} x &= \frac{b}{g}(Y - Z) \\ &= \frac{b}{g}(gq[1-(1-q)^{g-1}] - [1-(1-q)^g - gq(1-q)^{g-1}]) \\ &= 1 - \frac{b}{g} + \frac{b}{g}(1-\frac{1}{b})^g \end{aligned} \tag{3.19}$$

3.6.2 Validation of collision rate model

We have measured experimentally the collision rates on both synthetic random data sets and real data sets. The results on the synthetic and real data sets are shown in Figure 3.9 and Figure 3.10, respectively.

The real data sets are extracted from the netflow data set as described in Section 3.8.1. We have assumed random data distribution for the above analysis, while the netflow data set has a lot of clusteredness due to multiple packets in a flow. In order to validate our analysis using the real data, we grouped all packets of a flow into a single record, eliminating the effect of clusteredness. (We consider clusterness in a later subsection.) After eliminating clusteredness of the data, we extracted 4 data sets which have 1, 2, 3 and 4 attributes respectively. The number of groups in these data sets are 552, 1846, 2117, 2837 respectively. For the synthetic data sets, we generated data sets which have 500, 1000, and 2000 groups respectively. All multi-attribute random data sets have the same data distribution and hence the collision rates are the same as a one-attribute random data set. Therefore, we do not specify number of attributes in Figure 3.9.

The "rough model" curve is plotted according to Equation 3.10 and the "precise model" curve is plotted according to Equation 3.19. Collision rates of the real data match the precise model very well. In all the observed collision rates, more

than 95% of the experimental results have less than 5% difference from the precise model. The rough model differs greatly from the precise model when g/b is small but becomes similar as g/b gets large. The reason is that the rough model only captures the expected case, which occurs with low probability. When g becomes larger, the behavior gets closer to the average case, therefore the rough model gets close to the precise model.

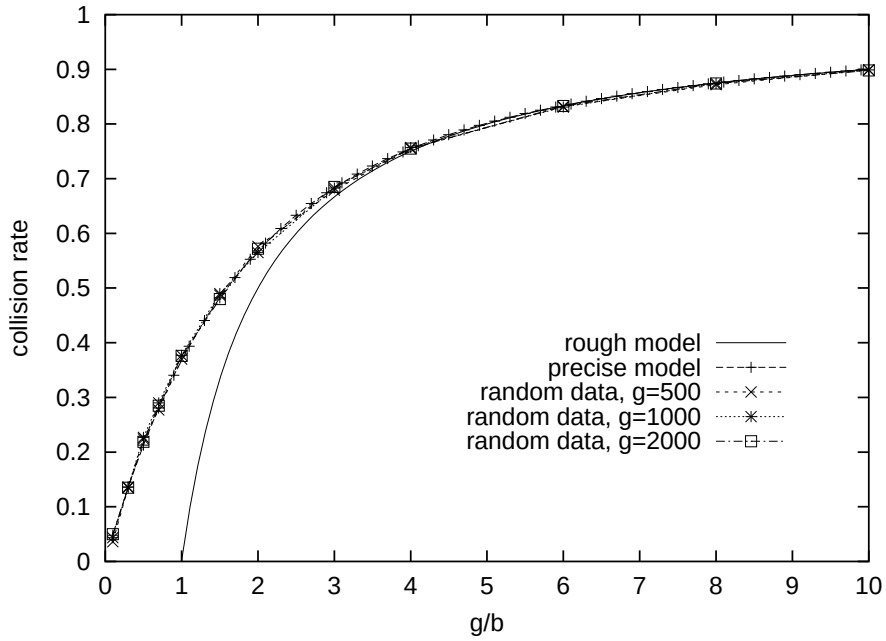


Figure 3.9: Collision rates of random data

For the rough model, the collision rate is only dependent on the ratio of g to b as we can see from Equation 3.10. We will show in Section **??** that the precise model is also dependent on g/b, though the function is different.

3.6.3 Clustered data

The above analysis was for randomly distributed data. However, real data streams, especially the packets in netflow data (which have exactly the same values for attributes such as source/destination IP/port), are clustered. Although packets

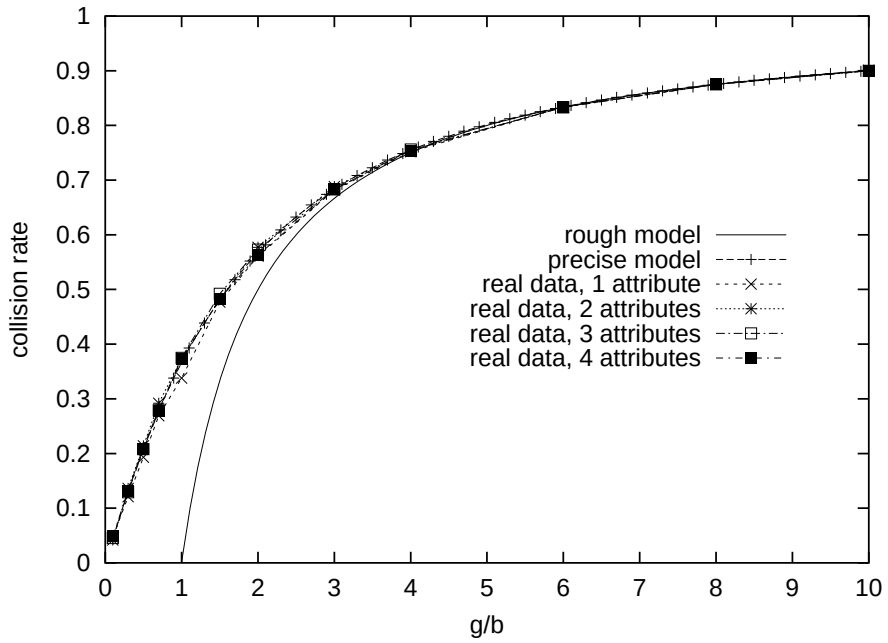


Figure 3.10: Collision rates of real data

from different flows are interleaved with each other in the stream, the likelihood of these interleaved flows hashing to the same bucket is very small. Therefore we can think of the packets in a flow going through a bucket without any collision until the end of the flow. To analyze collision rate for such clustered distributions, we should consider what happens at the per flow level. If we think of each flow as one record, then we can use the same formula as in the random distribution (Equation 3.9) to calculate the total number of collisions as follows.

$$n_c = \sum_{k=2}^{g} B_k n_{fg} k(1 - 1/k) \tag{3.20}$$

where n_{fg} is the number of flows in each group; B_k is still calculated by Equation 3.12. To obtain the collision rate, we divide n_c by the total number of records, $gn_{fg}l_a$, where l_a is the average length of all the flows. Then we have the collision rate for the data with a clustered distribution as follows.

$$x = \frac{b\sum_{k=2}^{g}\binom{g}{k}(1/b)^k(1-1/b)^{g-k}(k-1)}{gl_a} = (1-\frac{b}{g}+\frac{b}{g}(1-\frac{1}{b})^g)/l_a \qquad (3.21)$$

We can see that the difference of the collision rate on data with clusteredness from that of the random data is a linear relationship over average flow length l_a. We can view the collision rate of the random data as a special case of clustered data with $l_a = 1$. The average flow length can be computed by maintaining the number of times hash table bucket entries are updated before being evicted.

3.6.4 Approximating the low collision rate part

We can plot the collision rate as a function of g/b, which is shown in in Figure 3.11. According to our previous analysis, the hash table must have a low collision rate if we want to benefit from maintaining phantoms. Therefore, we examine the low collision rate part of this curve closely. A zoom in of the collision rate curve when collision rate is smaller than 0.4 as well as a linear regression of this part is shown in Figure 3.12.

We observe that this part of the curve is almost a straight line and the linear regression achieves an average error of 5%. The linear function for this part is

$$x = 0.0267 + 0.354 \cdot (g/b) \qquad (3.22)$$

Expressing this part of the collision rate linearly is important for the space allocation analysis as we will see in Section 3.7. In addition, since we now know how the collision rate is determined, we can suggest values of ϕ to use in the adapted greedy algorithm (by increasing space) of Section 3.5.1. For example, $\phi = 1$ could

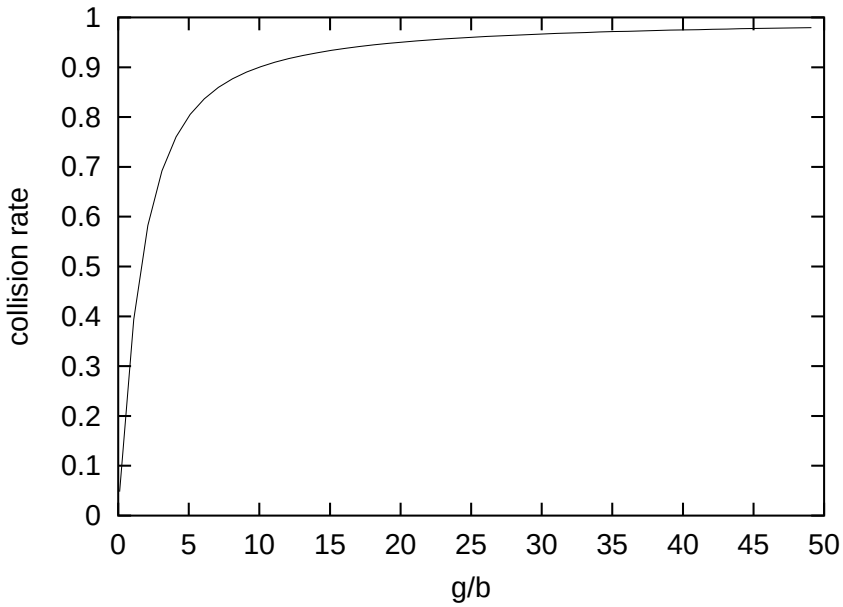


Figure 3.11: The collision rate curve

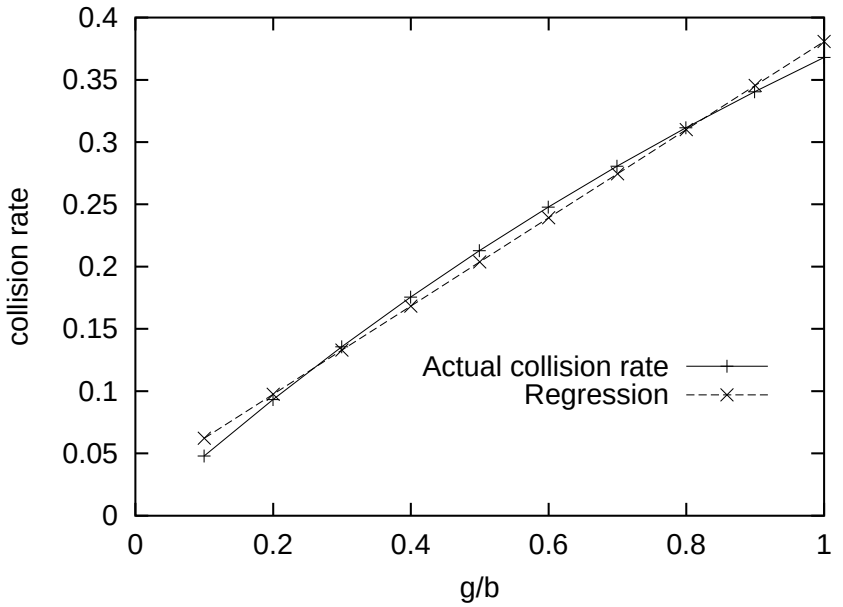


Figure 3.12: The low collision rate part

be a good choice, since it corresponds to a collision rate of about 0.37.

3.7 Space allocation

In this section, we consider the problem of space allocation, that is, given a configuration of certain relations (phantoms and queries) to be maintained, how to allocate the available space M to their hash tables so that the overall cost is minimized. We start with a simple two-level configuration in Section 3.7.1, and identify the difficulties in analyzing more complex configurations. Heuristics for space allocation are discussed in Section 3.7.4.

3.7.1 A case of two levels

We first study the case when there is only one phantom R_0 and it feeds all f queries, R_1, R_2, ..., R_f. Let x_0 be the collision rate of the phantom, x_1, x_2, ..., x_f be the collision rate of the queries. In order to benefit from maintaining a phantom, its collision rate must be low, therefore we only care about the low collision rate part of the collision rate curve. According to Section 3.6.4, this part of the curve can be expressed as a linear function $x = \alpha + \mu g/b$, where α=0.0267 and μ=0.354.[3] Since α is small, here we make a further approximation to let $x = \mu g/b$. We will discuss later how the results are affected when we consider α. Given the approximation, $x_i = \mu g_i/b_i$, $i = 0, 1, ..., f$. The total size is M, so $M = \sum_{i=0}^{f} b_i$. The cost of this configuration is

[3]Actually, even if the collision rate for the optimal allocation is a little higher than 0.4, we can still use linear regression for that part. The values of α and μ would be a little different, but experiments show that small variation in their values does not affect the result much.

$$
\begin{aligned}
e &= c_1 + fx_0c_1 + x_0\sum_{i=1}^{f} x_ic_2 \\
&= f\mu\frac{g_0}{b_0}c_1 + \mu\frac{g_0}{b_0}\sum_{i=1}^{f}\mu\frac{g_i}{b_i}c_2 + c_1 \\
&= \mu\frac{g_0}{b_0}(fc_1 + \mu c_2\sum_{i=1}^{f}\frac{g_i}{b_i}) + c_1 \\
&= \frac{\mu g_0}{M - \sum_{i=1}^{f} b_i}(fc_1 + \mu c_2\sum_{i=1}^{f}\frac{g_i}{b_i}) + c_1
\end{aligned}
\tag{3.23}
$$

e is a function of multiple variables, $b_1, b_2, ..., b_f$. To find out the minimum, we equate the partial derivatives of e to 0. In the following, we calculate the partial derivative of e over b_i, $i = 1, 2, ...f$.

$$
\frac{\partial e}{\partial b_i} = \frac{\mu g_0}{(M - \sum_{i=1}^{f} b_i)^2}(fc_1 + \mu c_2\sum_{i=1}^{f}\frac{g_i}{b_i}) + \frac{\mu g_0}{M - \sum_{i=1}^{f} b_i}\mu c_2 g_i(-\frac{1}{b_i^2})
$$

Let $\frac{\partial e}{\partial b_i} = 0$, then

$$
\frac{\mu g_0}{M - \sum_{i=1}^{f} b_i}\left(\frac{fc_1 + \mu c_2\sum_{i=1}^{f}\frac{g_i}{b_i}}{M - \sum_{i=1}^{f} b_i} - \frac{\mu c_2 g_i}{b_i^2}\right) = 0
$$

$\frac{\mu g_0}{M - \sum_{i=1}^{f} b_i}$ is non-zero, so

$$
\frac{fc_1 + \mu c_2\sum_{i=1}^{f}\frac{g_i}{b_i}}{M - \sum_{i=1}^{f} b_i} = \frac{\mu c_2 g_i}{b_i^2}
\tag{3.24}
$$

for $i = 1, 2, ..., f$.

Observe that left hand side of the equation is the same for any i. So we have

$$\frac{g_1}{b_1^2} = \frac{g_2}{b_2^2} = ... = \frac{g_f}{b_f^2} \tag{3.25}$$

that is, b_i is proportional to $\sqrt{g}_i$.

Let $b_i = \frac{\sqrt{g}_i}{\nu}$, $i = 1, 2, ...f$. Substituting this for b_i in Equation 3.24, we have

$$\mu c_2 M \nu^2 - 2\mu c_2 \sum_{i=1}^{f} \sqrt{g_i}\nu - f c_1 = 0 \tag{3.26}$$

This is a quadratic equation over ν. Solving it we have

$$\nu = \frac{\mu c_2 \sum_{i=1}^{f} \sqrt{g_i} \pm \sqrt{\mu^2 c_2^2 (\sum_{i=1}^{f} \sqrt{g_i})^2 + f\mu c_1 c_2 M}}{\mu c_2 M}$$

$\nu > 0$, so only the one with "+" before the square root on the numerator is the solution. So

$$\begin{aligned} b_i = \frac{\sqrt{g_i}}{\nu} &= \frac{\mu c_2 M \sqrt{g_i}}{\mu c_2 \sum_{j=1}^{f} \sqrt{g_j} + \sqrt{\mu^2 c_2^2 (\sum_{j=1}^{f} \sqrt{g_j})^2 + f\mu c_1 c_2 M}} \\ &= \frac{M}{\frac{\sum_{j=1}^{f} \sqrt{g_j}}{\sqrt{g_i}} + \sqrt{\left(\frac{\sum_{j=1}^{f} \sqrt{g_j}}{\sqrt{g_i}}\right)^2 + \frac{f c_1 M}{\mu c_2 g_i}}} \end{aligned} \tag{3.27}$$

where $i = 1, 2, ..., f$.

$$
\begin{aligned}
b_0 &= M - \textstyle\sum_{i=1}^{f} b_i \\
&= M - \frac{M\sum_{j=1}^{f}\sqrt{g_j}}{\sum_{j=1}^{f}\sqrt{g_j} + \sqrt{(\sum_{j=1}^{f}\sqrt{g_j})^2 + \frac{fc_1M}{\mu c_2}}}
\end{aligned}
\tag{3.28}
$$

$\frac{\partial e}{\partial b_i} = 0$ is a necessary condition of e taking the maximum or minimum. From the above analysis, we can see there is only one solution satisfying the necessary condition. It is easy to test that a different group of b_i values gives larger e than the e given by the b_i values in Equations 3.27 and 3.28. To show that b_i values in Equations 3.27 and 3.28 give the minimum of e, it is enough to show that the minimum does not occur at the boundary of the domain of the variables $b_0, b_1, ...b_f$. b_i satisfies that $0 < b_i < M$, $i = 0, 1, ..., f$ and $M = \sum_{i=0}^{f} b_i$. The boundary of the domain has at least one b_i approaching 0 from the right. When any of the b_i approaches 0 from the right, e approaches positive infinity. Therefore, the minimum does not occur at the boundary of the domain. Therefore the b_i values in Equations 3.27 and 3.28 gives the minimum of e. Here, we view b_i as real numbers, but in reality, they are integers, which may not take the exact values given by Equations 3.27 and 3.28. However, as M is a very large integer, typically 10,000 to 100,000 and hence b_i is usually of magnitude of thousands. The fractional parts of the exact values calculated from Equations 3.27 and 3.28 are negligible (less than 1/1000 of the actual value). Therefore we can still safely use Equations 3.27 and 3.28 to obtain the b_i values to get the minimum e. As tested in the experimental study, the cost of using this space allocation scheme always has less than 1% error from the optimal cost for two-level cases.

A key consequence of our analysis is that we should allocate space proportional to the *square root* of the number of groups in order to achieve the minimum cost.

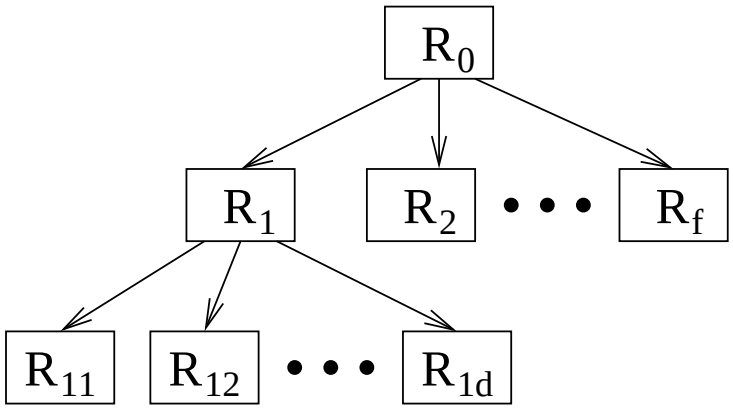


Figure 3.13: A case of three levels

Another interesting point is that b_0 (the space allocated to the hash table of the phantom) always takes more than half the available space.

3.7.2 A case of three levels

In this subsection, we consider a simple case of three levels. In this case, one phantom R_0 feeds R_1, R_2, ... R_f; where R_1 is a phantom and R_2, R_3, ... R_f are queries. R_1 feeds d queries R_{11}, R_{12}, ... R_{1d}. Figure 3.13 shows the feeding relationship of this configuration.

