Hot DB'02

Course Project Reports:
Hot Topics in Database Systems, Fall 2002

edited by Anastassia Ailamaki and Stavros Harizopoulos

Deepayan Chakrabarti, Stavros Harizopoulos, Hyang-Ah Kim, Suman Nath,
Demian Nave, Stratos Papadomanolakis, Bianca Schroeder, Minglong Shao,
Vladislav Shkapenyuk, and Mengzhi Wang

August 2003
CMU-CS-03-176

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This technical report contains eight final project reports contributed by ten participants in "Hot Topics in Database Systems," a CMU advanced graduate course offered by Professor Anastassia Ailamaki in Fall 2002. The course covers advanced research issues in modern database system design through paper presentations and discussion. In Fall 2002, topics included query optimization, data stream and adaptive query processing, continuous queries, self-tuning database systems, interaction between the database software and the underlying hardware, distributed and peer-to-peer DBMS, and XML applications. The participating students studied and evaluated the cutting-edge research papers from each topic, and addressed the related issues in in-class discussions.

Inspired by the course material, the students proposed and carried out a total of eight projects. The projects included innovative research and implementation and the students worked in teams of two or by themselves. The project reports were carefully evaluated by the students using a conference program committee-style blind-review process. The resulting camera-ready papers are available in this technical report. Several of these reports (as noted in their first page) have resulted in conference submissions. The projects were presented using posters and demos during a half-day HotDB workshop that was held at Carnegie Mellon on December 10, 2002.

Keywords: DBMS, databases, performance, transaction, histogram, XML, indexing, B-trees, SQL, multiple query optimization, workload characterization, network traffic monitoring.

Contents

Profiling the Resource Usage of OLTP Database Queries [*]

Bianca Schroeder
Carnegie Mellon University
bianca@cs.cmu.edu

ABSTRACT

The goal of this project is to gain a better understanding of what mechanisms are needed to implement different levels of priority for different transactions. Such priority classes could for example be used by a company to give higher priority to prefered customers or to prioritize transactions that are of an online nature compared to transactions that just do periodic maintenace work in the background.

Towards this end we first break down the resource usage of OLTP database queries to determine the bottleneck resource that the prioritization needs to be applied to. We find that in almost all scenarios lock wait times dominate total query execution times. Moreover, a closer analysis of the lock usage patterns reveals that reordering the lock wait queues could effectively prioritize transactions.

1. INTRODUCTION

The goal of this project is to obtain a detailed profiling of the resource usage of database requests for OLTP workloads. Our motivation in doing so is different from that of traditional database profiling: previous work on profiling has been done mainly to get insights into how to design better database systems; our aim is to evaluate the potential of *query scheduling* to improve database response times. More precisely, we want to collect measurements that will help us in answering the following question:

Question: *Suppose you want to implement differentiated services in a database system, i.e. you want to be able to give different priorities to different queries. What are the mechanisms needed and how do you implement them?*

We are interested in this question not only to be able to provide different classes of service, but we also hope to be able to use such prioritization functionality to schedule transactions in order to improve overall mean response times across the entire workload. The reason we believe that there is potential for improving database response times through scheduling is that currently all resources in a database systems are allocated fairly: for example the CPU is commonly scheduled through the operating systems which will typically follow a timesharing approach and the queues in front of locks are served in a FCFS order. However, scheduling theory suggests that by favoring short jobs mean response times can be greatly improved.

The answer to the above question depends on the resource usage patterns of the database queries: We first need to determine the bottleneck resource is, i.e. the resource at which resource a typical database query spends most of it's time during processing, since this is where prioritization should be applied. We then need to understand the usage patterns for this bottleneck resource in order to design good mechanisms for giving high-priority queries preferred access to this resource. Furthermore, knowledge of the distributions of the resource requirements at the bottleneck resource is also helpful in evaluating the potential of query scheduling and to design effective scheduling policies.

The purpose of this project is to analyze the resource usage of OLTP database queries in an experimental (in contrast to simulation based) environment with respect to the above question.

In our experiments we use two different database systems: IBM DB2, a popular commercial database system, and an open source system implemented on top of the Shore storage manager [2]. Our workload is based on the TPC-C benchmark [1], a standard benchmark for OLTP systems.

We find that in the majority of cases the transaction time is dominated by the time the transaction spends waiting to acquire locks. Surprisingly, even in the case of large databases (two or more times bigger than main memory) it is lock wait times and not I/O cost that dominate transaction times. Moreover, a closer look at the lock usage patterns reveals that simply reordering the lock wait queues could effectively prioritize transactions.

2. PREVIOUS WORK

To the best of my knowledge nobody so far has attempted to analyze database workloads with respect to the applicability of prioritization and query scheduling.

3. EXPERIMENTAL SETUP

As mentioned above, our experiments are based on the TPC-C benchmark [1]. This benchmark is centered around the principal activities (transactions) of an order-entry environment. More precisely, in the TPC-C business model, a wholesale parts supplier operates out of a number of ware-

[*]This work has been submitted to ICDE 2004.

.

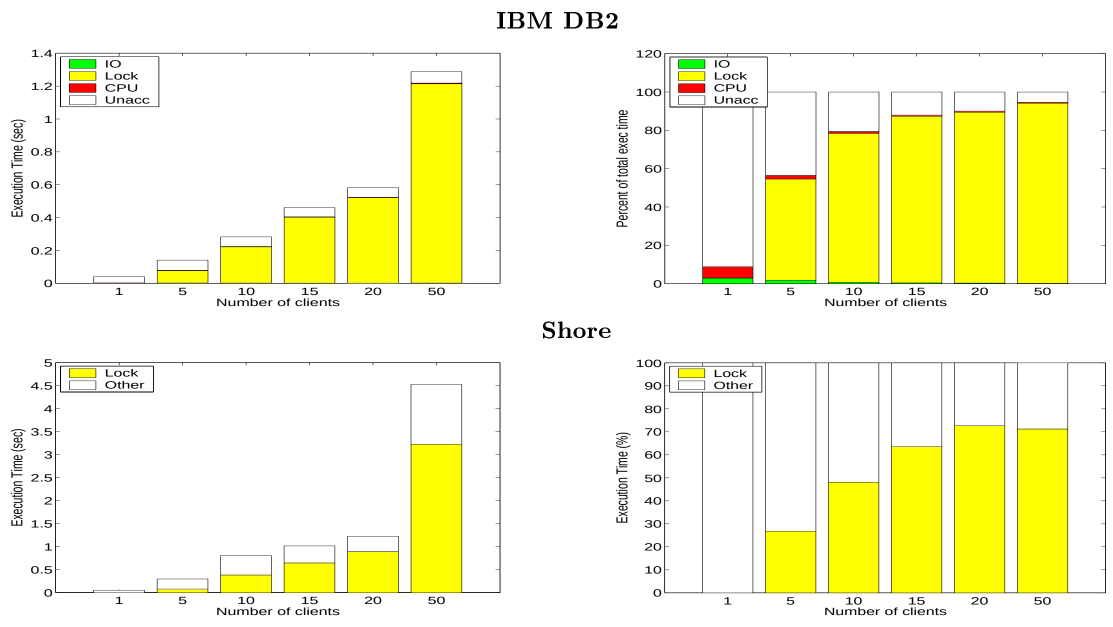


Figure 1: Level of concurrency: results for 2 warehouses and varying number of clients

houses and their associated sales districts. TPC-C simulates a complete environment where a population of terminal operators executes transactions against a database. These transactions include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses.

We run this workload on two different database systems: the commercial IBM DB2 system and a system based on Shore [2]. We break down resource usage into three different components: the time spent by a transaction on IO, CPU time and time waited to acquire locks. The time waited to acuqire locks is the total time the transaction spend during it's life time in the lock wait queue in front of a lock. In DB2 we can obtain this breakdown through the "get snapshot" functionality of DB2 and through the use of DB2 event monitors. Obtaining the same breakdown for Shore is more complicated. First, since it's not a commercial system, there's little built-in functionality for monitoring system statistics. Secondly, the system is based on user level threads (unlike DB2 which uses processes to handle transactions) which makes it difficult to obtain CPU and IO usage on a per transaction basis. We therefore limit the analysis of the Shore system to the analysis of its lock usage.

The experiments are done on an Intel Pentium 4, 2.20GHz with 1 GB of main memory running Linux 2.4.17.

4. RESULTS

In Section 4.1 we first determine the bottleneck resource. Then in Section 4.2 we analyze the resource usage at the bottleneck resource in more detail.

4.1 Determining the bottleneck resource

Since the resource usage of a DBMS depends on many system parameters, such as the workload or the size of the database, there is not one single answer to the question of which resource is the bottleneck resource. Hence, in this section we study the resource breakdown as a function of different system parameters. The parameters we take into account are level of concurrency, the size of the database and the type of the transactions.

In the TPC-C model the level of concurrency is determined by the number of terminal operators we simulated. In the following we refer to these terminal operators also as clients. The size of the database is given by the number of warehouses in the database, where one warehouse roughly corresponds to a size of 100 MB. The types of transactions are entering orders, delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses.

Section 4.1.1 studies the resource usage under different levels of concurrency and Section 4.1.2 analyzes the effect of the database size. In Section 4.1.3 we vary both the level of concurrency and the size of database, as suggested in the TPC-C guidelines. Finally, Section 4.1.4 is concerned with the different resource usage of different types of transactions.

4.1.1 Effect of level of concurrency

Figure 1 shows the execution time breakdown for experiments with 2 warehouses and number of clients ranging from 1 to 50. Figure 1(top) shows the results for the IBM DB2 system and Figure 1(bottom) shows the results for Shore. Note that 2 warehouses correspond to a database size of roughly 200 MB. Since the main memory of the server is 1 GB, this database is completely main memory resident.

The left plots in Figure 1 shows the breakdown of the execution time in absolute numbers (sec). The right plots show the contribution of the different resources as percentages.

We observe, that as expected, the total execution time grows with higher degrees of concurrency, since more clients contend for the same resources. More interestingly, we see that for both DBMS, the execution time is dominated by the lock wait times (except for very small number of clients). The contributions of CPU and IO are negligible. [1]

[1]As a side note, observe that for DB2 there is always some

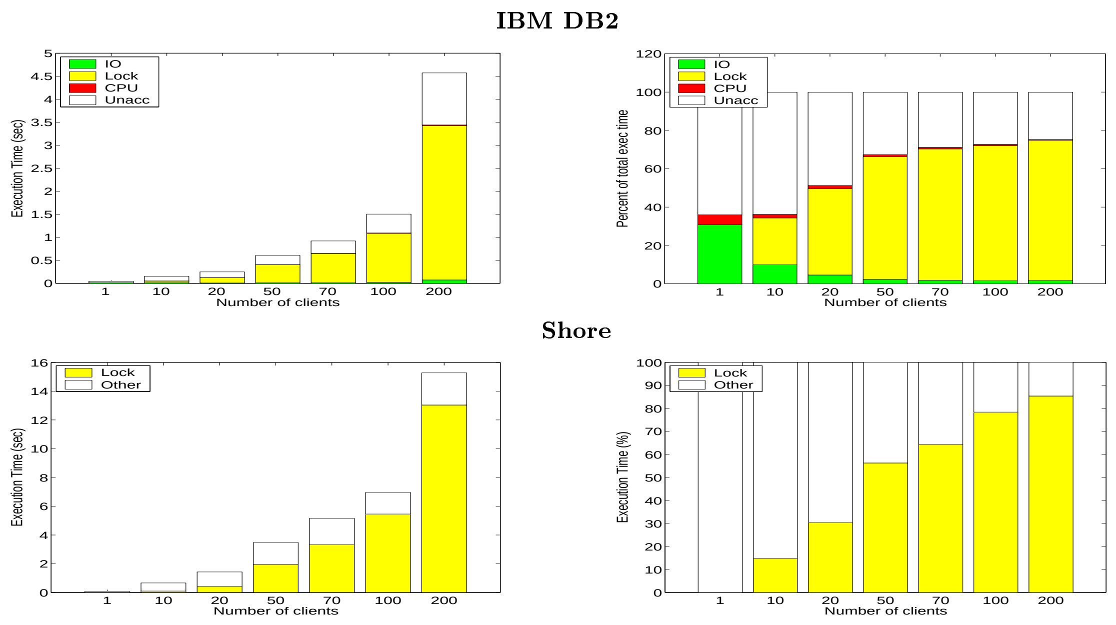


Figure 2: Level of concurrency: results for 10 warehouses and varying number of clients

Figure 2 shows the execution time breakdown for experiments with 10 warehouses, which corresponds to a database of size a little more than 1 GB. The buffer pool size we chose is 800 MB, so the database doesn't entirely fit into the bufferpool any more. As expected we see that the IO component is now larger than compared to the 2 warehouse case, but still lock wait times make up a significant fraction of the total execution time. In fact, even for a database of 30 WH, i.e. 3 times the size of main memory, the lock times still make up more than 50 percent of the total execution time for a realist number of clients (figure not shown here).

4.1.2 Effect of database size

In the experiments shown in Figure 3 we keep the number of clients constant at 10 and vary the size of the database from 1 to 20 warehouses.

It might seem surprising at first that the total execution time either hardly increases with growing database size (Shore) or even decreases (DB2). The reason lies in the significant decrease of lock wait times for larger databases: since we keep the number of clients constant, but increase the amount of data in the system the clients' accesses are distributed over a larger amount causing less contention. As a result, for both DBMS the percentage of lock wait times of the total execution times shrinks from 90 percent for 1 warehouse to only 10 percent for 20 warehouses. At the same time the contribution of IO increases for larger databases. E.g. for 20 warehouses IO time makes up more than 20 percent of the total time.

4.1.3 Results for TPC-C standard parameters

While we have previously always kept either the database size or the number of clients constant, in this section we scale the number of clients with the size of the database. The TPC-C benchmark suggests to use 10 clients per warehouse. Figure 4 shows the results for this choice of parameters for database sizes ranging from 1 warehouse to 20 warehouses.

Both database systems show a clear increase in absolute lock times with increasing database size and number of clients. For the shore system also the percentage of the lock wait time of the total execution time slightly increases, while for the DB2 system the percentage decreases. One explanation might be that the IO overhead for DB2 is bigger than that for Shore (which as a storage manager specializes in smart IO) causing the relative contribution of the locking be higher under Shore than under DB2 for big databases.

4.1.4 Effect of transaction type

Recall that the TPC-C workload consists of five different transaction types (the percentage of each type in the workload is given in parentheses):

- entering orders (*Neworder*) (44 %)
- delivering orders (*Delivery*) (4 %)
- recording payments (*Payment*) (44 %)
- checking the status of orders (*Ordstat*) (4 %)
- monitoring the level of stock at the warehouses (*Stocklev*) (4 %)

It turns out that the 5 transaction types can be classified into three categories with respect to their resource usage. Figure 5 shows one representative for each category. For all experiments in Figure 5 we scale the number of clients with the number of warehouses as suggested by TPC-C (using ten clients per warehouses)

The ordstat transactions form the first category which is completely IO bound. The reason for ordstat transactions being IO bound rather than lock bound is most likely that they are read-only operations and hence they never need to acquire a lock in exclusive mode.

portion of the time that is left unaccounted for. We haven't completely resolved this issue at this point, but have some indication that DB2 is not always correctly accounting for IO).

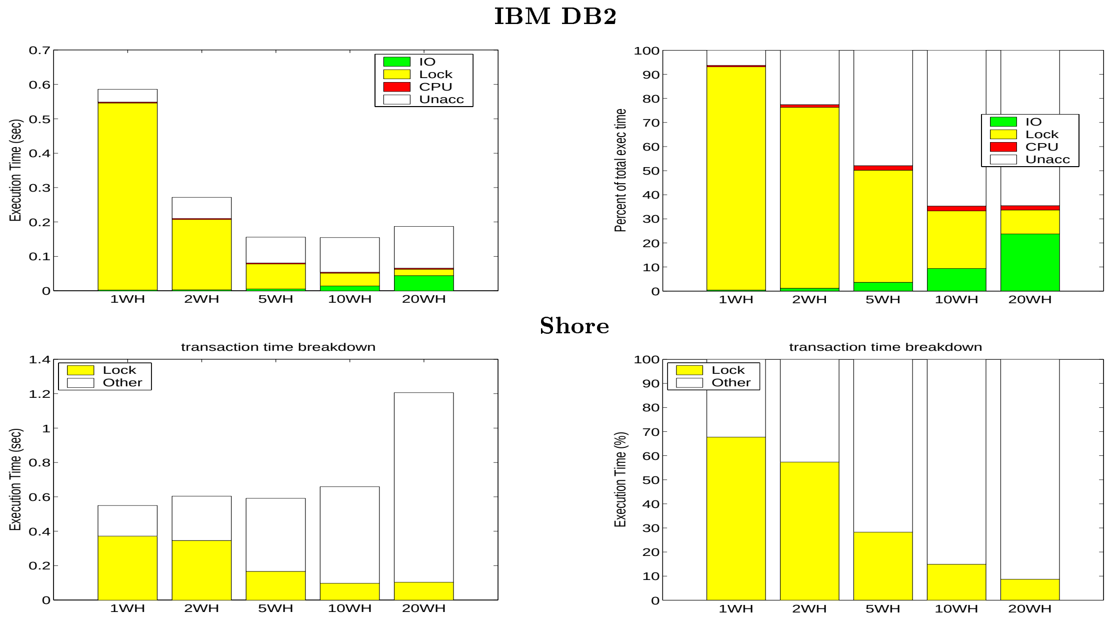


Figure 3: Size of the database: results for 10 clients and varying number of warehouses

The transaction types in the second category are completely lock bound. Payment, stocklevel and neworder fall into this category. Payment and neworder are write-heavy operations and therefore have higher chances of having to wait for locks than for example ordstat, since they need to obtain exclusive locks. Stocklev on the other hand is lock bound although it is a read-only operation like ordstat. The reason might be that stocklev reads data that is frequently updated forcing it to wait for transactions holding exclusive locks on data.

The third category is comprised of the delivery transactions, which can be either IO bound or lock bound depending on the size of the database. For in memory databases (less than 10 warehouses) the majority of the execution time is spent waiting for locks, for out of memory databases (more than 10 warehouses) IO time dominates execution times.

To summarize the results so far, we find that independently of the DBMS used (DB2 or shore) lock wait times dominate a transaction's execution times if the relation between the size of the database and the level of concurrency follows the TPC-C recommendations. This partially answers Question 1 from the introduction: based on our results so far locks are a bottleneck resource in many situations and hence it seems promising to concentrate on managing locks to implement priorities. To get more insights into the mechanisms that might be effective in implementing priorities based on locks, the next section analyzes the lock usage patterns in more detail.

4.2 Analyzing usage patterns at the bottleneck resource

We have seen that in most realistic scenarios locks are the bottleneck resource. To implement a successful prioritization and scheduling scheme, we need to understand the characteristics of the bottleneck resource usage in more detail.

As an example consider the question of whether the scheduling should be limited to letting high priority transactions jump ahead in the lock queue, or whether high priority transactions in the queue should also be allowed to abort a low priority transaction holding the lock. The first approach is in general more desirable since it is work-conserving: aborting the transaction in progress would not only waste the work that has been done on it so far, it also creates extra work since this transaction needs to be rolled back. On the other hand, just reordering the queue can only have an effect if in general the queue is long, i.e. a transactions wait most of their time for other transactions in the queue and not the transaction that's holding the lock.

In the following we give some more detailed information on the characteristics of lock wait times. All the information was obtained by instrumenting Shore.

Table 1 shows for various warehouse and client numbers the average total execution time per transaction, the average wait time per transaction, the average number of times a transaction has to wait and the average wait time for each lock that a transaction has to wait for.

WH	CL	Total/X (sec)	Wait/X (sec)	# Waits / X	Wait/Lock (sec)
1	10	0.57	.39	1.72	0.23
2	20	1.12	.82	1.81	0.45
5	50	3.04	2.31	1.78	1.29
10	100	7.68	6.13	1.85	3.30

Table 1: Lock wait time results obtained from Shore.

We see that on average each transaction waits less than 2 times for a lock (column 4), independently of number of warehouses and clients. However, each of these waits is costly: the average time for one lock wait ranges from 0.23 to 3.3 sec, depending on the setup, and is usually on the order of 40 percent of the average total execution time.

Next we try to answer the question whether the longs waits are due to waiting for one transaction holding the desired lock for a long time, or due to a long queue in front of

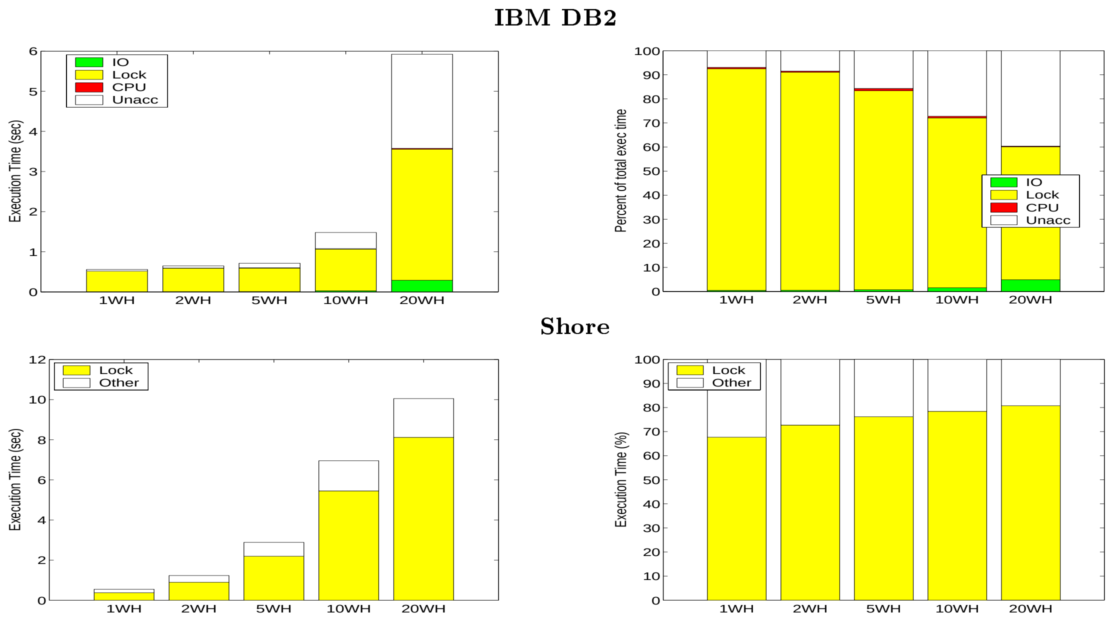


Figure 4: Scaling according to TPC-C guidelines: number of clients is always ten times the number of warehouses.

the lock. Table 2 shows the average length of the lock queue for various warehouse and client numbers. To better understand the relation between wait time and queue length we compare the queue length for as seen by long wait events to the queue length seen by short wait events. More precisely, we order the wait events by the length of their wait time and compute the average queue length seen by the top and the bottom 10 percent of these events, respectively. These are listed as LongWait-Qlength and ShortWait-Qlength in Table 2.

WH	CL	Avrg. Qlength	LongWait Qlength	ShortWait Qlength
1	10	5.78	5.81	2.58
2	20	6.04	6.60	2.41
5	50	6.23	7.05	2.84
10	100	6.50	7.11	2.93

Table 2: Relation between length of lock queue and lock wait time for Shore.

We observe that the average length of the lock queues is around 6, independent of the setup. We also observe that in the case of long waits the average queue length is much larger than in the case of short waits. This indicates that long waits are due to long queue lengths, rather than a single other transaction blocking the lock for very long.

Yet another way to look at the characteristics of the lock queue lengths and lock wait times is to consider the corresponding cumulative distribution functions (Figure 6). It is interesting to observe that both the cdf for the lock queue lengths and the cdf for the lock wait times follow a similar shape. The curves are mostly linear and indicate that most of the values are distributed uniformly over a certain interval. Also observe that in less than 15 percent of the cases the transaction who enters the wait queue is first in line (the only transaction waiting). In more than 50 percent of the cases more than 7 other transactions are ahead of a transaction entering the wait queue.

Based on the results in this section we believe that a prioritization scheme that simply reorders the lock wait queues might be an effective way of prioritizing transactions.

5. CONCLUSION

This paper studies the resource usage of OLTP transactions in order to better understand the mechanisms needed to implement differentiated service classes in DBMS. We find that for OLTP workloads with a reasonable relation between database size and level of concurrency, lock wait times dominate execution times. This is the case for both the commercial IBM DB2 system and an open source system based on Shore. A closer analysis of the lock usage patterns reveals that most of the lock wait times are spent in only a few waits which last very long due to a long lock wait queue. Our results suggest that a scheme that simply reorders the lock wait queues might effectively prioritizing transactions.

6. ACKNOWLEDGMENTS

Much of the work on IBM DB2 is joint work with David McWherter. Many thanks to Mengzhi Wang for providing the TPC-C kit for Shore.

7. REFERENCES

[1] Transaction Processing Performance Council. The tpc-c benchmark. `http://www.tpc.org/tpcc/default.asp`.
[2] University of Wisconsin. Shore - a high-performance, scalable, persistent object repository. `http://www.cs.wisc.edu/shore/`.

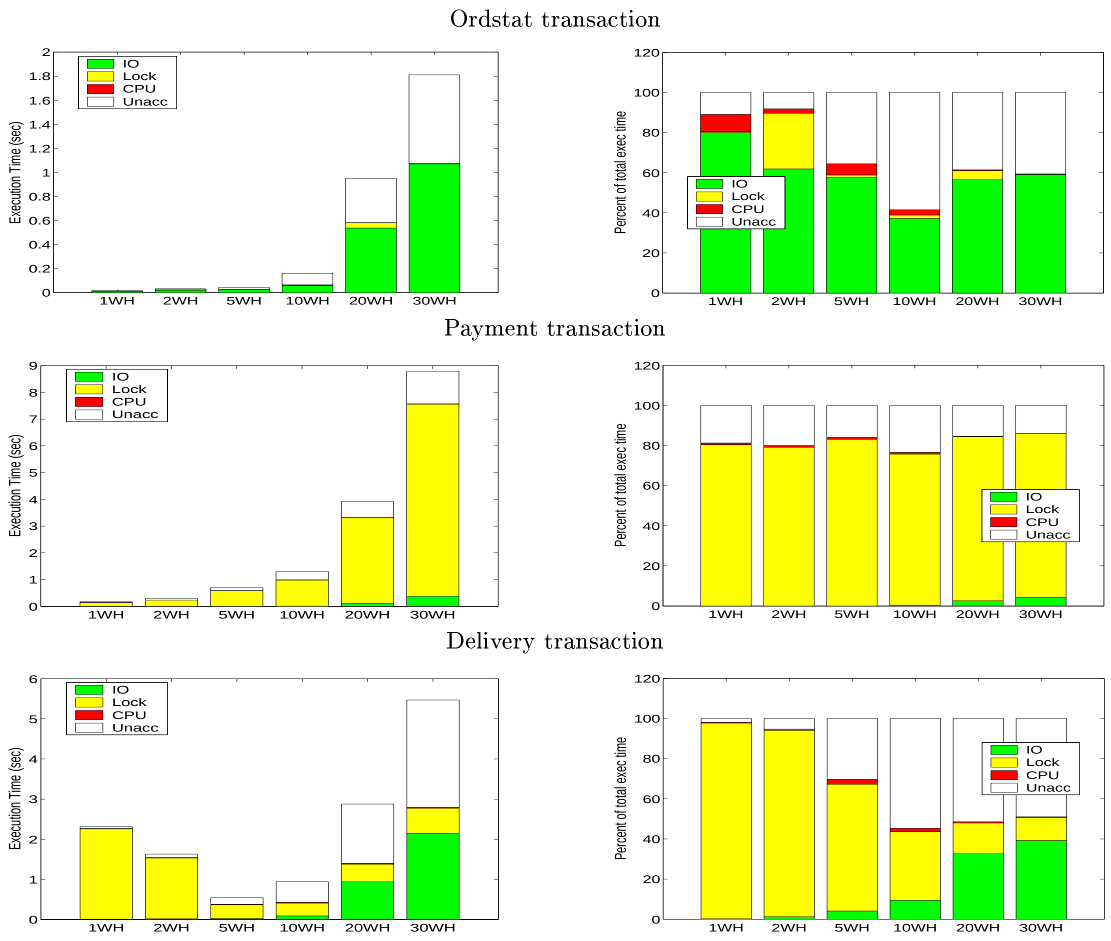


Figure 5: Transaction types: Results for ordstat, payment and delivery transactions on IBM DB2.

