T-Cube: Quick Response to Ad-Hoc Time Series Queries against Large Datasets

Maheshkumar Sabhnani, Artur Dubrawski, Andrew Moore

The Auton Lab, Carnegie Mellon University, Pittsburgh, PA, USA
{sabhnani,awd,awm}@cs.cmu.edu

Abstract. We present a novel data structure called T-Cube which dramatically improves response time to ad-hoc time series queries against large datasets. We have tested T-Cube on both synthetic and real world data (emergency room patient visits, pharmacy sales) containing millions of records. The results indicate that T-Cube responds to complex queries 1,000 times faster when compared to the state-of-the-art commercial time series extraction tools. This speedup has two main benefits: (1) It enables massive scale statistical data mining of large collections of time series, and (2) It allows its users to perform many complex ad-hoc queries without inconvenient delays.

1 Introduction

Time series data is abundant in many domains including finance, weather forecasting, epidemiology, and many others. Large scale bio-surveillance programs monitor status of public health against adverse events such as outbreaks of infectious diseases and emerging patterns in public health. They rely on data collected throughout a health management system (hospital records, health insurance companies records, lab test requests and results, issued and filled prescriptions, ambulance and emergency phone service calls, etc) as well as outside of it (school/workplace absenteeism, sales of non-prescription medicines, etc). The key objective is to as early as possible and as reliably as possible detect those changes in statistics of the data sources which may be indicative of a developing public health problem. One of the challenges the users of such systems face is that of data overload. The actual number of e.g. daily transactions of drug sales in pharmacies across a sizable country may be very large. The users need tools to enable timely analysis of those massive data sources. The analyses can be performed automatically (with a data mining software), however patterns discovered that way are almost always subject to a careful scrutiny through a manual drill-down. In both scenarios, massive screening of very large collections of data must be executed really fast in order to make these bio-surveillance systems useful in practice. A saving of just a few hours of detection latency of an outbreak of a lethal infectious disease can yield enormous monetary and social benefits [10], [11].

Most of the kinds of data mentioned above can be interpreted as time series of interval (e.g. daily) counts of events (such as number of certain type of drugs, e.g. anti-diarrheals sold; number of patients reporting to emergency department with specific symptoms, etc). These time series can be sliced-and-diced across multiple symbolic

dimensions such as location, gender and age group of patients, and so on. Computational efficiency of data mining operations which one may want to apply to such data, as well as the efficiency of accessing interesting information in a manual drill-down mode, heavily depend on the efficiency of extraction of series of counts aggregated for specific values of these demographic dimensions.

This paper introduces a new data structure designed to dramatically decrease the time of retrieval of such aggregates for any complex query combining values of symbolic variables. It achieves its efficiency by pre-computing and caching responses to all possible queries against the underlying temporal database of counts annotated with sets of symbolic labels, while keeping memory requirements in check.

2 Importance of Ad-hoc Queries against Temporal Databases

A typical transactional database contains multiple attributes. In temporal databases, one of the key features is *date* which allows ordering entries by time and creating time series representations of the contents. Other fields may be symbolic or real-valued, or have a special type. In this paper, we focus on temporal databases with *symbolic* attributes used to characterize *demographics* of the individual entries. The databases of our specific interest can be thought of as records of transactions – there is also a *count* component in them. Let us assume that all the attributes of the transactional dataset are symbolic except of the *date* field. Distinct values of such *demographic attributes* are *demographic values* or *properties*. E.g. postal code (called "zip code" in the US) is a demographic attribute and "15213" is one of possible values it can assume. Also let us define a *set of demographic properties* (*DPS*) to be a set of assignments of demographic values, one per each demographic attribute in the data. E.g. in a dataset containing *zip code* and *gender*, the particular combination of its values such as {15213, Male} is a DPS. Most databases we encounter in practice contain between ten and hundred thousand unique DPSs.

Often, time series queries extract data corresponding to conjunctions of demographic attribute-value pairs. A *simple query* is restricted to up to one value per demographic attribute in it. In a *complex query*, each attribute may be characterized by a set or a range of values. Each simple or complex query can use only a subset of all available demographic attributes (alternatively, the attributes not represented in such query may be plugged in with the complete list of their values: it would be equivalent to "do not care" symbol put next to that attribute name in a select query).

