Distributed and Parallel Databases, 2005

DOI: 10.1007/s10619-005-3296-1

Stream Cube: An Architecture for Multi-Dimensional Analysis of Data Streams

JIAWEI HAN hanj@cs.uiuc.edu
University of Illinois

YIXIN CHEN chen@cse.wustl.edu
Washington University, St. Louis

GUOZHU DONG gdong@cs.wright.edu
Wright State University

JIAN PEI jpei@cs.sfu.ca
Simon Fraser University, B. C., Canada

BENJAMIN W. WAH b-wah@uiuc.edu
University of Illinois

JIANYONG WANG jianyong@tsinghua.edu.cn
Tsinghua University, Beijing, China

Y. DORA CAI ycai@ncsa.uiuc.edu
University of Illinois

Recommended by: Ahmed Elmagarmid

Published online: xx xx

Abstract. Real-time surveillance systems, telecommunication systems, and other dynamic environments often generate tremendous (potentially infinite) volume of stream data: the volume is too huge to be scanned multiple times. Much of such data resides at rather low level of abstraction, whereas most analysts are interested in relatively high-level dynamic changes (such as trends and outliers). To discover such high-level characteristics, one may need to perform on-line multi-level, multi-dimensional analytical processing of stream data. In this paper, we propose an architecture, called stream_cube, to facilitate on-line, multi-dimensional, multi-level analysis of stream data.

For fast online multi-dimensional analysis of stream data, three important techniques are proposed for efficient and effective computation of stream cubes. First, a *tilted time frame* model is proposed as a multi-resolution model to register time-related data: the more recent data are registered at finer resolution, whereas the more distant data are registered at coarser resolution. This design reduces the overall storage of time-related data and adapts nicely to the data analysis tasks commonly encountered in practice. Second, instead of materializing cuboids at all levels, we propose to maintain a small number of *critical* layers. Flexible analysis can be efficiently performed based on the concept of *observation layer* and *minimal interesting layer.* Third, an efficient stream data cubing algorithm is developed which computes only the layers (cuboids) along a *popular path* and leaves the other cuboids for query-driven, on-line computation. Based on this design methodology, stream data cube can be constructed and maintained incrementally with a reasonable amount of memory, computation cost, and query response time. This is verified by our substantial performance study.

Stream data cube architecture facilitates online analytical processing of stream data. It also forms a preliminary data structure for online stream data mining. The impact of the design and implementation of stream data cube in the context of stream data mining is also discussed in the paper.

1. Introduction

With years of research and development of data warehousing and OLAP technology [9, 15], a large number of data warehouses and data cubes have been successfully constructed and deployed in applications, and data cube has become an essential component in most data warehouse systems and in some extended relational database systems and has been playing an increasingly important role in data analysis and intelligent decision support.

The data warehouse and OLAP technology is based on the integration and consolidation of data in multi-dimensional space to facilitate powerful and fast on-line data analysis. Data are aggregated either completely or partially in multiple dimensions and multiple levels, and are stored in the form of either relations or multi-dimensional arrays [1, 29]. The dimensions in a data cube are of categorical data, such as products, region, time, etc., and the measures are numerical data, representing various kinds of aggregates, such as *sum, average, variance* of sales or profits, etc.

The success of OLAP technology naturally leads to its possible extension from the analysis of static, pre-integrated, historical data to that of current, dynamically changing data, including time-series data, scientific and engineering data, and data produced in other dynamic environments, such as power supply, network traffic, stock exchange, telecommunication data flow, Web click streams, weather or environment monitoring, etc.

A fundamental difference in the analysis of stream data from that of relational and warehouse data is that the stream data is generated in huge volume, flowing in-and-out dynamically, and changing rapidly. Due to limited memory or disk space and processing power available in today's computers, most data streams may only be examined in a single pass. These characteristics of stream data have been emphasized and investigated by many researchers, such as [6, 7, 12, 14, 16], and efficient stream data querying, clustering and classification algorithms have been proposed recently (such as [12, 14, 16, 17, 20]). However, there is another important characteristic of stream data that has not drawn enough attention: *Most of stream data resides at rather low level of abstraction, whereas an analyst is often more interested in higher and multiple levels of abstraction.* Similar to OLAP analysis of static data, multi-level, multi-dimensional on-line analysis should be performed on stream data as well.

The requirement for multi-level, multi-dimensional on-line analysis of stream data, though desirable, raises a challenging research issue: "*Is it feasible to perform OLAP analysis on huge volumes of stream data since a data cube is usually much bigger than the original data set, and its construction may take multiple database scans?*"

In this paper, we examine this issue and present an interesting architecture for on-line analytical analysis of stream data. Stream data is generated continuously in a dynamic environment, with huge volume, infinite flow, and fast changing behavior. As collected, such data is almost always at rather low level, consisting of various kinds of detailed temporal and other features. To find interesting or unusual patterns, it is essential to perform analysis on some useful measures, such as sum, average, or even more sophisticated measures, such as regression, at certain meaningful abstraction level, discover critical changes of data, and drill down to some more detailed levels for in-depth analysis, when needed.

To illustrate our motivation, let's examine the following examples.

Example 1. A power supply station can watch infinite streams of power usage data, with the lowest granularity as individual user, location, and second. Given a large number of users, it is only realistic to analyze the fluctuation of power usage at certain high levels, such as by city or street district and by quarter (of an hour), making timely power supply adjustments and handling unusual situations.

Conceptually, for multi-dimensional analysis, one can view such stream data as a *virtual* data cube, consisting of one or a few measures and a set of dimensions, including one *time dimension,* and a few other dimensions, such as location, user-category, etc. However, in practice, it is impossible to materialize such a data cube, since the materialization requires a huge amount of data to be computed and stored. Some efficient methods must be developed for systematic analysis of such data.

Example 2. Suppose that a Web server, such as Yahoo.com, receives a huge volume of Web click streams requesting various kinds of services and information. Usually, such stream data resides at rather low level, consisting of time (down to subseconds), Web page address (down to concrete URL), user ip address (down to detailed machine IP address), etc. However, an analyst may often be interested in changes, trends, and unusual patterns, happening in the data streams, at certain high levels of abstraction. For example, it is interesting to find that *the Web clicking traffic in North America on sports in the last 15 minutes is 40% higher than the last 24 hours' average.*

From the point of view of a Web analysis provider, given a large volume of fast changing Web click streams, and with limited resource and computational power, it is only realistic to analyze the changes of Web usage at certain high levels, discover unusual situations, and drill down to some more detailed levels for in-depth analysis, when needed, in order to make timely responses.

Interestingly, both the analyst and analysis provider share a similar view on such stream data analysis: instead of bogging down to every detail of data stream, a demanding request is to provide on-line analysis of changes, trends and other patterns at high levels of abstraction, with low cost and fast response time.

In this study, we take Example 2 as a typical scenario and study how to perform efficient and effective multi-dimensional analysis of stream data, with the following contributions.

1. For on-line stream data analysis, both space and time are critical. In order to avoid imposing unrealistic demand on space and time, instead of computing a fully materialized cube, we suggest to compute a partially materialized data cube, with a *tilted time frame* as its time dimension model. In the *tilted time frame,* time is registered at different levels of granularity. The most recent time is registered at the finest granularity; the more distant time is registered at coarser granularity; the level of coarseness depends on the application requirements and on how old the time point is. This model is sufficient for most analysis tasks, and at the same time it also ensures that the total amount of data to retain in memory or to be stored on disk is small.

2. Due to limited memory space in stream data analysis, it is often too costly to store a precomputed cube, even with the *tilted time frame,* which substantially compresses the storage space. We propose to compute and store only two *critical layers* (which are essentially cuboids) in the cube: (1) an *observation layer,* called *o-layer,* which is the layer that an analyst would like to check and make decisions for either signaling the exceptions or drilling on the exception cells down to lower layers to find their corresponding lower level exceptions; and (2) the *minimal interesting layer,* called *m-layer,* which is the minimal layer that an analyst would like to examine, since it is often neither cost-effective nor practically interesting to examine the minute detail of stream data. For example, in Example 1, we assume that the *o-layer* is *user-region, theme,* and *quarter,* while the *m-layer* is *user, sub-theme,* and *minute*.
3. Storing a cube at only two critical layers leaves a lot of room at what to compute and how to compute for the cuboids between the two layers. We propose one method, called popular-path cubing, which rolls up the cuboids from the *m-layer* to the *o-layer*, by following one popular drilling path, materializes only the layers along the path, and leave other layers to be computed only when needed. Our performance study shows that this method achieves a reasonable trade-off between space, computation time, and flexibility, and has both quick aggregation time and exception detection time.