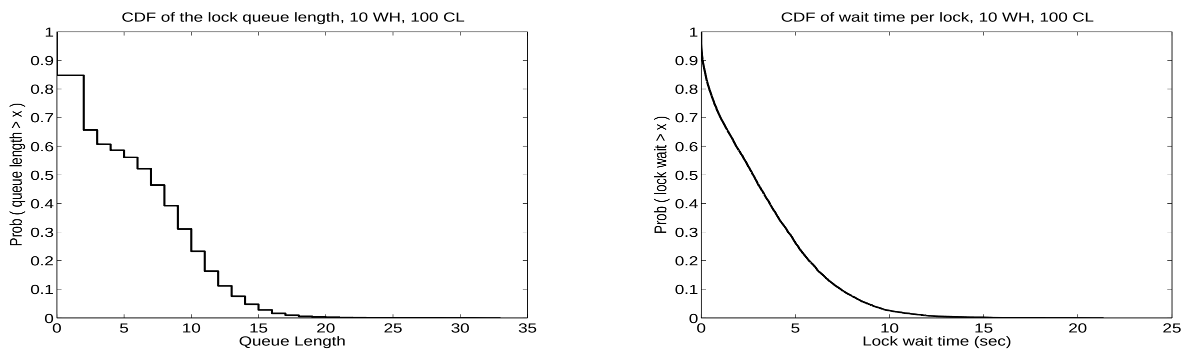


Figure 6: Cumulative distribution functions for length of lock queue (left) and time waited per lock (right).

Characterizing Change in SQL Workloads

Stratos Papadomanolakis, Minglong Shao
{stratos, shaoml}@cs.cmu.edu
Computer Science Department
Carnegie Mellon University

ABSTRACT

In this paper we analyze how changes in SQL workloads affect the performance of index selection tools, across multiple workloads and database platforms. Specifically, we use the SDSS and TPCH workloads, modifying them to model changes. The modifications target the selectivity (the number of rows retrieved by queries) and the projectivity (the number of attributes appearing in the 'select' clause of the query) for a number of queries in those workloads. We evaluate the performance of the original and the changed workloads in Microsoft's SQL Server and IBM's DB2. Also, we evaluate the operation of the index selection tools incorporated by those database systems with respect to the changed workloads. Our experimental results show that variations in the projectivity of the workload have the greatest impact on the quality of the selected indexes and on the total workload cost, while the overall performance is stable even with large changes in selectivity. We also identify cases where the index selection tools provide provably sub-optimal index configurations, both with respect to the overall improvement in workload cost and to the total required storage.

1. INTRODUCTION

Auto-tuning algorithms are a recently introduced component of database systems. Their function is to assist human administrators by automatically determining the optimal settings for a range of system parameters which are important for performance. Usually these parameters have obscure physical interpretation and are so tightly integrated with other aspects of system operation that it is virtually impossible to fine-tune them manually. To achieve optimality, auto-tuning software relies on detailed internal performance models and optimization methods. In addition, it has access to two levels of system information: The first is the current system state, for example the data organization and dataset sizes and statistics. The second is the usage access patterns, the workload that is applied to the database. Intuitively, an automatic tuning algorithm modifies critical aspects of the current system state so that the performance of a given workload is maximized. Workload driven optimization has been used to automate various tasks in modern database systems, like the selection of indexes and materialized views [1245], the declustering of tables into multiple processing nodes [6], and the management of statistics. The primary focus of this project is to define a model of *workload information* for relational databases in the context of auto-tuning algorithms and determine how their performance is affected when that information changes.

To that end, we first present a classification of query features, which we believe are relevant to the operation of automated tuning algorithms. A workload can be characterized by classifying each query according to this classification scheme. It is currently unknown how important each feature is for automated physical design. The relevance of a particular feature can be evaluated by introducing changes to it and then measuring its performance impact in two ways:

a) By measuring the performance of the changed workload, when it is executed on a database design that has been optimized for the original workload.

b) By reapplying the physical design algorithm with the new workload and comparing the overall performance between the original and the newly proposed physical design.

This methodology can help us quantify the impact of each query feature on the physical design (or any auto-tuning related) problem examined. The long term goal is to be able to analyze SQL workloads in terms of those features and predict the robustness of a given physical design or of an automated tuning algorithm with respect to those features. Another application for this classification would be the compression of massive SQL workloads (like those currently being accumulated in server-side logs) into smaller representative workloads.

For this project, we evaluate the relevance of two features, the projection list (the number of attributes in the 'select' clause) and the selectivity of expressions (which affects the total number of rows retrieved), with respect to index selection algorithms. The contributions of this work are the following:

1. We identify that variations in the size of the projection list of queries can greatly affect the performance characteristics of a given design. When the size of the projection list of a query increases to the point where no existing index contains all the projected attributes, its cost increases rapidly.

2. We identify that the projection list size also greatly affects the operation of index selection tools. The speedup provided by the recommended indexes degrades significantly when the size of the projection list is increased. The total space required by the recommended configurations is also affected.

3. We identify that query selectivity has negligible impact both on the workload cost for a given design and on the quality of the index recommendations.

4. By examining the workload cost on a query-by-query basis, we identify situations where the index selection tools recommended sub-optimal configurations.

The rest of the paper is organized as follows. Section 2 presents our classification of workload characteristics. Section 3 provides an overview of the related work. Our Experimental Methodology is described in Section 4. Section 5 discusses and analyzes the experimental results. Finally, Section 3 draws conclusions.

2. CHARACTERIZING WORKLOADS

To properly characterize the workload we need to know the features of each query. A *query feature* is a particular characteristic of an SQL statement that is relevant to automated physical design algorithms.

We identify two broad classes of features, *structural* and *statistical*. Structural query features correspond to the data objects accessed by the query and the operations applied on them. The statistical query features correspond to the statistical properties of the query's expressions (like their selectivity) that affect the selection of an execution plan for a query and its cost. The statistical features are dependent not only on the query, but also on the database contents.

The structural features loosely correspond to the clauses commonly present in SQL statements, 'select', 'from', 'where', 'group by' and 'order by'. They are presented in Table1. The statistical features, shown in Table2, indicate the statistical properties of the query expressions.

Table 1. Structural features of a query

Projection list	# attributes in the 'select' clause Attribute names in the 'select' clause
Selection list	# of attributes in the 'where' clause Attribute names in the 'where' clause Expressions in the 'where' clause and the attributes participating in them
From list	# Tables referenced (joined) Table names Nested subqueries.
Aggregation	# Attributes in aggregation clauses Attribute names in aggregation clauses
Order by	# attributes in order by clauses Attribute names in order by clauses

Table 2. Statistical features of a query

Selectivity	1. Selectivities of individual expressions 2. Total selectivity for each relation

In this work we are experimenting with the impact of the *projection list* structural feature (specifically the number of attributes, which we refer to as *projectivity*) and with the impact of varying the total selectivity for each relation.

3. RELATED WORK

Despite the number of recent publications on workload-driven tuning of relational databases there are no studies on detailed workload models. Current tools take the SQL text as input and use heuristics and the query optimizer to evaluate costs. No experimental results are provided to characterize workload change and their impact of auto-tuning algorithms. One recent study [3], however, focuses on the problem of workload compression, attempting to reduce input size and thus the running time for workload-driven applications (like index selection tools) without sacrificing the quality of their output. The concept of a *feature space* for SQL statements is introduced in [7] to reduce optimization overhead. Query features, like the number of joins and the cardinalities of the participating relations are used to support the definition of a similarity function between queries and the generation of query clusters. The objective in this case is to avoid the optimization overhead for a given query by *reusing* plans that were constructed for *similar* queries. None of the above studies considers the problem of characterizing the changes in query features and measuring their impact on workload performance or on the operation of auto-tuning algorithms.

Among various auto-tuning algorithms, index selection tools have been successfully applied in the commercial database systems. We chose two commercial index selection tools: The Index Tuning Wizard (ITW) used in Microsoft SQL Server and Index Advisor (IA) used in IBM DB2, on which we conduct the experiments. Given a workload, ITW computes the set of all indexes that might be helpful, then it goes on pruning that set by using a sample configuration evaluation tool, and at the end it combines all candidates to recommend the most promising configuration. IA first creates virtual indexes for each single query. It then invokes the query optimizer to evaluate these indexes and obtain the ratio of "benefit: cost" (benefit indicates the cost we can save by using this index, cost is the disk space used by the index) for each indexes. After that IA models the index selection problem as an application of classical knapsack problem, and adopts the algorithm of knapsack problem to recommend indexes.

4. EXPERIMENTAL METHODOLOGY

For our studies we use Sloan Digital Sky Survey (SDSS) workload and the TPC-H benchmark. The SDSS database is essentially a massive catalog of astronomical objects. The database schema is centered on the *PhotoObj* table, which stores the bulk of data for every object of interest, using a collection of 400 numerical attributes. The second largest table is the *Neighbors* table, which stores the spatial relationships between neighboring astronomical objects. The SDSS workload consists of 35 queries, all designed by astronomers to mine this huge repository of information. Most of the queries access only the *PhotoObj* table with the intent to isolate a specific subset of objects for further analysis. A small number of queries actually perform joins, primarily between the *PhotoObj* and the *Neighbors* tables, seeking particular spatial relationships. In the 65GB database available to us for experimentation, 22GB are allocated to *PhotoObj* and 5GB to *Neighbors*. For the SDSS workload, the cost of accessing those massive (and 'popular') tables dominates the total workload cost.

The TPC-H database models decision support applications. Similarly to SDSS, it is structured around a central table named *Lineitem*, which is also the largest table in the database. The second larger table is named *Orders*. We used the 10GB version of the TPC-H dataset. In this dataset approximately 7.6GB are allocated to *Lineitem and* 1.7GB are allocated to *Orders*. Similarly to the SDSS application, the cost of accessing the two largest tables dominates the cost for the entire workload.

The SDSS database is built using SQL Server 2000, and we implemented TPC-H on both SQL Server 2000 and DB2. We use the standard workloads (named *Standard* in the following sections) as a basis to generate variations, based on the classification of query features in Section 2. We model a changing workload by selecting a subset of the initial workload's queries and changing either their projectivity or the selectivity of their expressions.

4.1 Measuring /Changing Selectivity

We generated modified versions of both the SDSS and the TPC-H workloads with respect to the selectivity of their queries, while introducing minimal changes to their other structural and statistical features. For TPC-H, workloads with varying selectivity can be created by changing the values of predicates that are of the "attribute op value" type. The structure of query expressions and the actual attributes in the 'where' clause do not need to change. Also, when modifying query predicates, we focus only on those that involve the two largest tables: *Lineitem* and *Orders*. Since the operations on them alone are responsible for the largest fraction of the total workload cost, changes involving those two tables are likely to have the largest impact. By suitably modifying the 'op value' expressions in the predicates described above, we have been able to generate four different workloads.

"sel_min", where the selectivities of all predicates are changed to 10% (if they are larger than that), or keep the same if they are less.

"sel 10%", where the selectivities of all predicates involving the *Lineitem* and *Orders* tables are changed uniformly to 10%.

"sel 90%", where the selectivities of predicates have been changed uniformly to 90%

"sel_max", where the selectivities have been changed to 90% (if they were less than that), or kept the same if they are more.

For SDSS, modifying the selectivity is not straightforward. The queries that only access the *PhotoObj* table usually exhibit a very complex predicate structure. It is therefore impossible to determine a correct combination of values for all the predicates, in order to achieve a specific selectivity value. In addition to that, most of the predicates are not of the form *attribute op value,* rather involve complex operations. For those expressions, there is no way to modify any of their parameters to achieve a given selectivity value. Under those constraints, we generated 3 different workloads, with increasingly higher values for the selectivity of expressions.

"sel_OR", where we recombined arithmetical expressions using ORs instead of ANDs. This provided only a minor increase in the selectivity of the changed queries.

"sel_type", where we modified a number of predicates involving the *type* of the astronomical object that was being queried. This change provided a moderate selectivity increase.

"sel_sarg", where we changed a small number of predicates common to the majority of queries into SARGable expressions, of the form attribute *op* value. This provided the maximum increase in selectivity.

Finally, we did not select for modification any queries (either from SDSS or TPC-H) that involve joins. We did this because it is hard to define the query selectivity for joins. Also, it is hard to control the number of rows accessed from each table, as it depends on the join expressions themselves and the details of the query plan.

4.2 Measuring/Changing Projectivity

For TPC-H workloads, three workloads with different projectivities are generated, simply by adding attributes in the query's projection list.

"pro_min", where the projectivity of each modified query has been reduced to the minimum possible, by removing attributes from the 'select' clause.

"pro max 50%", where we increased the projectivity of each modified query to 50% of its maximum value.

"pro max -5", where we increased the projectivity of each query to 5 less than its maximum value.

"pro max", where we increased the projectivity of each query to its maximum value.

Similarly, for SDSS 3 different workloads were generated. *"min_proj"*, with minimal projectivity

"15_proj", where we added enough attributes in the projection list of queries, to reach the number of 15.

"30_proj", where we added enough attributes in the projection list to reach the number of 30.

5. EXPERIMENTAL RESULTS

5.1 SDSS

5.1.1 Performance of Base Design under Change

In this section we examine how the basic database design, optimized for the *Standard* workload, performs with changes in projectivity and selectivity. Figure 1 shows the overall workload cost with varying projectivity; Figure 2 shows the same with varying selectivity. For the rest of the experimental section we use the term cost to refer to the optimizer cost estimates.

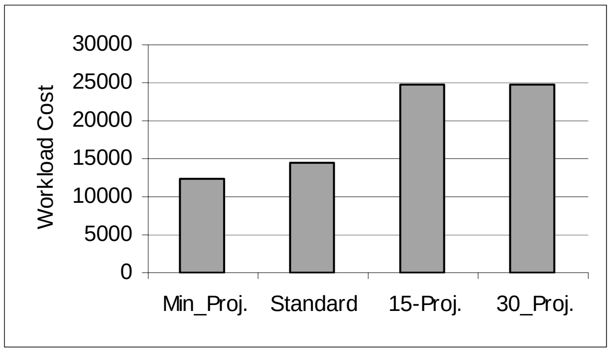


Figure 1. Total Cost with Varying Projectivity

Figure 1 shows that reducing the number of projection columns to the absolute minimum allowed, (*min_proj*) causes a 16% reduction in the overall workload cost, while increasing the projectivity to reach the target of 15 (*15_proj)* and 30 (*30_proj)* attributes causes the workload performance to degrade by 73%. The impact of selectivity variation is less severe. Figure 2 shows that increasing the selectivity for queries by a small (*sel_or*), medium (*sel_type*) or large *(sel_sarg*) amount we obtain 0.6%, 7.7% and 31% higher workload costs.

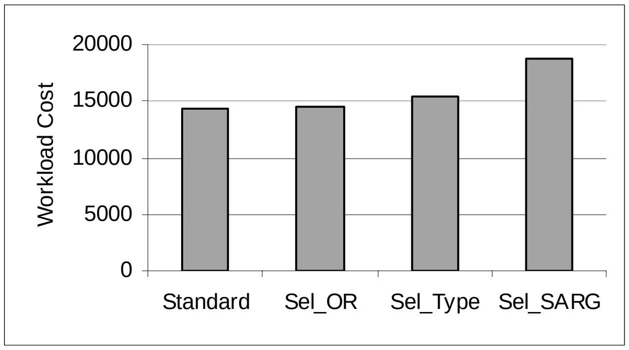


Figure 2. Total Cost for Varying Selectivity

The impact of projectivity on workload performance is significant, primarily because by increasing it, we remove the opportunity for index-only data access: No index in the current design contains all the attributes referenced in a particular query and therefore the base table must also be accessed. The effects of projectivity variations are detailed in Figure 3, where the costs for 5 SDSS queries are shown. The rest of the workload queries display similar behavior. The increased query cost for proj_15 and proj_30 workloads can be attributed to the introduction of *bookmark lookup* operations in the query plans. This operation represents accesses to pages of the base table (in addition to accessing an index) and is commonly used to retrieve the query's attributes, if they are not 'covered' by a particular index. The additional cost depends on the query selectivity: The largest the number of rows that must be retrieved, the largest the increase in the bookmark lookup overhead. Queries Q3, Q7 and Q13 of Figure 3 suffer a relatively large performance impact when the projectivity increases from the *Standard* to *15_Proj or 30_Proj,* because they have to access a large number of rows. Queries QSX1 and QSX5 on the other hand, have a negligible increase. Finally, there is no change in the workload performance when we change from 15_proj to 30_proj. This is expected, since the cost of the bookmark lookup operation does not depend on the actual number of attributes that are being accessed.

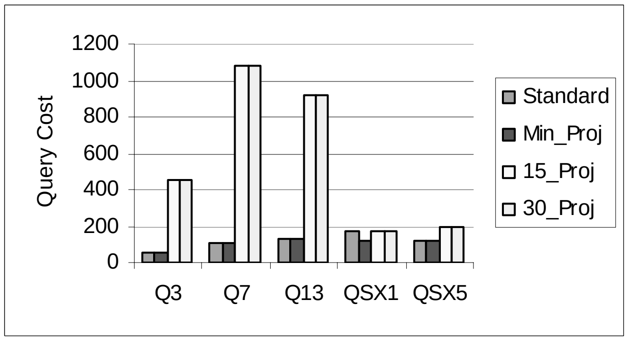


Figure 3. Query Costs with Varying Projectivity

Figure 4 shows the query costs for varying selectivity. As described previously, selectivity variations do not appear to have a significant performance impact. An exception is the final workload, *sel_sarg*, constructed to provide the largest possible number of rows for each query, which has a larger cost compared to *Standard.* The reason for the cost increase is again the existence of bookmark lookup operations.

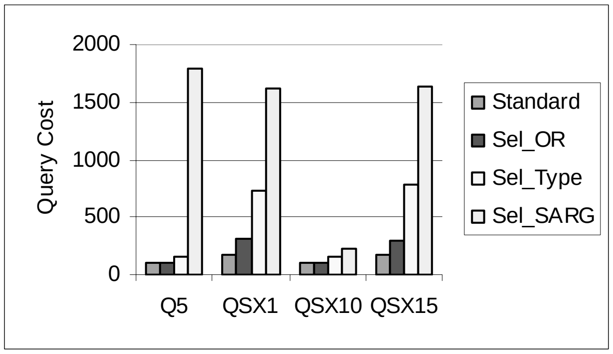


Figure 4. Query Cost with Varying Selectivity

Although increasing the selectivity of a query does not introduce new lookups, it increases the cost for the existing ones. For example, queries Q5 and QSX10 do not have a covering index in the original design. Thus they must access the base table, in order to retrieve all of their attributes. In the original design, this did not cost much because the number of extra pages that must be accessed is low. When the selectivity of a query increases, the cost of accessing the base table becomes higher. A different behavior was observed for queries QSX1 and QSX15. These queries also did not have a covering index, but the query optimizer chose to execute them using an index intersection operation.

To summarize, we found that increases in the projectivity cause large increases in the total workload cost, since queries not covered by an index need to access base tables. Changes in selectivity might cause performance degradation, but primarily for those queries that are not being covered by an index. We found that the queries not covered by an index were not affected.

5.1.1 Index Selection under Change

In this section we examine the impact of workload changes to the efficiency of the Index Wizard. The goal is to determine the effect on the tool's recommendations both with respect to performance and to the total indexing storage required.

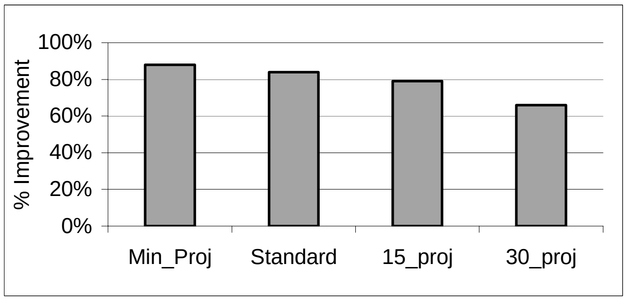


Figure 5. % Improvement with Varying Projectivity

Figures 5 and 6 show the performance improvement for the different workloads we tried. Projectivity variations appear to be important also for the Index Wizard. min_proj is improved by 4% more compared to *Standard. 15_proj* and *30_proj* are improved by 5% and 18% less. We did not observe any performance differences by varying the selectivity. *sel_or* obtains 1% less improvement compared to *standard,* while *sel_type* and *sel_SARG* obtain 1% and 2% less respectively.

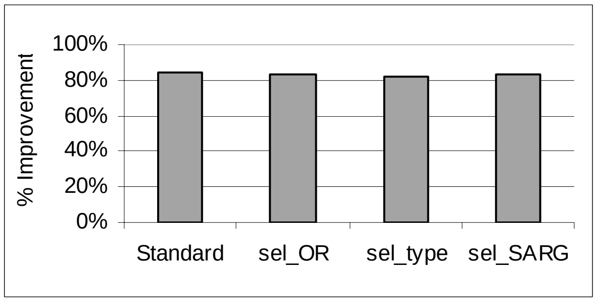


Figure 6. %Improvement with Varying Selectivity

The Index Tuning Wizard is tuned towards recommending as many 'covering' indexes as possible. By increasing the number of columns that must be included in an index the generation of 'covering' indexes is made more difficult. This explains the performance degradation in the *15_proj* and *30_proj* cases. Equivalently, if the projectivity is reduced, it becomes easier to 'cover' multiple queries and this is why the *min_proj* workload has a higher performance improvement compared to *Standard.* The feasibility of generating covering indexes essentially bounds the performance of the Index Tuning Wizard

Figure 7 shows the cost of queries for the index configurations corresponding to each workload. Queries Q3, Q4 and Q7 have significantly larger cost for the *30_proj* case compared to the *15_proj* case. This cost difference is unexpected, because the query cost should not depend on the number of attributes in the projection list. We observed that the Index Tuning Wizard chooses different designs for the *15_proj* and *30_proj* cases. In the 15_proj case, it recommends indexes that, although they do not cover queries Q3, Q4 and Q7, have very low access costs, presumably because not many base table pages are accessed by those queries anyway. In the *30_proj* case, although the use of exactly those indexes would still be feasible (the fact that the queries have a few more attributes in their projection list does not matter in terms of performance) the Index Tuning Wizard builds only very 'thin' indexes which are not very effective.

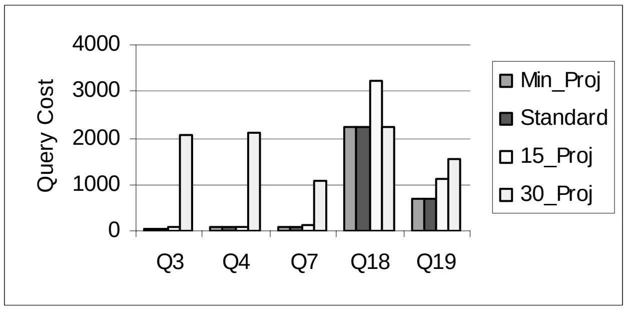


Figure 7. Optimized Query Cost with Varying Projectivity

The degradation for queries Q18 and Q19 is also unusual. We did not change the projectivity for those queries. They remain the same for all the workloads. Q18 and Q19 are joins, involving *PhotoObj* and *Neighbors*, and are sensitive to the existence of indexes on the two tables. The surprising result is that the index configuration for *proj_15* does not include any indexes on the Neighbors table, therefore penalizing all the queries that need to access that table. For the 30_proj, however, the Neighbors table is heavily indexed, and the performance for those two queries becomes identical to the *standard.*

The impact of selectivity variations in the index selection process is much less significant. Examining the queries in Figure 8, we can see that the Index Tuning Wizard did not recommend any indexes for queries Q2 and Q3 in the *sel_or* case, but these are the only two occurrences of this problem. Again, queries Q18 and Q20 have not been altered in any way. Their cost is significantly increased because for the sel_type and sel_sarg workloads, no indexes were recommended on the Neighbors table.

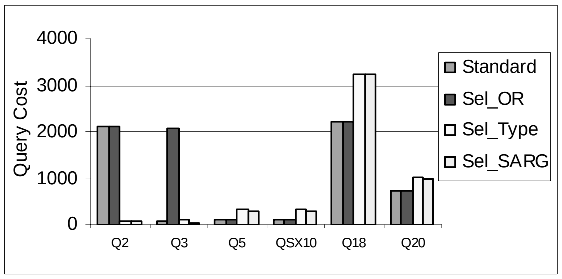


Figure 8. Optimized Query Cost with Varying Selectivity

Figures 9 and 10 study the effect of projectivities on the index storage space. We observe that the recommendation with the lowest performance (for *30_proj*), also consumes the largest space which, surprisingly, is primarily allocated to indexes on the *Neighbors* table, while the *PhotoObj* table is relatively lightly indexed. In general, we can observe an inverse correlation between workload performance improvement and storage size. In addition, no indexes are generated on the Neighbors table, for the *15_proj*, *sel_type* and *sel_sarg* workloads.

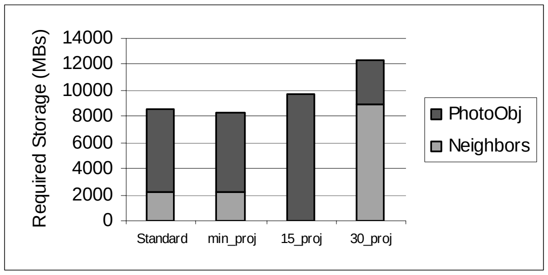


Figure 9. Index Storage for Varying Projectivity

To summarize, we found that the Index Tuning Wizard performance is bounded by its ability to cover as many queries in the workload as possible. We found that its recommendation for the 30_proj case is suboptimal. Also, in some experiments it failed to index *Neighbors* table, which could have been beneficial to performance. Finally, because the experiments involving selectivity failed to produce any significant performance variations, we can conclude that the selectivity of queries is not very important.

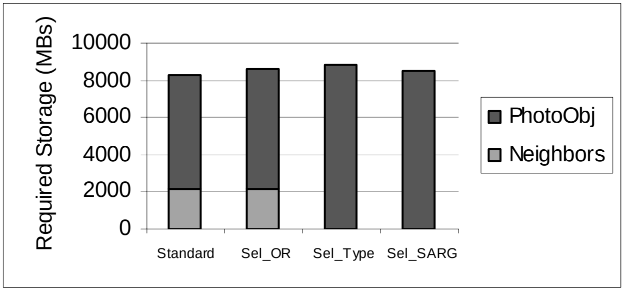


Figure 10. Index Storage for Varying Selectivity

5.2 TPC-H (SQL Server)

We repeated the same series of experiments with the 10GB TPC-H benchmark database on SQL Server. The experimental results confirm the conclusions derived from our experience with the SDSS database. In the following sections we briefly describe those results.

5.2.1 Performance of Base Design under Change

Figure 11 presents the running time estimates for workloads with modified projectivity, run on the initial database. The results confirm what was previously obtained for SDSS, that the projectivity variations have the largest impact on query execution time. We omitted the relevant graphs for selectivity, because we did not measure significant performance variations.

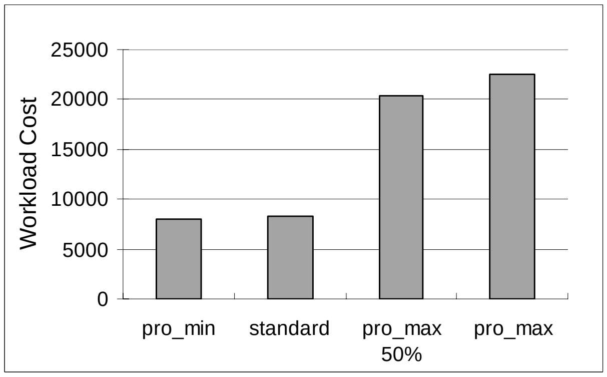


Figure 11. Total Cost with Varying Projectivity

5.2.2 Index Selection under Change

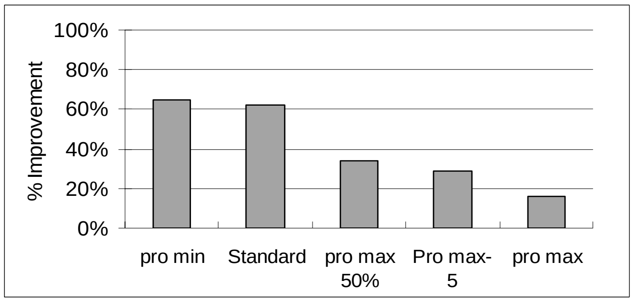


Figure 12. % Improvement with Varying Projectivity

Figure 12 presents the % improvement for the workloads with changed projectivity. There is a difference of more than 40% between the *standard* and the *pro_max* cases. So, the impact of projectivity increases in TPCH is more severe compared to SDSS. This can be related to the fact that the workload is, in general, harder to index (Figure 12 demonstrates that the baseline performance improvement is only 60%). An examination of what exactly makes index selection so hard for the *pro_max* workload is a subject for future work. We omitted the relevant graphs for selectivity, because it also did not provide any useful information.

