OLAP on Sequence Data

Eric Lo[1], Ben Kao[2], Wai-Shing Ho[2], Sau Dan Lee[2], Chun Kit Chui[2], and David W. Cheung[2]

[1] Department of Computing, The Hong Kong Polytechnic University
ericlo@comp.polyu.edu.hk
[2] Department of Computer Science, The University of Hong Kong
{kao, wsho, sdlee, ckchui, dcheung}@cs.hku.hk

Abstract. Many kinds of real-life data exhibit logical ordering among their data items and are thus sequential in nature. However, traditional online analytical processing (OLAP) systems and techniques were not designed for sequence data and they are incapable of supporting sequence data analysis. In this paper, we propose the concept of Sequence OLAP, or S-OLAP for short. The biggest distinction of S-OLAP from traditional OLAP is that a sequence can be characterized not only by the attributes' values of its constituting items, but also by the subsequence/substring patterns it possesses. This paper studies many aspects related to Sequence OLAP. The concepts of *sequence cuboid* and *sequence data cube* are introduced. A prototype S-OLAP system is built in order to validate the proposed concepts. The prototype is able to support "pattern-based" grouping and aggregation, which is currently not supported by any OLAP system. The implementation details of the prototype system as well as experimental results are presented.

1 Introduction

Traditional online analytical processing (OLAP) systems process records in a fact table and summarize their key statistics with respect to certain measure attributes. A user can select a set of dimension attributes and their corresponding levels of abstraction and an OLAP system will partition the data records based on those dimension attributes and abstraction levels. Records that share the same values in those dimension attributes (w.r.t. the selected abstraction levels) are grouped together. Aggregate functions (such as sum, average, count) are then applied to the measure attributes of the records in each group. An OLAP system then reports a summary (a.k.a. *cuboid*) by tabulating the aggregate values for all possible groups. OLAP is a powerful data analysis tool because it allows users to "navigate" or "explore" different levels of summarization by interactively changing the set of dimension attributes and their abstraction levels. In other words, users can navigate from one cuboid to another interactively in order to obtain the most interesting statistics through a set of pre-defined OLAP operations (such as roll-up, drill-down, slice, and dice).

Although powerful, existing OLAP systems only handle independent records. Many kinds of real-life data, however, exhibit logical ordering among their data items and are thus sequential in nature. Examples of sequence data include stock market data, web server access logs and RFID logs such as those generated by a commodity tracking

system in a supply chain. Similar to conventional data, there is a strong demand to warehouse and to analyze the vast amount of sequence data in a user-friendly and efficient way. Unfortunately, current OLAP systems and technologies were not designed for sequence data and they are incapable of supporting sequence data analysis. In this paper we study the issues of building a "Sequence OLAP" system, or an S-OLAP system for short.

[Applications] An S-OLAP system that analyzes sequence data has many applications. One motivating application is transportation planning. Today, many cities have implemented electronic transportation payment systems using RFID technology. Examples include Hong Kong's Octopus system, Japan's Kansai Thru Pass system and Washington DC's SmarTrip system. In those cities, every passenger carries a smart card (e.g., a card with a passive RFID chip [5]), which can be used as a form of electronic money to pay for various kinds of transportation (e.g., bus/subway). The electronic payment system generates huge volumes of data everyday (e.g., Hong Kong's Octopus system collected over 7 million transactions per day in 2003 [1]). The transactions performed by a user each day can form logical sequences in many different ways. For example, a sequence can be formed by clustering a user's transactions over 1-day, 1-week or 1-month periods.

With the enormous amount of sequence data available, an OLAP system that performs sequence summarizations would be of great value. For instance, if a transport-planning manager of Washington Metropolitan Area Transit Authority (WMATA) wants to rearrange the subway schedule, he may pose a query asking "the number of *round-trip* passengers and their distributions over all origin-destination station pairs within 2007 Quarter 4". Figure 1 presents an artificial WMATA dataset. We assume that a passenger registers an event/transaction into the system every time she enters (action = "in") or leaves a station (action = "out") through the turnstiles. Therefore, the *round-trip* semantics can be captured by the pattern (X, Y, Y, X), which means that all passengers who have first entered any station X (e.g., Pentagon), exited at any station Y (e.g., Wheaton), and then entered station Y (Wheaton) again and returned to station X (Pentagon) should be grouped together.[3] Furthermore, for each possible combination of X and Y, the aggregated number of passengers is counted and a tabulated view of the sequence data like the one shown in Figure 2 should be returned by the S-OLAP system.

The S-OLAP system should also allow a user to interactively change the grouping pattern and be able to answer iterative queries efficiently. For example, after studying the round-trip distribution in Figure 2, the manager might observe that there is a high concentration of people taking round-trips from Pentagon to Wheaton. He might want to further investigate whether those passengers would take one more follow-up trip and if so where they usually go. He can view this distribution by first performing a traditional *slice* OLAP operation on (Pentagon, Wheaton, Wheaton, Pentagon), followed by changing the grouping pattern to (X, Y, Y, X, X, Z), where the two newly appended symbols X, Z denote the third trip from station X (Pentagon) to any station Z.

[3] The formal query specification will be discussed shortly in Section 3.2 and a similar query specification is shown in Figure 3.

S-OLAP systems have many more applications. As another example, a marketing manager of an e-commerce company can use an S-OLAP system to identify some "lost-sales" page-clicking sequences by posing S-OLAP queries such as: "for all possible pairs of page combinations within 2007 Quarter 4, show the number of visitors per day, with a visiting pattern of (P, K)" on its web server access log, where P denotes any product page and K denotes any "killer page"[4] (e.g., a logout page). Again, the manager can interactively change the grouping pattern and the S-OLAP system should be able to efficiently answer those iterative queries so as to help the manager to drill-down into the actual reasons for the lost-sales.

[Contributions] From the above application examples, we can see that the biggest distinction of an S-OLAP system from a traditional OLAP system is that a sequence can be characterized not only by the attributes' values of its constituting events, but also by the subsequence/substring patterns it possesses. In other words, an S-OLAP system can support "pattern-based" grouping and aggregation — a very powerful concept and capability that is not supported by traditional OLAP systems.

