Knowl Inf Syst (2006)
DOI 10.1007/s10115-006-0024-8

Knowledge and Information Systems

REGULAR PAPER

Moonjung Cho · Jian Pei · Ke Wang

Answering ad hoc aggregate queries from data streams using prefix aggregate trees

Received: 8 September 2004 / Revised: 30 May 2005 / Accepted: 12 June 2005

Abstract In some business applications such as trading management in financial institutions, it is required to accurately answer ad hoc aggregate queries over data streams. Materializing and incrementally maintaining a full data cube or even its compression or approximation over a data stream is often computationally prohibitive. On the other hand, although previous studies proposed approximate methods for continuous aggregate queries, they cannot provide accurate answers. In this paper, we develop a novel *prefix aggregate tree* (PAT) structure for online warehousing data streams and answering ad hoc aggregate queries. Often, a data stream can be partitioned into the *historical segment*, which is stored in a traditional data warehouse, and the *transient segment*, which can be stored in a PAT to answer ad hoc aggregate queries. The size of a PAT is linear in the size of the transient segment, and only one scan of the data stream is needed to create and incrementally maintain a PAT. Although the query answering using PAT costs more than the case of a fully materialized data cube, the query answering time is still kept linear in the size of the transient segment. Our extensive experimental results on both synthetic and real data sets illustrate the efficiency and the scalability of our design.

Keywords Data warehousing · Data cube · Data stream · Online analytic processing (OLAP) · Aggregate query

M. Cho
Department of Computer Science and Engineering, State University of New York at Buffalo, Buffalo, NY 14260, USA
E-mail: mcho@cse.buffalo.edu

J. Pei (✉) · K. Wang
School of Computing Science, Simon Fraser University, 8888 University Drive, Burnaby, BC, Canada V5A 1S6
E-mail: {jpei,wangk}@cs.sfu.ca

1 Introduction

Data warehousing and online analytic processing (OLAP) are essential facilities for many data analysis tasks and applications. Given a multidimensional base table, a data warehouse materializes a large set of aggregates from the table. By proper indexes in a data warehouse, various aggregate queries (OLAP queries) can be answered online.

In general, the complete set of aggregate cells on a multidimensional base table can be huge. For example, if a base table has 20 dimensions and the cardinality of each dimension is 10, then the total number of aggregate cells is $11^{20} \approx 6.7 \times 10^{20}$. Even if only on average one out of 10^{10} aggregate cells is non-empty (i.e., covering some tuple(s) in the base table), the total number of non-empty aggregate cells still can be up to 6.7×10^{10}! Thus, computing and/or materializing a complete data cube is often expensive in both time and space, and hard to be online.

Recently, several important applications see the strong demands of online answering *ad hoc aggregate queries* over fast data streams. In this paper, we are particularly interested in the applications where the *accurate instead of approximate answers* to the queries are mandatory. For example, trading in futures market is often a high-risk and high-return business in many financial institutions. Transactional data and market data are collected in a timely fashion. Dealers often raise various ad hoc aggregate queries about the data in recent periods, such as "*list the total transaction amounts and positions in the last* 4 *hours, by financial products, counter parties, time-stamp (rounded to hour), mature date and their combinations*." In those applications, it is required to *maintain the recent data in a sliding window*, and *provide accurate and online answers to ad hoc aggregate queries over the current sliding window*.

Many previous studies proposed *approximate* methods to monitor aggregates over very fast and high cardinality data streams, such as network traffic data streams where the speed of a data stream can be of gigabytes per second and the cardinality of the IP addresses is 2^{32}. In such situations, it is impossible to obtain accurate answers. Approximate answers usually provide sufficiently good insights. However, the target applications investigated in this paper, such as the transactional data streams in business, are substantially different. First, accurate answers are mandatory in many business applications. This is particularly important for some business applications such as those in the financial industry. Second, the data streams studied here often are not extremely fast, and the cardinality of the data is not very huge. Instead, they have a manageable speed. For example, since the modern computers easily have gigabytes of main memory and typically the transactions in those applications will be in the scale of millions per day, it is reasonable to assume that the current sliding window of transactions can be held into main memory. Thus, it is possible to obtain accurate answers to ad hoc aggregate queries, though the task is still challenging.

Can traditional data warehousing techniques meet the requirement? A traditional data warehouse often updates in batch periodically, such as daily maintenance at nights or weekly maintenance during weekends. Such updates are often conducted offline. Online aggregate queries about the most recent data cannot be answered by the traditional data warehouses due to the delay of the incremental updates.

Then, can we maintain a materialized data cube over the sliding window? Unfortunately, the size of a data cube is likely exponential to the dimensionality and much larger than the sliding window. Moreover, the cubing runtime is also exponential to the dimensionality and often requires multiple scans of the tuples in the sliding window or the intermediate results. However, in a typical data stream, each tuple can be seen only once, and the call-back operations can be very expensive. Thus, a data cube resulted from a reasonably large sliding window is usually too large in space and too costly in time to be materialized and incrementally maintained online.

Can we materialize and incrementally maintain only a small subset of aggregates online by scanning the data stream only once, and still retain the high performance of online answering ad hoc aggregate queries? In other words, can we get a good tradeoff between the query answering efficiency and the efficiency of indexing and maintenance (i.e., the size of index and the time of building and maintaining the index)? To answer aggregate queries online, there are three factors needed to be considered, namely the space to store the aggregates, the time to create and maintain the aggregates, and the query answering time. While the existing static data cubing methods focus on reducing the last of them, *the goal of this paper is to trade off the query answering time a little bit against the space and time of incremental maintenance*.

In this paper, we address the following challenges.

Challenge 1 *Can we avoid computing the complete cube but still retain the capability of answering various aggregate queries efficiently?*

Our contribution We propose a solution that the transient segment (i.e., a sliding window) of a data stream is maintained in an online data warehouse, which is enabled by the idea of materializing only the *prefix aggregate cells* and the *infix aggregate cells*. We show that they form just a small subset of the complete data cube, and the total number of prefix aggregate cells is *linear* to the number of tuples in the sliding window and the dimensionality. With such a small subset of aggregates cells, many aggregate queries, including both point queries and range queries, still can be answered efficiently.

Challenge 2 *How can we compute, maintain and index the selected aggregates from a data stream?*

Our contribution We devise a novel data structure, *prefix aggregate tree* (PAT), to store and index the prefix aggregate cells and the infix aggregate cells. The size of PAT is bounded. Algorithms are developed to construct and incrementally maintain PAT. Our experimental results indicate that PAT is efficient and scalable for fast and large data streams.

Challenge 3 *How can we answer various aggregate queries efficiently?*

Our contribution We develop efficient algorithms to answer essential aggregate queries, including point queries and range queries. Infix links and the locality property of side-links of PAT enable various aggregate queries be answered

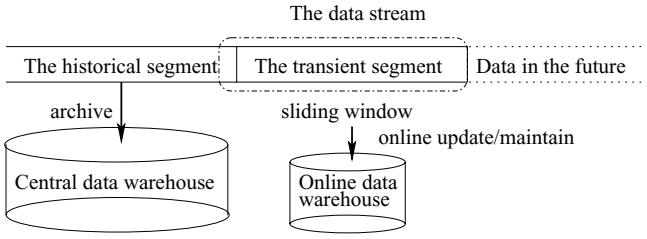

Fig. 1 The framework of warehousing data streams

efficiently. An extensive performance study shows that the query answering is efficacious over large and fast data streams.

The remainder of the paper is organized as follows. In Sect. 2, we describe the framework and review related work. The prefix aggregate tree structure as well as its construction and incremental maintenance are presented in Sect. 3. The query answering algorithms are developed in Sect. 4. The extensive experimental results are reported in Sect. 5. Section 6 concludes the paper.

2 Framework and related work

2.1 The framework

In this study, we model a *data stream* as an (endless) *base table* $S(T, D_1, \ldots, D_n, M)$, where T is an attribute of *time-stamps*, $D_1, \ldots, D_n$ are n *dimensions* in discrete domains, and M is the *measure*. For the sake of simplicity, we use positive integers starting from 1 as time-stamps.

In data stream processing, records are often collected in temporal order. Thus, it is reasonable to assume that the tuples having time-stamp τ arrive before the ones having time-stamp $(\tau + 1)$. Tuples having the same time-stamp may be in arbitrary order.

Traditional data warehouses can answer various aggregate queries efficiently. However, those data warehouses have to be incrementally maintained periodically and the maintenance is often offline. That is, it is difficult to answer aggregate queries on the recent data that has not been loaded into the data warehouse in the last update.

To tackle this problem, it is natural to divide a data stream into two segments: the *historical segment* and the *transient segment*, as illustrated in Fig. 1. Conceptually, the historical segment is the data arrived before the last update of the central data warehouse and thus has been archived. The transient segment, in turn, is the data that has not been archived in the central data warehouse, and should be updated and maintained online in an online data warehouse.

Such a framework of historical and transient segments appears in multiple applications and some prototype implementations of commercial databases. However, to the best of our knowledge, there exists no previous study on how to construct and maintain an online data warehouse for the transient segment.

Technically, should the online data warehouse store only the data in the transient segment? Consider the scenario that the central data warehouse is just

updated. Then, the online data warehouse contains very little data and many aggregate queries about the recent data cannot be answered using the online data warehouse. To avoid this problem, *the online data warehouse should maintain the tuples whose time-stamps are in a sliding window of size ω, where ω is the length of the periodicity that the central data warehouse conducts a regular update*.

In other words, at instant t ($t \geq \omega$), we assume that all the queries in the online data warehouse are about multi-dimensional aggregates of tuples falling in the sliding window of $[t - \omega + 1, t]$.

Aggregate functions can be used in the queries, such as `SUM`, `MIN`, `MAX`, `COUNT` and `AVG` in SQL. We consider the following two kinds of queries in this paper.