The rest of the paper is organized as follows. In Section 2, we define the basic concepts and introduce the research problem. In Section 3, we present an architectural design for online analysis of stream data by defining the problem and introducing the concepts of *tilted time frame* and *critical layers*. In Section 4, we present the *popular-path* cubing method, an efficient algorithm for stream data cube computation that supports on-line analytical processing of stream data. Our experiments and performance study of the proposed methods are presented in Section 5. The related work and possible extensions of the model are discussed in Section 6, and our study is concluded in Section 7.

2. Problem definition

In this section, we introduce the basic concepts related to data cubes, multi-dimensional analysis of stream data, and stream data cubes, and define the problem of research.

The concept of data cube [15] was introduced to facilitate multi-dimensional, multi-level analysis of large data sets.

Let $\mathcal{D}$ be a relational table, called the base table, of a given cube. The set of all *attributes* $\mathcal{A}$ in $\mathcal{D}$ are partitioned into two subsets, the *dimensional attributes DIM* and the *measure attributes M* (so $DIM \cup \mathrm{M} = \mathcal{A}$ and $DIM \cap M = \phi$). The measure attributes functionally depend on the dimensional attributes in $\mathcal{DB}$ and are defined in the context of data cube using some typical aggregate functions, such as COUNT, SUM, AVG, or some more sophisticated computational functions, such as standard deviation, regression, etc.

A tuple with schema $\mathcal{A}$ in a multi-dimensional space (i.e., in the context of data cube) is called a **cell**. Given three distinct cells c_1, c_2 and c_3, c_1 is an **ancestor** of c_2, and c_2 a **descendant** of c_1 iff on every dimensional attribute, either c_1 and c_2 share the same value, or c_1's value is a generalized value of c_2's in the dimension's concept hierarchy. c_2 is a

sibling of c_3 iff c_2 and c_3 have identical values in all dimensions except one dimension A where $c_2[A]$ and $c_3[A]$ have the same parent in the dimension's domain hierarchy. A cell which has k non-* values is called a k-d **cell**. (We use "$*$" to indicate "all", i.e., the highest level on any dimension.)

A tuple $c \in \mathcal{D}$ is called a **base cell**. A base cell does not have any descendant. A cell c is an **aggregated cell** iff it is an ancestor of some base cell. For each aggregated cell c, its values on the measure attributes are derived from the complete set of descendant base cells of c. An aggregated cell c is an **iceberg cell** iff its measure value satisfies a specified iceberg condition, such as measure $\geq val_1$. The data cube that consists of all and only the iceberg cells satisfying a specified iceberg condition I is called the **iceberg cube** of a database $\mathcal{D}$ under condition I.

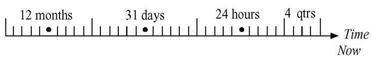
Notice that in stream data analysis, besides the popularly used SQL aggregate-based measures, such as COUNT, SUM, MAX, MIN, and AVG, *regression* is a useful measure. A stream data cell compression technique LCR *(linearly compressed representation)* is developed in [10] to support efficient on-line regression analysis of stream data in data cubes. The study [10] shows that for linear and multiple linear regression analysis, only a small number of *regression measures* rather than the complete stream of data need to be registered. This holds for regression on both the time dimension and the other dimensions. Since it takes a much smaller amount of space and time to handle regression measures in a multi-dimensional space than handling the stream data itself, it is preferable to construct regression (-measured) cubes by computing such regression measures.

A *data stream* is considered as a huge volume, infinite flow of data records, such as Web click streams, telephone call logs, and on-line transactions. The data is collected at the most detailed level in a multi-dimensional space, which may represent time, location, user, theme, and other semantic information. Due to the huge amount of data and the transient behavior of data streams, most of the computations will scan a data stream only once. Moreover, the direct computation of measures at the most detailed level may generate a huge number of results but may not be able to disclose the general characteristics and trends of data streams. Thus data stream analysis will require to consider aggregations and analysis at multi-dimensional and multi-level space.

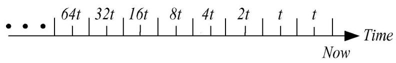
Our task is to support *efficient, high-level, on-line, multi-dimensional analysis of such data streams in order to find unusual (exceptional) changes of trends, according to users' interest, based on multi-dimensional numerical measures*. This may involve construction of a data cube, if feasible, to facilitate on-line, flexible analysis.

3. Architecture for on-line analysis of data streams

To facilitate on-line, multi-dimensional analysis of data streams, we propose a stream_cube architecture with the following features: (1) *tilted time frame,* (2) two *critical layers:* a *minimal interesting layer* and an *observation layer,* and (3) *partial computation of data cubes by popular-path cubing*. The stream data cubes so constructed are much smaller than those constructed from the raw stream data but will still be effective for multi-dimensional stream data analysis tasks.



a) A natural tilted time window model



b) A logarithmic tilted time window model

Frame no.	Snapshots (by clock time)
0	69 67 65
1	70 66 62
2	68 60 52
3	56 40 24
4	48 16
5	64 32

c) A progressive logarithmic tilted time window table

Figure 1. Three models for tilted time windows.

3.1. Tilted time frame

In stream data analysis, people are usually interested in recent changes at a fine scale, but long term changes at a coarse scale. Naturally, one can register time at different levels of granularity. The most recent time is registered at the finest granularity; the more distant time is registered at coarser granularity; and the level of coarseness depends on the application requirements and on how old the time point is (from the current time).

There are many possible ways to design a titled time frame. We adopt three kinds of models: (1) *natural tilted time window model* (figure 1(a)), (2) *logarithmic scale tilted time window model* (figure 1(b)), and (3) *progressive logarithmic tilted time window model* (figure 1(c)).

A *natural tilted time window model* is shown in figure l(a), where the time frame is structured in multiple granularity based on natural time scale: the most recent 4 quarters (15 minutes), then the last 24 hours, 31 days, and 12 months (the concrete scale will be determined by applications). Based on this model, one can compute frequent itemsets in the last hour with the precision of quarter of an hour, the last day with the precision of hour, and so on, until the whole year, with the precision of month.[1] This model registers only $4 + 24 + 31 + 12 = 71$ units of time for a year instead of $366 \times 24 \times 4 = 35{,}136$ units, a saving of about 495 times, with an acceptable trade-off of the grain of granularity at a distant time.

The second choice is *logarithmic tilted time model* as shown in figure l(b), where the time frame is structured in multiple granularity according to a logarithmic scale. Suppose the current window holds the transactions in the current quarter. Then the remaining slots are for the last quarter, the next two quarters, 4 quarters, 8 quarters, 16 quarters, etc., growing at an exponential rate. According to this model, with one year of data and the finest precision at quarter, we will need $\lceil \log_2(365 \times 24 \times 4) + 1 \rceil = 17$ units of time instead of $366 \times 24 \times 4 = 35{,}136$ units. That is, we will just need 17 time frames to store the compressed information.

The third choice is a *progressive logarithmic tilted time frame,* where snapshots are stored at differing levels of granularity depending upon the recency. Snapshots are classified into different *frame number* which can vary from 1 to *max_frame,* where $\log_2(T) - \textit{max-capacity} \leq \textit{max\_frame} \leq \log_2(T)$, *max-capacity* is the maximal number of snapshots held in each frame, and T is the clock time elapsed since the beginning of the stream.

Each snapshot is represented by its timestamp. The rules for insertion of a snapshot t (at time t) into the snapshot frame table are defined as follows: (1) if ($t \bmod 2^i) = 0$ but ($t \bmod 2^{i+1}) \neq 0$, t is inserted into *frame_number* i if $i \leq$ *max_frame*; otherwise (i.e., $i >$ *max_frame*), t is inserted into *max_frame*; and (2) each slot has a *max_capacity* (which is 3 in our example of figure l(c)). At the insertion of t into *frame_number* i, if the slot already reaches its *max_capacity*, the oldest snapshot in this frame is removed and the new snapshot inserted. For example, at time 70, since $(70 \bmod 2^1) = 0$ but $(70 \bmod 2^2) \neq 0$, 70 is inserted into frame-number 1 which knocks out the oldest snapshot 58 if the slot capacity is 3. Also, at time 64, since $(64 \bmod 2^6) = 0$ but *max_frame* $= 5$, so 64 has to be inserted into frame 5. Following this rule, when slot capacity is 3, the following snapshots are stored in the tilted time window table: 16, 24, 32, 40, 48, 52, 56, 60, 62, 64, 65, 66, 67, 68, 69, 70, as shown in figure l(c). From the table, one can see that the closer to the current time, the denser are the snapshots stored.