Figure 13 compares the index storage requirements. The most interesting result is a significant reduction in storage for the *pro_max* workload. For the others, increasing projectivity also causes increases in storage.

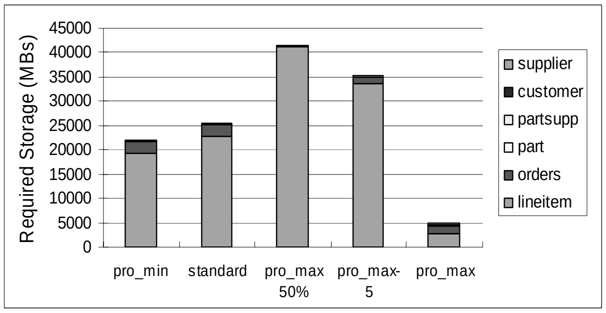


Figure 13. Index Storage for Varying Projectivity

6. CONCLUSIONS

For this project, we evaluate the relevance of two query features, projectivity and selectivity. We found that:

1. Projectivity can greatly affect the performance characteristics of a given design. Query cost increases a lot when no covering index can be found.

2. Projectivity also affects index selection. The speedup provided by the recommended indexes degrades with increasing projectivity. The total space required by the recommended configurations is also affected.

3. Query selectivity has negligible impact both on the workload cost for a given design and on the quality of the index recommendations.

4. By examining the workload cost on a query-by-query basis, we identify situations where sub-optimal indexes were recommended.

7. REFERENCES

1.Agrawal.,Chaudhuri,Narasayya,"Automated Selection of Materialized Views and Indexes for SQL Databases".VLDB 2K
2.Chaudhuri.,Narasayya."An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server". VLDB'97.
3.Chaunduri.,Gupta.,Narasayya.,"Compressing SQL Workloads" SIGMOD 2002.
4.Chauduri, Narasayya."Index Merging".ICDE '99.
5. Lohman et.al. "DB2 Advisor: An Optimizer Smart Enough to Recommend its Own Indexes". ICDE 2000.
6.Rao,Zhang,Lohman,Megiddo, "Automating Physical Database Design in a Parallel Database System". SIGMOD 2002.
7.Ghosh., Parikh., Sengar., Haritsa, "Plan Selection Based on Query Clustering", VLDB2002

Multi-query optimization opportunities in QPIPE

Stavros Harizopoulos and Vladislav Shkapenyuk

Computer Science Department, Carnegie Mellon University
{stavros, vshkap} @ cs.cmu.edu

ABSTRACT

QPIPE is a pipelined relational query execution engine for high performance DBMS that vertically traverses concurrent query pipelines, to exploit data and code locality [1]. In QPIPE, each operator is encapsulated in self-contained stages connected to each other through queues. This design allows for operator-based (rather than request thread-based) "vertical" query scheduling that (a) improves code and data affinity, and (b) reduces thread related overheads. QPIPE's staged infrastructure naturally groups accesses to common data sources thereby introducing a new opportunity for dynamic operator reuse across multiple concurrent queries.

Query subexpression commonality has been traditionally studied in *multiple query optimization* [2]. The optimizer is responsible for identifying common execution sub-paths across multiple queries during *optimization*. In this paper we explore potential techniques for exploiting common query subexpressions during a query's *lifetime*, inside QPIPE. We present experiments that demonstrate the feasibility and benefits of our approach. We also describe a demo of the proposed system.[1]

1. INTRODUCTION

Multiple query optimization [2][3] has been extensively studied over the past fifteen years. The objective is to exploit subexpression commonality across a set of concurrently executing queries and reduce execution time by reusing already fetched input tuples. As an example, consider the two queries shown in Figure 1. Query 1 performs a join on relations A and B, while Query 2 joins A, B, and C. On the lower part of the figure we show the two alternative plans the optimizer will consider for Q2. In this scenario, when both queries are presented together to the optimizer, it may be beneficial to choose the right plan for Q2 and compute the Join(A,B) only once, to be used by both queries. While this approach can introduce significant benefits, one can argue against the practicality of the algorithms, since the application is restricted to queries arriving in batches and the optimizer can face an exponentially increased search space.

Recently, there has been a proposal for *Staged Database Systems* [1]. The staged design breaks the DBMS software into multiple modules and encapsulates them into self-contained stages connected to each other through queues. Each stage exclusively owns data structures and sources, independently allocates hardware resources, and makes its own scheduling decisions. The motivation behind this staged, data-centric approach is to improve current DBMS designs by providing solutions (a) at the hardware level: it optimally exploits the underlying memory hierarchy and takes direct advantage of SMP systems, and (b) at a software engineering level: it aims at a highly flexible, extensible, easy to program, monitor, tune and evolve platform.

Central to the staged database system is *QPIPE*, its staged relational pipelined execution engine. In QPIPE, every relational operator is a stage. Query scheduling is operator-based (rather than request thread-based) thereby improving code and data affinity while reducing the context-switch overheads. Within this design lies a unique opportunity for performing multi-query optimization -style techniques that apply "on-the-fly," without delaying any query's execution and independently of the optimizer. Since queries dynamically queue up in front of the same operators, we can potentially identify ongoing common execution sub-paths and avoid duplicating operator work. This way, the burden of multiple query optimization can move from the optimizer to the run-time execution engine stages.

In this paper we explore potential techniques for exploiting common subexpressions across multiple queries inside QPIPE. We focus on QPIPE as a stand-alone relational execution engine (i.e. without the intervention of the optimizer, once it

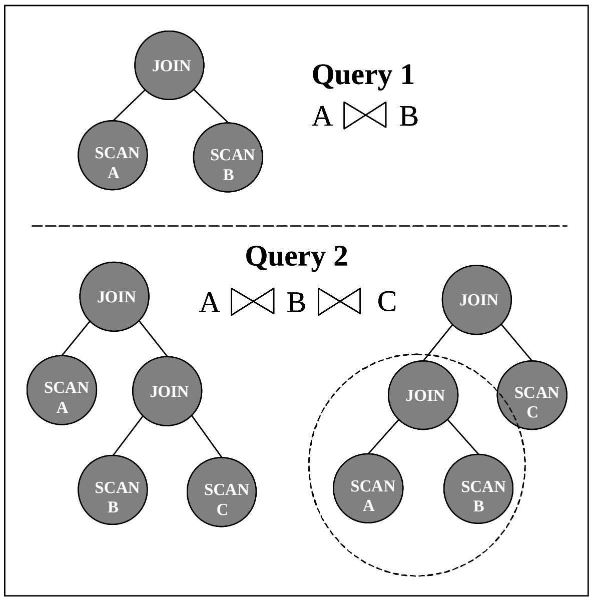
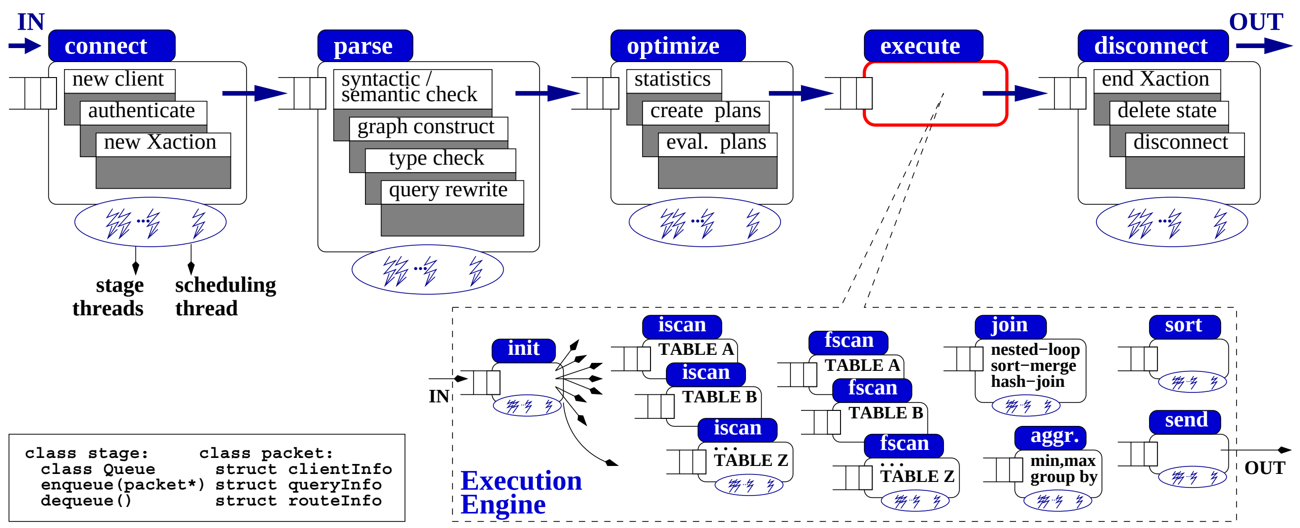


Figure 1. Two different queries that share a common subexpression

1. An extension of this work, titled "Run-time Cross-query Common Subtree Detection and Evaluation in Qpipe," was submitted to ICDE'04



```
class stage:      class packet:
  class Queue       struct clientInfo
  enqueue(packet*)  struct queryInfo
  dequeue()         struct routeInfo
```

Figure 2. The Staged Database System design. Each stage has its own queue and thread support. New queries queue up in the first stage, they are encapsulated into a "packet", and pass through the five stages shown on the top of the figure. A packet carries the query's "backpack": its state and private data. Inside the execution engine a query can issue multiple packets to increase parallelism.

produces a query plan) and describe the space of possible cases where a common subexpression can be computed only once, along with the associated trade-offs. We implement the most promising techniques and test their effectiveness in the experimentation section. We also built a graphical web-based environment for demonstrating common subexpression exploitation in QPIPE during runtime.

The rest of the paper is organized as follows. In the next section we discuss the Staged Database System design, focusing on its relational engine, QPIPE. Section 3 discusses previous efforts for pipelining in multiple query optimization. Then in Section 4 we describe new techniques for exploiting subexpression commonality in QPIPE. Section 5 carries the experimentation with the implemented algorithms and Section 6 describes the demo that comes with this paper. We conclude in Section 7.

2. QPIPE: STAGED EXECUTION ENGINE

The *Staged Database System* design [1] consists of a number of self-contained modules, each encapsulated into a *stage* (see also Figure 2). A stage is an independent server with its own queue, thread support, and resource management that communicates and interacts with the other stages through a well-defined interface. The first-class citizen in this design is the query, which enters stages according to its needs. Stages accept *packets*, each carrying a query's state and private data (the query's backpack), perform work on the packets, and may send the same or newly created packets to other stages. A stage provides two basic operations, enqueue and dequeue, and a queue for the incoming packets. The stage-specific server code is contained within dequeue. A packet represents work that the server must perform for a specific query at a given stage. It first enters the stage's queue through the enqueue operation and waits until a dequeue operation removes it. Then, once the query's current state is restored, the stage specific code is executed.

QPIPE is the relational execution engine of the staged database design. In QPIPE each relational operator is assigned to a stage (see also the dashed box in Figure 2). Although control flow amongst operators/stages still uses packets as in the top-level DBMS stages, data exchange within the execution unit exhibits significant peculiarities. Firstly, stages do not execute sequentially anymore. Secondly, multiple packets (as many as the different operators involved) are issued per each active query. Finally, control flow through packet enqueueing happens only once per query per stage, when the operators/stages are activated. This activation occurs in a bottom-up fashion with respect to the operator tree, after the init stage enqueues packets to the leaf node stages (similarly to the "push-based" model [7] that aims at avoiding early thread invocations). Dataflow takes place through the use of intermediate result buffers and page-based data exchange using a producer-consumer type of operator/stage communication.

The current implementation maps the different operators into five distinct stages (see also Figure 2): file scan (fscan) and index scan (iscan), for accessing stored data sequentially or with an index, respectively, sort, join which includes three join algorithms, and a fifth stage that includes the aggregate operators (min-max, average, etc.). Activation of operators/stages in the operator tree of a query starts from the leaves and continues in a bottom-up fashion, to further increase code locality (this is essentially a "page push" model). Whenever an operator fills a page with result tuples (i.e. a join's output or the tuples read by a scan) it checks for parent activation and then places that page in the buffer of the parent node/stage. The parent stage is responsible for consuming the input while the children keep producing more pages. A stage thread that cannot momentarily continue execution (either because the output page buffer is full or the input is empty) enqueues the current packet in the same stage's queue for later processing. QPIPE is implemented on top of PREDATOR's execution engine, which is a research prototype DBMS (www.distlab.dk/predator).

3. PREVIOUS EFFORTS

Traditional query optimization techniques rely on materialization of the common subexpressions to avoid recomputing shared results. Although these techniques can lead to significant performance gains, they do carry an additional overhead of writing and reading the results of shared subexpressions. Recent work [4] goes a step further by proposing to use pipelining to avoid unnecessary data materialization. However, pipelining is complicated by the fact that different consumers of the shared query nodes can have different consumption rates. In these scenarios a shared subexpression will be unable to produce any tuples after filling the input buffer of one of the consumers. An analytical model is proposed that allows to decide whether a pipelined schedule is valid or some of the pipelined edges will need to be materialized to avoid the operators blocking on slower consumers. Note that the decision whether to materialize or to pipeline is taken during query optimization time and is based on the assumptions about rates at which operators consume and produce the tuples.

4. PROPOSED TECHNIQUES

To illustrate how the proposed techniques are going to work, consider the two queries Q1 and Q2, and their operator trees, shown in Figure 3. Both queries share a common operator subtree and differ only in the root. If Q2 arrives in the DBMS after Q1 is inserted in the execution engine, a traditional optimizer has typically no means of identifying and taking advantage of the common subexpression. In QPIPE, Q1 will have to subsequently queue up in front of all stages that contain the Query's Execution Plan (QEP) operators. If SORT takes enough time for Q2 to queue up at the same stage, then Q2 can take advantage of all the work that Q1 has been doing.

A packet in QPIPE represents work to be done by the stage on the input tuples (pointed by the packet). The produced output tuples are stored into the parent record buffer (which, in turn, is the input buffer for the parent operator in the operator tree). Whenever two or more queries share the same subexpression (same operator subtree) then it's only necessary that one packet is active on the root of that subtree. This packet can work on behalf of all queries by copying pointers to the result tuples into the parent buffers of all queries. Then, only one query needs to maintain packets at the operators of the subtree. However, the different parent operators relying on a single operator for producing tuples can have different consuming rates. In that case we may no longer be able to maintain a single merged packet and may need to split the packets. Furthermore, file scans on the same relation but with different selectivity can still be merged but with different producing rates for the different

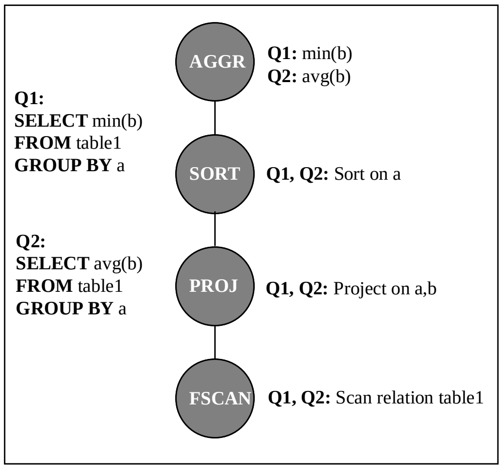


Figure 3. Two "mergeable" query plans

queries. Different producing rates may negate the effect of different parent consuming rates but they can also amplify it.

The data structure that contains the operator subtree at each node of the QEP is built during the execution initialization phase. The operator tree is traversed in prefix order and at each operator we encode all parameters, operations and attributes pertaining to a specific query. Since QPIPE first activates the leaves of the operator tree, we first describe the implementation of FSCAN and ISCAN, next SORT and AGGREGATES, and lastly, the join operators.

4.1 Scan operators

FSCAN. Two or more queries that open a file scan on the same relation can potentially merge into a single packet serving all common queries. The packet merging process in the FSCAN operator works as follows. Whenever a new scan packet is inserted into the queue, it will first examine whether there exists a scan packet that works on the same relation. If it does, then there are two cases:

1. The existing scan packet is still in its initialization state (it hasn't retrieved any tuples). In that case we can safely attach the new scan regardless of the predicate the new scan uses. For every tuple that the scan operation retrieves from a relation, it must go through the list of the merged packets to check if it matches any of the predicates and post the tuple into the respective scan's parent buffer.

2. The existing packet is already in the middle of the scan. In that case we have 3 alternatives: (a) Do nothing and enqueue a separate packet for the new scan. (b) Put the existing scan on hold until the newly arrived scan catches up with it. This can be dangerous, because in scenarios in which new scan packets constantly arrive we can infinitely delay the execution of the common part. (c) Break the new packet into two parts. The first part will scan the relation's part that is needed to catch up with the existing scan (this technique is also used in [6]). The second part will be merged with the existing scan packet. The problem with this solution is that it can destroy the tuple ordering for the

new scan. If the query plan depends on the file scan producing the tuples in certain order, this is unacceptable. In QPIPE, we can use information from the optimizer to figure out whether it is safe to break packets into two. Note that if the existing scan's parent operator didn't consume any tuples yet and the new scan has an identical predicate, we can safely merge these two scans. All the tuples that the first scan already posted into the parent buffer will need to be copied into the new scan's parent buffer of the new scan. This technique can have a significant benefit for highly selective scans by allowing us to attach to them very late in the process. Packet splitting is done by copying the scan iterator of the main packet to the ones that need to detach.

ISCAN. The most straightforward merging technique in index scans is to merge packets with exactly the same predicates, that haven't activated their parent yet. We can still exploit partially overlapping predicates, but the effectiveness heavily depends on the order the queries arrive at ISCAN and the nature of the overlapping predicates. For example, consider two queries that have the following predicates: P1:a>10, P2:a>15. If Q2 arrives while Q1 has still not crossed the value '15' in the index, then the two queries can merge. However, if the gap between the two values is large then we may end up delaying the second query. On the other hand, if Q2 is already activated when Q1 arrives at ISCAN then, if we don't care about the ordering of the output tuples, we can break Q1 index scan in to two scans and merge the second one (the same way as in FSCAN). Splitting is again done by copying the index scan iterator.

4.2 Sorting operators

SORT is a "stop-and-go" operator, in the sense that it needs to consume its whole input before starting producing any output tuples. This property effectively allows for a wide time window during which newly arrived queries at the same stage and with the same subexpression can merge with existing ones. There are 3 cases when inserting a new packet in the SORT's queue and find another packet working on the same subexpression. In all cases the new packet sets a flag to notify the children of that node that they no longer need to work.

1. The existing packet is still scanning or sorting the file. This is the easiest case since the newly arrived packet can safely attach to the working packet without any extra work.

2. The existing packet has started filling the parent buffer, but the parent operator is not yet activated. In this case we need to copy the existing packet's parent record buffer into the new packet's one. After this is done the two packets can be merged and proceed as normal.

3. The existing packet is well ahead into reading the sorted file. In this case merging the two packets is not possible but we can reuse the already sorted file. The new packet increases a usage counter on the sorted file (so that is not deleted when the current packet is done) and opens a new scan on the sorted file.

When the parents have different consuming rates we first try to double the buffers to avoid splitting. Otherwise, we split the packets by reusing the sorted file. A new scan is opened at the point the split happens and the sorted file's usage counter is increased.

4.3 Aggregating operators

New packets can always merge with existing ones as long as the parent operator hasn't consumed any tuples. Note that a simple aggregate expression (i.e. without a GROUP BY clause) produces a single tuple, so merging is always possible. Whenever there are multiple aggregate results, merging is possible while the parent operator is not yet activated. Since aggregate expressions in general can have a much lower tuple producing rate than consuming, the time window of opportunity for merging identical subexpressions can be wide.

Splitting is actually more complicated than in the previous operators. We are currently considering the following two alternatives: (a) Dump (materialize) the output of the active packet and point the detached packets to that output (this approach is also used in [4]). (b) Traverse the main packet's tree and copy all of its state into the detached packets. The trade-off in these two approaches is additional storage vs. the overhead for creating and activating all the children for the detached packets.

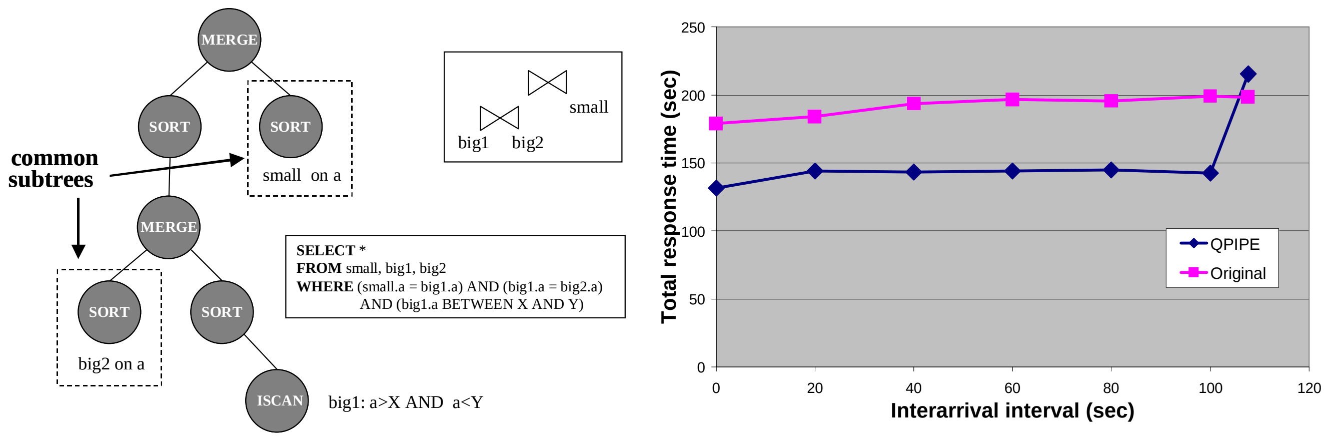
4.4 Pipelined join operators

Merging and splitting works exactly as in the aggregate operators. When joins produce output at a high rate (e.g., a nested-loop cross product) the time window for performing a merge can be very small. However, highly selective join predicates can lead to larger time windows.

5. EXPERIMENTATION

All the experiments are run against a synthetic database based on the Wisconsin benchmark specifications [5]. This suite specifies a simple schema and relatively simple select, project, join, and aggregate queries. The reason we opted for the Wisconsin benchmark (which, in today's context can be viewed as a *micro-benchmark*) and not for a TCP-C/H workload is that we wanted to have a deeper understanding of *why* our new design performs differently than the original system. With TCP-C/H it would have been difficult to map system behavior to the exact components of a specific query. The same approach has been successfully used to characterize modern commercial DBMS hardware behavior [8]. We used 100,000 200-byte tuple tables for the big tables (*big1* and *big2* in the experiments) and 10,000 tuples for the *small* table.

All experiments are run on a 4-way Pentium-III 700MHz SMP server, with 2MB of L2 cache per CPU, 4GB of shared RAM, running Linux 2.4.18. Both systems (QPIPE and PREDATOR's original execution engine) run the DBMS server on a single process implementing user-level threads. This way we effectively reduce the SMP system into a uniprocessor one. The reason we wanted to use that specific machine was to offload the processor running the DBMS server from OS-related tasks (since those will be naturally scheduled on the free processors). All numbers reported, unless otherwise noted, refer to the time a query spends inside the execution engine. Whenever



```
SELECT *
FROM small, big1, big2
WHERE (small.a = big1.a) AND (big1.a = big2.a)
        AND (big1.a BETWEEN X AND Y)
```

Figure 4. SORT reusing in 2 join queries

output tuples are produced, we discard those to avoid invoking client-server communication, since the query might be still active inside the execution engine. Before taking any measurements we first enable the results for verification purposes.

5.1 Implemented techniques

In all of the experiments the following techniques were implemented and tested: merging SCAN and ISCAN packets in their initialization phase, merging and splitting SORT packets at all phases, merging AGGREGATE and JOIN packets at all phases.

5.2 Merging at SORT

We sent two similar 3-way join queries to both our system and the original execution engine. The left part of Figure 4 shows the SQL statement and the execution plan for those queries. We used the integer attributes from the Wisconsin Benchmark for all joining and group-by attributes. The only difference in the two queries is the range of index scanning for table *big2* (i.e., the values of X and Y differ in the two queries). While both joins in the plan are *not* common across the two queries (because of the different index scan predicates) the actual join algorithm which is sort-merge in this case exhibits commonality in the sort operations. Both queries need to sort tables *big2* and *small* on attribute *a*. As long as the temporary sorted files are used (when created, sorted, or read), QPIPE can avoid duplicating the ongoing work for any newly arrived queries.

The graph in the right part of Figure 4 shows the total elapsed time from the moment the first query arrived until the system is idle again. We vary the interarrival time for the two queries from 0 secs (i.e.,the two queries arrive together) up to the time it takes the first query to finish when alone in the system (i.e., the second query arrives immediately after the first one finishes execution). The graph shows that QPIPE is able to perform packet merging at the SORT stage, thereby exploiting commonality for most of a query's lifetime (that's why the line for QPIPE remains flat most of the time) and provide up to 25% reduction in the total elapsed time. Note that both systems perform better when the queries arrive close to each other, since the system can overlap some of the I/Os between the 2 queries. Also, note that when both queries execute with no overlap between them (right most data points) QPIPE results into a slightly higher total elapsed time because of the stage queue overhead (this overhead actually pays off when there are multiple queries inside QPIPE).

5.3 Merging at AGGREGATES

In Figure 5, we repeat the same experiment with two different queries. These are aggregate queries that need to use the sort stage to satisfy the GROUP BY clause. Again, QPIPE is able to reuse the already sorted file for the second query.

5.4 Merging at JOIN

Figure 6 shows the plan for another pair of queries. This time, we let X1, Y1, X2, and Y2 be the same for both queries. In this case, QPIPE is able to merge the packets from the two different queries during the *merge* phase of the sort-merge join. Again QPIPE is able to produce up to 40% lower total elapsed time.

5.5 Throughput experiment

As a last experiment we created a random mix of queries from the Wisconsin Benchmark and used 2 clients that continuously picked a query from that mix and sent it to both systems. We measured the throughput for both systems. QPIPE was able to sustain 0.00262 queries/sec while the original system only 0.00201 queries/sec. These benefits came from merging packets at different operators during the experiment.

6. DEMO

We designed a demo for graphically displaying common subtree exploitation in QPIPE. We added code to QPIPE to periodically dump the state of each stage on a file. A cgi-bin script reads this information and continuously updates an html page containing information of the system's current state (queue utilization, query status and common subexpression exploitation). We opted for plain html and not Java, for simplicity. In the future we plan to use Java and provide smoother graphical representations and also introduce the ability to configure QPIPE and workload parameters.