To the best of our knowledge, the building of an S-OLAP system for analyzing sequence data has not been addressed previously in the research literature or in commercial products. This paper studies many aspects related to the design and implementation of an S-OLAP system. Our contributions can be summarized as follows:

1. The concept of Sequence OLAP (or S-OLAP for short) is presented. This includes the discussion of what a "Sequence Cuboid" (or S-cuboid for short) is, the relationships between different S-cuboids, and the concept of "Sequence Data Cube" (or S-cube for short).
2. Six S-OLAP-specific operations are identified. In traditional OLAP systems, users "navigate" or "explore" different levels of summarization (i.e., different cuboids) through a set of user-friendly operations (such as roll-up, drill-down, slice, and dice). In this paper we present six operations that are specific to S-OLAP, namely, (1) APPEND, (2) DE-TAIL, (3) PREPEND, (4) DE-HEAD, (5) PATTERN-ROLL-UP and (6) PATTERN-DRILL-DOWN. The six S-OLAP operations modify the grouping patterns and/or the abstraction level of the elements inside the grouping patterns such that users can interactively view the summarized data from different perspectives. In other words, the six operations allow users to navigate from one S-cuboid to another in the S-cube space with ease.
3. The implementation details of an S-OLAP prototype system are presented. The prototype system serves as an initial solution of the proposed Sequence OLAP concept. The architecture of the prototype system and two different approaches of computing S-cuboids are presented. While the first approach serves as a baseline of computing an S-cuboid, the second approach makes use of the concept of inverted index to facilitate the computation of S-cuboids and the processing of the six S-OLAP operations.
4. Comprehensive experiments have been conducted on the prototype system on both real and synthetic sequence data and the experimental results are presented. The

[4] This query answers a KDD-Cup 2000 data mining question [11] in an OLAP data exploratory way.

experiments on real data demonstrate how our proposed S-OLAP system answers some real life queries by performing sequence data analysis on real life sequence data. The experiments on synthetic data evaluate the performance of the S-OLAP prototype system under different settings.

5. Being the first to address the problem of Sequence OLAP, we have discovered a lot of interesting research issues throughout the project. As the last contribution of this paper, we present the research issues we have found. Overall, we believe that this paper serves as an interesting starting point towards more sophisticated and more general solutions for OLAP on sequence data.

[Roadmap] The rest of the paper is organized as follows. Section 2 gives an overview of work that is related to Sequence OLAP. Section 3 describes the concept of Sequence OLAP. Section 4 describes the technical details of the prototype S-OLAP system. Section 5 reports experimental and performance results. We discuss some research issues of Sequence OLAP in Section 6 and conclude our study in Section 7.

time	*card-id*	*location*	*action*	*amount*
2007-01-01T00:01	688	Glenmont	in	0
⋮	⋮	⋮	⋮	⋮
2007-10-01T00:01	23456	Pentagon	in	0
2007-10-01T00:02	9876	Pentagon	in	0
⋮	⋮	⋮	⋮	⋮
2007-10-01T01:59	9876	Wheaton	out	-2
⋮	⋮	⋮	⋮	⋮
2007-10-02T22:46	52	Wheaton	deposit	100
⋮	⋮	⋮	⋮	⋮
2007-12-25T20:48	6544	Wheaton	out	-3.5
⋮	⋮	⋮	⋮	⋮

Fig. 1. An event database

2 Related Work

Sequence Databases. Database systems used to be no formal support of sequence data until PREDATOR [19], [20]. PREDATOR extended the ADT approach of object-relational systems by treating the sequence type as an enhanced ADT (EADT). PREDATOR treats sequence data type as first class citizen and its query language SEQUIN includes a set of sequence operators for querying and manipulating sequences.

Since applications often involve both relational data and sequence data, the DEVise system [16] was proposed to model sequences as sorted relations. By storing sequence data using normal relations, it is much easier to query a combination of relational tables

(X, Y, Y, X)	COUNT
(Clarendon,Pentagon,Pentagon,Clarendon)	5,432
(Clarendon,Wheaton,Wheaton,Clarendon)	7,654
⋮	⋮
(Pentagon,Glenmont,Glenmont,Pentagon)	4,321
(Pentagon,Wheaton,Wheaton,Pentagon)	200,125
⋮	⋮
(Wheaton,Pentagon,Pentagon,Wheaton)	6,543

Fig. 2. A sequence OLAP query result

and data sequences. This approach enables more integrated optimization and evaluation. SRQL is an extension of SQL. It is used in the DEVise system for supporting queries on mixtures of sequences and relations. However, DEVise did not address the issues of warehousing and efficient analysis of sequence data. Moreover, SRQL itself is not expressive enough to express queries with complicated patterns such as recurring patterns. In view of this, [18] extended SRQL and proposed SQL-TS. With SQL-TS, one can express sophisticated sequential pattern queries. However, [18] did not address the issues of sequence data analysis as well.

OLAP. [8] first described the data-cube operator. Since then, a large number of papers have been written on the subject. Many of them focus on efficient algorithms for data cube construction. A few examples include: iceberg cube [4], bottom-up cube computation [2], and top-down cube computation [17]. None of these studies, however, address sequence data.

OLAP on unconventional data. In [7], the authors addressed how to store and analyze massive RFID-enabled workflow data, which is a very special type of sequence data. Their proposal made heavy use of a special property of workflow data that individual items in a supply chain tend to move together in bulky mode. Based on that property, [6] introduced the concept of *RFID-Cubiods*, which is a way to store RFID workflow data in relational databases that supports efficient data compression and specialized workflow data analysis. Stream data is another kind of sequence data. In [3], the authors studied how to build data cubes for time-series stream data. Nonetheless, none of these work address the problem of pattern-based grouping and analysis. Recently, Wiwatwattana et al. [22] discuss how to perform OLAP operations on XML data. Due to certain special properties of XML data (e.g., an XML element may have missing or repeated sub-elements), the authors point out that XML data is non-summarizable [12]. That is, a coarser aggregate cannot be computed solely from the corresponding finer aggregates. In [22], they approach the problem by proposing several aggregation relaxation models such that cube data becomes summarizable under such restricted models.

3 Sequence OLAP

In this section we first give an introduction to sequence data in Section 3.1. Then, we explain the concept of sequence cuboid in Section 3.2, which is the key concept in se-

quence OLAP. Afterwards, we explain the six proposed S-OLAP operations in Section 3.3. We describe the relationships between different sequence cuboids and the concept of a sequence data cube in Section 3.4. How these concepts could be implemented is discussed in Section 4. In the rest of this paper, we use the transportation planning application discussed in Section 1 as our running example.

3.1 Preliminary

The raw data of an S-OLAP system is a set of **events** that are deposited in an event database. An event e is modeled as an individual record/tuple in a way similar to those stored in a fact table in a traditional OLAP system. Figure 1 presents an event database for our running example. In Figure 1, an event is in the form of (*time*, *card-id*, *location*, *action*, *amount*). We assume that each passenger has only one smart card. Therefore, the first event in Figure 1 shows that a passenger with *card-id* 688 has entered Glenmont station (*action*="in") at time 00:01 on January 1st, 2007. Since the data is collected and consolidated from each station, we assume that events in the event database are ordered by the *location* and *time* attributes.

Similar to traditional OLAP systems, an event in an S-OLAP system consists of a number of **dimensions** and **measures** and each dimension may be associated with a **concept hierarchy**. In Figure 1, the attributes *time*, *card-id*, *location* and *action* are dimensions and the attribute *amount* is a measure. In our running example, we assume that the *location* attribute is associated with a concept hierarchy of two abstraction levels station $\rightarrow$ district, the *card-id* attribute is associated with a concept hierarchy individual $\rightarrow$ fare-group (e.g., student/regular/senior), and the *time* attribute is associated with a concept hierarchy time $\rightarrow$ day $\rightarrow$ week.