Point queries. At instant t, a point query is in the form of a *query cell* $(\tau, d_1, \ldots, d_n)$, where $t-\omega+1 \leq \tau \leq t$ or $\tau =^*$, and $d_i \in D_i \cup \{^*\}$. For example, consider a data stream of transaction records in an endless table *transaction (Time-stamp, Branch-id, Prod-id, Counter-party-id, Amount)* where *Branch-id*, *Prod-id* and *Counter-party-id* are the dimensions, and *Amount* is the measure. Suppose the sliding window is of size 24 hours. A point query may ask for "*the total amount of 'gold' at 10 am*", where the query cell is (*10am, *, gold, **). Here, symbols "*" in dimensions *Branch-id* and *Counter-party-id* mean every transaction in any branch and with any counter-party counts.

Particularly, when $\tau = *$, the aggregate over the whole sliding window is returned. As another example, query cell (**, Paris, *, **) stands for the total trading amount in Paris in the current sliding window, including all products and all customers.

Range queries. A range query specifies ranges instead of a specific value in some dimensions. Thus, a range query may cover multiple query cells. For example, a range query may ask for "*the total amount of 'gold' and 'oil' in Paris and London in the last* 2 *hours*", denoted as $([t-2, t]$, {Paris, London}, {gold, oil}, $*)$.

The answer to a range query is one aggregate over all the tuples falling in the range. For example, the above range query is answered by one total amount that covers all the transactions in the two cities and about the two products, in the last 2 hours.[1]

Here, we assume that in a range query, the time range is in the sliding window. If this is not the case, we can easily divide the range into two sub-ranges, one in the historical data, and the other in the sliding window. The query can be answered accordingly.

The above two kinds of OLAP queries are essential, though more complex queries can be raised. Many complex OLAP queries can be decomposed into a set of queries in the above two categories.

Now, the problem becomes *how to construct and incrementally maintain an online data warehouse and answer ad hoc aggregate queries*.

Problem statement. Given a data stream S and a size of sliding window ω. We want to construct and maintain an online data structure $W(t)$ so that, at any instant

[1] Alternatively, a list of the aggregates of the query cells falling in the range can be returned. For example, the above range query may be answered by a list of 4 aggregates corresponding to the combinations of values in the ranges of dimension *Branch-id* and *Product*. The two forms of answers can be derived by similar techniques.

t, any point queries and range queries about the data in time $[t - \omega + 1, t]$ can be answered precisely and efficiently from $W(t)$.

To be feasible for streaming data processing, $W(t)$ should satisfy the following two conditions.

1. The size of $W(t)$ is linear in the number of tuples in the current sliding window and the dimensionality; and
2. $W(t)$ can be constructed and maintained by scanning the tuples in the stream only once.

W is called an *online data warehouse* of stream S.

2.2 Related work

Chaudhuri and Dayal [8] and Widom [43] present excellent overviews of the major technical progresses and research problems in data warehousing and OLAP. It has been well recognized that OLAP is more efficient if a data warehouse is used.

The data cube operator [17] is one of the most influential operators in OLAP. Many approaches have been proposed to compute data cubes efficiently from scratch [6, 33, 34, 45]. In general, they speed up the cube computation by sharing partitions, sorts, or partial sorts for group-bys with common dimensions.

It is well recognized that the space requirements of data cubes in practice are often huge. Some studies investigate partial materialization of data cubes [6, 20, 22]. Methods to compress data cubes are studied in [25, 26, 37, 38, 42]. Moreover [4, 5, 41] investigate various approximation methods for data cubes.

How to implement and index data cubes efficiently is a critical problem. In [35, 38], Cubetree and Dwarf are proposed by exploring the prefix and suffix sharing among dimension values of aggregate cells. Quotient cube [25] is a non-redundant compression of data cube by exploring the semantics of aggregate cells, and QC-tree [26] is an effective data structure to store and index quotient cube. As compressions of a data cube, they can be used to answer queries directly, and quotient cube can further support some advanced semantic OLAP operations, such as roll-up/drill-down browsing.

Many studies have been conducted on how to answer various queries effectively and efficiently using fully or partially materialized data cubes, such as [10, 23, 27, 29, 39]. To facilitate the query answering, various indexes have been proposed. In [36], Sarawagi provides an excellent survey on related methods. A data warehouse may need to be updated in a timely fashion to reflect the changes of data. Refs. [18, 30–32] study the maintenance of views in data warehouses.

Recently, intensive research efforts have been invested in data stream processing, such as monitoring statistics over streams and query answering (e.g., [3, 13–15]) and multi-dimensional analysis (e.g., [9]). Please see Babcock et al. [2] for a comprehensive overview. While many of them focus on answering *continuous queries*, few of them consider answering *ad hoc queries* by warehousing data streams. Probably, the work most related to this paper is [9], where a linear-regression based approach is proposed to accumulate the multi-dimensional aggregates from a data stream, and a variation of the H-tree structure [20] is used to materialize some selected roll-up/drill-down paths for OLAP. However, their

method assumes that the streaming data can be summarized effectively by linear regression and can only provide approximate answers to (preset) aggregate queries, and no efficient method is presented to answer various ad hoc aggregate queries in general. Moreover, the selected roll-up/drill-down paths are hard to determine. It is unclear how the H-tree structure can be stored and incrementally maintained effectively for data streams.

Particularly, this work is related to the research on mining frequent itemsets from data streams [1, 7, 11, 12, 16, 24, 28, 40, 44]. Basically, for a given stream of transactions, where a transaction is a set of items, the frequent itemset mining problem for data streams is to maintain the set of itemsets that appear at least $\Delta \cdot n$ times in the transactions seen so far, where Δ is a minimum support threshold, and n is the number of transactions seen so far. Some methods put weights to transactions, and the more recent transactions have heavier weights. Frequency can be viewed as a type of aggregates. However, all of the previous methods are approximate approaches. They cannot provide the exact answers, though some methods can provide different types of quality guarantees.

To the best of our knowledge, this is the first study on warehousing and indexing data in a sliding window over data stream and answering ad hoc aggregate queries accurately.

On the other hand, tree and prefix-tree structures have been frequently used in data mining and data warehousing indices, including cube forest [23], FP-tree [21], H-tree [20], Dwarf structure [38] and QC-tree [26]. PAT is also a prefix tree data structure. We will further compare our approaches to those important existing methods in Sect. 3.2, after the major technical ideas of PAT are brought up.

3 Prefix aggregate tree (PAT)

In this section, we devise the prefix aggregate tree (PAT) data structure. We also develop algorithms to construct and incrementally maintain prefix aggregate tree. Hereafter, all aggregate queries are ad hoc ones.

3.1 Data structure

Consider a data stream $S(T, D_1, \ldots, D_n, M)$. Let the size of the sliding window be ω. In order to answer any aggregate query about the data in the sliding window, we have to store the tuples in the sliding window. An intuitive way to store the tuples compactly is to use a prefix tree, as shown in the following example.

Example 3.1 Let the data stream as our running example be $S(T, A, B, C, D, M)$, where T and M are the attributes of time-stamps and the measure, respectively. The tuples at instants 1 and 2 are shown in Table 1. Suppose aggregate function `SUM` is used, and the size of the sliding window $\omega = 2$.

If, we ignore the time-stamps and measures, the tuples in the sliding window can be organized into a prefix tree, as shown in Fig. 2a.

In order to store the information about the time-stamps and measures, we can register the information at the tree nodes, as shown in Fig. 2b. Each tree node has

Table 1 The tuples at instants 1 and 2 in stream $S(T, A, B, C, D, M)$

T	A	B	C	D	M
1	a_1	b_2	c_1	d_1	6
1	a_1	b_1	c_1	d_1	2
2	a_1	b_1	c_2	d_2	3
2	a_2	b_1	c_2	d_1	4

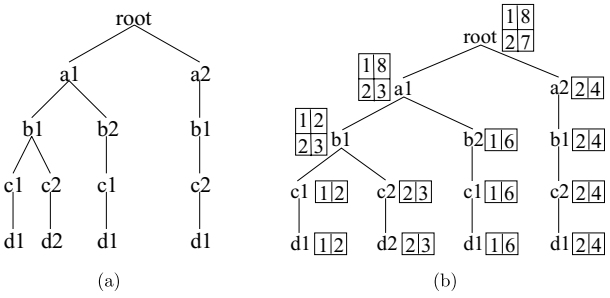

Fig. 2 Archiving a data stream in a prefix tree. **a** Storing tuples into a prefix tree. **b** Prefix tree with aggregate table at each node

an *aggregate table*, such that the time-stamps and the aggregates by instants are registered.

Clearly, the leaf nodes in the prefix tree record the tuples in the sliding window. Each internal node in the tree registers the aggregate of the tuples whose paths go through this node. For example, the node a_1 in the tree registers the aggregates of tuples having a_1 and stores them by instants.

In a prefix tree, one node can be represented by the path from the root of the tree to the node. Hereafter, we will write a node as a string of dimension values, such as a_1, a_1b_1 and $a_1b_1c_2$.

In the prefix tree shown in Fig. 2b, an internal node registers the aggregates of tuples sharing the "prefix" from the root to the node. They are called prefix aggregate cells.

Definition 3.1 (Prefix aggregate cell) Consider a data stream $S(T, D_1, \ldots, D_n, M)$, where T and M are attributes of time-stamps and measure, respectively. For any tuple, we always list the dimension values in the order of $D_1, \ldots, D_n$. Let S_t be the set of tuples in the current sliding window $[(t - \omega + 1), t]$ of S. An aggregate cell $c = (\tau, d_1, \ldots, d_n)$ is a *prefix aggregate cell* if (1) there exists a k such that $1 \leq k \leq n, d_1, \ldots, d_k$ are not $*$ and $d_{k+1}, \ldots, d_n$ are all $*$; and (2) there exists at least one tuple $c' = (d'_1, \ldots, d'_n)$ in S_t such that $d_i = d'_i$ for $(1 \leq i \leq k)$.

Theorem 1 *By storing only the prefix aggregate cells, any ad hoc aggregate queries about the current sliding window can be answered.*

Proof Clearly, the answer to any ad hoc aggregate query about the current sliding window can be derived from the complete set of tuples in the window. Let us consider tuples in the current window. If a tuple t is unique in the current sliding

window, t is (trivially) a prefix aggregate cell. If t is not unique, then there exists a prefix aggregate cell which has the same value as t on every dimension. In other words, the set of prefix aggregate cells covers all tuples in the current sliding window. □

Theorem 2 (Numbers of aggregate cells/prefix aggregate cells) *Given a base table of n dimensions and k tuples, let* n_{aggr} *and* n_p *be the number of aggregate cells and that of prefix aggregate cells, respectively. Then,* $2^n \leq n_{aggr} \leq (k \cdot (2^n - 1) + 1)$ *and* $(n + 1) \leq n_p \leq (k \cdot n + 1)$.

