Lightweight Top-K Analysis in DBMSs Using Data Stream Analysis Techniques

by

Jing Huang

A thesis submitted to the School of Computing

In conformity with the requirements for

the degree of Master of Science

Queen's University

Kingston, Ontario, Canada

(September, 2009)

Abstract

Problem determination is the identification of problems and performance issues that occur in an observed system and the discovery of solutions to resolve them. Top-k analysis is common task in problem determination in database management systems. It involves the identification of the set of most frequently occurring objects according to some criteria, such as the top-k most frequently used tables or most frequent queries, or the top-k queries with respect to CPU usage or amount of I/O.

Effective problem determination requires sufficient monitoring and rapid analysis of the collected monitoring statistics. System monitoring often incurs a great deal of overhead and can interfere with the performance of the observed system. Processing vast amounts of data may require several passes through the analysis system and thus be very time consuming.

In this thesis, we present our lightweight top-k analysis framework in which lightweight monitoring tools are used to continuously poll system statistics producing several continuous data streams which are then processed by stream mining techniques. The results produced by our tool are the "top-k" values for the observed statistics. This information can be valuable to an administrator in determining the source of a problem.

We implement the framework as a prototype system called Tempo. Tempo uses IBM DB2's snapshot API and a lightweight monitoring tool called DB2PD to generate the data streams. The system reports the top-k executed SQL statements and the top-k most frequently accessed tables in an on-line fashion. Several experiments are conducted to verify the feasibility and effectiveness of our approach. The experimental results show that our approach achieves low system overhead.

Acknowledgments

I would like to extend my sincerest gratitude to my supervisor, Professor Patrick Martin, for his all great guidance and advice over the years when I have pursued my education and research at Queen’s University.

I would like to thank Wendy Powley for her support. She has always been a wonderful source of advice and suggestions. Without her great work, I could not finish my thesis so successfully. I would also like to thank to Paul Bird and Dmitri Abrashkevich for all his help with this research.

I would like to give my gratitude to the School of Computing at Queen’s University for their support. I would also like to acknowledge IBM Canada Ltd., and MITACS for the gracious financial support they have provided.

I would like to thank my lab mates and fellow students for their encouragement, support, and kindness.

Finally I would like to express my sincerest appreciation to my family for their love and understanding.

Table of Contents

List of Figures

List of Tables

Glossary of Terms

CATALOG	DB2-maintained tables that contain descriptions of DB2 objects
CentOS	A freely-available operating system that is based on Red Hat Enterprise Linux
DBA	Database Administrator
DB2	IBM's Relational Database Management System
DBMS	Database Management System
DIRECT I/O	File system I/O that bypasses the OS-level page cache
DTW	DB2 Transaction Workload
OLTP	Online Transaction Processing
RAID	Redundant Array of Inexpensive Disks
RAW DEVICE	A special kind of block device file that allows accessing a storage device such as a hard drive
RDBMS	Relational Database Management System
SATA	Serial Advanced Technology Attachment
SQL	Structured Query Language
TABLESPACE	A storage location to store the database objects
VOLUME GROUP	A collection of physical volumes broken down into physical extents

Chapter 1
Introduction

1.1 Motivation

The desire to offer on-demand 24/7 services means there is pressure to quickly identify and resolve problems in a database management system (DBMS). Outage costs associated with problems on business-critical systems can range from thousands to millions of dollars per hour depending on the industry and the type of problem [13]. Database Management Systems (DBMSs) are the backbone of enterprise systems, storing petabytes of business data and serving various types of application systems, such as E-business systems, core-business systems, and other information systems. Any types of harm or malfunction to the DBMS can seriously impact the quality of service of the business system. Therefore, it is crucial to maintain the health of the DBMS.

Problem determination involves the detection, location, and resolution of problems and is a key task of database administrators (DBAs). However, system complexity and problem difficulty present DBAs with a tough challenge; often even skilled specialists cannot quickly identify the problems and resolve them. Therefore, there exists a strong demand for effective tools for problem determination.

Problem determination tools rely on the existence of sufficient monitoring data to support further analysis. Nowadays, commercial database systems expose their statistical information using two primary monitoring mechanisms: snapshot monitoring and event monitoring. Snapshot monitoring presents system activities for a given point in time by polling system statistics. Snapshots capture a sense of the current state of the DBMS. If a history of the system activities is required, multiple snapshots must be recorded in storage. Offline analysis can then be used to process the data.

Event monitoring captures all events of specified interest over a period of time. Event monitors can be thought of as a movie capturing what occurred over time [22]. Figure 1-1 shows the traditional scenario for database problem determination.

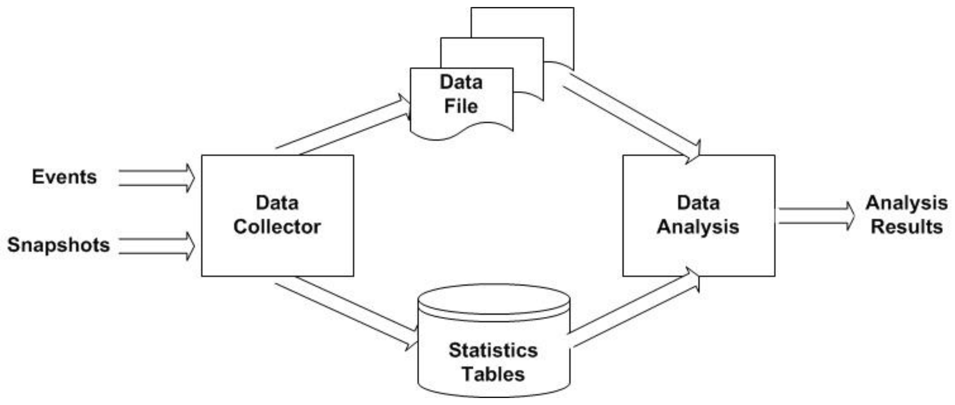


Figure 1-1 Traditional Database Problem Determination

1.2 Problem

Far from simply capturing current system statistics or recording counter values associated with system events, database problem determination often requires the historical extensions of the reported values and based on them, further analysis can be more meaningful. For event monitoring, tracing the history of database activities and obtaining a clear picture is straightforward. However, when there are many activities to be monitored, it is onerous for a DBA to define many events and filter out useful information from large result sets. For snapshot monitoring, more frequent polling means more system statistics are collected, thereby resulting in better analysis on the historical statistics. However, frequent polling inevitably incurs extra overhead on system resources due to the recording and processing of large amounts of data and this can interfere with key DBMS components, such as the query and storage engines. Less frequent polling can lessen the system overhead problem to some extent, but this is prone to the loss of relevant data, thereby leading to incorrect analysis.

Monitoring data can be thought of as a continuous stream of data with no fixed size limit as opposed to a static dataset which is typically stored in disk or other persistent medium. The traditional approach to analyze this type of data requires significant use of disk and offline processing. Thus, it can incur a large amount of overhead and interfere with the monitored system. In addition, data analysis often involves scanning each element multiple times and assumes that historical data can be retrieved in the future. When the data arrive at a high speed, it is inevitable to delay reporting analysis results. This type of analysis is hard to fulfill quickly identifying problems.

1.3 Research Statement

The objective of our research is to examine the viability of using stream analysis and mining techniques to provide effective support for problem determination. Based on the objective, we develop a prototype system using lightweight monitoring tools from IBM's

DB2® [18] to simulate stream of monitoring data and then apply stream analysis techniques.

Our work makes three main contributions. The first contribution is a lightweight Top-K analysis framework in which observed statistics are rendered as the elements of data streams and analyzed by stream summary processes. The second contribution is the revision of a stream mining algorithm, Space-Saving [1] to make it more sensitive to the fluctuation of the observed objects so as to improve the accuracy of our analysis. The third contribution is a prototype implementation of our framework using DB2 in which several lightweight DB2 monitoring tools and the revised Space-Saving algorithm are used to continuously generate the TOP-K most frequently executed SQL statements and the TOP-K most frequently accessed tables on the fly. The effectiveness of our stream-based approach is validated through a set of experiments with our prototype system.

1.4 Thesis Organization

The remainder of the thesis is organized as follows. Chapter 2 describes the background and related work. Chapter 3 presents our framework and implementation. In Chapter 4, we present a set of experiments to evaluate our approach and the experimental results. Chapter 5 draws conclusions about this thesis and outlines potential future work.

Chapter 2

Background and Related Work

Problem determination, which detects problems and proposes effective solutions to resolve them, is vital to the performance and health of a DBMS. When problems occur, it is important to perform accurate and timely diagnosis before making any changes to the system. An accurate diagnosis of the actual problem in the initial stage significantly increases the probability of success in resolving the problem.

2.1 Monitoring in a DBMS

Database monitoring involves continuously observing the values of system counters that describe the system’s state [26]. These counters are initialized when their applicable object becomes active and incremented during the operation of the database. They also can be reset to zero when the database becomes inactive [15]. Database monitoring is used for problem determination, performance management, and trend analysis [22]. Performance management ensures that system resources are used optimally, which helps avoid some potential problems. Trend analysis keeps the historical data from collected system counters and uses them to determine growth and trends in usage. Trends can help identify changes in overall system activity and plan upgrades if they are needed. The three key uses of database monitoring interact with each other. In our thesis, we mainly discuss about problem determination.

Traditionally, a database administrator (DBA) executes specific SQL queries or invokes primitive monitoring tools, such as snapshot monitoring and event monitoring, to capture the symptoms, and based on his/her past experience, identifies and fixes the problem. Snapshot monitoring, which can capture activity information about the observed database and any connected applications at a specific time, is useful for determining the status of a database system. For easy management, the monitored database activities and status values are generally classified into several logic groups. When specific groups of information are required to be collected, the corresponding monitoring switches are opened. Snapshot monitoring outputs the statistics relative to the monitored database activities via command line outputs, the statistical tables or snapshot API interface. Taken at regular intervals, snapshot monitoring can be also used to observe trends and forecast potential problems. In this case, the database system continuously keeps track of the activities and records statistics in persistent storage.

Event monitoring is another facility used to monitor a database system. It collects the information about the activities of the system and connected applications when specified events occur. Monitored events are specified in advance and are typically recorded into event log files as the database events occur. Event monitoring is suitable for notifying DBAs with immediate alerts. When using event monitoring to trace the history of some database activities, traversing the whole event logs to identify the specific events is necessary. If the events relative to the database activities are not defined or do not occur during the observation period, the related monitoring task cannot be completed. Event monitoring, like snapshot monitoring, has a tradeoff between accuracy of answers and the load on the observed database server. When a system is configured to collect many kinds of events, the volume of event logs tends to grow very fast. Conversely, if only a few kinds of events are logged, then important information could be missed.

Snapshot and event monitoring suffer from several drawbacks. First, manually monitoring often interferes with the observed database systems. Second, database systems typically have a plethora of measurements and statistics available about their activities and status that are obtained through different means. It is difficult for DBAs to obtain an overall picture of what is happening in the database. Third, the statistics captured by the DBAs are isolated from each other in time. Most of them generally reflect only the current condition of the monitored database system, so it is hard to infer historical trends or results. Fourth, database problems are unpredictable so the system must be monitored continuously and someone must be available to identify and correct problems. Fifth, because of the sheer volume of collected statistics and some complex data patterns or trends hidden in them, it is impossible for DBAs to identify problems manually.

Nowadays, commercial database systems are more sophisticated with thousands of components fulfilling a variety of functions. The cost of ownership, especially the cost of administration is increasingly expensive [32]. Traditional methods of manual problem determination hardly satisfy the current management requirement of the database systems [27] [14]. Therefore, there exists a pressing demand for greater automation in this area.

The IBM DB2 Health Center [15] and the Oracle Automatic Database Diagnostic Monitor (ADDM) [23] provide more sophisticated assistance for problem determination. These systems maintain performance statistics for problem detection and self-tuning purposes by continuously recording the events of database activities in a temporary repository or database tables and process further analysis on these collected statistics. However, to obtain an overall or historical picture of the observed database system, a large number of database events must be defined and many statistics must be kept in persistent storage. This leads to an increase in system overhead and possibly long delays before a problem is diagnosed; especially if a replaying or off-line analysis process is required. This type of automatic problem determination system is only an extension of the manual process that a DBA uses for problem determination. The other limitation of the state-of-the art automatic problem determination techniques is that they are designed primarily to assist DBAs in problem determination by providing more data which requires more complex monitoring and analysis functions. With the increase of complexity and the number of these functions, extra overheads incur.

Database monitoring is typically centralized although the centralized task can be distributed to several sub-tasks across network, the final or temporary decisions still require a combination of the sub-opinions and extra communication cost usually is expensive [4]. Monitoring a DBMS and problem determination not only require space and time efficiency, but also have stricter processing requirement on communication, compared with other data monitoring and analysis processes. SQLCM [26] is proposed to implement the centralized monitoring of database systems by using a revised event logging mechanism to maintain a lightweight aggregation table inside the database server to continuously monitor the observed database system. Some memory structures and implementation methods that SQLCM adopts are similar to what we use in our approach. However, inserting complex summarization and object-identification for the related system statistics inside the database sever not only places an increased load on the database system, but also does not directly support the on-line monitoring function across the different database platforms. In addition to this, the event logging approach SQLCM uses to maintain system statistics is relatively expensive and is not as flexible as snapshot monitoring.

2.2 Streaming Model & Stream Data Management System

Beyond the infrastructure to monitor the status of the database server so as to drill down to obtain the details only when a problem is suspected, the three key challenges processing monitoring database are the speed limitation (the data collected by a monitor can arrive at a high speed), the space limitation (the data is potentially unbounded and arrives continuously) and the performance impact on the observed system from monitoring and analysis. Neither of the prevalent mechanisms provides adequate support. To handle this type of data, a data stream model and streaming mining techniques are proposed here instead of traditional monitoring and analysis data processes.

In the data stream model [12] [3], an unbounded sequence of elements arrive in the form of continuous streams. Streaming data can be generated from various application systems, such as sensor data of sensor networks, stock tickers, transaction flows in retail chains, web records in web applications, performance and outlier tracing information in network monitoring, traffic recording in telecommunications and so on. A number of Data-Stream Management Systems (DSMSs), such as STREAMS [8], TelegraphCQ [28], Aurora [20], Gigascope [6] and CACQ [29] have been proposed. These systems gather data from multiple sensors or applications via a network and operate on data streams instead of static relations as in the RDBMSs. DSMSs are designed to extend the functionality of traditional DBMSs to support continuous queries and further support the monitoring of streaming-style applications.

Unlike conventional data which consists of a finite number of elements, the data in data streams are unbounded in size and there is often the need to access the entire history of some important values in data streams [9]. It is infeasible to store all the data from a data stream. Streaming data typically arrives at a rapid rate and in general, can be viewed only once and cannot be retrieved at a later date.

A sliding windows model [33] [3] is commonly used to allow the processing to keep up with the fast data arrival. At any point in time, a sliding window over a stream is a set of elements of the stream seen so far. Sliding windows may be fixed or variable in size and they can be classified into either time-based, where elements are ordered by timestamp indicating their arrival time in a data stream, or sequence-based, where timestamps are substituted with a series of sequence numbers which are continuously increasing with the insertion of new elements in the streams.

In addition to the sliding window model, load shedding [5] and sampling techniques are other approaches used to reduce the system load by discarding a portion of the data streams. Although these techniques are used, the amount of memory that can be used to maintain the information of the data streams is still limited. In stream processing, further summarizing of the data is often necessary. The amount of memory used to maintain a synopsis is relatively small compared to the actual scanned data.

2.3 Data Stream Mining

Unlike conventional data mining techniques, which typically make multiple passes over static data, stream processing can only scan the data once in the same order as it arrives. During the last few years, many attempts have been made to adapt existing data mining algorithms to data streams. For example, methods for building synopses or summaries are proposed and data models are built based on the most recent observation and evolve with the change of underlying data streams. Generally, current efforts mainly focus on online mining where a model of the data is built continuously and incrementally from the records as they flow into the system [12]. Three main classes of algorithms have been studied: data stream summarization, data stream classification and data stream clustering.

Data stream summarization maintains summaries of the observed data stream with a compact format. It includes three major forms: maintaining statistics of data streams, identifying frequent records in data streams, and detecting changes in data streams. In our thesis, we introduce the details about identifying frequent records or generating TOP-K elements in data streams and apply them to process the statistics collected from the database system.

Data stream classification extends classification algorithms such as the Decision Tree algorithm (DT) and adapts them to data streams. The algorithms generally build a data model (DM) based on some attributes of data records. The building process often requires a large amount of historical records which is not possible when we process data streams.

In addition, unlike a static data set, a data stream continuously changes over time, so the criteria for building a DM also may change during observation. This phenomenon is called concept drift. To adapt to the relatively small training data set and the concept drift in data streams, continuously updating the DM over time based on most recent records is necessary. For example, Domingos and Hulten propose using Hoeffding bounds to evaluate whether the sample of the attribute values extracted from the current window of a data stream can provide enough information to make a good decision [25].

Data stream clustering also extends traditional data clustering to data streams. Data clustering is the process of gathering input data into a number of groups or clusters [12]. In general, traditional clustering algorithms require all data sets before processing them and during processing repeatedly update the groups of data based on similarity of these data records. For example, a well-known clustering algorithm k-means repeatedly calculates the Euclidean distance between each record and the corresponding centroid in each data cluster to measure the similarity of them and update the clusters to minimize the within-cluster sum of squares. Some solutions are proposed to compute the Euclidean distances based on the data records in the current windows rather than the whole data stream and keep summaries of the properties of the data records, such as the cluster centroids and the related diameters, in memory.

Detecting changes in data streams is useful in areas such as network monitoring, traffic management and intrusion detection. Change detection algorithms can be classified into two major types: detecting distribution changes and detecting stream bursts. The distribution changes in data steams can be detected by comparing the distributions of the current window of data and a reference window that captures the underlying distribution of the stream. The bursts in a stream can be detected by checking whether some attribute values in the maintained summaries have been over the corresponding thresholds.

2.3.1 Frequent and TOP-K Elements

The two most critical challenges in processing streaming data are dealing with the unlimited data size and the rapid arrival rate. When designing a highly efficient, randomized approach for maintaining the critical statistics over a stream, we should consider the following three essential factors: the amount of storage space used (physical space and/or RAM), the time to update the data structure following the arrival of each element in the stream and the time to produce the result.

Many performance problems can be determined by a few key pieces of information such as “which queries are using the majority of the CPU” or “what are the top 5 tables that are being most heavily accessed?” This type of analysis task over data streams focuses only on atypical behavior in the monitored system, while habitual behaviors are ignored [4]. In other words, only a small number of “important” data subset requires detailed analysis and most of the data in the observed streams can be coarse-granularly scanned, even safely disregarded.

As mentioned previously, we can derive a significant amount of information by determining the TOP-K statistics for some portion of the system, for instance, the TOP-5 indices that are accessed most frequently. Many monitoring and analysis applications over data streams are characterized by numeric values or the frequencies of the specifically interesting objects. Sometimes, the detailed observation is only necessary for the elements whose corresponding numeric attributes' values are among the k largest, namely, TOP-K elements. The frequent elements problem is different from the TOP-K elements problem in that it requires the output of the elements whose frequency exceed ΦN, for a given threshold parameter Φ, where N is the total occurrences of an observed data stream. Since, TOP-K or most frequent elements are only the small part of the whole observed objects, the storage and processing burdens posed on the monitoring infrastructure can be dramatically cut down when we only inspect the limited scope of data streams accordingly.