In the logarithmic and progressive logarithmic models discussed above, we have assumed that the base is 2. Similar rules can be applied to any base α, where α is an integer and $\alpha > 1$. The tilted time models shown above are sufficient for usual time-related queries, and at the same time it ensures that the total amount of data to retain in memory and/or to be computed is small.

Both the natural tilted window model and the progressive logarithmic tilted time window model provide a natural and systematic way for incremental insertion of data in new windows and gradually fading out the old ones. To simplify our discussion, we will only use the natural titled time window model in the following discussions. The methods derived from this time window can be extended either directly or with minor modifications to other time windows.

In our data cube design, we assume that each cell in the base cuboid and in an aggregate cuboid contains a tilted time frame, for storing and propagating measures in the computation. This tilted time window model is sufficient to handle usual time-related queries and mining, and at the same time it ensures that the total amount of data to retain in memory and/or to be computed is small.

3.2. Critical layers

Even with the *tilted time frame* model, it could still be too costly to dynamically compute and store a full cube since such a cube may have quite a few dimensions, each containing multiple levels with many distinct values. Since stream data analysis has only limited memory space but requires fast response time, a realistic arrangement is to compute and store only some mission-critical cuboids in the cube.

In our design, two critical cuboids are identified due to their conceptual and computational importance in stream data analysis. We call these cuboids layers and suggest to compute and store them dynamically. The first layer, called *m-layer*, is the minimally interesting layer that an analyst would like to study. It is necessary to have such a layer since it is often neither cost-effective nor practically interesting to examine the minute detail of stream data. The second layer, called *o-layer*, is the observation layer at which an analyst (or an automated system) would like to check and make decisions of either signaling the

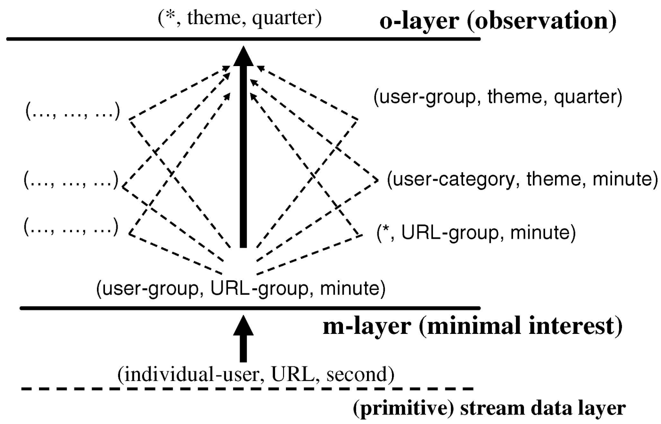


Figure 2. Two critical layers in the stream cube.

exceptions, or drilling on the exception cells down to lower layers to find their lower-level exceptional descendants.

Example 3. Assume that "(*individual-user, URL, second*)" forms the primitive layer of the input stream data in Example 1. With the *tilted time frame* as shown in figure 1, the two critical layers for power supply analysis are: (1) the *m*-layer: (*user_group*, *URL_group*, *minute*), and (2) the *o*-layer: (∗, *theme, quarter*), as shown in figure 2.

Based on this design, the cuboids lower than the *m*-layer will not need to be computed since they are out of the minimal interest of users. Thus the minimal interesting cells that our base cuboid needs to compute and store will be the aggregate cells computed with grouping by *user_group*, *URL_group*, and *minute*. This can be done by aggregations (1) on two dimensions, *user* and *URL,* by rolling up from *individual_user* to *user_group* and from *URL* to *URL_group*, respectively, and (2) on time dimension by rolling up from *second* to *minute*.

Similarly, the cuboids at the *o*-layer should be computed dynamically according to the tilted time frame model as well. This is the layer that an analyst takes as an observation deck, watching the changes of the current stream data by examining the slope of changes at this layer to make decisions. The layer can be obtained by rolling up the cube (1) along two dimensions to ∗ (which means *all* user_category) and *theme,* respectively, and (2) along time dimension to *quarter.* If something unusual is observed, the analyst can drill down to examine the details and the exceptional cells at low levels.

3.3. Partial materialization of stream cube

Materializing a cube at only two critical layers leaves much room for how to compute the cuboids in between. These cuboids can be precomputed fully, partially, not at all (i.e., leave everything computed on-the-fly), or precomputing exception cells only. Let us first examine the feasibility of each possible choice in the environment of stream data. Since there may be a large number of cuboids between these two layers and each may contain many cells, it is often too costly in both space and time to fully materialize these cuboids, especially for stream data. Moreover, for the choice of computing *exception cells* only, the problem becomes how to set up an exception threshold. A too low threshold may lead to computing almost the whole cube, whereas a too high threshold may leave a lot of cells uncomputed and thus not being able to answer many interesting queries efficiently. On the other hand, materializing nothing forces all the aggregate cells to be computed on-the-fly, which may slow down the response time substantially. Thus, it seems that the only viable choice is to perform partial materialization of a stream cube.

According to the above discussion, we propose the following framework in our computation.

Framework 3.1 (*Partial materialization of stream data*). The task of computing a stream data cube is to (1) compute two critical layers (cuboids): (i) *m-layer* (the minimal interest layer), and (ii) *o-layer* (the observation layer), and (2) materialize only a reasonable fraction of the cuboids between the two layers which can allow efficient on-line computation of other cuboids.

Partial materialization of data cubes has been studied in previous works [9, 19]. With the concern of both space and on-line computation time, the partial computation of stream data cube poses more challenging issues than its static counterpart: partial computation of nonstream data cubes, since we have to ensure not only the limited size of the precomputed cube and limited precomputation time, but also efficient online incremental updating upon the arrival of new stream data, as well as fast online drilling to find interesting aggregates and patterns. Obviously, such partial computation should lead to the computation of a rather small number of cuboids, fast updating, and fast online drilling. We will examine how to design such a stream data cube in the next section.

4. Stream data cube computation

From the above analysis, one can see that in order to design an efficient and scalable stream data cube, it is essential to lay out clear design requirements so that we can ensure that the cube can be computed and maintained efficiently in the stream data environment and can provide fast online multidimensional stream data analysis. We have the following design requirements.

1. A stream data cube should be relatively stable in size with respect to infinite data streams. Since a stream data cube takes a set of potentially infinite data streams as inputs, if the size of the base-cuboid grows indefinitely with the size of data streams, the size of stream data cube will grow indefinitely. It is impossible to realize such a stream data cube. Fortunately, with tilted time frame, the distant time is compressed substantially and the very distant data beyond the specified time frame are faded out (i.e., removed) according to the design. Thus the bounded time frames transform infinite data streams into finite, compressed representation, and if the data in the other dimensions of the base cuboid are relatively stable with time, the entire base-cuboid (with the time dimensions included) should be relatively stable in size.

2. A stream data cube should be incrementally updateable with respect to infinite data streams. Since a stream data cube takes potentially infinite data streams as inputs, it is impossible to construct the cube from scratch and the cube must be incrementally updatable. Any cube design that is not incrementally updatable cannot be used as the architecture of a stream cube.
3. The time taken for incremental computation of a stream data cube should be proportional to the size of the incremental portion of the base cuboid of the cube. To incrementally update a stream data cube, one must start from the incremental portion of the base cuboid and use an efficient algorithm to compute it. The time to compute such an incremental portion of the cube should be proportional (desirably, linear) to the size of the incremental portion of the base cuboid of the cube.
4. The stream data cube should facilitate the fast online drilling along any single dimension or along the combination of a small number of dimensions. Although it is impossible to materialize all the cells of a stream cube, it is expected that the drilling along a single dimension or along the combination of a small number of dimensions be fast. Materialization of some portion of the cube will facilitate such fast online presentation.

Based on the above design requirements, we examine the methods for the efficient computation of stream cubes.

4.1. Design of stream cube architecture: A popular path architecture

According to our discussion in Section 3, there are three essential components in a stream data cube: (1) *tilted time frame,* (2) two *critical layers:* a *minimal interesting layer* and an *observation layer,* and (3) *partial computation of data cubes.*

In data cube computation, iceberg cube [8] which stores only the aggregate cells that satisfy an iceberg condition has been used popularly as a data cube architecture since it may substantially reduce the size of a data cube when data is sparse. In stream data analysis, people may often be interested in only the substantially important or exceptional cube cells, and such important or exceptional conditions can be formulated as typical *iceberg conditions.* Thus it seems that iceberg cube could be an interesting model for stream cube architecture. Unfortunately, iceberg cube cannot accommodate the incremental update with the constant arrival of new data and thus cannot be used as the architecture of stream data cube. We have the following observation.