Let x_i be the collision rate of R_i, $i = 0, 1, ...f$; and x_{1i} be the collision rate of R_{1i}, $i = 0, 1, ...d$. Similar to the notation in the previous subsection, $x_i = \mu g_i / b_i$, $i = 0, 1, ..., f$, $x_{1i} = \mu g_{1i} / b_{1i}$, $i = 1, 2, ...d$. The total size is M, so $M = \sum_{i=0}^{f} b_i + \sum_{i=1}^{d} b_{1i}$. The cost of this configuration is

$$
\begin{aligned}
e &= c_1 + fx_0c_1 + dx_0x_1c_1 + x_0x_1\sum_{i=1}^{d} x_{1i}c_2 + x_0\sum_{i=2}^{f} x_ic_2 \\
&= x_0(fc_1 + dx_1c_1 + x_1c_2\sum_{i=1}^{d} x_{1i} + c_2\sum_{i=2}^{f} x_i) + c_1 \\
&= \mu\frac{g_0}{b_0}(fc_1 + dc_1\mu\frac{g_1}{b_1} + \mu^2c_2\frac{g_1}{b_1}\sum_{i=1}^{d}\frac{g_{1i}}{b_{1i}} + \mu c_2\sum_{i=2}^{f}\frac{g_i}{b_i}) + c_1 \\
&= \frac{\mu g_0(fc_1 + dc_1\mu\frac{g_1}{b_1} + \mu^2c_2\frac{g_1}{b_1}\sum_{i=1}^{d}\frac{g_{1i}}{b_{1i}} + \mu c_2\sum_{i=2}^{f}\frac{g_i}{b_i})}{M - \sum_{i=1}^{f} b_i - \sum_{i=1}^{d} b_{1i}} + c_1
\end{aligned}
\tag{3.29}
$$

e is a function of multi-variables, $b_1, b_2, ..., b_f, b_{11}, b_{12}, ...b_{1d}$. Similarly, we find the partial derivatives and let them equal to 0, and we obtain the following two equations:

$$
\frac{d\mu g_1c_1 + \mu^2g_1c_2\sum_{i=1}^{d}\frac{g_{1i}}{b_{1i}}}{b_1^2} = \frac{\mu c_2g_j}{b_j^2} = \frac{\mu^2g_1g_{1m}c_2}{b_1b_{1m}^2} \tag{3.30}
$$

where $j = 2, 3, ...f$, $m = 1, 2, ...d$.

$$
\frac{fc_1 + dc_1\mu\frac{g_1}{b_1} + \mu^2c_2\frac{g_1}{b_1}\sum_{i=1}^{d}\frac{g_{1i}}{b_{1i}} + \mu c_2\sum_{i=2}^{f}\frac{g_i}{b_i}}{M - \sum_{i=1}^{f} b_i - \sum_{i=1}^{d} b_{1i}} = \frac{\mu c_2g_j}{b_j^2} \tag{3.31}
$$

We have the following observations. First, relations at the same level with no children still have hash table size proportional to square root of the number of groups. So we let $b_i = \frac{\sqrt{g}_i}{\nu}$, $i = 2, 3, ...f$, and $b_{1i} = \frac{\sqrt{g}_{1i}}{\rho}$, $i = 1, 2, ...d$, substitute them in Equation 3.30, we have

$$
b_1 = \mu g_1\frac{\rho^2}{\nu^2} \tag{3.32}
$$

$$\nu^2(dc_1 + \mu c_2 \sum_{i=1}^{d} \sqrt{g_{1i}}\rho) = c_2\mu^2 g_1 \rho^4 \tag{3.33}$$

Substitute them in Equation 3.31. The result is an equation of order 8 in which the coefficients of the variable are parameters such as c_1, c_2, g_i, f, d in the above equations. Equations of order higher than 4 cannot be solved by radicals according to Galois' Theory, that is, we do not have a closed form solution for the equation. In addition, because the coefficients are parameters which can take wide range of values that we do not know in advance, we cannot determine whether the equation is solvable in advance. More general multiple-level configurations would result in even higher order equations which cannot be solved algebraically and we cannot determine whether they can be solved in advance, either. Therefore, we call configurations with more than three levels (inclusive) "**untractable**" configurations and propose heuristics to decide space allocation for them based on the analysis results we have for the two-level configuration. We understand that these equations can be solved numerically. However, numerical methods are iterative methods. They attempt to solve the equation by finding successive approximations to the solution from an initial guess such as the Newton's method. There is no guarantee on the number of iterations needed to find the solution and the speed of finding the solution depends on a good initial guess. Numerical methods may be too expensive computationally for our application where efficiency of the algorithm is essential. Therefore, we do not use numerical methods in our algorithm.

3.7.3 Other cases

In the previous subsections, we have obtained result of a simple two-level case and shown that three- or more than three-level cases cannot be solved. There are still other cases with less than two levels we haven't considered, yet. We give the results

of these cases as follows while omitting the analysis details since they follow similar derivations as in the cases discussed above.

1. The configuration has no phantom but only leaves, that is, this is a one-level case. This case can be solved. To achieve minimum cost, allocate space proportional to the square root of the number of groups.

2. The configuration has one phantom, but it does not feeds all queries, that is, some queries are fed from the stream directly. This is a two-level case, and it results in an equation of order 6, which cannot be solved.

3. The configuration has two phantoms, each of which feed some queries but they do not feed each other. Together they feed all queries. This is also a two-level case and it results in an equation of order 8, which cannot be solved.

In short, only the case with no phantom or the case with one phantom feeding all queries can be solved. Any other case cannot be solved.

3.7.4 Heuristics

For untractable configurations, we propose heuristics to allocate space based on the analysis of the solvable cases and partial results we can get from the untractable cases. Experimental results in Section 3.8 show that our proposed heuristics based on the analysis are very close to optimal and better than other heuristics.

In the analysis on the two-level case in Section 3.7.1, we observe from Equation 3.25 that the square of the number of buckets assigned to a relation, b_i^2 (R_i is a leaf relation), should be proportional to the number of groups of that relation, g_i, compared to other leaf relations at the same level. In the analysis on the three-level case in Section 3.7.2, we can observe similar behavior on relations at the same level

from Equation 3.30, that is, b_j of relations at the same level is proportional to $\sqrt{g_j}$. However, b_1^2 (R_1 is a non-leaf relation) is proportional to $d\mu g_1 c_1 + \mu g_1 c_2 \sum_{i=1}^{d} x_{1i}$ (note that $\mu g/b = x$), where x_{1i} are the collision rates of tables for the children of b_1. b_1 is affected not only by its own number of groups, but also its children's. From these observations, we gain the intuition that, generally, the number of buckets allocated to a relation at the same level should be proportional to the square root its number of groups. Moreover, if a relation feeds some other relations, we should add some weight to this relation according to the number of groups of the relations being fed. Having this intuition, we consider the following space allocation scheme (heuristic 1) which adds some weight to a relation when it has children to feed.

Heuristic 1: Supernode with Linear Combination (SL).

We only have the optimal solution for the two-level case, but not for a general case where we have more levels of relations in the configuration. Therefore, we introduce the concept of "supernode" so that we can use the two-level space allocation scheme recursively to solve the general case. Specifically, we start from the leaf level of the configuration. Each relation right above the leaf level together with all its children are viewed as a ***supernode***. The number of groups of this supernode is the sum of the number of groups of the relations that compose this supernode, that is, the phantom and the queries it feeds. Then we view the supernode as a query and do the above compaction recursively until the configuration contains only two levels. For a two-level configuration, we can allocate space optimally according to the analysis of Section 3.7.1. After the first space allocation, each query (some may be supernodes) is assigned some space. Then, we decompose each supernode to a two-level configuration and allocate the space of this supernode to the phantom and queries optimally inside the supernode again, that is, allocate space proportional

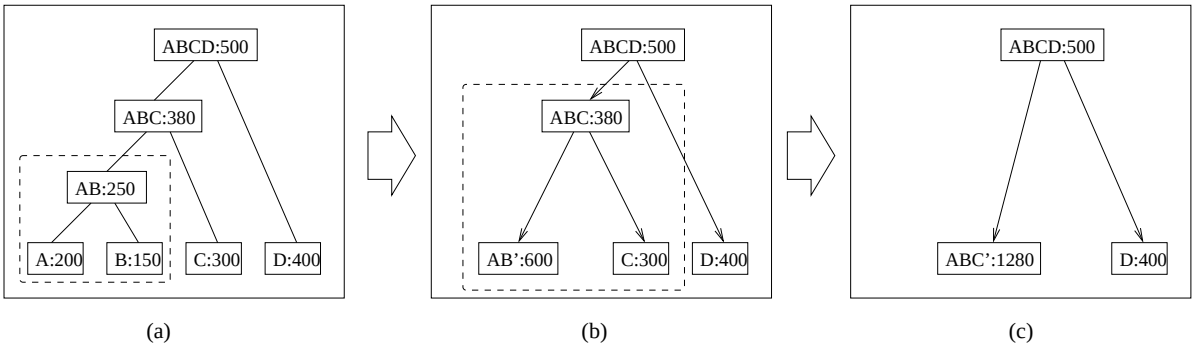


Figure 3.14: Heuristic SL

to the square root of the number of groups. We do this decomposition recursively until no supernode exists.

Figure 3.14 shows an example of how heuristic SL works. Figure 3.14 (a) is the configuration that needs to allocate space. This configuration has four queries: A, B, C, D, and three phantoms: AB, ABC, ABCD. The number of groups of each relation is written beside the relation. According to heuristic SL, relations A, B and AB are first combined into a supernode AB', whose number of groups is the sum of the number of groups of A, B and AB. The first combination results in Figure 3.14 (b). We still have more than two levels, so we continue to combine AB', C and ABC to a supernode, which results in a the two-level configuration as shown in Figure 3.14 (c). Now we can allocate space according to the analysis on the two-level configuration and unfold the supernodes one level after another until the original configuration.

We also try another heuristic described below (heuristic 2) which is the same as SL except the way we obtain the number of groups of the supernode.

Heuristic 2: Supernode with Square Root Combination (SR). Since in

the two-level case we see that the space should be proportional to the square root of the number of groups, we may also let the *square root* of the number of groups of the supernode be the sum of the *square roots* of all its relations, while the other steps are the same as in heuristic 1.

Note that both SL and SR give the optimal result for the case of one phantom feeding all queries. We will also try two other simple heuristics (heuristic 3 and 4) which are not based on our analysis as a comparison to the above two more well-founded heuristics.

Heuristic 3: Linear Proportional Allocation (PL). This heuristic simply allocates space to each relation proportional to the number of groups of that relation.

Heuristic 4: Square Root Proportional Allocation (PR). This heuristic allocates space to each relation proportionally to the square root of the number of groups of that relation.

Although we cannot obtain the optimal solution for space allocation of some cases through analysis, there does exist a space allocation which gives the minimum cost for each configuration. One way to find this optimal space allocation is to try all possibilities of allocation of space at certain granularity. For example, suppose the configuration has three relations, AB, A and B, where AB feeds A and B. The total space is 10. We can first allocate 1 to AB, 1 to A, and 8 to B. Then we try 1 to AB, 2 to A, and 7 to B, and so on. By comparing the cost of all these space allocation choices we will find the optimal one. We call this method

the **exhaustive space allocation (ES)**. Obviously this strategy is too expensive to be practical, but we use it in our experiments to compare with the four space allocation schemes and see how much the heuristics differ from the optimal choice. The results of ES are affected by the granularity of varying the space allocation. In our experiments, we found that using a granularity of 1% of M is small enough to provide accurate results.

The space allocation schemes are independent of the phantom choosing strategies, that is, given a configuration, a space allocation scheme will produce a space allocation no matter in what order the relations in the configuration are chosen. Therefore we will evaluate space allocation schemes and phantom choosing strategies independently.

3.7.5 Revisiting simplifications

From the beginning of the analysis on space allocation, we have made an approximation on the linear expression of the collision rate, that is, we let x equal $\mu g/b$ instead of $\alpha + \mu g/b$. We also did the analysis when we let $x = \alpha + \mu g/b$. The result of the case with no phantom is the same. The case with one phantom feeding all queries results in a quartic equation which can be solved, so we can still get an optimal solution for this case. However, because solving a quartic equation is much more complex than a quadratic equation and it is more involved to decide which solution of the quartic equation is the one we want, we use the approximated linear expression, that is, $x = \mu g/b$ for space allocation in our experiments. The experimental study shows that even with this approximation, the results are still very accurate.

We have also tried other ways of approximating Equation 3.19 such as: (1) $(1 - \frac{1}{b})^g \approx e^{-\frac{g}{b}}$; (2) taking the first few terms of the binomial expansion. However,

for alternative (1), the result is an even more complicated equation system, which is of higher degree and has variables in exponents. For alternative (2), if we take the first 3 terms of the binomial expansion, it's nearly a linear approximation, but with much worse accuracy; if we take more terms of the binomial expansion, the result is a much more complicated equation system than using our linear approximation and the accuracy is no better than our linear approximation.

We have made another simplification on the size of each hash table bucket entry in the analysis for ease of exposition. By using $M = \sum b_i$, we have assumed that a hash table entry has the same size for all relations in the LFTA. Actually, the size of a hash table entry for different relations can vary a lot. Suppose we use an int (4 byte) to represent each attribute or a counter. Then a bucket for relation A takes 8 bytes and a bucket for ABCD takes 20 bytes. If we denote the bucket entry size of relation i as h_i, then $M = \sum b_i h_i$. In this case, the results of the analysis are similar. Instead of allocating space proportional to $\sqrt{g}$, we should allocate space proportional to $\sqrt{g_i h_i}$. We have used such variable sized buckets in our implementation, and experimental study, discussed next.

For clustered data, collision rates should be divided by the average flow length l_a. To consider this in space allocation, we should allocate space proportional to $\sqrt{g_i h_i / l_i}$, where l_i is the average flow length of relation i.

3.8 Experiments

3.8.1 Experimental setup and data sets

We prototyped this framework in C in order to evaluate the different techniques we developed. We use 4 bytes as our unit of space allocation. Each attribute value and counter we instantiate has this size. As explained in Section 3.1.1, LFTAs are run

on a Network Interface Card with a memory constraint and use a small memory size to fit into L2 cache of the CPU, typically several hundred KB of memory is allowed. In accordance to operational streaming data managers [46], we consider M between 20,000 and 100,000 units of space (4 bytes each). The ratio of eviction cost to probe cost c_2/c_1 is is modeled as 50 in our experiments, which is also a ratio measured in operational data stream management systems [46].

We used both synthetic and real data sets in our evaluation. The real data set is obtained by *tcpdump* on a network server of AT&T. We extracted TCP headers obtaining 860,000 records with attributes source IP, destination IP, source port and destination port, each of size 4 bytes. The duration of all these packets is 62 seconds. There are 2837 groups in this 4-attribute relation. For other relations we extracted in this way, the number of groups varies from 552 to 2836. For the synthetic data sets, we generated 1,000,000 3 and 4 dimensional tuples uniformly at random with the same number of groups as those encountered in real data. All the experiments are run on a desktop with Pentium IV 2.6GHz CPU and 1GB RAM.

We adopt the following way to specify a configuration. "AB(A B)" is used to denote a phantom AB feeding A and B. We use this notation recursively. For example, the configuration in Figure 3.4(c) can be expressed as (ABCD(AB BCD(BC BD CD))).

3.8.2 Evaluation of space allocation strategies

Our first experiment aims to evaluate the performance of various space allocation strategies. In these experiments we derive our parameters from the real data set. Our observations were consistent across a large range of real and synthetic data sets. We vary M from 20,000 to 100,000 at steps of 20,000 and the granularity for increasing space while executing ES is set at 1% of M. In all experiments we

compute the cost using Equation 3.7 with a suitable model for collision rate, as described below.

Solvable configurations

We first experimentally validate the results of our analysis for the case of configurations for which we can analytically reason about the goodness of space allocation strategies.

For the case with no phantoms, (assuming $x = \mu g/b$ as collision rate) we compared the cost of the exhaustive space allocation (ES) with a scheme that allocates space according to our analytical expectations, namely, allocating space proportional to the square root of number of groups. We tested all possible configurations with no phantoms on the real data. The cost obtained by the scheme performing space allocation as dictated by our analytical derivations incurred a difference less than 1% compared to the optimal cost (obtained by ES). The small difference comes from our approximation to the collision rate, especially the value of μ, which can be slightly different from the value the optimal solution assumes.

For the case with only one phantom feeding all queries, we use our optimal space allocation scheme derived based on the approximation of collision rate x by $\mu g/b$. We again compare the accuracy of the space allocation scheme allocating space according to our analysis, to that of ES and test all possible configurations for on the real data set. The average relative difference between the cost obtained from our scheme and the optimal cost is usually less than 1% and the maximum observed was 2%. Therefore even with this approximation ($x = \mu g/b$) to the collision rate, the results are still quite accurate.

Untractable configurations

For untractable configurations, we evaluated several heuristics. We compared SL, SR, PL, PR as described in Section 3.7.4 and ES. We evaluated all possible configurations on the real data set (four attributes). The relative costs of the heuristics compared to the optimal cost (obtained by ES) are shown in Figures 3.15 to 3.18 for 4 representative configurations. The average relative costs of the heuristics compared to the optimal cost of all configurations are summarized in Table 3.2.

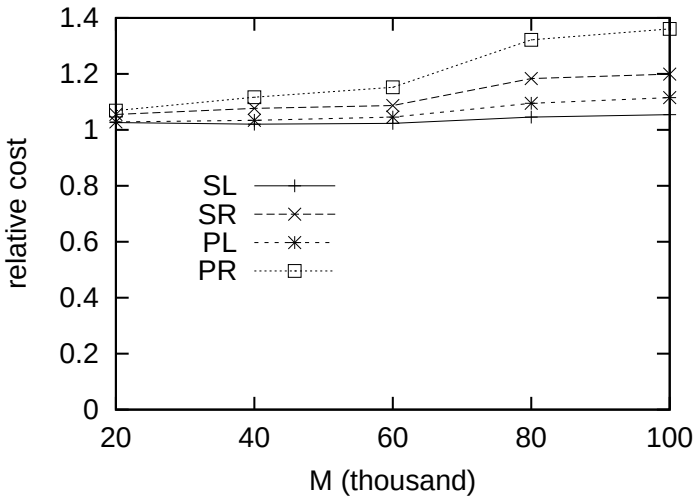


Figure 3.15: Space allocation for (ABC(AC(A C) B))

We observe that generally SL and SR are better than PL and PR. Thus, heuristics inspired by our analytical results appear beneficial. Except one case in Figure 3.17 when $M = 20,000$, SL is always the best. Relative costs of PL and PR can be as large as 35% more than the optimal cost. The cost of SR is smaller than those of PL and PR, but it is always more than the cost of SL. From Table 3.2, which shows the average relative costs of the four different heuristics compared to the optimal cost, we observe that SL is the best for all values of M. Therefore, SL may be a good choice of space allocation. To verify this, we further accumu-

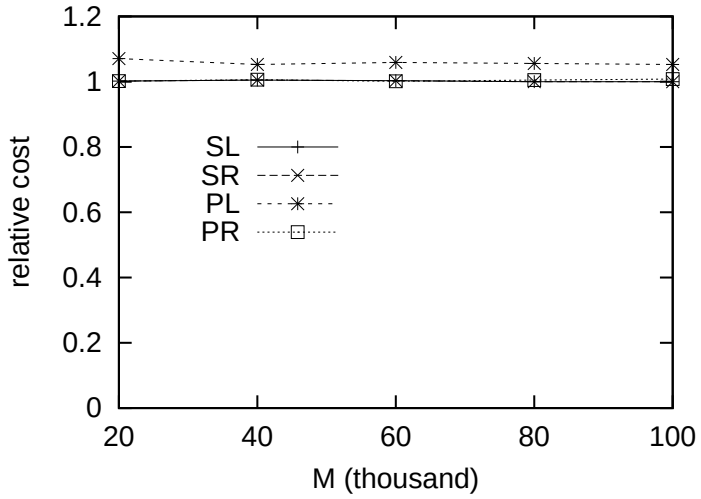


Figure 3.16: Space allocation for AB(A B) CD(C D)

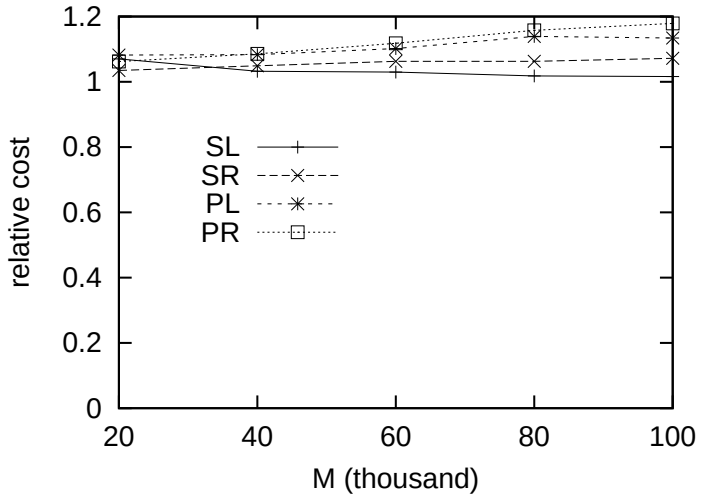


Figure 3.17: Space allocation for (ABCD(ABC(A BC(B C)) D))

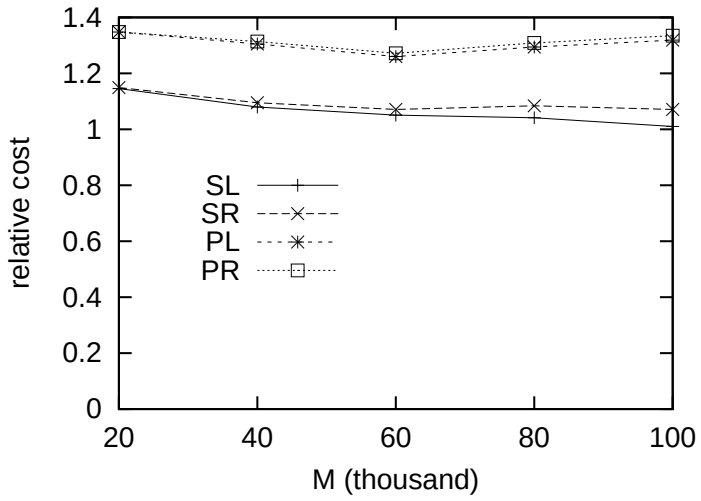


Figure 3.18: Space allocation for (ABCD(AB BCD(BC BD CD)))

M (thousand)	20	40	60	80	100
SL	1.060	1.030	1.022	1.032	1.023
SR	1.062	1.053	1.053	1.090	1.094
PL	1.158	1.142	1.146	1.214	1.234
PR	1.101	1.114	1.124	1.197	1.227

Table 3.2: Average relative costs of the four heuristics

late statistics in order to see in all configurations tested how frequently SL is the heuristic yielding the minimum cost.