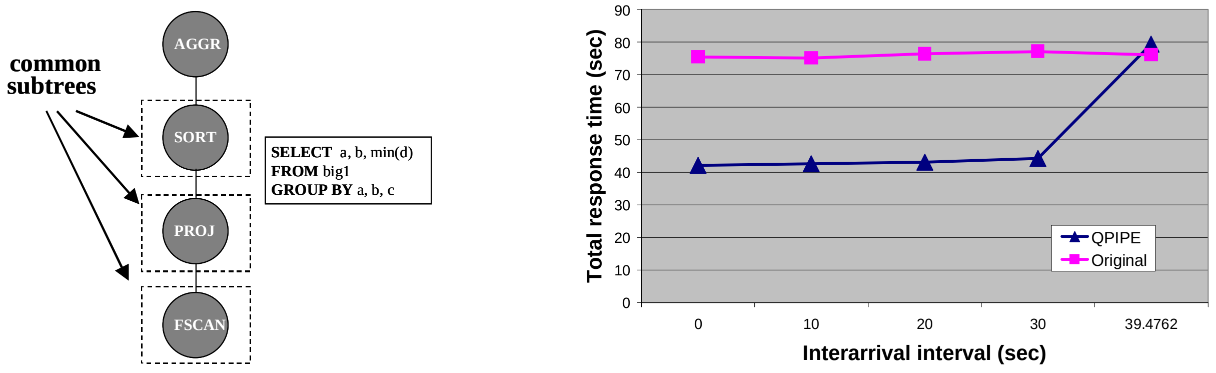


Figure 5. SORT reusing in 2 aggregate queries

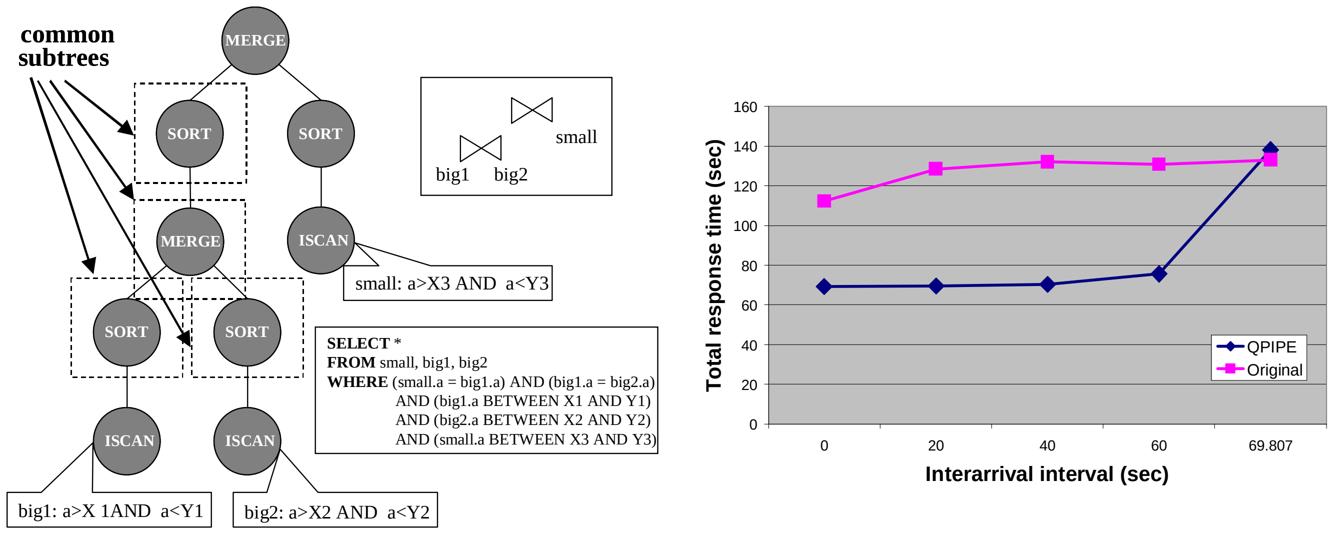


Figure 6. JOIN and SORT reusing in 2 join queries

7. CONCLUSIONS

Query subexpression commonality has been traditionally studied in *multiple query optimization* [2]. The optimizer is responsible for identifying common execution sub-paths across multiple queries during their optimization phase. In this paper we explored all the potential techniques for exploiting common query subexpressions during a query's lifetime, inside QPIPE. Similarly to [4], our approach relies heavily on pipelining of the results of common subexpressions. However, our techniques apply dynamically at run-time without needing to know in advance the consumption and producing rates of different operators in given queries. Furthermore, we avoid intermediate result materialization (in the case of a slow consumer) by either doubling the record buffers or performing packet splits. We presented experiments that demonstrate the feasibility and benefits of our approach.

8. REFERENCES

[1] S. Harizopoulos and A. Ailamaki. "A Case for Staged Database Systems." To appear, CIDR'03.

[2] T. K. Sellis. "Multiple Query Optimization." ACM Transactions on Database Systems, 13(1):23-52, March 1988.

[3] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. "Efficient and Extensible Algorithms for Multi Query Optimization." In Proc. SIGMOD, 2000.

[4] N. N. Dalvi, S. K. Sanghai, P. Roy, and S. Sudarshan. " Pipelining in Multi-Query Optimization." In Proc. PODS, 2001.

[5] D. DeWitt. "The Wisconsin Benchmark: Past, Present, and Future," in The Benchmark Handbook, J. Gray, ed., Morgan Kaufmann Publishers, San Mateo, CA (1991). `http://www.benchmarkresources.com/handbook/4.html`

[6] C. Cook. "Database Architecture: The Storage Engine". Miscrosoft SQL Server 2000 Technical Article, July 2001. Available online at: http://msdn.microsoft.com/library/

[7] G. Graefe. "Iterators, Schedulers, and Distributed-memory Parallelism". Software-practice and experience, Vol. 26 (4), 427-452, April 1996.

[8] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. "DBMSs on a modern processor: Where does time go?" In Proc. VLDB, 1999.

Entropy-Based Histogram Construction

Mengzhi Wang
Computer Science Department
Carnegie Mellon University
(mzwang@cs.cmu.edu)

ABSTRACT
Histograms give accurate selectivity estimation for range predicates. This project proposes to use entropy plot for measuring the accuracy of the histograms and introduces a top-down construction algorithm based on the measurement. The algorithm works for both one-dimensional and multi-dimensional histograms. The evaluation on synthetic workloads demonstrates that the algorithm produces histograms of high accuracy.

1. INTRODUCTION
Query optimizers in database systems take advantage of histograms for selectivity estimation of predicates. Accurate histograms help to produce better query plans. Histograms usually group tuples into buckets by the tuple values and store the summary information of the buckets. The histogram construction algorithm should focus on finding the combination of buckets of high uniformity to minimize the overall selectivity estimation error because the estimation assumes uniform tuple distribution within histogram buckets. However, existing histogram construction algorithms usually rely on heuristic rules rather than an accurate measurement for the quality of histograms.

This project proposes the entropy plot for measuring the accuracy of the histograms. The entropy plot tests the uniformity of the tuple distribution within the histogram buckets by calculating entropy value at different granularities. The measurement is able to guide the top-down histogram construction algorithm to refine the buckets of high irregular tuple distribution. The evaluation on synthetic workloads suggests that the entropy plot is effective in measuring the accuracy of the histograms and the algorithm is able to generate histograms of high quality.

The rest of the report is organized as follows. Section 2 lists some representative work on the area. Section 3 introduces the entropy plot as the measurement for histogram accuracy. Section 4 sketches the entropy-based top-down histogram construction algorithm for both one and multi-dimensional data. Section 5 compares the accuracy of the algorithm to other existing ones. Section 6 concludes the report.

2. RELATED WORK
One-dimensional histograms give selectivity estimation for predicates on a single attribute. Poosala et. al. provided a very nice taxonomy on one-dimensional histograms[7].

Multi-dimensional histograms are used for selectivity estimation of predicates on multiple columns. Techniques for constructing multi-dimensional histograms range from early equi-depth histograms [6] to recent ones including MHIST and SVD[8], Wavelets[5], Self-Tuning histograms[1, 2], and Baysian network[3, 4]. Most of these algorithms build the histograms by recursively splitting the buckets. However, the splitting is usually based on heuristic rules rather than the quality of the histograms.

This project proposes to use the entropy plot as the measurement for histogram accuracy. The histogram construction algorithm guided by the measurement, thus, is able to generate more accurate histograms.

3. UNIFORMITY AND ENTROPY
This section introduces the entropy plot and how to use it to measure the accuracy of a histogram.

3.1 Bucket Uniformity
Suppose we are building a histogram on a set of columns. The columns form a multi-dimensional Cartesian space. The histogram usually divide the multi-dimensional space into non-overlapping hyper-rectangles such that the whole space is covered by the hyper-rectangles. These hyper-rectangles are usually known as buckets. The histogram maintains the boundaries of the buckets and the number of tuples within the buckets.

The accuracy of the histogram depends on the tuple distribution within the histogram buckets. A range predicate is also a hyper-rectangle within the multi-dimensional space and the selectivity estimation is to find out the number of tuples within the hyper-rectangle. The estimation algorithm scans all the buckets that have overlap with the predicate and assumes uniform tuple distribution within each bucket to estimate the tuples in the overlap region. When a bucket has a partial overlap with the predicate, the contribution of the bucket is the product of the total number of tuples within the bucket and the ratio of the overlap volume to the bucket volume. Thus, the accuracy of the histogram is determined by how close the tuple distribution is within the buckets to the uniform distribution. The closer the buckets are to the uniformity, the more accurate the selectivity estimation will be. The goal of the histogram construction, therefore, is to divide the multi-dimensional space into buckets of high uniformity.

Uniformity of the tuple distribution within the histogram buckets comes from the following two aspects.

1. **Uniform spreads.** The column value combinations that appear in the relation should uniformly spread out within each bucket.
2. **Uniform frequencies.** All the column value combinations have the same frequencies in the relation.

Both conditions are critical for accurate selectivity estimation. MaxDiff(V,A) outperforms most one-dimensional histograms because it takes both spread and frequency into consideration during the construction; however, rather than using a measurement for bucket uniformity, it relies on the heuristic rule that the difference in area is a good indication of the bucket uniformity. With an accurate measurement of the bucket uniformity, the construction algorithm should produce histograms of higher quality.

This project proposes the entropy plot for bucket uniformity measurement. The entropy plot is based on the concept of entropy, which measures the uniformity of a discrete probability function. The following section presents entropy and entropy plot.

3.2 Entropy and Entropy Plots

Entropy measures how far away a probability distribution function is from the uniform distribution. **Entropy** on a discrete probability distribution function $P = \{p_i | i = 1, 2, \ldots, N\}$ is defined as

$$H(P) = -\sum_{i=1}^{N} p_i \log_2 p_i, \qquad (1)$$

The entropy value H equals to $\log_2 N$ if P is a uniform distribution function. High skewness of the distribution results in small entropy value. When H reaches the minimum value of 0, the probability function P looks like this:

$$p_i = \begin{cases} 1, & i = k; \\ 0, & i \neq k. \end{cases}$$

Applying entropy to a histogram bucket tells whether the column value combinations have uniform frequencies, but it provides no information about the spread uniformity of these value combinations. The latter information can be gathered by calculating the entropy value at different granularities.

Let's first use a one-dimensional bucket as an example to illustrate the idea. The value range of the bucket is divided into 2^n equi-length intervals at scale n and the number of tuples in each of the intervals are counted. The entropy value at scale n is then calculated using these counts as the probability that a tuple falls into these intervals. Plotting the entropy value against the scale yields the **entropy plot** for the bucket. It is first used in bursty traffic modeling [9] for measuring the burstiness of the traffic.

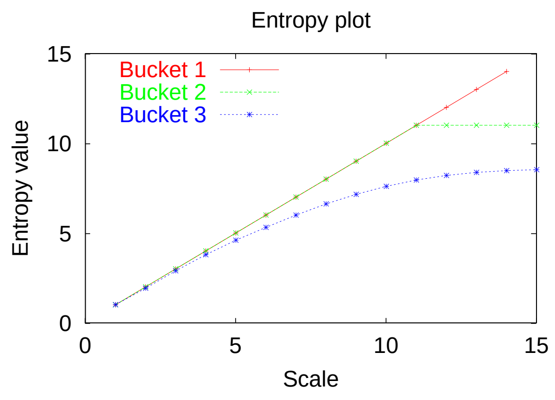


Figure 1: Uniformity and entropy plot.

The shape of the entropy plot tells how far away the bucket is from an ideally uniform bucket. If the bucket is uniform in both spread and frequency, the entropy value equals to n when the scale n is small as the equi-length intervals should contain roughly the same number of tuples. When the scale n becomes larger such that the interval is smaller than the spread, further dividing the intervals doesn't change the entropy value any more and the entropy plot shows a flat tail of $\log_2 V$, where V is the number of column value combinations within the bucket.

Figure 1 shows the entropy plots on three one-dimensional buckets.

- **Bucket 1** is uniform in both spread and frequency. The spread is 1. The entropy plot forms a line of slope 1. No further division of the bucket is needed.
- **Bucket 2** is also uniform in spread and frequency, but the spread is 32. The entropy plot starts with slope 1 and becomes flat after the interval is smaller than 32. No further division of the bucket is needed, either.
- **Bucket 3** is not uniform in either spread or frequency. Further division of the bucket into several buckets should yield better selectivity estimation.

Extending the entropy plot to multi-dimensional buckets is straightforward. At scale n, all the d dimensions of a bucket is divided into 2^n equi-length intervals and the entropy value is calculated on the $(2^n)^d$ hyper-cubes. If the bucket is uniform in both frequency and spread, the entropy plot assumes a slope of d at small scales and has a flat tail at value $\log_2 V$, where V is the number of distinctive value combinations in the bucket.

3.3 Error Estimation

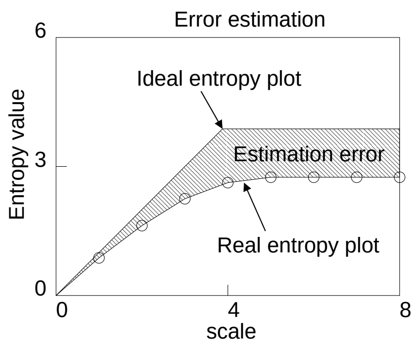


Figure 2: Error estimation for a bucket.

The distance between the entropy plots of a bucket and the ideal uniform bucket is proportional to the estimation error if the bucket is used for selectivity estimation. Ideally, the entropy plot for a uniform bucket has a slope of d at small scales and a flat tail of value $\log_2 V$, where V is the number of distinctive value combinations in the bucket. The real entropy plot of the bucket always lies beneath the ideal entropy plot. The area between the two entropy plots gives the selectivity estimation error (the shaded area in Figure 2).

Calculating the area requires complex computation. An approximation of the area is the sum of the difference between the entropy values at all the scales. In the following discussion of the algorithm, we use this approximation instead of the real area as the measurement of the selectivity estimation error.

4. ENTROPY-BASED HISTOGRAM CONSTRUCTION

This section describes the top-down histogram construction algorithm based on selectivity estimation error measurement.

4.1 Algorithm Overview

The construction algorithm adopts a top-down approach as outlined in Figure 3. It starts with a bucket that contains the whole multi-dimensional Cartesian space and recursively selects a bucket for splitting if more storage space is available. Two decisions are involved here: to select a bucket for division and to figure out how to split the bucket. The following two sections discuss the solutions in details.

4.2 Finding Split Bucket

The algorithm should select the bucket that yields the greatest reduction in the overall selectivity estimation error. We approximate this by choosing the one with the greatest selectivity estimation error given by the entropy plot. When comparing the errors of the buckets, several weighting factors can be taken into consideration.

- **Tuples in bucket.** Buckets with a large number of tuples in them should have high priorities over small buckets.
- **Workload.** Buckets of high access frequency should have high priorities.

- **INPUT:** The columns in a relation for histogram construction and the number of available storage space.
- **OUTPUT:** A histogram on the designated columns.
- **ALGORITHM:** Build a bucket to contain the whole multi-dimensional space. Repeat the following steps if there is still storage space available.
  1. Select a bucket for splitting
  2. Decide the best division for the bucket.
  3. Split the bucket into two buckets.

Figure 3: Top-down construction algorithm.

In the experiments, the algorithm weighs the estimation error by the number of tuples in the bucket and selects the one with the maximum weighted error. Actually, this reduces the expected estimation error if the query follows the tuple distribution.

4.3 Finding Split Point

Similarly, the best split point should produces the greatest reduction in the estimation error. Exhaustive search involves too much computation; so we use heuristic rules to find the best split point.

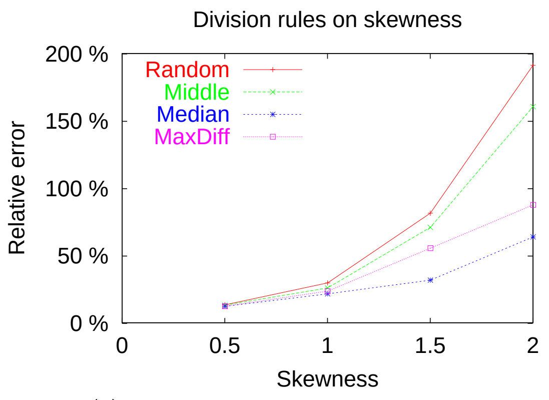
4.3.0.1 One-dimensional histograms.

The following rules for splitting the histogram buckets are considered.

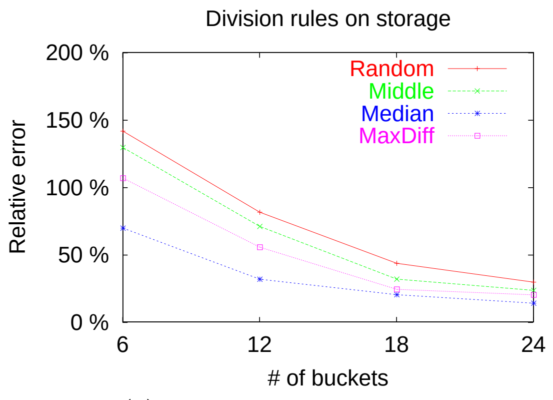
- **Rule Random** selects a random value within the bucket.
- **Rule Middle** chooses the value that divides the bucket into two buckets of the same number of distinctive value combinations.
- **Rule Median** uses the median value so that the two new buckets have roughly the same number of tuples.
- **Rule MaxDiff** employs the value that has the maximum difference in area change as in MaxDiff(V,A).

Figure 4 compares the performance of these rules on synthetic workload as described in Section 5.2. (a) compares the performance of the rules on data of different skewness with 12 bucket histograms and (b) shows the effect of the storage space on data of skewness 1.5. Overall, Rule Median works the best.

The difference between the rules becomes significant for data of high skewness as Figure 4 (a) has shown. For data of low skewness, the tuple distribution is very close to uniform; therefore, the position for the splitting doesn't make a lot of difference in the quality of the histograms. On the other hand, picking the wrong position can be disastrous for data



(a) On dataset of different skewness



(b) On different storage space

Figure 4: Comparison of the splitting rules for one-dimensional histograms. The error is averaged over 10,000 queries on 10 randomly generated datasets of the same skewness. Each dataset contains 500,000 tuples on 100 distinctive values ranging from 0 to 500. (a) assumes 12 buckets and (b) assumes the skewness of 1.5.

of high skewness. A good rule for finding the split point is very important.

The accuracy of the selectivity estimation improves when more storage space is allowed because the histograms approximate the real data distribution better with more buckets. The distinction for different splitting rules diminishes with more storage; however, the relative position doesn't change. Rule Median remains the best one for all the cases.

4.3.0.2 Multi-dimensional histograms.

The algorithm needs to decide not only the split point, but also along which dimension the splitting should be for multi-dimensional histograms. Once the splitting dimension is decided, the algorithm can use the similarly splitting point rules as in the one-dimensional histogram construction algorithm. Finding the splitting dimension also follows heuristic rules. Following are some of the rules.

- **Rule Optimal** finds the dimension that produces the maximum reduction in estimation error.
- **Rule Random** randomly picks a dimension.
- **Rule Range** selects the dimension with the largest value range.
- **Rule MaxDiff** picks the dimension with the greatest difference in area.

In the experiments, we use Rule Range to determine the split dimension and Rule Median for finding the split point.

4.4 Construction Cost

Calculation of the entropy plot needs calculation of entropy value on all the granularities. This can be done in a single pass of the data because the calculation on different is independent. After the bucket splitting, the algorithm needs to calculate the entropy plot for the two new buckets.

The entropy-based histogram construction algorithm requires more computation than MaxDiff(V,A) as we need to recalculate the entropy plots for the new buckets after each bucket splitting.

The construction cost for multi-dimensional histograms is in the same order as MHIST-2. MHIST-2 searches for the bucket with the maximum difference in area for splitting and the entropy-based algorithm searches for the one with the maximum estimation error. Both require recalculation of the statistics of the new buckets for further splitting.

5. EVALUATION METHOD

This section evaluates the entropy-based histogram construction algorithm using synthetic workloads. The results show that the entropy-based histograms provide more accurate selectivity estimations in most cases because of the right guideline for bucket splitting.

5.1 Methodology

We compare the accuracy of histograms by relative error. The relative error for a query is defined as

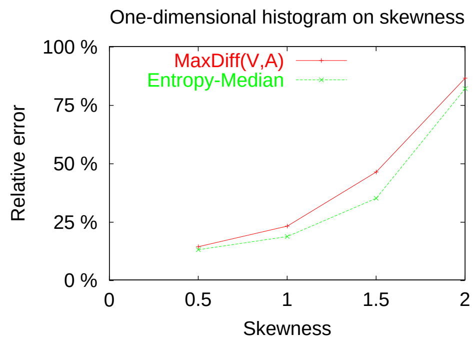
$$Error = \frac{Q - E}{Q}, \qquad (2)$$

where Q is the result size and E is the estimation from the histograms.

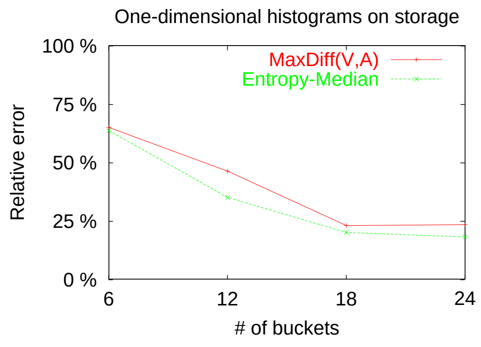
The following two sections present the comparison for one-dimensional and multi-dimensional histograms respectively. All the queries are generated randomly and they all have result sizes larger than 10. Each point in the graph is the average relative error of 10,000 queries over 10 random datasets. The dataset generation algorithm is presented in the following sections.

5.2 One-Dimensional Histograms

The one-dimensional histograms are evaluated with datasets of Zipfian distribution. There are total of 500,000 tuples on



(a) On dataset of different skewness



(b) On different storage space

Figure 5: Comparison of entropy-based one-dimensional histogram to MaxDiff(V,A).

100 distinctive values ranging from 0 to 500 and the tuple distribution has skewness ranging from 0.5 to 2.

Figure 5 compares the performance of the entropy-based histograms to MaxDiff(V,A) histograms on (a) data of different skewness for 12 bucket histograms and on (b) histograms of different size for data of skewness 1.5. All the histograms have the same type of buckets; so we show the storage space as the number of buckets. The histograms store the minimum value, the maximum value, the number of distinctive values, and the total number of tuples for a bucket. That is, 8 bytes per bucket. A histogram of 12 buckets occupies 96 bytes.

The entropy-based histogram outperforms MaxDiff(V,A) even though MaxDiff(V,A) is proved to have the best overall performance by previous study. Entropy-based histogram construction directs the bucket splitting to the one in the most need. On contrary, MaxDiff(V,A) bases the bucket boundaries on values of maximum area difference in the hope that it produces the uniform buckets. Apparently, the entropy-based histogram construction algorithm is more effective in producing high quality histograms.

5.3 Multi-Dimensional Histograms

Two datasets are used in comparing the multi-dimensional histograms.

- **Dataset Zipf** is generated with Zipfian distribution on random value combinations within two-dimensional space. The cardinality of the relation is 1,000,000. All the columns have 100 distinctive values ranging from 0 to 500. The value combinations are randomly generated along each dimension. The skewness ranges from 0.5 to 2.
- **Dataset Gaussian** has 10 independent Gaussian clusters of tuples within the two-dimensional space. The number of tuples of the clusters follows Zipfian distribution of skewness from 0.5 to 2. The standard deviation of the Gaussian clusters is 25. The value ranges from 0 to 500 on each dimension.

We present the results on the following histograms.

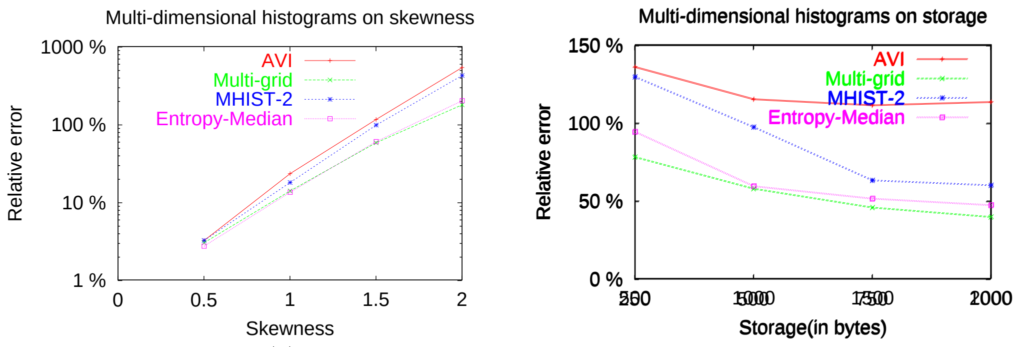
- **Attribute Value Independence.** One-dimensional histograms are maintained on the columns and the selectivity estimation assumes no correlation between the column values.
- **Multi-grids.** The multi-dimensional space is divided into regular grids and the boundaries of the grids are derived from the one-dimensional MaxDiff(V,A) histograms.
- **MHIST-2.** The algorithm recursively finds the bucket that has the maximum difference of area along one dimension among all the buckets and divides the bucket into two.
- **Entropy-Median.** The top-down multi-dimensional histogram construction algorithm uses the median value as the division point for splitting the buckets.

In the first case, all the storage space is divided evenly on all the dimensions for the one-dimensional histograms. For all the other histograms, each bucket takes up $2 \times d + 1$ values, where $2 \times d$ are for the boundaries on d dimensions and the additional value for the total number of tuples. We assume that each value takes 2 bytes in storage.

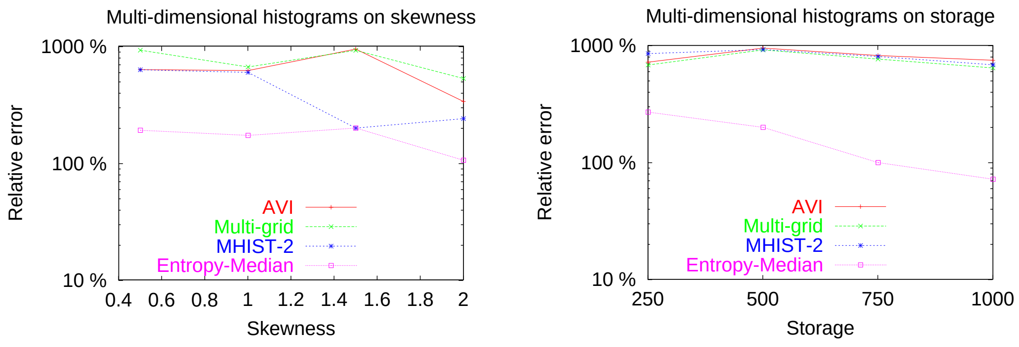
Figure 6 shows the relative error of these histograms on the two datasets. The graphs show the sensitivity of the histograms to the data skewness with 500-byte histograms and the sensitivity to the available storage space on datasets of skewness 1.5. Different histograms yield different accuracy for the two datasets because of the different characteristics of the datasets. Overall, the entropy-based histograms approximate both datasets better than the other histograms.

Generally, the histograms of the same size become less accurate for datasets of high skewness and additional storage space usually improves the accuracy of the histograms.

Entropy-based histograms produces accurate approximation of the tuple distribution in most cases. For clustered datasets like Dataset Gaussian, entropy-based histograms perform significantly better than all the other histograms by focusing on the regions of irregular tuple distribution. On dataset of higher degree of random tuple frequencies but less ran-



(a) Performance comparison on Dataset Zipf.



(b) Performance comparison on Dataset Gaussian.

Figure 6: Multi-dimensional histogram comparison.

dom value spreads, multi-grid histograms work as well as, sometimes even better than, the entropy-based histograms.

5.4 Summary

The entropy-based histogram construction algorithm identifies buckets of high degree of irregularity for further refinement. The result on one-dimensional histograms implies that entropy plot approximates the estimation error very well. Entropy-based multi-dimensional histograms produce histograms of high quality. Better heuristic rules on bucket splitting should lead to even better performance.