Framework 4.1 (*No iceberg cubing for stream data*). The iceberg cube model does not fit the stream data cube architecture. Nor does the exceptional cube model.

Rationale. With the incremental and gradual arrival of new stream data, as well as the incremental fading of the obsolete data from the time scope of a data cube, it is required that incremental update be performed on such a stream data cube. It is unrealistic to constantly recompute the data cube from scratch upon incremental updates due to the tremendous cost of recomputing the cube on the fly. Unfortunately, such an incremental model does not fit the iceberg cube computation model due to the following observation: Let a cell "$\langle d_i, \ldots, d_k \rangle$: m_{ik}" represent a $k - i + 1$ dimension cell with $d_i, \ldots, d_k$ as its corresponding dimension values and m_{ik} as its measure value. If $SAT(m_{ik}, iceberg\_cond)$ is false, i.e., m_{ik} does not satisfy the iceberg condition, the cell is dropped from the iceberg cube. However, at a later time slot t', the corresponding cube cell may get a new measure m'_{ik} related to t'. Since m_{ik} has been dropped at a previous instance of time due to its inability to satisfy the iceberg condition, the new measure for this cell cannot be calculated correctly without such information. Thus one cannot use the iceberg architecture to model a stream data cube unless recomputing the measure from the based cuboid upon each update. Similar reasoning can be applied to the case of exceptional cell cubes since the exceptional condition can be viewed as a special iceberg condition.

Since iceberg cube cannot be used as a stream cube model, but materializing the full cube is too costly both in computation time and storage space, we propose to compute only a *popular path* of the cube as our partial computation of stream data cube, as described below.

Based on the notions of the minimal interesting layer (the *m*-layer) and the tilted time frame, stream data can be directly aggregated to this layer according to the tilted time scale. Then the data can be further aggregated following one popular drilling path to reach the observation layer. That is, the *popular path* approach computes and maintains a single popular aggregation path from *m*-layer to *o*-layer so that queries directly on those (layers) along the popular path can be answered without further computation, whereas those deviating from the path can be answered with minimal online computation from those reachable from the computed layers. Such cost reduction makes possible the OLAP-styled exploration of cubes in stream data analysis.

To facilitate efficient computation and storage of the popular path of the stream cube, a compact data structure needs to be introduced so that the space taken in the computation of aggregations is minimized. A data structure, called H-tree, a hyper-linked tree structure introduced in [18], is revised and adopted here to ensure that a compact structure is maintained in memory for efficient computation of multi-dimensional and multi-level aggregations.

We present these ideas using an example.

Example 4. Suppose the stream data to be analyzed contains 3 dimensions, A, B and C, each with 3 levels of abstraction (excluding the highest level of abstraction "$*$"), as (A_1, A_2, A_3), (B_1, B_2, B_3), (C_1, C_2, C_3), where the ordering of "$* > A_1 > A_2 > A_3$" forms a high-to-low hierarchy, and so on. The minimal interesting layer (the *m*-layer) is (A_2, B_2, C_2), and the *o*-layer is $(A_1, *, C_1)$. From the *m*-layer (the bottom cuboid) to the *o*-layer (the top-cuboid to be computed), there are in total $2 \times 3 \times 2 = 12$ cuboids, as shown in figure 3.

Suppose that the popular drilling path is given (which can usually be derived based on domain expert knowledge, query history, and statistical analysis of the sizes of intermediate cuboids). Assume that the given popular path is $\langle (A_1, *, C_1) \rightarrow (A_1, *, C_2) \rightarrow (A_2, *, C_2) \rightarrow (A_2, B_1, C_2) \rightarrow (A_2, B_2, C_2)\rangle$, shown as the darkened path in figure 3. Then each path of an H-tree from root to leaf is ordered the same as the popular path.

This ordering generates a compact tree because the set of low level nodes that share the same set of high level ancestors will share the same prefix path using the tree structure. Each tuple, which represents the currently in-flow stream data, after being generalized to the *m*-layer, is inserted into the corresponding path of the H-tree. An example H-tree is shown in figure 4. In the leaf node of each path, we store relevant measure information of

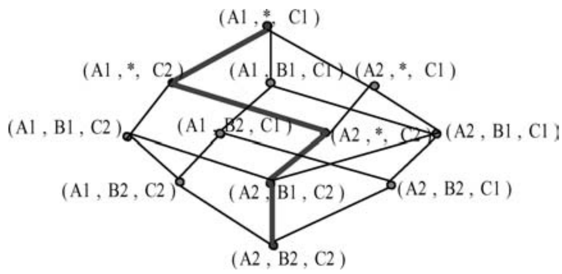


Figure 3. Cube structure from the *m*-layer to the *o*-layer.

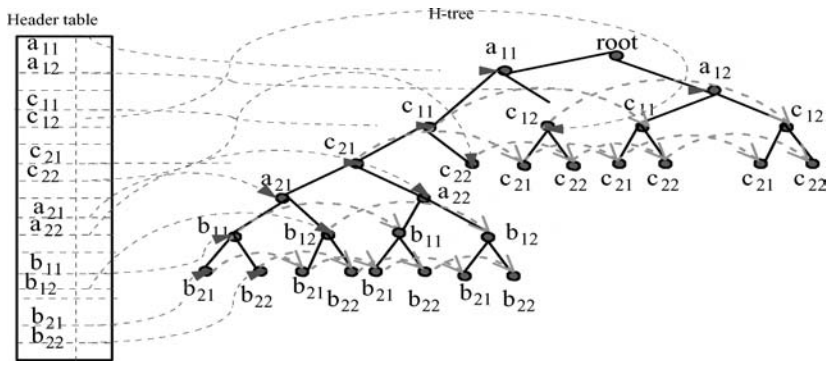


Figure 4. H-tree structure for cube computation.

the cells of the *m*-layer. The measures of the cells at the upper layers are computed using the H-tree and its associated links.

An obvious advantage of the *popular path approach* is that the nonleaf nodes represent the cells of those layers (cuboids) along the popular path. Thus these nonleaf nodes naturally serve as the cells of the cuboids along the path. That is, it serves as a data structure for intermediate computation as well as the storage area for the computed measures of the layers (i.e., cuboids) along the path.

Furthermore, the H-tree structure facilitates the computation of other cuboids or cells in those cuboids. When a query or a drill-down clicking requests to compute cells outside the popular path, one can find the closest lower level computed cells and use such intermediate computation results to compute the measures requested, because the corresponding cells can be found via a linked list of all the corresponding nodes contributing to the cells.

4.2. Algorithms for cube measure computation

With popular path stream data cube design and the H-tree data structure, the popular-path-based stream data cubing can be partitioned into three stages: (1) the initial computation of (partially materialized) stream data cube by popular-path approach, (2) incremental update of stream data cube, and (3) online query answering with the popular-path-based stream data cube.

Here we present the three corresponding algorithms, one for each stage of the popular-path-based stream data cubing.

First, we present an algorithm for computation of initial (partially materialized) stream data cube by popular-path approach.

Algorithm 1 (Popular-path-based stream cube computation). Computing initial stream cube, i.e., the cuboids along the *popular-path* between the m-layer and the o-layer, based on the currently collected set of input stream data.

Input. (1) multi-dimensional multi-level stream data (which consists of a set of tuples, each carrying the corresponding time stamps), (2) the m and o-layer specifications, and (3) a given popular drilling path.

Output. All the aggregated cells of the cuboids along the popular path between the m- and o- layers.

Method.

1. Each tuple, which represents a minimal addressing unit of multi-dimensional multilevel stream data, is scanned once and generalized to the m-layer. The generalized tuple is then inserted into the corresponding path of the H-tree, increasing the count and aggregating the measure values of the corresponding leaf node in the corresponding slot of the tilted time frame.
2. Since each branch of the H-tree is organized in the same order as the specified popular path, aggregation for each corresponding slot in the tilted time frame is performed from the m-layer all the way up to the o-layer by aggregating along the popular path. The step-by-step aggregation is performed while inserting the new generalized tuples in the corresponding time slot.
3. The aggregated cells are stored in the nonleaf nodes in the H-tree, forming the computed cuboids along the popular path.

Analysis. The H-tree ordering is based on the popular drilling path given by users or experts. This ordering facilitates the computation and storage of the cuboids along the path. The aggregations along the drilling path from the m-layer to the o-layer are performed during the generalizing of the stream data to the m-layer, which takes only one scan of stream data. Since all the cells to be computed are the cuboids along the popular path, and the cuboids to be computed are the nonleaf nodes associated with the H-tree, both space and computation overheads are minimized.

Second, we discuss how to perform incremental update of the stream data cube in the popular-path cubing approach. Here we deal with the "always-grow" nature of time-series stream data in an "on-line," continuously growing manner.