Proof When tuples share some values in some dimensions, they share the corresponding aggregate cells. When all tuples in the base table have the same value on every dimension, n_{aggr} is minimized. When the k tuples do not share any common value in any dimension, each tuple leads to $2^n - 1$ unique aggregate cells, and all tuples share aggregate cell $(*, \ldots, *)$. Thus, n_{aggr} is maximized to $(k \cdot (2^n - 1) + 1)$.

When tuples share some prefixes, they share the corresponding prefix aggregate cells. When all tuples in the base table have the same value on every dimension, n_p is minimized. When the k tuples do not share any prefix, each tuple leads to n prefix aggregate cells, and all tuples share aggregate cell $(*, \ldots, *)$. Thus, n_{aggr} is maximized to $(k \cdot n + 1)$. □

Theorem 2 indicates that the number of prefix aggregate cells is, in the worst case, linear in the number of tuples in the sliding window and the dimensionality, and thus is substantially smaller than that of all aggregate cells. It suggests that the set of prefix aggregate cells is a promising candidate of an online data warehouse for a data stream.

Given a prefix tree of the prefix aggregate cells, aggregate queries can be answered by browsing the tree and extracting the related tuples in the current sliding window. However, if the current sliding window is large and thus the prefix tree is also large, browsing a large tree may not be efficient. We should build some light-weight index in the tree to facilitate the search.

Let us consider how to derive the aggregate for $(*, b_1, *, *)$ from the prefix tree in Fig. 2b. To answer this query, we need to access all the tuples having value b_1 on dimension B. To facilitate the search, it is natural to introduce the *side links* that link all nodes carrying the same label together.

Can we add side links arbitrarily? Let us consider how to compute aggregate $(a_1, *, c_1, *)$ from the tree in Fig. 2b. To answer this query, we want to access the nodes carrying label c_1 in the subtree rooted at node a_1. That is, we want a *local* linked list of nodes having label c_1 in the subtree rooted at node a_1.

Clearly, maintaining multiple linked lists is not a good idea. Instead, we should construct a linked list that has the *locality property: in any subtree, the nodes carrying the same label should be linked consecutively.*

Moreover, if we can register the aggregate of all tuples having c_1 in the a_1-subtree, the query can be answered even faster. This information can be registered as the head of the sub-linked list of c_1 in the a_1-subtree.

To accommodate the above ideas, we can construct a linked list of all nodes having c_1 in the a_1-subtree, and set up a pointer to the head of the sub-linked list at node a_1. The corresponding aggregate, $(a_1, *, c_1, *)$ should also be stored and associated with the head of the linked list.

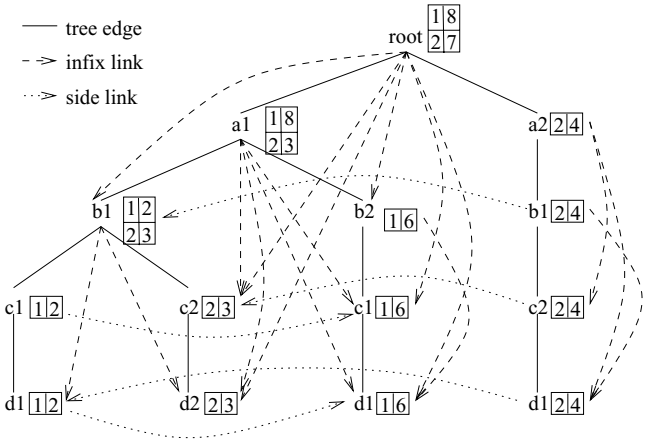

Fig. 3 Prefix aggregate tree (the aggregate tables for infix links are omitted to make the graph easy to read)

The above ideas can be generalized. For example, side links can be built in the prefix tree in Fig. 2b, resulting in a *prefix aggregate tree* structure, as shown in Fig. 3.

Definition 3.2 (Prefix aggregate tree) In general, given the current sliding window of a data stream, a *prefix aggregate tree* (PAT for short) is a prefix tree of the prefix aggregate cells in the window with the following two kinds of links.

- *Side links* All nodes having the same dimension value as the label are linked together such that the locality property is hold: For any subtree, all the nodes carrying the same label in the subtree are linked consecutively.
- *Infix links* At node $v = d_1 \cdots d_k$ $(1 \leq k \leq (n-2))$, for every dimension value $d_{i,j}$ on dimension D_i $((k+2) \leq i \leq n)$ that appears in the subtree rooted at v, all the nodes carrying label $d_{i,j}$ in the subtree rooted at v are linked consecutively by side-links. An *infix link* is built from v to the head of the sublist, and an aggregate table is stored and associated with the infix link recording the aggregates of $c = (d_1, \ldots, d_k, *, \ldots, *, d_{i,j}, *, \ldots, *)$. c is called an *infix aggregate cell*.

Theorem 3 *Consider a base table of n dimensions and the cardinality of each dimension is l. In the worst case, the number of infix aggregate cells (and thus the infix links in a PAT) is*

$$\sum_{i=1}^{n-2} l^i (n-i-1).$$

Proof From the definition of PAT, the number of infix aggregate cells and that of infix links are identical. The worst case happens when every possible combination of dimension values appears in the base table, where the base table has l^n unique tuples. Each internal node in the PAT has l children. Thus we have the upper bound. In such a situation, there are $(l+1)^n$ aggregate cells. □

Theorems 2 and 3 show that, in the worst case, the set of prefix aggregate cells and infix aggregate cells is still a small subset of all the aggregate cells. Practical data is usually skewed. As verified by our experimental results, the PAT is much smaller than the size of all aggregate cells.

The size of a node in a PAT is regular. For a prefix aggregate cell $(d_1, \ldots, d_k, *, \ldots, *)$, the corresponding node in the tree is at the k-th level (the root node is at level 0), and stores the following information: (1) The aggregate table, which has 2 columns, the time-stamp and the aggregate, and at most ω records; (2) Pointers to up to l_{k+1} children, where l_{k+1} is the cardinality of dimension D_{k+1}; (3) At most $\sum_{i=k+2}^{n} l_i$ infix aggregate links and the corresponding aggregate tables, where l_i is the cardinality of dimension D_i; and (4) A side link to the next node at the same level carrying the same label.

The total size of such a tree node is $O(\omega + l_{k+1} + \omega \cdot \sum_{i=k+2}^{n} l_i + 1) = O(\omega \cdot \sum_{i=k+1}^{n} l_i)$. Comparing to the number of tuples in a base table, which can be easily in millions, the number of dimensions and the cardinality in each dimension are often pretty small. All nodes at the same level of the tree have the same size. A PAT can be easily stored and managed in main memory or on disk.

We assume that an order of dimensions is used to construct a PAT. In fact, the order of dimensions affects the size of the resulting PAT. Heuristically, if we order the dimensions in cardinality ascending order, then the tuples may have good chances to share prefixes and thus the resulting PAT may have a small number of nodes. The tradeoff is that the tree nodes may be large due to the large number of infix links. On the other hand, if we sort dimensions in the cardinality descending order, then the number of nodes may be large but the nodes themselves may be small. Theoretically, finding the best order to achieve a minimal PAT is an NP-hard problem. This problem is similar to the problem of computing a minimal FP-tree by ordering the frequent items properly. In Sect. 5, we will study the effect of ordering on the size of PAT by experiments.

3.2 Comparison: PAT versus previous methods

Prefix tree (trie) structures have been extensively used in the previous studies on data mining and data warehousing. PAT is another prefix tree structure. At the first glance, PAT may look similar to some of the previous structures, including Cube forest [23], FP-tree [21], H-tree [20], Dwarf structure [38] and QC-tree [26]. However, there are some essential differences.

An FP-tree [21], a data structure for frequent itemset mining, records transactions in a prefix tree structure such that transactions sharing common prefixes also collapse to the same prefix in the tree. There are three critical differences between an FP-tree and a PAT. First, FP-tree is for transaction data and PAT is for relation data. While transactions may have different lengths, all tuples stored in a PAT have the same length. Infix links do not appear in an FP-tree. Second, FP-tree does not bear the locality property. Instead, the side links in an FP-tree are built as transactions arrive. As shown later, the locality property in a PAT facilitates the aggregate query answering substantially. Last, an FP-tree is for frequent itemset mining. During the mining, an FP-tree is scanned multiple times, and the mining results are output. A PAT is for aggregate query answering. It is built and

maintained by one scan of the data stream. A query answering algorithm searches the PAT to answer aggregate queries.

Cube forest [23], H-tree [20], Dwarf structure [38] and QC-tree [26] are for data warehousing. PAT distinguishes itself from those designs in the following two aspects.

First, *PAT stores only prefix and infix aggregates, while most of the previous structures, except for* H-tree*, potentially store all aggregates, though compression is explored*. As a result, the number of nodes in PAT is linear in the number of tuples in the sliding window and the dimensionality, while those structures are exponential to the dimensionality. Moreover, the size of tree nodes in PAT is regular, as analyzed before. The advantages on size and regularity of tree nodes make PAT feasible for streaming data.

An *H-tree* is a prefix tree of the tuples in base table. Thus, it is also linear in the dimensionality. A PAT can be regarded as an extension of an *H-tree* to support aggregate queries for data streams. The critical difference is that an *H-tree* does not have infix links and thus the query answering on an *H-tree* directly can be very costly.

Second, *PAT is indexed by infix links and side-links, and the side-links have the locality property.* As will be shown soon, the locality property and the infix links facilitate query answering substantially. In most of the previous structures, the search is based on values and is conducted dimension by dimension.

In terms of size, a PAT is larger than an *H-tree*: the difference is infix aggregates and infix links. The size of the infix aggregates and the infix links is quantified in Theorem 3. The infix links have to meet the locality requirement. Theorem 4 will discuss the procedure. As will be shown, it takes the extra cost in time linear in the dimensionality to maintain the locality.