If there is a logical ordering among a set of events, the events can form a **sequence**. In our running example, a logical ordering could be based on the *time* attribute. Therefore, the traveling history of passenger 688 can be denoted by the sequence which consists of all the events with *card-id* 688, ordered by the *time* attribute.

3.2 Sequence Cuboid

In traditional OLAP, a cuboid is formed by partitioning records based on a set of dimension attributes, each under a specific abstraction level. In sequence OLAP, an S-cuboid is a logical view of sequence data at a particular degree of summarization in which sequences can be characterized not only by the attributes' values, but also by the subsequence/substring patterns they possess.

Figure 3 shows a cuboid specification Q_1 which is used as our running example. Q_1 is similar to the first example S-OLAP query we presented in the Introduction. Q_1 asks for the number of round-trip passengers and their distributions over all origin-destination station pairs for each day and for each fare-group, within Quarter 4 of 2007. Figure 4 shows the conceptual view of the building process of an S-cuboid for Q_1 and the details are explained below.

The specification of an S-cuboid is inspired by SQL-TS [18] and consists of six parts:(1) WHERE clause (2) CLUSTER BY clause, (3) SEQUENCE BY clause, (4)

```
1.  SELECT                COUNT(*)
2.  FROM                  Event
3.  WHERE                 time >= 2007-10-01T00:00 AND
4.                        time < 2007-12-31T24:00
5.  CLUSTER BY            card-id AT individual,
6.                        time AT day
7.  SEQUENCE BY           time ASCENDING
8.  SEQUENCE GROUP BY     card-id AT fare-group,
9.                        time AT day
10. CUBOID BY             SUBSTRING (X, Y, Y, X) WITH
11.                       X AS location AT station,
12.                       Y AS location AT station
13.                       LEFT-MAXIMALITY (x1, y1, y2, x2) WITH
14.                       x1.action = "in" AND
15.                       y1.action = "out" AND
16.                       y2.action = "in" AND
17.                       x2.action = "out"
```

Fig. 3. S-cuboid specification Q_1

SEQUENCE GROUP BY clause, (5) CUBOID BY clause and (6) Aggregation Functions.[5]

1. [Selection] A WHERE clause is adopted from SQL in order to select only events of interest. Lines 3 and 4 in Figure 3 specify that only events within 2007 Q4 are selected (see Figure 4 Step 1).
2. [Clustering] A CLUSTER BY clause is borrowed from [18] in order to specify events that are elements of a sequence to be clustered together. Each attribute in the CLUSTER BY clause is associated with an abstraction level in a concept hierarchy. Lines 5 and 6 in Figure 3 specify that events should be clustered together according to the attributes *card-id* and *time*, at the abstraction levels of individual and day, respectively. In other words, events that shared the same *card-id* value and happened in the same day should form a cluster. However, events in the same cluster are not necessarily ordered at this stage (see Figure 4 Step 2).
3. [Sequence Formation] A SEQUENCE BY clause is borrowed from [18] in order to form a sequence from a cluster of events. Events in each cluster form exactly one sequence. For example, Line 7 in Figure 3 specifies that the clustered events should form sequences according to their occurrence *time* (see Figure 4 Step 3).
4. [Sequence Grouping] A SEQUENCE GROUP BY clause is introduced such that sequences whose events share the same dimensions' values are further grouped together to form a **sequence group**. The attributes in the SEQUENCE GROUP BY clause form the set of **global dimensions** and each of them is associated with an abstraction level in the concept hierarchy. For instance, Lines 8 and 9 in Figure 3 specify that individual user sequences within the same fare-group and whose events occurred in the same day should form a sequence group (see Figure 4 Step 4). If the SEQUENCE GROUP BY clause is not specified, all sequences form a single sequence group.

[5] Although the clauses CLUSTER BY and SEQUENCE BY also exist in TS-SQL, they have different semantics in S-cuboid specification.

5. [Pattern Grouping] A CUBOID BY clause is introduced in order to specify the logical view of the sequence data that the user wants to see. The CUBOID BY clause consists of three sub-parts: (a) Pattern Template, (b) Cell Restriction and (c) Matching Predicate. Step 5 in Figure 4 illustrates pattern grouping and the details are explained below.

   (a) Pattern Template. A pattern template consists of a sequence of **symbols**, each associated with a domain of values. The domain of values is specified as the domain of an attribute at certain abstraction level. The set of *distinct* symbols in a pattern template form the set of **pattern dimensions**. The set of pattern dimensions together with the set of global dimensions define the partitioning of an S-cuboid (i.e., the cells of an S-cuboid).

   The pattern template defines the format of the substring/ subsequence patterns to be matched against the data sequences. By SUBSTRING(X, Y, Y, X) or SUBSEQUENCE(X, Y, Y, X), we mean a substring/subsequence pattern template (X, Y, Y, X) is specified. Lines 10 to 12 in Figure 3 show an example substring pattern template with two pattern dimensions X and Y, each represents a *location* value at the station abstraction level.

   Each cell is associated with a **pattern**. A pattern can be instantiated from a pattern template by a set of values that are associated with the symbols. If two symbols in a pattern template are the same, then they should be instantiated with the same value. For example, the pattern (Pentagon,Wheaton,Wheaton,Pentagon) is an instantiation of pattern template (X, Y, Y, X) but the pattern (Pentagon,Wheaton,Glenmont,Pentagon) is not.

   If a data sequence matches the pattern of a particular cell, and if it further satisfies the cell restriction and the matching predicate ((b) and (c) below), then it is assigned to that cell. Note that since a data sequence may match multiple patterns, it may be assigned to more than one cuboid cell.

   (b) Cell Restriction. The cell restriction defines how to deal with the situations when a data sequence contains multiple occurrences of a cell's pattern and what content of the data sequence should be assigned to the cell (for the purpose of aggregation, to be done later). One type of cell restriction is **left-maximality-matched-go** [18]. For example, when a cell with a substring pattern (a,a) is matched against a data sequence $\langle$aabaa$\rangle$, the **left-maximality-matched-go** cell restriction states that only the first matched substring/subsequence (i.e., the first "aa" in $\langle$aabaa$\rangle$) is assigned to the cell. This cell restriction is specified by the keyword LEFT-MAXIMALITY. In general, depending on the applications, more cell restrictions can be defined. For example, one can define a **left-maximality-data-go** cell restriction where the whole data sequence $\langle$aabaa$\rangle$, not only the matched content $\langle$aa$\rangle$, is assigned to the cell. As another example, we can also define an **all-matched-go** cell restriction where all substrings/subsequences that match the pattern are assigned to the cell (i.e., the two aa's in $\langle$aabaa$\rangle$ are assigned to the cell).