The process is essentially an incremental computation method illustrated below, using the tilted time frame of figure 1. Assuming that the memory contains the previously computed m and o-layers, plus the cuboids along the popular path, and stream data arrive every second. The new stream data are accumulated (by generalization) in the corresponding H-tree leaf nodes. If the time granularity of the m-layer is minute, at the end of every minute, the data will be aggregated and be rolled up from leaf to the higher level cuboids. When reaching a cuboid whose time granularity is quarter, the rolled measure information remains in the corresponding minute slot until it reaches the full quarter (i.e., 15 minutes) and then it rolls up to even higher levels, and so on.

Notice in this process, the measure in the time interval of each cuboid will be accumulated and promoted to the corresponding coarser time granularity, when the accumulated data reaches the corresponding time boundary. For example, the measure information of every four quarters will be aggregated to one hour and be promoted to the hour slot, and in the mean time, the quarter slots will still retain sufficient information for quarter-based analysis. This design ensures that although the stream data flows in-and-out, measure always keeps up to the most recent granularity time unit at each layer.

We outline the incremental algorithm of the method as follows.

Algorithm 2 (Incremental update of popular-path stream cube with incoming stream data). Incremental computing stream cube, i.e., the cuboids along the *popular-path* between the *m*-layer and the *o*-layer, based on the previously computed cube and the newly input stream data.

Input. (1) a popular path-based stream data cube, which also includes (i) the *m* and *o-layer* specifications, and (ii) a given popular drilling path, and (2) a set of input multi-dimensional multi-level stream data (which consists of a set of tuples, each carrying the corresponding time stamps).

Output An updated stream data cube (i.e., the updated popular-path cuboids (between the *m*- and *o*-layers).

Method.

1. Each newly coming tuple, which represents a minimal addressing unit of multi-dimensional multi-level stream data, is scanned once and generalized to the *m*-layer. The generalized tuple is then inserted into the corresponding path of the H-tree. If there exists a corresponding leaf node in the tree, increase the count and aggregating the measure values of the corresponding leaf node in the corresponding slot of the tilted time frame. If there exists no corresponding leaf node in the tree, a new leaf node is created in the corresponding path of the H-tree.
2. Since each branch of the H-tree is organized in the same order as the specified popular path, aggregation for each corresponding slot in the tilted time frame is performed from the *m*-layer all the way up to the *o*-layer by aggregating along the popular path. The step-by-step aggregation is performed while inserting the new generalized tuples finishes.
3. If it reaches the time when a sequence of data in the lower-level time slots should be aggregated to a new slot in the corresponding higher level titled time window, such aggregation will be performed at each level of the popular path. If it reaches the time when the data in the most distant time slot should be dropped from the valid time scope, the slot in the corresponding time window will be cleared.
4. The so computed aggregated cells are stored in the nonleaf nodes in the H-tree, forming the computed cuboids along the popular path.

Analysis. Based on our design of the tilted time window, such incremental computation can be performed along the popular path of the H-tree. Moreover, the aggregations along the drilling path from the *m*-layer to the *o*-layer are performed when the input stream data come to the *m*-layer, which takes only one scan of stream data. Since all the cells in the titled time windows in the cuboids along the popular path are incrementally updated, the cuboids so computed are correctly updated stream cube, with minimal space and computation over-heads.

Finally, we examine how fast online computation can be performed with such a partially materialized popular-path data cube. Since the query inquiring the information completely contained in the popular-path cuboids can be answered by directly retrieving the information stored in the popular-path cuboids, our discussion here will focus on retrieving the information involving the aggregate cells not contained in the popular-path cuboids.

A multi-dimensional multi-level stream query usually provides a few instantiated constants and inquires information related to one or a small number of dimensions. Thus one can consider a query involving a set of instantiated dimensions, $\{D_{ci}, \ldots, D_{cj}\}$, and a set of inquired dimensions, $\{D_{ql}, \ldots, D_{qk}\}$. The set of relevant dimensions, D_r, is the union of the sets of instantiated dimensions and the inquired dimensions. For maximal use of the precom-puted information available in the popular path cuboids, one needs to find the highest-level popular path cuboids that contains D_r. If one cannot find such a cuboid in the path, one will use the *base cuboid* at the *m*-layer to compute it. Then the computation can be performed by fetching the relevant data set from the so found cuboid and then computing the cuboid consisting of the inquired dimensions.

The online OLAP stream query processing algorithm is presented as follows.

Algorithm 3 (Online processing of stream OLAP query). Online processing of stream OLAP query given the precomputed stream data cube, i.e., the cuboids along the *popular-path* between the *m*-layer and the *o*-layer.

Input. (1) a popular path-based stream data cube, which includes (i) the *m* and *o*-layer specifications, and (ii) a given popular drilling path, and (2) a given query whose relevant dimension set is D_r, which in turn consists of a set of instantiated dimensions, $\{D_{ci}, \ldots, D_{cj}\}$, and a set of inquired dimensions, $\{D_{qi}, \ldots, D_{qk}\}$.
Output. A computed cuboid related to the stream OLAP query.
Method.

1. Find the highest-level popular path cuboids that contains D_r. If one cannot find such a cuboid in the path, one will use the *base cuboid* at the *m*-layer to compute it. Let the found cuboid be S.
2. Perform selection on S using the set of instantiated dimensions as set of constants, and using the set of inquired dimensions as projected attributed. Let S_c be the set of multidimensional data so selected.
3. Perform on line cubing on S_c and return the result.

Analysis. Based on our design of the stream data cube, the highest-level popular path cuboid that contains D_r should contain the answers we want. Using the set of instantiated dimensions as set of constants, and using the set of inquired dimensions as projected attributed, the so-obtained S_c is the minimal set of aggregated data set for answering the query. Thus online cubing on this set of data will derive the correct result. Obviously, such a computation process makes good use of the precomputed cuboids and will involve small space and computation overheads.

5. Performance study

To evaluate the effectiveness and efficiency of our proposed stream cube and OLAP computation methods, we performed an extensive performance study on synthetic datasets. Our result shows that the total memory and computation time taken by the proposed algorithms are small, in comparison with several other alternatives, and it is realistic to compute such a partially aggregated cube, incrementally update them, and perform fast OLAP analysis of stream data using such precomputed cube.

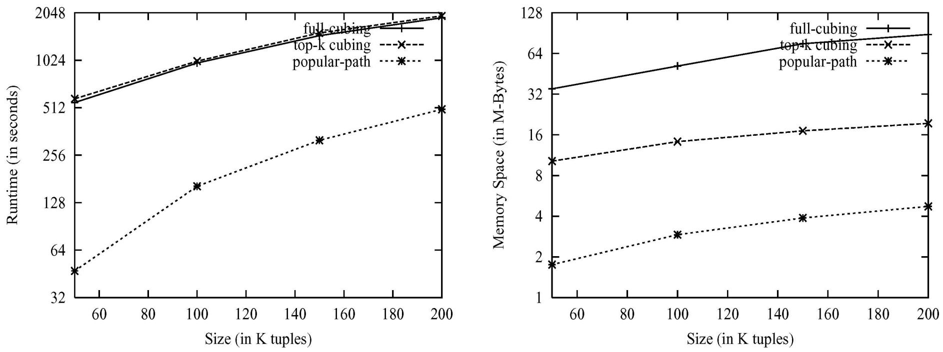
Here we report our performance studies with synthetic data streams of various characteristics.[2] The data stream is generated by a data generator similar in spirit to the IBM data generator [5] designed for testing data mining algorithms. The convention for the data sets is as follows: $D3L3C10T400K$ means there are 3 dimensions, each dimension contains 3 levels (from the *m*-layer to the *o*-layer, inclusive), the node fan-out factor (cardinality) is 10 (i.e., 10 children per node), and there are in total 400 K merged *m*-layer tuples.

Notice that all the experiments are conducted in a static environment as a simulation of the online stream processing. This is because the cube computation, especially for full cube and top-k cube, may take much more time than the stream flow allows. If this is performed in the online streaming environment, substantial amount of stream data could have been lost due to the slow computation of such data cubes. This simulation serves our purpose since it clearly demonstrates the cost and the possible delays of stream cubing and indicates what could be the realistic choice if they were put in a dynamic streaming environment.

All experiments were conducted on a 2 GHz Pentium PC with 1 GB main memory, running Microsoft Windows-XP Server. All the methods were implemented using Sun Microsystems' Java 2 Platform, Standard Edition, version 1.4.2.