3.3 PAT construction

We consider constructing a PAT by reading tuples into main memory one by one (or batch by batch), and each tuple can be read into main memory only once the algorithm is presented in Fig. 4 and elaborated in this subsection.

Example 3.2 Let us construct a PAT by reading the tuples in Table 1 one by one.

A PAT is initialized as a tree with only one node, the root. Then, The first tuple, $(1, a_1, b_2, c_1, d_1, 6)$, is read and inserted into the tree. For each node in the path, a row $(1, 6)$ is registered in the aggregate table. The infix links from *root*

Algorithm 1 Tree construction
Input: the current sliding window B
Output: a PAT
Method:
initialize a tree with only the root node;
read the tuples into main memory one by one, one at each time;
FOR each tuple DO
  insert the tuple into the tree;
  FOR each node on the path of the inserted tuple DO
    update the aggregate at the node;
    if it is a new node, adjust infix links and side-links according to Theorem 4

Fig. 4 The PAT construction algorithm by scanning tuples one by one

to a_1b_2, $a_1b_2c_1$ and $a_1b_2c_1d_1$, infix links from a_1 to $a_1b_2c_1$ and $a_1b_2c_1d_1$, and infix links from a_1b_2 to $a_1b_2c_1d_1$ are created. The corresponding infix aggregate cells are $(*, b_2, *, *)$, $(*, *, c_1, *)$, $(*, *, *, d_1)$, $(a_1, *, c_1, *)$, $(a_1, *, *, d_1)$ and $(a_1, b_2, *, d_1)$, respectively. Once the information is recorded in the tree, we do not need the tuple any more.

Then, we read the second tuple $(1, a_1, b_1, c_1, d_1, 2)$ and insert it into the tree. The aggregate values at nodes *root* and a_1 should be both updated to $(1, 8)$, since they are on the path of the inserted tuple. The infix links from *root* to a_1b_1 and from a_1b_1 to $a_1b_1c_1d_1$ are created and associated with the infix aggregate cells $(*, b_1, *, *)$ and $(a_1, b_1, *, d_1)$, respectively.

The remaining tuples can be inserted similarly. It can be verified that the resulting PAT is exactly the one shown in Fig. 3.

The construction of a PAT by scanning tuples one by one has two major components: building the prefix tree, which is straightforward, and creating/maintaining the correct infix links and side-links, which should follow the procedure justified in the following theorem so that the locality property is respected.

Theorem 4 *Let T be a PAT that satisfies the locality property. When a new node v of label d_i is created, the following procedure adjusts the side-links and infix-links so that the resulting PAT preserves the locality property: If v is a child of the root node, no infix link and side-link are needed; else, the side-links and infix links with respect to v should be adjusted as follows:*

1. *The closest ancestor v' of v should be allocated such that v' has an infix link of d_i;*
2. *If v' does not exist, then an infix link should be built from every ancestor of v in the path to v, except for the parent node of v.*
3. *Otherwise, let u be the node pointed by the d_i-infix link of v', and V be the set of ancestor nodes of v' whose d_i-infix links also point to u. v should be inserted into the front of the sublist pointed by the d_i-infix link of v', and the d_i-infix links of v' and nodes in V should point to v.*

Proof The correctness of Step 2 is clear since in such a case, v is the first node carrying label d_i. The corresponding infix links should be created.

Suppose PAT T already has some nodes carrying label d_i. To preserves the locality property, v should be inserted to the head of the non-empty consecutive sublist of its closest ancestor. Once the locality property holds for the smallest subtree containing the new node, the locality property holds for any larger subtrees containing the smallest subtree, since the PAT before the insertion has the locality property. □

The complexity of the procedure described in Theorem 4 is linear in the dimensionality. At each node, a table of aggregates is maintained. Each table has two columns: time-stamp and aggregate, and at most ω rows. The aggregate at instant t should be stored at the (t mod ω)-th row. Therefore, the cost of maintaining and searching the table is constant.

Algorithm 3 Incremental maintenance
Input: the PAT T at instant t and the tuples at instant $(t+1)$
Output: the PAT at instant $(t+1)$
Method:
insert tuples at instant $(t+1)$ into T, FOR each node v on the path DO
IF v's aggregate table contains a row of instant $(t-\omega+1)$, THEN remove the row from the aggregate table;
IF v is a leaf node THEN put v into the LUT list of $LUT = t+1$;
FOR each node v in the LUT list of $LUT = t-\omega+1$ DO
search v's ancestors upward until a node having aggregates after instant $t-\omega+1$ is encountered;
remove v and those ancestors of v and the infix links, adjust the related side-links, too

Fig. 5 The PAT incremental maintenance algorithm

3.4 Incremental maintenance

Suppose we have a PAT at instant t. At instant $(t+1)$, the new data tuples should be read in and inserted into the tree, and the data at instant $(t-\omega+1)$ should be removed, so that the sliding window is moved forward to $[t-\omega+2, t+1]$. The algorithm is shown in Fig. 5. We explain the critical details as follows.

To be efficient, should insertions go first or deletions first? Consider a tree node v whose aggregate table contains only an aggregate at instant $(t-\omega+1)$. Suppose that some tuples from the stream at instant $(t+1)$ will contribute a new row in v's aggregate table. If deletions go first, the node would be removed since its aggregate table is empty after the deletion. Then, the insertion of the tuples at instant $(t+1)$ will have to recreate the node. To avoid such an unnecessary deletion-and-re-creation, we should let the insertions of tuples at instant $(t+1)$ go first before the deletions of tuples at instant $(t-\omega+1)$.

Insertion of tuples at instant $(t+1)$ can be done in the way similar to Algorithm 1 (Fig. 4), the tree construction by scanning the tuples one by one. That is, we take the existing PAT at instant t, and insert the tuples at instant $(t+1)$ into the tree.

Please note that, during the insertion, we do not need to scan any tuples in the previous instants. The only tuples scanned are those at instant $(t+1)$.

Now, let us consider *how to delete the tuples whose time-stamps are* $(t-\omega+1)$. A naïve method is as follows. We search the PAT. For each node that contains an aggregate at instant $(t-\omega+1)$ in its aggregate table, the corresponding row in the aggregate table should be removed. If the aggregate table becomes empty, then the node should be deleted.

The above naïve method is costly. There can be many nodes in the tree containing aggregates at instant $(t-\omega+1)$. Aggressively updating a large number of nodes may degrade the online performance. Moreover, how to locate the nodes containing aggregates at instant $(t-\omega+1)$ is another problem. Browsing the whole tree can be very expensive.

Here, we propose a *lazy approach*: the nodes whose aggregate tables have only rows of instants $(t-\omega+1)$ or earlier have to be removed at instant $(t+1)$, in order to release the space. Other than that, the deletions of the old aggregates of instant $(t-\omega+1)$ from the nodes are deferred and conducted as a byproduct of future insertions. The idea is elaborated in the following example.

Table 2 The tuples at instant 3

T	A	B	C	D	M
3	a_1	b_2	c_2	d_2	5
3	a_2	b_2	c_1	d_2	1

Example 3.3 Suppose the tuples at instant 3 are as shown in Table 2. Since the size of the sliding window $\omega = 2$, the tuples at instant 3 should be inserted and the tuples of instant 1 should be removed. Let us consider how the PAT in Fig. 3 should be incrementally maintained.

We first insert the tuples at instant 3 into the PAT. Tuple $(3, a_1, b_2, c_2, d_2, 5)$ is inserted from the root node as a path "a_1-b_2-c_2-d_2". A record $(3, 5)$ will be stored at the first row of the aggregate table, since $3 \bmod 2 = 1$. It overwrites record $(1, 8)$ automatically. Similarly, we update the aggregate tables at nodes a_1 and a_1b_2, and the aggregate tables for the related infix links, respectively. Please note that the removal of data at instant 1 from these nodes are conducted as a *byproduct* of the insertions, i.e., we do not actively search for the nodes whose aggregate tables having rows of instant 1. To complete the insertion, two new nodes, $a_1b_2c_2$ and $a_1b_2c_2d_2$, are created. Following Theorem 4, the aggregate tables at these nodes as well as the appropriate side-links and infix links are adjusted. The second tuple, $(3, a_2, b_2, c_1, d_2, 1)$ can be inserted similarly.

Then, we should remove all those nodes whose aggregate tables contain only rows of instant 1. To find such nodes, we maintain an integer for each leaf node, called the *last update time-stamp* (LUT), which is the latest time-stamp that the node is updated. All leaf nodes having the same LUT are linked together as a linked list. By browsing the linked list for LUT$=1$, we remove the leaf nodes and their ancestors that have only aggregates at instant 1. The upward search stops when the first ancestor having aggregates at instant other than 1 is encountered. In this example, nodes $a_1b_1c_1$, $a_1b_1c_1d_1$, $a_1b_2c_1$, and $a_1b_2c_1d_1$ are removed. The resulting tree is shown in Fig. 6.

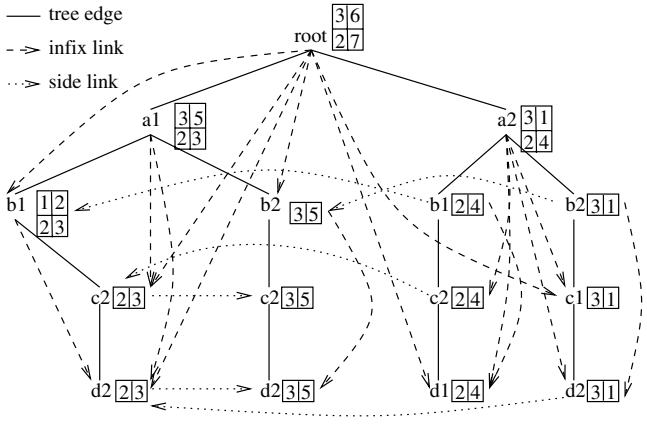

Fig. 6 Prefix aggregate tree at instant 3

Please note that the aggregate table at node a_1b_1 still contains the aggregate at instant 1. However, this information does not affect our query answering. This row will be removed in the future. For example, at instant 4, if there is a new tuple having a_1b_1 as a prefix, the row will be overwritten and the node will be updated. Otherwise, the node will be removed when we clean up nodes containing only aggregates at instant 2 or earlier.