   (c) Matching Predicate. A matching predicate is further introduced for selecting data sequences of interests. In order to specify a predicate, a sequence of **event placeholders** are introduced after the cell restriction. Line 13 in Figure 3 shows an example. The four event placeholders $x1$, $y1$, $y2$ and $x2$ in Line 13 represent the

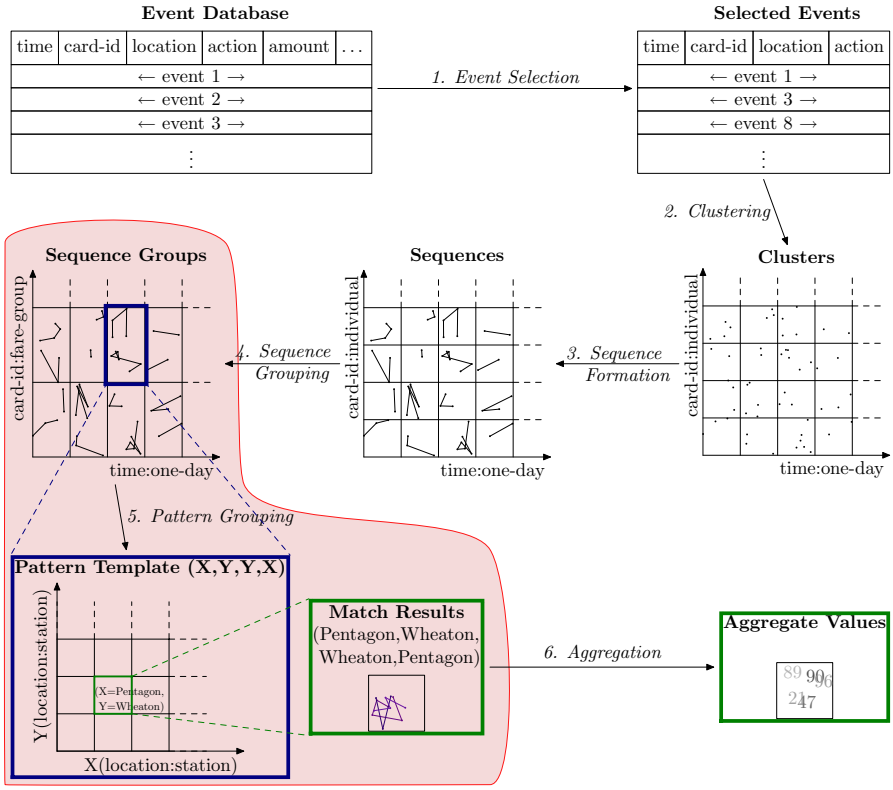

Fig. 4. The conceptual view of building an S-cuboid for Query Q_1

matched events (not only the *location* values) and the predicate in Line 14 specifies that the *action* attribute value of the first matching event $x1$ must equal "in".

6. [Aggregation] Finally, an aggregation function should be specified in the SELECT clause in order to define the aggregate function to be applied to the sequences in each S-cuboid cell. In S-OLAP, the COUNT aggregation function counts the number of matched substrings/subsequences that are assigned to a cell (see Figure 4 Step 6).

Figure 4 illustrates the steps of building an S-cuboid for our example query Q_1. After all steps, a 4D S-cuboid (the shaded area in Figure 4) with two global dimensions (*time*:day, *card-id*:fare-group) and two pattern dimensions (X,Y) is built.

Note that the current S-cuboid specification can be further extended if necessary. For example, other aggregation functions, such as SUM, can be incorporated as long as its semantics is clearly defined. As an example, consider two data sequences $s1$:$\langle e_1, e_2 \rangle$ and $s2$:$\langle e_3, e_4 \rangle$ that are assigned to a cell. We can define SUM as the sum of the measures of all the events occured in $s1$ and $s2$ (i.e., SUM = $\sum_{i=1}^{4} e_i$.*amount*). Alternatively, if desire, we can sum over the first occurring event in each sequence (i.e., SUM = e_1.*amount* +e_3.*amount*). Furthermore, the current S-cuboid specification only supports

substring or subsequence pattern templates. It can be extended so that pattern templates of regular expressions can be supported.

3.3 Sequence OLAP Operations

OLAP is a powerful analytical and decision-supporting tool because it provides a set of operations (e.g., roll-up, drill-down) for a user to interactively modify the cuboid specification (i.e., changing the set of dimension attributes and/or their abstraction levels) and thus enables a user to navigate from one cuboid to another to explore the big cube space with ease.

Since an S-cuboid is defined by a set of global dimensions and pattern dimensions, any changes to these elements transform an S-cuboid to another. In our S-OLAP design, we adopt the same set of OLAP operations, namely, roll-up, drill-down, slice, and dice for the manipulations of the global dimensions. For example, the transport-planning manager might modify the S-OLAP query Q_1 so that passengers are grouped based on individual. To achieve this, we perform a drill-down operation on the global dimension *card-id*, going from the abstraction level fare-group to a lower abstraction level individual.

For pattern manipulation, we propose six S-OLAP operations, namely, APPEND, PREPEND, DE-TAIL, DE-HEAD, PATTERN-ROLL-UP (P-ROLL-UP) and PATTERN-DRILL-DOWN (P-DRILL-DOWN). The first four operations add/remove a pattern symbol to/from a pattern template, while the last two operations modify the abstraction level of pattern dimensions. In particular, the APPEND operation appends a pattern symbol to the end of the pattern template. For example, after learning about the round-trip distribution resulted from Q_1, the manager might observe that there is a particularly high concentration of people traveling round-trip from Pentagon to Wheaton. He might want to further investigate whether those passengers would take one more trip and if so where they usually go. Two APPEND operations plus a modification of the matching predicate give the cuboid specification Q_2 in Figure 5 (only the CUBOID BY clause is shown for brevity). Q_2 transforms the original 4D S-cuboid to a 5D S-cuboid (with global dimensions (*time*:day, *card-id*:fare-group) and pattern dimensions (X,Y,Z), where Z is a new pattern dimension). The other three operations that modify pattern length can be similarly defined: PREPEND — add a symbol to the front of the pattern template; DE-TAIL — remove the last symbol from the pattern template; DE-HEAD — remove the first symbol from the pattern template.

A P-ROLL-UP operation moves the abstraction level of a pattern dimension one level up the concept hierarchy, while a P-DRILL-DOWN operation moves a pattern dimension one level down. As an example, after viewing the trip distribution resulted from the above query Q_2, the transportation manager might find that there are too many station pairs, which makes the distribution reported by the S-cuboid too fragmented. He may want to roll up the *location* pattern dimension Z from the station level to the district level. For that, the P-ROLL-UP changes Line 13 in Figure 5 to: "Z AS *location* AT district".