In addition, although there is enough data to produce complex models, only simple models can be produced since the system is unable to take full advantage of the data [24]. In most cases, only probabilistic estimates are possible using analysis over data streams due to the limited resources. To provide results for TOP-K element problems, computing approximate counts for elements of interest is necessary because an exact answer requires access to the entire set of observed data, which is infeasible for many practical stream rates and window sizes [10] [2]. Several structures based on continuously summarizing the data seen so far are utilized to maintain approximate counts over time. Since K is typically very small compared to the number of unique objects, maintaining small summary data structures usually can suffice to keep the statistics of the observed data stream.

There are several algorithms proposed to approximate the most frequent elements. These algorithms can be divided into two types based on how the frequencies of elements are summarized: counter-based algorithms and sketch-based algorithms [1]. Counter-based algorithms such as StatStream (without approximate) [33], StickySampling [14] and Space-Saving [1], are sampling-based algorithms where several samples of the observed data stream are selected over time using some probabilistic criterion. The set of sampled elements changes with time according to some algorithmic constraints. In the statistics gathering process, the key operation for Counter-based algorithms is counting. Typically, the process maintains *m* active counters and each counter has an associated category that it monitors. A counter can be incremented, decremented, or reset so as to monitor a different category. The amount of space counter-based algorithms require often depends on the distribution of the frequency of the items in the data stream.

Sketch-based algorithms, such as LossyCounting [10], CountSketch [21] and GroupTest [11], generally examine each element of the monitored data stream rather than only a subset of sampled elements. Because of this, they are less affected by the order of the elements in the stream. During processing, each element is usually mapped into one of a set of counters which are stored in histograms, bit-maps, or application-specific data structures by using one or multiple hash functions. The counters are updated when a certain element arrives. Eventually, the representative counters are queried for the element frequency with expected loss of accuracy due to hashing collisions. Typically, outputting the result for one element requires the calculation across several representative counters. Due to the fact the entire stream is scanned and the analysis involves more complex processing, the sketch-based algorithms are more expensive than counter-based techniques, although they usually provide relatively higher accuracy. The sketch-based algorithms, therefore, are not ideal for environments which require relatively faster and more lightweight analysis.

Chapter 3

A Lightweight Top-K Analysis Framework

In this chapter we describe a framework that we call Tempo, designed for real-time lightweight TOP-K analysis in DBMSs. Our approach uses stream processing and a revised Space-Saving algorithm to determine the K most frequent values of a property (TOP-K results) from the stream of records. This chapter outlines the approaches we adopt, in particular, the means to lightweight, continuous monitoring of the database, and the Space-Saving algorithm that is used in our approach.

3.1 Framework Overview

Problem determination involves detecting and diagnosing situations that affect the operational states or availability of the observed systems. The goal of problem determination is to maximize system availability by minimizing the time it takes to locate problems and recover the system to a normal state. This is accomplished by collecting sufficient monitoring information to quickly detect meaningful conditions, diagnosing the underlying problems, and applying available knowledge to restore normal system operations. At the same time, this process must not incur significant system overhead so as to not interfere with the original operation of the observed system. Three major criteria for a good problem determination solution include:

- The processes should be fast.

- Analyzing collected statistics should be continuous, without delay.
- The overhead of monitoring and analysis tools should be low.

Traditional DBMS problem determination tools typically collect system statistics and store them on disk where the data is analyzed using offline analysis techniques. This traditional approach does not satisfy the criteria described above, as processing of the statistics is delayed and many of the tools used for monitoring the DBMS incur significant overhead. Our goal is to develop a lightweight, continuous method of data analysis that provides important information for problem determination in a timely and efficient manner.

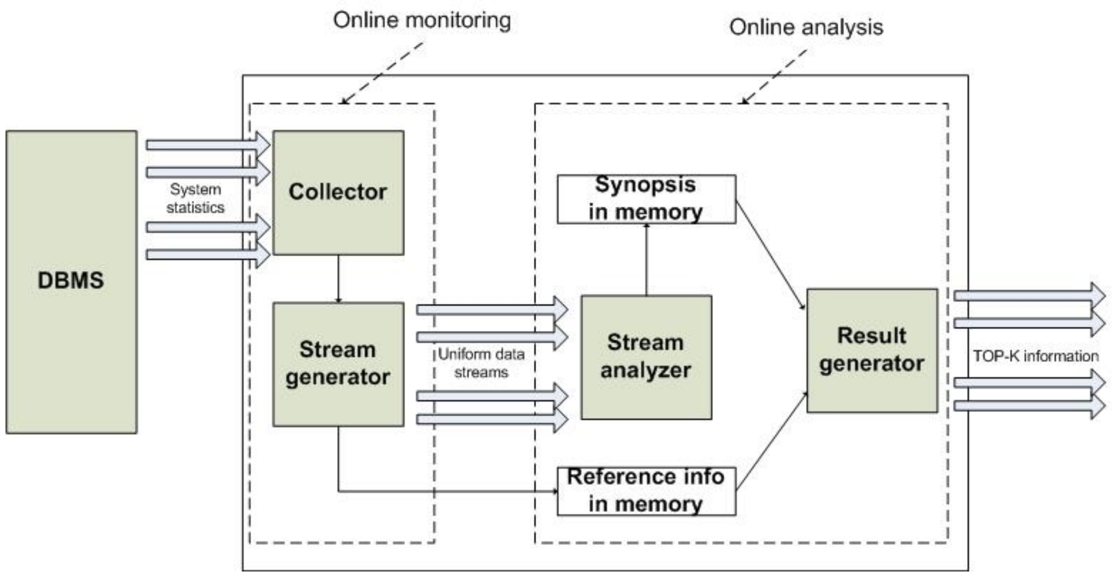


Figure 3-1 Tempo Framework

The proposed Tempo framework is shown in Figure 3-1. In this framework, the input system statistics are collected to simulate an infinite data stream and the analysis process uses stream mining techniques to determine the output. The framework consists of two main modules; online monitoring and online analysis. Details of these processes are explained in the following sections.

3.1.1 Online Monitoring

Monitoring plays a key role in problem determination, in general, and TOP-K analysis, in particular. Tempo's monitoring subsystem contains two modules, the Collector and the Stream Generator. The Collector repeatedly uses the DBMS monitoring tool, which is snapshot monitoring for DB2 in the case of our prototype, to gather system statistics from the database system. The Stream Generator formulates a stream of information from the collected data and passes it to the analysis module. The interface between the two modules is DBMS independent so Tempo can be made to work for other DBMSs by modifying part of the online monitoring module.

Collector

In our prototype the DB2 snapshot tool (see Appendices B and C for details) is used to poll system statistics in order to provide Tempo with a continuous stream of monitoring data. Compared with other monitoring mechanisms, such as event monitoring or trigger monitoring, snapshot monitoring is more lightweight and flexible. However, when using the snapshot mechanism, there is a trade-off between overhead and accuracy. By taking frequent snapshots, one can record exhaustive information regarding system activity with a high degree of accuracy. However, the frequent calls to system routines or commands due to frequent polling incur higher overhead and interfere with the observed system.

Infrequent polling, on the other hand, can reduce the monitoring costs, but this comes at the expense of the loss of system statistics and, hence, loss of accuracy.

Typical data returned by a snapshot monitor includes cumulative counters that indicate system statistics such as the number of sort overflows, or the number of logical or physical I/Os that have occurred since the counter was reset. In DB2 v9.5, some aggregated system statistics are provided. This information is continuously computed inside the database server, thereby reducing the costs of frequent system calls and communications. For example, we can obtain the total CPU time for executing an SQL statement, or the total number of times a table has been accessed since the beginning of the monitoring of the table or the last reset of the monitoring via a single snapshot execution. Using these summarized statistics, polling can be done less frequently without loss of data and the overhead is maintained at a relatively low level.

Sources of possible information loss include database shutdown, or inactivity for a period of time, and overflow of the monitoring cache which occurs when more data is being collected than can be held in the cache. In this case, data must be flushed out of the cache in order to accommodate the most recent information. Other than potential information loss which shares similar problems to the infrequently polling, directly collecting accumulating database statistics also faces other issues such as the interruption or reset of the accumulating statistics, all of which will result in incorrect estimation on the activities and status of the observed objects.

In order to compensate for data loss, we introduce a method whereby we predict the current values of observed objects based on their last and current arrival values and the current and last observed states of the DB2 database system. The pseudo code in Figure

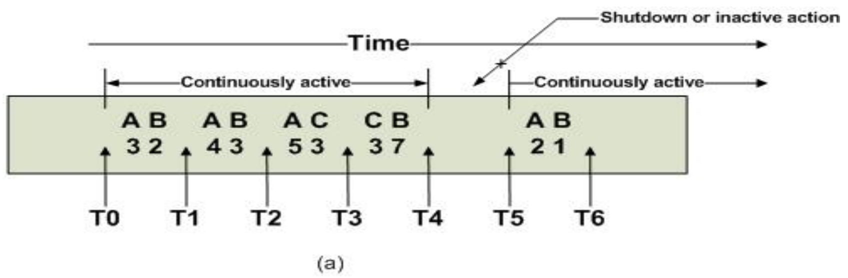
3-2 describes how we compute the current counts, ***C_current***, of an observed object. The last and current arrival counts of an observed object are denoted by ***C_last*** and ***C_arrival*** respectively. There exist three database states: active, inactive, and shutdown. ***S*** denotes the state change of the observed database and can be the permutation of any two states. When a database system is inactive (there is no action for a long time) or the DBMS is shut down, the counts stored in the monitoring cache are reset to 0. On the other hand, if the database is continuously in an active state, the counts in the monitoring cache are updated according to the activity in the database.

```
For each observed unique object $O_i$
DO
    Boolean Flag;
    Long long int Ref_no; /* the definition of Ref_no and reference table refers to the later part of the
Chapter or Chapter 4 */
    int   C_last;
    int   C_arrival;

    C_arrival = the times $O_i$ has been observed in the interval; /* the value comes from snapshot polling */
    Ref_no = hash_func ($O_i$);     /* we use Paul Hsieh's hash function*/
    Flag = lookup_in_reference (reference_type, Ref_no, &C_last); /* reference_type : table or SQL*/

    IF Flag is not true THEN /*the object has not been observed before*/
        C_last =0;
        push_ref_table (reference_type, Ref_no) /* add a new object into the reference table */
    ENDIF;

    CASE (S)
        WHEN S is continuously active or changed from inactive to active THEN
            IF C_arrival >= C_last THEN
                C_current = C_arrival   –   C_last;
            ELSE
```

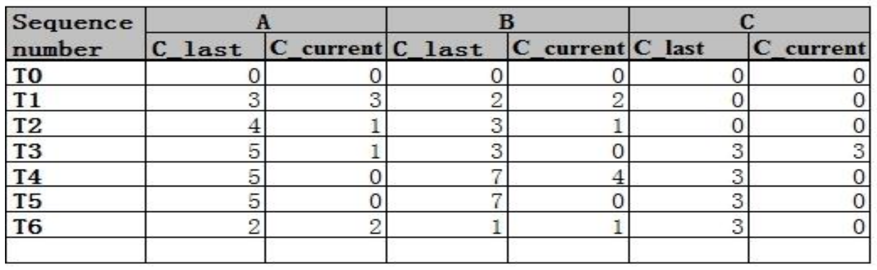
```
                /* we suppose there exists a "hidden" database status change from active to inactive
        happened in the interval between the current and last observation points. */
                        C_current = C_arrival;
                ENDIF;

            WHEN S is continuously inactive or shutdown or changed between the two states THEN
                    /* If C_arrival is not equal to 0 we suppose there exists a "hidden" database state
                    change from inactive to active happened in the interval between inactive and current
                    observation point. */
                    C_current = C_arrival;
            WHEN S is changed from inactive or shutdown to active THEN
                    C_current = C_arrival;
        END CASE;
        C_last = C_arrival; /* the accumulating counts since the value of counter is reset to 0 at most
recent time */
        /* update the last value of an object in the reference table */
        Update_last (reference_type, Ref_no, C_last) ;

        IF C>0 THEN
                InsertIntoStreams (current_sequence, Ref_no, C_current);
        ENDIF;
DONE;
```

Figure 3-2 Pseudo Code to Predict the Current Count of an Observed Object

By predicting the current counts of observed objects, we attempt to compensate for missed observations. In addition, we choose a polling interval of 6 seconds to avoid missing multiple system state changes in an interval. It is almost impossible to shutdown the database or to make it inactive twice within 6 seconds. Using prediction, we can continuously observe the history of the objects in which we are interested even when the system is inactive for a period of time or shut down. Similarly, if an object is not observed for several intervals due to overflow in the monitoring cache, we assume that there are no new occurrences of the object. However, as soon as the occurrence of the object can be stored in the monitoring cache again, we predict its current accumulated occurrences in the same way.



(a)



Sequence number	A		B		C	
	C_last	C_current	C_last	C_current	C_last	C_current
T0	0	0	0	0	0	0
T1	3	3	2	2	0	0
T2	4	1	3	1	0	0
T3	5	1	3	0	3	3
T4	5	0	7	4	3	0
T5	5	0	7	0	3	0
T6	2	2	1	1	3	0

(b)

Figure 3-3 Example of the Prediction of Observed Objects' Current Occurrences

Figure 3-3 shows an example of predicting the current occurrences of the observed objects. We suppose that the observation starts at time T0 and the observed database is continuously active from T0 to T4. Between T4 and T5, the database system is shutdown or inactive and then becomes active again at T5 (In the two continuously active periods T1 to T4 and T5 to present, there might be several missing observations of system status's change that we cannot capture). We also suppose that the monitoring cache of the database system can hold two observed objects' statistics. In the sequence of occurrences of the observed objects, their current arrival counts or the statistic values captured and accumulated inside the monitoring cache are recorded at each time point.

Figure 3-3(b) shows the predicted current occurrences for observed objects A, B and C at each observed point and the corresponding last arrival counts. We use object A to illustrate the predicting process. At T0, object A has its initial frequency, 0. During T0 to T1, the object A is observed 3 times. Thus, after T1, C_current(T1) of A is assigned to 3 (C_arrival(T1) of A – C_last(T0) of A) and the C_last(T1) of A is set to 3 (C_current(T1) of A + C_last(T0) of A). During T1 to T2 and T2 to T3, we repeat the same accumulating and predicting processes. During T3 to T4, there is no observation for object A. Thus, we do not need to process Object A and there is no change for the C_last(T3) of A. C_current(T4) of A is 0 (we do not compute the value). During T4 to T5, the database system is inactive and we do not observe the occurrences of any object. In fact, if there exists some value for an object, we suppose that there will be some missing database change from inactive to active and we will have to predict the C_last and C_current for the object to compensate for the loss. Thus, for A, C_last(T5) is equal to C_last(T4). During T5 to T6, the database is active again and we observe object A twice and C_last(T6) and C_current(T6) of A is equal to the observation times of A during the period.

The cost to directly store, sort, or search for observed objects that are physically large in size (such as the text of SQL statements) is expensive. To reduce the cost, we use a hash function to map the object to an ID (referred to as *ref_no)* that is used as a reference number in our data structures. Each element in the reference table, which is used to maintain a ordered observed object list, mainly consists of three values, the object's reference number (*ref_no*), the object body (*object_body*), and the object's_last_value (*object_last_value*) which is used to predict the current occurrences of an object.

When an object is observed, its ref_no is generated from the object body and is used as a key to traverse the reference table. If this is the first occurrence of this object, a new element is added to the reference table in the appropriate position. The table is maintained as a static linked list sorted in order of ref_nos. If the ref_no has been in the reference table, we only return the ref_no for the further generation of a new element in the corresponding data stream. To facilitate the traversal in the reference table, we divide the reference table into several small sections and keep each head pointer of the sections in memory. The reference table is periodically scanned and unused data objects are removed.

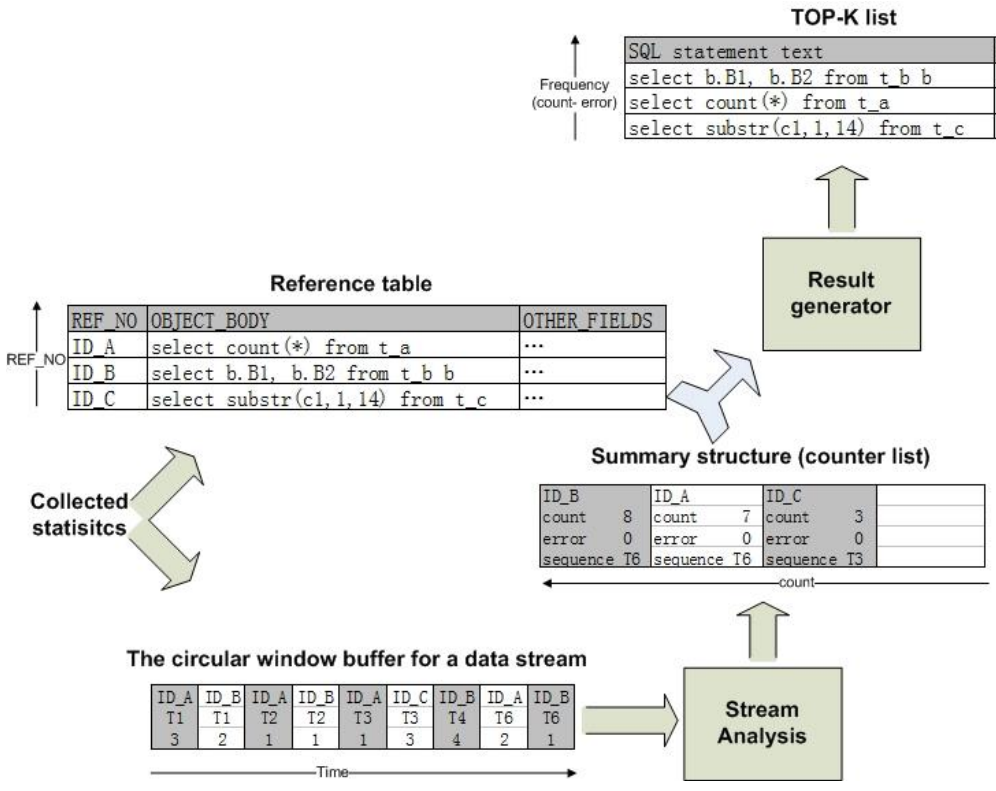

TOP-K list
Frequency (count- error)
SQL statement text | frequency
select b.B1, b.B2 from t_b b | 8
select count(*) from t_a | 7
select substr(c1,1,14) from t_c | 3
Result generator
Reference table
REF_NO
REF_NO | OBJECT_BODY | OTHER_FIELDS
ID_A | select count(*) from t_a | …
ID_B | select b.B1, b.B2 from t_b b | …
ID_C | select substr(c1,1,14) from t_c | …
Collected statisitcs
Summary structure (counter list)
ID_B count 8 error 0 sequence T6
ID_A count 7 error 0 sequence T6
ID_C count 3 error 0 sequence T3
count
The circular window buffer for a data stream
ID_A T1 3 | ID_B T1 2 | ID_A T2 1 | ID_B T2 1 | ID_A T3 1 | ID_C T3 3 | ID_B T4 4 | ID_A T6 2 | ID_B T6 1
Time
Stream Analysis

Figure 3-4 Data Structures used in Tempo

The sequence of the occurrences of the observed objects A, B and C are maintained in the various structures of Tempo shown in Figure 3-4. ID_A, ID_B and ID_C are mapped from their object bodies or object texts by a hash function and then the object bodies and IDs are stored in the reference table. Based on the IDs and their observations and time sequence, a data stream is generated into the circular window buffer structure and processed through several structures by the streaming analysis until the final TOP-K results. In the structures, the observed objects are searched and stored according to their ID, a numeric reference number. The purpose of separate structures with a key element is to reduce the storage cost and speed up searching and sorting in the structures. The details for the structures except for the reference table are introduced in the following sections.