6. CONCLUSIONS

The project proposes a top-down histogram construction algorithm employing the entropy plot as the measurement of bucket uniformity. By calculating the entropy value at different scales for histogram buckets, the entropy plot approximates the selectivity estimation error of the buckets for range predicates. Thus, the construction algorithm is able to focus on buckets of high estimation error for refinement, resulting in histograms of high accuracy.

The algorithm handles both one-dimensional and multi-dimensional data. The evaluation has shown that the entropy-based histogram construction algorithm is able to generate high quality histograms and outperforms existing histograms in most cases.

7. REFERENCES

[1] Ashraf Aboulnaga and Surajit Chaudhuri. Self-tuning histograms: building histograms without looking at data. In *SIGMOD'99*, 1999.

[2] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. STHoles: a multidimensional workload-aware histogram. In *SIGMOD'01*, 2001.

[3] Lise Getoor, Ben Taskar, and Daphne Koller. Selectivity estimation using probabilistic models. In *SIGMOD'01*, 2001.

[4] D. Margaritis, C. Faloutsos, and S. Thrun. NetCube: a scalable tool for fast data mining and compression. In *VLDB'01*, 2001.

[5] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. Wavelet-based histograms for selectivity estimation. In *SIGMOD'98*, 1998.

[6] M. Muralikrishna and D. J. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In *SIGMOD'88*, 1988.

[7] V. Poosala, Y. Ioannisdis, P. Haas, and E. Shekita. Improved histograms for selectivity estimation of range predicates. In *SIGMOD'96*, 1996.

[8] Viswanath Poosala and Yannis E. Ioannidis. Selectivity estimation without the attribute value indepedence assumption. In *VLDB'97*, 1997.

[9] M. Wang, A. Ailamaki, and C. Faloutsos. Capturing the spatio-temporal correlation in real traffic data. In *Performance'02*, 2002.

On the Performance of Main-Memory Indices

Démian Nave[*]
Pittsburgh Supercomputing Center
Carnegie Mellon University
Pittsburgh, PA 15213
dnave@psc.edu

ABSTRACT

Databases in main memory rely heavily upon fast and efficient in-core indices, much as traditional databases rely upon highly-tuned out-of-core indices. Several cache-friendly index data structures have been suggested and validated experimentally and/or theoretically. However, there is a distinct lack of empirical data covering a wide range of architectures, workloads, index data structures, and other parameters to justify (or disprove) the need for cache-aware data structures on current hardware.

The study conducted in this paper is a first look at the performance of typical main-memory index data structures. Results show that the T-tree is a poor choice for main memory indices, while B-tree variants perform very well across many problem sizes. Additionally, experiments interchanging linear and binary search within a tree node show the non-intuitive result that linear search generally outperforms binary search.

1. INTRODUCTION & RELATED WORK

The design of data structures and algorithms meant to tolerate the ever widening "processor-memory gap" is a fundamental challenge in building efficient database systems [2, 6]. Available experimental evidence, although primarily measuring performance of disk–based systems, intuitively suggest similar problems for databases in main memory.

Chilimbi et al. [10] describe general techniques such as structure reordering and tree coloring for improving the cache performance of pointer-based data structures at the programming and compiler level. Graefe and Larson [12] review existing techniques to improve the cache performance of B-trees, including prefetching, cache-line sized nodes, and compression. Ailamaki et al. [1] propose a per-page columnar layout of on-disk relations, and show that this fairly simple change results in meaningful improvements to a commercial database system. Madden et al. [14] briefly describe their use of *State Modules* (STeMs) [15] which encapsulate an index data structure for processing continuous joins on data streams. In this context, an efficient main-memory index data structure is essential to support high-throughput responses to continuous queries.

Cache–oblivious algorithms [11] and data structures [3, 4] have been shown to be optimal with respect to the number of memory block transfers, independent of the transfer (e.g. cache block) size. Though theoretically optimal, existing cache-oblivious data structures are not yet ready for general use; Prokop conjectures that at cache–obliviousness costs at least $O(\log N)$ versus the best application–specific algorithms and data structures. Indeed, insertion and deletion in the best available data structure has *amortized* complexity $O(\log_B N + \log^2 N/B)$, where N is the number of keys, and B is an unspecified unit of transfer. This is a logarithmic factor more expensive than the guaranteed $O(\log_B N)$ insertion and deletion complexity of B-trees.

On the other hand, *cache–conscious* data structures [9] take advantage of prior knowledge of the underlying memory hierarchy to better organize data for both temporal and spatial locality. A number of cache-conscious index data structures (for general keys) have been proposed, including T-trees [13], Cache-Sensitive B$^+$-trees (CSB$^+$-trees) [17], partial key T- and B-trees (pkT-, pkB-trees) [5], prefetching B$^+$-trees (pB$^+$-trees) [7], and fractal prefetching B$^+$-trees (fpB$^+$-trees) [8]. Each of these data structures is designed to make best use of the memory hierarchy by intelligently organizing both the nodes of the search tree and the tree itself.

The T-tree [13] appears to be the earliest suggested index data structure designed to take advantage of the memory hierarchy. A T-tree is a balanced binary search tree whose nodes contain several keys. The left child of a T-tree node contains keys less than the minimum key in the node, while the right child contains keys more than the maximum key. As new keys are inserted and deleted, rotations like those of an AVL tree are performed to keep the tree balanced. T-trees have the advantage of simplicity, but the depth of the tree and the need to balance the tree by rotations appear to decrease performance versus cache-conscious trees [16].

CSB$^+$-trees [17] store the children of a B$^+$-tree node con-

[*]This work was supported by the NIH National Center for Research Resources #P41 RR06009

HotDB'02, December 15, 2002, Pittsburgh, PA.

	CPU/MHZ	RAM	OS	Compiler
Alpha 4-way	21264A/667	4GB	Tru64 Unix V5	cxx V6.5
Intel 2-way	PIII/733	.5GB	Linux V2.4.18	g++ V3.2

Table 1: Basic parameters of the machines used to collect data.

tiguously in memory. As a result, internal tree nodes need contain only a single pointer to the beginning of the memory block containing its children. This allows more keys to be packed into a tree node, both increasing fan-out and decreasing tree depth. The improved search performance comes at the price of increased cost of insertion and deletion, since more keys must be copied each time a node is split or coalesced. The authors propose segmented nodes and memory preallocation to reduce the cost of updates, but their experiments demonstrate that updates are still more expensive than in typical B$^+$-trees. It may be worthwhile to note that their experiments do not make clear the impact on performance of better memory management (rather than cache management).

Partial key trees [5] are designed to improve cache performance by storing only the difference between lexicographically adjacent keys. This has the effect of reducing cache misses in the case of arbitrary-length keys, but updates and searches are more expensive than a typical main-memory (but not necessarily cache-conscious) B-tree implementation. The authors conjecture that partial-key trees will outperform T- and B-trees as long as processor speed improves more quickly than main memory latency.

Chen et al. [7] suggest the insertion of hardware prefetch instructions into the search and range-scan functions of a typical main-memory B$^+$-tree. Rather than explicitly modifying the index data structure to reduce cache misses, this technique serves to provide better overlap of cache misses and useful work by leveraging the non-blocking cache and highly parallel pipeline offered in recent processors. Their simulations show that, given an architecture fitting the simulation parameters, B$^+$-tree search can be improved by nearly 30% by using larger nodes and by inserting instructions to prefetch adjacent cache-line sized blocks within a node.

In a later paper [8], Chen at al. propose a two-level data layout scheme to help optimize both disk and cache performance. At the coarse level, a B$^+$-tree with large node width is constructed as the primary layout. The nodes of this tree are either page-size nodes enclosing small second-level search trees (disk-first), or variable-width nodes enclosing interior nodes from a cache-optimized tree (cache-first). The performance data suggest that these data layouts are quite effective at improving disk and cache performance, but at the expense of increased space (highly undesirable for in-core databases). Even so, the combination of prefetching with a more advanced main-memory specific tree layout may prove to be highly effective in improving the performance of in-core indices.

Although a variety of main-memory index schemes are available, it is not clear which offer the most promise in improving the performance of main-memory database or data stream operations. Very little "real-world" experimentation has been provided; in fact, most of the experiments assume both fixed-size keys and a uniform key space (even in the case of the partial key trees [5], though they use large key sizes), and many present simulator data as the primary source of results. Moreover, the cache-aware data structures seem to incur some performance penalty with respect to their simpler counterparts. For example, pkT-trees and pkB-trees require more expensive search algorithms which affect overall performance, and all of the cache-conscious indices except pB$^+$-trees incur significant overhead from complex updates.

Finally, over all of the work reviewed, there was very little variety in the experimentation; consequently, there seems to be no clear evidence to motivate any of the cache-aware alternatives to simple B-trees and their variants. The study conducted in this paper is a first look at the performance of popular main-memory index data structures, with the primary aim of focusing attention on the best-performing data structure for future (or even current) improvements. The data presented in Section 3 were gathered from several index structures: T-trees, standard B-trees, "fat" B-trees in which leaves are packed with keys, and standard B$^+$-trees. Implementation details are provided in Section 2. Results show that:

1. All of the B-tree variants performed far better than T-trees, and basic B-trees performed better than fat B-trees and B$^+$-trees for small problem sizes (2 million operations or smaller).

2. For the problem sizes studied, binary search within an index node performed significantly better only for T-trees (this data will not be presented). For the B-tree variants, binary search outperformed linear search only for large node sizes, but generally did not reduce run-time below the best time with linear search.

2. IMPLEMENTATION

Four basic data structures were implemented for the performance analysis presented in Section 3: T-trees, standard B-trees, "fat" B-trees in which leaves are filled with keys, and standard B$^+$-trees. For all of the data structures, keys can be referenced directly or indirectly, depending upon the word size of the target machine and the needs of the application. Also, updates were implemented recursively, while searches were implemented iteratively. These choices were made solely to reduce the complexity of the implementation.

T-Trees A T-tree is a wide-node variant of an AVL binary search tree. Like an AVL tree, search time is bounded due to the balance condition, and balance is maintained by rotations (simple pointer adjustments). However, a T-tree node can contain several keys, thus potentially reducing the height of the search tree and

		Size	Line Size	Word Size	Assoc.	Lat.	Miss
Alpha 4-way	Icache	64K	64B	8B	pseudo 2-way	3	13
	Dcache	64K	64B		set 2-way	4	14
	L2	8M	64B		direct	6	≈ 110
Intel 2-way	Icache	16K	32B	4B	4-way	3	13
	Dcache	16K	32B		4-way	3	14
	L2	256K	32B		8-way	4	≈ 115

Table 2: Memory hierarchy and other configuration parameters for each machine. Latencies are expressed in cycles.

improving cache utilization. T-trees are simple to implement, but fair poorly due to high data movement and search costs. The overhead per node is 2 pointers (one to each child) and one or two integers, depending upon how the key count and node balance variables are stored. In the experiments, the 2 integer version was used to help align the node in memory.

B-Trees The prototypical B-tree is an efficient multi-way search tree in which balance is maintained by constructing the tree roots-first through node splitting, merging, and rebalancing. Each node of a B-tree normally contains m pointers to child subtrees and $m-1$ keys from the indexed data set. The value of key i in a B-tree node is guaranteed to lie between the values of the keys in subtrees i and $i+1$. The fundamental benefit of B-trees is the shallow tree depth–$O(\log_m N)$–where N is the number of indexed keys. A B-tree node requires a boolean flag to indicate whether the node is a leaf or interior, and an integer containing the number of keys. A space efficient means of storing these values is by using bit-fields, which also helps to align a tree node in memory.

"Fat" B-trees Fat B-trees support the obvious optimization of packing $2m-1$ keys into the leaf nodes of a B-tree (interior nodes are unchanged). Some overhead may be incurred due to having two different memory layouts and two functionally different node types, but in many cases the savings in memory versus storing empty child pointers can offset these costs.

B$^+$-Trees A B$^+$-tree is a B-tree variant in which the leaf nodes of the tree contain the keys to be indexed, while the interior tree nodes contain only a subset of the keys to guide searches. B$^+$-trees are ideal for workloads which require fast linear searching of indexed while still enabling fast searching due to a shallow tree depth. Updates are somewhat more difficult with B$^+$-trees as compared to B-trees, and in particular deletion.[1] B$^+$-tree nodes are identical to fat B-trees except for leaf nodes, in which an additional "sequence pointer" takes the place of a key to link one leaf node with the next leaf in key order.

In each of these data structures, keys are grouped together with the corresponding child pointers grouped immediately after. This layout optimizes searches within a node by clustering keys in contiguous cache blocks. For shallow trees, cache misses due to searching down the tree would intuitively be less costly than those searching within a (wide) node. For the B-tree variants, interior and leaf nodes were constrained to be the same size (with different numbers of keys) for simplicity. The basic variable in all of the data structures is the node size, which bounds the maximum number of keys per node. For the T-tree, the difference between the minimum and maximum number of keys was fixed at 2. For the B-tree variants, if m is the desired minimum number of interior keys, then the minimum and maximum number of interior and leaf node keys are:

- maximum $2m-1$ for all interior nodes (and B-tree leaf nodes);
- minimum $2m$ and maximum $4m-1$ keys for fat B-tree leaves; and
- minimum $2m-1$ and maximum $4m-2$ keys for B$^+$-tree leaves.

These constraints were chosen to ensure that exactly $4m-1$ pointers or keys were stored per node. In the case of the B$^+$-tree, one slot in a leaf node is reserved for a sequence pointer to allow for range scans over the tuples pointed to by the leaf keys. Keys are assumed to be either pointers into a key (e.g. for string keys), or the key value itself (e.g. when a table consists of only a column of integral values). This decision was made as a trade-off to storing arbitrarily sized keys inside a node; while this is possible, it is far easier to indirectly reference a key from the tuple it indexes. Furthermore, storing arbitrary keys inside a node would increase the memory footprint of the index and potentially increase the tree height; both of these effects in aggregate are arguably less desirable than the additional costs of storing pointers to the keys.

3. EXPERIMENTATION

Data were collected from two systems: a 4-way Alpha SMP and a 2-way Intel SMP, using only a single processor on each machine. These SMP machines were chosen since they represent typical database platforms available today. Table 1 gives the basic system parameters for these machines. Memory hierarchy configurations and other CPU-related parameters appear in Table 2. All code was written in C++ without templates or polymorphism, and native compilers

[1]At the expense of some additional logic, deletion could be implemented by marking off deleted keys, since the index is expected to grow.

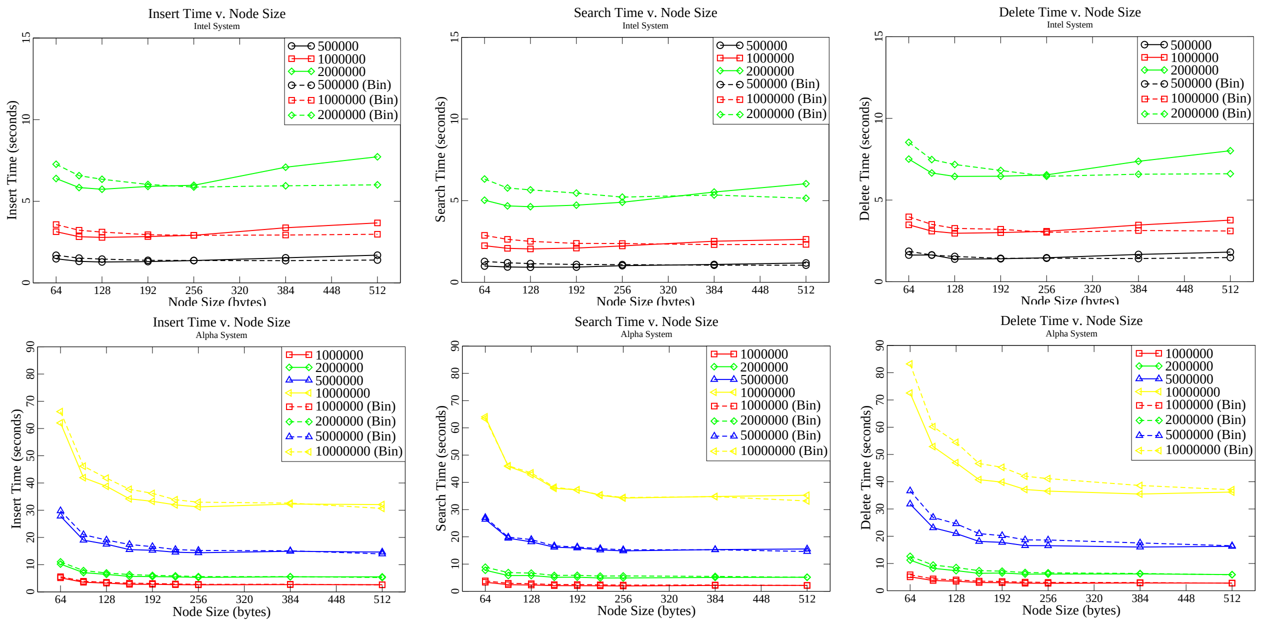


Figure 1: ***B-tree insertion, search, and deletion performance—Intel (Top) and Alpha (Bottom) systems.*** **These plots show that different experiment sizes produce similar trends. "(Bin)" in the legend refers to the binary search experiments (dashed lines).**

with architecture-specific optimization were chosen for compilation when available.[2]

The results presented below were drawn from experiments over the following group of parameters:

Systems	Alpha, Intel
Operations	Insert, Delete, Search
Operation count	250,000–10,000,000
Node sizes	64–512 bytes
Node search	Linear and Binary Search

The times shown in all plots are the average of 10 runs (Alpha system) or 5 runs (Intel system). Note that outliers—data points more than 1 standard deviation from the mean—are included since there was little effect on the final data. Less than 5% of data consisted of outliers, all of which can be attributed to spontaneous operating system cycles on otherwise idle machines.

The following set of experiments were conducted:

The Insertion Experiment. In each insertion experiment, a stable index was first built from an in-memory table consisting of 1 million random word-size keys. Then, N random word-size keys were inserted into the table, where N is one of the problem sizes described above.

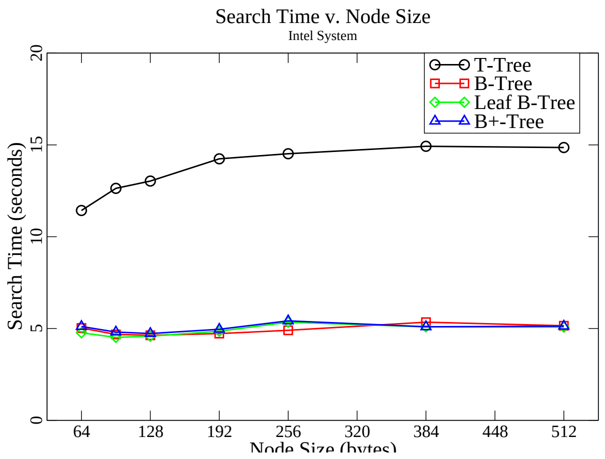
The Search Experiment. In each search experiment, a stable index of N random word-size keys was first built from an in-memory table, where N is one of the problem sizes described above. Then, the search procedure was called for each of the N keys, randomly shuffled.

The Deletion Experiment. In each deletion experiment, a stable index of $N + 1,000,000$ random word-size keys was first built from an in-memory table, where N is one of the problem sizes described above. Then, the deletion procedure was called on N keys, randomly shuffled.

Figure 1 depicts the raw insertion, deletion, and search performance data for the B-tree implementation on both systems for several experiment sizes. These data show the "optimal" node sizes at the minima of the computation time curves for each problem size; in general, the minima lie on the linear search curves. In the Intel experiments, the optimal size falls between 96 and 128 bytes, while, in the Alpha experiments, the optimal size falls between 224 and 256 bytes. Note that additional experiments are needed to determine if prefetching within a node would further improve performance for linear node search, since the optimal node sizes are already quite large. Also, large node sizes generally degrade performance; this suggests that node-grouping B-tree variants like CSB$^+$-trees would most likely perform poorly, especially for update-heavy workloads.

It is interesting to note that, although binary search offers lower performance than linear search at large node sizes, the computation time with binary search flattens as the node size grows. This is to be expected, since intuitively, binary search is cheaper than linear search when the logic and memory-to-cache data movement required to adjust the

[2]Compaq CXX V6.5 offers very powerful optimization options, like interprocedural optimizations and object file reorganization. The only advanced option I used was blind (i.e. without profiling) object file reorganization to potentially improve instruction ordering.



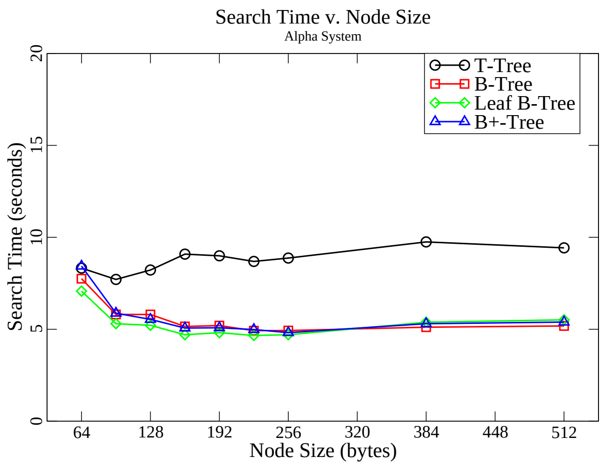


Figure 2: ***Index search performance for 2 million operations—Intel (Top) and Alpha (Bottom) systems.*** **T-trees perform poorly, while the B-tree variants all have similar performance. Note that the optimal node sizes are quite large, at 128 bytes for the Intel system and 256 bytes for the Alpha system.**

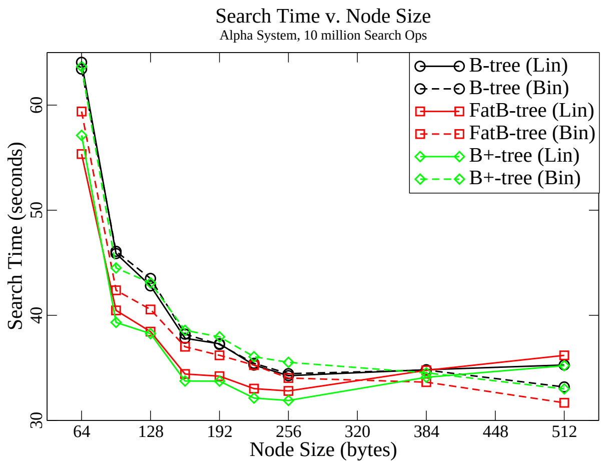


Figure 3: ***B-tree, Fat B-tree, and B^+-tree search performance for 10 million operations—Alpha system.*** **The B^+-tree index offers the best search performance for this large problem size.**

search interval endpoints becomes less expensive than the memory-to-cache data movement cost of linear search. This trend holds for all of the B-tree variants; in the T-tree experiments (not shown), binary search always performs better than linear search due to the higher T-tree node occupancy (generally $m - 2$ elements if m is the maximum number of keys).

Figure 2 depicts the search performance for all of the implemented index structures (insertion and deletion show roughly the same trends as in fig. 1, with deletion being the most expensive operation). The various B-tree computation time curves for the Intel experiments are composed of data from both the linear and the binary node search experiments to better demonstrate the performance of those data structures. Times to the left of the 256 byte node size correspond to linear search, while those to the right correspond to binary search. The 256 byte node size was chosen as the transition because the linear and binary search time curves for the B-tree variants crossed at or near 256 bytes in all of the Intel experiments. The Alpha plots are not adjusted, since the crossover occurs at or near 512 bytes in these experiments.

Note that, for the large Alpha experiments, the crossover node size from linear search to binary search appears to be moving closer to the optimal node size for linear search. A closer view for 10 million searches with the B-tree, fat B-tree, and B^+-tree appears in fig. 3. Additional experiments are needed to follow this trend. This figure also shows that the B^+-tree performs slightly better than the other B-tree variants in this large search experiment. For insertion and deletion, however, the fat B-tree performs the best due to reduced logic versus the B^+-tree, and due to decreased tree height and better memory performance at the leaves.

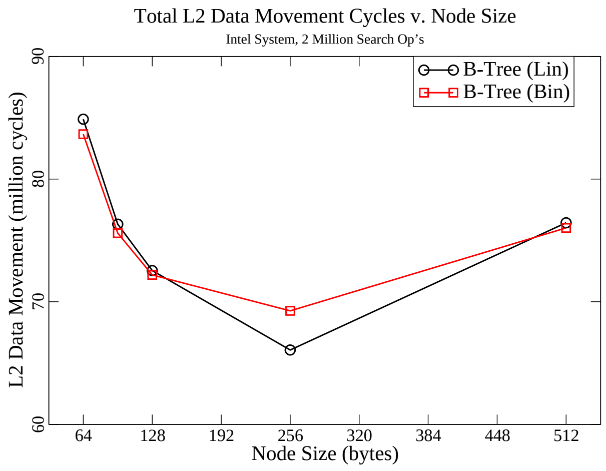


Figure 4: ***Total L2 data movement cost (cycles) for 2 million B-tree searches—Intel system.*** **This figure depicts the total number of cycles that the L2 was busy moving data to and from the CPU.**

The final plot, fig. 4, depicts the total number of cycles that the L2 was busy moving data to and from the CPU during the B-tree 2 million search experiment on the Intel system. This data gives some idea of the upper-level memory traffic seen during this search experiment. For the 128 byte node size, roughly 73 million cycles were spent by the L2 moving data, or about .1 seconds out of 5. This plot also shows that the L2 spent about the same amount of time moving data for both linear and binary search at the 128 byte node size. This is not surprising since, on average, the nodes of the B-tree are 70% full (around 11 keys with a 128 byte node size), and both linear and binary search would visit about half of those keys.

4. CONCLUSIONS

The results of the study conducted in this paper show that the T-tree, an often implemented index, is a poor choice for main memory indices, even for the simple queries used in the experiments. B-trees, on the other hand, perform very well, with a basic B-tree implementation performing well with small problems, and with the fat B-tree performing better for large problems. The data also support the non-intuitive result that linear search generally outperforms binary search for the optimal node sizes found in the experiments.

There is a wide variety of experiments yet to perform to better understand how main-memory indices behave on available hardware. For example:

- Much larger experiments should be performed than those presented here; incorrect conclusions could be drawn from the small problem size experiments.
- The cache performance of the B-tree indices should be measured more rigorously with hardware performance counters where available. For example, the Intel Pentium 4 processor provides a wealth of information through its performance counters.
- Cache misses cost more cycles on a Pentium 4 than on the Pentium III studied here, so it may be worthwhile to test cache-aware data structures on the Pentium 4.
- Application-specific microbenchmarks should be run with real query mixes to guage the effects of interaction between index operations.

Additional experimental data is available upon request from the author.

5. ACKNOWLEDGEMENTS

A portion of this work was performed using the computational facilities of the Terascale system at the Pittsburgh Supercomputing Center, Pittsburgh, PA. The author is also indebted to the reviewers for gently suggesting ways to greatly improve an original draft of this work.

6. REFERENCES

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *Proceedings of 27th VLDB Conference*, pages 169–180, 2001.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. Dbmss on a modern processor: Where does time go? In *Proceedings of the 25th VLDB Conference*, pages 266–277, 1999.

[3] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious b-trees. In *Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science*, pages 399–409, 2000.

[4] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 29–38, 2002.

[5] P. Bohannon, P. Mcllroy, and R. Rastogi. Main-memory index structures with fixed-size partial keys. In *Proceedings of the ACM SIGMOD Conference*, pages 163–174, 2001.

[6] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the 25th VLDB Conference*, pages 54–65, 1999.

[7] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. In *Proceedings of the ACM SIGMOD Conference*, pages 235–246, 2001.

[8] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin. Fractal prefetching b+-trees: optimizing both cache and disk performance. In *Proceedings of the ACM SIGMOD Conference*, pages 157–168, 2002.

[9] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN PLDI Conference*, pages 1–12, 1999.

[10] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Making pointer-based data structures cache conscious. *IEEE Computer*, 33(12):67–74, 2000.

[11] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache–oblivious algorithms. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science*, pages 285–297, 1999.

[12] G. Graefe and P.-Å. Larson. B-tree indexes and cpu caches. In *Proceedings of the 17th International Conference on Data Engineering*, 2001.

[13] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *Proceedings of the 12th VLDB Conference*, pages 294–303, 1986.