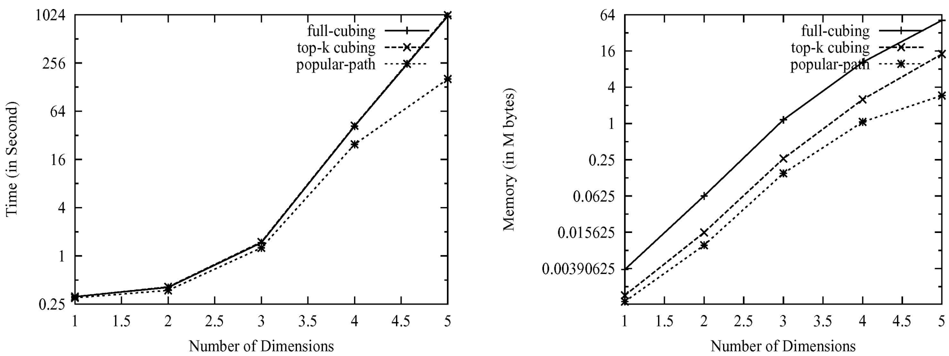
Our design framework has some obvious performance advantages over some alternatives in a few aspects, including (1) *tilted time frame* vs. *full non-tilted time frame,* (2) *using minimal interesting layer* vs. *examining stream data at the raw data layer,* and (3) *computing the cube up to the apex layer* vs. *computing it up to the observation layer.* Consequently, our feasibility study will not compare the design that does not have such advantages since they will be obvious losers.

Since a data analyst needs fast on-line response, and both space and time are critical in processing, we examine both time and space consumption. In our study, besides presenting the total time and memory taken to compute and store such a stream cube, we compare the two measures (time and space) of the *popular path* approach against two alternatives: (1) the *full-cubing* approach, i.e., materializing all the cuboids between the *m*- and *o*-layers, and (2) the *top-k cubing* approach, i.e., materializing only the top-k measured cells of the cuboids between the *m*- and *o*-layers, and we set top-k threshold to be 10%, i.e., only top 10% (in measure) cells will be stored at each layer (cuboid). Notice that top-k cubing cannot



a) Time vs. size b) Space vs. size

Figure 5. Cube computation: time and memory usage vs. no. tuples at the m-layer for the data set $D5L3C10$.



a) Time vs. # dimensions b) Space vs. # dimensions

Figure 6. Cube computation: Time and space vs. no. of dimensions for the data set $L3C10I100K$.

be used for incremental stream cubing. However, since people may like to pay attention only to top-k cubes, we still put it into our performance study (as initial cube computation). From the performance results, one can see that if top-k cubing cannot compete with the popular path approach, with its difficulty at handling incremental updating, it will not likely be a choice for stream cubing architecture.

The performance results of stream data cubing (cube computation) are reported from figures 5 to 7.

Figure 5 shows the processing time and memory usage for the three approaches, with increasing size of the data set, where the size is measured as the number of tuples at the m-layer for the data set $D5L3C10$. Since *full-cubing* and *top-k cubing* compute all the cells from the m-layer all the way up to the o-layer, their total processing time is much higher than popular-path. Also, since *full-cubing* saves all the cube cells, its space consumption is much higher than popular-path. The memory usage of *top-k cubing* falls in between of the two approaches, and the concrete amount will depend on the k value.

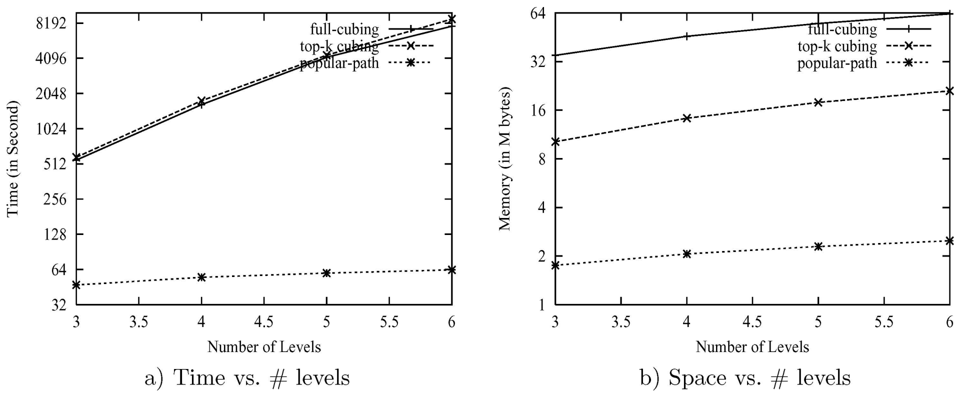


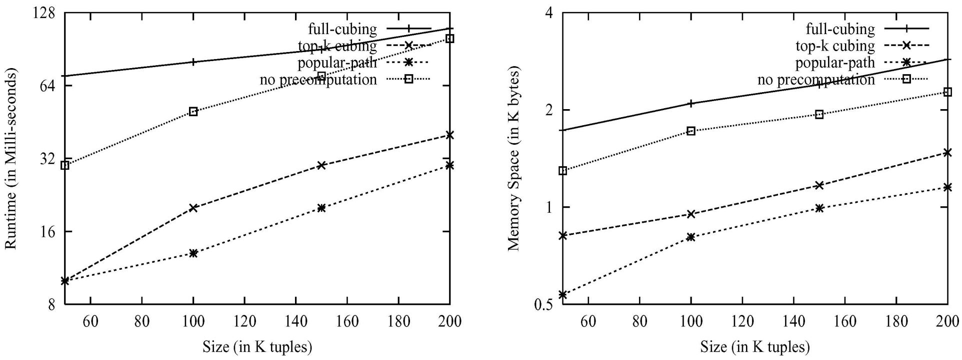
Figure 7. Cube computation: Time and space vs. no. of levels for the data set $D5C10T50K$. (a)Time vs. no. levels. (b) Space vs. no. levels.

Figure 6 shows the processing time and memory usage for the three approaches, with an increasing number of dimensions, for the data set $L3C10T100K$. figure 7 shows the processing time and memory usage for the three approaches, with an increasing number of levels, for the data set $D5C10T50K$. The performance results show that popular-path is more efficient than both *full-cubing* and *top-k cubing* in computation time and memory usage. Moreover, one can see that increment of dimensions has a much stronger impact on the computation cost (both time and space) in comparison with the increment of levels.

Since incremental update of stream data cube carries the similar comparative costs for both popular-path and *full-cubing* approaches, and moreover, *top-k cubing* is inappropriate for incremental updating, we will not present this part of performance comparison. Notice that for incrementally computing the newly generated stream data, the computation time should be shorter than that shown here due to less number of cells involved in computation although the total memory usage may not reduce due to the need to store data in the layers along the popular path between two critical layers in the main memory.

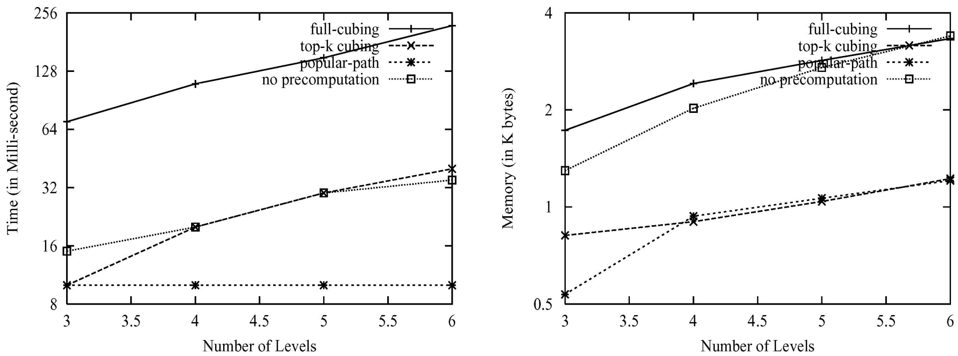
Here we proceed to the performance study of stream query processing with four different approaches: (1) *full-cubing,* (2) *top-k cubing,* (3) popular-path, and (4) *no precomputation,* which computes the query and answer it on the fly. The reason that we added the fourth one is because one can compute query results without using any precomputed cube but using only the base cuboid: the set of merged tuples at the *m*-layer.

Figure 8 shows the processing time and memory usage vs. the size of the base cuboid, i.e., the number of merged tuples at the *m*-layer, for the data set $D5L3C10$, with the data set grows from 50 to 200 K tuples. There are 5 dimensions in the cube, and the query contains two instantiated columns and one inquired column. The performance results show that popular-path costs the least amount of time and space although *top-k cubing* could be a close rival. Moreover, *no precomputation,* though more costly then the previous two, still costs less in both time and space than the fully materialized stream cube at query processing.



a) Time vs. # size b) Space vs. # size

Figure 8. Stream query processing: Time and space vs. no. of tuples at the *m*-layer.