Even for databases with all demographic attributes indexed separately, simple queries require aggregation at run time which might be expensive. However, the number of just simple queries is exponential in the number of demographic attributes and the number of values present in each demographic attribute. The number of possible complex queries is very much higher than that as the exponential factor gets doubled. It is thus not practical to pre-compute and store the answers to all possible queries in a straightforward fashion. Database management (DBM) systems often provide tools to pre-cook materialized views and use stored procedures which help quickly answering complex queries. Those DBM techniques fail in the ad-hoc settings, where the user queries are not known in advance.

We are aware of at least two good examples of practical settings in which ad-hoc queries are needed. Firstly, take a database used by public health officials for bio-surveillance. These officials perform disease outbreak monitoring on a daily basis. It often involves investigation of alerts of possible problems flagged by the automated software systems. For such investigations, the health officials need to execute large number of complex ad-hoc queries in order to extract data needed for interpretation of the alerts and/or for isolating their likely causes, while differentiating real outbreaks from false alarms. Such investigations need to be executed in a timely fashion, and long waits for data extracts are not acceptable. Secondly, automated statistical analyses or data mining procedures executed on such databases require access to a large number of different projections of data, if they are to comprehensively and purposively screen for possible indications of public health problems. Long waits for the corresponding extracts may (and often do) render such systems impractical as both of these usage cases heavily rely on quick responses to complex queries. They would dramatically benefit if caching responses to all possible queries in advance was available.

2.1 Related Work

Standard approach to handling ad-hoc queries in commercial databases is that of On-Line Analytical Processing (OLAP). The idea relies on *data cubes*, cached data structures extracted from (usually only parts of) the original data and made in the form allowing for fast ad-hoc querying of pre-selected subsets of aggregated data [1]. For the sake of brevity we do not review the details of OLAP technology here, but these methods are known to suffer from long build times (typically hours for the databases of sizes and complexities similar to those used to illustrate results in this paper) and huge memory requirements (causing the need to rely on high-end database servers). Additionally, as we observed empirically, data cubes still typically require O(1.0 second) or longer to respond to a complex query on the datasets which we tested. Such latency is an inconvenience to users who want to perform multiple ad-hoc queries in an online fashion. It also hampers statistical analyses which may require millions of complex queries which would take days of processing time using industry-standard OLAP data cubes. More details and illustrative examples can be found in [3].

There are different types of implementations of data cubes based on how counts are stored internally. Relational OLAP (ROLAP) stores all counts in the data cube in the form of a relational table. Multi-dimensional OLAP (MOLAP) directly stores the counts as a multi-dimensional array (hence the response to any simple query can be obtained in constant time which makes them faster than ROLAP counterparts, but their memory requirements grow exponentially with the number of dimensions in data). A few other popular variants of data cube implementations include: Hybrid-OLAP (HOLAP) and Hierarchical OLAP. HOLAP combines the benefits of ROLAP (reasonable memory requirements) and MOLAP (faster response time). Idea of HOLAP follows intuition that the data cubes built at higher abstraction levels are denser than those build at the lower levels, and they often use MOLAP at coarser levels and ROLAP at finer levels. Hierarchical OLAP is used when the various values of a demographic attribute in data are connected to each other using some hierarchy. For

e.g. date attribute can be split into year, month, or day and then aggregations at year or month level could be obtained using data cubes. Irrespective of the implementation, the goal of data cubes is to respond to simple and complex queries for large practical databases as fast as possible. With growing demand for data mining and statistical analyses against large databases, innovation work has been focusing on improving data cube performance [7],[8],[9].

Data cubes are closely related to another technology which originated from computer science research: *Cached Sufficient Statistics*. Similarly to data cubes, cached statsitics structures pre-compute answers to queries, but they cover all possible future queries, and aim at efficiency of not only data retrieval, but also their (most often in-memory) representations. AD-Trees are very good examples of such data structures.

AD-Tree [2] stands for All-Dimensional Trees and it is designed to efficiently represent counts of all possible co-occurrences among multiple attributes of symbolic data. This is very important in many scenarios involving statistical modeling of such data, where most operations require computing aggregate counts, ratios of counts or their products. Quick access to counts of arbitrary subsets of demographic properties is essential for overall performance of analytic tools relying on them. AD-Trees have been shown to dramatically speed-up notoriously expensive machine learning algorithms including Bayesian Network learning [2], Empirical Bayes' Screening, Decision Tree learning and Association Rule learning [5]. The attainable speedups range from one to four orders of magnitude with respect to previously known efficient implementations. These efficiencies are attainable at moderate memory requirements, which are easy to control. Moore describes in [2] the details of the structure, its construction algorithm as well as fundamental characteristics of the AD-Trees. There are also dynamic implementations of AD-Trees [6] which help grow AD-Tree parts on demand and can be more memory efficient than fundamental implementations. AD-Trees are the best of the existing solutions to symbolic data representation when it comes to very quickly responding to ad-hoc queries against large datasets. The idea of T-Cube originates in AD-Trees.