In summary, at instant $(t + 1)$, the incremental maintenance algorithm only scans the tuples having time-stamp $(t + 1)$ once, and inserts them into the existing PAT. By following the LUT list of $(t-\omega+1)$, the maintenance algorithm removes those tree nodes and the corresponding infix links whose aggregate tables have only rows of instant $(t - \omega + 1)$ or earlier. It never browses the complete PAT during the incremental maintenance.

Theorem 5 *The time complexity of constructing and incrementally maintaining a PAT is $O(nl)$, where n is the number of tuples needed to inserted into the PAT, and l is the cardinality.*

Proof The complexity follows the algorithms in Figs. 4 and 5. For each tuple, the insertion time and the time to maintain the locality of the infix links are linear in the dimensionality l. □

4 Aggregate query answering

We consider how to answer aggregate queries of two categories: point queries and range queries.

4.1 Answering point queries

Point queries can be answered efficiently using a PAT. The algorithm is shown in Fig. 7. We illustrate the major idea in the following example.

Example 4.1 (Point query answering) Let us use the PAT shown in Fig. 3 to answer some illustrative point queries about the sliding window [1, 2] in the data stream of our running example.

First, consider query cell $q_1 = (1, a_1, b_1, *, *)$, which itself is a prefix aggregate cell. Its aggregate, 2, is registered in the aggregate table of node a_1b_1. Following the path from the root to the node, we retrieve the answer immediately.

Let us consider query cell $q_2 = (*, *, b_1, *, *)$. It is not a prefix aggregate cell. Instead, it is an infix aggregate cell. Thus, by the aggregate table associated with the infix link of label b_1 at node *root*, we can retrieve the aggregate. Please note that the aggregates are stored by instants in the aggregate table, i.e., two rows (1, 2) and (2, 7) are in the aggregate table of the infix link. We need to get the sum of them since $\tau = *$ in this query cell.

As the third example, let $q_3 = (*, *, b_1, *, d_1)$. This query cannot be answered by one node in the tree. Instead, following the infix link of label b_1 at node *root*, we can reach the linked list of all nodes having b_1. Following the side-links, we can access all the nodes of b_1 in the tree.

Algorithm 4 Answering point queries
Input: the PAT T at instant t and a query cell $q(\tau, d_1, \ldots, d_n)$
Output: the aggregate of the query cell
Method:
IF $\tau \neq *$ and the root's aggregate table does not have a row about τ THEN RETURN($null$);
let *current-node*=*root*;
$m = search(current\text{-}node,\ q)$;
RETURN(m);
Function $search(current\text{-}node,\ q\text{-}cell)$
suppose *q-cell* $= (\tau, d_1, \ldots, d_l)$;
IF $d_1 \neq *$ THEN
IF *current-node* has a child v of label d_1 THEN
IF $\tau \neq *$ and there is no row of instant τ in the aggregate table of v THEN RETURN($null$);
ELSE *current-node*$= v$; $q' = (\tau, d_2, \ldots, d_l)$; $m = search(current\text{-}node, q')$; RETURN($m$);
ELSE RETURN(m);
ELSE //case $d_1 = *$
IF $d_1 = \cdots = d_l = *$ THEN RETURN the aggregate at *current-node*;
let d_i be the first dimension non-$*$ value in q;
IF *current-node* has no infix link of label d_i THEN RETURN(m);
IF $\tau \neq *$ and the aggregate table of the infix link of d_i has no row of τ THEN RETURN(m);
$m = 0$;
follow the side links, visit every node carrying label d_i in the subtree of v, FOR each node v' in the linked list DO
let *current-node*$= v'$; $q' = (d_{i+1}, \ldots, d_l)$; $m = aggr(m, search(current\text{-}node,\ q'))$
RETURN(m)

Fig. 7 The algorithm answering point queries

For each node in the linked list, we recursively retrieve the aggregates from the infix link of label d_1 at the node. For example, at node a_1b_1, by the infix link of d_1, we immediately know the aggregate of $(*, a_1, b_1, *, d_1)$ is 1 even without visiting any node of d_1. Similarly, at node a_2b_1, we can retrieve the aggregate of $(*, a_2, b_1, *, d_1)$, 4, from the infix link. The sum of the aggregates from the infix aggregate cells, 6, answers the query.

As another example, let us consider query cell $q_4 = (*, a_2, *, c_2, d_2)$. Following the tree edge *root*-a_2 and infix link of c_2 at node a_2, we reach the local linked list of c_2 in the subtree of a_2. The locality property of the side-links and the infix link avoids the search of the complete linked list of c_2. Since $a_2b_1c_2$ has only one child, which is $a_2b_1c_2d_1$, the query returns $null$. Here, $null$ means there is not any tuple matching the aggregate cell.

The aggregate tables can also be used to prune the search. For example, consider query cell $q_5 = (2, a_1, b_2, *, d_1)$. It follows the path a_1b_2 in the PAT. However, the aggregate table at node a_1b_2 does not contain any row of instant 2. Thus, we can return $null$ immediately without searching the subtree any more.

As can be seen, in answering point queries, infix links and side-links are used to search only the related tuples. Moreover, the locality property of side-links guarantees that we do not need to search any extra nodes or branches.

4.2 Answering range queries

In principle, a range query can be rewritten as a set of point queries. Then, Algorithm 4 can be called repeatedly to answer the queries. However, calling Algorithm 4 and thus searching the PAT repeatedly may not be efficient.

Here, we propose a *progressive pruning* approach, as exemplified in the following example.

Example 4.2 (Range query answering) Let us consider how to answer range query $(2, *, \{b_0, b_1\}, *, \{d_1, d_2\})$. The query cell can be rewritten as a set of 4 queries cells: $(2, *, b_0, *, d_1)$, $(2, *, b_0, *, d_2)$, $(2, *, b_1, *, d_1)$ and $(2, *, b_1, *, d_2)$. Algorithm 4 can be called four times to answer the point queries, respectively, and the sum, 7, should be returned.

Instead of calling Algorithm 4 four times, we can search the PAT as follows. We start from the root node, since the first non-$*$ dimension value in the query cell should be either b_0 or b_1 on dimension B, we search the infix links of b_0 and b_1 from the root node. Since there exists no infix link of b_0, we prune the range query cell to $(2, *, b_1, *, \{d_1, d_2\})$. At the same time, we only need to search subtrees rooted at the nodes having label b_1, which are linked by the side-links. That is, a part of the search space is also pruned.

There are two nodes carrying label b_1 in the PAT, a_1b_1 and a_2b_1. We search them one by one. For node a_1b_1, since the next non-$*$ dimension value in the query cell should be either d_1 or d_2, and the time-stamp is 2, only the infix links of d_1 and d_2 are searched, and only the infix link of d_2 has a row of time-stamp 2. Thus, the aggregate 3 is extracted. Similarly, aggregate value 4 is extracted from the subtree of a_2b_1. Thus, the sum 7 is returned.

As shown in Example 4.2, the major idea of progressive pruning method for answering range queries is that, instead of searching a PAT many times, we conduct the search using the range query from the root of a PAT. At each node under the search, the query range is narrowed using the information of the available infix links and the corresponding aggregate tables, and the unnecessary nodes are pruned from the search space using the range specification in the query. By progressive pruning, we search the PAT only once for any range query.

5 Experimental results

In this section, we report the experimental results from a systematic performance study. All the algorithms are implemented in C++ on a laptop PC with a 2.8 GHz Pentium 4 processor, a 60 GB hard drive, and 512 MB main memory, running Microsoft Windows XP operating system. In all of our experiments, the PATs reside in main memory.

We generated the synthetic data sets following the Zipf distribution. To generate the data sets, our data generator takes the parameters of the Zipf factor, the dimensionality, the number of tuples, and the cardinality in each dimension. To generate a tuple, we generate the data for each dimension independently following the Zipf distribution. The dimensions in the synthetic data sets are independent, and there is no correlation among any dimensions.

Such a data generation method is popularly used in many previous studies on data cube and data warehouse computation, including [6, 33, 38, 42]. To some extent, it is a benchmark approach to generating synthetic data sets for data cube computation.

In our test, we also used the weather data set [19] which is a real data set. The weather data set is well accepted as a benchmark data set for data cube computation [6, 33, 38, 42].

We tested three methods: the PAT method developed in this paper, the BUC method as described in [6] and a baseline method. The baseline method just simply stores and sorts all tuples in the current sliding window. As expected, the baseline method uses the least main memory to store the data and costs the least runtime to maintain the current sliding window. The tradeoff is the slowest query answering performance. To answer any query, the baseline method has to scan the tuples in the current sliding window. A binary search can be used to locate the tuples matching the time interval of the query. On the other hand, the BUC method computes the whole data cube. It costs the most in computing the whole cube and storing the aggregates. We measure both the main memory cost of BUC and the size of the data cube computed by BUC, which is stored on disk. To answer a query, BUC only needs to conduct a binary search to allocate the corresponding aggregate tuple. Thus, the query answering time is fast. PAT is in between: to compute and store the aggregates for the current sliding window, it costs more space and runtime than the baseline method but less than the BUC method; on the other hand, in query answering, it searches less than the baseline method accordingly.

5.1 Query answering

We first report the performance of answering point queries. The results on range queries are similar. In each test, we randomly select 1000 aggregate cells from the data cube and use them as the point queries. In other words, all queries are on non-empty aggregates.

Beyer and Ramakrishnan [6] does not give a query answering algorithm for BUC. In this performance study, we store the whole data cube *in main memory* as a table, and all tuples are sorted according to the dictionary order. Therefore, answering any point query using the whole cube can be done by a binary search. This is probably the best case of query answering using a data cube. In real applications, usually a whole data cube cannot be held into main memory. To this extent, our experiments favor the query answering using the whole cube. In Fig. 8, we measure the query answering time taken by 1000 queries. All curves are plotted in logarithmic scale.

From the figures, we can clearly see that both PAT and BUC have a much better query answering performance than the baseline method. The baseline method is two orders of magnitudes slower. That simply indicates that, in order to answer aggregate queries online, materializing some aggregate cells is very effective.