Stream Generator

Statistics are passed to the stream generator on a FIFO (First in First out) basis and placed in a memory structure called the circular window buffer, which is used to pass the statistics to the subsequent analysis process. The circular window buffer, in which elements can be reused, makes it feasible to store an approximate history of the most important observed objects in memory.   We now only have to consider how to eliminate less important information and how to maintain the most important information in the circular window buffer. Our strategy is straightforward. We always place more emphasis on recent elements. If there is not enough space when new elements arrive, the oldest elements, that is, the elements with the smallest sequence numbers are removed from the structure to accommodate the newer elements.

To simplify the interface between the collector process and the analysis process, we use a uniform data format to represent the elements in the circular window buffer.

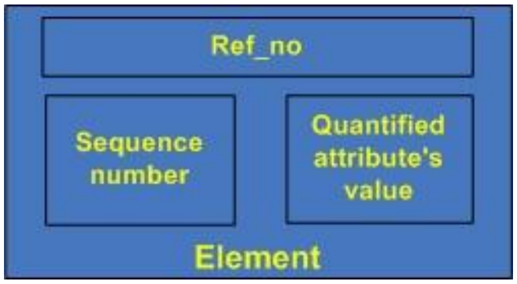


Figure 3-5 Unified Element Structure

As shown in Figure 3-5, an element consists of a reference number, ref_no, a sequence number, which is automatically generated for each element in a data stream to identify its arrival time and order, and a quantified value referred as to the evaluated attribute value in the analysis process (currently, we define the value as the occurrence counts or frequencies of observed objects). Through the use of uniform elements, adding new types of observed objects or new streaming algorithms will be achieved by simply translating and formalizing the processed data into our uniform elements. This method also insulates the analysis component from changes in data format resulting from various types of DBMS monitoring or version changes in the same DBMS.

A data stream has an associated current sequence number, indicating the time at which objects arrive in the circular window buffer. The time sequence in Tempo does not increase as a new element arrives. Instead, the occurrences of all objects from a polling action are only associated to one sequence value.

3.1.2 Online Analysis

The Online Analysis module, as shown in Figure 3-1, consists of two components, the Stream Analyzer and the Report Generator. The Stream Analyzer scans the data in the sliding windows and summarizes the data. The Report Generator generates the Top-K answers from the summaries.

Stream Analyzer

The data arriving from the Collector is a stream of uniform data elements (Figure 3-5). To cope with the rapid arrival of data, we use a fixed-size, sequence-based sliding window as we mentioned in Chapter 2. To implement a sliding window process, three basic operations are involved: insert, delete, and scan. A sliding window model is maintained on a data stream by alternately inserting a new element when it arrives and deleting an obsolete element from the window. The summary process scans the contents of the current window, thus processing the elements contained in the window.

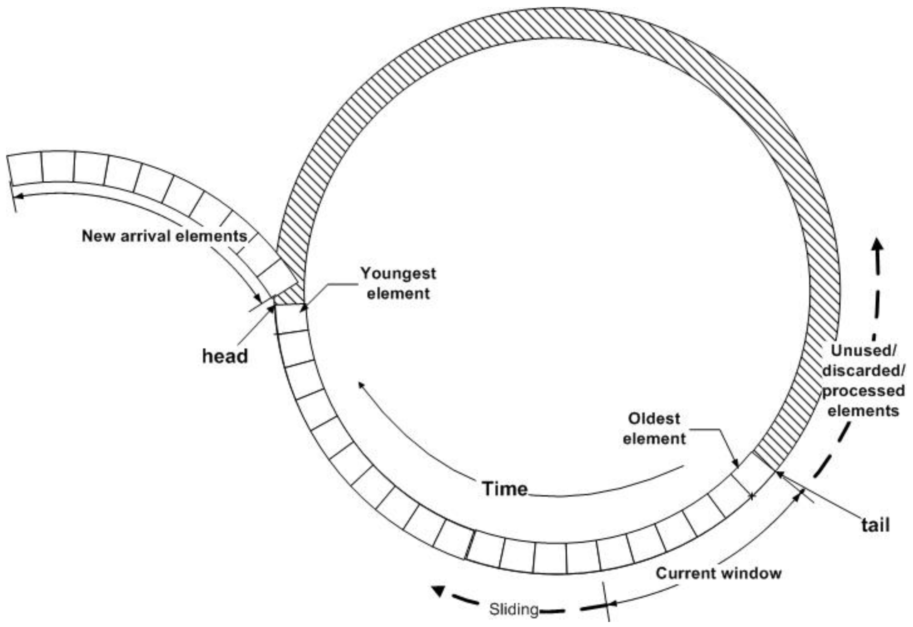


Figure 3-6 Structure for Stream Processing

As shown in Figure 3-6, elements in the circular window buffer are arranged from the tail to the head, ordered by time. The current window presents the group of elements which are currently being fed into the summary process. The size of the current window is equal to the length of the summary structure, the counter list. While the summary process is scanning the data from the tail of the window buffer to the head to perform synopsis generation, the current window is moving forward and returns the space occupied by the processed elements to the unused element area (the shaded part in Figure 3-6). When new elements arrive, if there is enough space in the unused area, we directly allocate space for them from the unused area adjacent to the current head and move the head to the new position. If the whole structure is saturated, we discard the oldest elements starting from the tail to the head and release space for new arrivals, even if the values have not yet been processed. To avoid the potential contention, the three operations on the data stream are mutually exclusive. Thus, the actions of the summary process to view the contents of the current window and the actions of feeding new arrival elements must lie in different processing cycles.

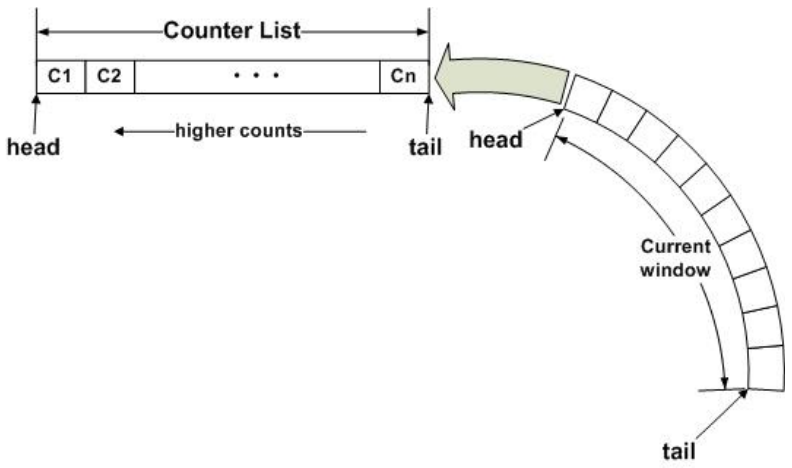


Figure 3-7 Space-Saving Algorithm

The analysis of the streaming data is done using a counter-based stream mining algorithm called Space-Saving [1], as illustrated in Figure 3-7. The Space-Saving algorithm builds an in-memory, stream-summary structure called a counter list where several counters are set to maintain the statistics of the observed objects. Each counter consists of 5 elements; an identifier of the observed object (ref_no), the approximate frequency associated with a ref_no (count), the maximum estimated error (error), and the sequence number of the last observation of the object. The counter list is maintained as a sorted list in descending order based on the current count of the objects. As elements are scanned by the summary process, the counter list is checked to determine if the element's

ID (ref_no) is in the list. If so, the current count stored for the object is increased by the value indicated in the newly arrived element. If the object is not currently found in the counter list, then it should be added. If the number of counters in the counter list does not exceed a pre-set maximum, then a new counter is allocated to represent the new observed object. The current count for this element is initialized to the value held by the newly arrived element and the counter is positioned in the counter list. The estimation error of the new element is initialized to 0.

The estimation error ε_i of an element e_i is the maximum possible error for the element. Since there is always some doubt concerning the current count of an element, the value of the estimation error must be non-negative. When a counter is first used for an observed element there is no error for the element because the counter list is not saturated and the new object has not replaced another object. At this point, their current counts are equal to their actual frequencies.

If the maximum number of counters has been reached and a new object arrives, the element stored in the tail, which has the minimum count value of the counter list, is replaced with the new arrival. The count of the tail counter increases by the new arrival value and its estimation error is set to the counter value of the object being replaced. Since the counter list is shown in descending order, the minimum count of the counter list must be the replaced one.

If the new element has not been observed previously, then the estimation error of the element is, at most, the minimum count of the counter list at the current moment. This is because the new minimum guaranteed frequency of the new element is equal to the current minimum count of the counter list, which is the count of the replaced element, plus the arrival occurrences of the new element minus the estimation error of the new element.

If the new element has been observed before, and say that at the time T_j it was replaced by another element, then its estimated error is less or equal than the minimum count at that time, min(T_j). In addition, min(T_j) must be less than or equal to the current minimum count of the counter list since the minimum count of the counter list monotonically increases over time. Therefore, the estimated error of the new element must be less than or equal to the current minimum count of the counter list.

After updating a count value for an object, its position in the counter list may need to be adjusted. In doing so, the most important elements gradually move to positions close to the head of the counter list while the less important elements move to the tail area and are eventually replaced over time.

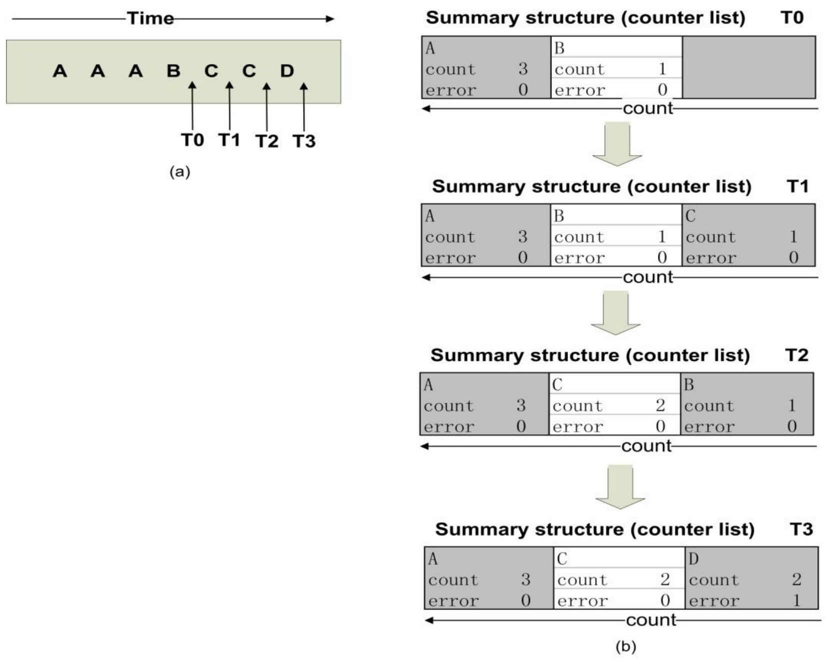


Figure 3-8 Diagram of Processing Streaming Data by Space-Saving Algorithm

Figure 3-8(a) shows a sequence of elements in a data stream and Figure 3-8(b) illustrates the summarization process of these elements in the counter list with 3 counters. At T0, two counters in the counter list are occupied to record the summarized statistics for elements A and B. When new element C arrives at T1 it is stored in the third counter and its error is set to 0. When the element C is observed again at T2, its count is increased by 1, the newly obtained count value of the element from T1 to T2. The counters are then re-sorted and element C moves forward while element B moves to the tail. When a new element D arrives there are no free counters in the counter list so the element B stored in the tail counter is replaced by the element D. The error of element D is set to 1, the count of the evicted element B, and its count is increased by 1, the occurrences of the element D.

Generation of TOP-K Answers

In order to generate TOP-K results, we traverse the counter list, and output the first K elements sorted by their minimum guaranteed frequency, which is the elements' count minus their maximum possible error. In practice, we choose the number of counters in the counter list, m, to be greater than K, since the minimum guaranteed frequencies in the first K counters are not necessarily larger than those of the K to m elements due to their possible errors. In addition, a larger number of counters in the counter list can reduce unnecessary element exchanges so as to cut down the potential error, which improves the precision of the algorithm.

```
Algorithm: QueryTop-$k$($m$ counters, Integer $k$)
begin
  Bool order = true;
  Bool guaranteed = false;
  Integer min-guar-freq = $\infty$;
  for $i = 1 \ldots k${
    output $e_i$;
    If (($count_i - \varepsilon_i$) $<$ min-guar-freq)
      min-guar-freq $= (count_i - \varepsilon_i)$;
    If (($count_i - \varepsilon_i$) $< count_{i+1}$)
      order = false;
  }// end for
  If ($count_{k+1} \leq$ min-guar-freq){
    guaranteed = true;
  }else{
    output $e_{k+1}$;
    for $i = k + 2 \ldots m${
      If (($count_{i-1} - \varepsilon_{i-1}$) $<$ min-guar-freq)
        min-guar-freq $= (count_{i-1} - \varepsilon_{i-1})$;
      If ($count_i \leq$ min-guar-freq){
        guaranteed = true;
        break;
      }
      output $e_i$;
    }
  }
  return( guaranteed, order )
end;
```

Figure 3-9 Reporting TOP-K Elements [1]

Figure 3-9 shows the algorithm to report the TOP-K elements. In Theorem 2 of the Space-Saving algorithm paper [1], the author states "Whether or not E_i occupies the i-th position in the counter list, $count_i$, the counter at position i, is no smaller than F_i, the frequency of the element with rank i, E_i". In other words, $count_{k+1,}$ the counter value of the element at the k+1 position is the upper bound of F_{k+1} and any element e_i, is guaranteed to be among the TOP-K results if its minimum guaranteed frequency, $count_i - \varepsilon_i$, exceeds

count_{k+1}, the count of the element at position k+1 where ε_i denotes the estimation error of the element e_i.

In the first loop of QueryTop-k, the minimum guaranteed frequencies of the first k elements in the counter list are compared with the count of the k+1st element. If the minimum guaranteed frequency of every element exceeds the count of the k+1st element, the algorithm can guarantee that the reported k elements are the TOP-K result and set the Boolean variable ***guaranteed*** to true. If the number of elements which can satisfy the condition are less than k, the second loop is repeated for the rest of the counter list until a element k' is checked and k' checked elements are guaranteed to be the TOP-K', where k' > k. In most cases, the algorithm only needs to finish the first loop and output only the first K elements.

The ***guaranteed*** Boolean variable in QueryTop-k denotes whether all the reported k elements are guaranteed to be among the TOP-K elements or if all the frequencies ($\text{count}_i - \varepsilon_i$) are not less than count_{k+1}, where $i \leq k$. The ***order*** Boolean variable denotes whether the reported k elements have guaranteed order; that is, if all i, $\text{count}_i - \varepsilon_i \geq \text{count}_{i+1}$, where $i \leq k$. The author provides further details on the algorithm [1].

If we only execute the first loop to produce an approximate TOP-k result (the actual number of elements output may be less than k), the complexity of the algorithm is Θ(k). If a more precise TOP-k result is required then we execute the second loop and generate k' elements, where k' >k. In the second loop, we execute, at most, an extra m-k iterations, where m is the length of the counter list, therefore, the complexity of the algorithm is Θ(m).

3.2 Revised Space-Saving algorithm

The workload in a real production environment is generally mixed and its patterns often fluctuate which renders the observed database system statistics unpredictable. If the distribution of observed objects is skewed and a small number of the more frequent elements always obtain the majority of the occurrences during the observation period, the original Space-Saving algorithm can provide an accurate answer for a TOP-K query, even when space is restricted. However, under some circumstances, the accuracy of the algorithm can be less than what we expect. Consider the following scenarios:

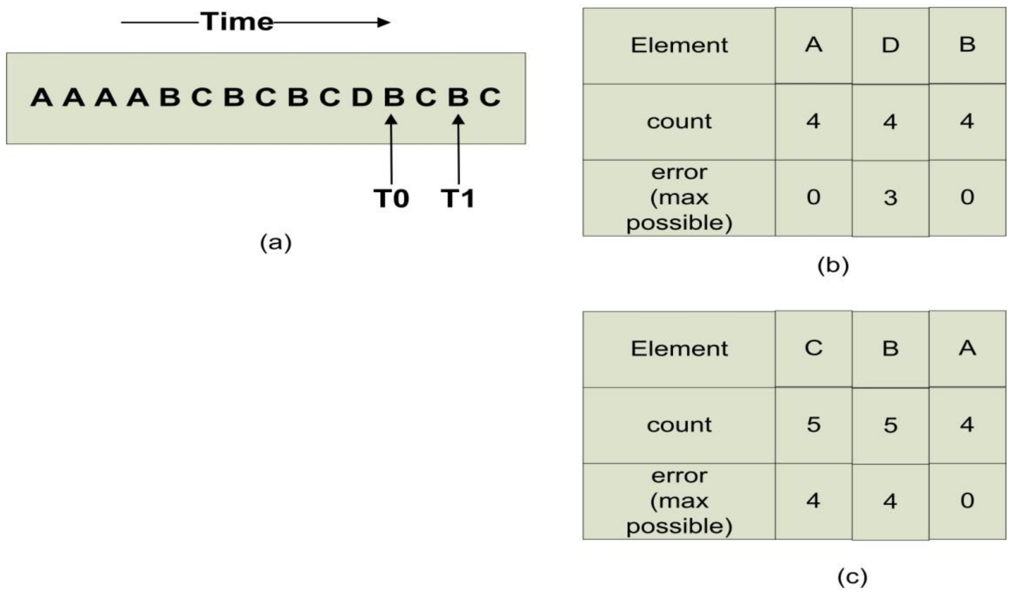
Scenario1: A large number of objects are observed, thus, the number of objects is far greater than the number of counters in the stream-summary structure.

Scenario2: The most frequently observed objects in a data stream change often.

Scenario3: Objects are observed frequently at the beginning of the stream and then cease to occur.

In these three scenarios, it is likely that observed objects that appear frequently earlier in the stream and have not occurred for a long period of time still occupy the higher position(s) in the stream-summary structure. Even if the counts of objects that arrive later reach similar values, they likely do not move into the positions occupied by the older objects due to the limited view of the current window compared with the whole size of the observed data stream and the strategies of the Space-Saving algorithm such as only replacing the tail counter and special error computing methods. Unfortunately, we cannot guarantee the order of the final TOP-K elements if the underlying workload is similar to these scenarios.

For example, consider the sequence of elements in Figure 3-10(a).



Element	A	D	B
count	4	4	4
error (max possible)	0	3	0

Element	C	B	A
count	5	5	4
error (max possible)	4	4	0

Figure 3-10 An Example to Illustrate the Problems with the Original Space-Saving Algorithm

For a list of 3 counters Figure3-10(b) and Figure3-10(c) show the contents of the counters at time T0 and T1, respectively. At T0, although element B has been in the stream for the fourth time, the head position is still occupied by the element A. At T1, based on Space-Saving Algorithm, the TOP-K result is ACB but the results are not the correct order, which is BCA. If we replace element A or D, instead of B when element C arrives after time T0, the problem is alleviated. However, this knowledge is dependent on past or future experience and is intractable in the real-time stream style of processing. To handle these problems, we introduce revisions to the original Space-Saving algorithm. The revisions are based on the following three assumptions:

1. The more recently observed elements are more likely occur in the future.
2. Stale elements can be made fresh by losing parts of their counts.
3. The maximum estimated error can be proportionally reduced after a portion of counts of an observed object have been faded away.