3 T-Cube

T-Cube (Time series Cube) is an in-memory data structure designed for very fast retrieval and analysis of additive data streams such as e.g. time series of counts. It is a derivative of the idea of AD-trees. We show how AD-Trees can be easily and efficiently extended to the domain of time series analysis.

T-Cube consists of two main components: D-Cache and AD-Tree (note that because AD-Tree is designed to work with symbolic data, the T-Cube is applicable to temporal datasets with symbolic demographic attributes).

D-Cache is represented as a two-dimensional matrix with rows containing one *DPS* (a set containing one value per demographic value) and columns containing time duration. Table 1 below shows the structure of D-Cache assuming a retail dataset has two demographic attributes (*color* and *size* of a t-shirt) and there are '*T*' days in the database. The result of any time series query (simple or complex) against such dataset is an aggregated time series over the rows (or *DPSs*) of D-Cache that match the query

condition. Hence using D-Cache only, the query response is linear in the number of distinct *DPSs* in the data. This can be slow if the number of rows in the D-cache is large.

Table 1. D-Cache data structure for retail dataset.

	Day_1	Day_2	...	Day_T
(Red, Small)	10	15	...	20
(Red, Large)	5	1	...	7
...	...	...	...	...
(Blue, Medium)	7	5	...	9
(Green, Small)	5	4	...	10

The goal of using the AD-Tree structure is to reduce the query response time from linear to logarithmic in the number of rows in D-Cache. We build an AD-Tree using the rows in D-cache that contain the different *DPSs* present in the data. Traditionally each data node of AD-Tree [2] contains a single count of occurrences corresponding to the particular DPS. But now in T-Cube representation, the data node consists of a series of counts of size *T*. This time series is the sum of all the rows that match the current data node in D-Cache. We call the matching rows as the *leaf-list* for the corresponding data node. As we go from top to the bottom of the tree, the leaf-list size of data nodes keeps reducing in size and eventually becomes of size one, which corresponds to a single row in the D-Cache. Hence AD-Tree stores responses to all possible simple queries against the database. Figure 1 shows the AD-Tree representation of the t-shirt database shown in Table 1. The data nodes with time series are shown as circles with shaded bars and the vary nodes over demographic attributes are shown with rectangles. D-Cache combined with AD-Tree structure makes the T-Cube. We now have a data structure that can be used as a basic tool by other statistical data analysis algorithms to quickly retrieve any time series from the database. Hence we have extended the utility of AD-Trees to the time series domain.

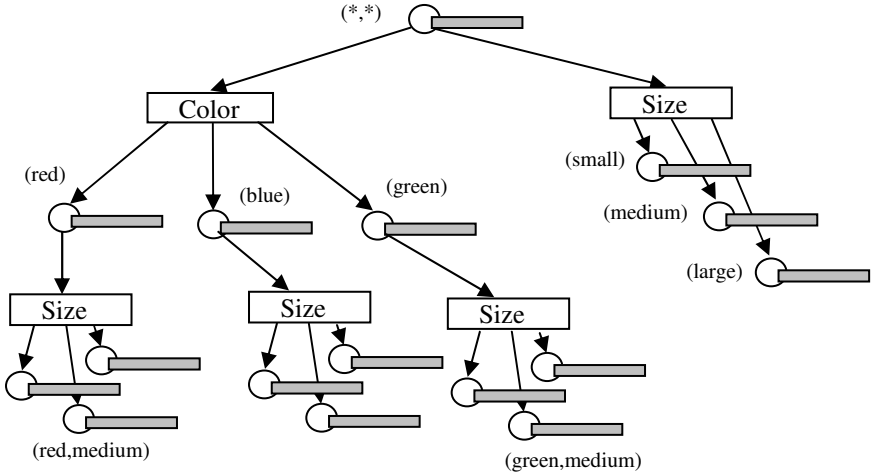


Fig. 1. AD-Tree for T-Shirt database.