PAT and BUC have comparable performance in terms of query answering. However, as discussed before and will be shown later in this section, PAT computes and materializes much fewer aggregate cells than BUC. Thus, the space overhead for storing a PAT is much smaller than the space for all the aggregate cells computed by BUC. Moreover, BUC needs to scan the base table multiple times to compute a complete data cube. Therefore, PAT is a nice tradeoff between space and query answering time.

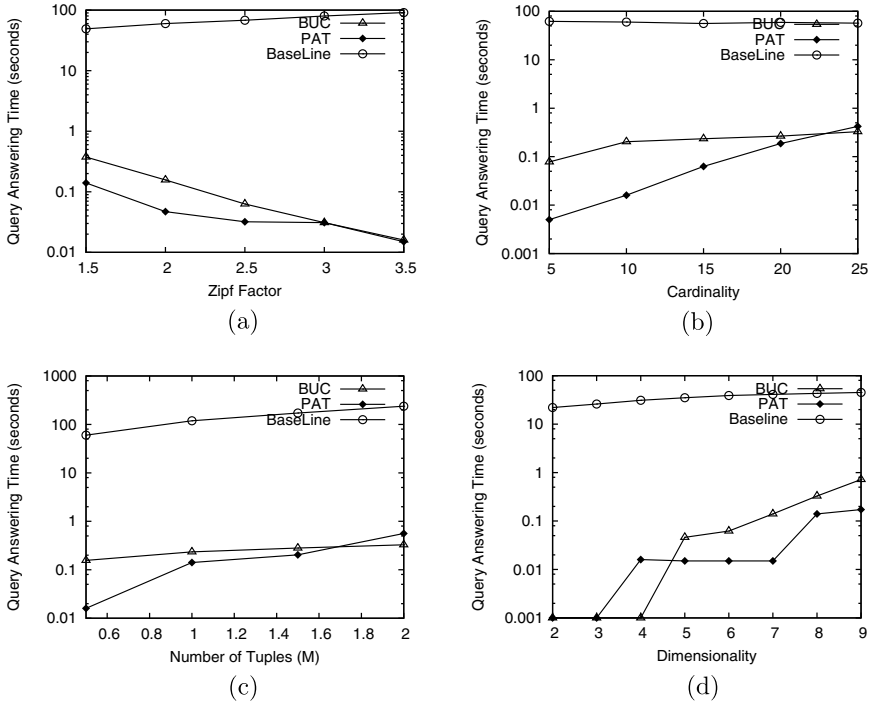

Fig. 8 Results on query answering (all curves are plotted in logarithmic scale). **a** Query answering time vs. Zipf factor (dim = 10, cardinality = 10, no. of tuples = 500 K, no. of queries = 1 K). **b** Query answering time vs. cardinality (Zipf = 2, dim = 10, no. of tuples = 500 K, no. of queries = 1 K). **c** Query answering time vs. no. of tuples (Zipf = 2, dim = 10, cardinality = 10, no. of queries = 1 K). **d** Query answering performance on the weather data set

5.2 Building prefix aggregate trees

We tested the size of the data cube computed by BUC, the size of PAT, the number of aggregates computed, the memory usage (the highest watermark of memory usage during the running of the programs) and scalability (runtime) of the three methods. Four factors are considered: the Zipf factor, the dimensionality, the cardinality of the dimensions and the number of tuples in the current sliding window. The results are consistent. Some results are shown in Fig. 9.

As shown Fig. 9a, the baseline method is not sensitive to the Zipf factor at all, since it simply maintains the tuples in the current sliding window and does not precompute any aggregate. Both BUC and PAT are sensitive the the Zipf factor: the smaller the Zipf factor, the more distinct aggregates exist in the data set. However, PAT runs faster than BUC since it computes much less aggregate cells than BUC.

Figure 9b measures the size of the data cube of BUC, the size of the PAT and the size of the current sliding window in Baseline. The size of PAT counts both the prefix aggregates, infix aggregates and links. The number of tuples varies from 500 thousand to 2.5 million so that the scalability of the methods are tested. All three methods are roughly linear on the number of tuples, but PAT and the baseline method generate much smaller results than BUC. In other words, Baseline does

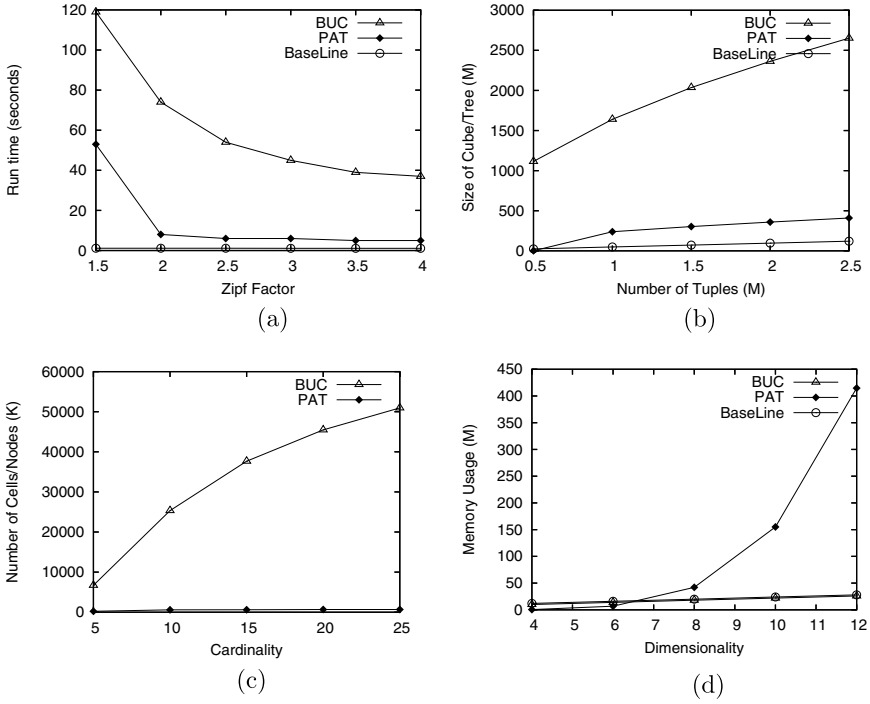

Fig. 9 Results on constructing PAT. **a** Runtime vs. Zipf factor (dim = 10, cardinality = 10, no. of tuples = 500 K). **b** Size of cube/tree vs. no. of tuples (Zipf = 2, dim = 10, cardinality = 10). **c** no. of aggregates vs. cardinality (Zipf = 2, dim = 10, no. of tuples = 500 K). **d** Memory usage vs. dimensionality (Zipf = 2, cardinality = 10, no. of tuples = 500 K)

not generate any aggregates but only the base tuples are maintained. PAT generates the prefix aggregates and the infix aggregates, which form a substantially small subset of all the aggregates generated by BUC. Even when the whole data cube is over 2.5 GB, the PAT including the links occupies less than 500 MB, which is less than three times of the size of all base tuples and can be easily accommodated in main memory.

Figure 9c shows that the number of aggregate cells (including prefix aggregates and infix aggregates) in PAT is linear in the cardinality of the dimensions. The trend is mild. When the cardinality increases, the data set becomes sparse and thus the total number of aggregates also increases. BUC computes all aggregates. As shown in the same figure, the increase of all aggregates is sub-linear in our experiments, but the number of all aggregates is much larger than the number of prefix aggregates and infix aggregates computed by PAT.

Figure 9d shows the memory usage of the three methods. Please note that BUC stores the aggregates on disk. It only maintains the base table in main memory. Thus, it uses the same amount of main memory as the baseline method. Since the PAT resides in main memory, the memory usage of constructing a PAT increases as the dimensionality increases. When there are many dimensions, PAT has many levels.

In summary, a PAT is usually much smaller than the size of a data cube. Constructing a PAT is also much faster than computing a data cube using BUC.

5.3 Incremental maintenance

When testing the performance of incremental maintenance, we always set the number of tuples at each instant to a constant. A sliding window of 10 instants was used. We set the number of tuples in the original PAT to 500K. We only compare the PAT method and the baseline method. For BUC, there is not incremental maintenance algorithm. Thus, to incrementally maintain all the aggregates, we have to compute the data cube on the new data and merge the new aggregates with the existing ones. It has a similar performance as shown in Sect. 5.2. Some results are shown in Fig. 10.

Figure 10a shows the maintenance runtime versus the Zipf factor. It is consistent with Fig. 9a. With a lower Zipf factor, the data set is sparser and thus PAT computes more prefix aggregates and infix aggregates. The baseline method is constant since it does not compute any aggregates. However, by comparing Figs. 9a and 10a, we can see that the incremental maintenance time is much shorter, since many paths in the existing PAT can be reused in the incremental maintenance.

As shown in Fig. 10b, the incremental maintenance time of PAT increases as the dimensionality increases, since the tree becomes larger and taller on high dimensional data sets. The maintenance time of Baseline also increases, but it is linear.

In Fig. 10c, we tested the scalability of the incremental maintenance of PAT and Baseline with respect to the number of new tuples at each instant. The result shows that both PAT and Baseline have an approximately linear scalability. This is consistent with the analysis of the PAT incremental maintenance algorithm.

Figure 10d examines the size of the PAT with respect to the number of new tuples at each instant. Interestingly, we observed that, under a given data distribution, the size of the PAT is stable and insensitive to the number of tuples in the incremental part. In other words, the size of PAT mainly depends on the number of tuples in the sliding window, and is stable during the incremental maintenance. This is a nice property for data stream processing: no matter how large the data stream is, we will have an index structure of a stable size.

Figure 10e shows the memory usage with respect to the Zipf factor. Again, when the Zipf factor is small, the data is sparse and thus not many prefixes can be shared. When the Zipf factor becomes larger, the data becomes more skewed, and the PAT becomes smaller due to the more sharing of the prefixes. The baseline uses constant memory in the maintenance, since it only needs to load the current sliding window into main memory.

In summary, incremental maintenance of a PAT is scalable in both runtime and space usage with respect to the size of sliding window.