10. CUBOID BY SUBSTRING (X, Y, Y, X, X, Z) WITH
11. X AS *location* AT station,
12. Y AS *location* AT station,
13. Z AS *location* AT station
14. LEFT-MAXIMALITY($x1$, $y1$, $y2$, $x2$, $x3$, $z1$) WITH
15. $x1$.action = "in" AND $x1$.location = "Pentagon" AND
16. $y1$.action = "out" AND $y1$.location= "Wheaton" AND
17. $y2$.action = "in" AND $y2$.location = "Wheaton" AND
18. $x2$.action = "out" AND $x2$.location = "Pentagon" AND
19. $x3$.action = "in" AND $x3$.location = "Pentagon" AND
20. $z1$.action = "out"

Fig. 5. S-cuboid specification Q_2

3.4 Sequence Data Cube

In traditional OLAP, given a set of dimensions and a set of concept hierarchies associated with the dimensions, we can define a cuboid for each of the possible subsets of the given dimensions and abstraction levels. This results in a lattice of cuboids, each showing the data at a different level of summarization. The lattice of cuboids is then referred to as a data cube. Likewise in S-OLAP, given a set of global and pattern dimensions and a set of concept hierarchies that is associated with the dimensions, we can also define an S-cuboid for each of the possible subsets of the given dimensions and abstraction levels. The set of S-cuboids also form a lattice and we call this lattice a **Sequence Data Cube** (S-cube). Similar to the concept of traditional data cubes, an S-cuboid at a coarser granularity is at a higher level in the lattice, which means it contains fewer global and/or pattern dimensions, or the dimensions are at a higher level of abstraction.

There are two key differences between a traditional data cube and an S-cube. First, there is a finite number of cuboids in a data cube while the number of S-cuboids in an S-cube is infinite. In theory, users may introduce any number of pattern dimensions into the pattern template by S-OLAP operations like APPEND and PREPEND. For example, a pattern template $(X, Y, Z, A, B, C, \dots)$ is possible in which all pattern dimensions refer to the same dimension attribute, say, *location*. Consequently, an S-cube in theory includes an infinite number of S-cuboids although users seldom pose S-OLAP queries with long pattern template in practice.

Second, in general, data in an S-cuboid is *non-summarizable*. That is, an S-cuboid at a higher level of abstraction (i.e., coarser aggregates) cannot be computed solely from a set of S-cuboids that are at a lower level of abstraction (i.e., finer aggregates) without accessing the base data. According to [12], summarizability only holds when the data is *disjoint* and *complete* during data partitioning. However, an S-cuboid may put a data sequence into multiple cells which violates the disjointness requirement. Consider a dataset with only one data sequence s_3 $\langle$Pentagon,Wheaton,Pentagon, Wheaton,Glenmont$\rangle$. If the pattern template is SUBSTRING(X, Y, Z), then s_3 contributes a count of one to all three cells [Pentagon,Wheaton, Pentagon:c_1], [Wheaton,Pentagon,Wheaton:c_2], and [Pentagon, Wheaton,Glenmont: c_3] because s_3 matches all three substrings (c_1,c_2 and c_3 denote the counts of the cells). If we perform a DE-TAIL operation, i.e., the pattern template is changed to SUBSTRING(X, Y), then the cell [Pentagon, Wheaton:

c_4] should have a count of one (as s_3 matches the pattern only once under the left-maximality-matched-go cell restriction). However, if we compute c_4 by aggregation, we get $c_4 = c_1 + c_3 = 2$, an incorrect answer. This observation, which serves as a counter-example, demonstrates that in general, data in an S-cuboid is *non-summarizable*.

As we will show in the next section, the properties of having an infinite number of S-cuboids and non-summarizability make the implementation of an S-OLAP system very challenging. The main reason is that many existing OLAP optimization techniques (e.g., full cube materialization) are no longer applicable nor useful in implementing an S-OLAP system. We describe the details and our solutions in the next section.

4 Implementation

In the last section we presented the concept of S-OLAP and now we present the technical details of its implementation.

4.1 S-OLAP System

In order to implement an S-OLAP system, the first technical question we have to solve is: "(a) *how to efficiently compute an S-cuboid?*" In traditional OLAP, many researchers have proposed the use of various auxiliary data structures (e.g., bitmap index [15] and join index [21]) to speed up the cuboid construction process. We have to answer the same question for our S-OLAP prototype system.

The second technical challenge is: "*(b) how to support the proposed S-OLAP operations such that a sequence of S-OLAP queries can be efficiently evaluated?*" In traditional OLAP, cube materialization [9] is a popular approach in which some cuboids are computed in advance such that they can be used to answer various OLAP queries efficiently. The approach of *full materialization* refers to the precomputation of all cuboids (i.e., the full cube) and the approach of *partial materialization* refers to the precomputation of a subset of cuboids (i.e., the subcube). Since summarizability generally holds in traditional data cube, partial materialization is useful because (i) if a query result (a cuboid) has already been materialized, the answers can be returned right away, and (ii) even if a query result has not been materialized, a coarser aggregate can still be computed solely from the corresponding finer aggregates by exploiting appropriate materialized cuboids without accessing the base data. Consequently, iterative queries can be answered efficiently.

In S-OLAP, full materialization is not practical because the number of pattern dimensions is unbounded. Meanwhile, the non-sum-marizability of S-cubes invalidates the power of partial materialization because an S-cuboid cannot be computed from other S-cuboids via simple aggregations. As a result, instead of precomputating S-cuboids, our approach is to precompute some other auxiliary data structures so that queries can be computed online using the pre-built data structures.

Figure 6 shows the architecture of our prototype S-OLAP system. Events are stored as tuples in relational databases or as events in native sequence databases. Similar to traditional OLAP systems, a user can pose their S-OLAP queries through a **User Interface**. The User Interface provides certain user-friendly components to help a user

specify an S-cuboid (e.g., offering some drag-and-drop facilities). Furthermore, a user can perform the six S-OLAP operations through the interface. Given an S-cuboid query, the **S-OLAP Engine** searches a **Cuboid Repository** to see if such an S-cuboid has been previously computed and stored. If not, the S-OLAP engine either computes the S-cuboid from scratch or computes the S-cuboid with the help of certain **Auxiliary Data Structures**. The computed S-cuboid is then added to the Cuboid Repository. (If storage space is limited, the Cuboid Repository could be implemented as a cache with an appropriate replacement policy such as LRU (least-recently-used).)

During the computation of an S-cuboid, the S-OLAP System starts with the first four steps of S-cuboid formation as illustrated in Section 3.2, i.e., (1) Selection, (2) Clustering, (3) Sequence Formation and (4) Sequence Grouping. These four steps can be off-loaded to an existing sequence database query engine and the constructed sequence groups can be cached in a **Sequence Cache** for efficiency. After the first four steps, the sequence groups are stored in a q-dimensional array (where q is the number of global dimensions). Once the sequence groups are formed (or loaded from the sequence cache), the S-OLAP Engine starts the S-cuboid construction. We have investigated two simple approaches for this S-cuboid construction step. The first one is a counter-based method and the second one uses inverted indices as the auxiliary data structure. In the following discussion, we assume that the left-maximality-matched-go cell restriction is used.