In Table 3.3, we present the percentage of configurations tested in which SL yields minimum cost among the four heuristics, as well as for the cases that SL does not yield the minimum cost, how far its cost is from the cost of the best heuristic. These results (which are representative of a large set of experiments conducted) attest that SL behaves very well across a wide range of configurations. Even in the cases that it is not the best it remains highly competitive to the best solution. Therefore we would choose SL for space allocation in our algorithms.

M (thousand)	20	40	60	80	100
SL being best (%)	44	89	89	89	100
Relative difference from the best (%)	2.2	0.006	0.15	0.6	0

Table 3.3: Statistics on SL

3.8.3 Evaluation of the greedy algorithms

We now turn to the evaluation of algorithms to determine beneficial configurations of phantoms. We will evaluate the greedy algorithm GS and our proposed greedy algorithm GC. GC makes use of the SL space allocation strategy; we refer to this combination as GCSL (algorithm GC using SL space allocation). For GS, we would add space of ϕg each time a phantom is added in the current configuration under consideration until there is not enough space for any additional phantom to be considered. At this point we allocate the remaining space to relations already in the configuration proportional to their number of groups. We also consider the following method to obtain the optimal configuration cost. We explore all possible combinations of phantoms and for each configuration we use exhaustive space (ES) allocation to calculate the cost, choosing the configuration with the minimum overall cost. We will refer to this method as EPES in the sequel. Costs are computed using Equation 3.7 and our approximation to the collision rate.

Phantom choosing process

We first look at the query set {A, B, C, D} on a 4-dimensional uniform random data set with M set as 40,000. Figure 3.19 presents the cost of the different algorithms. The costs are normalized by the cost of EPES (the optimal cost). Algorithm GS has a parameter ϕ. Since a good value of ϕ is not known a priori, we vary it and observe the trend of the cost resulted from different ϕ values. The cost of GS first decreases and then increases, as ϕ increases. If ϕ is too small, each phantom is

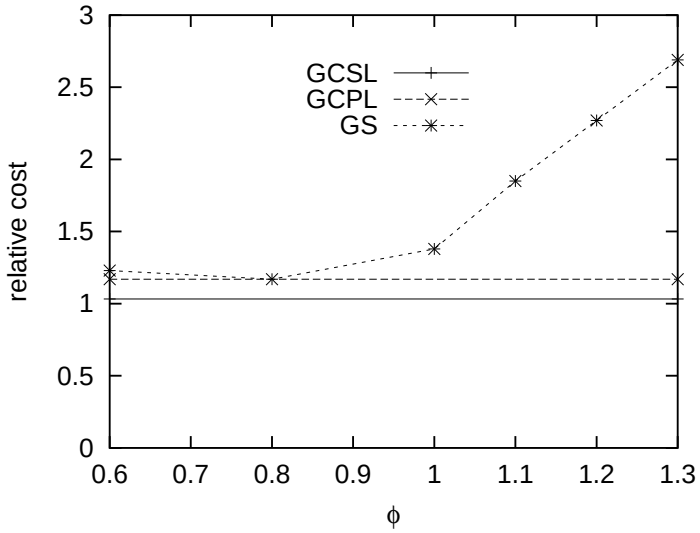


Figure 3.19: Comparison of phantom choosing algorithms

allocated a small amount of space, at the expense of high collision rate. On the other hand, if ϕ is too large, each phantom has low collision rate, but each phantom takes too much space and prohibits addition of further phantoms, which could be beneficial. This alludes to a knee in the cost curve signifying the existence of an optimum value. Algorithms GCSL and GCPL do not have the parameter ϕ, so their costs are constant in the Figure. For the GCSL algorithm, cost is lower than the cost of GS for any ϕ, because when we adjust the space allocation and calculate the cost each time a phantom is added, we are essentially adapting ϕ to a better value. The gap between the minimum point of the GS curve and GCSL is due to the space allocation scheme. Using GC in conjunction with PL space allocation, yields a curve which precisely lower bounds GS. Thus, GCSL benefits from both the way we choose phantoms and the way space is allocated in these phantoms.

Figure 3.20 presents the change in the overall cost in the above scenario as each phantom is chosen. We observe that the first phantom introduces the largest decrease in cost. The benefit decreases as more phantoms are added and for GS with $\phi = 0.6$, the cost goes up when adding the third phantom. Note that the third

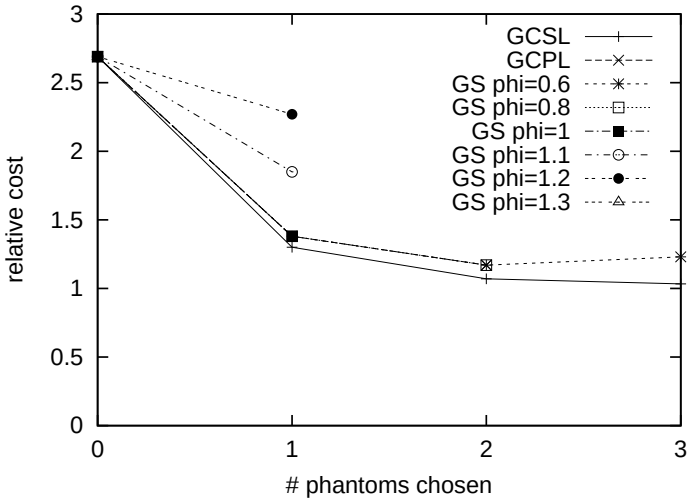


Figure 3.20: Phantom choosing process

phantom added by GS with $\phi = 0.6$ is different from the third phantom added by GCSL due to the differences in space allocation. For GS with $\phi = 1.2, 1.3$ there is no space to add more than one phantom.

We conducted an additional experiment by varying M and observing the resulting cost. We use synthetic data with four attributes comparing GCSL and GS. We normalize their respective costs by the cost of EPES. For GS, it's impossible to vary ϕ for a single query window as this is fixed at the start of the window. Therefore we set $\phi = 0.8$ for these experiments. The results are shown in Figure 3.21. GCSL always has a cost lower than 1.1 of the optimal. The cost of GS has a drop at M=40,000 because the ϕ value we used is based on the optimal ϕ value at this point. We can see that GCSL is always better than GS for all M values.

Validating cost estimation framework

With our next experiment we wish to validate our cost estimation framework against the real measured errors. We implemented the hash tables and we let a uniform random data set pass through the phantoms and queries computing the

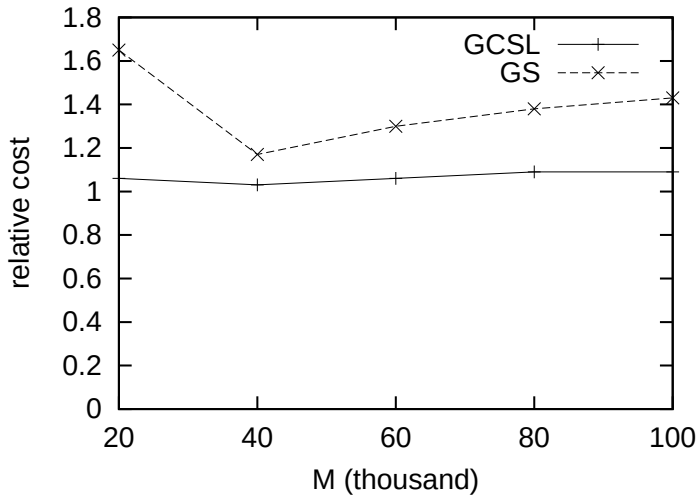


Figure 3.21: Cost comparison

desired aggregates. The phantoms are chosen and the corresponding space allocation is conducted, using our heuristics. We count the collisions in the hash tables and calculate the true cost of this configuration. We normalize the actual cost of GCSL and GS by the actual cost of the optimal (according to our cost model) configuration obtained by EPES; the relative actual costs are shown in Figure 3.22. For GS, we tried different ϕ values, and only the one with the lowest cost at each value of M is presented in the figure.

We can see that the actual cost of GCSL is always much lower than that of GS, even we could always choose the best ϕ for GS (which is impossible in practice). When M=60,000, the cost of GCSL is as low as 26% of the cost of GS. While GS can have cost as high as 6 times the optimal cost, GCSL is always within 3 times the optimal cost. The reason that GCSL does not achieve the optimal cost perfectly is as follows. When we have a phantom feeding other relations, the data in the phantom and the data fed to its children are correlated. We have assumed random data distribution in the collision rate model. The correlation between the data makes the collision rate model less accurate and therefore causes some error in the

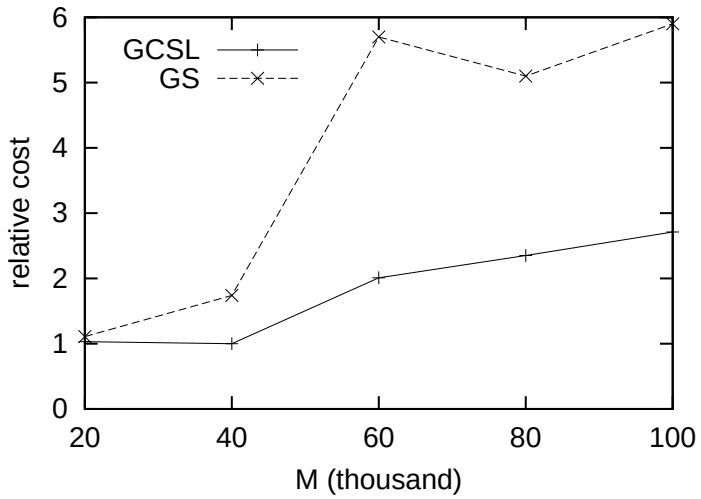


Figure 3.22: Comparison on synthetic data set: GCSL vs. GS

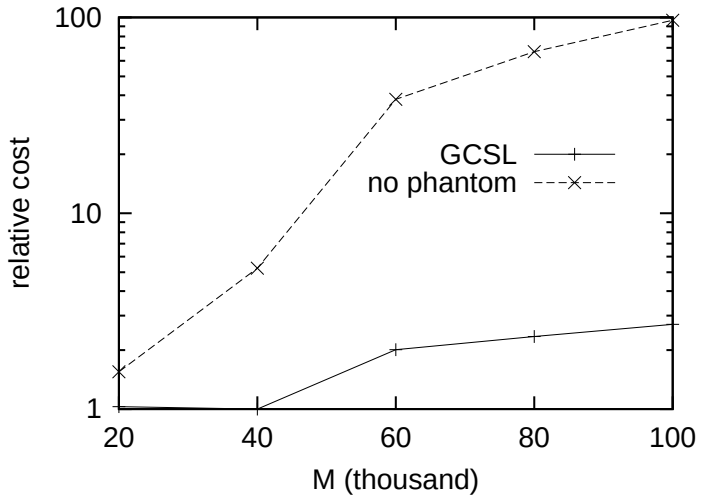


Figure 3.23: Comparison on synthetic data set: GCSL vs. no phantom

space allocation. However, from the result we can see the error is not that big and GCSL still obtains good approximation of the optimal cost, especially compared with GS.

We conducted a large set of experiments quantifying the accuracy of our estimation framework against actual measurements. In general, difference between the predictions of the cost model and the actual cost becomes large as M increases. The relative cost difference of GCSL compared to the optimal cost also increases as M increases. This is due to two factors: first when M is very large then collision rates are very small and become increasingly difficult to capture analytically. Second, for large M there are many phantom levels and as a result errors accumulate across multiple levels. However, despite certain inaccuracy, our technique results in a reasonable low cost compared to the optimal cost and outperforms GS considerably, for a variety of data sets, especially for low values of M (which is the common case in practice).

In order to validate the effectiveness of phantoms for computing multiple aggregates, we conducted the following experiment. We run the same queries without maintaining any phantoms and we compare the cost with the cost of GCSL. The results are presented in Figure 3.23. It is evident that maintaining phantoms does reduce the cost greatly (more than an order of magnitude).

Experiments with real data

We repeated our validation experiment using real data this time and the query set {AB, BC, BD, CD}. Again we let the real data set stream by the configuration we have obtained using our algorithms and report the resulting actual costs incurred. Once again actual costs are normalized by the actual cost incurred by the EPES strategy. Flow length is derived temporally.

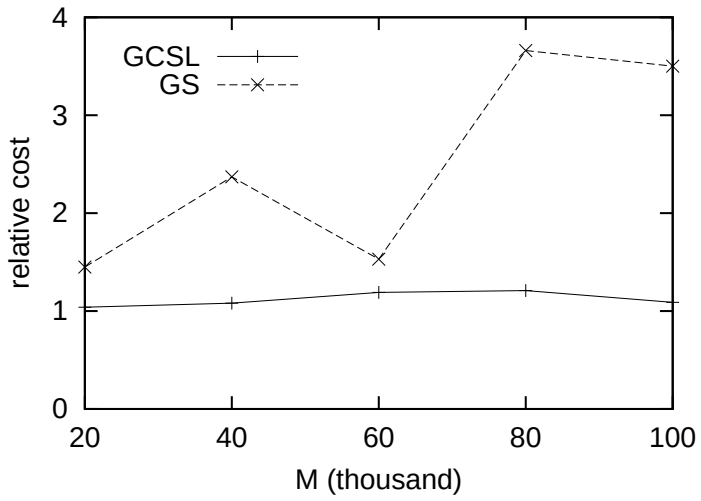


Figure 3.24: Comparison on real data set: GCSL vs. GS

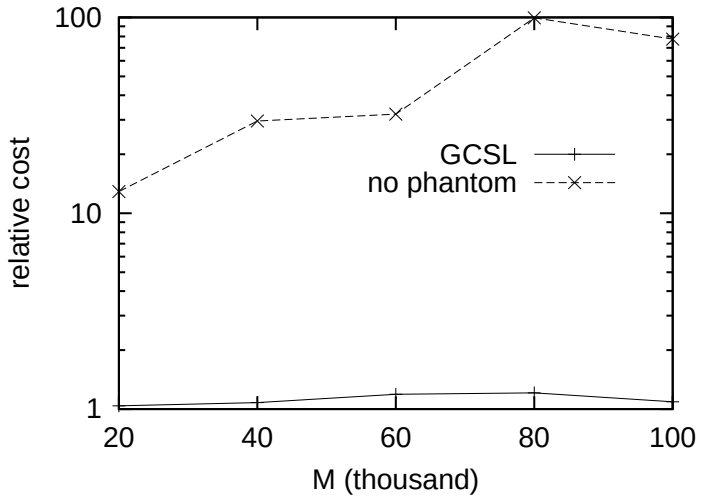


Figure 3.25: Comparison on real data set: GCSL vs. no phantom

Figure 3.24 presents the results. It is evident that GCSL outperforms GS. Once again we compare the cost of GCSL and the cost incurred without the maintenance of any phantoms. GCSL offers an improvement up to about 100 compared to the cost incurred without the use of phantoms as shown in Figure 3.25.

Peak load constraint

The update cost at the end of epoch as described in Section 3.3.2 can be calculated according to Equation 3.8. This update cost must be within the peak load constraint E_p. If the update cost E_u exceeds E_p, we can use two methods to resolve it: *shrink* and *shift*. The shrink method shrinks the space of all hash tables proportionally. The shift method shifts some space from queries to phantoms since c_2 is much larger than c_1 and a major part of the update cost is incurred by queries. For the real data set and the query set {AB, BC, BD, CD}, given a space allocation, we calculate its E_u; then we set E_p to a percentage of E_u and use the two methods to reallocate space. After the reallocation, we run the data through the configuration and we compute the cost when $M = 40,000$. The results are in Figure 3.26. When E_p is not much smaller than E_u, the shift method performs better; while when E_u is much larger than E_p, the shrink method performs better. The reason is that when E_u is close to E_p, a small shift to reduce E_u suffices. When E_u and E_p differ by much, a major shift in space results in non optimal space allocation and thus shrink is better. Similar behavior is observed when M is set as other values. 3.25. In terms of the performance of our algorithms, the running time of GCSL in all configurations we tried is sub-millisecond, which is negligible compared to a time window of one minute. While typically we just have a few aggregate queries running together, our algorithm has good scalability since the computation cost is linear to the number of relations in the feeding graph.

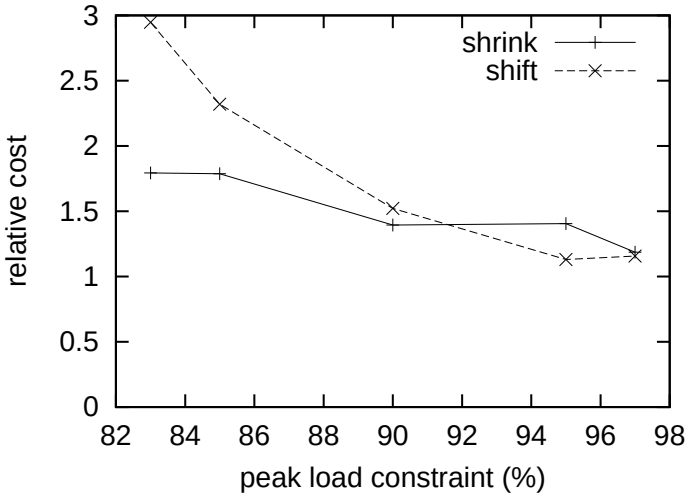


Figure 3.26: Peak load constraint

3.9 Summary

In this chapter, we attacked the problem of efficient aggregation over network data streams. Our work is based on the architecture of Gigascope and our focus is an optimization problem when faced with multiple aggregations. We first reviewed how a single aggregation is processed in Gigascope. Then we examined the problem of multiple aggregations and introduced our insight of maintaining additional queries, called phantoms, to facilitate sharing among queries. We formalized our problem which consists of two sub-problems, phantom choosing and space allocation. We showed that the phantom choosing problem is NP-complete and therefore we proposed a greedy algorithm to solve it. An existing algorithm allocates space proportionally to the number of groups of relations and greedily chooses phantoms with the maximum benefit until the total available space runs out. Our greedy algorithm is new in the sense that we always allocate all the available space to the phantoms optimally so as to obtain the optimal cost given the phantoms. We keep adding phantoms greedily with the maximum cost reduction until no additional

phantom can reduce the cost. To obtain the optimal cost given a configuration (a set of queries and phantoms to maintain), we have to estimate collision rates of hash tables accurately and allocate space to relations optimally. Subsequently, we did in-depth mathematical analysis on these two issues. We showed that the space allocation problem can be solved for very limited cases while the other cases can not be solved generically, therefore we proposed heuristics for space allocation based on our analysis. We also derived a model to estimate the collision rate of the hash tables, which is a key component in the space allocation scheme. Finally, we presented our experimental study, which showed that our proposed space allocation heuristic yields costs very close to the optimal cost and our proposed phantom choosing heuristic beats existing algorithms. Furthermore, we demonstrated that maintaining phantoms has huge benefit over the scheme without phantoms.