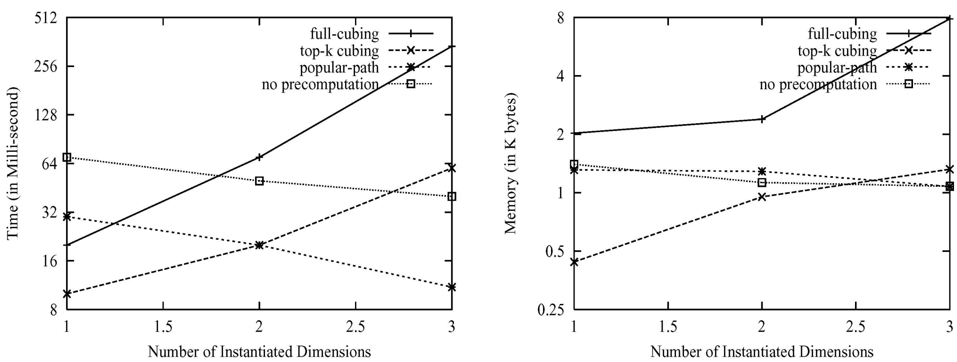


a) Time vs. # of levels b) Space vs. # of levels

Figure 9. Stream query processing: Time and space vs. no. of levels.

Figure 9 shows the processing time and memory usage vs. the number of levels from the *m* to *o* layers, for the data set $D5C10T50K$, with the number of levels grows from 3 to 6. There are 5 dimensions in the cube, and the query contains two instantiated columns and one inquired column. The performance results show that popular-path costs the least amount of time and space and its query processing cost is almost irrelevant to the number of levels (but mainly relevant to the size of the tuples) with slightly increased memory usages. Moreover, *top-k cubing* and *no precomputation* takes more time and space when the number of levels increases. However, *full-cubing* takes the longest time to respond to a similar query although its response time is still in the order of 200 millisecond.

Finally, figure 10 shows the processing time and memory usage vs. the number of instantiated dimensions where the number of inquired dimensions maintains at one (i.e., single dimension) for the data set $D5L3C10T100K$. Notice that with more instantiated dimensions, the query processing cost for popular-path and *no precomputation* is actually



a) Time vs. # of instantiated dimensions   b) Space vs. # of instantiated dimensions

Figure 10. Stream query processing: Time and space vs. no. of instantiated dimension.

dropping because it will search less space in the H-tree or in the base cuboid with more instantiated constants. Initially (when the number of instantiated dimensions is only one, the *full-cubing* and *top-k cubing* are slightly faster than popular-path since the latter (popular-path) still needs some online computation while the former can fetch from the precomuted cubes.

From this study, one can see that popular-path is an efficient and feasible method for computing multi-dimensional, multi-level stream cubes, whereas *no precomputation* which computes only the base cuboid at the *m*-layer, could be the second choice. The full-cubing is too costly in both space and time, whereas *top-k cubing* is not a good candidate because it cannot handle incremental updating of a stream data cube.

6. Discussion

In this section, we compare our study with the related work and discuss some possible extensions.

6.1. Related work

Our work is related to: (1) on-line analytical processing and mining in data cubes, and (2) research into management and mining of stream data. We briefly review previous research in these areas and point out the differences from our work.

In data warehousing and OLAP, much progress has been made on the efficient support of standard and advanced OLAP queries in data cubes, including selective cube materialization [19], iceberg cubing [8, 18, 26, 28], cube gradient analysis [11, 21], exception [24], intelligent roll-up [25], and high-dimensional OLAP analysis [22]. However, previous studies do not consider the support for stream data, which needs to handle huge amount of fast changing stream data and restricts that the a data stream can be scanned only once. In contrast, our work considers complex measures in the form of stream data and studies OLAP and mining over partially materialized stream data cubes. Our data structure, to certain extent, extend the previous work on H-tree and H-cubing [18]. However, instead of computing a materialized data cube as in H-cubing, we only use the H-tree structure to store a small number of cuboids along the popular path. This will save a substantial amount of computation time and storage space and lead to high performance in both cube computation and query processing. We have also studied whether it is appropriate to use other cube structures, such as star-trees in StarCubing [28], dense-sparse partitioning in MM-cubing [26] and shell-fragments in high-dimensional OLAP [22]. Our conclusion is that H-tree is still the most appropriate structure since most other structures need to either scan data sets more than once or know the sparse or dense parts beforehand, which does not fit the single-scan and dynamic nature of data streams.

Recently, there have been intensive studies on the management and querying of stream data [7, 12, 14, 16], and data mining (classification and clustering) on stream data [2–4, 13, 17, 20, 23, 27]. Although such studies lead to deep insight and interesting results on stream query processing and stream data mining, they do not address the issues of multidimensional, online analytical processing of stream data. Multidimensional stream data analysis is an essential step to understand the general statistics, trends and outliers as well as other data characteristics of online stream data and will play an essential role in stream data analysis. This study sets a framework and outlines an interesting approach to stream cubing and stream OLAP, and distinguishes itself from the previous work on stream query processing and stream data mining.

In general, we believe that this study sets a new direction: *extending data cube technology for multi-dimensional analysis of data streams*. This is a promising direction with many applications.

6.2. Possible extensions

There are many potential extensions of this work towards comprehensive, high performance analysis of data streams. Here we outline a few.

First, parallel and distributed processing can be used to extend the proposed algorithms in this promising direction to further enhance the processing power and the performance of the system. All of the three algorithms proposed in this study: *initial computation of stream data cubes, incremental update of stream data cube,* and *online multidimensional analysis of stream data,* can be handled by different processors and processed in a parallel and/or distributed manner. In the fast data streaming environment, it is desirable or sometimes required to have at least one processor dedicated to stream query processing (on the computed data cube) and at least another one dedicated to incremental update of data streams. Moreover, both incremental update and query processing can be processed by parallel processors as well since the algorithms can be easily transformed into parallel and/or distributed algorithms.

Second, although a stream cube usually retains in main memory for fast computation, updating, and accessing, it is important to have its important or substantial portion stored or mirrored on disk, which may enhance data reliability and system performance. There are several ways to do it. Based on the design of the tilted time frame, the distant time portion in the data cube can be stored on the disk. This may help reduce the total main memory requirement and the update overhead. The incremental propagation of data in such distant portion can be done by other processors using other memory space. Alternatively, to ensure the data is not lost in case of system error or power failure, it is important to keep a mirror copy of the stream data cube on disk. Such a mirroring process can be processed in parallel by other processors. In addition, it is possible that a stream cube may miss a period of data due to software error, equipment malfunction, system failure, or other unexpected reasons. Thus a robust stream data cube should build the functionality to run despite the missing of a short period of data in the tilted time frame. The data so missed can be treated by special routines, like data smoothing, data cleaning, or other special handling so that the overall stream data can be interpreted correctly without interruption.

Third, although we did not discuss the computation of complex measures in the data cube environment, it is obvious that complex measures, such as sum, avg, min, max, last, standard deviation, linear regression and many other measures can be handled for the stream data cube in the same manner as discussed in this study. However, it is not clear how to handle holistic measures [15] in the stream data cubing environment. For example, it is still not clear how some holistic measures, such as quantiles, rank, median, and so on, can be computed efficiently in this framework. This issue is left for future research.

Fourth, the stream data that we discussed here are of simple numerical and categorical data types. In many applications, stream data may contain spatiotemporal and multimedia data. For example, monitoring moving vehicles and the flow of people in the airport may need to handle spatiotemporal and multimedia data. It is an open problem how to perform online analytical processing of multidimensional spatiotemporal and multimedia data in the context of data streams. We believe that spatiotemporal and multimedia analysis techniques should be integrated with our framework in order to make good progress in this direction.

Fifth, this study has been focused on multiple dimensional analysis of stream data. However, the framework so constructed, including tilted time dimension, monitoring the change of patterns in a large data cube using an *m*-layer and an *o*-layer, and paying special attentions on exception cells, is applicable to the analysis of non-stream time-series data as well.

Finally, this study is on multidimensional OLAP stream data analysis. Many data mining tasks require deeper analysis than simple OLAP analysis, such as classification, clustering and frequent pattern analysis. In principle, the general framework worked out in this study, including tilted time frame, minimal generalized layer and observation layers, as well as partial precomputation for powerful online analysis, will be useful for in-depth data mining methods. It is an interesting research theme on how to extend this framework towards online stream data mining.

7. Conclusions

In this paper, we have promoted on-line analytical processing of stream data, and proposed a feasible framework for on-line computation of multi-dimensional, multi-level stream cube.