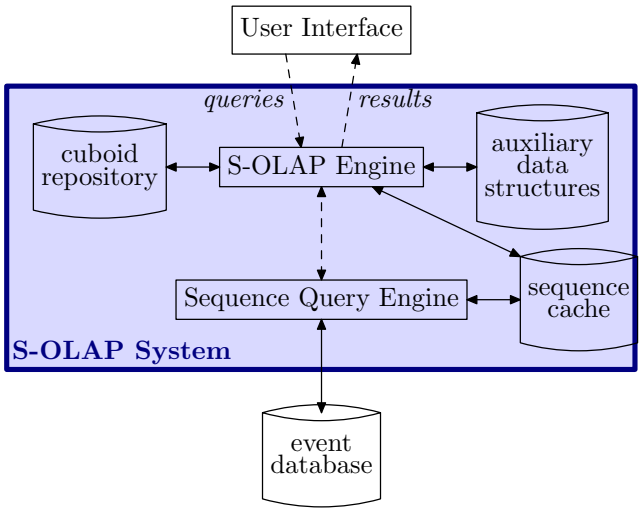

Fig. 6. Architecture of S-OLAP System

4.2 S-cuboid Construction

We present approaches to address the two technical challenges that we raised previously i.e., (a) efficient S-cuboid computation and (b) efficient processing of S-OLAP operations.

The first approach is a counter-based method (CB), in which each cell in an S-cuboid is associated with a counter. To determine the counters' values, the set of sequences in each sequence group is scanned. For each sequence s, we determine the cells whose associated patterns are contained in s. We increment each of such counters by 1. The CB approach addresses challenge (a). For challenge (b), CB takes the result of applying each S-OLAP operation as a specification of a new S-cuboid and computes the S-cuboid from scratch.

The second approach is based on inverted indices (II), in which a set of inverted indices are created by pre-processing the data offline. During query processing, the relevant inverted indices are joined online so as to address challenge (a). The by-products of answering a query is the creation of new inverted indices. As we will discussed shortly, such indices can assist the processing of a follow-up query. The inverted list approach thus potentially addresses challenge (b) as well.

Before we delve into the details, we remark that the two approaches we present here are only two "first-attempt" solutions to the Sequence OLAP problem and we believe that there are a lot of potentials for further optimization. For example, we can study the problem of computing iceberg [4] S-cuboids, or online aggregation [10] of S-cuboids, etc. All these ideas are interesting research topics and we discuss these issues in more detail in Section 6.

Counter-Based Approach In the counter-based method, we maintain a counter for each cell in an S-cuboid. All relevant counters are looked-up and incremented when the data sequences are scanned. If the number of counters is small enough to fit in memory, it is an efficient single-pass algorithm.

For each sequence group that is obtained from the first four S-cuboid formation steps, we invoke the procedure COUNTERBASED in Figure 7 with all sequences in the group and the CUBOID BY specification as input. The procedure performs the pattern grouping step and the aggregation step and returns an n-dimensional array (where n is the number of pattern dimensions). An entry $C[v_1, \ldots, v_n]$ in the n-dimensional array C stores the number of sequences that match the substring pattern $(v_1, \ldots, v_n)$. The procedure repeats for each sequence group and finally a $(q + n)$-dimensional S-cubiod is obtained.

Note that the COUNTERBASED procedure in Figure 7 is for substring pattern matching only. Subsequence pattern can be easily supported by modifying Line 7 in Figure 7.

Although this counter-based method is simple, its performance may degrade when the number of counters far exceeds the amount of available memory because counters are paged in for each sequence in the scan. Furthermore, this algorithm does not facilitate the processing of iterative S-OLAP queries as it computes an S-cuboid from scratch every time we apply an S-OLAP operation to transform an S-cuboid.

Algorithm COUNTERBASED
Input: (a) A set of sequences S from a sequence group; (b) A pattern template T=SUBSTRING$(Y_1, \ldots, Y_m)$ with m pattern symbols and n pattern dimensions $P_1, \ldots, P_n$ ($n \leq m$); (c) a cell restriction σ; and (d) a matching predicate ρ.
Output: An array C of n dimensions
1. Let $dom(P_i)$ be the domain of pattern dimension P_i at the specified abstraction level
2. /** Initialize the counters **/
3. **for** each pattern $(v_1, \ldots, v_n)$, where $v_i \in dom(P_i)$
4. Set entry $C[v_1, v_2, \ldots, v_n] = 0$
5. /** Do the grouping and counting **/
6. **for** each sequence s in S
7. **for** each unique substring t of s in the form of $\langle y_1, \ldots, y_m \rangle$, where each $y_i \in dom(P_i)$, t matches an instantiation of T and t satisfies ρ and σ
8. $C[y_1, \ldots, y_m]$++
9. **return** C

Fig. 7. Procedure COUNTERBASED

sid	*card-id*	*event-sequence* (only the station values are shown for brevity)
$s1$	688	⟨Glenmont,Pentagon,Pentagon,Wheaton,Wheaton,Pentagon⟩
$s2$	23456	⟨Pentagon,Wheaton,Wheaton,Pentagon⟩
$s3$	1012	⟨Clarendon,Pentagon⟩
$s4$	77	⟨Wheaton,Clarendon,Deanwood,Wheaton⟩

N.B. Events at odd positions have action "in"
whereas events at even positions have action "out"

Fig. 8. An example sequence group in Query Q_1 (day="2007-12-25", fare-group="regular")

Inverted Index Approach The inverted index approach follows a semi-online computation strategy. It involves two basic algorithms: one for computing inverted indices and one for constructing S-cuboids based on the inverted indices. The basic idea is similar to the idea of shell fragment cubes in [13], in which we partition the pattern dimensions into a set of low dimensional *pattern fragments*, and each fragment is represented by an inverted index. Using the precomputed inverted indices, we can dynamically assemble and compute S-cuboid cells of the required S-cuboid online.

The inverted index approach shares the same first four steps of S-cuboid formation as in the counter-based approach. Therefore, after the first four steps, a number of sequence groups are formed. To illustrate the inverted index approach, we consider substring patternsand the sequence group shown in Figure 8. We assume each sequence is identified by a unique *sid* attribute.

To precompute inverted indices, we have developed a construction algorithm, BUILDINDEX. It creates a size-m inverted index L_m, where m is a user-specified parameter. L_m is a set of inverted lists. An inverted list, denoted by $L_m[v_1, \ldots, v_m]$, is associated with a length-m substring pattern $(v_1, \ldots, v_m)$. Each element in the pattern is chosen from the domain of a pattern dimension at a particular abstraction level. The list stores the *sid*s of all sequences that match the substring patterns associated with it. For example, considering the *location* pattern dimension at the station abstraction level, two inverted indices L_1 and L_2 constructed for our data sequence group are shown in Figure 10 (empty lists, such as L_2[Clarendon,Clarendon], are not shown). For notational convenience, given a pattern template T, we use L_m^T to denote a subset of L_m such that an inverted list $L_m[v_1, \ldots, v_m]$ is in L_m^T if the pattern $(v_1, \ldots, v_m)$ is an *instantiation* of the template T. For example, considering the lists in Figure 10, we have $L_2^{(X,X)} = \{l_5, l_9\}$[6]. Also $L_2^{(X,Y)}$ includes all the lists in L_2 if there are no restrictions on X and Y. The algorithm, BUILDINDEX, is summarized in Figure 9.