CHAPTER 4

Approximate Nearest Neighbor Search Over Data Streams

As discussed in Section 1.2, network security applications such as intrusion detection and virus detection need to perform similarity search, which translates to (approximate) nearest neighbor queries over the network data streams. In this chapter, we present a technique to process approximate nearest neighbor queries over data streams. While this technique is applicable to general data stream applications such as those described in Section 4.1, we can fit it into the two-level query processing architecture of Gigascope fairly well.

The rest of the chapter is organized as follows: Section 4.1 gives the motivation for approximate nearest neighbor search over data streams by some example applications including network monitoring. We introduce a new type of approximate nearest neighbor query, the *e-approximate k nearest neighbor (ekNN) query* in Section 4.2. Section 4.3 gives a synopsis for our proposal to process the ekNN query.

In Section 4.4, we propose a general framework to reduce information while still answering the ekNN problem with some error bound. A brute-force method based on this framework is also presented in this section. Then we present a technique called aDaptive Indexing on Streams by space-filling Curves (DISC) to efficiently process ekNN queries from the maintained data in Section 4.5. Section 4.6 shows how to process the sliding window version of the ekNN query by DISC and Section 4.7 shows how to deploy DISC in the two-level query processing architecture of Gigascope. Section 4.8 reports the results of our experimental studies.

4.1 Motivation and applications

In many applications, including geographic information systems, content-based retrieval and data mining, finding the k Nearest Neighbors (kNN) to a query object is one of the most frequent operations. The database research community has in recent years provided several novel solutions to efficient kNN processing [154, 24, 146]. The kNN problem can be defined as follows: Given a set of points $S = \{P_0, P_1, ..., P_n\}$ in a d-dimensional space V, and a query point $Q \in V$, find a set kNN which contains k points in S such that, for any $P \in kNN$ and for any $P' \in S - kNN$, $dist(Q, P) \leq dist(Q, P')$, where $dist()$ is a function to return the distance between two points.

To further improve performance, the $(1 + \epsilon)$-approximate nearest neighbors problem ($\epsilon \geq 0$) [10, 106] has been introduced which is defined as follows: Find a point $P \in S$ that is an $(1 + \epsilon)$-approximate nearest neighbor of the query point Q, so that for any point $P' \in S$, $dist(P, Q) \leq (1 + \epsilon)dist(P', Q)$. The k $(1 + \epsilon)$-approximate nearest neighbors problem can be similarly defined [11]. Here ϵ is in fact a bound for the relative error of the k-th nearest neighbor distance, which is

specified by the users before the query.

KNN queries over multi-dimensional data streams is a pressing concern when mining streams for unknown patterns. For example, in computer aided manufacturing (CAM) systems, sensors are used to monitor the position, shape[1], size, surface characterization, material properties, etc, of parts passing through on a production line. The data are collected and sent to a control system. The control system analyzes the feedback information and then adjusts the parameters of the production line so as to control the quality of the parts. Often, we tend to identify parts with similar shape to a given part in order to discover patterns of other features. In highway traffic monitoring, sensors are embedded on highways to observe the passing vehicles. Estimates of vehicle speed and length can be obtained and utilized to provide useful traffic related information. Similarly in network traffic monitoring, network traffic streams (IP traffic) are usually logged using special programs, such as CISCO's netflow. The network management system will monitor the network packet header information to obtain information on traffic flow patterns as discussed in Section 1.2.

In addition, data stream applications typically operate in an environment where memory is limited (relative to the size of the stream) so that it is not feasible to work with the entire data set in memory. For this reason, one has to resort to approximate kNN answers in the case of continuously evolving data streams. All previous proposals for approximate kNN queries require the user to specify a relative error bound ϵ ($\epsilon \geq 0$) beforehand. However, in certain applications, absolute error bounds are more critical and preferable. In the CAM example, a query typically specifies absolute errors: "Identify 10 parts that are most similar in size to a given part A. The query specifies that as long as a part's resultant error

[1]Even parts on a same production line have slightly different shapes and sizes due to manufacturing errors.

(that is, the root-sum-square of the errors in width and length) to those of the 10 most similar parts is not more than 0.1mm the answer is acceptable." In the highway traffic monitoring example, it may also be more intuitive to specify errors by absolute bounds: "Find the 20 vehicles that are close to position A. An answer is acceptable as long as its distance to A is not larger than say 10 meters than that of the 20 closest vehicles." Similar examples can be drawn from the field of network monitoring and other engineering applications, in which users have good knowledge of the absolute errors acceptable.

4.2 Problem formulation

Motivated by such applications, we introduce a new type of approximate nearest neighbor query, called the *e*-**approximate k nearest neighbor (ekNN) query**, in which the answers are bounded by absolute value instead of relative one. The data stream records have multiple attributes and we can represent them as a multi-dimensional point in a multi-dimensional space. In this chapter, a point means a data record. Formally, we define the ekNN query as following:

Definition 1 (ekNN) *Given a data set S and a query point Q, find a set $ekNN$ which contains k points in S such that for any $P \in ekNN$ there exists a point $P' \in kNN$ (the actual kNN set of Q) and $dist(Q,P) \leq dist(Q,P') + e$, where e is a bound for the absolute error of the k-th nearest neighbor distance.*

Subsequently, we define the *e*-**approximate kNN query over data streams** as follows:

Definition 2 (ekNN over data streams) *Let X be a sequence of points $(P_0, P_1, P_2, ...)$. X can be either finite or infinite. Each element $P_i (i = 0, 1, 2, ...)$ of X is a point in d-dimensional space and is allowed to be read for at most once in the order of*

the sequence. Let S_t be the set of points of X that have been read at time t. At any time t and for any query point Q, find the ekNN of Q from the elements of S_t.

In particular, we identify and provide solutions to the following ekNN problems on data streams:

1. **memory optimization for a given error bound**: given an error bound e, use as little memory as possible to answer ekNN queries.

2. **error minimization for a given memory size**: given a fixed amount of memory, achieve the best accuracy for ekNN queries.

4.3 Synopsis of our proposal

We first propose a general framework which aims to reduce the amount of information to be stored while guaranteeing a provable error bound. Specifically, we partition the underlying data space into equal square-shaped cells, and then we prove that in each cell we only need to store at most G (for a user specified value G) points[2] to guarantee some error bound. We will prove that the error bound is guaranteed for any ekNN query where $k \leq G$. Next, to facilitate efficient maintenance of G points in each cell, we propose a technique called aDaptive Indexing on Streams by space-filling Curves (DISC), in which points are stored in the leaf nodes of the B*-tree with the Z-values [126] of their cells as keys. DISC has two important properties: first, it only allocates memory for those points that are necessary to guarantee the error bound; second, by merging cells, DISC can adjust the structure to meet the memory constraint. These two properties make it adaptive to different data distributions. In addition, being a B*-tree based indexing structure, DISC provides fast access to a given cell. This facilitates efficient updates

[2] "A point in a cell" means "a point spatially contained in a cell".

and query processing. Overall, DISC can achieve our goals of minimizing memory usage for a given error bound or obtaining best accuracy for a given memory constraint while retaining efficient updates and query processing. We present the ekNN search algorithm based on DISC and also show how to modify DISC to support sliding window ekNN queries. Extensive performance studies using synthetic and real data sets were conducted, and the results demonstrate that DISC is both query and memory efficient. Note that since DISC is essentially a B*-tree based technique, it can also be used as a disk-based structure.

4.4 The framework

In this section, we propose a framework towards solving the ekNN problem with a guaranteed error bound. As we shall see, this scheme provides possibility to reduce the information to be stored, however, the scheme in itself does not guarantee achieving the goal of memory optimization or error minimization. The data structure used to implement it is also critical to achieve these two optimizations. Therefore we will first present the scheme, followed by analysis on adopting the most suitable structure to realize it.

Our overall approach consists of segmenting the underlying space into a number of cells and identifying dynamically a number of points to be stored in each cell (called the *footprints* of the data) as data stream passes by. We observe that, in order to guarantee the error bound e, which is the largest distance between two points in a cell, for kNN queries, we only need to maintain at most k points in each cell. In the case of data streams, the number of data is very large so that usually exceeds k in many cells. Therefore, by maintaining only k points, we can reduce the data to be stored. In the following, our scheme based on this observation is

formally presented.

4.4.1 Capturing the footprints

We consider the problem in a d-dimensional metric space V, which is a set of points with an associated distance function $dist$. The distance function $dist$ has the following properties for any points P_1, P_2, P_3 in V:

1. $dist(P_1,P_2)=dist(P_2,P_1)$
2. $dist(P_1,P_2)>0$ $(P_1 \neq P_2)$ and $dist(P_1,P_2)=0$ $(P_1=P_2)$
3. $dist(P_1,P_2) \leq dist(P_1,P_3)+dist(P_2,P_3)$

We divide the data space into a number of square-shaped cells and maintain at most G (G is a user specified constant) points in each cell. Specifically, as data stream passes by, each data point is placed in the cell it belongs to. If a cell already contains G points, there would be $G+1$ points including the new one. Then, we discard a point according to some discarding policy. The discarding policy is clearly application dependent. For example, if the most recent information is of interest we will always delete the oldest point. When processing ekNN queries, we invoke an exact kNN query on the set of points maintained, that is, the *footprints* of the stream data. Contrasting the kNN answers obtained from the footprints of the data set and on the original data set, we prove that the difference of their k-th nearest neighbor distance is within e, which equals the largest distance between two points in a cell. So the kNN on the footprints is an approximate answer for the kNN query on the original data set with error bound e. We start by defining some functions necessary for the derivations that follow and formalize the scheme for capturing the footprints. Some commonly used symbols in this chapter are summarized in Table 4.1. As we are dealing with a totally different problem in this chapter from the one

studied in Chapter 3, the symbol meanings listed below are only valid within this chapter.

Table 4.1: Symbols

Symbol	**Meaning**
c	A cell
d	Dimensionality
$dist(P_1, P_2)$	Function that returns the distance between the two points P_1 and P_2
e	The error bound of the k-th nearest neighbor distance
$far(S, P)$	Function that returns the farthest point in set S to point P
kNN	The set of the k nearest neighbors
$ekNN$	The set of the e-approximate k nearest neighbors
m	The order of the Z-curve
P	A data record, which is viewed as a multi-dimensional point
p_i	The i-th coordinate of point P
Q	A query point
S	A set of points
t	Current time
T	Some period of time
u	The number of segments a dimension is divided to
V	A metric data space
W	A query window
W_s	The smallest query window that contains $ekNN$

We assume that the data space is normalized to a unit hypercube. Each of the d dimensions of V is divided equally into u segments (therefore V is divided into u^d cells). Let S be a set of points in V and c a cell in V. Define $S(c)$ as $\{P \in S | P \in c\}$, that is, the subset of S that is in the cell c.

Let M be a mapping on S which is defined as follows:

for each cell c of V, if $|S(c)| > G$, image of $S(c)$ is the set of any G points in $S(c)$;

if $|S(c)| \leq G$, image of $S(c)$ is $S(c)$. S is the union of $S(c)$ for all the cells, and hence the image of S is the union of the image of $S(c)$ for all the cells.

Let S' be the image set of S under mapping M. Formally, the points in S' are the ***footprints*** of the points in S. For any query point $Q \in V$, kNN is the set of k nearest neighbors of Q in S and kNN' is the set of k nearest neighbors of Q in S'. Let $far(S, Q)$ be the function returning the point in S, which is of largest distance to Q among all the points of S.

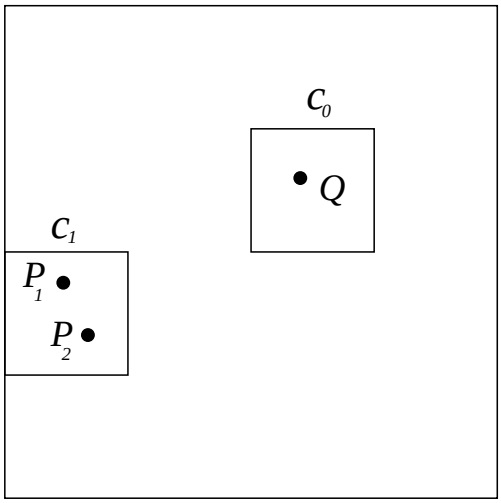


Figure 4.1: Diagram to explain Theorem 2

Theorem 2

For any positive integer $k \leq G$,

$$dist(far(kNN', Q), Q) \leq dist(far(kNN, Q), Q) + d_M$$

and the bound is tight, where d_M is the maximum possible distance of any two points (not necessarily data points) within a cell.

Proof Suppose query point Q is in cell c_0 as Figure 4.1 shows. If all points in kNN are in S', then kNN'=kNN, and $far(kNN', Q) = far(kNN, Q)$, therefore

$$dist(far(kNN', Q), Q) \leq dist(far(kNN, Q), Q) + d_M$$

holds.

If any point in kNN is not in S', say $P_1 \in kNN$ and $P_1 \notin S'$. Suppose $P_1 \in c_1$ (note that c_1 could be the same cell as c_0). $P_1 \notin S'$ means $|S(c_1)| > G$, and then S' must have G points in c_1. Let $P_2 = far(S'(c_1), Q)$, then

$$dist(far(KNN', Q), Q) \leq dist(P_2, Q)$$

$G \geq k$, therefore

$$dist(far(kNN', Q), Q) \leq dist(P_2, Q) \tag{4.1}$$

According to the triangle inequality

$$dist(P_2, Q) \leq dist(P_1, Q) + dist(P_1, P_2) \tag{4.2}$$

P_2 and P_1 are in the same cell, therefore

$$dist(P_1, P_2) \leq d_M \tag{4.3}$$

$P_1 \in kNN$, therefore

$$dist(P_1, Q) \leq dist(far(kNN, Q), Q) \tag{4.4}$$

From inequalities 4.1, 4.2, 4.3 and 4.4, we obtain
$dist(far(kNN', Q), Q) \leq dist(far(kNN, Q), Q) + d_M$

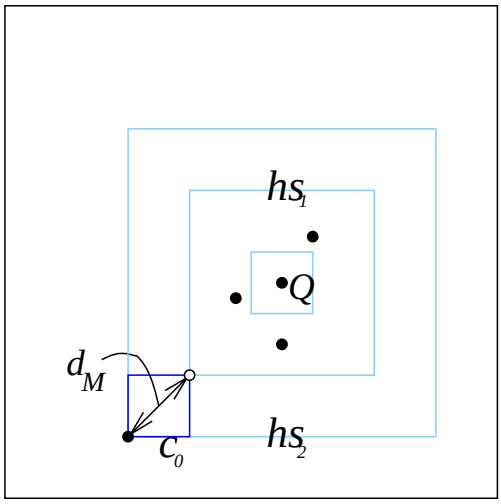


Figure 4.2: A example of the tight bound

Next we show that the bound is tight by constructing a scenario where the equality holds. The scenario is described as follows. Let $k = G$, and the query point Q be at the center of a cell as shown in Figure 4.2. A hypersquare hs_1 centered at Q contains $k-1$ points and hypersquare hs_2 is the minimum hypersquare larger than hs_1. Among the cells outside hs_1 and inside hs_2, only cell c_0 at the corner of hs_2 contains points. As shown in Figure 4.2, let the corner of c_0 which is also a corner of hs_1 be point P_1, and let another corner of c_0 which is also a corner of hs_2 be point P_2. There are $k+1$ cells falling in c_0, one at P_1 shown as a circle (we let this point be within c_0), and the other k points at P_2 shown as a black dot (we let these points be within c_0). The one point at P_1 is discarded and the k points at P_2 are maintained. Since there are less than k points in hs_1, all of the points in hs_1 are maintained. When we search kNN based on the maintained points, we would search to the hypersquare hs_2. The real k-th nearest neighbor is the point at P_1, but since it gets discarded, we would instead get a point at P_2 as the k-th nearest neighbor as the approximate answer. $QP_2 = QP_1 + d_M$, therefore this is the case

where $dist(far(kNN', Q), Q) = dist(far(kNN, Q), Q) + d_M$ □

According to the theorem, if we divide the data space into u^d equal cells and use the above scheme to process the ekNN problem, then e equals d_M. In addition, if the maximum number of points maintained in a cell is G, for any ekNN query where $k \leq G$, the above error bound is guaranteed. For example, if we maintain at most 5 points in a cell, then we can also search for 2NN with an error bounded by $e = d_M$. Note that d_M is determined by the distance function. Without loss of generality, we use the Euclidean distance function in the following discussions and our experimental studies. For the Euclidean metric, the maximum distance of two points within a cell is the length of the diagonal, $\sqrt{d}/u$, and therefore the error bound is $e = \sqrt{d}/u$.

4.4.2 An array-based method

A first method to implement this general scheme would be to organize the data in memory as a big d-dimensional array. Each element of the array represents a cell in the space. We may store at most G points in each cell, so each array element is a structure consisting of G d-dimensional points. Stream data elements are placed in cells on demand as data stream passes by. If there are already G points, we discard one of them based on the discarding policy. Processing of ekNN queries using the array is straightforward. We just need to calculate the borders of the square which encloses the ekNN query sphere and check all the elements within the borders. In what follows, we refer to this method as the *array-based method*.

For the array-based method, we can calculate the memory size needed by the following equation:

$$Mem_{array} = u^d \cdot G \cdot d \cdot sizeof(attribute) \tag{4.5}$$

The array-based method is straightforward, and its processing is simple and fast in terms of memory accesses (reads/writes) and processor time. However, the memory required is proportional to u^d, which is very large when u is large. This static memory allocation strategy can cause excessive memory usage, especially for small error bounds, which implies a large value of u. Real data are often skewed and may be sparse; most cells contain much fewer than G points or even none at all, resulting in poor utilization of the statically allocated memory space. It is obvious that a structure capable of adapting to different data distributions is more desirable.

4.5 The DISC method

To better utilize memory, cells that do not contain data points should not be explicitly maintained as opposed to the array-based method. Even within one cell, the number of points may be different, so space usage is different. This calls for a smart strategy to allocate space to each cell.

Besides the central objective of minimizing memory usage, the method should also provide fast updates and query processing. For the error minimization problem, the method may need some self-adjusting mechanism to achieve smallest error.

As discussed in the previous section, the array-based method needs too much memory despite its fast updates and query processing. Or we can organize the cells by a linked list and dynamically allocate only necessary space for each cell. The memory size problem is solved largely (we still have some extra cost due to the links), but the number of node accesses for update and query processing is linear to the number of points. On average, half the size of the linked list is accessed to locate a point. This is prohibitive for data stream applications.

A third way is to use a dynamic indexing structure such as an R-tree or a B-tree. On one hand, it dynamically allocates space in the unit of a leaf node so as to avoid excessive memory overheads as in the array-based method. This means we don't maintain cells explicitly; we only maintain the footprints of the data, that is, the data necessary to guarantee the error bound for the queries. On the other hand, the index provides fast access to the entries in the nodes. It is not as fast as the array-based method, but typically several node accesses are enough, which is much more efficient than linked lists in terms of updates and query processing. A dynamic index is in fact a compromise of the above two, and therefore it avoids the deficiency of either one.

A straightforward structure for multi-dimensional data is the R-tree or some of its variants. A point is stored as a leaf node entry. Since we need to differentiate between points from different cells, an identifier, *id*, is stored along with each point.