3.2.1 Fading Techniques for Processing Streaming Data

When we observe the history of objects on a data stream, we believe that more importance should be placed on recent element occurrences as opposed to those in the past. For example, Chen et al. [34] use a tilted-time window to allocate more units of the summarization of recent observations and fewer units for the past. Giannella et al. [7] propose a fading framework in which the faded percentage, FP, is computed based on a fading factor and is used to vary the window size and discount the support of the past transactions.

One difference between our fading framework and that of Giannella is that our fading factor is applied to the summarized results and periodically discounts the counts of the elements in the summary-structure as opposed to each primitive object as soon as their new occurrences arrive at the current window. Also, unlike Giannella’s fading framework in which the discount percentage for the counts of the observed objects in a certain past period are fixed, our approach keeps the past counts of the objects without or with less discount or fading which are continuously observed from the past to the present. The purpose of the fading factor in Tempo is to avoid the stagnation of information in the counter list, and to make the counter list more sensitive to changes in of the observed data stream by placing more emphasis on new observations.

```
Algorithm: Revised-Space-Saving (m counters, stream S)
begin
for each element, e with its occurrences of the current window, o in S {
    If e is monitored {
        Let $count_i$ be the count of the counter of e
        Increase the $count_i$ by o;
        Assign current sequence number to the sequence number of $counter_i$
    } else { //The replacement step
        Let $e_m$ be the element with least hits, min
        Replace $e_m$ with e;
        Increase the $count_m$ by o;
        Assign $\varepsilon_m$ the value min;
        Assign current sequence number to the sequence number of $counter_m$
    }
}// end for
end;
/* to avoid frequent fading and re-sort the counter list we periodically do fading and
re-sorting */
When a schedule is triggered
Boolean FlagDiff;
Int Temp_count;
Double remaining_per /* What percentage of the counts of an element will be
faded away */
for each counter in the counter list{
    Let $count_i$ be the count of the counter we just processed
```

```
        If the sequence number of counter$_i$ is not equal to the current sequence {
            remaining_per = Compute_fading (current sequence number - the
        sequence number of counter$_i$ );
            Temp_count = round(remaining_per * count$_i$)
            If (Temp_count != count$_i$) {
                count$_i$ = Temp_count;
                $\varepsilon_i$ = $\varepsilon_i$ * remaining_per;
                Assign the current sequence number to the sequence number of
        counter$_i$
                From the position of the counter to the tail of the counter list,
        searching another counter, m whose count is less or equal then count$_i$;
                If (There exists a counter$_m$)
                    Place counter$_i$ before counter$_m$;
                else
                    Place counter$_i$ at the tail
            }
        }
    }// end for
    End;
```

Figure 3-11 Revised Space-Saving Algorithm

The revised algorithm is shown in Figure 3-11. The bold sections indicate the additions made to the original Space-Saving Algorithm. First, when a counter is replaced in the stream-summary structure, or a new occurrence of an observed object is added, we set/reset the sequence number of a counter with the current sequence number in the observed data stream. In doing so, we attach a sequence number to each of the observed elements which helps to quantify the recency of the frequency of an element. Second, based on the difference between the last sequence number of each counter and the current sequence on a data stream we periodically estimate how much of their counts should be faded away. The formula to compute the remaining percentage (RP) is defined as follows:

$$RP=\left(1-\frac{a}{100}\right)^{\log_{10}^{C.s-e_i.s+1}} \quad \textbf{(1)}$$

where a denotes the fading factor which can be varied in terms of the observed stream and is a percentage from 0 to 100; e_i denotes the $i-th$ element in the counter list; $e_i.s$ denotes the sequence number stored in e_i and $C.s$ denotes the current sequence number of the observed data stream. The use of a logarithm exponential in the fading formula dampens excessive fading.

The formula to compute the remaining frequency for an object is defined as follows:

$$remains(n)=\left(remains(n-1)+\sum_{j=1}^{m} f_j\right)\times RP \quad \textbf{(2)}$$

Where remains(n) denotes the remaining frequency value of an observed object after n-1 fading intervals and remains(n-1) is the remaining value after the last fading checkpoint. $\sum_{j=1}^{m} f_j$ denotes the sum of the observations of the object during the $n-th$ checking interval. The $e_i.s$ in RP is the sequence value of the object at $m-th$ observation (the most recent observation) in the interval.

With our fading strategy, frequently observed elements can maintain their accumulated frequency values so they are more likely to remain resident in the counter list. On the other hand, the frequency values of stale elements drop dramatically due to the frequent losses in the weight of their frequencies and these items are more likely to be eliminated. At each fading checkpoint, there are always some differences between the current sequence number on the processed data stream and the sequence numbers of these elements stored in the corresponding counters and the remaining percentage of the frequency values is computed based on the differences.

One of the most important reasons for using the Space-Saving algorithm is to maintain historical information regarding the observed database. That is, we need to maintain the most important information associated with an observed object from the beginning of the monitoring process. Therefore, when we fade away some of an element's counts, we attempt to prevent its remaining counts from excessively fading in the near future by resetting its sequence number to the current sequence of the observed stream. As such, at any time, as soon as an observed object's frequency increases, resulting from new occurrences in the current window, we reset its sequence value. That is, we regard any object which is, at present, updated as an important element. A certain percentage of its count is obtained as a result of the most-recent summarization whereas other parts can be traced back to its past activities. All occurrences are of equal importance since the less important part of the past information has been faded away at previous fading checkpoints.

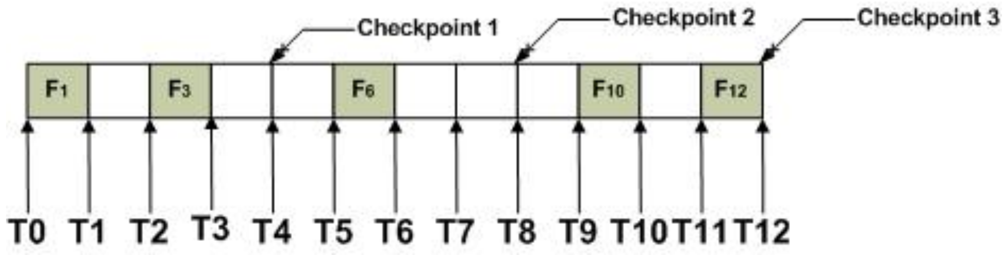


Figure 3-12 Fading Example

To clarify the difference between our fading approach and Giannella's, we show an example in Figure 3-11. There are twelve observation intervals from T0-T1 to T11-T12 and three fading checking points are set at T4, T8 and T12. We use periodic fading instead of fading every interval, because frequently computing faded values for every observed object and re-sorting them in the counter list is expensive. In the twelve intervals, we observe the object's frequencies F_1, F_3, F_6, F_{10} and F_{12} times in intervals T0-T1, T2-T3, T5-T6, T9-T10 and T11-T12, respectively. Based on the Giannella's fading method, the faded occurrence of the object from T0 to T12 is

$$faded' = F_1 \times b^{T12-T1} + F_3 \times b^{T12-T3} + F_6 \times b^{T12-T6} + F_{10} \times b^{T12-T10} + F_{12} \times b^{T12-T12}$$

where $b \in (0,1)$ and the remaining number of occurrences for the period up to T12 is

$$\mathrm{F}_1 + \mathrm{F}_3 + \mathrm{F}_6 + \mathrm{F}_{10} + \mathrm{F}_{12} \text{ - faded'} + \text{remains}(0)$$

where remains(0) is the remaining occurrences of the object before T0.

In our method, we recursively compute the remaining value instead of the faded occurrence. The total remaining occurrences after fading at T12 is

$$remains(3) = remains(2) = F_{10} + F_{12} + (remains(1) + F_6) \times \left(1 - \frac{a}{100}\right)^{\log_{10}(T_8 - T_6 + 1)}$$

where a is a fading factor ranging from 0 to 100. There is no loss from T8 to T12 due to the fact that the object has been observed at checkpoint 3, T12, where

$$remains(1) = (remains(0) + F_1 + F_3) \times \left(1 - \frac{a}{100}\right)^{\log_{10}(T_4 - T_3 + 1)}.$$

The formula to compute what percentage of occurrences of an observed object should be kept is described later.

3.2.2 Revised Summary-Structure and TOP-K Generation Process

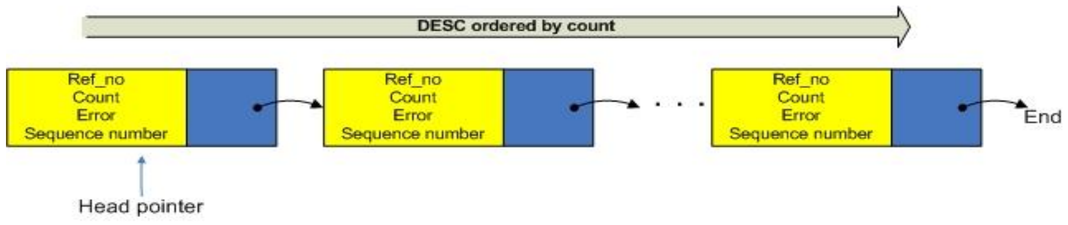


Figure 3-13 Stream-Summary Structure

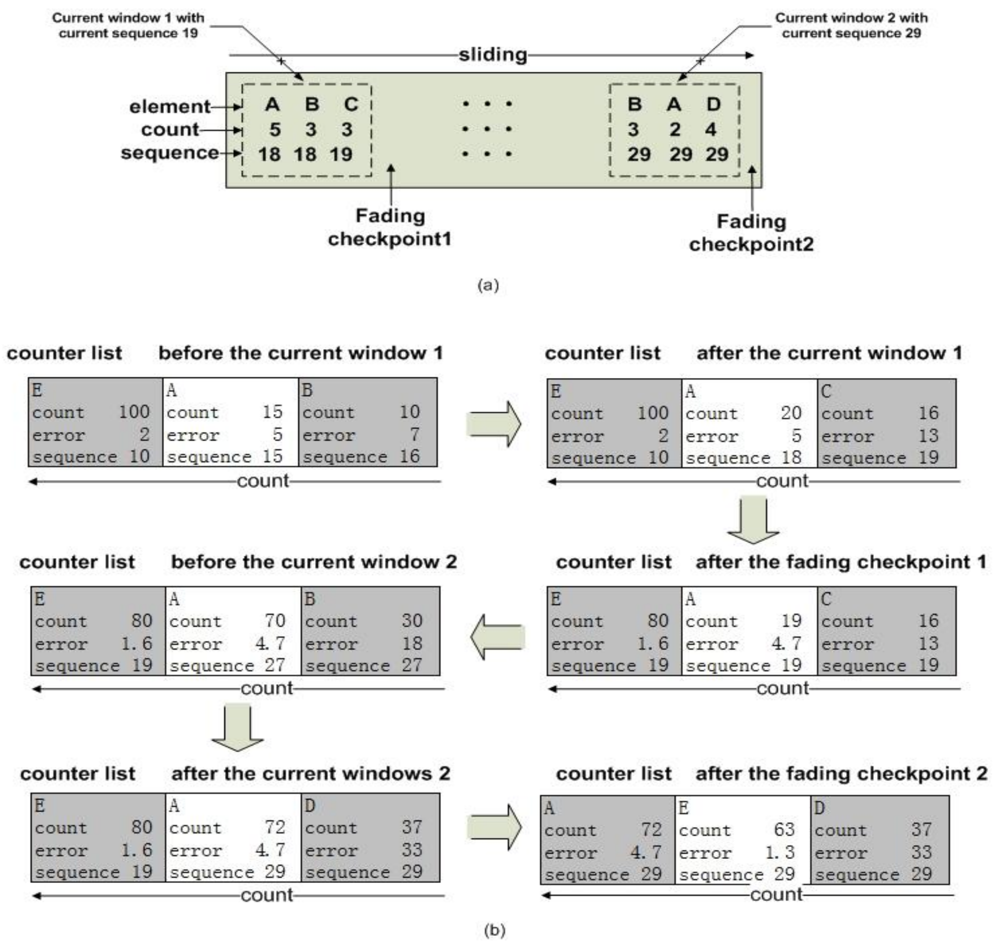
The stream-summary structure consists of a number of counters which are used to record the summary of the observed objects. The structure used in Tempo is a static linked list as illustrated in Figure 3-13. In revising the Space-Saving algorithm we chose to re-implement the summary structure (originally a double dynamic linked list with a parent bucket which links the elements with same count value together [1]) as a singly linked list. The reasoning for this change is based on the following:

1. To monitor database activities, the size of the counter list should be relatively small. The cost for locating a position in a singly linked, sorted list is not expensive and the singly linked list structure saves on the space of one pointer per element.

2. Compared with the dynamic linked list, all elements of the static linked list can be allocated in advance, which eliminates the cost of frequent system calls.

3. During a long monitoring period, many observed objects are unlikely to share the same counts. Therefore, it is unnecessary to allocate an extra bucket for each separate linked list in which elements share the same counter value.

3.2.3 An Example of the Revised Space-Saving Algorithm



counter list before the current window 1

E	A	B
count 100	count 15	count 10
error 2	error 5	error 7
sequence 10	sequence 15	sequence 16

counter list after the current window 1

E	A	C
count 100	count 20	count 16
error 2	error 5	error 13
sequence 10	sequence 18	sequence 19

counter list after the fading checkpoint 1

E	A	C
count 80	count 19	count 16
error 1.6	error 4.7	error 13
sequence 19	sequence 19	sequence 19

counter list before the current window 2

E	A	B
count 80	count 70	count 30
error 1.6	error 4.7	error 18
sequence 19	sequence 27	sequence 27

counter list after the current windows 2

E	A	D
count 80	count 72	count 37
error 1.6	error 4.7	error 33
sequence 19	sequence 29	sequence 29

counter list after the fading checkpoint 2

A	E	D
count 72	count 63	count 37
error 4.7	error 1.3	error 33
sequence 29	sequence 29	sequence 29

Figure 3-14 Diagram for Revised Space-Saving Algorithm

Figure 3-14 demonstrates the fading process using an example. The observed data stream is shown in Figure 3-14(a). The current window size is three, equal to the size of the counter list. The sequence number of the observed stream starts at 0 and is increased over time. The current window of the stream slides forward, summarizing the stream's contents. In Figure 3-14(b), the summarization processes for the windows 1 and 2 are shown. During the processing of the data stream, the fading schedule is triggered every 10 sequence numbers.

In the example, we chose two fading checkpoints, checkpoint 1 and checkpoint 2, at sequence numbers 19 and 29, respectively. When the current window 1 is processed, element A is already in the counter list so the count of element A is increased by the new arrival value 5 and its sequence number 5 is replaced with 15. Similarly, element B records its new occurrences. Eventually, element C arrives, replacing element B and the count value of the counter which represents the element C is increased by 3. Its error is replaced with the evicted element B's count 13 and its sequence number is set to the sequence number of element C in the stream. When the data stream reaches the first fading checkpoint, the fading process is triggered. We use the fading formula to compute the remaining percentages for the all elements in the counter list. For instance, the RP of E is $\left(1-\frac{20}{100}\right)^{\log_{10}(9-10+1)}$=0.8, where the fading factor is 20. Therefore, the remaining count of the element E to be faded is 100 * 0.8 =80. Since the count of the element E is changed, we need to proportionally fade away the error of the element E and get the remaining error value 1.6. When computing the minimum guaranteed frequency, we round the error value to an integer value. Since the sequence number of the element C is equal to the current sequence number of the stream, there is no fading for the element. The remaining count of the element C is $round\left(20\times\left(1-\frac{20}{100}\right)^{\log_{10}(9-18+1)}\right)$=19 and its remaining error is 4.7. Similarly, we execute the fading process again when the stream reaches the fading checkpoint 2. The difference between the processes at the two checkpoints is that after fading at the second checkpoint, the count of element E is less than that of element A, therefore, re-sorting occurs and element A occupies the head position in the counter list and the element E recedes into the second position.

3.3 Implementation

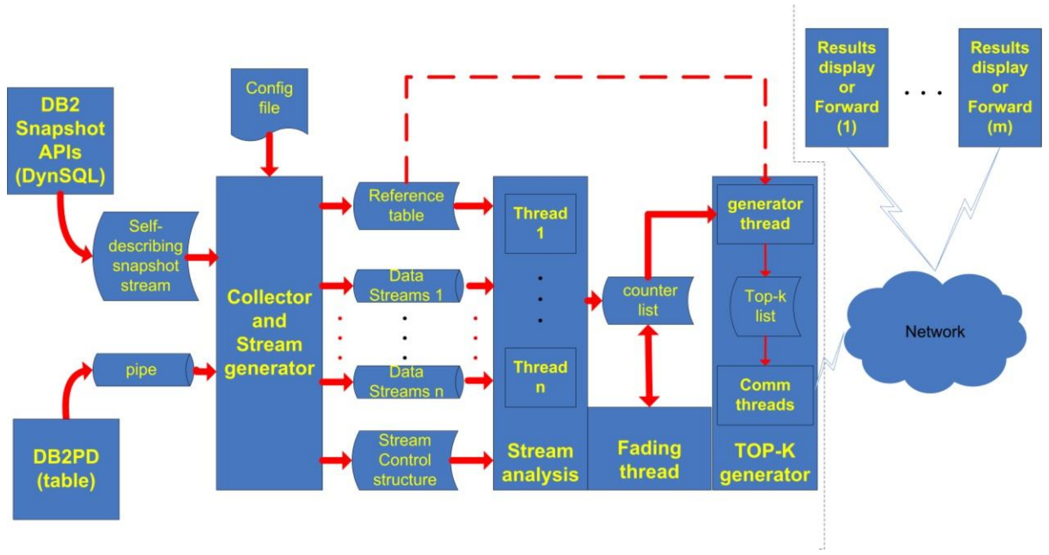


Figure 3-15 Structure of Tempo Prototype

Figure 3-15 presents the structure of the Tempo prototype. Our prototype implementation specifically monitors the TOP-K most frequently executed SQL statements and the TOP-K most frequently accessed tables. Our approach can be generalized to produce TOP-K results for any metrics that can be monitored.

Our prototype system, Tempo is implemented using the programming language C and UNIX shell on a machine running LINUX. It is linked to the IBM DB2 management system through IBM's C routine libraries by using –ldb2 parameter when compiling related C code. In addition, to support multi-threading in LINUX environment, we also link POSIX pthread libraries by using –lpthread parameter when compiling them. Tempo consists of three main modules: lightweight monitoring, stream analysis and TOP-K generation. Each module is executed concurrently by a specific group of processes. For speeding up the processing of stream analysis and TOP-K generation, parallel threads are used to process same functions concurrently.

3.3.1 Lightweight Monitoring

As shown in the left-most part of Figure 3-15, we use different tools to monitor the accessed tables and the executed SQL statements. To capture the statistics relative to the accessed tables, we employ DB2PD (see Appendix C for details) which monitors database activities by invoking the snapshot API (see snapshot API C samples in IBM DB2 information center [18] for using these API C routines) and provides users with a formatted output. Results are accessed by opening files or pipes. DB2PD can be used to obtain information regarding the status of the observed database system, which is used to control streaming generation and predict the current occurrences of the observed objects. The system status information is translated into the format <status_flag, sequence_num, status_value>. This is similar to the structure of the observed objects in the data streams, <ref_no, sequence_num, quantified_value> and is also stored in the data streams.

DB2PD does not provide information associated with executed SQL statements and their activities. To get this information, we employ snapshot APIs to poll their execution actions and then parse a self-describing data stream in the user-defined buffer (see

Appendix B for details). To be more specific, we first locate the SQLM_ELM_DYNSQL logical data group in the self-describing data stream and then extract the texts of the observed dynamic SQL statements and their execution times respectively from elements SQLM_ELM_STMT_TEXT and SQLM_ELM_NUM_EXECUTIONS under the logic data group.