4 Datasets

We tested T-Cube on three different datasets: two real and one synthetic. All datasets had records from a single year. This section describes the dataset characteristics: size, attributes, and properties. Even though the first two datasets are from biosurveillance domain, their size and characteristics are similar to most practical datasets: a few attributes having large number of properties (zip codes, companies, etc.) and most attributes with small number of properties (gender, company size, etc.).*Emergency Room Chief complaint data* (ED) dataset contained hospital emergency room patient visit records from four different states (PA, NJ, OH and UT). Each record consisted of the following attributes: visit *Date*, *Syndrome*, patient home *Zip Code*, *Age Group*, *Gender* and *Count*. The patient home *Zip Code* had 21,179 distinct values. Note that even though the hospital is located in one of the four states, the patient home *Zip Code* can be from any part of the U.S. This includes home *Zip Codes* of patients visiting from other states in the country too. of Attribute *Syndrome* represented the reported chief complaint by the patient and had 8 distinct values (examples: Respiratory, GI, Constitutional, etc.). Attribute *Age Group* could take one of the three values: Adult, Child and Unknown. Similarly attribute *Gender* could also take three values: Male, Female, and Unknown. The dataset had approximately 3.4 million records. There were a total of 120,604 *DPSs* in the data.

Over-the-counter (OTC) data contained medication sales in different pharmacy stores throughout the U.S. The dataset had records from more than 10,000 stores nationwide. Each record contained the following attributes: purchase *Date*, store *Zip Code*, medicine *Category*, sale *Promotion*, and *Count*. The data had 8,119 distinct *Zip Codes*. The attribute *Category* represented the class of medicine (e.g. Cough/Cold, Baby/Child Electrolytes, etc.) and had 23 different values. Store promotions on medications sold were stored in the attribute *Promotion* as 'Y' or 'N'. Attribute *Count* stored the purchased quantity. The dataset had a total of 356,545 *DPSs* in the data.

Most practical datasets have a few demographic attributes: ED (4) and OTC (3). Some datasets may have many demographic attributes each having only a small set of distinct values (say, binary). *Sparse binary synthetic* (SYN) data was build to show that T-Cubes can handle such datasets with a large set of demographic attributes. The data contained 32 attributes: *Date*, *Count*, *Zip Code*, and 29 *binary* (0, 1) attributes. Each of the 29 binary attributes were 95% sparse, i.e. they took the value *0* 95% of the times in all the records. Attribute *Zip Code* randomly took one of the 10,000 values for each record. Attribute *Date* was in the range of one year. Attribute *Count* too values from 5 to 10 again randomly. We generated 12 million records in one year duration with 4.5 million *DPSs* in the data.

5 Experiments

We now describe the various experiments performed using T-Cubes on the datasets described in section 4. In particular, section 5.1 shows results for T-Cube building time and memory utilization. Section 5.2 suggests ways of managing the exponential memory requirement of T-Cubes to build AD-Tree. Section 5.2.1 discusses attribute

ordering, section 5.2.2 discusses effect of controlling tree depth and section 5.2.3 briefs on the most-common-value trick for AD-Tree. All the experiments in this paper were performed on a system with AMD Opteron 242 Dual CPU (1600 MHz) processor and 16GB system memory. The system was running on CentOS 4 x86_64 operating system.

5.1 T-Cube Build Time and Memory Utilization

We present the build times and memory utilization for the T-Cube in Table 2. It was not possible to build the AD-Tree for SYN dataset because the dataset required more than 16GB of system memory. The build time for D-Cache is approximately 3-15 minutes where as the build time for AD-Tree is only a few seconds. The build times indicate that most of time needed to build the T-Cube is spent in making the D-Cache. Once all the DPS time series for are cached in the D-Cache structure, making the AD-Tree is very fast. Also note that most of the building time in D-Cache is spent to read the data from the disk. Since the D-Cache structure is build only once, we only pay one time cost for building this structure. Hence spending a few minutes just to build the D-Cache is fine.

As contrary to build time, the memory requirements of AD-Tree are much more than D-Cache. For e.g. ED dataset, we need only 36MB of memory to store D-Cache but the AD-Tree requires 676MB of memory (20 times). This is due to the fact that the memory requirement of AD-Tree is exponential in the number demographic attributes and corresponding properties. Note that the memory demand for D-Cache and AD-Tree will grow linearly with the time duration in the dataset.