5.4 The effect of the order of dimensions

We also tested the effect of different orders of dimensions on the size of the PATs and the runtime of PAT construction. We made up a synthetic data set of 10

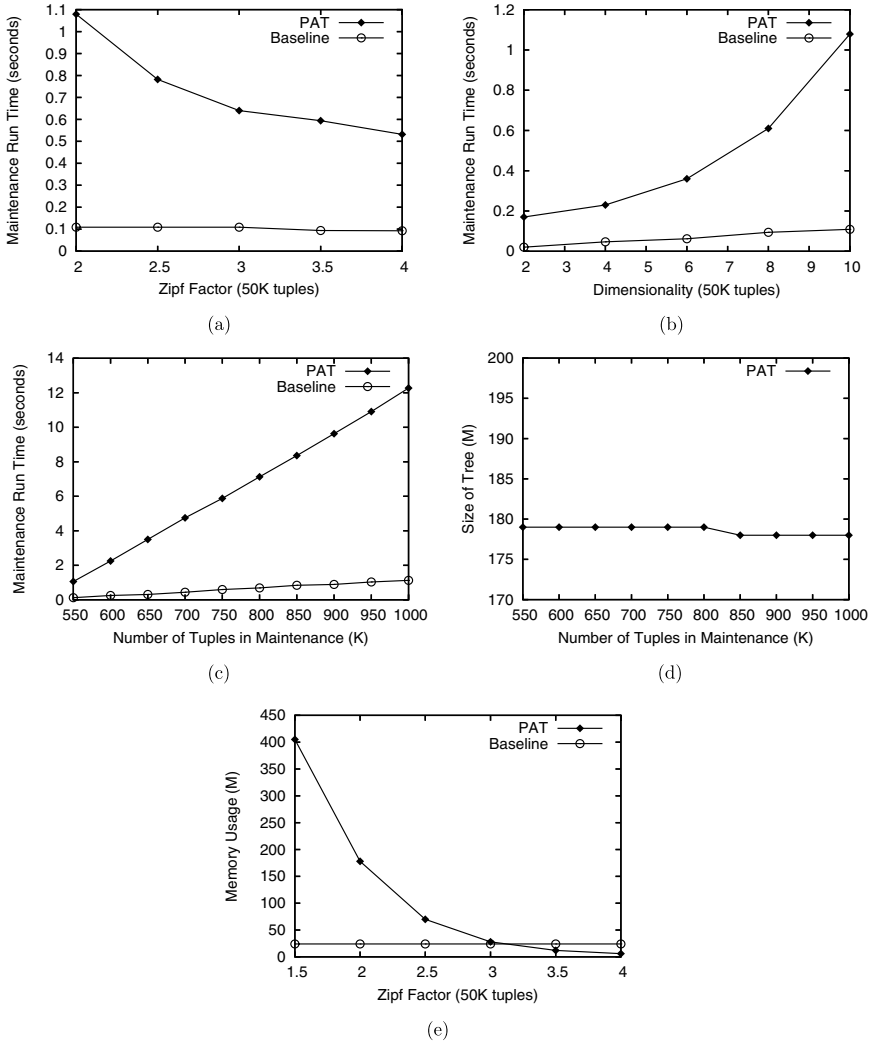

Fig. 10 Results on incremental maintenance. **a** Runtime vs. Zipf factor (dim = 10, cardinality = 10, no. of new tuples = 50 K). **b** Runtime vs. dimensionality (Zipf = 2, cardinality = 10, no. of new tuples = 50 K). **c** Runtime vs. no. of new tuples (Zipf = 2, dim = 10, cardinality = 10). **d** Size of tree vs. no. of new tuples (Zipf = 2, dim = 10, cardinality = 10). **e** Memory usage vs. Zipf factor (dim = 10, cardinality = 10, no. of new tuples = 50 K)

dimensions, Zipf factor 3 and 1 million tuples. The i-th dimension ($1 \leq i \leq 10$) has a cardinality of i. We tested the effects of the following 4 orders of dimensions:

- R_1: cardinality ascending order;
- R_2: cardinality descending order;
- R_3: D_5-D_6-D_4-D_7-D_3-D_8-D_2-D_9-D_1-D_{10}; and
- R_4: D_1-D_{10}-D_2-D_9-D_3-D_8-D_4-D_7-D_5-D_6.

Table 3 The effect of orders of dimensions

Order	Runtime	No. of nodes	Tree size (bytes)
R_1	16.37	6, 433	1, 521, 640
R_2	16.67	16, 441	2, 548, 404
R_3	17.14	8, 694	2, 180, 736
R_4	16.74	8, 575	1, 812, 868

The results are shown in Table 3. The PAT construction time is insensitive to the order of dimensions, since the number of tree node accesses is basically the same no matter which order is used.

Both the number of nodes in the PAT and the size of the tree are sensitive to the orders. With order R_1, putting dimensions of low cardinality ahead strongly facilitates the prefix sharing, and leads to the smallest number of nodes. As discussed at the end of Sect. 3.1, at level i, the size of the tree nodes is proportional to the sum of cardinalities in dimensions D_{i+1} to D_n. Therefore, the average size of tree nodes using order R_1 is the largest. However, the advantage of reduction on number of nodes well overcomes the disadvantage of large tree node size. Thus, order R_1 achieves the smallest tree. Order R_2 suffers from deficiency in sharing the prefixes. Although its average tree node size is the smallest, the tree size turns out to be the largest. Orders R_3 and R_4 stay in between.

Based on this experiment, we recommend ordering the dimensions in cardinality ascending order to explore possible sharing of prefixes. However, there is no theoretical guarantee that the cardinality ascending order always leads to the smallest tree.

5.5 Results on the weather data set

We also tested the PAT construction using the well-accepted weather data set [19], which contains 1, 015, 367 tuples and nine dimensions. The dimensions with the cardinalities of each dimension are as follows: station-id (7037), longitude (352), solar-altitude (179), latitude (152), present-weather (101), day (30), weather-change-code (10), hour (8), and brightness (2). Eight data sets with 2–9 dimensions are generated by projecting the weather data set on the first k dimensions ($1 \leq k \leq 9$). Figure 11 shows the results.

Interestingly, the size of the PAT on the complete data set is only 263 MB, comparable to the size of QC-tree (241.2 MB as reported in [26]), a recently developed data cube compression method. However, to construct a QC-tree, the base table has to be scanned and sorted multiple times, and the incremental maintenance of a QC-tree is more costly than PAT. The PAT construction is also much faster than computing the whole cube using BUC but slower than Baseline (Fig. 11b). As indicated in [25], construction of a quotient cube is slower than BUC, since extra work is needed to achieve compression.

In summary, the experimental results on the real data set are consistent with the observations that we obtained from the experiments on the synthetic data sets.

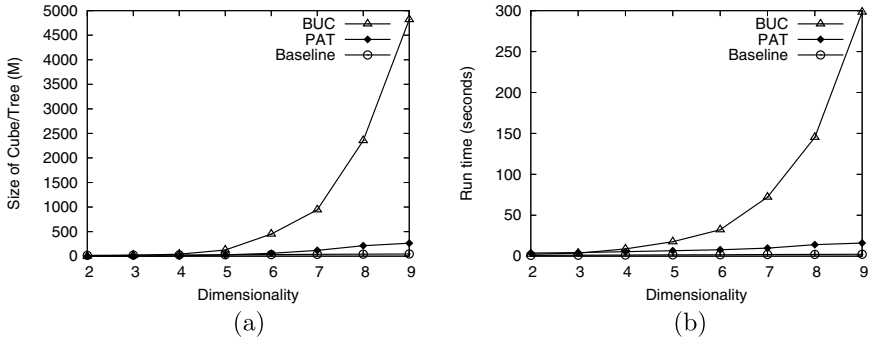

Fig. 11 Results on real data set weather. **a** Size of the tree. **b** Construction time

5.6 Summary

Based on the above experimental results, we have the following observations. First, the size of a PAT is substantially smaller than that of a data cube. That makes the PAT feasible in space for data streams. Second, our algorithms for constructing and incrementally maintaining a PAT are efficient and highly scalable for data streams. The construction and maintenance cost is dramatically smaller than the cost of materializing the whole cube. Third, query answering using a PAT is comparable to the best cases using a full cube. It is much faster than the baseline method. The PAT approach can be regarded as a good tradeoff between the construction/maintenance cost and the query answering performance.

6 Conclusions

Online warehousing data streams and answering ad hoc aggregate queries are interesting and challenging research problems with broad applications. In this paper, we propose a novel PAT data structure to construct an online data warehouse. Efficient algorithms are developed to construct and incrementally maintain a PAT over a data stream, and answer various ad hoc aggregate queries. We present a systematic performance study to examine the effectiveness and the efficiency of our design.

Acknowledgements We are grateful to the reviewers for their insightful comments, which help to improve the quality of this paper. Jian Pei's research is supported in part by NSERC Discovery Grant 312194-05, NSF Grant IIS-0308001, and a President's Research Grant of Simon Fraser University. Ke Wang's research is supported in part by a grant from NSERC. All opinions, findings, conclusions and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

1. Arasu A, Manku GS (2004) Approximate counts and quantiles over sliding windows. In: Proceedings of the 23rd ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems (PODS'04), Paris, France