[14] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the ACM SIGMOD Conference*, pages 49–60. ACM Press, 2002.

[15] V. Raman. *Interactive Query Processing*. PhD thesis, UC Berkeley, 2001.

[16] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In *Proceedings of the 25th VLDB Conference*, pages 78–89, 1999.

[17] J. Rao and K. A. Ross. Making b+–trees cache conscious in main memory. In *Proceedings of the ACM SIGMOD Conference*, pages 475–486, 2000.

Counting the Number of Flows for Continuous Monitoring [*]

Hyang-Ah Kim
School of Computer Science, Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213. USA
hakim+@cmu.edu

ABSTRACT

A network flow is a stream of packets with the same IP header patterns. In this paper, we address the problem of counting distinct flows seen on a high speed link. Because the packet arrival rate is high in today's networks, the flow counting program, which must peek at all incoming packets, should run fast to keep up with the wire speed. The demand for fast processing limits the number of memory references and requires fast memory that is small and expensive.

Bitmap-based algorithms proposed for the fast estimation of the number of active flows [8] have the problem of underestimation. We suggest a timestamp-vector based algorithm that exploits the fast estimation of bitmap-based algorithms without the underestimation problem. Instead of keeping all active flow information, our algorithm hashes flows to a timestamp-vector and estimates the number of flows based on the timestamp-vector. Compared to the bitmap-based algorithms, the timestamp-vector algorithm reduces the possibility of underestimation of the number of active flows. Experiments with an IP header trace suggest that the timestamp-vector algorithm reduces the error of the bitmap-based algorithm by 82%. The timestamp-vector needs more memory space than the bitmap of the same number of entries but the memory requirement is still low enough to implement the algorithm on a small, fast cache memory.

1. INTRODUCTION

Network traffic monitoring and measurement system is a promising application that could benefit from streaming database techniques. Network traffic monitoring requires processing a stream of fast arriving packets. The packet stream is not only fast arriving but also potentially unbounded in size so that it is not possible to store all the packets in the storage. Streaming database techniques such as efficient stream processing and the use of approximation, synopsis, and sampling could help a network monitoring system to process IP packets quickly and reduce the memory requirement to service queries on various statistics of network traffic.

Sprint has developed a network traffic measurement system, called *Continuous Monitoring System* [14], which stores re-

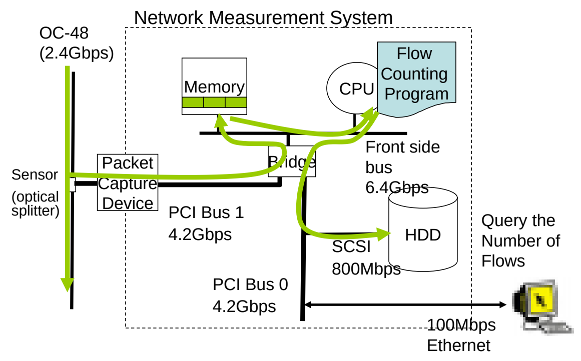


Figure 1: Continuous Monitoring System

cent packet traces from a high speed optical link into a circular buffer in local hard disk while performing online analysis to extract statistics on the traces. Figure 1 shows the architecture of Continuous Monitoring System, which is deployed on a high speed computer connected to an optical splitter of an OC-48 link. The network administrator may move the traces in the circular buffer to other place before it has been erased if detailed offline investigation on the snapshot is required.

The online analysis programs such as a flow counting program indicate when the administrator needs to look into the trace stored in the buffer. In addition, the result of the analysis programs works as a summary of the network traffic, which is much smaller than the original trace and can be kept permanently for future reference. The online analysis can be thought as a (static) continuous query over packet streams. As discussed in many papers on stream database systems [12, 4, 3], the online analysis needs to be fast enough to keep up with the data arrival rate, and uses small amount of storage.

In this paper, we design an algorithm for the Continuous Monitoring System's online analysis program that estimates the number of active flows. A *flow* is a stream of network packets having the same packet header patterns. For example, a flow can be thought of as a stream of packets that have the same source IP address, destination IP address, source port number, destination port number, and protocol field. [1] A *Flow identifier* is a combination of values in certain IP

[*]An extension of this work is published and presented in Globecom 2003 General Conference with the title of "Counting Network Flows in Real Time."

[1]The definition of a flow is user-configurable as long as the

header fields that distinguish one flow from another.

Counting the number of active flows is useful for DoS attack detection, port scan detection, and other network problem detection. A sudden increase in the number of active flows can be a symptom of DoS attacks or port scan if we define a flow with a combination of source address and source/destination port numbers. Sudden decrease in the number of flows may indicate that there are network routing problems or changes. Upon such a big change, network administrators take the traces in the hard disk of Continuous Monitoring System and investigate the problem.

There exist algorithms to estimate the number of active flows using bitmap data structures [8] but these algorithms tend to underestimate the actual number of flows. In this paper, we investigate why they underestimate the flow count, and then extend the algorithms with a timestamp-vector to address the problem. The timestamp-vector algorithm needs more memory than the bitmap-based algorithms. However, the timestamp-vector algorithm still requires less memory than the approach of storing per-flow states while estimating the number of flows more accurately. In our experiment with a real IP packet trace, the timestamp-vector algorithm reduces the estimation error by 82% compared to a bitmap-based algorithm using the same amount of memory.

The remainder of the paper is organized as follows. Section 2 presents the previous related work. In Section 3, we present applications that make use of flow counting or may benefit from the accurate and fast flow counting algorithms. In Section 4, we explain the bitmap-based algorithm in detail and describe the problem of underestimation in online analysis of the number of flows. Section 5 presents the timestamp-vector algorithm we are proposing and Section 6 shows experimental results. We conclude the paper with discussing the issues requiring further study and the way to improve the accuracy of the algorithm.

2. RELATED WORK

General purpose traffic measurement systems, such as Cisco's Netflow [6], report per-flow records for very fine grained flows. The systems can be used for flow counting, but are not optimized for the purpose. These systems need a large amount of memory to keep per-flow record.

Ideally, the states should be in high speed SRAM (which is expensive and relatively small) to gather statistics on quickly arriving packets. However, Netflow stores the per-flow state in slower speed DRAM, which slows down the whole packet processing because there are so many flows. Cisco recommends the use of sampling at speeds above OC-3: Netflow samples one packets out of x packets and only the sampled packets result in updates to the flow cache that keeps the per-flow state. Unfortunately, sampling packets and then estimating the number of flows from the set of sampled packets results in extremely poor accuracy. This is because uniform random sampling tends to produce more samples of large flows, thereby biasing any simple estimator that uses the number of flows in the sample.

There are two main approaches to counting distinct records or items: sampling based counting [5, 11] and synopsis based counting [9, 16]. Sampling-based approaches are known to be inaccurate and substantial lower bounds on the required sample size are shown by Charikar et al. [5]. The most widely applicable synopsis method is *probabilistic counting* introduced by Flajolet and Martin [9]. Whang et al. [16] proposed a probabilistic counting algorithm using bitmap.

Recently, Estan et al. [8] applied the linear-time probabilistic counting for the flow counting problem and built a set of algorithms. All those algorithms count the number of flows in a discrete window or a certain measurement interval. In Section 4, we look into the problem caused by the discrete measurement interval.

We expect the use of sliding windows instead of discrete measurement intervals can mitigate the problem. Cormode et al. [7] proposed an algorithm to compute Hamming Norms of data streams and the Hamming Norm of a single stream is the number of distinct values in the stream. The algorithm considers the insertion and the deletion of values but it is not clear how to use the deletion operation to discard the expired flows. Gibbons et al. [10] addressed the counting problem for sliding windows. However, for the practical use of the algorithm, we need modify the algorithm greatly in order to reduce the large time overhead for the linked-list maintenance and the space overhead to store an additional hash table (hashed by the flow identifiers) and multiple instances of Wave for higher accuracy.

The timestamp-vector-based algorithm we are proposing in this paper is similar to the one presented in Estan et al. [8] but addresses the problem of discrete measurement intervals by using timestamp-vector.

3. APPLICATIONS OF FLOW COUNTING

In this section, we present the applications that can benefit from accurate flow count estimation algorithms.

- **Detecting port scans** Intrusion detection systems warn of port scans when a source opens too many connections within a given time. The widely deployed Snort IDS [13] uses the naive approach of storing a record for each active connection (flow). Since the number of sources can be high, it requires large amount of memory. The scalability is particularly important in the context of the recent race to provide intrusion detection systems to keep up with wire speed [1].
- **Detecting denial of service attacks** To differentiate between legitimate traffic and an attack, we can use the fact that DoS tools use fake source addresses chosen at random. If for each of the suspected victims we count the number of sources of packets that come from some networks known to be sparsely populated,

definition is based on the distinct IP header patterns. In this paper, we use source and destination IP addresses and ports along with protocol number to define a flow and a flow is terminated if there is no activity for a certain duration. However, the algorithm should work with different definitions of a flow such as packets originated from specific destination IP addresses or belonging to a specific applications

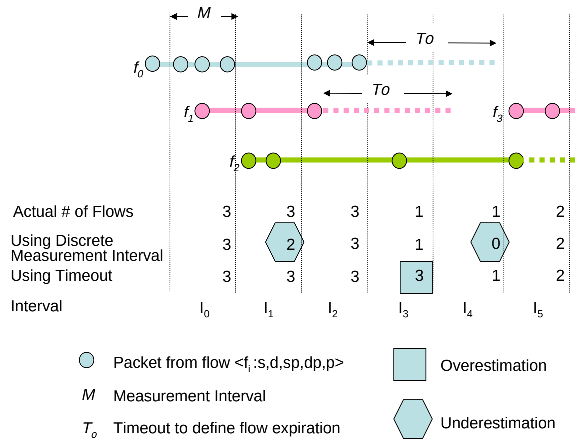


Figure 2: Measurement Method and Estimation Error

a large count is a strong indication that a DoS attack is in progress.

- **General measurement** Often counting the number of distinct values in given header fields can provide useful data. For example, one could measure the number of sources using a protocol version or variant to get an accurate image of protocol deployment. Another example is dimensioning the various caches in routers, which benefits from prior measurements of typical workload. The caches include packet classification caches, multicast route caches for Source-Group state, and ARP caches.
- **Packet scheduling** While there are scheduling algorithms that compute the fair share of the available bandwidth without using per-flow state (ex. CSFQ [15]), they require explicit cooperation between edge and core routers. Being able to count the number of distinct flows that have packets in the queue of a router might allow the router to estimate the fair share without outside help.

4. PROBLEM OF DISCRETE MEASUREMENT INTERVAL

Bitmap-based algorithms accurately estimate the number of flows seen in a fixed measurement interval, M [8]. The main idea of the algorithm is to update a bitmap structure during a measurement interval and, at the end of every measurement interval, count the number of updated entries in the bitmap to estimate the number of flows. Then, he bitmap is flushed for the next measurement interval. To update the bitmap structure, the algorithm uses a hash function[2] on the flow identifiers to map each flow to a bit of the bitmap. All the bits in the bitmap are set to zero at the beginning of a measurement interval. Whenever a packet comes in, the flow identifier is calculated and the bit corresponding to the identifier is set to 1. Assuming b is the size of the bitmap and z is the number of zero bits at the end of the measurement interval, the estimated number of active flows, $\hat{n}$, is $b \ln\left(\frac{b}{z}\right)$. To provide a better accuracy, they propose variations of the algorithm but all of them are designed to estimate the numbers in a fixed and discrete interval.

The disadvantage of using discrete intervals (or discrete windows) is the possibility of underestimation. Figure 2 depicts the problem. Even though flow f_0 is still active, the algorithm does not know the existence of f_0 in the interval I_1 because there is no packet belonging to f_0 in I_1. Therefore, the estimation after I_1 is lower than the total number of actually existing flows.

Figure 3 shows how much we underestimate if we use methods based on a fixed and discrete measurement interval. We draw the graph of the actual number of flows by recording the start and end times of all the flows in a 10 minute-long IP packet trace and querying the number of active flows at the moment every minute. Then, we split the trace into discrete M second-long pieces and count the total number of flows observed in each piece to emulate the algorithms using discrete measurement intervals.

If the measurement interval M is short, the underestimation becomes large because we miss a lot of flows that are still active but happen to send no packet during the interval. If we employ larger M, the possibility of underestimation decreases but the algorithm returns the accumulated number of flows for M seconds and the result is different from what we want.[3] Moreover, the algorithm with large M cannot quickly respond to a sudden change in the number of flows because it should wait until the measurement interval M finishes. Considering the purpose of online monitoring, we cannot tolerate such large delay.

One approach to address the underestimation problem is to employ the concept of timeout and keep the information of the flows that have sent packets in last T_o seconds. By the way, we should be careful when we choose the value for T_o since timeout introduces the possibility of overestimation as we see in the interval I_4 in Figure 2.

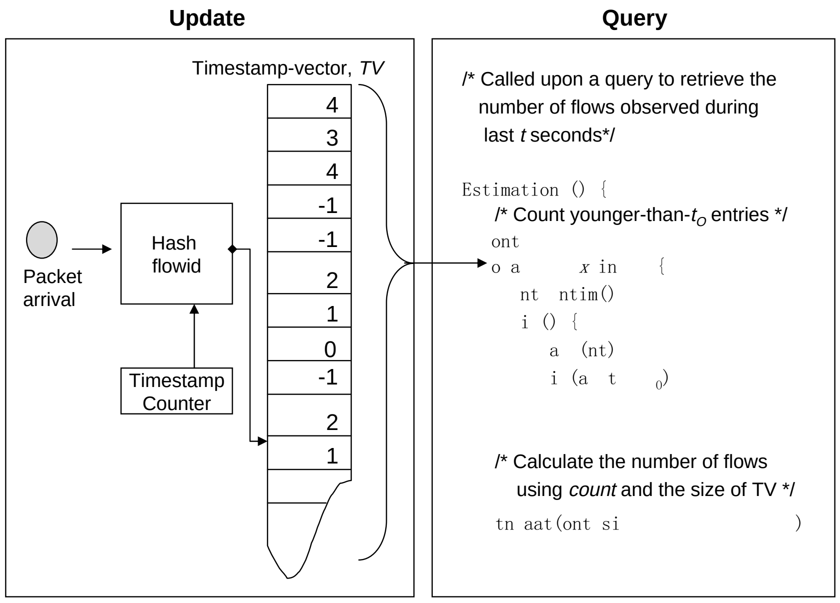
5. TIMESTAMP-VECTOR ALGORITHM

5.1 Algorithm

Instead of a bitmap, we use a timestamp-vector that keeps the timestamp of a packet arrival. Let's start with 32bit long timestamp. The update of the vector and estimation is done in the same way as in the bitmap-based algorithms. The difference is that, when a packet comes in, we update the value of the timestamp-vector entry with the timestamp of the packet arrival. When we need to count the number of entries to estimate the number of flows, we compare the values stored in the timestamp-vector against the current time and only count the entries updated in last T_o seconds.

[2]Estan et al. assume in their analysis that the hash function distributes the flows randomly. In an adverse setting, the attacker who knows the hash function could produce flow identifiers that produce excessive collisions thus evading detection. This is not possible if we use a random seed for the hash function. In our work and analysis, we also assume such a hash function that uniformly distribute flow identifiers.

[3]The point in the graph of $M = 60sec$ corresponds to the total number of flows observed in the last 60second interval. Some flows might be terminated early in the interval, but they are reported at the end of the interval.



```
/* Called upon a query to retrieve the
   number of flows observed during
   last t seconds*/

Estimation () {
    /* Count younger-than-t_o entries */
    ont
    o a       x in     {
        nt  ntim()
        i () {
            a  (nt)
            i (a  t     0)

    /* Calculate the number of flows
       using count and the size of TV */
    tn aat(ont si                     )
```

Figure 4: Flow Counting with timestamp-vector

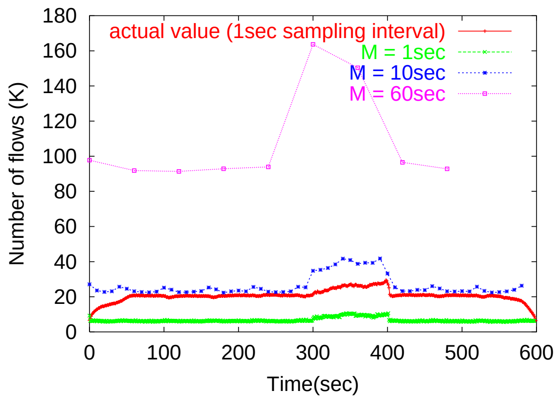


Figure 3: Limitations of flow counting with discrete measurement intervals

Given c is the number of entries updated in last t seconds, the estimated number of flows, $\hat{n} = b \ln \frac{b}{b-c}$. Figure 4 depicts the operations for the timestamp-vector update and the query on the number of flows. By selecting an appropriate T_o, we can avoid the underestimation caused by short measurement intervals. Also, the timestamp-vector is not flushed out at the end of the measurement interval and thus, we could query the number of flows active for an arbitrary period.

5.2 Implementation Issues

Compared to the bitmap-based algorithms, the timestamp-vector algorithm requires more memory. If we employ a 32bit timestamp, the timestamp-vector will be 32 times larger than a bitmap of the same number of entries. However, if we use smaller counters instead of storing the 32bit timestamp and round the counter when the counter reaches the maximum, we can reduce the size of the timestamp-vector. For example, if we need to take into account the flows observed in less than t seconds, only $\log_2 t + 1$ bits for each entry is sufficient: $\log_2 t$ bits for counter value, and 1 bit to indicate the entry is never touched.

6. EVALUATION

6.1 Experimental Setup

The purpose of our experiment is to compare the accuracy of the algorithms, and measure the time for processing. We have implemented the bitmap-based flow counting algorithm and the timestamp-vector algorithm. Then, we compare the error against the result of the program counting the actual number of flows.

We have run the implemented algorithms on a Red Hat Linux 7.1 in Pentium III 900MHz machine with a 256KB L2 cache, and 1GB main memory. We simulate Continuous Monitoring System by reading an IP packet header trace from a hard disk connected with SCSI. Note that our measurement omits the overhead caused by data transfer from the packet capture card to the main memory in Figure 1, and the total execution time in real Continuous Monitoring System would be slightly longer than the value we have from the experiment.

For the experiments, we use a 10 minute-long IP packet header trace collected from an OC-48(2.5Gbps) link in the *Internet2* network [2]. During the trace collection, the average traffic bandwidth was 800Mbps and the average packet transfer rate was 140Kpkts/sec. The trace contains 885K flows when we use 64sec as the idle time value to decide the end of each flow. Table 1 gives a summary description of the trace. Throughout the experiment, we assume 64sec idle time to terminate a flow.

6.2 Configuring Timeout T_o

Name	IPLS-CLEV-20020814-100000
Duration	2002.08.14 10:00 10:10am EST
Average Traffic Volume	800Mbps
Average Packet Rates	140Kpkts/sec
Flows	884537

Table 1: OC48 Trace used for the experiment

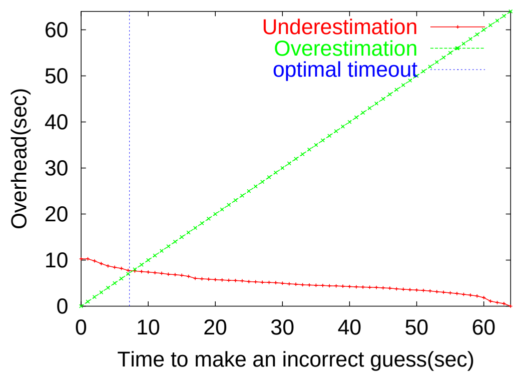


Figure 5: Normalized Underestimation and Overestimation Overhead

In this section, we examine what is the appropriate value for T_o to minimize the possibility of underestimation and overestimation mentioned in Section 4. Assuming we query the number of flows every second, we try to choose appropriate T_o. We collected the inter-packet-arrival time in flows and computed the overhead defined by the minutes when we can make a wrong guess on a flow expiration. Since we use 64sec idle time to decide the real flow termination, T_o should be less than 64sec to avoid large overhead caused by overestimation. Figure 5 shows the overhead we obtain by analyzing the trace. The two overhead graphs cross when $T_o = 7$. That means we can get better estimation for the actual number of flows, when we query with $T_o = 7$ every second.

6.3 Accuracy

Now we compare the accuracy we can achieve with the timestamp-vector algorithm to the bitmap-based algorithm. For accuracy measure, we use the distance of two functions $f(x)$ and $g(x)$ defined by $\|f-g\| = \sqrt{\sum_x (f(x) - g(x))^2}$. We compute the distance of the result of each algorithm to the actual number of flows sampled every second to retrieve the number of currently active flows. We make a query every second and when we use the timestamp-vector algorithm, we use 7 for the value of T_o. We run the experiment with three different hash functions (CRC based hash functions) and average the distances. For the bitmap-based algorithm, we use the same amount of memory for maps (5KBytes).[4] With the timestamp-vector algorithm, we reduced the error by 82%. Figure 6 depicts the results of the algorithms and the actual number of flows graph. To see the possibility

[4]We chose the number of entries in the timestamp-vector based on the analysis of Estan et al. [8]. The timestamp-vector size guarantees that the algorithm estimates the total number of *observed* flows during T_o with 1for bitmap-based algorithm.

Algorithm	Distance
Bitmap based ($M = 1$)	353823.33
Bitmap based ($M = 7$)	96076.45
timestamp-vector based ($T_o = 7$)	65913.18

Table 2: Distance to the real number of flows

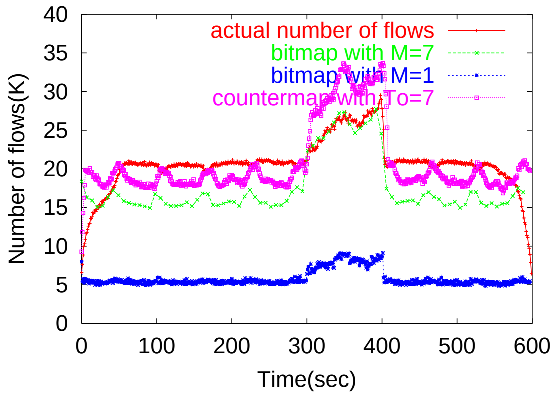


Figure 6: Comparison of two algorithms

that we get better estimation with longer measurement interval for bitmap based algorithm, we also present the result measured with 7sec measurement interval for bitmap based algorithm. The result with 7sec interval is still far from the actual number of flows and, moreover, it is not able to report the number of flows more often than 7 seconds.

6.4 Time Requirement

We measure the time each algorithm takes to process the 10minute long trace. Both the bitmap-based algorithm and the timestamp-vector algorithm take about 40seconds to process the trace. The fast execution is due to the simple algorithm to update the data structures and the small requirement for the memory. 4KBytes bitmaps or timestamp-vectors fit into a small high-performance memory. When we store a record for each active flow as Snort does, we need to allocate at least about 2.8Mbytes memory to store the flow information in the trace we are using. Since the memory is dynamically allocated and we cannot keep all the records in a small cache, the processing requires more time than when we use bitmap or timestamp-vector.

7. FUTURE WORK

- **Adaptive T_o**

  It is the most challenging to predict the end of a flow when we implement the online flow counting algorithm. Since we do not know the correct end of a flow only with the information in the IP packet header, we had to rely on the timer. A large timeout value introduces great amount of overestimation overhead. Especially, if there are many short flows, the overestimation overhead is much greater than the benefit we will have by eliminating the possibility of underestimation. If there exist many short flows, we had better choose small T_o for our timestamp-vector algorithm and if there are many long flows with large packet inter-arrival intervals, it will be better to use larger T_o. In this work, we chose T_o suitable for the trace we used but if the flow duration distribution or the characteristics of the inter-packet-arrival time are different from those observed in our trace, our T_o will not be appropriate any longer. Moreover, with advent of new Internet applications, we expect the traffic pattern changes over time. To address this problem, we need to make the algorithm and its application adaptively adjust T_o depending on the previous network observations.

- **Protocol Specific Flow Termination**

  In practice, protocol specific termination information such as TCP FIN field is used in conjunction with the timeout method to find out the exact flow termination [6, 13] but, in this work, we only considered a timeout method to indicate flow termination. The reliance on the timeout leads to overestimation as we saw in Section 4. We need to extend our algorithm to exploit the protocol specific information to reduce the chance of overestimation. For example, we can add an extra bit for each entry in the timestamp-vector to mark if we have observed a protocol specific termination packet of the flow mapped to the entry. When we query the number of currently active flows, we only need consider the entries that are not expired and whose termination indication bit are not set. Considering TCP is a dominant protocol in the current Internet and TCP uses the explicit flow termination packets we can exploit, the extension of the algorithm will greatly reduce the overestimation at the cost of time and space.

8. CONCLUSION

In this paper, we proposed an algorithm to estimate the number of distinct flows seen on a high speed link. The packet arrival rate is high in today's networks and, therefore, the flow counting algorithm should run fast to keep up with the wire speed. The demand for fast processing requires to limit the number of memory references and exploit fast memory that is usually small and expensive.

There exist algorithms estimating the number of distinct flows using a small bitmap and it requires small amount of memory and a few number of memory references. However, the algorithm tends to underestimate the number of active flows when deployed for online analysis. We extended the bitmap-based algorithms using a timestamp-vector instead of a bitmap and avoid the case underestimating the number of flows. By reducing underestimation, the timestamp-vector algorithm improved the estimation accuracy greatly while still using a small amount of memory compared to the techniques to store per-flow states.

9. REFERENCES

[1] Cisco offers wire-speed intrusion detection, December 2000. http://www.nwfusion.com/reviews/2000/1218rev2.html.

[2] NLANR MNA team. NLANR MOAT: PMA Trace Archive, September 2002. http://pma.nlanr.net/Traces/long/ipls1.html.

[3] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3), September 2001.

[4] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams - a new class of data management applications. In *the 28th International Conference on Very Large Data Bases*, August 2002.

[5] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. Towards estimation error guarantees for distinct values. In *the 9th Symposium on Principles of Database Systems*, 2002.

[6] Cisco. Netflow. http://www.cisco.com/warp/public/732/Tech/netflow.

[7] G. Cormode, M. Datar, P. Indyk, and S. Muthukrishnan. Comparing data streams using hammming norms (how to zero in). In *the 28th VLDB Conference*, August 2002.

[8] C. Estan, G. Varghese, and M. Fisk. Counting the number of active flows on a high speed link. Technical Report CS2002-0705, UCSD, May 2002.

[9] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for database applications. *Journal of Computer and System Sciences*, 31(2):182–209, October 1985.

[10] P. B. Gibbons and S. Tirthapura. Distributed streams algorithms for sliding windows. In *the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, August 2002.

[11] P. Haas, J. Naughton, S. Seshadri, and L. Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *the 21st VLDB Conference*, 1995.

[12] S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *ACM SIGMOD International Conference on Management of Data*, June 2002.

[13] M. Roesch. Snort - lightweight intrusion detection for networks. In *the 13th Systems Administration Conference, USENIX*, 1999.

[14] Sprint. Ip monitoring project, 2002. http://ipmon.sprintlabs.com.

[15] I. Stoica, S. Shenker, and H. Zhang. Core-stateless fiar queueing: A scalable architevture to approximate fair bandwidth allocations in high speed networks. In *the ACM SICOMM*, September 1998.

[16] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems*, 15(2):208–229, 1993.

Statistical Analysis of Histograms

Deepayan Chakrabarti
Center for Automated Learning and Discovery
School of Computer Science
Carnegie Mellon University
deepay@cs.cmu.edu

ABSTRACT

In modern databases, histograms are the tools of choice for storing statistics regarding the distribution of data. They are heavily used by the query optimizer, and their accuracy is critical to the performance of the database system. But, the information stored in histograms is usually used in a purely ad-hoc manner; to our knowledge, there has been no proper statistical study of histograms, and how to use them in the database context. Here, we attempt to do that, and find statistically-correct solutions to several problems. These include showing how histograms can yield expected sizes of range queries and joins, and how to update histograms using feedback in a statistically sound manner. We also propose to attach an extra value (called *variance*) to each bucket in a histogram, along with the height of that bucket. We show how this is extremely useful for various other operations.