Table 2. Building time and memory utilization of T-Cubes

	Building Time (secs)		Memory (MB)	
	D-Cache	AD-Tree	D-Cache	AD-Tree
ED	124	16	36	676
OTC	944	26	77	1116
SYN	913	-	808	-

5.2 Managing Memory Usage

Table 2 shows that T-Cubes require exponential amount of memory. Every time we run a query againt the T-Cube we need to pay for utilizing such large amounts of memory. And as in the case of SYN dataset, the AD-Tree is more than 16GB. This section addresses various ways in which we can reduce the memory demand of T-Cubes. This benefit is acheived at the cost of slower query response time.

5.2.1 Demographic Attribute Ordering

In the AD-Tree structure, the left part of the AD-Tree contains more nodes than the right (see Figure 1). This is because the left children of the root have more demographic attributes than the right. We found (empirically) that if the demographic at-

tributes are arranged in decreasing order of arity (number of distinct values) from left to right in the AD-Tree then the tree has smaller number of nodes in total. Intuitively, this heuristic will prevent the attributes with more distinct values from duplicating in multiple children of the AD-Tree root. For e.g., we should arrange the ED dataset demographic attributes in the following order: patient home *Zip Code*, *Syndrome*, *Age Group*, and *Gender*.

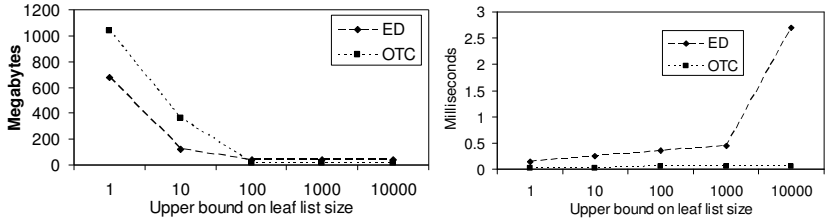
Table 3 shows the memory savings by using this heuristic. It is clear from the Table that arranging attributes in decreasing order of their arity saves memory as compared to increasing order. This heuristic results in smaller memory reduction at no effect on query response time. We save nearly 200MB for ED data with 140k nodes less for the case of ED dataset which is significant. We still are unable to build the SYN dataset within 16GB of memory.

Table 3. Effect of demographic attribute ordering.

	Memory (MB)		#nodes	
	Increasing	Decreasing	Increasing	Decreasing
ED	875	676	484k	346k
OTC	1049	1039	565k	552k
SYN	-	-	-	-

5.2.2 Controlling Tree Depth

Each data node (shown as circle with gray bar in Figure 1) aggregates a set of rows from the D-Cache. We define the leaf list size of any node in the AD-Tree as the number of rows matching the node in the D-Cache. In a fully grown AD-Tree, the leaf list size of leaves in the tree is exactly one. To reduce the memory requirement, we can set an upper bound on the leaf list size of all data nodes in the tree. We represent this parameter as the *r-value*. A large *r-value* for the AD-Tree will correspond to a smaller tree and hence lower memory usage. This will come at a cost of slower response time since now even for a simple query we might have to aggregate rows in the D-Cache.



(a) Memory usage (b) Response Time

Fig. 2. Effect of controlling AD-tree depth using *r-value*

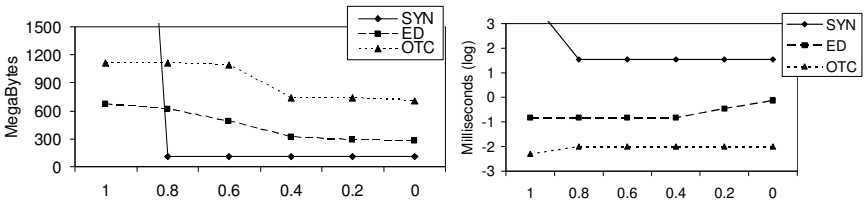
Figure 2 shows the memory requirement of ED and OTC dataset with varying *r-value* on the x-axis. For ED data, the memory requirement falls from 676MB to 38MB (almost 20 fold less) when the *r-value* is changed from 1 to 100 (Figure 2(a)). Similar effect is seen for OTC data. Also from Figure 2(b) when the upper bound is 1,

the query response time 0.1 milliseconds and when the bound is 1000 the response is 0.5 milliseconds. Note that there is drastic reduction in memory requirement with very little difference in response time. Also *r-value* of 100 seems to give the most benefit in terms of memory-time trade off. We omit SYN data because even at *r-value* of 10,000 it required more than 16GB of memory.