3.3.2 Stream Analysis

To rapidly process multiple data streams simultaneously, we run multiple analysis threads with each thread serving one data stream. When an analysis thread scans the current window of a data stream, it copies the contents of the current window into its local buffer. During scanning, the thread acquires a lock in order to block access to the current window by other processes. We use the revised Space-Saving algorithm outlined above to summarize the contents stored in the local buffer and maintain the summarized information in the counter list. When the information in the counter list is updated by newly arrived elements, another lock is held by the thread. This is used to avoid contention among the analysis thread, the fading thread, and the TOP-K generator thread.

To fade away partially stale counter values of the observed objects, we periodically run another thread, the fading thread, to scan and hold the lock for access to the elements in the counter list. During this period, the fading thread computes the fading percentage to determine how many counts for each element should be faded away in the terms of the difference between the sequence numbers in the counters and the current sequence number of the data stream. Updating the elements' counts and errors is based on the above computed fading percentage and the counter list is sorted based on the new counter values of the elements across the counter list. To avoid unnecessary computing and traversing across the processed counter list (according to the fading formula, the updating for the counts of some elements rarely happens when the sequence difference is fairly small), we set the interval for triggering the fading schedule to a longer time.

3.3.3 TOP-K Generation

The QueryTop-k function in the original Space-Saving algorithm reports the TOP-K elements in Θ(m), where m is the size of the counter list. Of the first k elements of the counter list, not all of them can be guaranteed to be in the TOP-K result with their minimum guaranteed frequency $count_i - \varepsilon_i \geqslant count_{k+1}$, where $i \leqslant k$. Therefore, the results reported by the original Space-Saving Algorithm are approximate. In our approach, we use a selection sorting algorithm to directly re-sort the counter list by the minimum guaranteed frequency, and output the first k elements. Since the fact the $count_i - \varepsilon_i$ is extremely similar to $count_i$ when the m is enough large, and the original counter list is ordered by count values of the elements, we can, therefore, regard the counter list as almost sorted by the minimum guaranteed frequency and the complexity of reporting TOP-K elements still remains Θ(m).

TOP-K generation is based on the requests from clients. Once the communication thread receives a request with parameters K and the object type (0: Dynamic SQL statements, 1: Table), it forwards the request to the generator thread. The generator thread allocates a temporary list and re-sorts the corresponding elements in the counter list.

After that, it looks up the ref_ids of the chosen candidates in the reference table and replaces the ref_ids with the corresponding object texts. Eventually, it stores the re-sorted TOP-K result into the temporary list and the communication thread, which made and forwarded the original request, returns the results to the client.

Chapter 4

Experimental Evaluation

4.1 Purpose of Evaluation

In this section, we conduct a set of experiments to evaluate the system with the following criteria:

1. The overhead introduced by Tempo.
2. The impact on throughput.
3. System accuracy in generating TOP-10 executed SQL statements and accessed tables.

We examine a number of factors that affect system accuracy, including:

1. The number of counters in the summary structure.
2. The variation of data patterns, which is variation in the frequency of objects over time.
3. The size of the fading factor.

4.2 Experimental Environment

4.2.1 System Platform

The experiments are run on an Intel Core dual 2.66 MHz processor with 4 GB of memory running Linux 64-bit CentOS 5 and DB2 v9.5. Four RAID (level 0) physical volume groups provide space for the database system, such as data, indexes, catalog, temporary tablespaces and database logs. Each volume group includes 3 striped SATA disks. The local disk on the machine is used to hold the OS and Tempo system and to store the statistics and trace log produced by our prototype system in our experiments. The 2.5G memory of database instance is allocated when DB2 starts up. The database contains a 10G data tablespace across three RAID groups and a 2G index tablespace allocated in another RAID group. All tablespaces are configured with Direct I/O and raw device to improve database performance.

4.2.2 Workloads

We use a mix of two workloads in our experiments. The main workload is an Online Transaction Processing (OLTP) workload generated by a TPC-C [31] like driver, called DTW [19] (see Appendix A for details). The DTW workload accesses 9 tables via 30 unique SQL statements. This number of objects was found to be insufficient for our experiments, since the trend of the accuracy of Tempo is required to be observed when the number of the unique objects in the monitored data stream far exceeds the number of the traced elements in the counter list. We therefore add a custom workload (see Appendix E for details) in which 70 additional SQL statements are randomly executed. Each execution of an additional SQL statement brings an access to an additional table. In the overhead evaluation, the DTW workload is mainly observed by Tempo and the custom workload is only used to saturate the Stream-Summary structure during the initialization period. During the accuracy evaluation experiments, the percentage of the workload mix is varied according to the different experimental requirements.

4.3 Overhead Evaluation

4.3.1 Monitoring Approaches (Experiments)

We compare the overhead of Tempo with other monitoring solutions. As a baseline, we measure the overhead of running the DTW workload alone and then measure the overhead in a number of different monitoring configurations. We conducted experiments with the following configurations:

Configuration 1: Baseline measurement

This configuration is used to obtain a baseline measurement of system overhead and DTW throughput and is used to compare with the other configurations.

Configuration 2: Tempo without fading

In this configuration, we run the DTW workload and Tempo with 25 counters (we found in our experiments that 25 or more counters provide an acceptable level of accuracy. The number of counters in the counter list has an indirect effect on the initial size of the internal reference tables, thereby influencing the amount of IO and CPU overhead.

Configuration 3: Tempo with fading

In this configuration, we run the DTW workload and Tempo with fading. We add the fading feature to determine if our fading function introduces extra overhead.

Configuration 4: DB2TOP (background mode)

To make a fair comparison, we run DB2TOP (see Appendix A for details) only with the parameters “dynamic SQL” and “table”. When DB2TOP is running, TOP-10 information relative to executed SQL statements and accessed tables is generated and recorded into files. At the end of the testing, we sum the size of the files to compute the cost of extra storage usage for different approaches.

Configuration 5: DB2TOP (collector mode)

As in Configuration 4, only the parameters “dynamic SQL” and “table” functions are used when running DB2TOP in this experiment. In collector mode, DB2TOP does not directly generate TOP-10 executed dynamic SQL statements and accessed tables in real time, but instead it collects enough data that can be analyzed offline by local or remote network processes in the future. We use the local analysis in our comparison. The remote processing still requires extra network buffers, temporary storage and other network resources. Eventually, the same amount of collected data has to be temporarily stored and transmitted and the extra costs caused by the introduction of the analysis via network are similar to the local analysis.

We compare our approach with DB2TOP, because it shares similar functionality with Tempo. It is relatively lightweight compared with other monitoring tools such as snapshot with catalog tables, snapshot CLP, and event monitoring. Like Tempo, it is memory-based, using memory-based database snapshot APIs and maintaining analyzed statistics in main memory. However, the temporary results, such as the current TOP-K information must be output and stored to disk. Therefore, we believe that the comparison of overhead and impact on the observed DB2 system between Tempo and DB2TOP is reasonable. We do not compare against the other monitoring methods mentioned above, because the extra overhead incurred by these approaches is much larger than the overhead with DB2TOP and Tempo.

The random execution of transactions in the DTW workload means that individual runs will always vary to some degree. We take several steps to minimize the difference between experiments. First, each experiment consists of ten runs and the average overheads and related standard deviations are computed as the final experimental results. Second, prior to each run, the DTW database is restored to its initial state. Third, we use the same transaction ratio of the DTW workload in each run. Fourth, each run lasts for 30 minutes which is long enough to smooth out the differences in actual transactions executed. To avoid system overload, which can incur incorrect estimates of the use of system resources, we execute the DTW workload with only 4 clients.

4.3.2 Experimental Metrics and Collection Method

Overhead Measurement

The most widely used metrics for evaluating system overheads are CPU, IO and memory utilization rates. In our experiments, the Linux command "*sar"* provides us with the CPU, IO and memory measurements. Because the data are distributed across several disks, we first calculate the IO utilization rate for each disk device including raid disk array and local disk devices and then compute the average IO utilization rate across all disk devices. We determine how much physical memory is used by subtracting the amount of buffers and cache from the used memory (effective free memory) [16]. We collect the statistics about CPU, IO and memory utilization after a five minute initialization period. Since the memory used for the database system belongs to the cached memory usage and the total database memory is configured as a fixed amount in advance, the effective used memory we calculate in our experiments excludes this part. To be more specific, we use used memory - buffered memory - cached memory to compute the effective used memory. In addition, to avoid the difference in the different experimental run, we execute the specific command as root privilege to clean out the memory used to buffer and cache the file systems in the initialization period.

Throughput Measurement

In order to evaluate the impact of Tempo on the observed system, we use the throughput of the DTW workload, or the number of transactions completed per second. The system requires a “warm up” period (five minutes) before its performance stabilizes. During this period the database bufferpools need to become occupied; memory structures are initialized and the DTW clients start up. We do not collect and compute various statistics during this period. We compute the average transaction rate over a 10 minute period when the system is saturated and all system recourses are fully utilized, but not overloaded.

Extra Space Measurement

Finally, we observe the amount of used disk space required to continuously determine the TOP-K information by invoking the Linux command, “ls” to obtain the size of the collected files (Appendix D shows the details about how to use DB2PD to collect necessary statistics in the files).

The statistics to estimate CPU, memory and IO overheads and the throughput of the DTW workload are collected by repeatedly invoking the system monitor command, “sar” and scanning the DTW report every 10 seconds. During the testing period, we use a 6 second interval for the snapshot monitoring to poll database statistics in Tempo and DB2TOP.

4.3.3 Overhead Results and Analysis

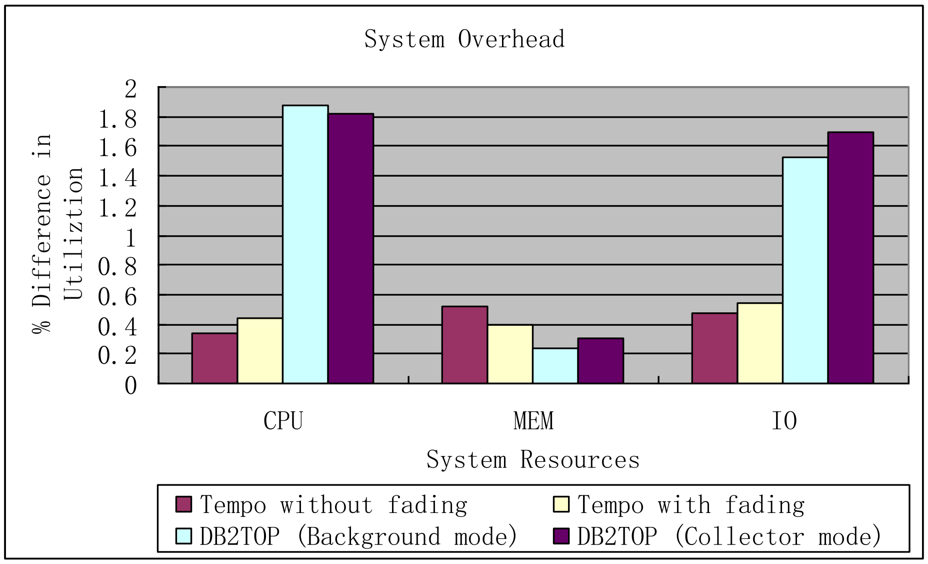


Figure 4-1 Overhead Comparison of Tempo vs. DB2TOP

Configurations	CPU utilization (%) and increment (%)	CPU standard deviation	Memory utilization (%) and increment (%)	Memory standard deviation	IO utilization (%) and increment (%)	IO standard deviation
Baseline measurement	86.32(0.00)	0.19	6.34(0.00)	0.02	20.43(0.00)	0.23
Tempo without fading	86.66(0.34)	0.22	6.86(0.52)	0.37	20.91(0.48)	0.33
Tempo with fading	86.76(0.44)	0.21	6.74(0.40)	0.04	20.97(0.54)	0.25
DB2TOP (background)	88.20(1.88)	0.35	6.58(0.24)	0.02	21.96(1.53)	0.36
DB2TOP (Collector)	88.14(1.82)	0.67	6.64(0.30)	0.18	22.12(1.69)	0.16

Table 4-1 System Overheads and Related Standard Deviations

Figure 4-1 shows the percentage increase of CPU, memory and IO utilizations in configurations 2, 3, 4 and 5 as compared with the utilizations in the base configuration. Table 4-1 shows the detailed utilization rates and percent differences and standard deviations. In terms of CPU utilization rate, we see that the increase in cost due to Tempo in configurations 2 and 3 is negligible. The CPU utilization of Tempo with and without fading rises 0.34% and 0.44% respectively. The extra CPU cost in Tempo with fading is a little higher, because in fading mode, periodically computing the fading value for each counter and further sorting and updating elements in the counter lists consumes some CPU resources. For the DB2TOP configurations, as shown in configurations 4 and 5, the CPU overhead increases are about 4 times greater than with Tempo.

Since the amount of memory allocated to the observed database is fixed and the memory uses of the compared monitoring tools are similar, we exclude the used memory for the database system from the comparison. Comparing the memory utilization, Tempo requires an extra 0.4% - 0.52% memory (about 16M) above the base configuration to build its summarization lists, streaming windows, reference tables and other internal structures. It also occupies more memory than DB2TOP. However, the size of the extra memory space is relatively small. In addition, we allocate and free memory occupied by reference tables in block units instead of in element units as in DB2TOP so our memory management strategy reduces the number of system calls and the consumption of CPU resource.

DB2TOP has higher IO costs than Tempo. Tempo only incurs an extra 0.5% in IO demand, compared with a 1.53% increase for DB2TOP. For DB2TOP in collector mode (configuration 5), the IO overhead is more than 1.69% due to the fact that each system statistic it gathers is written to disk, which is more expensive than DB2TOP in background mode where temporary statistics relative to TOP-K information are periodically summarized in memory. The extra IO overhead of Tempo is due primarily to the PIPE communication between DB2PD and our collection process, which is in charge of collecting database statistics for accessed tables and generating the data stream of information. If we only use snapshot APIs to implement this function instead of DB2PD, as we do for executed SQL statements, we can further cut the IO costs in Tempo.

The standard deviations for IO, CPU and memory in the different runs are small. This means that the average values reported are valid performance indicators.

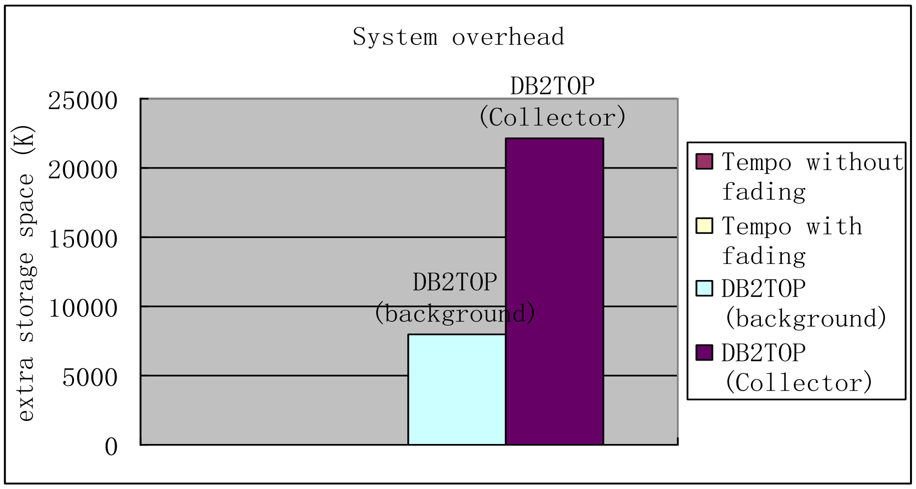


Figure 4-2 Extra Disk Space Required for Running DB2TOP

Configurations	Extra Storage Space (K)	Standard Deviation
Baseline measurement	0.00	0.00
Tempo without fading	0.00	0.00
Tempo with fading	0.00	0.00
DB2TOP (background)	7934.18	64.21
DB2TOP (Collector)	22116.73	145.50

Table 4-2 Extra Storage Space under Different Testing Conditions and Related Standard Deviations

Figure 4-2 shows the extra storage space required using the different monitoring approaches. Related details and standard deviations from 10 runs are shown in Table 4-2. Tempo does not require any additional storage space to store temporary data since it is kept in main memory. DB2TOP, on the other hand, requires a large amount of disk space to complete similar tasks. Comparing the two versions of DB2TOP, the collector mode requires more disk space than the background mode. The reason is that, unlike DB2TOP background mode, which only writes current statistics in database current monitoring heaps into files, the DB2TOP collector mode stores the history of the information during the whole test period. In our experiments, running DB2TOP in collector mode for 30 minutes requires an extra 22MB of disk space to keep the historical information.

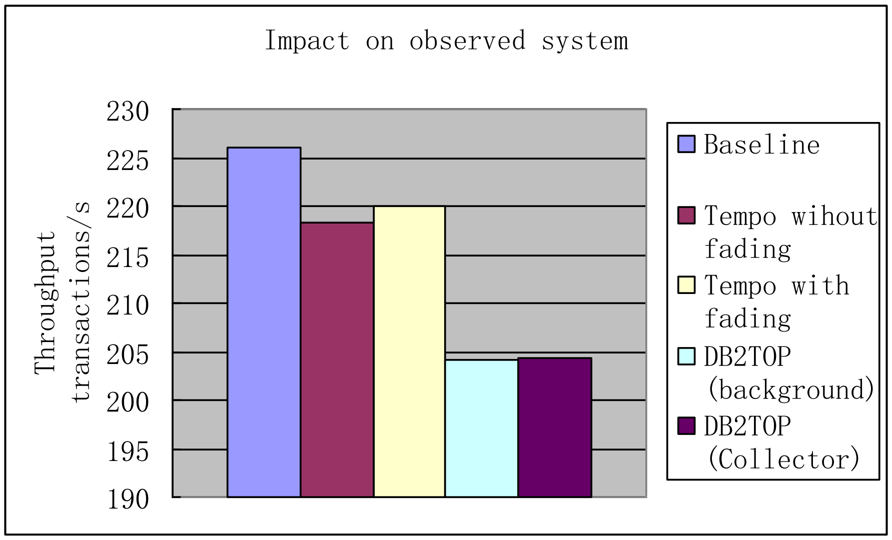


Figure 4-3 Throughput Comparison between Tempo and DB2TOP

Configurations	Throughput (transactions/s)	Throughput standard deviation
Baseline measurement	226.11	2.19
Tempo without fading	218.35	4.52
Tempo with fading	219.93	0.53
DB2TOP (background)	204.06	1.36
DB2TOP (Collector)	204.33	0.76

Table 4-3 Throughputs under Different Testing Conditions and Related Standard Deviations

Figure 4-3 and Table 4-3 show the throughputs when we use the different monitoring approaches. The average throughput resulting from executing the DTW workload alone is 226.11 tps (transactions per second). Tempo with fading and without fading decreases the DTW throughput 2.73% and 3.43% to 219.93 tps and 218.35 tps, respectively. This is due to opening the monitoring switches for dynamic SQL and tables. Running DB2TOP on the DTW workload decreases throughput by about 22 tps or 9.7%. DB2TOP has more impact on the observed system due to the fact that it opens more monitoring switches and, as observed in the previous experiments, has higher CPU and IO overhead.

In Tempo, we only open the necessary switches for polling the required system statistics. However, in DB2TOP, all 8 switches are always opened during execution. We also find that DB2TOP requires more critical system resources, such as IO and CPU than Tempo so it is more likely to incur resource contention.