An alternative approach, which we adopt in this thesis, is to employ a B*-tree[3] [101] together with a space-filling curve mechanism. Space-filling curves have been used to linearize multi-dimensional data spaces. Various types of space-filling curves exist in the literature; without loss of generality we adopt the Z-curve [126]. Efficient algorithms to compute Z-values can be found in [126]. Each cell corresponds to a Z-value. Footprints of the data stream are stored in the leaf nodes of a B*-tree using their corresponding cell Z-values as keys. Such an approach is expected to be more efficient than the R-tree scheme for the following reasons. Although a point is the unit of storage, a cell is the unit most of our operations deal with as we will see later in the algorithms. To locate a cell by the Z-value in a B*-tree, for each level of the tree, we only need to compare the search key with one value, since there is no overlap in the Z-values. In an R-tree, we need to compare

[3]We employ the B*-tree for indexing (instead of B$^+$-tree) as its node utilization is about 85% or higher.

the coordinates of the cell with $2d$ values (lower bound and upper bound for each dimension) for each level of the tree and there is overlap between the MBRs of the R-tree, which translates to more node accesses to update and search the R-tree. In addition, since the R-tree stores more information as keys, the fan-out of the R-tree nodes becomes lower and the height larger.

Another advantage of organizing the footprints in the Z-order is that cells can be arranged in a total order while maintaining cell proximity. The R-tree also keeps the points belonging to the same cell spatially close, but it still happens that they scatter in nearby MBRs. In DISC, points in the same cell are always consecutively stored in the leaf nodes. This property facilitates accesses on the cell level and make possible a very fast *merge-cells* operation, which is required for the error minimization problem and described in Section 4.5.2. We will also compare DISC to the R-tree in our experimental study. Since several points may belong to the same cell and have the same key in DISC, our B*-tree is designed to accommodate entries with equal keys. For the R-tree method, we have used the R*-tree [23] variant, which has a higher node utilization (about 73%). Moreover, we have also used the Z-values as the id's of cells for the R*-tree method.

Since we are utilizing space-filling curves, each dimension of the data space is partitioned into a number of intervals equal to an integral power of 2, the same for all dimensions. Let m denote the order of the Z-curve, then $u = 2^m$.

4.5.1 Index creation

We begin by considering the first problem, namely the memory optimization problem for a given error bound e. To guarantee that this error bound is met by our query answers, we calculate the order of the Z-curve m_e according to Theorem 2 as follows.

$$\sqrt{d}/2^{m_e} \leq e$$

Then

$$m_e \geq \log_2(\sqrt{d}/e) \tag{4.6}$$

The larger the value of m_e, the more memory is required; we let m_e be the smallest integer that can satisfy inequality 4.6.

$$m_e = \lceil \log_2(\sqrt{d}/e) \rceil \tag{4.7}$$

Algorithm **Build Index**, shown in Figure 4.4, describes how the index is constructed. In the algorithm, we initialize the value of m to m_e.

Before we discuss the algorithm, let us consider the second optimization problem, namely error minimization given a specific memory size constraint. The basic idea of the algorithm is to adjust the order of the Z-curve, m, to achieve the best accuracy while satisfying the constraint. Our aim is to minimize the error bound e in the ekNN search. Since the larger the value of m, the smaller the error bound e, and the data distribution is not known apriori, we start with a sufficiently large value for m, which can be set from the domain knowledge; the exact value depending on the arithmetic precision we are working with. As data arrive, it may turn out that m is too large and hence memory is exhausted; in this case, we merge small cells into a larger one, discard some points and still maintain at most G points in the larger cell. As a result some memory is freed, and processing of the stream continues. The Z-curve properties enable us to merge cells efficiently. In particular, a Z-value for a cell can be mapped efficiently (using simple bitwise operations) to Z-values corresponding to a curve of different order. For brevity, we omit the details which can be found in [126]. Related properties hold for other curves

as well. Each time we need to perform cell merging, we will combine 2^d adjacent small cells into a larger cell as shown in Figure 4.3, in which cells c_0, c_1, c_2, c_3 are combined to form cell $c_0^{'}$. The larger cell is still square-shaped. After merging the cells, the order of the Z-curve becomes $m - 1$. The index construction algorithm for this case is similar to that for the memory minimization problem; the difference lies in the merging phase. For brevity, we include this phase in the description of algorithm **Build Index**.

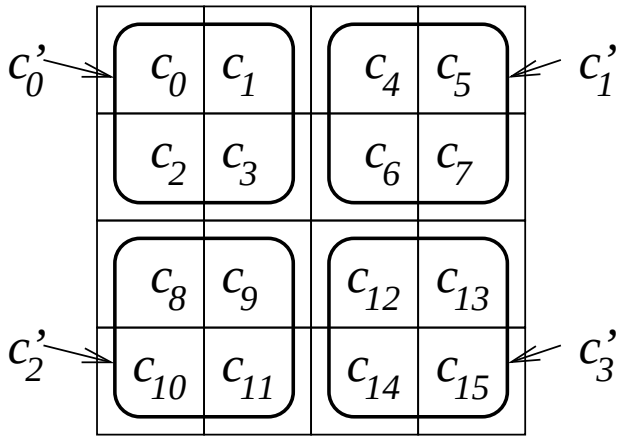


Figure 4.3: Cell Merging

We are now ready to look at algorithm **Build Index** (see Figure 4.4). In line 1, we let $m = m_e$ for the memory optimization problem and let m be a large enough integer for the error minimization problem. In line 5, we should determine which point to discard according to the discarding policy. In our realization of the algorithm, we simply discard the new point P.

In the analysis of Section 4.4.1 we have assumed the data space is normalized to a unit hypercube. This may have difficulty when the maximum and minimum of the data are unknown. In DISC, we would set the maximum/minimum to safely large/small values. For example, we can use 10 times (suppose the data are positive) the observed maximum value in the history as the maximum value of the

Algorithm **Build Index**
1 Initialize m
2 Read data from the stream, denote the point read in as P, calculate the Z-value of P, and we know which cell it belongs to, denote it as c
3 Search the B*-tree and obtain the number of points that also belong to cell c, denote the number as N_c
4 If $N_c < G$
    Insert P to the B*-tree
5 Else
    Among P and the G points in c, discard 1 and keep the other G points in the B*-tree
6 If memory runs out /*This only happens for the error minimization problem*/
    Merge cells and let $m = m - 1$.
    /* The merge cells algorithm is presented in the next subsection. */
7 Go to 2
End Build Index

Figure 4.4: Algorithm **Build Index**

data space. This may result in most of the data gathered at the center of the data space. It will not cause a problem for DISC, because no memory would be wasted for the empty space. And this just shows the advantage of DISC's adaptation to the data distribution.

4.5.2 Algorithms to merge cells

For the error minimization problem, we adopted an adaptive approach that consists of merging 2^d adjacent cells to form a larger one in order to meet the memory constraint. Figure 4.3 shows a 2-dimensional example where the order of the Z-curve m equals 2 before merging. c_0 to c_{15} are the cells before merging. c'_0 to c'_3 are the cells after merging. The subscripts are the Z-values of the cells. Let us denote the larger cell as $M(c)$ if it contains c before merging, then $M(c_0) = M(c_1) = M(c_2) = M(c_3) = c'_0$. In general,

$$M(c_{zv}) = c'_{\lfloor zv/2^d \rfloor} \tag{4.8}$$

where zv is the Z-value of the cell. Let S be a point set. We refer to the cells before merging as *old cells* and to the larger cells after merging as *new cells*. We present two algorithms to merge cells. The first cell merging algorithm applies to any index structure (including DISC and R-tree) that adopts our general scheme, that is, to maintain at most G points in each cell. The second cell merging algorithm is specially designed to exploit DISC's special property that the points are ordered according to the value of the Z-curve (versus the R-tree where points have no ordering). The latter scheme, referred to as the *bulk cell merging* scheme, scans all the leaf nodes once, and hence is expected to be more efficient than the former *general cell merging* algorithm.

Algorithm **General Merge-Cells (GMC)**
1 For i from 0 to $2^{m-1} - 1$
2 Search the index and obtain the number of points in the new cell c'_i, denote the number as $N_{c'_i}$
3 If $N_{c'_i} > G$
Discard $N_{c'_i} - G$ points according to the discarding policy
End General Merge-Cells

Figure 4.5: Algorithm **GMC**

In the first algorithm **General Merge-Cells** (GMC), we examine each new cell in the order of the Z-curve. For each new cell, we search the index and find all points belonging to this cell. If there are at most G points in the cell, we will leave them in the index; otherwise, we delete some of them according to the discarding policy and retain only G points. Algorithm **GMC** is presented in Figure 4.5.

While the GMC algorithm is straightforward and applies to any structure, it is quite expensive since it searches the index 2^{m-1} times.

The second algorithm **Bulk Merge-Cells** (BMC), utilizes the property that the points in the leaf nodes of the B*-tree are ordered according to the Z-values. The 2^d adjacent points which will form a larger cell are adjacent in the leaf nodes, so we only need to scan all the leaf nodes once and merge the points in adjacent 2^d old cells into a new cell. Difference to an R-tree, the entries with close keys in the B*-tree are adjacent to each other, therefore in addition to deleting extra points in a new cell, we also need to move the remaining G points into the same cell. We use a *write cursor* pointing to the place where we would store the next points. Algorithm **BMC** is presented in Figure 4.6. The BMC algorithm requires the cells to be merged into a larger one stored adjacent to each other. This is not the case with the R-tree, therefore we can not use the BMC algorithm on an R-tree.

In line 16 of BMC (Figure 4.6), rebuilding internal nodes based on existing leaf

```
Algorithm Bulk Merge-Cells (BMC)
1   Free all the internal nodes
2   Let ln be the first leaf node. Set write cursor at
    the beginning of ln. Let point set S be empty.
3   While (ln) //when ln is not NULL
4       For each point P in ln
5           If this is the first point in the first leaf node
6               c' = M(c), where c is the cell P belongs to
                S = S ∪ P
7           Else if P ∈ c'
                S = S ∪ P
8           Else if P ∉ c' //We entered the next cell
9               If |S| > G
                    Discard |S| − G points from S
10              Write the points in S to the position of
                write cursor and move the write cursor
                forward accordingly
11              Let S = ∅
12              S = S ∪ P
13              c' = M(c), where c is the cell P belongs to
14      ln = right neighbor of ln
15  Free all the leaf nodes after the write cursor
16  Rebuild internal nodes of the B*-tree based on the
        leaf nodes
End Bulk Merge-Cells
```

Figure 4.6: Algorithm **BMC**

nodes is very similar to bulk loading of a B^+-tree. We do not discuss the details here for brevity.

Comparing the two merging algorithms, we note that BMC scans the leaf nodes only once, while GMC entails many searches and updates for each new cell. So BMC is expected to be faster than GMC. We will compare them in the experiments.

We note that the merge-cells operation is expensive compared to other operations, especially when the memory is large. As it may take a while to reduce the order of the curve by 1, stream processing may be disrupted. Fortunately, it is not necessary to finish merging all cells at once. Cell merging can be performed incre-

mentally. When the system load is heavy, say, there is a burst of incoming data or many queries, we stop the merge operation at the current new cell we are working on and record this stop position. If the update or the query accesses the points before that stop position, we process them assuming the order of the Z-curve to be $m-1$; if data belonging to cells after the stop position are required, we process them assuming the order of the Z-curve to be m. If the search involves more than one cell, some of which may be old and some are new, query processing is performed assuming the order of the Z-curve in the new cells , $m-1$. Old cells that are accessed in the search are temporarily combined to form larger new cells, but they are in fact merged later as cell merging resumes. The error bound returned with the query results in this case, is the one associated with the order $m-1$. Both GMC and BMC can be performed incrementally. However, it is important to complete the operation fast.

4.5.3 Query processing

In this section, we present the ekNN query processing algorithm. The input of an ekNN query is a query point Q and an integer k. As analyzed in Section 4.4.1, an ekNN query on the original data set, is a kNN query on footprints of the data. Figure 4.7 gives detailed steps of the algorithm **KNN search**, which we use to perform the kNN search on the footprints of the data.

Let c_Q be the cell Q belongs to. Denote as Q' the center point of c_Q and as W a query window which is a d-dimensional interval $[wl_1, wh_1]$, $[wl_2, wh_2]$, ..., $[wl_d, wh_d]$. First, we initiate a square-shaped window query centered at Q' with an initial side length of $1/u$ and then increase it gradually. We maintain a k candidate answer set which always contains the nearest k points to Q within the current query window. The function $near(W, Q)$ returns the distance between Q and W's nearest

Algorithm **KNN Search**
1 $S = \emptyset$
2 For i from 1 to d
  $wl_i = q'_i - \frac{1}{2u}$; $wh_i = q'_i + \frac{1}{2u}$
3 WindowQuery(W). From the points in W, get the k nearest points to Q and put them in S; if there are less than k points in W, put all of them in S.
4 if $|S| < k$ or $near(W, Q) < far(S, Q)$
5 for i from 1 to d
   $wl_i = wl_i - \frac{1}{u}$; $wh_i = wh_i + \frac{1}{u}$
6 Go to 3
7 return S
End KNN Search

Figure 4.7: Algorithm **KNN Search**

side (or nearest hyperplain when we have more dimensions) to Q. The algorithm terminates when $near(W, Q)$ is larger than or equal to the k-th nearest point in the candidate answer set. All the points outside the query window are farther from Q than $near(W, Q)$. So when the algorithm terminates, the farthest point in the candidate set is the k-th nearest point to Q among all the points inside and outside the query window. To avoid searching cells which are already visited in the previous iteration, we maintain a list of addresses of the B*-tree leaf nodes visited. WindowQuery(W) is a function to retrieve all the points in window query W. In DISC, each leaf node of the B*-tree corresponds to a continuous segment of the Z-curve. An efficient window query algorithm proposed in [22] accesses only those nodes with their corresponding Z-curve segments intersecting the query window. We use this algorithm for our WindowQuery() function. In our implementation, we have used $1/u$ as the initial side length of the window query. Sometimes, this may not be optimal if the final query window is large. We may consider estimating the kNN distance from some query history to optimize this initial query window.

Next we show an example of the Algorithm KNN Search. Suppose G=2, and we

are searching 2NN. Suppose the data we have maintained are as shown in Figure 4.8, where the black dots are data points and the circle is the query point. We can see u=8, and since G=2, we maintain at most 2 points in each cell. Figure 4.9

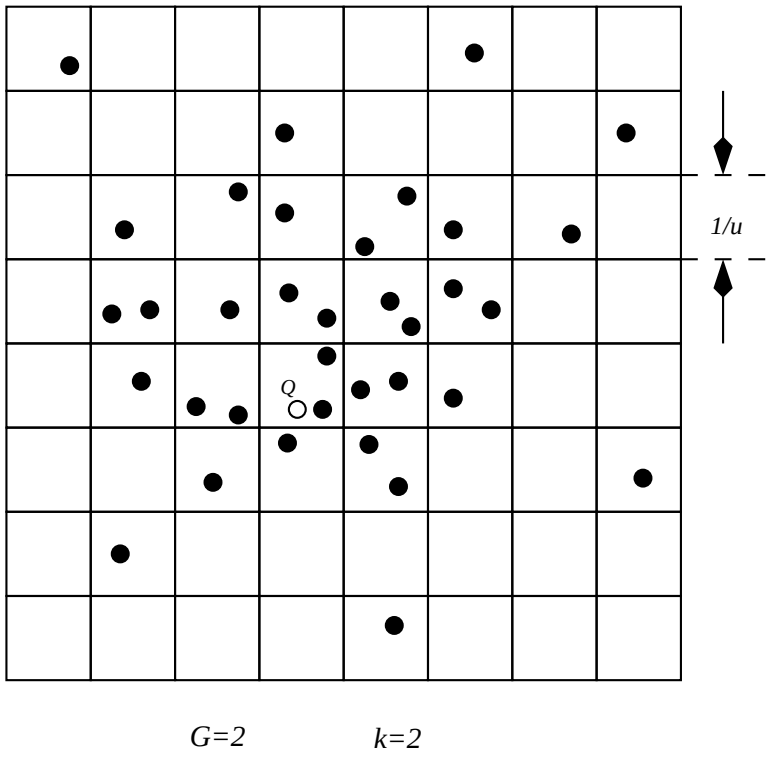


Figure 4.8: An example of KNN search

shows a closer look at the nine cells surrounding Q. Q is in the cell $ABCD$(we represent the cell by its four corners here). To perform the KNN search, we first initiate the window query $ABCD$ (we represent the window by its four corners here), which is centered at the center of the cell $ABCD$ and with the side length of $1/u$. We denote this window query by W. Within W, we find the two nearest candidates to Q, that is, points X and Y. Among the four sides of W, the nearest side to Q is AB, so $near(W, Q)$ returns the distance between Q and the side AB. The 2nd nearest candidate to Q is X, and we can see QX is larger than $near(W, Q)$, therefore we enlarge W by adding a lane of cells around it. By this means the side length of W becomes $3/u$ and now W becomes the square $HIJK$. We still obtain

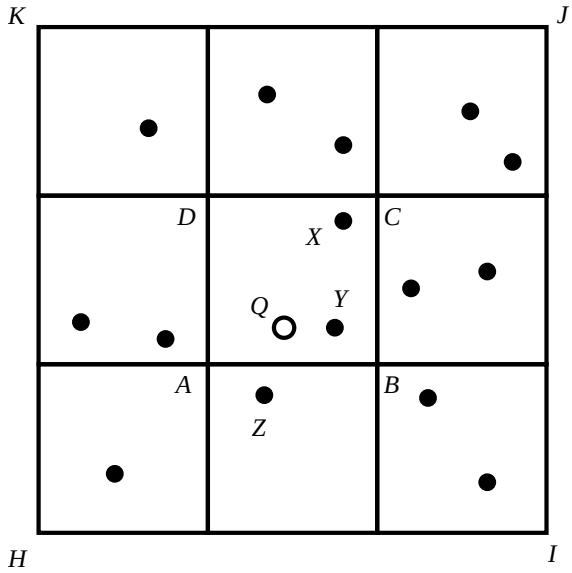


Figure 4.9: An example of KNN search (close look)

the nearest two data points to Q within W, which are Z and Y. Now $near(W, Q)$ is the distance between Q and the side HI. It is larger than the distance between Q and its 2nd nearest candidate Z, therefore the search terminates and we get the approximate 2NN, which are Y and Z.

For continuous ekNN queries, we maintain the $ekNN$ set as follows. Let W_s be the smallest window centered at Q' that contains all the points in $ekNN$. When a new data point P comes and $P \in W_s$, we may need to discard some points according to the discarding policy (for example, in the sliding window query discussed in the next subsection, points older than T_{sw} are discarded). If a point in $ekNN$ is discarded, the $ekNN$ set would have fewer than k points at the moment. After discarding, there are 3 cases to consider: 1) *There are still k points in ekNN*. If P is nearer to Q than the farthest point in $ekNN$, then P will replace the farthest point; otherwise $ekNN$ is kept unchanged. 2) *There are fewer than k points in ekNN and P is nearer to Q than the farthest point in ekNN before discarding.* We add P to $ekNN$ and start kNN search as in the one-time search algorithm, but

we set the initial search window as W_s. 3) *There are fewer than k points in ekNN and P is not nearer to Q than the farthest point in ekNN before discarding.* We just start kNN search as in the one-time search algorithm with the initial search window W_s. The proof of the above algorithm is straightforward and we omit it here due to the limitation of space.

4.6 Processing sliding window queries by DISC

All the DISC algorithms discussed so far work on the entire data stream. In certain applications, recent stream data are of greater interest. In this section, we study how the algorithms can be modified to process sliding window ekNN queries. Specifically, we hope to identify the $ekNN$ of a query point Q among all data stream elements arriving in the last T_{sw} time units.

DISC is capable of supporting such sliding window ekNN queries by simply employing a time-based discarding policy. Let t be the current time. Assume that each arriving stream record is tagged with a timestamp signifying its arrival time. Algorithm **Build Index** can be modified for the sliding window model as follows. When inserting a point P to a cell c, we first check the timestamp of existing points in c. We then delete the stale points, that is, the points that arrived earlier than $t - T_{sw}$. Finally, we insert P. For algorithm **KNN Search**, we only place points arriving later than $t - T_{sw}$ to the candidate answer set S. At any time, if we encounter stale points (during index building or kNN searching), we delete them immediately. Such modifications enable DISC to answer sliding window ekNN queries correctly. However, if there are records in the index that are older than $t - T_{sw}$, but no incoming record is added to the cells the old ones belong to, such stale data will remain in the index, occupy space and affect space utilization.

To avoid this, we need an operation to eliminate such stale data. This can be accomplished by scanning all the points and deleting stale data from the index. However, such an operation is expected to be time consuming. Again, like the cell merging process, this stale data elimination process can be done incrementally. There exists a tradeoff between memory utilization and processing capability. To achieve best accuracy when addressing the error minimization optimization problem in the sliding window model, we eliminate stale data before each call to the **Merge-Cell** operations. This way, some additional space becomes available and it may be possible to avoid cell merging.