We have proposed a general stream cube architecture and a stream data cubing method for on-line analysis of stream data. Our method uses a *tilted time frame,* explores *minimal* *interesting and observation layers,* and adopts a *popular path approach* for efficient computation and storage of stream cube to facilitate OLAP analysis of stream data. Our performance study shows that the method is cost-efficient and is a realistic approach based on the current computer technology. Recently, this stream data cubing methodology has been successfully implemented in the MAIDS (Mining Alarming Incidents in Data Streams) project at NCSA (National Center for Supercomputing Applications) at University of Illinois, and its effectiveness has been tested using online stream data sets.

Our proposed stream cube architecture shows a promising direction for realization of on-line, multi-dimensional analysis of data streams. There are a lot of issues to be explored further. For example, besides H-cubing [18], there are other data cubing methodologies, such as multi-way array aggregation [29], BUC [8], and Star-cubing [28], it is interesting to examine other alternative methods for efficient online analysis of stream data. Furthermore, we believe that a very important direction is to further develop data mining methods to take advantage of multi-dimensional, multi-level stream cubes for single-scan on-line mining to discover knowledge in stream data.

Acknowledgments

The work was supported in part by research grants from U.S. National Science Foundation grants IIS-02–9199 and IIS-03-08215, Office of Naval Research, Natural Science and Engineering Research Council of Canada, and the University of Illinois. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies. This paper is a substantially revised and major value-added version of a paper, *"Multi-Dimensional Regression Analysis of Time-Series Data Streams,"* by *Yixin Chen, Guozhu Dong, Jiawei Han, Benjamin W. Wah,* and *Jianyong Wang,* in VLDB'2002, Hong Kong, China, August, 2002.

Notes

1. We align the time axis with the natural calendar time. Thus, for each granularity level of the tilt time frame, there might be a partial interval which is less than a full unit at that level.
2. We also tested it for some industry data sets and got similar performance results. However, we cannot discuss the results here due to the confidentiality of the data sets.

References

1. S. Agarwal, R. Agrawal, P.M. Deshpande, A. Gupta, J.F. Naughton, R. Ramakrishnan, and S. Sarawagi, "On the computation of multidimensional aggregates," in Proc. 1996 Int. Conf. Very Large Data Bases (VLDB'96), Bombay, India, Sept. 1996, pp. 506–521.
2. C. Aggarwal, J. Han, J. Wang, and P.S. Yu, "A framework for projected clustering of high dimensional data streams," in Proc. 2004 Int. Conf. Very Large Data Bases (VLDB'04). Toronto, Canada, Aug. 2004, pp. 852–863.
3. C. Aggarwal, J. Han, J. Wang, and P.S. Yu, "On demand classification of data streams," in Proc. 2004 Int. Conf. Knowledge Discovery and Data Mining (KDD'04), Seattle, WA, Aug. 2004, pp. 503–508.
4. C.C. Aggarwal, J. Han, J. Wang, and P.S. Yu, "A framework for clustering evolving data streams," in Proc. 2003 Int. Conf. Very Large Data Bases (VLDB'03), Berlin, Germany, Sept. 2003.

5. R. Agrawal and R. Srikant, "Mining sequential patterns," in Proc. 1995 Int. Conf. Data Engineering (ICDE'95), Taipei, Taiwan, March 1995, pp. 3–14.
6. B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in Proc. 2002 ACM Symp. Principles of Database Systems (PODS'02), Madison, WI, June 2002, pp. 1–16.
7. S. Babu and J. Widom, "Continuous queries over data streams," SIGMOD Record, vol. 30, pp. 109–120, 2001.
8. K. Beyer and R. Ramakrishnan, "Bottom-up computation of sparse and iceberg cubes," in Proc. 1999 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'99), Philadelphia, PA, June 1999, pp. 359–370.
9. S. Chaudhuri and U. Dayal, "An overview of data warehousing and OLAP technology," SIGMOD Record, vol. 26, pp. 65–74, 1997.
10. Y. Chen, G. Dong, J. Han, B.W. Wah, and J. Wang, "Multi-dimensional regression analysis of time-series data streams," in Proc. 2002 Int. Conf. Very Large Data Bases (VLDB'02), Hong Kong, China, Aug. 2002, pp. 323–334.
11. G. Dong, J. Han, J. Lam, J. Pei, and K. Wang, "Mining multi-dimensional constrained gradients in data cubes," in Proc. 2001 Int. Conf. on Very Large Data Bases (VLDB'01), Rome, Italy, Sept. 2001, pp. 321–330.
12. J. Gehrke, F. Korn, and D. Srivastava, "On computing correlated aggregates over continuous data streams," in Proc. 2001 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'01), Santa Barbara, CA, May 2001, pp. 13–24.
13. C. Giannella, J. Han, J. Pei, X. Yan, and P.S. Yu, "Mining frequent patterns in data streams at multiple time granularities," in H. Kargupta, A. Joshi, K. Sivakumar, and Y. Yesha (eds), Data Mining: Next Generation Challenges and Future Directions. AAAI/MIT Press, 2004.
14. A.C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss, "Surfing wavelets on streams: One-pass summaries for approximate aggregate queries," in Proc. 2001 Int. Conf. on Very Large Data Bases (VLDB'01), Rome, Italy, Sept. 2001, pp. 79–88.
15. J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh, "Data cube: A relational aggregation operator generalizing group-by, cross-tab and sub-totals," Data Mining and Knowledge Discovery, vol. 1, pp. 29–54, 1997.
16. M. Greenwald and S. Khanna, "Space-efficient online computation of quantile summaries," in Proc. 2001 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'01), Santa Barbara, CA, May 2001, pp. 58–66.
17. S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan, "Clustering data streams," in Proc. IEEE Symposium on Foundations of Computer Science (FOCS'00), Redondo Beach, CA, 2000, pp. 359–366.
18. J. Han, J. Pei, G. Dong, and K. Wang, "Efficient computation of iceberg cubes with complex measures," in Proc. 2001 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'01), Santa Barbara, CA, May 2001, pp. 1–12.
19. V. Harinarayan, A. Rajaraman, and J.D. Ullman, "Implementing data cubes efficiently," in Proc. 1996 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'96), Montreal, Canada, June 1996, pp. 205–216.
20. G. Hulten, L. Spencer, and P. Domingos, "Mining time-changing data streams," in Proc. 2001 ACM SIGKDD Int. Conf. Knowledge Discovery in Databases (KDD'01), San Fransisco, CA, Aug. 2001.
21. T. Imielinski, L. Khachiyan, and A. Abdulghani, "Cubegrades: Generalizing association rules," Data Mining and Knowledge Discovery, vol. 6, pp. 219–258, 2002.
22. X. Li, J. Han, and H. Gonzalez, "High-dimensional OLAP: A minimal cubing approach," in Proc. 2004 Int. Conf. Very Large Data Bases (VLDB'04), Toronto, Canada, Aug. 2004, pp. 528–539.
23. G. Manku and R. Motwani, "Approximate frequency counts over data streams," in Proc. 2002 Int. Conf. Very Large Data Bases (VLDB'02), Hong Kong, China, Aug. 2002, pp. 346–357.
24. S. Sarawagi, R. Agrawal, and N. Megiddo, "Discovery-driven exploration of OLAP data cubes," in Proc. Int. Conf. of Extending Database Technology (EDBT'98), Valencia, Spain, March 1998, pp. 168–182.
25. G. Sathe and S. Sarawagi, "Intelligent rollups in multidimensional OLAP data," in Proc. 2001 Int. Conf. on Very Large Data Bases (VLDB'01), Rome, Italy, Sept. 2001, pp. 531–540.
26. Z. Shao, J. Han, and D. Xin, "MM-Cubing: Computing iceberg cubes by factorizing the lattice space," in Proc. 2004 Int. Conf. on Scientific and Statistical Database Management (SSDBM'04), Santorini Island, Greece, June 2004, pp. 213–222.

27. H. Wang, W. Fan, P.S. Yu, and J. Han, “Mining concept-drifting data streams using ensemble classifiers,” in Proc. 2003 ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining (KDD’03), Washington, DC, Aug. 2003.
28. D. Xin, J. Han, X. Li, and B.W. Wah, “Star-cubing: Computing iceberg cubes by top-down and bottom-up integration,” in Proc. 2003 Int. Conf. Very Large Data Bases (VLDB’03), Berlin, Germany, Sept. 2003.
29. Y. Zhao, P.M. Deshpande, and J.F. Naughton, “An array-based algorithm for simultaneous multidimensional aggregates,” in Proc. 1997 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD’97), Tucson, Arizona, May 1997, pp. 159–170.