2. Babcock B, Babu S, Datar M, Motwani R, Widom J (2002) Models and issues in data stream systems. In: Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems (PODS'02), Madison, WI
3. Babu S, Widom J (2001) Continuous queries over data streams. SIGMOD Record 30:109–120
4. Barbara D, Sullivan M (1997) Quasi-cubes: exploiting approximation in multidimensional databases. SIGMOD Record 26:12–17
5. Barbara D, Wu X (2000) Using loglinear models to compress datacube. In: 'WAIM'2000', pp 311–322
6. Beyer K, Ramakrishnan R (1999) Bottom-up computation of sparse and iceberg cubes. In: Proceedings of the 1999 ACM-SIGMOD international conference on management of data (SIGMOD'99), Philadelphia, PA, pp 359–370
7. Chang JH, Lee WS (2003) Finding recent frequent itemsets adaptively over online data streams. In: KDD '03: Proceedings of the 9th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM Press, pp 487–492
8. Chaudhuri S, Dayal U (1997) An overview of data warehousing and OLAP technology. SIGMOD Record 26:65–74
9. Chen Y, Dong G, Han J, Wah BW, Wang J (2002) Multi-dimensional regression analysis of time-series data streams. In: Proceedings of the 2002 international conference on very large data bases (VLDB'02), Hong Kong, China
10. Cohen S, Nutt W, Serebrenik A (1999) Rewriting aggregate queries using views. In: Proceedings of the 18th ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems, Philadelphia, Pennsylvania, ACM Press, pp 155–166
11. Cormode G, Korn F, Muthukrishnan S, Srivastava D (2003) Finding hierarchical heavy hitters in data streams. In: Proceedings of the 19th international conference on very large data bases (VLDB'03), Berlin, Germany
12. Cormode G, Muthukrishnan S (2003). What's hot and what's not: tracking most frequent items dynamically. In: PODS '03: Proceedings of the 22nd ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems, ACM Press, New York, NY, USA, pp 296–306
13. Datar M, Gionis A, Indyk P, Motwani R (n.d.) Maintaining stream statistics over sliding windows (extended abstract), citeseer.nj.nec.com/491746.html
14. Dobra A, Garofalakis M, Gehrke J, Rastogi R (2002) Processing complex aggregate queries over data streams. In: Proceedings of the 2002 ACM-SIGMOD international conference management of data (SIGMOD'02), Madison, Wisconsin
15. Gehrke J, Korn F, Srivastava D (2001) On computing correlated aggregates over continuous data streams. In: Proceedings of the 2001 ACM-SIGMOD international conference management of data (SIGMOD'01), Santa Barbara, CA, pp 13–24
16. Giannella C, Han J, Pei J, Yu P (2004) Mining frequent patterns in data streams at multiple time granularities. In: Kargupta H, Joshi A, Sivakumar K, Yesha Y (eds) Next generation data mining, AAAI/MIT
17. Gray J, Bosworth A, Layman A, Pirahesh H (1996) Data cube: a relational operator generalizing group-by, cross-tab and sub-totals. In: Proceedings of the 1996 international conference data engineering (ICDE'96), New Orleans, Louisiana, pp 152–159
18. Gupta A, Mumick IS, Subrahmanian VS (1993) Maintaining views incrementally. In: Buneman P, Jajodia S (eds) Proceedings of the 1993 ACM SIGMOD international conference on management of data, Washington, D.C., ACM Press, pp 157–166
19. Hahn CJ, Warren SG, London J (1994) Edited synoptic cloud reports from ships and land stations over the globe, 1982–1991. Available at http://cdiac.esd.ornl.gov/.
20. Han J, Pei J, Dong G, Wang K (2001) Efficient computation of iceberg cubes with complex measures. In: Proceedings of the 2001 ACM-SIGMOD international conference on management of data (SIGMOD'01), Santa Barbara, CA, pp 1–12
21. Han J, Pei J, Yin Y (2000) Mining frequent patterns without candidate generation. In: Proceedings of the 2000 ACM-SIGMOD international conference management of data (SIGMOD'00), Dallas, TX, pp 1–12
22. Harinarayan V, Rajaraman A, Ullman JD (1996) Implementing data cubes efficiently. In: Proceedings of the 1996 ACM-SIGMOD international conference on management of data (SIGMOD'96), Montreal, Canada, pp 205–216

23. Johnson T, Shasha D (1997) Some approaches to index design for cube forests. Bull Tech Comm Data Eng 20:27–35
24. Karp RM, Papadimitriou CH, Shenker S (2003) A simple algorithm for finding frequent elements in streams and bags. ACM Trans Database Syst (TODS) 28(1):51–55
25. Lakshmanan L, Pei J, Han J (2002) Quotient cube: How to summarize the semantics of a data cube. In: Proceedings of the 2002 international conference very large data bases (VLDB'02), Hong Kong, China
26. Lashmanan L, Pei J, Zhao Y (2003) QC-Trees: An efficient summary structure for semantic OLAP. In: Proceedings of the 2003 ACM SIGMOD international conference on management of data (SIGMOD'03), San Diego, California
27. Levy AY, Mendelzon AO, Sagiv Y, Srivastava D (1995) Answering queries using views. In: Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems, San Jose, California, ACM Press, New York, pp 95–104
28. Manku GS, Motwani R (2002) Approximate frequency counts over data streams. In: Proceedings of the 2002 international conference on very large data bases (VLDB'02), Hong Kong, China
29. Mendelzon AO, Vaisman AA (2000) Temporal queries in OLAP. In: Abbadi AE, Brodie ML, Chakravarthy S, Dayal U, Kamel N, Schlageter G, Whang K-Y (eds) VLDB 2000, Proceedings of the 26th international conference on very large data bases, Cairo, Egypt, Morgan Kaufmann, pp 242–253
30. Mumick IS, Quass D, Mumick BS (1997) Maintenance of data cubes and summary tables in a warehouse. In: Peckham J (ed) SIGMOD 1997, Proceedings ACM SIGMOD international conference on management of data, Tucson, Arizona, USA, ACM Press, pp 100–111
31. Quass D, Gupta A, Mumick IS, Widom J (1996) Making views self-maintainable for data warehousing. In: Proceedings of the 1996 international conference parallel and distributed information systems, Miami Beach, Florida, pp 158–169
32. Quass D, Widom J (1997) On-line warehouse view maintenance. In: Peckham J (ed) SIGMOD 1997, Proceedings ACM SIGMOD international conference on management of data, Tucson, Arizona, USA, ACM Press, pp 393–404
33. Ross K, Srivastava D (1997) Fast computation of sparse datacubes. In: Proceedings of the 1997 international conference very large data bases (VLDB'97), Athens, Greece, pp 116–125
34. Ross KA, Zaman KA (2000) Optimizing selections over datacubes. In: Statistical and scientific database management, pp 139–152. citeseer.nj.nec.com/article/ross98optimizing.html
35. Roussopoulos N, Kotidis Y, Roussopoulos M (1997) Cubetree: Organization of and bulk updates on the data cube. In: Peckham J (ed) SIGMOD 1997, Proceedings ACM SIGMOD international conference on management of data, Tucson, Arizona, USA, ACM Press, pp 89–99
36. Sarawagi S (1997) Indexing OLAP data. Bull Tech Com Data Eng 20:36–43
37. Shanmugasundaram J, Fayyad U, Bradley PS (1999) Compressed data cubes for OLAP aggregate query approximation on continuous dimensions. In: Proceedings of the 5th ACM SIGKDD international conference on knowledge discovery and data mining, ACM Press, San Diego, California, United States, pp 223–232
38. Sismanis Y, Roussopoulos N, Deligiannakis A, Kotidis Y (2002) Dwarf: Shrinking the petacube. In: Proceedings of the 2002 ACM-SIGMOD international conference management of data (SIGMOD'02), Madison, Wisconsin
39. Sristava D, Dar S, Jagadish HV, Levy AV (1996) Answering queries with aggregation using views. In: Proceedings of the 1996 international conference very large data bases (VLDB'96), Bombay, India, pp 318–329
40. Teng W-G, Chen M-S, Yu PS (2003) A regression-based temporal pattern mining scheme for data streams. In: Proceedings of the 19th international conference on very large data bases (VLDB'03), Berlin, Germany
41. Vitter JS, Wang M, Iyer BR (1998) Data cube approximation and historgrams via wavelets. In: Proceedings of the 1998 international conference on information and knowledge management (CIKM'98), Washington DC, pp 96–104
42. Wang W, Lu H, Feng J, Yu JX (2002) Condensed cube: An effective approach to reducing data cube size. In: Proceedings of the 2002 international conference on data engineering (ICDE'02), San Fransisco, CA

43. Widom J (1995) Research problems in data warehousing. In: Proceedings of the 4th international conference on information and knowledge management, Baltimore, Maryland, pp 25–30
44. Yu JX, Chong Z, Lu H, Zhou A (2004) False positive or false negative: Mining frequent itemsets from high speed transactional data streams. In: Proceedings of the 30th international conference on very large data bases (VLDB'04), Toronto, ON, Canada
45. Zhao Y, Deshpande PM, Naughton JF (1997) An array-based algorithm for simultaneous multidimensional aggregates. In: Proceedings of the 1997 ACM-SIGMOD international conference management of data (SIGMOD'97), Tucson, Arizona, pp 159–170

Author Biographies

Moonjung Cho is a Ph.D. candidate in the Department of Computer Science and Engineering at State University of New York at Buffalo. She obtained her Master from same university in 2003. She has industry experiences as associate researcher for 4 years. Her research interests are in the area of data mining, data warehousing and data cubing. She has received a full scholarship from Institute of Information Technology Assessment in Korea.

Jian Pei received the Ph.D. degree in Computing Science from Simon Fraser University, Canada, in 2002. He is currently an Assistant Professor of Computing Science at Simon Fraser University, Canada. In 2002–2004, he was an Assistant Professor of Computer Science and Engineering at the State University of New York at Buffalo, USA. His research interests can be summarized as developing advanced data analysis techniques for emerging applications. Particularly, he is currently interested in various techniques of data mining, data warehousing, online analytical processing, and database systems, as well as their applications in bioinformatics. His current research is supported in part by Natural Sciences and Engineering Research Council of Canada (NSERC) and National Science Foundation (NSF). He has published over 70 papers in refereed journals, conferences, and workshops, has served in the program committees of over 60 international conferences and workshops, and has been a reviewer for some leading academic journals. He is a member of the ACM, the ACM SIGMOD, and the ACM SIGKDD.

Ke Wang received Ph.D from Georgia Institute of Technology. He is currently a professor at School of Computing Science, Simon Fraser University. Before joining Simon Fraser, he was an associate professor at National University of Singapore. He has taught in the areas of database and data mining. Ke Wang's research interests include database technology, data mining and knowledge discovery, machine learning, and emerging applications, with recent interests focusing on the end use of data mining. This includes explicitly modeling the business goal (such as profit mining, bio-mining and web mining) and exploiting user prior knowledge (such as extracting unexpected patterns and actionable knowledge). He is interested in combining the strengths of various fields such as database, statistics, machine learning and optimization to provide actionable solutions to real life problems. Ke Wang has published in database, information retrieval, and data mining conferences, including SIGMOD, SIGIR, PODS, VLDB, ICDE, EDBT, SIGKDD, SDM and ICDM. He is an associate editor of the IEEE TKDE journal and has served program committees for international conferences including DASFAA, ICDE, ICDM, PAKDD, PKDD, SIGKDD and VLDB.