Algorithm BUILDINDEX
Input: (a) A set of sequences S from a sequence group; (b) A pattern template T=SUBSTRING$(Y_1, \ldots, Y_m)$ with m pattern symbols and n pattern dimensions $P_1, \ldots, P_n$ ($n \leq m$);
Output: An array L_m, which is an m-dimensional array and each array entry contains a list of sequence *sid*s.
1. Let $dom(P_i)$ be the domain of pattern dimension P_i at the specified abstraction level
2. /* Scan the sequence group S */
3. **for** each sequence s in S
4. **for** each unique substring t of s in the form of $\langle y_1, \ldots, y_m \rangle$, where each $y_j \in dom(P_j)$ and t matches an instantiation of T
5. add *sid* of s into $L_m[y_1, \ldots, y_m]$
6. **return** L_m^T

Fig. 9. Procedure BUILDINDEX

[6] Technically speaking, L_2[Clarendon,Clarendon] is also in $L_2^{(X,X)}$. Since the list L_2[Clarendon,Clarendon] is empty, we omit it in our discussion.

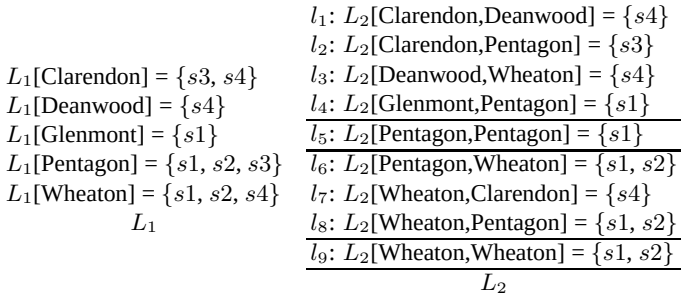

Fig. 10. Inverted indices of a sequence group

Given a set of precomputed inverted indices, computing an S-cuboid becomes fairly simple. Consider a query Q_3 that inquires the statistics of single-trip passengers. The cuboid specification of Q_3 is shown in Figure 11 (only the CUBOID BY clause is shown). Q_3, which specifies a pattern template (X, Y), can be answered by $L_2^{(X,Y)}$ (which is the same as L_2 since X, Y are unrestricted). For each instantiation (v_1, v_2) of (X, Y), the count of the S-cuboid cell of pattern (v_1, v_2) can be computed by simply retrieving the inverted list $L_2[v_1, v_2]$, and counting the number of sequences in the list that satisfy the cell restriction and predicate (i.e., Lines 13-15 in Figure 11). Figure 12 shows the non-zero entries of the 2D S-cuboid computed.

S-cuboids of higher dimension can also be computed by *joining* inverted indices. For example, consider query Q_1, which specifies a pattern template (X, Y, Y, X). We answer Q_1 in two steps, assuming that L_2 is materialized. We first compute $L_3^{(X,Y,Y)}$ (i.e., the set of inverted lists for any length-3 patterns that are instantiations of (X, Y, Y)). This can be done by joining $L_2^{(X,Y)}$ with $L_2^{(Y,Y)}$. The semantics of $R = L_2^{(X,Y)} \bowtie L_2^{(Y,Y)}$ is that a list $l \in R$ iff $l = L_2[v_1, v_2] \cap L_2[v_3, v_3]$ such that $L_2[v_1, v_2] \in L_2^{(X,Y)}$, $L_2[v_3, v_3] \in L_2^{(Y,Y)}$ and $v_2 = v_3$. Using our running example, $L_2^{(X,Y)} = L_2$ and $L_2^{(Y,Y)} = \{l_5, l_9\}$. The list intersections performed by the join is illustrated in Figure 13. Sequences in the lists in R are then checked by scanning the database to eliminate invalid entries. For example, refer to Figure 13, list l_{12} is obtained by $l_5 \cap l_5 = \{s1\}$. Since $s1$ does not contain the substring pattern (Pentagon,Pentagon,Pentagon), $s1$ is removed from the list. The resulting index gives $L_3^{(X,Y,Y)}$. The index $L_4^{(X,Y,Y,X)}$ can be obtained by joining $L_3^{(X,Y,Y)}$ with $L_2^{(Y,X)}$ in a similar fashion. Figure 14 shows the only non-empty list resulted. Finally, the count of an S-cuboid cell can be computed by retrieving the corresponding list in $L_4^{(X,Y,Y,X)}$, verifying the sequences against cell restrictions and predicates, and counting the valid ones. In our example, only one cell [Pentagon,Wheaton,Wheaton,Pentagon] has a count of 1, all others are 0.

The query processing algorithm QUERYINDICES is summarized in Figure 15. For all S-OLAP queries, we can invoke QUERYINDICES to compute an S-cuboid from scratch. During query evaluation, if QUERYINDICES requires an inverted index that is not available, then QUERYINDICES would build the proper inverted index at run-time. This on-demand building process would increase the initial query time. However, the

10. CUBOID BY SUBSTRING (X, Y) WITH
11. X AS *location* AT station,
12. Y AS *location* AT station
13. LEFT-MAXIMALITY ($x1$, $y1$) WITH
14. $x1$.*action* = "in" AND
15. $y1$.*action* = "out"

Fig. 11. Query specification Q_3

(*station*,*station*)	*count*
(Clarendon,Pentagon)	1
(Deanwood,Wheaton)	1
(Glenmont,Pentagon)	1
(Pentagon,Wheaton)	2
(Wheaton,Clarendon)	1
(Wheaton,Pentagon)	2

Fig. 12. A 2D S-cuboid for query Q_3

subsequent iterative queries, which are obtained by successive applications of S-OLAP operations and highly correlated to the previous queries, would be benefited from the newly computed inverted indices. We now discuss how the six S-OLAP operations could make use of existing inverted indices to obtain better performance. Recall that, for a sequence of iterative queries, Q_a, Q_b, Q_c, if a query has been evaluated before and its result is cached, the evaluation can be skipped and the cached result can be returned right away. For example, if we perform an APPEND on Q_a to obtain Q_b, followed by a DE-TAIL to obtain Q_c, then Q_c is the same as Q_a and the cached result can be returned.

1. [APPEND**]** We explain the implementation of the APPEND operation by the following iterative queries Q_a, Q_b, Q_c. We use Q_3 (shown in Figure 11) as Q_a. The second query Q_b is obtained by APPENDing a symbol Y to Q_a and therefore its pattern template is (X, Y, Y).[7] The final query Q_c is obtained by APPENDing one more symbol X to Q_b. The first query Q_a can be directly evaluated by QUERYINDICES. That is, the inverted index $L_2^{(X,Y)}$ in Figure 10 is scanned and the number of sequences that satisfy the cell restriction and matching predicate in each list is counted. The result of Q_a is shown in Figure 12.