4.4 Accuracy Evaluation

4.4.1 Purpose of Accuracy Evaluation

Tempo is built on the Space-Saving algorithm and uses database snapshots to simulate a data stream. Both of these facts introduce inaccuracies which may lead to incorrect answers for the TOP-K information in Tempo. The purpose of this set of experiments is to examine the amount of inaccuracy introduced in the results. The inaccuracy is determined by comparing two instances of Tempo, which we call the standard instance and the testing instance. In the standard instance, we configure related parameters such as the fading factor and the number of counters to make the size of the counter lists big enough to hold every object and their frequencies so we are guaranteed to obtain correct results. In the testing instance, we vary the parameters controlling the analysis in order to obtain different TOP-K results that can be compared with results the standard instance.

Accuracy Metrics

We introduce a metric, called the approximate accuracy rate (AAR), to measure the accuracy of Tempo. AAR expresses the average amount of similarity between the results of the standard instance and the testing instance for all top-K requests over a run. The AAR is defined as follows:

$$AAR = \frac{\sum_{i=1}^{N} (n_i / K)}{N} \times 100\% \qquad \textbf{(3)}$$

where N denotes the total times we request TOP-K results and n_i denotes the number of elements which can be simultaneously observed in the TOP-K answers of the two instances for the i-th time. K, the number of results in the TOP-K, is set to 10. When we compute AAR, we ignore the ordering difference in the results from the two instances.

In our evaluation, there are several factors which affect the accuracy of the results. Some of them are derived from the approaches we use to collect database statistics or process the data. Other factors are due to the experiments we design to evaluate these approaches. We introduce experiment-related factors in the next section.

Approach-Related Factors Affecting the Accuracy of the Results

1. Limited Database Monitoring Cache

Although current DBMSs such as DB2 can support simple aggregation of general monitoring statistics inside the database server and output summarized statistics, there still exists a relatively high possibility that, over time, some monitoring information is missing due to the limited resources inside a DBMS. For example, if there are too many unique SQL statements issued during a time interval, it may not be possible to keep the summary of all the SQL statement executions in the database monitoring cache. Thus, when we use snapshot monitoring or other mechanisms to capture statistics, it is highly likely that some TOP-K items are lost.

2. Limited Space in our Sliding Windows

Our sliding windows are built in main memory so we cannot guarantee that enough space is allocated to accommodate all the system statistics we gather with snapshot monitoring. Since some elements may be dropped we must ensure that important items are maintained. Our approach to maintaining the most important information in memory is described in Chapter 3.

3. Insufficient Space in Summarization Structures

The Space-Saving algorithm, as described in Chapter 3, uses a limited number of counters to maintain the frequency of the most important elements, so not all elements and their frequencies can be maintained. We therefore cannot guarantee that the occurrences of important elements will not be ignored due to incorrect decisions by the algorithm.

4.4.2 Experimental Setup

Experiment-Related Factors Affecting the Accuracy of the Results

1. Experimental Data

The accuracy of Tempo is evaluated using a workload generated by DTW clients. Transactions are generated randomly by the clients so the runs in different experiments cannot be guaranteed to be exactly identical. Thus, we need to run the same experiments several times and then observe averages over sufficiently long periods of time and standard deviations.

2. Difference of the Testing Instance and the Standard Instance of Tempo

We compute the accuracy rate for a run based on a comparison between the standard instance and the testing instance of Tempo. The accuracy of the comparison is affected by the differences between the workloads input to the two instances caused by the workload generator.

We adopt the following practices in carrying out our experiments in order to alleviate some of the inaccuracies:

a. We always run the two instances of Tempo on the same database. Thus, the system statistics observed by the two instances are almost identical during the whole period.

b. We let the pair of instances of Tempo run concurrently with the same interval for polling system statistics, which helps to eliminate the differences due to timing of the data collection between the two instances.

c. In our experiments, there are a limited number of objects which can be observed (79 accessed tables and 100 executed SQL statements), which greatly reduces the potential inaccuracies due to the space limitations as described above. In some extreme real-life conditions, hundreds of tables may be accessed and thousands of SQL statements may be issued, we therefore recommend that the size of the buffer for a sliding window is on the order of ten times that of the counter list), This still does not affect the space efficiency of Tempo. If there is overflow in a sliding window, we throw out the oldest elements which cannot be processed in time and allow the most recent data to enter the window buffer. We note, however, that overflow does not happen during any of our experiments. If required in the future, we can consider more complex and efficient shedding strategies to filter out the least important elements.

The inaccuracy caused by insufficient space for the counter lists is one of the most important issues affecting accuracy. We conduct a set of experiments to find out the counter list size with respect to the tradeoff between accuracy and memory usage.

Other than the number of counters, we examine two additional factors, namely the fading factor and the observed data patterns in our accuracy experiments. The experiments and experimental results and analysis are presented in the following sections.

4.4.3 Experiment 1: Impact of the Number of Counters under Stable Workload Condition

Experiment Description:

In the experiment we set the fading factor of the two instances of Tempo to 0 and run the experiment ten times. In the beginning of each run, we run the workload for a warm up period to fill up the counter list so that we can observe the impact of the size of the list on the accuracy. We vary the number of counters in the testing instance in different runs to compare the results with those of the standard instance (there are always free units in the standard instance and no element exchange). The test is repeated for counter numbers varying from 10 to 40. Each time, we increase the value by 5.

The main purpose of this experiment is to evaluate the accuracy of Tempo with different counter list sizes. In addition, we wish to identify appropriate values for the number of counters which satisfy the following two conditions:

a. The level of accuracy is acceptable.

b. We can clearly see the changes in AAR as we vary the test parameters.

If possible, we hope that the shape of the curves of TOP-K SQL statements and TOP-K tables are similar and there exist values for the number of counters that are appropriate for both curves. If there are several such points, we choose the best one and fix the number of counters at that value in subsequent experiments.

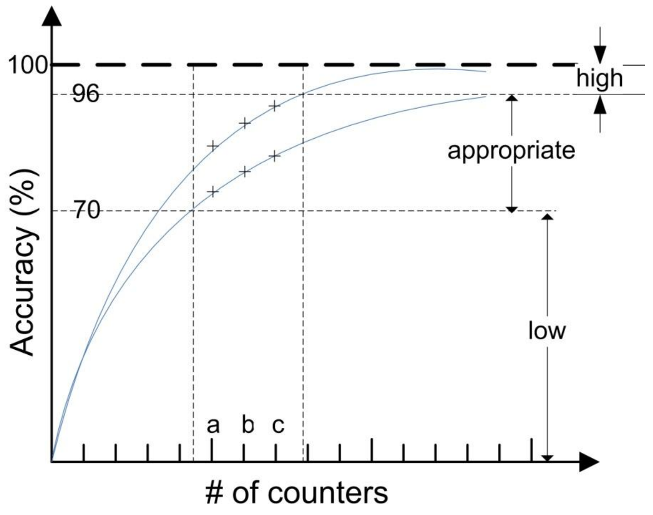


Figure 4-4 Selecting Thresholds for the Number of Counters

Figure 4-4 illustrates our method for choosing the number of counters to be used in the experiments in the following sections. We identify value ranges of the accuracy. When the value of the accuracy is in the low range, it is lower than an acceptable threshold, which we choose to be 70%. When it lies in the high range, which we choose to be above 96%, the range is too narrow to observe the accuracy trend with the change in other factors. In the example in Figure 4-4, there are only three counters a, b and c which can satisfy the above conditions. We would choose c as the best one because at this point the accuracies of the two curves reach their highest values in the appropriate range.

Experimental Results and Analysis:

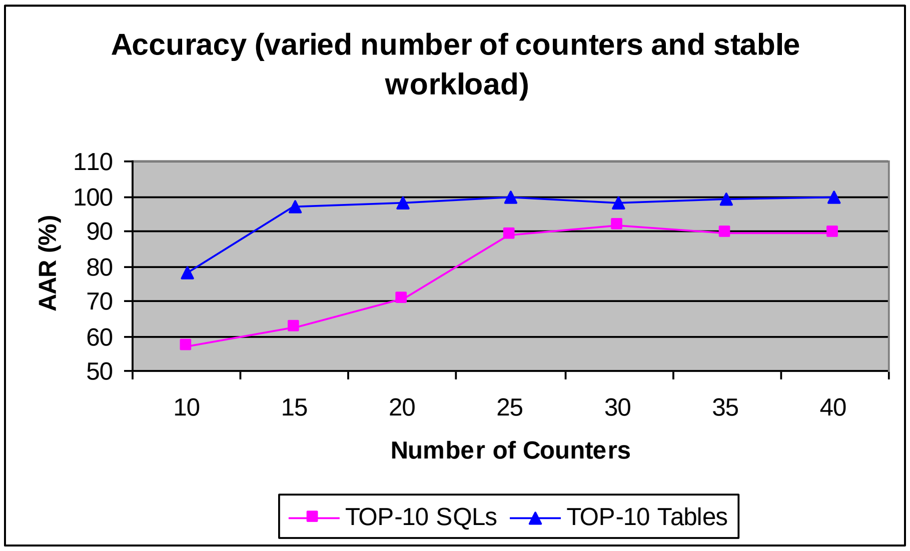


Figure 4-5 Accuracy of Executed SQL Statements and Accessed Tables when the Pattern of Underlying Workload is Smooth

Object type	Number of counters	Average AAR (%)	Standard deviation
TOP-10 SQLs	10	57.00	1.90
TOP-10 SQLs	15	62.32	2.07
TOP-10 SQLs	20	70.52	1.86
TOP-10 SQLs	25	89.09	1.01
TOP-10 SQLs	30	91.64	1.64
TOP-10 SQLs	35	89.27	0.75
TOP-10 SQLs	40	89.57	0.81
TOP-10 tables	10	78.07	1.37
TOP-10 tables	15	97.00	1.74
TOP-10 tables	20	98.23	1.94
TOP-10 tables	25	99.59	0.39
TOP-10 tables	30	98.23	1.94
TOP-10 tables	35	99.05	1.08
TOP-10 tables	40	100.00	0.00

Table 4-4 Average AARs and Related Standard Deviations when the Pattern of Underlying Workload is Smooth

Figure 4-5 shows the trend of the accuracy of Tempo with a varied number of counters under stable workload conditions and Table 4-4 shows the detailed AARs and related standard deviations for each number of counters. We notice that with the growth of the number of counters, the accuracy of Tempo steadily increases. The accuracy for TOP-10 tables reaches more than 98% with a counter list size greater than 20. The accuracy for TOP-10 SQL statements reaches a stable point a little later than the TOP-10 tables curve. Its accuracy reaches, and then is constant at about 90% with a counter list size greater than 25. Compared with the actual number of unique observed objects (79 accessed tables and 100 SQL statements), the counter size we use is relatively small, which shows that for our test cases, our approach can satisfy the necessary precision without compromising space efficiency.

However, we also note that the accuracy of TOP-10 SQL statements is always lower than the accuracy of TOP-10 tables and the former become stable later than the latter. The main reason for this is the difference in the ratio of the number of unique objects to the given size of counter list between executed SQL statements and accessed tables. When there are not enough counters to hold all unique observed elements in the counter lists, it is likely that, for the same number of counters, elements are replaced more often when processing SQL statements than when processing tables.

In addition to observing the accuracy trend of Tempo, we need to establish reasonable parameter values for our approach for the given workload. The values for the number of counters that satisfy our two conditions are 15 and 20 on the TOP-10 tables curve and 20, 25, 30 and 40 on the TOP-10 SQLs curve. The point at which the number of counters is 20 meets our additional condition that the values of the number of the counters on the two curves overlap. However, with 20 counters, the accuracy of TOP-10 tables is too high to observe any change of accuracy when varying the fading factor in the subsequent experiments. The high range as shown in Figure 4-5 for TOP-10 tables is [98.23%, 100%], which is narrow and only 1.77%. Therefore, we choose a more frequently varying workload in Experiment 2 to reduce the accuracy to an appropriate level so as to enlarge the upper range of observations.

4.4.4 Experiment 2: Impact of Variability of Workload

Experiment Description:

In a real environment, the workload patterns of production systems are not as uniform as those generated by DTW, but instead, frequently fluctuates with sporadic one-time transactions or bursts of frequent transactions interspersed with long-running queries. The amount of variability in the mix of the workload can have significant impact on the accuracy of the statistics gathered. In our approach, specifically, increased variability requires a large number of counters to maintain high accuracy levels.

In this experiment, we introduce fluctuation into the observed streaming data by alternately executing the DTW workload and the custom workload. The fluctuation rate of the workload determines how often the workload changes during a run. To be more specific, we use fluctuation rates 0, 2 3, 6 and 10 in our experiments. Fluctuation rate 0 means that there is only the DTW workload executed and a fluctuation rate of n, where n is 2, 3, 6 or 10, means the workload alternates between the 2 workloads n times.

Based on our findings in the first experiment, the number of counters is set to 20 and the fading factor is set to 0. Each run of the experiment lasts for 30 minutes, excluding initialization period and each type of frequency experiment is repeated 10 times. The two instances of Tempo send and receive a TOP-K request every 20 seconds. Based on this experiment, we choose an appropriate fluctuation rate for the workload used in the subsequent experiments.

Experimental Results and Analysis:

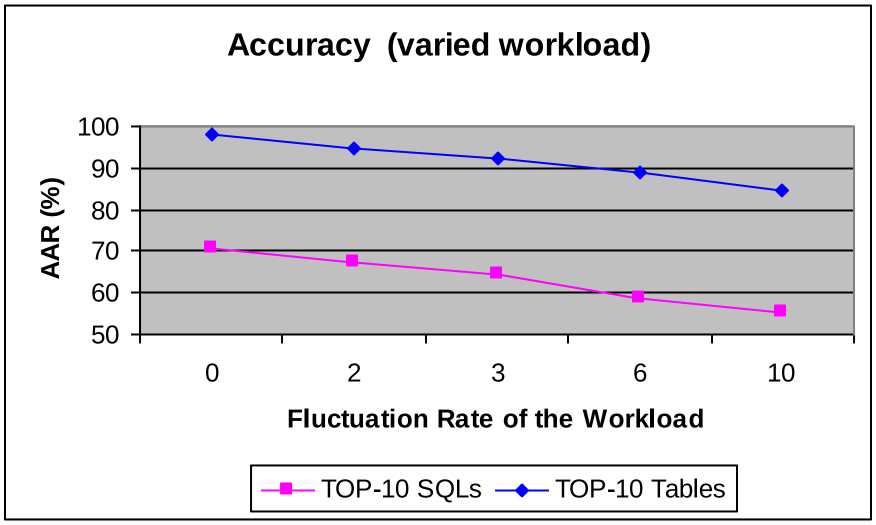


Figure 4-6 Accuracy of Executed SQL Statements and Accessed Tables with the Varied Workload Fluctuation Rate

Object type	Fluctuation Rate of the workload	Average AAR (%)	Standard deviation
TOP-10 SQLs	0	70.52	1.86
TOP-10 SQLs	2	67.40	0.87
TOP-10 SQLs	3	64.50	1.13
TOP-10 SQLs	6	58.45	1.20
TOP-10 SQLs	10	55.07	1.90
TOP-10 tables	0	98.23	1.94
TOP-10 tables	2	94.53	1.26
TOP-10 tables	3	92.27	1.01
TOP-10 tables	6	89.13	1.08
TOP-10 tables	10	84.78	1.26

Table 4-5 Accuracy Values and Related Standard Deviations of TOP-10 Executed SQL Statements and Accessed Tables on Varied Workload Fluctuation Rate

Figure 4-6 shows the accuracy of Tempo for workloads with different fluctuation rates and Table 4-5 shows the detailed AAR values and related standard deviations. We note that the accuracy of Tempo decreases with the increasing fluctuation rate of the workload, which means that the smoother the pattern of the observed workload, the higher accuracy we can obtain. A frequently fluctuating workload results in a frequent exchange of elements at the tail of the counter list and additional sorting activities in the counter list. Thus, some important elements fail to maintain their position, and instead, are moved from the head of the counter list to lower positions or even swapped out of the counter list due to a series of sorting and exchanging actions. It is very clear that frequent changes in the underlying workload have side-effects on the precision of Tempo for generating TOP-K information. As shown in Table 5-5, when the fluctuation rate of the observed workload is greater than 6, the accuracy of Tempo for TOP-10 SQLs hovers between 55% and 59%, which is far lower than when the workload is more stable.

The accuracy for TOP-K tables, on the other hand, always maintains a high degree of precision, regardless of fluctuations in the workload. The sharp contrast between TOP-10 SQL statements and TOP-10 tables indicates that sufficient space, or more counters can alleviate the impact of fluctuations. As we observed in TOP-K tables, when the ratio of the number of counters to the number of observed unique elements is relatively high, even if the pattern of the observed workload frequently changes, we still can obtain good accuracy. However, in a real system, there are likely many unique elements in an observation. Allocating a large number of counters is expensive; not only with respect to memory, but also processing since it broadens the searching and sorting range in the counter lists as well as the related reference table. In fact, space increases much faster in the reference tables than in the counter lists when we add extra counters. On the one hand, there are more unique elements in the reference tables, thereby creating a larger searching and sorting range. On the other hand, the reference tables store unique object ids and the objects themselves, which are relatively large in the case of statements. Therefore, although more counters can yield higher precision, we still need to seek other, more efficient methods, to lessen the problem. Our third experiment below evaluates the fading factor which is used to help keep the most important information in the counter lists by eliminating less important information over time, which is not at the expense of space and precision. We observe that the standard deviations in Table 4-5 remain small, even with increased variability in the workload, which means that the average values reported are still meaningful.

In addition to investigating the impact of the variability in the workload on the accuracy of our method, experiment 2 is also intended to determine a frequency of the workload fluctuation that can be used in experiment 3. Based on the results, we choose to use a frequency value of 10, because in both curves of TOP-10 SQLs and the curve of TOP-10 tables with the frequency of the workload fluctuation, the accuracy can lie in the appropriate range as shown in Figure 5-4 below or above which there is enough range space, which is easier for us to observe the change of the accuracy as the value of the fading factor increases.