5.2.3 Most-Common-Value

The number of children of a vary node (shown as rectangles in Figure 1) in the AD-Tree is equal to the number of distinct values present in the demographic attribute. Let us define the child of the vary node with the maximum leaf list size be the most-common-value child (ties broken randomly). Moore & Lee [2] showed that it is possible to remove this child completely from the tree and still be able to build it later if needed. This is because the sum of all children of the vary node is exactly equal to that of the parent. This definition can be used recursively to generate the complete tree for and below the most-common-value node. Hence, deleting the most-common-value node helps reduce memory requirement of AD-Tree because this results in removing the child with most dense sub-tree beneath.

The most-common-value results in immense memory savings for datasets with demographic attributes having skewed distribution in their distinct values (SYN dataset). But if the MCV node is queried, the response time suffers because a lot of time is spent regenerating the MCV sub-tree. For e.g. if we use the MCV trick at a vary node with 10,000 children each having a almost similar leaf list size, then we will have to add 10,000 time series at run time to respond to the MCV child query. This is computationally expensive. Hence we define a parameter called MCV fraction (γ) that is the ratio of the leaf list size of MCV node and the leaf size of the vary node. We use this parameter as lower bound and use MCV trick at a vary node only if this bound is satisfied. This parameter tries to balance the memory savings with query response time.



(a) Memory usage (b) Response time

Fig. 3. Effect of Most-Common-Value

Figure 3 shows the memory usage and response time for different values of γ (on the x-axis). γ=1 corresponds to not using MCV at all and γ=0 means MCV applied at each vary node. Figure 3(a) shows the memory usage of SYN dataset to be around 113MB when γ=0.8 as compared to >16GB when γ=1. This result indicates MCV trick is very useful for datasets with attributes having skewed distribution. There is 30-40% savings in memory for ED and OTC dataset when $\gamma \leq 0.4$. Figure 3(b) shows the effect of γ on the query response time. The query response time is almost the same

for $\gamma \geq 0.4$. Hence if we had to pick a single MCV value for all three datasets, we can choose $\gamma=0.4$.

6 Performance

In section 5 we saw that T-Cubes can be built in minutes. We also presented methods by which the exponential memory requirement (GB's for SYN data) of the AD-Tree can be reduced to a few hundred megabytes for all of the three datasets. Also the response time for simple queries is within a fraction of milliseconds. In this section we present results for performance on complex queries. We also compare T-Cube with three other state-of-art data cube tools.

6.1 Complex Query Response Time

In Section 1, we defined complex queries as the ones which allow each demographic attribute to take multiple values in the query. Hence a query which requests counts for 1,000 *Zip Codes* is more complex than the query which only requests it for 10 of them. Let us define the complexity, β, in terms of the fraction of values that an attribute is allowed to take from its legal set of distinct values. Hence β will range in [0,1] and higher values will indicate higher complexity of a query.

Figure 4 shows the response time with varying values of query complexity β. The parameters for the tested datasets were as follows: *r-value* = 1000 and $\gamma = 0.8$. $\beta = 0.5$ indicates that each attribute can take up to 50% of its distinct values in the query. The results indicate that T-Cubes need only milliseconds to respond to even highly complex queries.

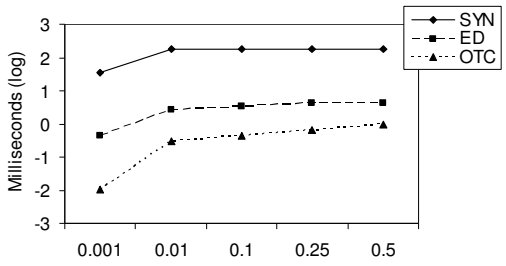


Fig. 4. Query response time for complex queries.

6.2 Comparison with Commercial Tools

We also compared performance of T-Cube against other data cube tools available commerically. Due to privacy concerns we do not list the names of these tools here. But most of these data cube tools are widely used in many practical OLAP applications. The data used for these tests had three demographic attributes with 1,000, 10,

and 5 distinct values respectively, 12 million records of transactions and covered a period of one year. The experiments were performed on a system with 2.4 GHz CPU ands 2 GB memory, running Windows XP operating system.