1. INTRODUCTION

[1] Histograms are the heart of most database catalogs. They are used to maintain approximate information about the data stored in the database. While considering a particular query plan, current optimizers use heuristics based on histogram data to estimate the sizes of intermediate and final result sets generated by that plan. The goodness of the final generated query plan depends to a great extent on the accuracy of these numbers. A statistical study of the problem of estimating these numbers would give a lot of insight into the problem. We would also understand the assumptions that the heuristics inherently make, and the conditions when such assumptions are valid.

There has been some recent work on updating histograms using feedback of results obtained during query execution. The general procedure of building the feedback loop is described in [7], and Aboulnaga and Chaudhuri [1] describe heuristics for updating histograms based on this data. We explore this idea in detail, and find the assumptions under which these heuristics operate. This is useful in situations where the assumptions are not expected to hold; in such cases, the user could be pre-warned.

In addition, we propose to use statistical techniques to find measures of confidence on the estimates of result sizes. This might be important if the optimizer gives the user an estimate of the expected time required for query execution. A higher confidence level on the result sizes would translate into higher confidence regarding the optimality of the query plan. Another major utility of confidence values is in updating histograms. We will describe our technique for doing this later in the paper.

The rest of the paper is organized as follows. Section 2 describes related work in the area of using/updating histograms. In Section 3, we present our model of the histogram, and show why this is a reasonable model. Based on this model, we develop theoretical results for using/updating histograms in Section 4. This is followed by our conclusions in Section 5.

2. RELATED WORK

There has been a lot of work on maintaining statistics about the data stored in a database. One nice model was described by Chen and Roussopoulos [3], where the authors use a linear combination of *model functions* to approximate the true distribution of data. The coefficients of the linear combination are adjusted using query feedback. But, in this case, figuring out the correct family of model functions might be hard; not all datasets will easily fit the same model family.

In general, histograms are the method of choice for storing statistics about the data in today's databases. Hence, we focus on histograms for our work. Poosala et al [6] give an excellent breakdown of the existing histograms into several categories. The categories most relevant to our work are:

- *Equi-Width* histograms: The maximum minus minimum value in each bucket of the histogram is approximately the same for all buckets.
- *Equi-Depth* histograms: The frequency of points in each bucket is approximately the same for each bucket.

Our model is also general enough to handle cases where the histogram buckets are neither equi-width nor equi-depth.

In [5], Poosala and Ioannidis attempt to estimate join sizes using two different techniques. The first is to build a multi-dimensional histogram, and the second uses Singular Value Decomposition. But building a multi-dimensional histogram is costly, and SVD requires sampling followed by operations on large matrices. We attempt to find out how existing one-dimensional histograms can be used for join-size estimation, and this is a different problem.

[1]Submitted to ICDE 2004

Gibbons et al [4] describe a method of maintaining histograms by always keeping a backing sample. In our approach, we would like to use the intermediate results during query execution to maintain the histogram, avoiding the need for a backing sample.

Stillger et al [7] describe the LEO (LEarning Optimizer) used in DB2. This uses a feedback loop inside the database, where intermediate results are remembered for future queries. But in this case, the intermediate results are primarily just stored, and no attempts are made to update the histograms themselves.

One paper which describes work very related to ours is that of Aboulnaga and Chaudhuri [1], where the authors also try to update histograms using query results. But, theirs is a more ad-hoc (heuristic based) technique; we use statistics to tackle the problem. We show that under certain assumptions, the heuristics are actually validated by statistics.

3. MODELING THE PROBLEM

In previous sections, we have described the usefulness of histograms in current database systems, and made the case for a formal treatment of the problem. In this section, we start this process by building a model for the histograms. In the following discussion, we always try to present the assumptions being made.

We begin with one-dimensional histograms. For each bucket of the histogram, we typically store the range of values it encompasses, and the number of points falling within that range. For our analysis, the exact range is not so important as the *spread*, that is, the difference between the maximum and minimum values allowed in that bucket. Let us number all the buckets in a histogram from 1 to $MaxBuckets$. We represent bucket number x with the symbol bx. The width of this bucket is represented by w_{bx}, and the height by h_{bx}. Thus, for an equi-width histogram, w_{bx} is approximately constant for all x, and for an equi-depth histogram, the h_{bx} values are almost constant. In some cases, we might want to talk of the height of some subset of a bucket; that is, the number of datapoints falling within a certain portion of one bucket. If this subset region of a bucket x is represented by $\mathcal{S}x$, then we represent the height within this region by $h_{\mathcal{S}x}$.

Now, the height of a bucket h_{bx} is the number of datapoints which fall within the range of bucket x. Thus, it is a random variable with a Binomial distribution, which can be fully described by specifying the total number of datapoints N and the probability px of a datapoint falling in bucket x. Thus,

$$h_{bx} \sim Bin(N, px) \tag{1}$$

Normally, we will not always have exact values for N or px. But if N is large (which is true in general), the Binomial distribution can be well approximated by a Normal distribution, with mean H_{bx} and standard deviation σ_{bx}.

$$h_{bx} \sim \mathcal{N}(H_{bx}, \sigma_{bx}) \tag{2}$$

The disadvantage of using this equation is that it allows h_{bx} to be a continuous variable, whereas we know that h_{bx} can only have discrete (integral) values. The advantage is that using this equation makes the following analysis much easier, given that the Normal distribution is very friendly to analytical studies. For large N, there should be very little negative effects of making this assumption. Hence, from now on, we consider h_{bx} to be normally distributed. All the symbols described above are concisely shown in Table 1.

With this formulation, the height of a bucket h_{bx} has an expected value of H_{bx} and a variance of σ^2_{bx}. This information can be easily maintained in the histogram, by storing these two values for each bucket. In current systems, no measure of variance is kept; this is equivalent to storing only the value of H_{bx}.

Whenever the histogram is completely recomputed by a complete pass over the data, the variance in h_{bx} is zero, for all buckets x; that is, we have the maximum possible confidence in the value of h_{bx}. But over time, this is changed due to inserts and deletes in the database (updates can be viewed as a combination of insert and delete). Updating the corresponding histogram bucket for every insert/delete would slow down the database a lot; so keeping the histogram up-to-date does not seem possible. Thus, the variance of h_{bx} increases over time, and our confidence in the histogram decreases.

To model this, we propose that insertions and deletions occur according to a Poisson distribution. This effectively assumes that the expected number of inserts/deletes on every bucket increases in proportion to the length of time since the histogram was completely computed. Using $Nadd_{bx}(t)$ and $Ndel_{bx}(t)$ to denote the number of insertions and deletions respectively on bucket x over time interval t,

$$\begin{aligned} Nadd_{bx}(t) &\sim Pois(\lambda^{add}_{bx}.t) && (3)\\ Ndel_{bx}(t) &\sim Pois(\lambda^{del}_{bx}.t) && (4)\\ & && (5) \end{aligned}$$

Here, λ^{add}_{bx} and λ^{del}_{bx} are the proportionality constants for additions and deletions respectively on bucket x. We shall use λ_{bx} when the discussion refers to both insertions and deletions.

For both insertions and deletions, the proportionality constant could be modeled in at least the two following ways:

- It could be the same for all buckets. This assumes that the probability of additions/deletions is uniformly distributed over the entire domain of the data.

$$\lambda_{b1} = \lambda_{b2} = \ldots = \lambda_{b-MaxBuckets} \tag{6}$$

- It could be proportional to the height of the corresponding bucket. This assumes that the probability of insertions/deletions into a bucket is proportional to the number of elements already in the bucket. So the buckets with the most elements would see the maximum activity.

$$\lambda_{bx} \propto h_{bx} \tag{7}$$

The correct way to model this proportionality constant would depend on the properties of the database.

4. THEORETICAL RESULTS

In the previous section, we described the variables in the problem, and gave methods to model them. In this section, we shall use those models to find statistical answers to questions about how to use histograms.

Symbol	Meaning
bx	Bucket number x
h_{bx}	Actual height of bucket number x
$h_{\mathcal{S}x}$	Height in a subset $\mathcal{S}x$ of bucket x
H_{bx}	Mean/Expected height of bucket x
σ_{bx}	Standard deviation of h_{bx}
w_{bx}	Width of bucket number x
N	Total number of datapoints
px	Probability of a datapoint falling in bucket x
$MaxBuckets$	Total number of buckets in a histogram

Table 1: ***Table of Symbols***

The questions we wish to ask (and answer) are:

- Estimating the result size of a range query
- Estimating the result size of a join, where one-dimensional histograms exist on the joining attribute in both tables
- Updating a one-dimensional histogram, based on feedback on the exact result size of a range query
- Quantifying the decay in quality of the histogram over time, due to insertions and deletions

We now look at each of these problems in turn.

4.1 Estimating the result size of a range query

If there is a one-dimensional histogram on the attribute of the range query, then it can provide information about the result size. Let us denote the range by $\mathcal{R}$. Let the intersection of $\mathcal{R}$ with bucket x be denoted by $\mathcal{R}_{bx}$. Thus,

$$\mathcal{R} = \sum_{i=1}^{MaxBuckets} \mathcal{R}_{bi}$$

To proceed further, we need the following two assumptions:

1. The heights of the buckets in the histogram are mutually independent. This means that knowing the height of one bucket gives us no extra information about the height of any other bucket in the histogram.
2. *Inside* a bucket, the datapoints are spread uniformly over the width of the bucket. Since we do not have any information about the distribution of datapoints inside a bucket, this appears to be a fair assumption. The assumption should work better when the width of each bucket is relatively small.

These assumptions allow us to approximate the probability distribution of the datapoints within the the entire domain, using just the information provided by the histogram.

For bucket x, the subset which is included in the range query is $\mathcal{R}_{bx}$. Let us call the height of this subset (that is, the number of datapoints which fall inside this subset of the box) $h_{\mathcal{R}_{bx}}$, and its width $w_{\mathcal{R}_{bx}}$.

The number of points which fall within $\mathcal{R}$ is a random variable; let us call it $h_{\mathcal{R}}$. Thus,

$$h_{\mathcal{R}} = \sum_{i=1}^{MaxBuckets} h_{\mathcal{R}_{bi}} \tag{8}$$

Now, our estimate of the result of the range query is just the expected number of points that fall within $\mathcal{R}$, which is just the expected value of $h_{\mathcal{R}}$.

$$\begin{aligned} E[h_{\mathcal{R}}] &= E\left[\sum_{i=1}^{MaxBuckets} h_{\mathcal{R}_{bi}}\right] & (9)\\ &= \sum_{i=1}^{MaxBuckets} E[h_{\mathcal{R}_{bi}}] & (10)\end{aligned}$$

Now, **given** the value of h_{bx}, the height of the subset is distributed as:

$$h_{\mathcal{R}_{bx}} \sim Bin(h_{bx}, \frac{w_{\mathcal{R}_{bx}}}{w_{bx}}) \tag{11}$$

So:

$$\begin{aligned} E[h_{\mathcal{R}_{bx}}] &= E[E[h_{\mathcal{R}_{bx}} \mid h_{bx}]] & (12)\\ &= E\left[h_{bx} \cdot \left(\frac{w_{\mathcal{R}_{bx}}}{w_{bx}}\right)\right] & (13)\\ &= H_{bx} \cdot \left(\frac{w_{\mathcal{R}_{bx}}}{w_{bx}}\right) & (14)\end{aligned}$$

This is essentially saying that expected height of a subset of a bucket is the expected height of the entire bucket, multiplied by the fraction of the bucket included in the subset.

$$\begin{aligned} Var(h_{\mathcal{R}_{bx}}) &= E[Var(h_{\mathcal{R}_{bx}} \mid h_{bx})] \\ &\quad + Var(E[h_{\mathcal{R}_{bx}} \mid h_{bx}]) & (15)\\ &= E\left[h_{bx} \cdot \left(\frac{w_{\mathcal{R}_{bx}}}{w_{bx}}\right) \cdot \left(1 - \frac{w_{\mathcal{R}_{bx}}}{w_{bx}}\right)\right] \\ &\quad + Var\left(h_{bx} \cdot \left(\frac{w_{\mathcal{R}_{bx}}}{w_{bx}}\right)\right) & (16)\\ &= H_{bx} \cdot \left(\frac{w_{\mathcal{R}_{bx}}}{w_{bx}}\right) \cdot \left(1 - \frac{w_{\mathcal{R}_{bx}}}{w_{bx}}\right) \\ &\quad + \left(\sigma_{bx} \cdot \left(\frac{w_{\mathcal{R}_{bx}}}{w_{bx}}\right)\right)^2 & (17)\end{aligned}$$

where H_{bx} is the mean of random variable h_{bx}, that is, the expected value of the height of bucket x. This gives the variance on the estimated result size.

Using Equation 14 in Equation 10,

$$E[h_{\mathcal{R}}] = \sum_{i=1}^{MaxBuckets} H_{bi} \cdot \left(\frac{w_{\mathcal{R}_{bi}}}{w_{bi}}\right) \tag{18}$$

This is the estimate for the result size of the range query. We note that only assumption 2 is used to derive this result.

We also want to find the variance of this result. The variance would give us some idea of the confidence we can have in our estimate. Now,

$$\begin{aligned} Var(h_{\mathcal{R}}) &= \sum_{i=1}^{MaxBuckets} Var(h_{\mathcal{R}_{bi}}) & (19)\\ &= \sum_{i=1}^{MaxBuckets} H_{bi} \cdot \left(\frac{w_{\mathcal{R}_{bi}}}{w_{bi}}\right) \cdot \left(1 - \frac{w_{\mathcal{R}_{bi}}}{w_{bi}}\right) \\ &\quad + \left(\sigma_{bx} \cdot \left(\frac{w_{\mathcal{R}_{bx}}}{w_{bx}}\right)\right)^2 & (20)\end{aligned}$$

We used both assumptions to find the variance.

Thus, Equations 18 and 20 give the expectation and variance of the result size of the range query. This leads to the issue of the method of interpreting the variance, and presenting it to the user in an understandable way. The best way to do this is to provide confidence intervals; that is, to say that the result lies within a certain range of values with, say, 95% probability.

To be able to give precise confidence intervals, it is necessary to understand the distribution of $h_{\mathcal{R}}$ in detail. We can consider two cases:

- When $\mathcal{R}$ is aligned with bucket boundaries (that is, there is no bucket which is partially inside $\mathcal{R}$; either it is completely inside, or completely outside): here, $h_{\mathcal{R}}$ is the sum of several normal variables, and hence is itself normal.
- When $\mathcal{R}$ is **not** aligned with bucket boundaries: here, we could find no pre-existing distribution for $h_{\mathcal{R}}$. But since it is the sum of several independent random variables (see equation 8), we can use the Central Limit Theorem (Casella and Berger [2]) to approximate it with a normal distribution. The approximation gets better as the number of buckets contained in $\mathcal{R}$ increases, which happens when there are lots of buckets in the histogram (a fine-grained histogram) and when $\mathcal{R}$ is relatively large.

In both these cases, we end up considering $h_{\mathcal{R}}$ to be a normal variable. There exist proper confidence bounds for normal variables, which can be found in standard Statistics textbooks [2]. To give an example, the *true* result of the range query lies between $(E[h_{\mathcal{R}}] - 1.64 * Var(h_{\mathcal{R}}))$ and $(E[h_{\mathcal{R}}] + 1.64 * Var(h_{\mathcal{R}}))$ with 90% probability. We can find such confidence bounds depending on the probability value that the user gives the system.

One other interesting fact is that the variance of the result, as shown in Equation 20 depends on the expected height of buckets H_{bx} and the fraction of the range query falling inside a bucket, but **not** on the actual width of the bucket itself. This tends to support the view that it is better to have some wide buckets, if that allows the heights of all the buckets to be kept low. On the other hand, this reasoning is highly dependent on the assumptions made previously; if we have wider buckets, there is less likelihood of the uniformity assumption holding. Thus, we would expect the "correct" combination of heights vs widths in a histogram to be somewhere in between equi-depth and equi-height. This is exactly what is observed by Poosala et al [6].

4.2 Estimating the result size of a join, where one-dimensional histograms exist on the joining attribute in both tables

To statistically analyze the join of two tables over one attribute, we assume the following:

1. The joining attribute is countable (that is, it cannot be a floating point number).
2. The distributions of attribute values in the two tables are independent.
3. Datapoints are uniformly distributed within each bucket.

Assumption 1 is needed because with uncountably infinite sets, the probability of having two attribute values being the same (which is needed for a join) becomes zero. Assumption 2 is a decidedly weak assumption, and possibly far from the truth. But, it appears to be the only fair assumption in the absence of extra data. Assumption 3 is again only a fair assumption in the absence of extra data.

Suppose we are joining tables A and B. Let the domain of the joining attribute be $\mathcal{D}$. The number of datapoints having attribute value i ($i \in \mathcal{D}$) is a random variable, which we represent by $ht_A(i)$ and $ht_B(i)$ for tables A and B respectively. We represent the result of joining A and B by $Join(A, B)$. Then,

$$\begin{aligned} E[Join(A,B)] &= E\left[\sum_{i\in\mathcal{D}} ht_A(i).ht_B(i)\right] && (21)\\ &= \sum_{i\in\mathcal{D}} E[ht_A(i).ht_B(i)] && (22)\\ &= \sum_{i\in\mathcal{D}} E[ht_A(i)].E[ht_B(i)] && (23)\end{aligned}$$

Now, from assumption 3,

$$ht_A(i) \sim Bin\left(height_A(bucket(i)), width_A(bucket(i))\right) \tag{24}$$

where $bucket(i)$ is the bucket corresponding to attribute value i for table A, and $height_A(bucket(i))$ and $width_A(bucket(i))$ are the height and width of that corresponding bucket. The notation is similar for table B. Thus,

$$E[ht_A(i)] = \frac{height_A(bucket(i))}{width_A(bucket(i))} \tag{25}$$

Using Equation 25 in Equation 23, we get:

$$\begin{aligned} &E[Join(A,B)] && (26)\\ &= \sum_{i\in\mathcal{D}} \frac{height_A(bucket(i))}{width_A(bucket(i))}\frac{height_B(bucket(i))}{width_B(bucket(i))} \end{aligned}$$

This gives the formula for estimating the cardinality of a join. We see that it iterates over every value in the domain $\mathcal{D}$ of the joining attribute. In practice, we need to iterate only over those values which occur in both A and B. It can be simplified in the case when the bucket boundaries are the same for both tables.

4.3 Updating a one-dimensional histogram, based on feedback on the exact result size of a range query

Let us denote the range of the query by $\mathcal{R}$. As in Section 4.1, we break up this range based on the buckets in the histogram.

$$\mathcal{R} = \sum_{i=1}^{MaxBuckets} \mathcal{R}_{bi}, \text{and}, h_{\mathcal{R}} = \sum_{i=1}^{MaxBuckets} h_{\mathcal{R}_{bi}}$$

Thus, our aim in this subsection is: **given** the *true* number of datapoints falling within the range $\mathcal{R}$, we wish to update the histogram using this information.

More precisely, suppose we are given that

$$\sum_{i=1}^{MaxBuckets} h_{\mathcal{R}_{bi}} = S \tag{27}$$

where S is the correct result of the range query. Then, what can we say about the the height of a bucket, given the result of the range query $(h_{bi} \mid \left(\sum_{i=1}^{MaxBuckets} h_{\mathcal{R}_{bi}} = S\right))$?

Consider a bucket x. Either it is completely contained inside $\mathcal{R}$ or it isn't. Let us consider the case when it is completely contained. Thus, $\mathcal{R}_{bx}$ is actually the entire bucket x.

Denoting the probability distribution of a variable by $p(.)$ and all the buckets (from 1 to $MaxBuckets$) by $AllBuckets$, we have:

$$p\left(h_{bx} \mid \sum_{i=1}^{MaxBuckets} h_{\mathcal{R}_{bi}} = S\right)$$

$$\propto \; p\left(\sum_{i=1}^{MaxBuckets} h_{\mathcal{R}_{bi}} = S \mid h_{bx}\right).p\left(h_{bx}\right) \tag{28}$$

$$\propto \; p\left(\sum_{i\in AllBuckets, i\neq x} h_{\mathcal{R}_{bi}} = S - h_{bx} \mid h_{bx}\right).p\left(h_{bx}\right) \tag{29}$$

$$\propto \; p\left(\sum_{i\in AllBuckets, i\neq x} h_{\mathcal{R}_{bi}} = S - h_{bx}\right).p\left(h_{bx}\right) \tag{30}$$

This follows from an application of Bayes Theorem, and holds for all buckets which are fully contained within the range query. For the last step, we are assuming that the heights of the buckets are independent of each other. This is the same as assumption 1 of Section 4.1. For all buckets completely outside the range query, the independence of buckets means that the range query gives no new information about them, and so, no update is required for them.

Now, from our model (Section 3), we have:

$$h_{bi} \sim \mathcal{N}\left(H_{bi}, \sigma_{bi}\right)$$

Since all the h_{bi}s are independent for the different buckets i, we have:

$$\sum_{i\in AllBuckets, i\neq x} h_{bi} \tag{31}$$

$$\sim \mathcal{N}\left(\sum_{i\in AllBuckets, i\neq x} H_{bi}, \sqrt{\sum_{i\in AllBuckets, i\neq x} \sigma_{bi}^2}\right)$$

Using Equation 32 in Equation 30, we see that $p\left(h_{bx} \mid \sum_{i=1}^{MaxBuckets} h_{\mathcal{R}_{bi}} = S\right)$ is the product of two normals, and is hence another normal with mean and variance given by:

$$Mean = E\left[h_{bi} \mid \sum_{i=1}^{MaxBuckets} h_{\mathcal{R}_{bi}} = S\right]$$

$$= \frac{H_{bx}.\left(\sum_{i\in AllBuckets, i\neq x} \sigma_{bi}^2\right)}{\sum_{i\in AllBuckets} \sigma_{bi}^2} + \frac{\left(S - \sum_{i\in AllBuckets, i\neq x} H_{bi}\right).\sigma_{bi}^2}{\sum_{i\in AllBuckets} \sigma_{bi}^2} \tag{32}$$

$$Variance = Var\left(h_{bi} \mid \sum_{i=1}^{MaxBuckets} h_{\mathcal{R}_{bi}} = S\right)$$

$$= \frac{\sigma_{bx}\sqrt{\sum_{i\in AllBuckets, i\neq x} \sigma_{bi}^2}}{\sqrt{\sum_{i\in AllBuckets} \sigma_{bi}^2}} \tag{33}$$

Thus, once the true result size of a range query is given, we can update the values H_{bx} and σ_{bx} for every bucket x which is completely within the range $\mathcal{R}$ of the range query. The important contribution of this set of equations is that it gives us a way to update the histogram given the true result of a range query, *using* our previous knowledge of the histogram.

There are a few insights from this set of equations:

- The equations assume that we already have some existing values for the mean and variance of the buckets, before we get the result of the range query. This will usually be true, since, these values will be generated when the histogram is first created.

- Aboulnaga and Chaudhuri [1] had considered this problem, but had not taken variances into account. When we have no idea of variances, we can assume that h_{bx} is actually binomially distributed, as shown in Equation 1. In such a scenario, the variance $\sigma_{bx}^2 \propto$ the mean H_{bx}. When we plug this into our equations for updating the histogram, the results obtained are *exactly the same* as the heuristics that Aboulnaga and Chaudhuri end up using.

There is also the case when some buckets are only partially contained in the range query. The ST-histogram uses the same heuristic in this case too. In spite of a lot of effort, we could not find any analytical expression to update those buckets. The main problem is that under those circumstances, the distribution we end up getting is not any well-known distribution; and it does not appear to be very amenable to analytical techniques. We still hope to work more on that particular problem. We also note that, even when there are some buckets which are only partially contained in the range query, the update mechanism described above works for all other buckets which are fully contained within the range query.

4.4 Quantifying the decay in quality of the histogram over time, due to insertions and deletions

Whenever we recompute the histogram by doing a complete pass over the database, our confidence in the histogram becomes maximum, and the variance in the heights of the buckets becomes zero. But, over time, due to insertions and deletions, the quality of the histogram degrades; and the variance in the heights of the buckets increases. In Section 3, we proposed a Poisson model of insertions/deletions. We now use this model to show how to change the variance in the heights of the buckets over time.

We first briefly recapitulate the model described in Section 3. We use $Nadd_{bx}(t)$ and $Ndel_{bx}(t)$ to denote the number of insertions and deletions respectively on bucket x over t time units.

$$\begin{aligned} Nadd_{bx}(t) &\sim Pois(\lambda_{bx}^{add}.t) && (34) \\ Ndel_{bx}(t) &\sim Pois(\lambda_{bx}^{del}.t) && (35) \\ & && (36) \end{aligned}$$

Here, λ_{bx}^{add} and λ_{bx}^{del} are the rates of additions and deletions respectively on bucket x. These values will vary for different systems, and need to be measured for each database.

Given the mean (H_{bx}) and variance (σ_{bx}^2) for bucket x at some time instant, we want to check how these values would have changed over the following t time units. Again, there is one assumption to be made:

1. The rate of insertions/deletions is very low compared to the heights of the buckets. If this is not the case, we have to analytically make sure the height of buckets does not go negative, and the analysis becomes more complicated. Also, when the insertions and deletions are few, the basic normal distribution of the height is not changed too much. Thus, we can continue assuming that the normal distribution properly represents the bucket heights.

Now,

$$h_{bx}(t) = h_{bx}(0) + Nadd_{bx}(t) - Ndel_{bx}(t) \qquad (37)$$

$$\begin{aligned} E\left[h_{bx}(t)\right] &= E\left[h_{bx}(0)\right] + E\left[Nadd_{bx}(t)\right] - E\left[Ndel_{bx}(t)\right] \\ &= H_{bx}(0) + \left(\lambda_{bx}^{add} - \lambda_{bx}^{del}\right).t \qquad (38) \end{aligned}$$

The analytical formula for the variance is heavily dependent on how the rates λ_{bx} are related to the heights h_{bx}. When they are independent of each other (and this is a severe limitation), we have:

$$\begin{aligned} Var\left(h_{bx}(t)\right) &= Var\left(h_{bx}(0)\right) \\ &+ Var\left(Nadd_{bx}(t)\right) + Var\left(Ndel_{bx}(t)\right) \\ &= \sigma_{bx}^2(0) + \left(\lambda_{bx}^{add} + \lambda_{bx}^{del}\right).t \qquad (39) \end{aligned}$$

An obvious alternate case is when the rate is proportional to the height; we still have no analytical solutions for that case.

The above equations tells us the expected height and variance in height of a bucket if there have been no queries (and the corresponding histogram updates) over it for the past t time units.

5. CONCLUSIONS

Histograms are the heart of most database catalogs. They are used to maintain approximate information about the data stored in the database. We give, to our knowledge, the first statistical analysis of histograms in a database context. We develop statistically sound solutions to various problems involving histograms. These include: how to get the estimated cardinality of a range query or a join, and how to properly update the histogram if we are given the *true* result of a range query. This allows us to progressively update the histogram based on feedback from the query execution stage. It also implies that the portions of the histogram which get accessed most will also be the ones which are most up-to-date.

We also introduce the concept of storing an extra value (called *variance*) in each histogram bucket, along with the expected height of the bucket (which is what histograms currently store). This variance is very useful in properly updating the histogram, and can also be used to provide confidence ranges on questions we ask of the histogram.

One other interesting observation is that the variance in the result of a range query depends on the heights of the buckets in a histogram, but not on their width. This means that it might be good to minimize the height of all buckets, given the constraints on storage space for the histogram. This is a possible plus-point for equi-depth histograms as against equi-width histograms.

6. REFERENCES

[1] Ashraf Aboulnaga and Surajit Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *SIGMOD*, 1999.

[2] George Casella and Roger L. Berger. *Statistical Inference*. Duxbury Press, 1990.

[3] Chung-Min Chen and Nick Roussopoulos. Adaptive selectivity estimation using query feedback. In *SIGMOD*, 1994.

[4] Phillip B. Gibbons, Yossi Matias, and Viswanath Poosala. Fast incremental maintenance of approximate histograms. In *VLDB*, 1997.

[5] Viswanath Poosala and Yannis E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *VLDB*, 1997.

[6] Viswanath Poosala, Yannis E. Ioannidis, Peter J. Haas, and Eugene J. Shekita. Improved histograms for selectivity estimation of range predicates. In *SIGMOD*, 1996.

[7] Michael Stillger, Guy Lohman, Volker Markl, and Mokhtar Kandil. LEO - DB2's learning optimizer. In *VLDB*, 2001.