4.4.5 Experiment 3: Impact of Fading Factor

Experiment Description:

In this experiment, we fix the counter list size at 20 from the appropriate points in Experiment 1 and choose the workload fluctuation rate 10 in order to maximize the variation in the workload. We vary the fading factor from 0 to 14 in the different runs to observe the impact of the fading factor on the accuracy of Tempo. We repeat each run 10 times and take the average as in previous experiments

Experimental results and analysis:

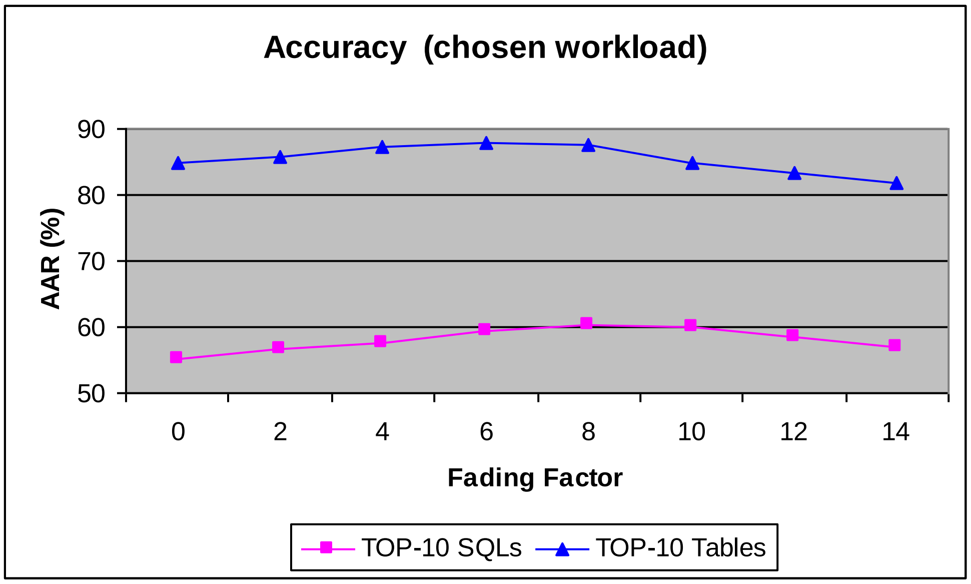


Figure 4-7 Accuracy of Executed SQL Statements and Accessed Tables when the Pattern of the Underlying Workload is Fluctuant

Object type	Fading factor	Average AAR (%)	Standard deviation
TOP-10 SQLs	0	55.07	1.9
TOP-10 SQLs	2	56.77	1.69
TOP-10 SQLs	4	57.5	1.11
TOP-10 SQLs	6	59.52	1.68
TOP-10 SQLs	8	60.23	1.13
TOP-10 SQLs	10	60.12	1.06
TOP-10 SQLs	12	58.35	1.17
TOP-10 SQLs	14	56.88	1.91
TOP-10 tables	0	84.78	1.26
TOP-10 tables	2	85.72	1.3
TOP-10 tables	4	87.15	1.12
TOP-10 tables	6	87.83	1.26
TOP-10 tables	8	87.65	0.99
TOP-10 tables	10	84.97	2.31
TOP-10 tables	12	83.2	1.93
TOP-10 tables	14	81.72	1.97

Table 4-6 AAR Values and Related Standard Deviations of TOP-10 Executed SQL Statements and Accessed Tables when the Pattern of the Underlying Workload is Fluctuant

Figure 4-7 shows the accuracy curves for TOP-10 tables and TOP-10 SQLs when we vary the fading factor and Table 4-6 shows the detailed accuracy rates and related standard deviations. We note that with increasing values of the fading factor, the accuracy of Tempo initially increases. When the fading factor value reaches 6 in the TOP-10 tables curve and 8 in the TOP-10 SQLs curve, the accuracy of Tempo reaches its peak of 87.83% and 60.23% for the TOP-10 tables and TOP-10 SQLs, respectively. Following this point, the accuracy drops. Note that when the fading factor reaches 12 in TOP-K SQL statements, the accuracy rate is lower than the original value with no fading. This behaviour indicates two things. First, an appropriate fading factor can improve the precision of Tempo for generating TOP-10 accessed tables as well as TOP-10 executed SQL statements. Second, excessively fading the frequency of the elements which are currently held by the counter lists will result in a decrease in the accuracy, even worse than in the non-fading condition. In fact, the improvement of the accuracy derived from the appropriate value of the fading factor reflects how well the value matches the pattern of the underlying workload. The best value for the fading factor given the fixed number of counters can be gained just when the largest number of insignificant elements can be removed from the current counter lists due to a decrease in their importance, where "importance" is the elements' occurrence frequencies. This allows more potential candidates to increase their importance if they have been in the lists or enter the lists if they just arrive. Another factor, the ratio of the number of counters versus the number of observed unique elements, also has an effect on the best choice of the value for the fading factor.

Chapter 5
Conclusions and Future Work

5.1 Conclusions

Precise and effective problem determination is valuable to DBAs in order to maintain a DBMS's health. However, traditional approaches for problem determination often suffer from extra system overhead which inevitably impacts the performance of the observed system. Additionally, the subsequent analysis on a large amount of static data is typically performed offline and is expensive with respect to both time and space, which can result in a long time delay before an answer is obtained. In this thesis we investigate the use of data stream processing to provide lightweight approach to problem determination in a DBMS such as DB2. Our approach combines two pivotal processes, namely online monitoring and online analysis, to eventually produce a list of TOP-K results.

In developing the framework, an existing stream analysis algorithm, called the Space-Saving algorithm, is adapted to our problem. Collected system statistics are processed in a single pass of streaming data in memory and synopses are maintained in a summary structure. We provide an improvement to the algorithm by adding a fading technique to discount stale occurrences of the observed objects, thus emphasizing the most recent observations. This makes the streaming analysis more sensitive to the frequent fluctuations of the observed database system and provides higher precision under such circumstances. Finally, after fading, the summarized synopses of the observed objects can be ordered by their minimum guaranteed frequencies and the first k elements are produced as the results.

The monitoring facilities of current DBMS systems do not naturally support this streaming style of analysis. Typically, the collected data are static and stored in persistent storage. Therefore, a stream generation process was embedded between the monitoring process and the streaming analysis. In our prototype system Tempo, we simulate multiple streams of collected DB2 statistics inside memory with a uniform format. Each stream serves a streaming analysis process for a specific type of observed object.

To investigate the practicality and efficiency of our proposed solution, we use our prototype system to perform identification and ranking of the TOP-K executed SQL statements and the TOP-K accessed tables in real-time for an IBM DB2 database system. In Tempo, the revised Space-Saving algorithm is used to summarize the collected statistics passing through the data streams. Periodically, a fading module scans every observed object in the summary-structure and discounts their frequencies to fade away obsolete information.

Experiments are presented to evaluate the overhead of our streaming approach, the impact of the approach on the observed system, and the accuracy of Tempo. The evaluation results show that the overhead incurred by Tempo is relatively small and that it does not severely impact the observed system. Compared with an existing monitoring tool, DB2TOP, Tempo can entirely eliminate the need for a large amount of temporary storage for tracing a long history of observed objects, at the expense of slightly higher memory space. Configured with enough counters, Tempo can obtain a satisfactory level of accuracy. Even if when there are a limited numbers of counters and the workload fluctuates among different sets of queries, with an appropriate fading factor, the accuracy of Tempo can be improved and still remain at an acceptable level.

5.2 Future Work

Currently, our work provides an effective solution to light-weight Top-K analysis. However, to use our work to support problem determination in autonomic DBMSs, there remain some key research issues.

First, we have seen that an adequate number of counters and an appropriate fading factor are prerequisites for obtaining a satisfactory precision. In our experiments, under different workload patterns, we attempt to vary several values for the two parameters in a specific range and observe the trends of the accuracy of Tempo. The best choices for the two parameters are determined by several factors, such as the distribution of the observed objects and their frequencies, the total number of unique observed objects, and the ratio between the number of unique observed objects and the number of traced elements in the counter list. Further research and experiments are needed to examine these issues. In addition, to apply Tempo to autonomic DBMSs, we have to make the choices dynamic. Thus, we need to investigate other solutions to dynamically predict the change of underlying observed objects and their frequencies and, based on the detected information, to adjust these parameters for the current or upcoming workload pattern.

Second, in a real environment, there are hundreds of system metrics that can be monitored, such as database CPU utilization for SQL statements, lock statistics, or buffer pool usage. How to process so many different metrics in real-time is a challenge.

Processing mixed metrics in a single data stream requires a new shedding strategy rather than our current approach of simply eliminating older elements in the circular window structure. Different types of metrics also have different statistical criteria. How to quantify them and unify them into the same data stream requires further investigation.

Third, rapidly identifying the SQL statements is another problem we have to face. In Tempo, we use a hash function to generate IDs for formalized Dynamic SQL statements. However, when using this method to process static SQL statements or dynamic SQL statements with different schema names, either the IDs are not unique, or the SQL statements with same SQL template and different parameters are considered as different objects. Chaudhuri et al. propose SQL signatures [26] based on an explanation or execution plan tree to resolve the problem in an on-line fashion. However, generating and maintaining these types of trees over a data stream could be expensive, both in terms of space and time.

References

[1] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams, *Proceedings of the 10th International Conference on Database Theory (ICDT)*, Edinburgh, United Kingdom, pages 398-412, 2005.

[2] A. Arasu and G.S. Manku. Approximate counts and quantiles over sliding windows, *Proceedings of the 2004 ACM Symposium on Principles of Database Systems*, pages 286 - 296, 2004.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems, *Proceedings of the 2002 ACM Symposium on Principles of Database Systems*, pages 1–16, June 2002.

[4] B. Babcock and C. Olston. Distributed top-k monitoring, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD 2003)*, pages 28 - 39, 2003.

[5] B. Babcock, M. Datar, and R. Motwani. Load shedding techniques for data stream systems, *Proceedings of the 2003 Workshop on Management and Processing of Data Streams*, 2003.

[6] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A stream database for network applications, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 647 - 651, New York, NY, USA, 2003.

[7] C. Giannella, J. Han, J. Pei, X. Yan, and P. S Yu. Mining frequent patterns in data streams at multiple time granularities, *Proceedings of the NSF Workshop on Next Generation Data Mining*, 2002.

[8] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. The VLDB Journal, pages 120 - 139, 2003.

[9] D. Carney, U. Centiemel, M. Cherniak, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams – A new class of data management applications, *Proceedings of the 28th International Conference on Very Large Data Bases*. Hong Kong, China, pages 215 – 226, 2002.

[10] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 356 - 357, Aug. 2002

[11] G. Cormode and S. Muthukrishnan. What's Hot and What's Not: Tracking most frequent items dynamically, *Proceedings of the 22nd ACM PODS Symposium on Principles of Database Systems*, pages 296 - 306, San diego, California, USA, 2003.

[12] H. Abdulsalam, D.B. Skillicorn, and P. Martin. Mining data-streams. In P. Poncelet, F. Masseglia, and M. Tessiere, editors, *Success and New Directions in Data Mining*, pages 302 - 324. Idea Group Inc. (IGI), October 2007.

[13] IBM. Automating problem determination: A first step toward self-healing computing systems, October 2003,
http://www.ibm.com/developerworks/autonomic/library/ac-summary/ac-prob.htm

[14] IBM. Autonomic computing: IBM's perspective on the state of information technology, 2001,
http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf.

[15] IBM. IBM DB2® 9.5 System Monitor Guide and Reference, March, 2008,

ftp://ftp.software.ibm.com/ps/products/db2/info/vr95/pdf/en_US/db2f0e951.pdf.

[16] IBM. Linux Performance and Tuning Guidelines, Chapter 1, July 2007, *http://www.redbooks.ibm.com/redpapers/pdfs/redp4285.pdf.*

[17] IBM. The db2pd tool: A new DB2 UDB utility for monitoring DB2 instances and databases, April 2005, *http://www.ibm.com/developerworks/db2/library/techarticle/dm-0504poon2/*

[18] IBM. IBM DB2® Database for Linux, UNIX, and Windows Information Center, June 2009, *https://publib.boulder.ibm.com/infocenter/db2luw/v9r5/index.jsp*

[19] IBM confidential document. DTW Tool Kit Tutorial, August 31, 2005.

[20] J. Widom and R. Motwani. Query processing, resource management, and approximation in a data stream management system, *Proceedings of the First Biennial Conference on Innovative Data Systems,* Asilomar, CA, USA, 2003.

[21] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams, *Proceedings of the 29th International Colloquium on Automata, Languages and Programming,* pages 693–703, Springer-Verlag, 2002.

[22] M. Hubel. DB2 Basics: The whys and how-tos of DB2 UDB monitoring, IDUG Solutions Journal, Aug. 2004, *http://www.ibm.com/developerworks/data/library/techarticle/dm-0408hubel/index.html#author*

[23] Oracle. Oracle® Database Performance Tuning Guide11g Release 1, July 2008, *http://download.oracle.com/docs/cd/B28359_01/server.111/b28274.pdf.*

[24] P. Domingos and G. Hulten. Catching up with the data: Research issues in mining data streams. *Proceedings of the 2001 ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD)*, Santa Barbara, California, 2001.

[25] P. Domingos and G. Hulten. Mining high-speed data streams. *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 71 – 80, 1Boston, Massachusetts, 2000.

[26] S. Chaudhuri, A. Konig and V. Narasayya. SQLCM: A continuous monitoring framework for relational database engines, *Proceedings of the 20th International Conference on Data Engineering*, Toronto, Canada, pages 473 - 485, September 2004.

[27] S. Chaudhuri and G. Weikum. Rethinking database system architecture: towards a self-tuning RISC-style database system, *Proceedings of the 26th VLDB Conference*, Sept. 2000.

[28] S. Chandrasekaran. Telegraphcq: Continuous dataflow processing for an uncertain world, 2003.

[29] S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams, *Proceedings of the 2002 ACM SIGMOD International Conference*, 2002.

[30] T. Wang, Sh. Li. DB2 problem determination using db2top utility, Dec 2008, *http://www.ibm.com/developerworks/data/library/techarticle/dm-0812wang/.*

[31] TPC Benchmark C, Standard Specification, Revision 5.10.1, February 2009, *http://www.tpc.org/tpcc/spec/tpcc_current.pdf.*

[32] Toward Autonomic Computing with DB2 Universal Database, SIGMOD Record, Vol. 31, No. 3, September 2002.

[33] Y. Zhu and D. Shasha. StatStream: Statistical monitoring of thousands of data streams in real time, *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, Hong Kong, China, pages 358 - 369, 2002.

[34] Y. Chen, G. Dong, J. Han, B.W.Wah, and J.Wang. Multi-dimensional regression analysis of time-series data streams, *Proceedings of the 2002 International Conference Very Large Data Bases (VLDB'02)*, pages 323–334, Hong Kong, China, Aug. 2002.

Appendix A DTW workload

IBM Database Transaction Workload (DTW) is a TPCC-like OLTP workload, simulating a warehouse order entry system in which a number of warehouses associated with sales districts are maintained.

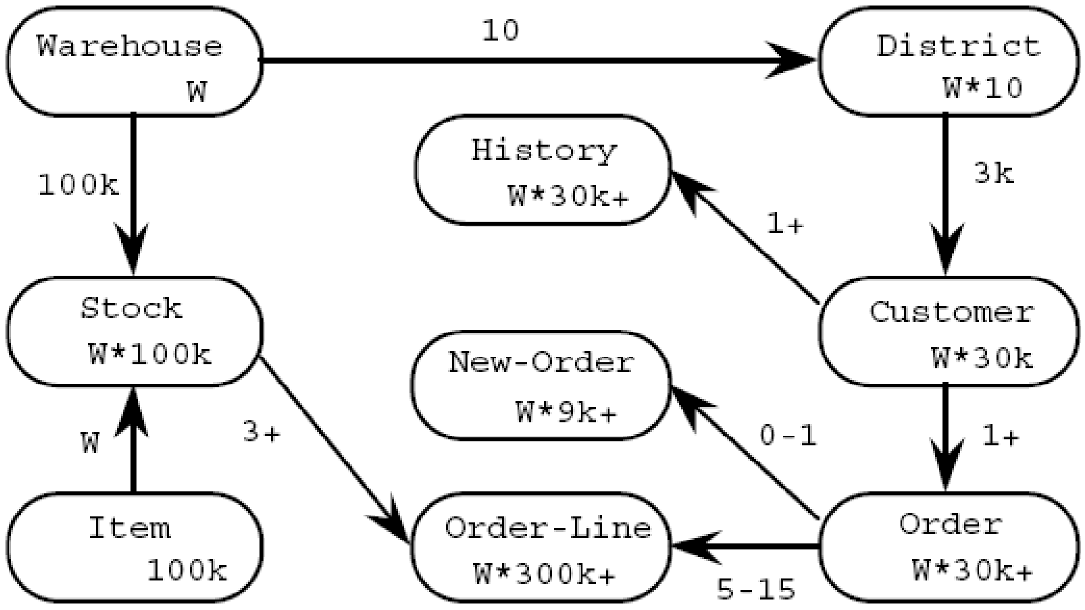


Figure A- 1 DTW Database Entity-Relationship Diagram

As shown in Figure A-1, each warehouse in the DTW supplies ten sales districts, and each district serves three thousand customers. On the other hand, each warehouse tries to maintain stock for the 100,000 items in the Company's catalog and fill orders from that stock. Five different transactions can be issued. The transactions and their percentage of the transaction mix are:

- New Order (~45%): a new order entered into the database
- Payment (~43%): a payment recorded as received from a customer

- Order Status (~5%): an inquiry as to whether an order has been processed
- Stock Level (~5%): an inquiry as to what stocked items have a low inventory
- Delivery (~5%): an item is removed from inventory and the status of the order is updated

The DTW database is scaled according to the number of warehouses. In our experiments, the number of warehouses is set to 10. The performance metric of the DTW workload is the number of transactions completed per second (TPS).

The components of DTW database are defined to consist of the following nine tables and associated indexes:

1. WAREHOUSE

Column name	Type name	Length	Nulls
W_ID	SMALLINT	2	No
W_NAME	CHARACTER	10	No
W_STREET_1	CHARACTER	20	No
W_STREET_2	CHARACTER	20	No
W_CITY	CHARACTER	20	No
W_STATE	CHARACTER	2	No
W_ZIP	CHARACTER	9	No
W_TAX	REAL	4	No
W_YTD	DOUBLE	8	No

Unique index ware_idx1 on (w_id) include (w_tax)

2. DISTRICT

Column name	Type name	Length	Nulls
D_ID	SMALLINT	2	No
D_W_ID	SMALLINT	2	No

D_NAME	CHARACTER	10	No
D_STREET_1	CHARACTER	20	No
D_STREET_2	CHARACTER	20	No
D_CITY	CHARACTER	20	No
D_STATE	CHARACTER	2	No
D_ZIP	CHARACTER	9	No
D_TAX	REAL	4	No
D_YTD	DOUBLE	8	No
D_NEXT_O_ID	INTEGER	4	No

unique index dist_idx1 on (d_id, d_w_id)

3. ITEM

Column name	Type name	Length	Nulls
------------------------------	------------------	--------	------
I_ID	INTEGER	4	No
I_IM_ID	INTEGER	4	No
I_NAME	CHARACTER	24	No
I_PRICE	INTEGER	4	No
I_DATA	VARCHAR	50	No

Unique index item_idx1 on (i_id) include(i_price,i_name,i_data)

4. STOCK

Column name	Type name	Length	Nulls
------------------------------	------------------	--------	------
S_I_ID	INTEGER	4	No
S_W_ID	SMALLINT	2	No
S_REMOTE_CNT	SMALLINT	2	No
S_QUANTITY	SMALLINT	2	No
S_ORDER_CNT	SMALLINT	2	No
S_YTD	INTEGER	4	No
S_DIST_01	CHARACTER	24	No
S_DIST_02	CHARACTER	24	No
S_DIST_03	CHARACTER	24	No

S_DIST_04	CHARACTER	24	No
S_DIST_05	CHARACTER	24	No
S_DIST_06	CHARACTER	24	No
S_DIST_07	CHARACTER	24	No
S_DIST_08	CHARACTER	24	No
S_DIST_09	CHARACTER	24	No
S_DIST_10	CHARACTER	24	No
S_DATA	VARCHAR	50	No

Unique index stock_idx1 on (s_i_id, s_w_id) include (s_quantity)

5. CUSTOMER

Column name	Type name	Length	Nulls
------------------------------	------------------	--------	------
C_ID	INTEGER	4	No
C_D_ID	SMALLINT	2	No
C_W_ID	SMALLINT	2	No
C_FIRST	VARCHAR	16	No
C_MIDDLE	CHARACTER	2	No
C_LAST	VARCHAR	16	No
C_STREET_1	VARCHAR	20	No
C_STREET_2	VARCHAR	20	No
C_CITY	VARCHAR	20	No
C_STATE	CHARACTER	2	No
C_ZIP	CHARACTER	9	No
C_PHONE	CHARACTER	16	No
C_SINCE	TIMESTAMP	10	No
C_CREDIT	CHARACTER	2	No
C_CREDIT_LIM	DOUBLE	8	No
C_DISCOUNT	REAL	4	No
C_DELIVERY_CNT	SMALLINT	2	No
C_BALANCE	DOUBLE	8	No
C_YTD_PAYMENT	DOUBLE	8	No
C_PAYMENT_CNT	SMALLINT	2	No

C_DATA1	CHARACTER	250	No
C_DATA2	CHARACTER	250	No

Unique index cust_idx1 on (c_w_id, c_d_id, c_id)

Index cust_idx2 on (c_w_id, c_d_id, c_last, c_first, c_id)

6. HISTORY

Column name	Type name	Length	Nulls
------------------------------	------------------	--------	------
H_C_ID	INTEGER	4	No
H_C_D_ID	SMALLINT	2	No
H_C_W_ID	SMALLINT	2	No
H_D_ID	SMALLINT	2	No
H_W_ID	SMALLINT	2	No
H_DATE	TIMESTAMP	10	No
H_AMOUNT	INTEGER	4	No
H_DATA	CHARACTER	24	No

7. NEW_ORDER

Column name	Type name	Length	Nulls
------------------------------	------------------	--------	------
NO_O_ID	INTEGER	4	No
NO_D_ID	SMALLINT	2	No
NO_W_ID	SMALLINT	2	No

Unique index nu_ord_idx1 on (no_w_id, no_d_id, no_o_id)

8. ORDERS

Column name	Type name	Length	Nulls
------------------------------	------------------	--------	------
O_ID	INTEGER	4	No
O_C_ID	INTEGER	4	No
O_D_ID	SMALLINT	2	No
O_W_ID	SMALLINT	2	No
O_ENTRY_D	TIMESTAMP	10	No

O_CARRIER_ID	SMALLINT	2	Yes
O_OL_CNT	SMALLINT	2	No
O_ALL_LOCAL	SMALLINT	2	No

Unique index ordr_idx1 on (o_w_id, o_d_id, o_id)

Unique index ordr_idx2 on (o_w_id, o_d_id, o_c_id, o_id desc)

9. ORDER_LINE

Column name	Type name	Length	Nulls
OL_O_ID	INTEGER	4	No
OL_D_ID	SMALLINT	2	No
OL_W_ID	SMALLINT	2	No
OL_NUMBER	SMALLINT	2	No
OL_I_ID	INTEGER	4	No
OL_SUPPLY_W_ID	SMALLINT	2	No
OL_DELIVERY_D	TIMESTAMP	10	Yes
OL_QUANTITY	SMALLINT	2	No
OL_AMOUNT	INTEGER	4	No
OL_DIST_INFO	CHARACTER	24	No

Unique index oline_idx1 on (ol_w_id, ol_d_id, ol_o_id, ol_number) include (ol_i_id,ol_amount)

The five transactions respectively consist of several Dynamic SQL statements. The SQL statements are described as the following:

- **New order transaction**

1. SELECT d_tax, d_next_o_id from district where d_id = ? AND d_w_id = ?

2. SELECT w_tax, c_discount, c_last, c_credit FROM warehouse, customer WHERE w_id = ? AND c_id = ? AND c_w_id = ? AND c_d_id = ?