Table 4 shows the comparison of performance. Each of the commercial tools required a different amount of memory to represent the test data, however for all tools, the response time improves with the increase of the amount of used memory. But still these tools require seconds to respond to a complex query. T-Cube on the other hand is able to respond in milliseconds, i.e. 1,000 times faster than the commercial tools. The two versions of T-Cubes (row 4 & 5) vary on the value of γ (section 5.2.2). The first one uses γ=1000 and in the second was set to 10.

Table 4. Performance comparison of T-Cube with commercial tools

Query Engine	Type	Memory	Response Time
Tool 1	RDMS	330 MB	6.8 sec
Tool 2	In memory	231 MB	7.6 sec
Tool 3	In memory	1+ GB	3.5 sec
T-Cube-1	In memory	236 MB	22 milliseconds
T-Cube-2	In memory	845 MB	5 milliseconds

7 Conclusions

T-Cube is an efficient tool for representing additive time-series data labeled with a set of symbolic attributes. It is especially useful for retrieving responses to ad-hoc queries against large datasets of that kind. We showed that typically a T-Cube performs that task around 1,000 times faster than currently available commercial tools. This efficeny is very useful in both in on-line investigation and statistical data mining application scenarios. Rapid aggregation of time series across large sets of data made posssible by T-Cube, becomes an enabling capability which makes manual lookups as well as many complex analyses feasible. T-Cube can be used as a general tool for any application requesting access to time series data from a database. From the application's perspective it is transparent: it acts just like the database itself, but an incredibly quickly responding one.

The size of memory used by T-Cube can be finely controlled by managing the tree-depth or using the MCV trick. There is trade-off between memory usage and response time. Higher memory usage results in faster response times. With right set of parameters, most datasets showed considerable memory consumption reduction with only linear increase in response time. The datasets we have tested so far were manageable by T-Cube representation fitting in less than 1GB of memory with the majority requiring less than 200MB. This shows that T-Cubes can be used in practice on regular desktop computers and the need for large and expensive servers can be alleviated.

T-Cubes are simple to setup and easy to use. Typically, it takes only minutes to build one from data. Database users don't need to define any stored procedures, or materialized views in order to make that happen. Once a T-Cube is built, it is ready to respond to any simple or complex query. The power of quickly retrieving any ad-hoc query makes T-Cubes very useful in a few specific applications which are known to

the authors. We suspect their potential utility reaches far beyond their original area of deployment, which is bio-surveillance. We are hoping to see T-Cubes widely used.

References

1. J. Han, M. Kamber. Data Mining: Concepts and Techniques. Morgan Kaufmann Publishers, 2000.
2. A. Moore, M. Lee. Cached sufficient statistics for efficient machine learning with large datasets. *Journal of Artificial Intelligence research*, 8, 67-91, 1998.
3. M. Sabhnani, A. Moore, A. Dubrawski. T-Cube: Fast extraction of Time Series from Large Datasets, Technical Report, Carnegie Mellon University, CMU-ML-06-104.
4. M. Sabhnani, D. Neill, A. Moore, F. Tsui, M. Wagner, and J. Espino. Detecting anomalous patterns in pharmacy retail data. *Proceedings of the KDD 2005 Workshop on Data Mining Methods for Anomaly Detection*, 2005.
5. B. Anderson, A. Moore. AD-trees for Fast Counting and for Fast Learning of Association Rules. *Knowledge Discovery from Databases Conference*, 1998.
6. P. Komarek, A. Moore. A Dynamic Adaptation of AD-trees for Efficient Machine Learning on Large Data Sets. *Proceedings of the 17th International Conference on Machine Learning*, 495-502, 2000.
7. J. Gray, A. Bosworth, A. Layman, H. Pirahesh; Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. ICDE 152-159, 1996.
8. N. Roussopoulos, Y. Kotidis, and M. Roussopoulos. Cubetree: organization of and bulk incremental updates on the data cube. *In Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 89-99, Arizona, May 1997.
9. V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. *In Proceedings of ACM SIGMOD '96*, pages 205-216, Montreal, June 1996.
10. M. Wagner, F. Tsui, J. Espino, V. Dato, D. Sittig, R. Caruana, L. McGinnis, D. Deerfield, M. Druzdzel, D. Fridsma. The emerging science of very early detection of disease outbreaks. *Journal of Public Health Management Practice*, Nov;7 (6):51-9, 2001.
11. M. Tsui, J. Espino, M. Wagner. The Timeliness, Reliability, and Cost of Real-time and Batch Chief Complaint Data. RODS Laboratory Technical Report, 2005.