Historical Queries in IrisNet

Suman Nath
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA-15232, USA
sknath@cs.cmu.edu

ABSTRACT

Recently there has been a spur of research in processing queries on data stream. However, the issues related to storing historical data from a continuous stream and enabling queries on that data have not yet been adequately addressed in the literature. In this paper we investigate the challenges for enabling historical queries in a real system, containing (XML) data derived from continuous sensor feeds. We present the first implementation of the basic mechanisms to deal with the challenges within a XML database engine, and report the preliminary results. We also pose a number of very specific open challenges on this topic.

1. INTRODUCTION

The widespread deployment of sensors and the proliferation of the Internet have fuelled the development of applications that treat data as a continuously evolving stream, rather than a fixed set [5, 10]. One class of such applications operate on wide area sensor networks and depend on storing, processing and querying the data generated by sensors [4, 7]. However, modern sensors (*e.g.*, webcams) may produce a mammoth volume of data with high rate which poses significant challenges in storing, processing and querying the events notified by the sensors.

For many applications, storing the historical sensor data and enabling queries on that are extremely important. For example, a monitoring system may like to post-mortem the data stream to detect the reason of an unusual event happened in the past. Another example can be data mining applications that may like to discover patterns in the data streams, and thus predict the future. Current query processing research on data stream, that mostly focuses on dealing with the current data or data within a pre-specified window [3, 10], is not sufficient to store large volume of sensor data over an extended period of time and hence not suitable for these types of applications. Some data stream systems include the possibility of incorporating query processing on historical data [5, 6], but they have not yet investigated the challenges related to it and the mechanisms to address them.

In this paper we address the challenges related to storing historical sensor data and enabling queries on that in the context of IrisNet(Internet-Scale Resource-Intensive Sensor Network services) [11], an infrastructure for deploying wide area sensor services, which is currently under development at Intel Research, Pittsburgh. We also describe the design and implementation of the basic mechanisms to deal with the challenges.

Project paper for the course 15-823: HOT topics in DATABASE SYSTEMS

Our approach is to store a reasonable approximation of the historical data. The approximation of the stored data is crucial at the situations where sensors produce a huge amount of data and either the data should be stored in the main memory for fast query processing or the secondary storage is too small to store enough sensor data to answer historical queries. Since the stored data is an approximation of the real sensor data, this approach provides only approximate answers to the queries. An efficient mechanism should work within the limited storage and still reasonably approximate the answers to most of the queries.

The contributions of this paper are:

- We address the main challenges to enable historical queries in IrisNet.
- We survey a class of approximate storage schemes suitable for IrisNet, and provide simple guidelines for choosing the right one. We also propose an exponentially decaying sampling technique, which can be very useful for many applications.
- We show how the mechanisms for storing historical data and enabling query on that can be implemented in an XML database engine.

The rest of the paper is organized as follows. Section 2 gives a brief description of the IrisNet. Section 3 discusses the main challenges of enabling historical queries on IrisNet. Section 4 describes the approximate storage schemes to be implemented in IrisNet, and used by application developer in the way discussed in Section 5. Section 6 describes the semantics of the query on the approximate storage schemes we have implemented in IrisNet. Section 7 presents the experimental results. Finally, we pose a few specific research questions in Section 8 and conclude in Section 9.

2. THE IRISNET

IrisNet is an infrastructure for services (*e.g.*, a parking space finder (PSF)) based on resource rich sensors (*e.g.*, webcams connected to laptops). IrisNet is composed of a dynamic collection of two types of nodes: *Sensing Agents* (SA) and *Organizing Agents* (OA). Nodes in the Internet participate as hosts for SAs and OAs by downloading and running IrisNet modules. OAs run on regular PCs in the Internet and create a self organizing overlay network to maintain a distributed database of the real time sensor feeds and other historic data and support queries on that data. SAs run on laptop/PDA class processors connected to the sensors (*e.g.*, webcams) and collect the raw sensor feeds, extract useful information from that, and send the filtered information to the OAs.

Service developers deploy sensor-based services by orchestrating a group of OAs dedicated to the service. As a result, each OA participates in only one sensor service (a single physical machine may run multiple OAs), while an SA may provide its sensor feeds and processing capabilities to a large number of such services.

IrisNet envisions a rich and evolving set of data types, aggregate fields, etc., within a service and across services, best captured by self-describing tags. In addition, each sensor takes readings from a geographic location, so it is natural to organize sensor data into a geographic/political-boundary hierarchy. Hence, sensor-derived data is stored in XML since it is well-suited to representing such data. IrisNet uses the XPATH query language, because it is the most-widely used query language for XML, with good query processing support.

The group of OAs for a single service is responsible for collecting and organizing sensor data in order to answer the particular class of queries relevant to the service (*e.g.*, queries about parking spots for a PSF service). Each OA has a local database for storing sensor-derived data; these local databases combine to constitute an overall sensor database for the service. IrisNet relies on a hierarchically organized database schema (using XML) and on corresponding hierarchical partitions of the overall database, in order to define the responsibility of any particular OA. Thus, each OA owns a part of the database in the hierarchy. An OA may also cache data from one or more of its descendants in the hierarchy. A common hierarchy for OAs is geographic, while it is also possible for a service to define indices based on non-geographic hierarchies. In IrisNet, such hierarchies are reflected in the XML schema. Each service can tailor its database schema and indexing to the particular service's needs, because separate OA groups are used for distinct services.

A user's query, represented in the XPATH language, selects data from a set of nodes in the hierarchy. IrisNet provides mechanisms to route the query to the OA able to answer the query. Upon receiving a query, an OA queries its local database and cache, and evaluates the result. If necessary, it gathers missing data by sending subqueries to its children OAs, who may recursively query their children, and so on. Finally the answers from the children are combined and the result is sent back to the user.

IrisNet currently supports queries only on the most recent sensor data. In the next section, we address the problem of supporting queries on historical sensor data in it.

3. HISTORICAL QUERIES ON STREAM

To support historical queries on a typical sensor based service developed in IrisNet, the following primary challenges need to be addressed first.

1. Since sensors may produce data at a huge rate, and the data stream is assumed to be infinite, OAs cannot store the whole stream. Then what should be the mechanisms for storing the historical data? If multiple such mechanisms are given, which mechanism is the right choice for a particular type of data?
2. Assuming that IrisNet provides a set of mechanisms for storing historical data, how can the service developers use them within their services? How can they tune the parameters of these mechanisms?
3. How can a query be answered using the historical data stored using some predefined storage mechanism? What will be the semantics of the answers when the data required for answering the query is stored only partially or approximately?

In the rest of the paper we describe how we address these challenges in IrisNet. In Section 8 we pose some more challenges (mostly for optimization purposes) we plan to address in our future research.

4. STORING HISTORICAL DATA

We consider data entering IrisNet as a time ordered series of tuples `(streamID, timestamp, data)`. All the tuples in a stream have the same `streamID`. The `data` can range from simple scalar value (*e.g.*, average parking spot availability) to complex multimedia object(*e.g.*, image of a parking spot).

Like other data in IrisNet, historical data is stored as XML documents. However, historical data may comprise large objects (*e.g.*, multimedia data). Those large objects are stored as external files in the secondary memory, and the XML documents contain the corresponding file names. This is necessary for the current XML database engine implementations (*e.g.*, Xindice [1]) which need the whole XML documents to be in the main memory for processing them. Moreover, this enables inter-service sharing, as well as more efficient image and query processing.

Since, the data stream is assumed to be infinite, OAs can store only part or approximation of the historical data. In the rest of this section we describe two different storage models and some schemes for this purpose. Unless specified explicitly, the schemes have been implemented in IrisNet using the Java APIs of Xindice, IrisNet's local XML database engine.

4.1 Partial Storage Model

In this model tuples are stored over a sliding window (*e.g.*, last one thousand tuples or last one hour). The model behaves like a step function: all the tuples within the specified window are stored while the tuples outside the window are discarded. The model is suitable when all the historical queries refer to the times within the sliding window and the available storage allows a large enough circular buffer to store all the tuples within the sliding window.

The advantage of this model includes its simplicity and its accuracy of the answers to the queries within the window. However, this approach has a number of limitations. It cannot answer queries on the data outside the window. Even if the data is periodic, queries outside the window can not be answered if the window is smaller than the period. Storing all the tuples over the complete window may be a huge waste, since user may not care about the *exact* answer for some event happened long ago within the window – only a reasonable approximation may suffice. Moreover, if the storage is limited and the tuple arrival rate is high, the window can be too small to be useful in practice. All these suggest that, a step function like sliding window is not be the most effective storage model for many applications.

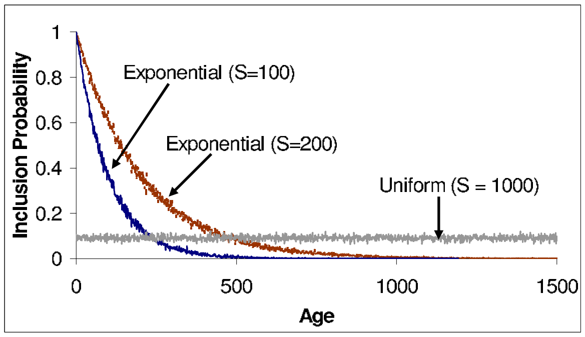


Figure 1: Empirical distribution of inclusion probability with different sampling schemes

4.2 Approximate Storage Model

In this model tuples with old timestamps are stored only approximately and thus data spanning an extended period of time can be stored within the limited storage. Many different approximation schemes can be imagined within this model. In contrast to the sliding window model where the window is fixed, this model fixes the amount of available storage and the approximation scheme. This approach is ideal for many applications having limited storage and requiring only approximate answers to the queries on relatively old data.

One important factor needs to be decided while using this model is the right approximation scheme to use. The decision depends on the amount of available storage, type (*e.g.*, scalar or multimedia) and nature (*e.g.*, rapidly or slowly changing) of the data being stored and the type of the query needs to be answered. We here describe a set of approximation schemes. We also provide simple guidelines for choosing the right one from them.

4.2.1 Sampling using aging function

This scheme stores only a *fixed size random sample* of the tuples. *Fixed size* indicates that the target sample size is pre-specified. *Random sample* indicates that, at any point of time, the inclusion probability of a tuple seen previously in the stream is given by a particular probability distribution.

Sampling is a good choice for storing historical data in a limited storage when individual tuples are independent and there is no prior knowledge about the types of queries that would be made on the data (*e.g.*, data mining application).

Given that the sample is taken over the whole life of the stream whose size is not known a priori, it is *not* trivial to maintain such a random sample [8]. We here consider two fixed size random sampling techniques.

Simple random sampling. The classic reservoir sampling algorithm[8], an online sampling algorithm over a fixed sample size ($= S$) and unknown stream size, ensures *uniform* inclusion probability of the tuples seen so far. The algorithm works as follows.

1. *Add the first k items to the sample.*
2. *On arrival of the n'th tuple, accept it with probability k/n. If accepted, choose an evictee from the sample set uniformly at random, to be replaced by the n'th tuple.*

Figure 1 shows the result of the simulation of this algorithm with $S = 1000$ and total stream size $N = 10,000$ and verifies the fact that all the tuples in the stream belong to the sample with a constant probability S/N [8].

This uniform sampling scheme is suitable in the situations where all the tuples seen so far are equally important.

Exponentially decaying sampling. For most of the applications recent tuples are more important than the old tuples, and so the sample should include more of the recent tuples so that queries on recent data can be answered more accurately than those on old data.

In such a sample, inclusion probability of a tuple decreases with its age (we define age of a tuple to be the number of tuples arrived after it), according to some predefined aging function. However, the sample should be continuously updated with the arrival of tuples so that at any point of time, the distribution of the probability of any of the tuples seen so far being present in the sample follows the aging function. In other words, the aging function should *slide* with time.

We here propose an exponentially decaying aging function, where the inclusion probability exponentially decreases with the tuple's age. For example, consider that the sample stores a tuple with age between t and $t+1$ hour with probability a^{-t}, where a is some constant. This means, at any point of time, it contains all the tuples arrived in last one hour, $1/a$ of the tuples arrived in the previous hour and so on. The following algorithm approximates this behavior.

1. *Add the first k tuples to the sample.*
2. *On arrival of a tuple, choose an evictee from the sample set uniformly at random, to be replaced by the newly arrived tuple.*

We now calculate the probability that a tuple with age t is present in the sample. When the tuple first arrived, it is always included into the sample. On arrival of each tuple after that, the probability that the tuple is selected as an evictee is $1/k$. Thus, the probability that it has not been evicted from the sample when its age is t is $(1 - 1/k)^t \approx e^{-t/k}$. Thus the inclusion probability exponentially decreases with the age t.

Figure 1 shows the inclusion probability of the tuples sampled using this approach with sample size $S = 100$ and 200 and stream size $N = 10,000$. The graph shows that the inclusion probability decreases exponentially with the age of the tuples, as expected.

Exponentially decaying sampling is more suitable in situations where recent tuples are more important than the old ones.

4.2.2 Sliding window of reference tuples

This scheme uses a circular buffer to store only the *reference* tuples so the tuples not stored (*non-reference* tuples) can be approximated (*e.g.*, using interpolation) from the reference tuples. The decision of whether a tuple is considered to be a reference tuple depends on some *selection criteria*. For example, if the data is scalar, and the selection criterion is `difference = 2`, then a tuple will be stored as a reference tuple only if its value differs from the last reference tuple by at least 2. The selection criteria present tradeoffs between the accuracy of approximating the non-reference tuples and the time span over which data can be stored within the limited storage. Very fine grained selection criteria store more reference tuples and hence provide better approximation of the non-reference tuples. On the other hand, since more reference tuples are stored, only the history over a limited time may be maintained.

This scheme is more appropriate for data that do not change too often over time.

```
<state @id='PA'> <city @id='Pitt'> <block @id=1>
   <available-parking-spots>
      <historical> yes </historical>
      <storage> SAMPLE_EXP_DECAY </storage>
      <sample-size> 100 </sample-size>
      <value> 0 </value> <timestamp> 0</timestamp>
   </available-parking-spots>
</block></city></state>
```

Figure 2: Specification of the storage scheme within the XML schema

4.2.3 *Lossy Compression*

Lossy compression techniques can be used to store historical multimedia data. For example,

- For video data, frames can be sampled using exponentially decaying sampling described before. Moreover, the fidelity of sampled frames may decrease with their age. For example, recent video frames can be stored as high resolution color frames, while the older frames may be stored as low resolution, black and white frames. Finally, each video frame can be compressed using existing multimedia compression algorithms.
- Scalar data represented as a time series can be compressed using wavelets. The fidelity of compression may decrease with age of the data.

As mentioned before, the multimedia data will be stored as external files and compressed using an external compression engine. Note that, the compression is transparent to the IrisNet, since XML documents only use the file names to refer to the objects. IrisNet has not yet incorporated any compression engine to support this scheme.

4.2.4 *Synopsis*

If the types of historical queries are known a-priori, synopses data structures [9, 3] can be used to maintain statistics about the historical data. Synopsis can maintained be over a sliding window or over the whole life time of the stream. Currently, IrisNet does not support any synopsis data structures.

5. PROGRAMMING MODEL

Different storage schemes supported by IrisNet are exposed to the service developer through well known APIs. In the similar spirit of XML based RPC mechanisms [2], a service developer specifies the storage schemes to be used and corresponding parameters in self describing XML tags, as part of the XML schema. Since the choice of approximation scheme depends on the type and nature of the data, storage schemes may be defined at the granularity of each individual data item within the schema.

Figure 2 shows part of the schema used by the PSF service where the service developer wants to store the historical data of available parking spots. The storage scheme to be used is the exponential decay sampling with sample size 100.

6. QUERY MECHANISM

In IrisNet, queries can be asked along two different axes: space and time. As described in Section 2, IrisNet supports queries on current sensor data in any geographic locations within its hierarchy. If the data spans the space covered by multiple OAs, IrisNet generates subqueries, gathers the required data by routing the subqueries to the corresponding OAs, and finally aggregates the responses to generate the final answer. Incorporating historical queries enables IrisNet to process queries along the time axis too.

Since historical data is stored only approximately, answers to the queries on it will be approximate too. The semantics of the answer to a query depend on the query itself, the type and nature of the data and the approximation scheme and parameters used to store the historical data. In addition to the query answer, IrisNet provides some extra *hints* about the stored data on which the query was processed. The hints, along with the query itself, data type and the approximation scheme used provides the users with the semantics of the answer.

We have implemented four different storage schemes for historical data: sliding window storage (SW), uniform (US) and exponentially decaying sampling (ES) and reference tuple storage (RT). IrisNet currently supports the following two types of queries on these storage schemes.

- **Time point query:** This type of historical queries refer to some fixed point of time t in the past (*e.g.*, total available parking spots at 10 a.m. today). SW storage scheme can answer the query only if t is within the window, in which case the answer is accurate. RT scheme can answer the query if there is any reference tuple beyond the query time point. In all the cases, the query is answered using the tuple with the largest timestamp $t' \leq t$. The timestamp t' of the tuple used to answer the query is returned as part of the answer as the *hint*.
- **Time range query:** This type of historical queries refer to a window of time (t_1, t_2) in the past (*e.g.*, average available parking spots between 10-11 a.m. today). For all the schemes except the RT storage, the query is answered using all the tuples with timestamps t' such that $t_1 \leq t' \leq t_2$. (Again, SW storage scheme can answer the query only if both the t_1 and t_2 are within the window.) For RT storage, all the tuples between the two tuples with timestamps t'_1 and t'_2 are used, where t'_1 is the largest timestamp with $t'_1 \leq t_1$ and t'_2 is the largest timestamp with $t'_2 \leq t_2$. Total number of tuples used to calculate the answer, t'_1 and t'_2 are provided as hints with the answer.

  The semantics of the answers, however, also depend on the particular storage scheme used. For example, the answer to the query for average parking spots on data stored using US storage is an un-weighted average whereas the same on data stored using ES storage is an *exponentially weighted* average.

The implementation of the time-point queries in IrisNet is straightforward. XPATH provides functions to select the tuple with the largest timestamp $t' \leq t$. Time range queries for the schemes other than the RT storage is also straightforward to implement using XPATH's range query. However, to implement a time-range query with the range (t_1, t_2) with RT storage, two subqueries need to be made to find the `position`s (with respect to all the siblings) of the two boundary tuples. Then all the tuples between these two positions are extracted from the database and used to answer the query. To produce the final query answer, the tuples

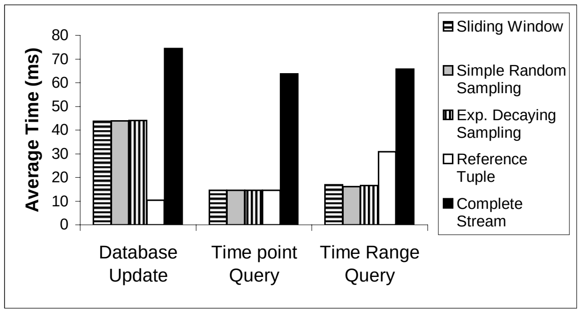


Figure 3: Time taken by database operations under different storage scheme

selected using the time-range query may need further processing (*e.g.*, calculating the average value of the tuples) within the Java code.

7. EXPERIMENTAL RESULTS

This section presents an experimental study of our implementation, which seeks to answer the following two questions: (1) What is the raw performance of the implementation, in terms of database update and querying time? and (2) What is the approximation error of the answer to a historical query under different storage schemes?

7.1 Experimental Setup

We have implemented the mechanisms to support queries on historical data in IrisNet by extending OA's code written in Java, and using OA's local XML database Xindice. The queries are made using the XPATH language. All the results presented here are produced using an OA running on Redhat 7.3 Linux in a lightly loaded Pentium-III, 1.8 GHz machines with 512 MB RAM.

We here consider four approximate storage schemes we have implemented in IrisNet: SW (sliding window), US (uniform sampling), ES (exponentially decaying sampling) and RT (reference tuple) storage. We compare them with the complete storage (CS) scheme, where the complete stream is stored in the database. In all the storage schemes, we assume that the storage size is constrained to store at most 1000 tuples while the whole stream consists of 10,000 tuples. We use a synthetic workload of the stream where the data values of the tuples denote the total number of available parking spots. The number of available spots is modelled using a random walk model, with equal probability of the value of a newly arrived tuple being unchanged, increased or decreased by one (equivalently, the probability of a car entering or leaving the lot) from that of the previous tuple.

Since database update is expected to be much more frequent than the historical queries, maintaining indices on the database is expensive (in general, maintaining index in XML database is more difficult than maintaining that in relational database). So for evaluating the queries we consider unindexed database. We consider time-point and time-range queries, where times are chosen using a Zipf like distribution, with the recent times chosen more often than the old times as the query point. On average, a query is made on arrival of 10 tuples.

7.2 Costs of database operations

Figure 3 shows the average time required for different database operations in our implementation. The first set of bars shows the per tuple database update time. On contrary of our belief, we found that the database update time in Xindice increases linearly with the size of the database! Since approximate storage schemes use smaller database (equal to the sample size), they get advantage in database update time even though each of their updates requires multiple database operations (deletion of the evictee, insertion of the new tuple etc.). This explains why the CS scheme has worse database update time than the approximate storage schemes. All approximate schemes, except the RT scheme, take almost equal database update time (which is proportional to the sample size). The RT scheme stores a tuple only if it is different from the last reference tuple by at least a threshold of 2. This explains why its per tuple database update time is less than the other approximate schemes.

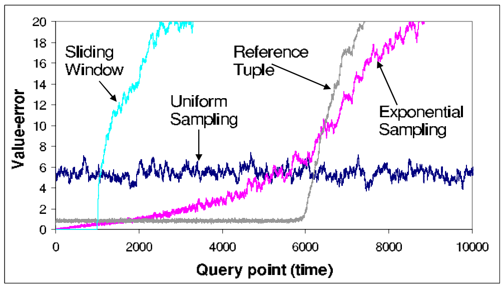
Figure 3 also shows that queries on CS scheme take much more time than those on approximate storage, due to the fact that without any index, queries take times proportional to the database size. For time point query, all the approximate schemes take almost same average query response time because of their equal sample size. The figure also shows that, time-range queries take slightly more time than the time point queries. This is because of the extra processing of the tuples outside Xindice (to compute the average, for example). The semantics of the queries in RT scheme suggest that, for each time range query, two subqueries need to be made, which explains its high query response time for time range queries.

7.3 Approximation errors in query answers

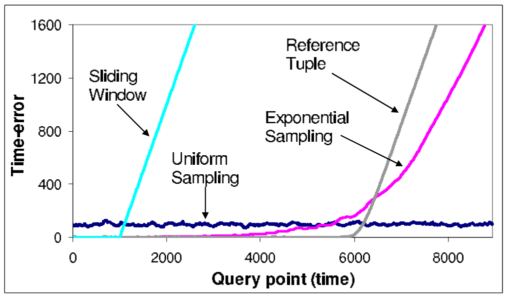
We consider two different types of approximation errors of time-point query answers while evaluating the performance of different storage schemes. The first one, *value-error*, is the difference of the answer found using the complete storage and that found using the approximate storage scheme. This depends on the nature of the data (*e.g.*, how often the values change), and we here use the synthetic stream described in Section 7.1. The second type of error, called *time-error*, is the difference of the time specified in the query and the timestamp of the tuple used to answer the query. This is independent of the data value, and indicates the temporal closeness of the answer desired and the answer returned by the system. Although we here evaluate the time-point queries, errors for time-range queries can be explained similarly.

Figure 4(a) and 4(b) show the value-error and time-error respectively, of different storage schemes. SW provides accurate answers if the query time point is within the window size (=1000), while the RT scheme provides almost accurate answers over a larger window (=6000) with the same amount of storage. Beyond those window, these two schemes just use the tuple closest to the query time point, and thus experience very bad value and time-errors. For US scheme, error remains constant all over the life time of the stream, but it is worse than the errors with SW and RT schemes within their window. ES scheme provides very good accuracy of the queries on recent time, and the error worsens with the time point of the query.

Figure 4(c) shows the average query response errors assuming that most of the queries are made on recent time points (*i.e.*, the query time point follows a Zipf-like distribution). As expected, RT and ES schemes give very small error for such queries.



(a) Value-Errors of query answers



(b) Time-Errors of query answers

	Value Error	Time Error
Sliding Window	4.19	35128.89
Reference Tuple	0.52	50.95
Uniform Sampling	3.26	104.19
Exp. Sampling	0.67	15.65

(c) Average Error

Figure 4: Accuracy of query answers with different storage schemes.

8. OPEN CHALLENGES

This section lists a few challenges that are yet to be addressed in the context we have discussed in this paper.

8.1 Resource management

Resources (*e.g.*, storage, computation) are allocated per instance of the approximation scheme. If a service wants to archive n different data items (*e.g.*, number of available parking spots, number of red cars, snapshot of the spots etc.), n different approximation schemes will be instantiated for it, one for each data item. Since, the accuracy of the historical query increases with the amount of resources used by the approximation scheme, multiple services running on the same physical machine may contend with each other for resources. One big challenge in this context would be to manage the available resource, and dynamically allocate it among the contending services.

8.2 Adaptivity of the approximation schemes

The appropriate approximation scheme to be used depends on the nature of the data, which may change over time. For example, during the busy period of the day, the total number of available spots in a parking lot changes frequently, and so ES scheme may be a good choice for storing historical data. However, at night or on weekends, when the parking spot availability data do not change too often, the best approach may be to sample less often or to use the RT scheme. Dynamically detecting the nature of the data, and accordingly adapting the approximation scheme is a big challenge in this context.

Dynamic resource management poses other challenges for the storage mechanisms to adapt with the available resource. For example, if the sample size of the ES scheme is increased, it is not clear how the sampling algorithm should be adapted accordingly, or how the immediate queries should be answered with the expected semantics.

8.3 Error propagation along the space axis

If the historical query spans a geographic region covered by multiple OAs, IrisNet produces the final answer by aggregating the responses from different OAs. However, different OAs may use different approximation schemes for the same data, or even if the they all use the same approximation scheme, they may use different parameters (*e.g.*, due to different resource constraints, different OAs may have different sample size). Thus, different degree of approximation errors form different OAs will be propagated along the path of aggregation. It is not obvious what the semantics of the final answer would be with this type of aggregation.

9. FUTURE WORK AND CONCLUSION

In this paper we have presented the challenges to store historical data and enable queries on that in a system where data comes as a continuous stream from powerful sensors. We have presented the implementation and performance evaluation of the basic mechanisms implemented in IrisNet to address the challenges. We also have presented the bigger picture and a few specific research challenges in this domain. The work is ongoing. The next steps we are actively taking include addressing the challenges posed in section 8.

10. REFERENCES

[1] Apache xindice. http://xml.apache.org/.

[2] XML RPC. http://www.xmlrpc.com/.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of 21st ACM Symposium on Principles of Database Systems (PODS)*, June 2002. Madison, Wisconsin, USA.

[4] P. Bonnet, J. E. Gehrke, and P. Seshadri. Towards sensor database systems. In *Proceedings of the Second Intl. Conf. on Mobile Data Management*, January 2001. Hongkong.

[5] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams – a new class of data management applications. In *Proceedings of Very Large Databases (VLDB)*, August 2002. Hongkong, China.

[6] S. Chandrasekaran and M. Franklin. Streaming queries over streaming data. In *Proceedings of Very Large Databases (VLDB)*, August 2002. Hongkong, China.

[7] A. Deshpande, S. Nath, P. B. Gibbons, and S. Seshan. Cache-and-query for wide area sensor databases. Submitted for publication.

[8] C. Fan, M. Muller, and I. Rezucha. Development of sampling plans by using sequential (item by item) selection techniques and digital computers. *Journal of the American Statistical Association*, 57:387–402, 1962.

[9] P. B. Gibbons and Y. Matias. Synopsis data structures for massive data sets. *DIMACS: Series in Discrete Mathematics and Theoretical Comp. Sc.: Special Issue on External Memory Algorithms and Visualization*, A, 1999.

[10] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the ACM SIGMOD int. conference on management of data*, June 2002. Madison, Wisconsin, USA.

[11] S. Nath, A. Deshpande, Y. Ke, P. B. Gibbons, , B. Karp, and S. Seshan. Irisnet: An architecture for compute-intensive wide-area sensor network services. Submitted for publication, Intel Technical Report IRP-TR-10, 2003.