3. UPDATE district SET d_next_o_id = ? WHERE d_w_id = ? AND d_id = ?

4. INSERT INTO orders VALUES (?, ?, ?, ?, ?, ?, ?, ?)

5. INSERT INTO new_order VALUES (?, ?, ?)

6. SELECT i_price, i_name, i_data FROM item WHERE i_id = ?

7. SELECT s_quantity, s_dist_01, s_dist_02, s_dist_03, s_dist_04, s_dist_05, s_dist_06, s_dist_07, s_dist_08, s_dist_09, s_dist_10, s_ytd, s_order_cnt, s_remote_cnt, s_data FROM stock WHERE s_w_id = ? AND s_i_id = ?

8. UPDATE stock SET s_quantity = ?, s_order_cnt = ?, s_ytd = ?, s_remote_cnt = ? WHERE s_w_id = ? AND s_i_id = ?

9. INSERT INTO order_line VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)

- **Delivery transaction**

10. SELECT MIN(no_o_id) FROM new_order WHERE no_w_id = ? AND no_d_id = ?

11. DELETE FROM new_order WHERE no_w_id = ? AND no_d_id = ? AND no_o_id = ?

12. UPDATE orders SET o_carrier_id = ? WHERE o_id = ? AND o_w_id = ? AND o_d_id = ?

13. SELECT SUM(ol_amount) FROM order_line WHERE ol_w_id = ? AND ol_d_id = ? AND ol_o_id = ?

14. UPDATE ORDER_LINE SET ol_delivery_d = ? WHERE ol_w_id = ? AND ol_d_id = ? AND ol_o_id = ?

15. SELECT o_c_id, o_ol_cnt FROM orders WHERE o_id = ? AND o_w_id = ? AND o_d_id = ?

16. SELECT c_balance, c_delivery_cnt FROM customer WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?

17. UPDATE customer SET c_balance = ?, c_delivery_cnt = ? WHERE c_w_id = ? AND c_d_id = ? AND c_id= ?

- **Payment transaction**

18. SELECT c_id, c_first FROM customer WHERE c_last =? AND c_w_id = ? AND c_d_id = ? ORDER BY c_first

19. SELECT c_first, c_middle, c_last, c_street_1, c_street_2, c_city, c_state, c_zip, c_phone, c_since, c_credit, c_credit_lim, c_discount, c_balance, c_ytd_payment, c_payment_cnt, c_data1, c_data2 FROM customer WHERE c_id = ? AND c_w_id = ? AND c_d_id = ?

20. UPDATE customer SET c_data1 = ?, c_data2 = ? WHERE c_id = ? AND c_w_id = ? AND c_d_id = ?

21. UPDATE customer SET c_balance = ?, c_ytd_payment = ?, c_payment_cnt = ? WHERE c_id = ? AND c_w_id = ? AND c_d_id = ?

22. SELECT w_street_1, w_street_2, w_city, w_state, w_zip, w_name, w_ytd FROM warehouse WHERE w_id = ?

23. SELECT d_street_1, d_street_2, d_city, d_state, d_zip, d_name, d_ytd FROM district WHERE d_id = ? AND d_w_id = ?

24. INSERT INTO history VALUES (?, ?, ?, ?, ?, ?, ?, ?)

- **Order status transaction**

25. SELECT c_id, c_first FROM customer WHERE c_last = ? AND c_w_id = ? AND c_d_id = ?

26. SELECT o_id, o_entry_d, o_carrier_id, o_ol_cnt FROM orders WHERE o_w_id = ? AND o_d_id = ? AND o_c_id = ?

27. SELECT ol_i_id, ol_supply_w_id, ol_quantity, ol_amount, ol_delivery_d FROM order_line WHERE ol_w_id = ? AND ol_d_id = ? AND ol_o_id = ?

28. SELECT c_first, c_middle, c_last, c_balance FROM customer WHERE c_id = ? AND c_w_id = ? AND c_d_id = ?

- **Stock level transaction**

29. SELECT d_next_o_id FROM district WHERE d_w_id = ? AND d_id = ?

30. SELECT count(distinct S_I_ID) FROM ORDER_LINE, STOCK WHERE OL_W_ID = ? AND OL_D_ID = ? AND OL_O_ID < ? AND OL_O_ID > ? AND S_I_ID = OL_I_ID AND S_W_ID = OL_W_ID AND S_QUANTITY < ?

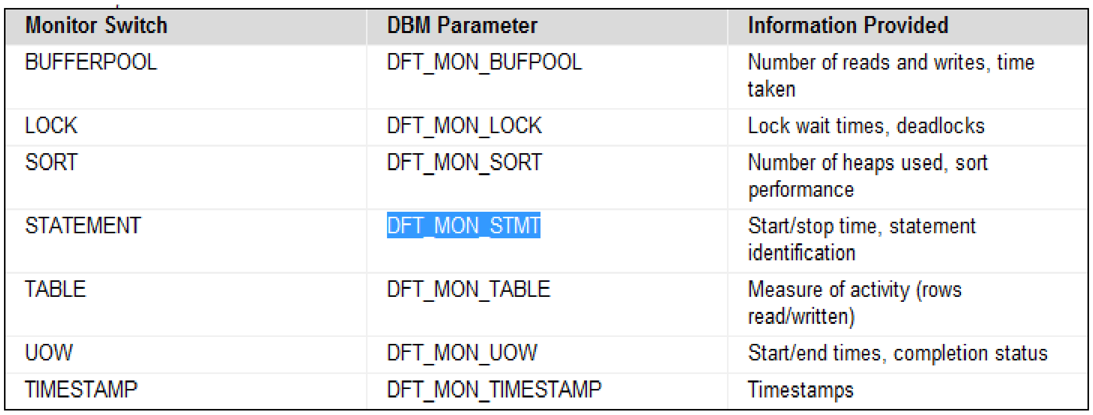
Appendix B DB2 SNAPSHOT

Snapshot monitoring provides information at a specific point in time; that is, it provides a picture of the current state of activity in the database manager for a particular object, or group of objects. Snapshot monitoring is useful in determining the current state of a database and its objects and applications. Snapshots can be obtained from the CLP (Command Line Processor), from SQL table functions, or by using the snapshot monitor APIs in a C or C++ application. In our research, we use the snapshot API.

System monitor switches

Collecting system monitor data introduces processing overhead for the database manager. For example, in order to calculate the execution time of SQL statements, the database manager must make calls to the operating system to obtain timestamps before and after the execution of every statement. These types of system calls are generally expensive. Another form of overhead incurred by the system monitor is increased memory consumption. For every monitor element tracked by the system monitor, the database manager uses its memory to store the collected data.

In order to minimize the overhead involved in maintaining monitoring information, monitor switches control the collection of potentially expensive data by the database manager. Each switch has only two settings: ON or OFF. If a monitor switch is OFF, the monitor elements under that switch's control do not collect any information. The monitoring switches in DB2 are shown in Table B-1:



Monitor Switch	DBM Parameter	Information Provided
BUFFERPOOL	DFT_MON_BUFPOOL	Number of reads and writes, time taken
LOCK	DFT_MON_LOCK	Lock wait times, deadlocks
SORT	DFT_MON_SORT	Number of heaps used, sort performance
STATEMENT	DFT_MON_STMT	Start/stop time, statement identification
TABLE	DFT_MON_TABLE	Measure of activity (rows read/written)
UOW	DFT_MON_UOW	Start/end times, completion status
TIMESTAMP	DFT_MON_TIMESTAMP	Timestamps

Table B- 1 Snapshot Monitoring Switches [15]

Self-describing monitoring data stream

The Database System Monitor stores information it collects in entities called monitor/data elements. Each monitor element stores information regarding one specific aspect of the state of the database system. In addition, monitor elements are identified by unique names and store a certain type of information. Monitor elements collect data for one or more logical data groups. A logical data group is a collection of monitor elements that gather database system monitoring information for a specific scope of database activity. Monitor elements are sorted in logical data groups based on the levels of information they provide.

The snapshot API returns the snapshot output as a self-describing data stream in the user-supplied buffer. Figure B-1 shows the structure of the data stream.

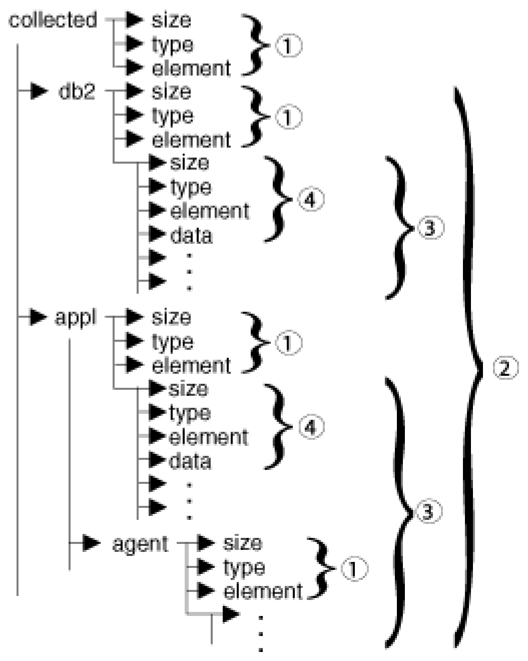


Figure B- 1 Structure of the Self-Describing Data Stream in Snapshot Cache [15]

1. Each logical data group begins with a header that indicates its size and name. This size does not include the volume of data taken up by the header itself.
2. Size in the collected header returns the total size of the snapshot.
3. The size element in other headers indicates the size of all the data in that logical data group, including any subordinate groupings.
4. Monitor element information follows its logical data group header and is also self-describing.

Appendix C DB2PD

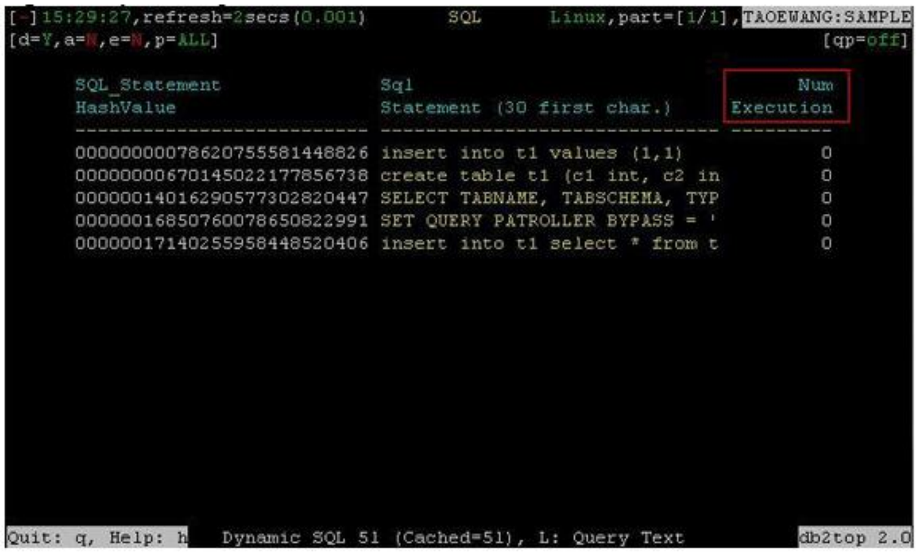
DB2 UDB V9.5 comes with a utility for collecting statistics of DB2 instances and databases called DB2PD [17]. The tool can provide diagnostic information very quickly and in a very non intrusive manner. The tool attaches directly to the DB2 shared memory set to pull out monitor and system information without having to go through the DB2 engine, thus making it very lightweight. DB2PD provides more than twenty options to display information about database transactions, tablespaces, table statistics, dynamic SQL, database configurations, and many other database details. A single DB2PD command can retrieve multiple areas of information and can route the output to files. The utility can also be invoked a specified number of times within a specified period of time, to help understand changes over time. There are two ways to invoke the DB2PD utility, namely an interactive mode or directly from an operating system command prompt. In our research, we invoke the utility at an operating system command prompt by entering the db2pd command with the command option –tcbstats (Returns information about tables and indexes) to extract the statistics information relative to accessed tables. Table C-1 shows the 22 options available for the DB2PD command and related valid scopes

db2pd option	**Description**	**Scope**
agents	Returns information about agents	Instance
applications	Returns information about applications	Database
bufferpools	Returns information about the buffer pools	Database
Catalogcache	Returns information about the catalog cache	Database
dbcfg	Returns the settings of the database configuration parameters	Database
dbmcfg	Returns the settings of the database manager configuration parameters	Instance
dynamic	Returns information the execution of dynamic SQL	Database
fcm	Returns information about the fast communication manager	Instance
help	Returns help information of the db2pd command	NA
logs	Returns information about the logs	Database
locks	Returns information about the locks	Database
mempools	Returns information about the memory pools	Both
memsets	Returns information about the memory sets	Both
osinfo	Returns information about the operating system	Instance
recovery	Returns information about recovery activity	Database
reopt	Returns information about Cached SQL statements that were reoptimized using REOPT ONCE option applications	Database
reorg	Returns information about table reorganization	Database
static	Returns information about the execution of static SQL and packages	Database
sysplex	Returns information about the list of servers associated with the database alias for all databases or for a particular database	Instance
tablespace	Returns information about the table spaces	Database
tcbstats	Returns information about tables and indexes	Database
transactions	Returns active transaction information	Database
version	Returns information about the current DB2 version and level	Instance

Table C- 1 DB2PD Options and Related Valid Scopes [17]

Appendix D DB2TOP

The snapshot monitor is one of the most commonly used tools to collect information in order to narrow down a problem. However, most entries in snapshots are cumulative values and show the condition of the system at a point in time. Manual work is needed to get a delta value for each entry from one snapshot to the next. The DB2TOP tool comes with DB2, and can be used to calculate the delta values for those snapshot entries in real time. DB2TOP can be run in two modes, interactive mode or batch mode. In interactive mode, the user enters command directly at the terminal text user interface and waits for the system to respond. Figure D-1 shows the detailed information for each cached SQL statement when invoking DB2TOP in interactive mode.



```
[-]15:29:27,refresh=2secs(0.001)        SQL      Linux,part=[1/1],TAOEWANG:SAMPLE
[d=Y,a=N,e=N,p=ALL]                                                     [qp=off]

     SQL_Statement              Sql                              Num
     HashValue                  Statement (30 first char.)     Execution
     -------------------------- ------------------------------ ---------
     00000000078620755581448826 insert into t1 values (1,1)            0
     000000000670145022177856738 create table t1 (c1 int, c2 in        0
     000000014016290577302820447 SELECT TABNAME, TABSCHEMA, TYP        0
     000000016850760078650822991 SET QUERY PATROLLER BYPASS = '        0
     000000017140255958448520406 insert into t1 select * from t        0







Quit: q, Help: h   Dynamic SQL 51 (Cached=51), L: Query Text          db2top 2.0
```

Figure D- 1 DB2TOP Dynamic SQL Screen

We can complete similar monitoring tasks in batch mode as we do in the evaluation experiments. When using the -b option, DB2TOP runs in background mode and displays information in CSV format. DB2TOP can be run in background mode in combination with reading snapshot data from a collection file using the -f <file> option. Valid sub-options for -b are:

- d : Database
- l : Sessions
- t : Tablespaces
- b : Bufferpools
- T : Tables
- D : Dynamic SQL
- s : Statements
- U : Locks
- u : Utilities
- F : Federation
- m : Memory pools

When using the -C option, DB2TOP runs in snapshot collector mode. Raw snapshot data is saved in a file named with <db2snap-<dbname>-<Machine><bits>.bin> by default (unless -f is specified). Specifying multiple sub-options for collector mode (-C) is supported. To include lock information in the collection file, we use -x along with -C. Valid sub-options for -C are:

- b : Bufferpools
- D : Dynamic SQL
- d : Database
- F : Federation
- l : Sessions
- s : Statements
- T : Tables
- t : Tablespaces
- U : Locks

The other two options of DB2TOP are used in our evaluation experiments are –o (specify the output file name) and –i (polling interval).

Appendix E Custom Workload

The custom workload is designed to provide additional observed objects in our experiments. The additional observed objects consist of 70 tables (G_i, i=1...70) and 70 SQL statements (select count(*) from G_i, i=1...70). Every table used in the custom workload shares the same table structure. The structure is shown as the following:

```
table Gi (
    i1 smallint not null,
    name char(15)
)
```

Each SQL statement is executed randomly. Based on the different random ranges of the executions of the SQL statements, they are divided into two groups. The first group of SQL statements is executed more times than the second group. The first group includes 20 SQL statements and the second group includes 50 SQL statements (the frequency of the first group of elements can match with the frequencies of the elements generated by DTW). Therefore, the data of custom workload are skewed and a majority of the observed objects, or frequent elements are more distinct. Due to the fact that each SQL statement in our custom workload only has access to one table, one execution of a SQL statement generates one access to the table. Therefore, similar to the executed SQL statements, the pattern of accessed tables is also skewed.