We should take care when processing continuous ekNN queries over sliding windows. Even no new points come in W_s, there still could be stale data due to time. Therefore, in this case we need to check whether the set contains stale data in each time unit to guarantee the correctness of the $ekNN$ set. Or if the $ekNN$ answers are not requested all the time, we can check for stale data when we retrieve answers from the maintained $ekNN$ set. If there were stale records, we discard them and invoke the kNN search on the footprints with the initial search window W_s. This is still much faster than invoking the search from scratch.

4.7 Deploying DISC in Gigascope

To deploy DISC in Gigascope, we need to decompose the processing into LFTA and HFTA components. Recall that the LFTA should filter out most traffic to reduce the data volume passed to the HFTA, while those complex operations can be performed by the HFTA. In case of DISC, we would let the indexing algorithm be run at the LFTA and the kNN search algorithm be run at the HFTA which are described in detail in the following.

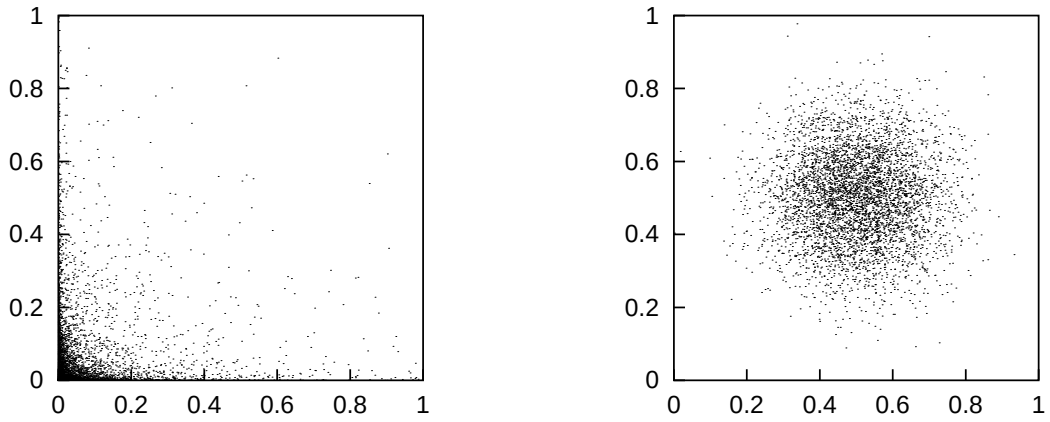
During an epoch (a tumbling time window), algorithm *Build Index* is run at LFTA to maintain the footprints of this epoch. Let n be the number of records that arrive in this epoch. We estimate the computation needed at LFTA as follows. In algorithm *Build Index*, step 2 calculate the Z-value of a record, which involves some bit operations and is very efficient. Step 3 searches the B*-tree and count the number of points belonging to the cell that the new point belong to, which takes $O(\log(n))$ time and $O(1)$ time, respectively. Then step 4 and 5 insert the new point and may discard one point, which takes constant time when no node overflow happens. Therefore the whole procedure takes $O(\log(n))$ time, which can be efficiently done at LFTA. If we are dealing with the error minimization problem, we may need to merge cells (step 6), which is a rather expensive operation. We can take two measures to alleviate this problem. First, as discussed in Section 4.5.2, we can perform the cell merge incrementally. Second, we try to use a good choice of the order of the Z-curve, m, so that cell merge happens rarely. This good choice can be made according to the previous m value, since each epoch is a short time (e.g., 1 minute) and data distribution of adjacent epochs should not change greatly. This may result in an underestimate of m and hence a larger error bound than what could be achieved if choosing the right m, but we would expect the underestimation to be small due to the time locality. If we have chosen a small m according to the previous epoch, we need to know when to increase m. We use the memory usage to decide this. If the memory usage is below some threshold, the current m value may be too small and we increase it by 1 in the next epoch.

At the end of the current epoch, everything maintained in the B*-tree is passed to the HFTA. As we only need the maintained data in the leaf nodes of the B*-tree to answer the queries, we only need to copy all the leaf nodes to the HFTA. We always remember the address of the first leaf node of the B*-tree while building

it. When we copy the leaf nodes, we first copy the first leaf node; then we follow the pointer from one leaf node to the next leaf node to get all the other leaf nodes repeatedly. Because we would build a new B*-tree in the next epoch, we have to free the space used by the B*-tree of this epoch. To free them one by one following the downward pointers may be too expensive. We can use a memory object for the B*-tree here, which can free all the memory allocated to the B*-tree at once. After the B*-tree (which contains the "footprints" of the data stream in this epoch) is copied to the HFTA, we can run algorithm *KNN Search* to process queries over the data. As the data volume has been greatly reduced, the HFTA should have enough computation power to process the queries. The time period we can pose query upon is any epoch before the current one.

4.8 Experiments

In this section, we present the results of an extensive experimental study using DISC. While we have implemented and worked with an in-memory version of DISC, DISC is also applicable for secondary storage. The experiments are performed on a desktop computer with Pentium IV 2.6G CPU and 1G RAM. In our study we employed both synthetic and real data sets. We generated exponentially and normally distributed data sets of varying dimensionality. Figure 4.10 shows 2-dimensional images of the two data distributions. The real data set is still the one obtained by *tcpdump* on a network server of AT&T as described in Section 3.8.1. We have extracted certain fields (source IP, destination IP, source Port and destination Port) from all the records to form data sets of different dimensionalities. Such logs were aggregated temporally and ekNN queries were issued using the total number of bytes and associated packet rate attributes. All the data are normalized



(a)Exponential distribution (b) Normal distribution

Figure 4.10: Data distributions

in the range of [0,1]. By default, we let G equal 20 and we set the order of the Z-curve as 10, which implies an error bound of 0.00138 in a 2-dimensional space. For the in-memory B*-tree, we used a default node size of 1024 bytes. First, we focus our experiments on a 2-dimensional space examining DISC's memory usage and accuracy and compare the two cell merging algorithms. Then we examine the behavior of DISC on higher dimensions.

4.8.1 Memory usage of DISC

In a first series of experiments, we study the memory usage of DISC as data stream passes by. No existing structures or algorithms were proposed to process (approximate) kNN queries over streams as discussed in the related work 2.5.2. Therefore we would compare DISC with the R*-tree [23] indexing under our general scheme to see which one is more efficient. Figures 4.11 to 4.13 present the memory used by DISC and the R*-tree as a function of the observed data stream size (in number of points) on 2-dimensional exponentially distributed and normally distributed data sets and the real data set.

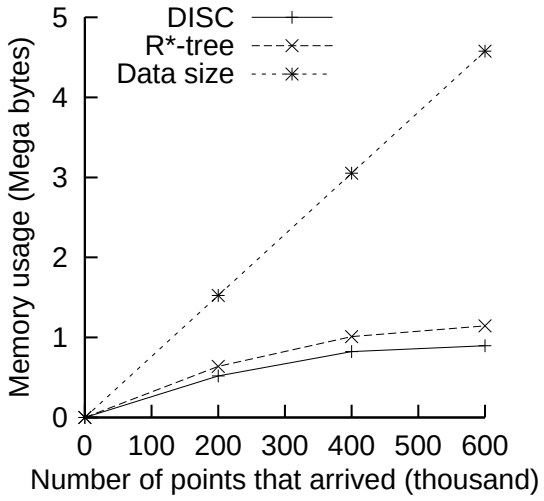


Figure 4.11: Memory Usage of DISC: Exponentially distributed data

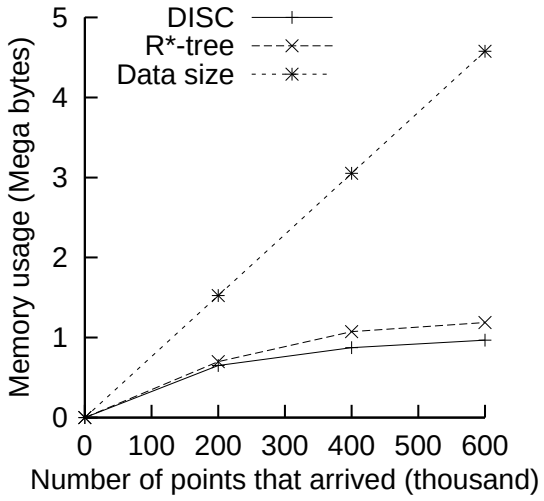


Figure 4.12: Memory Usage of DISC: Normally distributed data

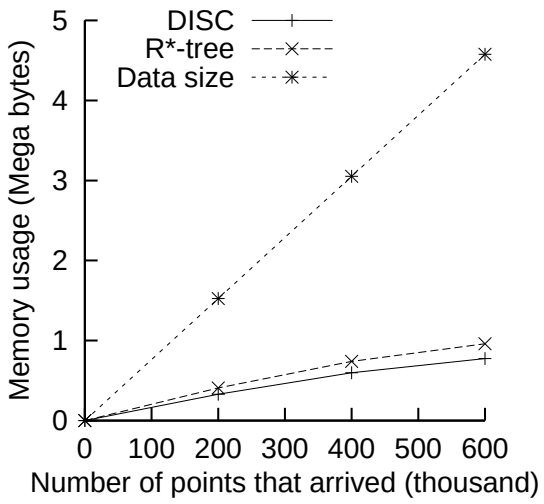


Figure 4.13: Memory Usage of DISC: Netflow data

As the data continually arrive and their cumulative size increases, the memory usage of DISC increases also at first, but the increase slows down soon as more data arrive. At first, all the cells are empty and therefore all of the data are stored as footprints. But as more data come in, more and more cells become full (containing up to G points) so that memory usage almost remains constant. When 600K data points have arrived, the memory used by DISC is 10∼25% of the size of the data. Using Equation 4.5, we calculate, for this setting that the amount of memory needed for the array based method is 41943040 bytes, which is more than 8 times the data size. These results show that DISC does adapt to different data distributions because it only stores necessary cells and in each cell, necessary points to guarantee the error bound, while the array-based method suffer from the static memory allocation greatly. The huge space cost of the array-based method make it not applicable in stream applications. In all the following experiments, the array-based method always needs at least several times the space of the original data to operate, therefore we will not compare DISC with it again. We also observe that the memory usage of the R*-tree is always a little higher than DISC. This is

because while the R*-tree also allocates space only to the points requiring explicit storage, the leaf node utilization rate of the R*-tree (about 73%) is lower than that of the B*-tree (about 85%).

To see how how some parameters such as the node size of the B*-tree or the R*-tree and G affect the memory usage of DISC, we varied the node size and G respectively while maintaining other parameters constant. The memory usage for different node sizes when 600K netflow data points have arrived is presented in Figure 4.14. The memory usage decreases slightly as the node size increases. This is because for larger nodes, higher node utilization rate can be achieved because we have less internal nodes. However, the effect of node size is small compared to the total data size. In the remainder of the experiments, we used 1024 as the default node size.

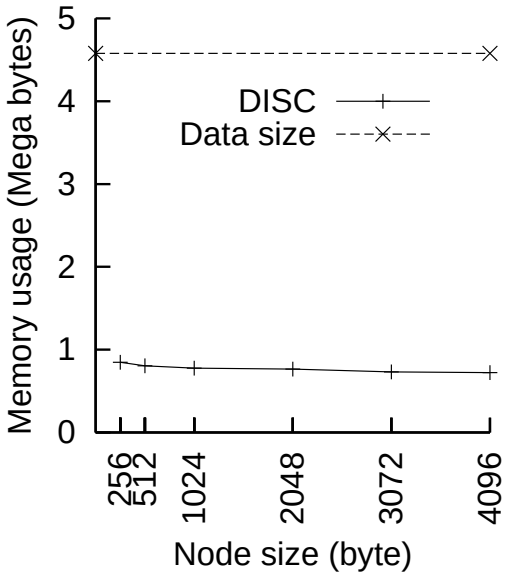


Figure 4.14: Effect of Node Size

Figure 4.15 presents memory usage as a function of G for netflow data. Memory usage increases as G increases in an almost linear fashion, according to expectation.

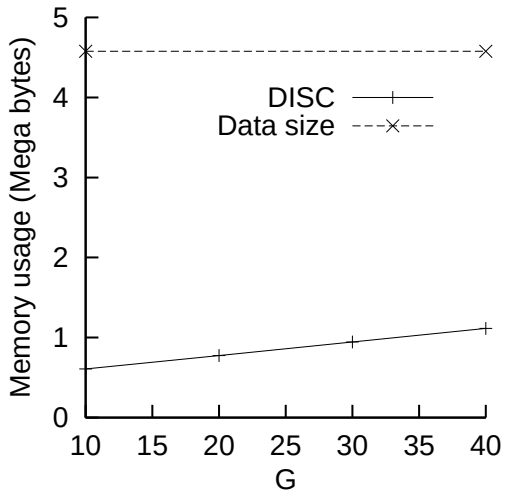


Figure 4.15: Effect of G

This demonstrates that DISC handles the allocation of the available memory in a space efficient fashion. Experiments over the synthetic data sets show similar behavior. It is expected that the memory usage of DISC would reach the size of the stream data if G is too large, but it will not be too much beyond the stream size. In the worst case that G is infinite, all the stream data are maintained. The memory usage of DISC would be a little more than the stream size considering the space utilization of the B*-tree, but it will not grow excessively as the array-based method, which may use many times the size of the stream. In many applications, tens of nearest neighbors are enough and G can be determined from domain knowledge or query history. In these cases, DISC is still quite useful. In the remainder of the experiments, we have used 20 as the default value of G, which is a reasonable number used in data mining applications.

4.8.2 Accuracy of DISC

While DISC can guarantee a theoretical error bound of e, we run experiments to assess the actual errors. We generated 200 queries following the same distribution as the data. We scan the original data to find the exact kNN to each query and also employ DISC to identify the $ekNN$. We then compare the exact kNN distance and the ekNN distance to obtain the actual error. The results are presented as averages over the 200 queries in Figure 4.16. The figure shows the comparison between the error bound e and the actual error for the (exponentially distributed, normally distributed and netflow) data streams as the data arrive. We observe that the average actual errors are less than one third of the theoretical error bound. These results demonstrate the accuracy of DISC. In all our experiments, we have also observed that the maximum actual errors are smaller than the theoretical error bounds, which further confirms the effectiveness of DISC.

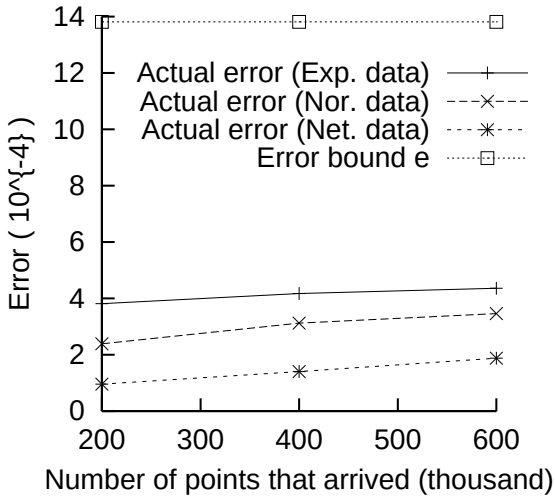


Figure 4.16: Accuracy vs. Arrived Data Size

In our next experiment we evaluate the impact of the order of the space-filling curve on our scheme. We vary the order of the Z-curve from 8 to 11 and see how it

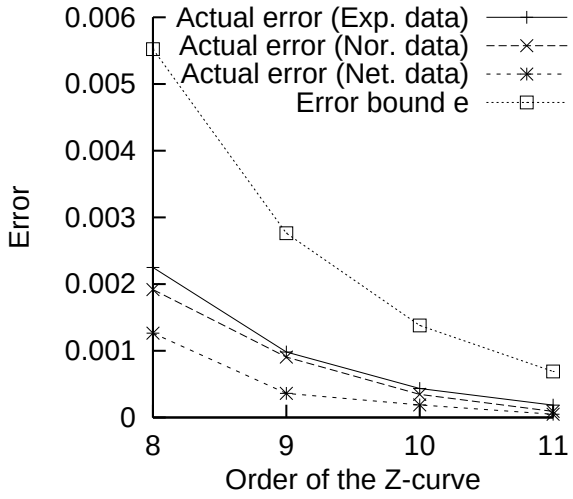


Figure 4.17: Accuracy vs. Order of the Z-curve

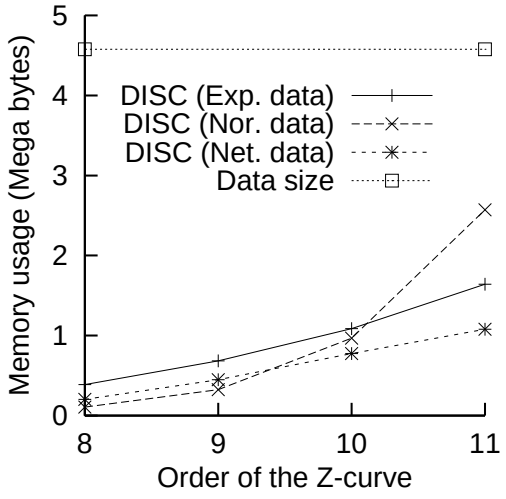


Figure 4.18: Memory Usage vs. Order of the Z-curve

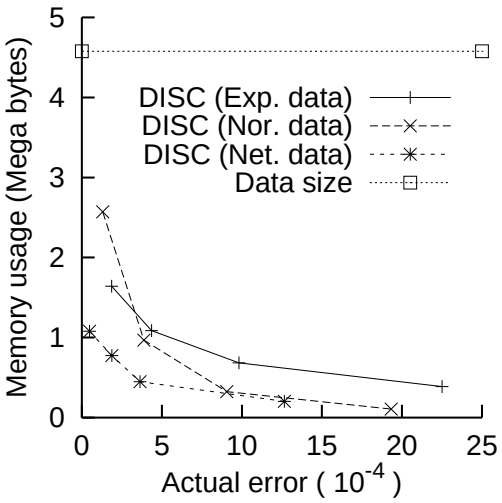


Figure 4.19: Memory Usage vs. Accuracy

affects the actual errors and the memory usage. The error bound and actual errors for different orders of the Z-curve are shown in Figure 4.17. The memory usage for different orders of the Z-curve are shown in Figure 4.18. As the Z-curve order increases, the error bound e and the actual errors decrease, while the actual errors are always much smaller than e. On the other hand, as the Z-curve order increases, the memory usage also increases. This is because we have more cells and we need maintain more points in total.

To see the relationship between the memory usage and the accuracy, we present for different error bounds, their corresponding memory usage versus the corresponding actual errors when 600K data points have arrived in Figure 4.19. The memory usage increases as actual errors decrease. This shows that DISC can easily trade error for memory space by suitably setting the order of the Z-curve.

To show that the above absolute errors are reasonably small, we also present the relative kNN distance errors they correspond to in Figure 4.20. For the netflow data, $ekNN$ has a relative error of 5% when the memory usage is about 1MB, which is less than 1/4 of the original data size. Even when the memory usage is

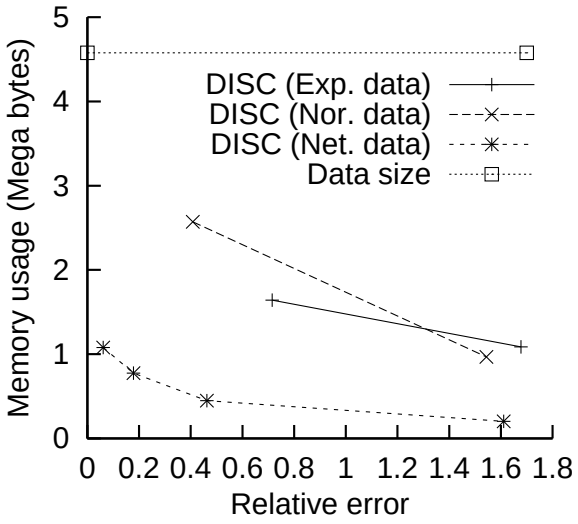


Figure 4.20: Memory Usage vs. Relative Error

only 200KB, which is less than 5% of the original data size, $ekNN$ has a relative error of 1.6. For the exponentially and normally distributed data sets, $ekNN$ also has small relative errors while use much less memory size than the data size.

4.8.3 GMC vs. BMC

In this experiment, we evaluate the two merge-cell algorithms. We have implemented the GMC algorithm for both DISC and the R*-tree. We also implemented the BMC algorithm, which only applies to DISC. We trigger the Merge-cell operation when 200K, 400K and 600K data points have arrived. (In fact, the merge-cell operation should be invoked in the case of the error minimization problem only when available memory runs out. Here we call it explicitly to observe its behavior under varying data size.) We calculate the number of node accesses and response time as measures of their performance. The results for the real data set are shown in Figure 4.21 and 4.22. We can see that under the DISC scheme, GMC needs much more node accesses than BMC (about 300 to 600 times). This is because

in GMC, we need to traverse the tree for each new cell. To support a reasonably small error bound, usually the order of the Z-curve is large, which is 10 in our experiments. So we have to traverse the tree $2^{9\times 2} = 262144$ times, and each traversal incurs several node accesses (descend the tree and locate the points to the new cell). While in BMC, we only scan all the leaf nodes once (which ranges from hundreds to a few thousand in our experiments). GMC for the R*-tree turns out to be marginally better than its DISC counterpart. This is because in the R*-tree, when some points are discarded from a cell, we are not required to move the remaining points together while in the B*-tree this is necessary. The response time has similar trend. In the experiments, the GMC algorithm takes several minutes to finish while the BMC algorithm takes only 1 or 2 seconds. So clearly, only the BMC algorithm is applicable in practice. This is an additional reason that makes DISC preferable over other approaches. Despite its efficiency, we can still perform incremental cell merging with BMC as described in Section 4.5.2 in case the memory is very large and the system load is heavy.

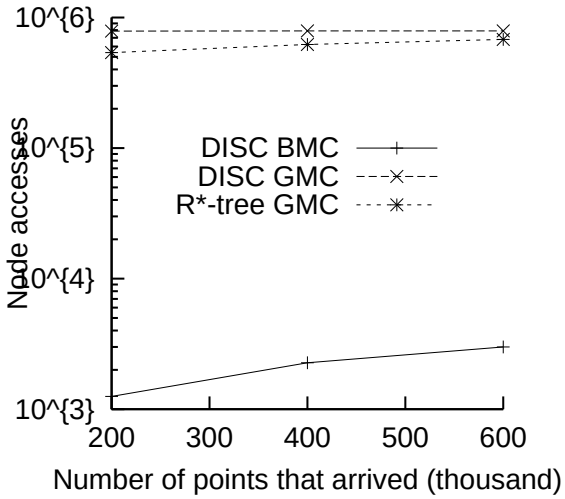


Figure 4.21: Node accesses of GMC and BMC

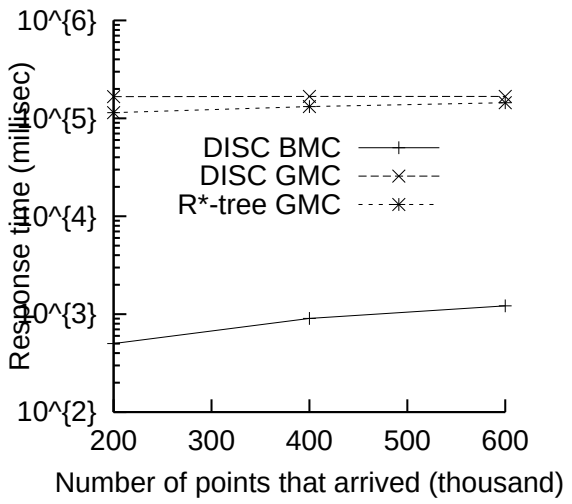


Figure 4.22: Response time of GMC and BMC

4.8.4 Updates and query processing

To evaluate the update and query processing performance of DISC, we measured the number of node accesses of updates, one-time ekNN query and continuous ekNN query processing for DISC and the R*-tree[4]. The cost of a continuous ekNN query consists of the cost of the initial one-time ekNN query and the cost of maintaining the $ekNN$ set continuously. The maintenance cost is the possible search cost when a point in W_s arrives as described in the continuous ekNN algorithm. Specifically, maintaining the $ekNN$ set involves possible kNN search during the insertion of new points. Therefore, the update cost with continuous ekNN queries running is expected to be higher than the usual update cost.

In our experiments, the query costs of the one-time ekNN queries are averaged from 200 queries which follow the same distribution as the real data set. For continuous ekNN queries, we use the same queries but run 10 continuous queries simultaneously each time. The update costs are averaged from the 600K points

[4]Usually we don't have updates in network data streams. DISC is applicable to general data streams where we may have updates in other applications such as moving object data, therefore we also evaluate the update operation for completeness.

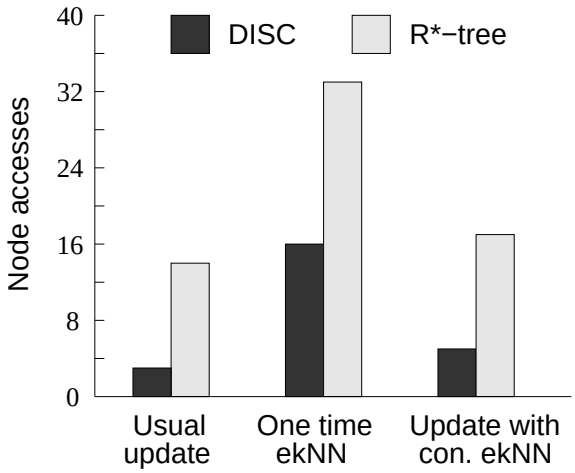


Figure 4.23: Update and Query Cost

inserted. G is still set as 20. The results on the netflow data set are shown in Figure 4.23. First we observe that for all the operations, DISC has much lower node access cost than the R*-tree. The reason is that in DISC we only store the Z-value as the key, but in the R*-tree we need to store $2d$ values as keys so the fan-out of the tree is lower and hence the height of the tree larger. In addition, there are overlaps between the MBRs of the R*-tree, which also incurs more node accesses. We also notice that the query processing cost is not large in terms of node accesses. This is largely due to the Z-order keeping the proximity of the spatial points and the efficient WindowQuery() algorithm. In addition, in two-dimensional space, the points are dense. For skewed data, most points are clustered at a relatively small region and so do the queries. So for most queries, after locating the cell the query belongs to, we need only a few number of node accesses to retrieve near points. The cost of the continuous ekNN is mainly expressed in the additional part of the update cost. We can see that, update with continuous ekNN queries running costs a little more than the usual update, but the increase is not great. Therefore, the continuous ekNN query processing is still quite efficient.

4.8.5 DISC on data sets of other dimensions

We study the behavior of DISC when the number of attributes of interest in the record increases (and as a result the dimensionality of stream elements increases as well). Figure 4.24. shows the memory usage of DISC as 3-dimensional synthetic data stream passes by. We still set G as 20 and the order of the Z-curve as 10, which corresponds to an error bound of 0.00169 in 3-dimensional space.

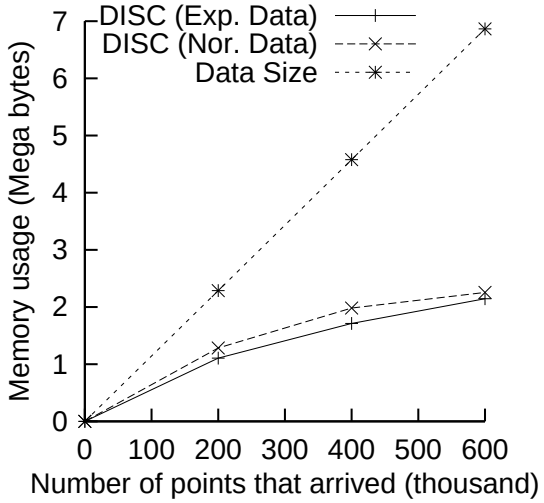


Figure 4.24: Memory usage of DISC on 3D data sets

The results are similar to those of 2-dimensional data (compare with Figures 4.11 and 4.12). DISC uses much less memory compared to the original data size and its memory usage does not increase significantly as the number of arriving data elements increases. As dimensionality increases, DISC tends to occupy more memory than in the 2-dimensional case; this is expected as in higher dimensions, points become relatively sparse and therefore distributed in more cells, which have to be maintained. Similar to the experiments on 2-dimensional data, the average actual errors are much lower than the error bounds as shown in Figure 4.25.

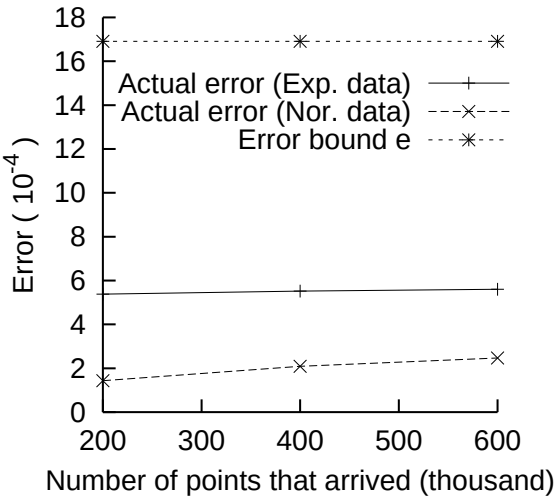


Figure 4.25: Accuracy of DISC on 3D data sets

4.9 Summary

In this chapter, we attacked the problem of efficiently processing nearest neighbor queries over network data streams. We approached this problem by introducing the *e-approximate k nearest neighbor (ekNN) query*, which specifies the error bound as an absolute error e. We proposed a framework that makes it possible to reduce the information needed to answer ekNN queries with a guaranteed error bound. Specifically, we divide the data space in to cells and only need to maintain at most G records in each cell in order to guarantee some error bound, where G is a user defined parameter. Under this framework, we further proposed a novel technique called aDaptive Indexing on Streams by space-filling Curves (DISC), under our proposed framework, to efficiently maintain data and process queries from the maintained data. DISC has efficient insertion, deletion and kNN search operations. We also proposed an efficient merge-cell algorithm for DISC, which is essential to adjust DISC to the data distribution of the data stream. By DISC, we attain two optimization goals: *memory optimization for a given error bound*,

and *error minimization for a given memory size*. We also discussed sliding window query processing and how to deploy DISC in the Gigascope DSMS. Finally, we presented our experimental study, which shows that DISC can achieve the goal of memory optimization or error minimization while having efficient updates and query processing.

CHAPTER 5

Conclusions and Future Work

5.1 Conclusions

In recent years, the emergence of a class of new applications, where the data input is of a very large volume (possibly infinite) and arrives at the system at a very high speed, has come to the attention of the database research community. Applications are real, such as sensor networks, web based services, and especially network monitoring tasks. Two basic network monitoring tasks, traffic management and security, call for efficient processing of two types of queries: aggregate queries and nearest neighbor queries. In this thesis, we have tackled these two types of queries under the data stream model. We summarize our contributions and results on these two problems below.

For aggregate queries, our goal has been to achieve minimum overall cost when multiple aggregate queries are given. The cost reduction relies on sharing computation among queries. We have based our study on the two-level (LFTA and HFTA)

query processing architecture of the Gigascope DSMS. Our first contribution to the problem is the insight that when computing multiple aggregate queries that differ only in their grouping attributes, it is often beneficial to additionally compute and maintain *phantoms* at the LFTA. Phantoms are fine-granularity aggregate queries that, while not of interest to the user, allow for shared computation between multiple aggregate queries over a high speed data stream. Next, we have formulated the multiple aggregation problem. Specifically, user queries and phantoms that could be beneficial are organized as a feeding graph. We need to choose phantoms from this feeding graph to form a configuration, which consists of relations to be instantiated in the LFTA. For a given configuration, we have formulated the intra-epoch and end-of-epoch costs. The problem is to achieve minimum intra-epoch cost (the cost for simplicity) while satisfying the constraint imposed by the end-of-epoch cost. As Gigascope adopts a subaggregate and superaggregate paradigm in data cube computation, and uses hash tables to maintain the computation for aggregations, how to allocate space to the hash tables of the instantiated relations is also a problem since the hash table size is flexible. Accordingly, we have identified two sub-problems for the multiple aggregation problem here: phantom choosing and space allocation. We have proven that the phantom choosing problem is NP-complete; hence, we have chosen to use a greedy algorithm. The view selection problem shares certain similarities with our problem but also exhibits differences. Adapting the greedy algorithm used in the view selection problem to our problem has proven cumbersome and impractical, and we have proposed a novel greedy algorithm that can fully utilize the given memory space. However, a prerequisite of our proposed greedy algorithm is an accurate estimation of the collision rate of the hash table, given the number of groups of the relation and the hash table size. We have derived such a collision rate model and validated its accuracy by both

synthetic and real data streams. Next, we have also attacked the space allocation problem through an in-depth mathematical analysis, and found that configurations with no phantom or with only one phantom feeding all queries can be solved algebraically ("solvable"), but all other cases cannot ("unsolvable"). Therefore, we have further proposed heuristics to process unsolvable configurations based on the analysis of the solvable cases and the partial results we can get from the unsolvable ones.

Through an extensive experimental study using both synthetic and real data sets, we have shown that our proposed heuristic results in near optimal configurations (within 15-20% most of the time). Overall, our algorithm always outperforms the algorithm adapted from the view selection problem, and through maintaining phantoms, we achieve performance gain more than an order of magnitude.

On nearest neighbor queries, we have observed that many applications can tolerate an absolute error bound of the answers. Thus we have introduced the *e-approximate k nearest neighbor (ekNN) query*, a new type of approximate nearest neighbor queries that specifies the error bound as an absolute value instead of a relative one. We have focused our study on this type of queries over data streams in this thesis. Towards efficient processing of ekNN queries over data streams, we have first proposed a framework which partitions the data space into regular cells and maintains in each cell at most G data points. We have proven that processing kNN queries based on the maintained data points (called "footprints") for any $k \leq G$ can guarantee answers with error less than the length of the diagonal of one cell. However, implementing this framework in a straightforward manner does not generate good performance. We have further proposed using space-filling curves (the Z-curve is used in our study without loss of generality) to order the cells, and then a B*-tree to maintain the points which are clustered according to the Z-values

of the cells they belong to. The whole scheme, called aDaptive Indexing on Streams by space-filling Curves (DISC), has several virtues such as efficient maintenance of data and query processing, and more importantly, it can adapt itself to minimize memory usage or error through adjusting the order of the Z-curve (that is, size of the cells).

We have also conducted extensive experiments using both synthetic and real data sets to study the performance of DISC. DISC outperforms its competitor, an R^*-tree based implementation, in both memory usage and query processing. Moreover, its *bulk merge cell* operation for adjusting the order of the Z-curve is by far more efficient than the alternatives, which makes the technique feasible for data stream applications.

5.2 Future work

There are several directions we could extend the work of multiple aggregations. Currently, our technique only works for traditional aggregate queries such as SUM, COUNT, AVG, etc. However, there is more interest on holistic aggregates (quantiles and heavy hitters). An immediate question, therefore, is whether we can process these holistic aggregates via the same idea. Another limitation of our technique is that it only provides sharing among queries that are grouped in tumbling windows of the same size. One extension is to apply our techniques to queries that vary not only in grouping relations, but also time window sizes. Our technique relies on the two-level query processing architecture of Gigascope. We may also consider whether it can be applied to more general DSMSs, and be modified to work in a sliding window query model.

On the DISC technique for ekNN queries, one improvement we could make is to

vary the cell sizes for different locations of the data space. Locations with denser data points could have smaller cells and locations with sparser data points could have larger cells. By this means, we are able to meet various error requirements of the user while further reducing space usage.

In terms of practice, we are considering deploying both techniques in a real DSMS. This raises important research questions at the system level, in terms of interactions of such algorithms with the current system, and issues related to adaptivity and frequency of execution, etc.

BIBLIOGRAPHY

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The design of the borealis stream processing engine. In *Conference on Innovative Data Systems Research (CIDR)*, pages 277–289, Asilomar, USA, 2005.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. F. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. B. Zdonik. Aurora: A data stream management system. In *ACM International Conference on Management of Data (SIGMOD)*, page 666, San Diego, USA, 2003.

[3] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional samples for approximate answering of group-by queries. In *ACM International Conference on Management of Data (SIGMOD)*, pages 487–498, Dallas, USA, 2000.

[4] N. Alon, P. Gibbons, Y. Matias, and M. Szegedy. Tracking join and self-join sizes in limited storage. In *ACM SIGACT-SIGMOD-SIGART Symposium*

on Principles of Database Systems (PODS), pages 10–20, Philadephia, USA, 1999.

[5] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *ACM Symposium on Theory of Computing (STOC)*, pages 20–29, Philadephia, USA, 1996.

[6] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. Stream: The stanford stream data manager. In *ACM International Conference on Management of Data (SIGMOD)*, page 665, San Diego, USA, 2003.

[7] A. Arasu, S. Babu, and J. Widom. An abstract semantics and concrete language for continous queries over streams and relations. Technical report, Stanford University Database Group, 2002.

[8] A. Arasu and G. Manku. Approximate counts and quantiles over sliding windows. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 286–296, Paris, France, 2004.

[9] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *International Conference on Very Large Data Bases (VLDB)*, pages 336–347, Toronto, Canada, 2004.

[10] S. Arya, D.M. Mount, N.S. Netanyahu, R. Silverman, and A.Y. Wu. An optimal algorithm for approximate nearest neighbor searching. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 573–582, Arlington, USA, 1994.

[11] S. Arya, D.M. Mount, N.S. Netanyahu, R. Silverman, and A.Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of ACM*, 45(6):891–923, 1998.

[12] R. Avnur and J.M. Hellerstein. Eddies: Continuously adaptive query processing. In *ACM International Conference on Management of Data (SIGMOD)*, pages 261–272, Dallas, USA, 2000.

[13] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain : Operator scheduling for memory minimization in data stream systems. In *ACM International Conference on Management of Data (SIGMOD)*, pages 253–264, San Diego, USA, 2003.

[14] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 1–16, Madison, USA, 2002.

[15] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 633–634, San Francisco, USA, 2002.

[16] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *International Conference on Data Engineering (ICDE)*, pages 350–361, Boston, USA, 2004.

[17] B. Babcock, M. Datar, R. Motwani, and L. O'Callaghan. Maintaining variance and k-medians over data stream windows. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 234–243, San Diego, USA, 2003.

[18] B. Babcock and C. Olston. Distributed top-k monitoring. In *ACM International Conference on Management of Data (SIGMOD)*, pages 28–39, San Diego, USA, 2003.

[19] S. Babu, K. Munagala, J. Widom, and R. Motwani. Adaptive caching for continuous queries. In *International Conference on Data Engineering (ICDE)*, pages 118–129, Tokyo, Japan, 2005.

[20] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3):109–120, 2001.

[21] A. D. Barbour, L. Holst, and S. Janson. *Poisson Approximation*. Oxford Science Publications, 1992.

[22] R. Bayer. The universal B-tree for multidimensional indexing: General concepts. *World-Wide Computing and Its Applications 97*, pages 10–11, 1997.

[23] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *ACM International Conference on Management of Data (SIGMOD)*, pages 322–331, Atlantic City, USA, 1990.

[24] S. Berchtold, C. Böhm, H.V. Jagadish, H.-P. Kriegel, and J. Sander. Independent quantization: An index compression technique for high-dimensional data spaces. In *International Conference on Data Engineering (ICDE)*, pages 577–588, San Diego, USA, 2000.

[25] S. Berchtold, D. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *International Conference on Very Large Data Bases (VLDB)*, pages 28–39, Bombay, India, 1996.

[26] M.W. Bern. Approximate closest-point queries in high dimensions. *Information Processing Letters*, 45(2):95–99, 1993.

[27] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *International Conference on Mobile Data Management*, pages 3–14, Hong Kong, China, 2001.

[28] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S.B. Zdonik. Monitoring streams - a new class of data management applications. In *International Conference on Very Large Data Bases (VLDB)*, pages 215–226, Hong Kong, China, 2002.

[29] D. Carney, U. Çetintemel, A. Rasin, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. In *International Conference on Very Large Data Bases (VLDB)*, pages 838–849, Berlin, Germany, 2003.

[30] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, S. Krishnamurthy W. Hong, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Conference on Innovative Data Systems Research (CIDR)*, Asilomar, USA, 2003.

[31] S. Chandrasekaran and M. J. Franklin. Streaming queries over streaming data. In *International Conference on Very Large Data Bases (VLDB)*, pages 203–214, Hong Kong, China, 2002.

[32] S. Chandrasekaran and M. J. Franklin. Remembrance of streams past: Overload-sensitive management of archived streams. In *International Con-*

ference on Very Large Data Bases (VLDB), pages 348–359, Toronto, Canada, 2004.

[33] U. Charkravarthy and J. Minker. Processing multiple queries in database systems. *IEEE Database Engineering Bulletin*, 5(3):38–44, 1982.

[34] D. Chatziantoniou, M. O. Akinde, T. Johnson, and S. Kim. The MD-join: An operator for complex OLAP. In *International Conference on Data Engineering (ICDE)*, pages 524–533, Heidelberg, Germany, 2001.

[35] S. Chaudhuri, G. Das, and V. Narasayya. A robust, optimization-based approach for approximate answering of aggregate queries. In *ACM International Conference on Management of Data (SIGMOD)*, pages 295–306, Santa Barbara, USA, 2001.

[36] S. Chaudhuri, R. Motwani, and V. R. Narasayya. Random sampling for histogram construction: How much is enough? In *ACM International Conference on Management of Data (SIGMOD)*, pages 436–447, Seattle, USA, 1998.

[37] S. Chaudhuri, R. Motwani, and V. R. Narasayya. On random sampling over joins. In *ACM International Conference on Management of Data (SIGMOD)*, pages 263–274, Philadelphia, USA, 1999.

[38] J. Chen, D. J. DeWitt, and J. F. Naughton. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *International Conference on Data Engineering (ICDE)*, pages 345–356, San Jose, USA, 2002.

[39] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *ACM International Conference on Management of Data (SIGMOD)*, pages 379–390, Dallas, USA, 2000.

[40] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. B. Zdonik. Scalable distributed stream processing. In *Conference on Innovative Data Systems Research (CIDR)*, Asilomar, USA, 2003.

[41] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *International Conference on Very Large Data Bases (VLDB)*, Athens, Greece, 1997.

[42] G. Cormode, M. Datar, P. Indyk, and S. Muthukrishnan. Comparing data streams using hamming norms (how to zero in). In *International Conference on Very Large Data Bases (VLDB)*, pages 335–345, Hong Kong, China, 2002.

[43] G. Cormode, T. Johnson, F. Korn, S. Muthukrishnan, O. Spatscheck, and D. Srivastava. Holistic UDAFs at streaming speeds. In *ACM International Conference on Management of Data (SIGMOD)*, pages 35–46, Paris, France, 2004.

[44] G. Cormode and S. Muthukrishnan. What's hot and what's not: tracking most frequent items dynamically. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 296–306, San Diego, USA, 2003.

[45] C. Cortes, K. Fisher, D. Pregibon, and A. Rogers. Hancock: a language for extracting signatures from data streams. In *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 9–17, Boston, USA, 2000.

[46] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *ACM International Conference on Management of Data (SIGMOD)*, pages 647–651, San Diego, USA, 2003.

[47] C. D. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, and O. Spatscheck. Gigascope: high performance network monitoring with an SQL interface. In *ACM International Conference on Management of Data (SIGMOD)*, page 623, Madison, USA, 2002.

[48] C. D. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. The gigascope stream database. *IEEE Data Engineering Bulletin*, 26(1):27–32, 2003.

[49] M. Datar and S. Muthukrishnan. Estimating rarity and similarity on data stream windows. In *European Symposium on Algorithms*, pages 323–334, Rome, Italy, 2002.

[50] A. Deligiannakis and N. Roussopoulos. Extended wavelets for multiple measures. In *ACM International Conference on Management of Data (SIGMOD)*, pages 229–240, San Diego, USA, 2003.

[51] E. D. Demaine, A. López-Ortiz, and J. Ian Munro. Frequency estimation of internet packet streams with limited space. In *European Symposium on Algorithms*, pages 348–360, Rome, Italy, 2002.

[52] A. Deshpande and J. M. Hellerstein. Lifting the burden of history from adaptive query processing. In *International Conference on Very Large Data Bases (VLDB)*, pages 948–959, Toronto, Canada, 2004.

[53] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A query languagage for XML. *http://www.w3.org/TR/NOTE-xml-ql*.

[54] A. Dobra, M. N. Garofalakis, J. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. In *ACM International Conference on Management of Data (SIGMOD)*, pages 61–72, Madison, USA, 2002.

[55] A. Dobra, M. N. Garofalakis, J. Gehrke, and R. Rastogi. Sketch-based multi-query processing over data streams. In *International Conference on Extending Database Technology (EDBT)*, pages 551–568, Heraklion, Greece, 2004.

[56] N. Duffield, C. Lund, and M. Thorup. Learn more, sample less: control of volume and variance in network measurement. *IEEE Transactions on Information Theory*, 51(5):1756–1775, 2005.

[57] M. Dwass. *Probability and statistics: an undergraduate course*. W. A. Benjamin, 1970.

[58] A. Arasu et al. STREAM: The stanford stream data manager. *IEEE Data Engineering Bulletin*, 26(1):19–26, 2003.

[59] F. Fabret et al. Filtering algorithms and implementation for very fast publish/subscribe. In *ACM International Conference on Management of Data (SIGMOD)*, pages 115 – 126, Santa Barbara, USA, 2001.

[60] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing iceberg queries efficiently. In *International Conference on Very Large Data Bases (VLDB)*, pages 299–310, New York, USA, 1998.

[61] A. Faradjian, J. Gehrke, and P. Bonnet. GADT: A probability space ADT for representing and querying the physical world. In *International Conference on Data Engineering (ICDE)*, pages 201–211, San Jose, USA, 2002.

[62] J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan. An approximate L1-difference algorithm for massive data streams. In *Symposium on*

Foundations of Computer Science (FOCS), pages 501–511, New York, USA, 1999.

[63] W. Feller. *An introduction to probability theory and its applications*, volume I. John Wiley & Sons, Inc, 1968.

[64] R. F. S. Filho, A. Traina, and C. Faloutsos. Similarity search without tears: The omni family of all-purpose access methods. In *International Conference on Data Engineering (ICDE)*, pages 623–630, Heidelberg, Germany, 2001.

[65] S. Finkelstein. Common expression analysis in database applications. In *ACM International Conference on Management of Data (SIGMOD)*, pages 235–245, Orlando, USA, 1982.

[66] P. Flajolet. Approximate counting: A detailed analysis. *BIT*, 25(1):113–134, 1985.

[67] P. Flajolet and G. Martin. Probabilistic counting. In *Symposium on Foundations of Computer Science (FOCS)*, pages 76–82, Tucson, USA, 1983.

[68] L. Fu and S. Rajasekaran. Evaluating holistic aggregators efficiently for very large data sets. *VLDB Journal*, 13(2):148–161, 2004.

[69] L. Gao and X. S. Wang. Continually evaluating similarity-based pattern queries on a streaming time series. In *ACM International Conference on Management of Data (SIGMOD)*, pages 370–381, Madison, USA, 2002.

[70] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. In *ACM International Conference on Management of Data (SIGMOD)*, pages 13–24, 2001.

[71] A. C. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Fast, small-space algorithms for approximate histogram maintenance. In *ACM Symposium on Theory of Computing (STOC)*, pages 389–398, Montreal, Canada, 2002.

[72] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *International Conference on Very Large Data Bases (VLDB)*, pages 79–88, Roma, Italy, 2001.

[73] J. Goldstein and R. Ramakrishnan. Contrast plots and P-Sphere trees: Space vs. time in nearest neighbour searches. In *International Conference on Very Large Data Bases (VLDB)*, pages 429–440, Cairo, Egypt, 2000.

[74] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *International Conference on Data Engineering (ICDE)*, pages 152–159, New Orleans, USA, 1996.

[75] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *ACM International Conference on Management of Data (SIGMOD)*, pages 58–66, Santa Barbara, USA, 2001.

[76] S. Guha and B. Harb. Wavelet synopsis for data streams: Minimizing non-euclidean error. In *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 88–97, Chicago, USA, 2005.

[77] S. Guha, C. Kim, and K. Shim. Xwave: Approximate extended wavelets for streaming data. In *International Conference on Very Large Data Bases (VLDB)*, pages 288–299, Toronto, Canada, 2004.

[78] S. Guha and N. Koudas. Approximating a data stream for querying and estimation: Algorithms and performance evaluation. In *International Conference on Data Engineering (ICDE)*, pages 567–576, San Jose, USA, 2002.

[79] S. Guha, N. Koudas, and K. Shim. Data-streams and histograms. In *ACM Symposium on Theory of Computing (STOC)*, pages 471–475, Crete, Greece, 2001.

[80] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques and applications. *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing*, 18(2):3–18, 1995.

[81] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *ACM International Conference on Management of Data (SIGMOD)*, pages 47–57, Boston, USA, 1984.

[82] P. A. V. Hall. Optimization of single expressions in a relational data base system. *IBM Journal of Research and Development*, 20(3):244–257, 1976.

[83] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *ACM International Conference on Management of Data (SIGMOD)*, pages 205–216, Montreal, Canada, 1996.

[84] J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. A. Shah. Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, 2000.

[85] M. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. Technical Report 1998-011, Digital Equipment Corporation, System Research Center, May 1998.

[86] J. Hershberger and S. Suri. Adaptive sampling for geometric problems over data streams. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 13–18, Paris, France, 2004.

[87] G.R. Hjaltason and H. Samet. Ranking in spatial databases. In *Symposium on Large Spatial Databases (SSD)*, pages 83–95, Portland, USA, 1995.

[88] Traderbot home page. *http://www.traderbot.com*.

[89] M. Horton, D. Culler, K. PIster, J. Hill, R. Szewczyk, and A. Woo. Mica, the commercialization of microsensor motes. *Sensors*, 19(4):40–48, 2002.

[90] J.-H. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. B. Zdonik. High-availability algorithms for distributed stream processing. In *International Conference on Data Engineering (ICDE)*, pages 779–790, Tokyo, Japan, 2005.

[91] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *ACM Symposium on Theory of Computing (STOC)*, pages 604–613, Dallas, USA, 1998.

[92] Y. E. Ioannidis and S. Christodoulakis. Optimal histograms for limiting worst-case error propagation in the size of join results. *ACM Transactions on Database Systems (TODS)*, 18(4):709–748, 1993.

[93] Y. E. Ioannidis and V. Poosala. Balancing histogram optimality and practicality for query result size estimation. In *ACM International Conference on Management of Data (SIGMOD)*, pages 233–244, San Jose, USA, 1995.

[94] Y. E. Ioannidis and V. Poosala. Histogram-based solutions to diverse database estimation problems. *IEEE Data Engineering Bulletin*, 18(3):10–18, 1995.

[95] iPolicy Networks home page. *http://www.ipolicynetworks.com.*

[96] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel. Optimal histograms with quality guarantees. In *International Conference on Very Large Data Bases (VLDB)*, pages 275–286, New York, USA, 1998.

[97] H.V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang. iDistance: An adaptive b^+-tree based indexing method for nearest neighbor search. *ACM Transactions on Database Systems (TODS)*, 30(2):364–397, 2005.

[98] T. Johnson, S. Muthukrishnan, and I. Rozenbaum. Sampling algorithms in a stream operator. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1–12, Baltimore, USA, 2005.

[99] T. Johnson, S. Muthukrishnan, V. Shkapenyuk, and O. Spatscheck. A heartbeat mechanism and its application in Gigascope. In *International Conference on Very Large Data Bases (VLDB)*, pages 1079–1088, Trondheim, Norway, 2005.

[100] N. Katayama and S. Satoh. The SR-tree: an index structure for high-dimensional nearest neighbor queries. In *ACM International Conference on Management of Data (SIGMOD)*, pages 369–380, Tucson, USA, 1997.

[101] D. E. Knuth. *The Art of Computer Programming, Volume 3.* Addison Wesley, 2002.

[102] F. Korn, S. Muthukrishnan, and D. Srivastava. Reverse nearest neighbor aggregates over data streams. In *International Conference on Very Large Data Bases (VLDB)*, pages 814–825, Hong Kong, China, 2002.

[103] N. Koudas, B. C. Ooi, K.-L. Tan, and R. Zhang. Approximate NN queries on streams with guaranteed error/performance bounds. In *International Conference on Very Large Data Bases (VLDB)*, pages 804–815, Toronto, Canada, 2004.

[104] S. Krishnamurthy, S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: An architectural status report. *IEEE Data Engineering Bulletin*, 26(1):11–18, 2003.

[105] S. Krishnamurthy, M. J. Franklin, J. M. Hellerstein, and G. Jacobson. The case for precision sharing. In *International Conference on Very Large Data Bases (VLDB)*, pages 972–986, Toronto, Canada, 2004.

[106] E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. In *ACM Symposium on Theory of Computing (STOC)*, pages 614–623, Dallas, USA, 1998.

[107] Per-Ake Larson and H. Z. Yang. Computing queries from derived relations. In *International Conference on Very Large Data Bases (VLDB)*, pages 259–269, Stockholm, Sweden, 1985.

[108] C. Li, E. Y. Chang, H. Garcia-Molina, and G. Wiederhold. Clustering for approximate similarity search in high-dimensional spaces. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 14(4):792–808, 2002.

[109] L. Liu, W. Han, D. Buttler, C. Pu, and W. Tang. An XML-based wrapper generator for web information extraction. In *ACM International Conference on Management of Data (SIGMOD)*, pages 540–543, Philadelphia, USA, 1999.

[110] L. Liu, C. Pu, and W. Han. Xwrap: An xml-enabled wrapper construction system for web information sources. In *International Conference on Data Engineering (ICDE)*, pages 611–621, San Diego, USA, 2000.

[111] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 11(4):610–628, 1999.

[112] L. Liu, C. Pu, W. Tang, D. Buttler, J. Biggs, T. Zhou, P. Benninghoff, W. Han, and F. Yu. CQ: A personalized update monitoring toolkit. In *ACM International Conference on Management of Data (SIGMOD)*, pages 547–549, Seattle, USA, 1998.

[113] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *International Conference on Data Engineering (ICDE)*, pages 555–566, San Jose, USA, 2002.

[114] S. Madden, M.A. Shah, J.M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *ACM International Conference on Management of Data (SIGMOD)*, pages 49–60, Madison, USA, 2002.

[115] G. Manku and R. Motwani. Approximate frequency counts over data streams. In *International Conference on Very Large Data Bases (VLDB)*, pages 346–357, Hong Kong, China, 2002.

[116] G. Manku, S. Rajagopalan, and B. G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *ACM International Conference on Management of Data (SIGMOD)*, pages 426–435, Seattle, USA, 1998.

[117] G. Singh Manku, S. Rajagopalan, and B. G. Lindsay. Random sampling techniques for space efficient online computation of order statistics of large data sets. In *ACM International Conference on Management of Data (SIGMOD)*, pages 251–262, Philadelphia, USA, 1999.

[118] Y. Matias, J. Vitter, and M. Wang. Dynamic maintenance of wavelet-based histograms. In *International Conference on Very Large Data Bases (VLDB)*, pages 101–110, Cairo, Egypt, 2000.

[119] J. Mirkovic, S. Dietrich, D. Dittrich, and P. Reiher. *Internet denial of service : attack and defense mechanisms*. Prentice Hall, 2005.

[120] R. Morris. Counting large numbers of events in small registers. *Communications of the ACM*, 21(10):840–842, 1978.

[121] R. Motwani and D. Thomas. Caching queues in memory buffers. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 541–549, New Orleans, USA, 2004.

[122] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Singh Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *Conference on Innovative Data Systems Research (CIDR)*, Asilomar, USA, 2003.

[123] J. F. Naughton, D. J. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash, R. Ramamurthy, J. Shanmugasundaram, F. Tian, K. Tufte, S. Viglas, Y. Wang, C. Zhang, B. Jackson, A. Gupta, and R. Chen. The niagara internet query system. *IEEE Data Engineering Bulletin*, 24(2):27–33, 2001.

[124] S. Northcutt, M. Cooper, M. Fearnow, and K. Frederick. *Intrusion signatures and analysis*. New Riders, 2001.

[125] S. Northcutt and J. Novak. *Network intrusion detection : an analyst's handbook*. New Riders, 2000.

[126] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 181–190, Waterloo, Canada, 1984.

[127] S. Papadimitriou, A. Brockwell, and C. Faloutsos. Adaptive, hands-off stream mining. In *International Conference on Very Large Data Bases (VLDB)*, pages 560–571, Berlin, Germany, 2003.

[128] P.Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *International Conference on Very Large Data Bases (VLDB)*, pages 541–550, Roma, Italy, 2001.

[129] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. In *ACM International Conference on Management of Data (SIGMOD)*, pages 256–276, Boston, USA, 1984.

[130] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. In *ACM International Conference on Management of Data (SIGMOD)*, pages 294–305, Montreal, Canada, 1996.

[131] V. Raman, A. Deshpande, and J. M. Hellerstein. Using state modules for adaptive query processing. In *International Conference on Data Engineering (ICDE)*, pages 353–364, Bangalore, India, 2003.

[132] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *ACM International Conference on Management of Data (SIGMOD)*, pages 447–458, Montreal, Canada, 1996.

[133] N. Roussopoulos. View indexing in relational databases. *ACM Transactions on Database Systems (TODS)*, 7(2):256–290, 1982.

[134] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *ACM International Conference on Management of Data (SIGMOD)*, pages 71–79, San Jose, USA, 1995.

[135] Y. Sakurai, M. Yoshikawa, S. Uemura, and H. Kojima. The A-tree: an index structure for high-dimensional spaces using relative approximation. In *International Conference on Very Large Data Bases (VLDB)*, pages 516–526, Cairo, Egypt, 2000.

[136] T. Sellis. Multiple query optimization. *ACM Transactions on Database Systems (TODS)*, 13(1):23–52, 1998.

[137] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *International Conference on Data Engineering (ICDE)*, pages 25–36, Bangalore, India, 2003.

[138] A. Siegel. On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications. In *Symposium on Foundations of Computer Science (FOCS)*, pages 20–25, Research Triangle Park, North Carolina, USA, 1989.

[139] The JPEG 2000 standard. *http://www.jpeg.org/jpeg2000/index.html*.

[140] M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. In *USENIX Technical Conference*, New Orleans, USA, 1998.

[141] FAQ: Network Intrusion Detection Systems. *http://www.ticm.com/kb/faq/*.

[142] N. Tatbul, U. Çetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *International Conference on Very Large Data Bases (VLDB)*, pages 309–320, Berlin, Germany, 2003.

[143] W.-G. Teng, M.-S. Chen, and P.S. Yu. A regression-based temporal pattern mining scheme for data streams. In *International Conference on Very Large Data Bases (VLDB)*, pages 93–104, Berlin, Germany, 2003.

[144] V. V. Vazirani. *Approximation algorithms*. Springer, 2001.

[145] J. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

[146] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *International Conference on Very Large Data Bases (VLDB)*, pages 194–205, New York, USA, 1998.

[147] K.-Y. Whang, B. T. Vander-Zanden, and H.M. Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems (TODS)*, 15(2):208–229, 1990.

[148] D. A. White and R. Jain. Similarity indexing with the SS-tree. In *International Conference on Data Engineering (ICDE)*, pages 516–523, New Orleans, USA, 1996.

[149] E. Wong and K. Youssefi. Decompositiona strategy for query processing. *ACM Transactions on Database Systems (TODS)*, 1(3):223–241, 1976.

[150] Z. Xiang, K. Ramchandran, M. T. Orchard, and Y. Q. Zhang. A comparative study of dct- and wavelet-based image coding. *IEEE Transactions on Circuits and Systems for Video Technology*, 9(5):692C695, 1999.

[151] Y. Xing, S. B. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. In *International Conference on Data Engineering (ICDE)*, pages 791–802, Tokyo, Japan, 2005.

[152] Y. Yao and J. Gehrke. The Cougar approach to in-network query processing in sensor networks. *SIGMOD Record*, 31(3):9–18, 2002.

[153] Y. Yao and J. Gehrke. Query processing in sensor networks. In *Conference on Innovative Data Systems Research (CIDR)*, Asilomar, USA, 2003.

[154] C. Yu, B.C. Ooi, K.-L. Tan, and H. V. Jagadish. Indexing the distance: An efficient method to knn processing. In *International Conference on Very Large Data Bases (VLDB)*, pages 421–430, Roma, Italy, 2001.

[155] R. Zhang, N. Koudas, B. C. Ooi, and D. Srivastava. Multiple aggregations over data streams. In *ACM International Conference on Management of Data (SIGMOD)*, pages 299–310, Baltimore, USA, 2005.