The implementation of an APPEND operation is very similar to QUERYINDICES. In our example, the first APPEND operation (i.e., the evaluation of Q_b) is implemented by first performing $L_2^{(X,Y)} \bowtie L_2^{(Y,Y)}$ to obtain $L_3^{(X,Y,Y)}$ and then counting the number of sequences in $L_3^{(X,Y,Y)}$ (Figure 13) that satisfy the cell restriction and the matching predicate. Similarly, the last APPEND operation (i.e., the evaluation of Q_c) is implemented by first joining $L_3^{(X,Y,Y)}$ with $L_2^{(Y,X)}$ to obtain $L_4^{(X,Y,Y,X)}$, and then counting the number of sequences in $L_4^{(X,Y,Y,X)}$ (Figure 14) that satisfy the cell restriction and

[7] For brevity, we only focus on the changes of the pattern template and do not discuss the changes of other constructs such as the matching predicate here.

the matching predicate. Note that the last APPEND operation does not build the inverted index $L_4^{(X,Y,Y,X)}$ from scratch.

	list-intersection	sid-intersection	$\{sid\}$
l_{10}: $L_3^{(X,Y,Y)}$[Clarendon,Pentagon,Pentagon]	$l_2 \cap l_5$	$\{s3\} \cap \{s1\}$	$\{\}$
l_{11}: $L_3^{(X,Y,Y)}$[Glenmont,Pentagon,Pentagon]	$l_4 \cap l_5$	$\{s1\} \cap \{s1\}$	$\{s1\}$
l_{12}: $L_3^{(X,Y,Y)}$[Pentagon,Pentagon,Pentagon]	$l_5 \cap l_5$	$\{s1\} \cap \{s1\}$	~~$\{s1\}$~~
l_{13}: $L_3^{(X,Y,Y)}$[Wheaton,Pentagon,Pentagon]	$l_8 \cap l_5$	$\{s1, s2\} \cap \{s1\}$	~~$\{s1\}$~~
l_{14}: $L_3^{(X,Y,Y)}$[Deanwood,Wheaton,Wheaton]	$l_3 \cap l_9$	$\{s4\} \cap \{s1, s2\}$	$\{\}$
l_{15}: $L_3^{(X,Y,Y)}$[Pentagon,Wheaton,Wheaton]	$l_6 \cap l_9$	$\{s1, s2\} \cap \{s1, s2\}$	$\{s1, s2\}$

Fig. 13. $L_3^{(X,Y,Y)}$

	list-intersection	sid-intersection	$\{sid\}$
l_{16}: $L_4^{(X,Y,Y,X)}$[Pentagon,Wheaton,Wheaton,Pentagon]	$l_{15} \cap l_8$	$\{s1, s2\} \cap \{s1, s2\}$	$\{s1, s2\}$

Fig. 14. $L_4^{(X,Y,Y,X)}$

2. [PREPEND**]** The PREPEND operation is very similar to the APPEND operation. Continue with the above iterative queries example. Assume that we further PREPEND a symbol Z to Q_c to obtain a new query Q_d and the resulting pattern template is (Z, X, Y, Y, X). Similar to the APPEND operation, this PREPEND operation is implemented by joining $L_2^{(Z,X)}$ with $L_4^{(X,Y,Y,X)}$ to obtain $L_5^{(Z,X,Y,Y,X)}$. Note that with $L_4^{(X,Y,Y,X)}$ computed, the domain (i.e., the set of all possible instantiations) of X is known. Therefore, $L_2^{(Z,X)}$ does not contain all lists in L_2, as X is restricted.

3. [DE-HEAD **and** DE-TAIL**]** The DE-HEAD and the DE-TAIL operations rely more on the caching feature of the S-OLAP system. Continue with the above iterative queries example. If we apply a DE-HEAD operation after the evaluation of Q_d, we essentially restore the query back to Q_c. Therefore, the system can return the cached S-cuboid of Q_c as the answer. However, another DE-HEAD operation results in a new query Q_e with pattern template (Y, Y, X). Since we have not built the inverted index $L_3^{(Y,Y,X)}$ during the process (see the table on the next page), Q_e is evaluated from scratch, by invoking QUERYINDICES directly.

Query	Pattern Template
Q_a (=Q_3)	(X, Y)
Q_b	(X, Y, Y)
Q_c	(X, Y, Y, X)
Q_d	(Z, X, Y, Y, X)
Q_e	(Y, Y, X)

The DE-TAIL operation is similar to the DE-HEAD operation. If there are proper inverted indices available or the query has been evaluated before, the DE-TAIL operation could be processed by retrieving a cached result. Otherwise, we invoke QUERYINDICES.

Algorithm QUERYINDICES
Input: (a) A set of sequences S from a sequence group; (b) A pattern template T=SUBSTRING$(Y_1, \ldots, Y_m)$ with m pattern symbols and n pattern dimensions $P_1, \ldots, P_n$ ($n \leq m$); (c) a cell restriction σ; and (d) a matching predicate ρ.
Output: An array C of n dimensions
1. Let $dom(P_i)$ be the domain of pattern dimension P_i at the specified abstraction level
2. /** Initialize the counters **/
3. **for** each pattern $(v_1, \ldots, v_n)$, where $v_i \in dom(P_i)$
4. Set entry $C[v_1, v_2, \ldots, v_n] = 0$
5. /** Look-up inverted index $L_m^{(Y_1,\ldots,Y_m)}$ and join the inverted indices if necessary **/
6. **while** $L_m^{(Y_1,\ldots,Y_m)}$ is not available
7. /** Join the indices according to the pattern template and intersect the sequence lists **/
8. $L_{i+1}^{(Y_1,\ldots,Y_{i+1})} = L_i^{(Y_1,\ldots,Y_i)} \bowtie L_2^{(Y_i,Y_{i+1})}$ (where $L_i^{(Y_1,\ldots,Y_i)}$ is the largest available inverted index)
9. Scan the database to eliminate invalid entries and cache $L_{i+1}^{(Y_1,\ldots,Y_{i+1})}$
10. **for** each entry $L_m^{(Y_1,\ldots,Y_m)}[v_1, \ldots, v_m]$ in $L_m^{(Y_1,\ldots,Y_m)}$
11. $C[v_1, \ldots, v_n]$ equals to the number of sequences in $L_m^{(Y_1,\ldots,Y_m)}[v_1, \ldots, v_m]$ that satisfy σ and ρ.
12. **return** C

Fig. 15. Procedure QUERYINDICES

4. [P-ROLL-UP**]** The P-ROLL-UP operation can be efficiently implemented if there are proper inverted indices available. Assume we apply a P-ROLL-UP operation on Q_a such that the pattern dimension Y on the *location* attribute of the new query Q_A is rolled-up from the station abstraction level to the district abstraction level. This P-ROLL-UP operation can be efficiently implemented by taking the unions of the lists in $L_2^{(X,Y)}$ whose second elements in their patterns share the same *district* value. We denote the resulting inverted index $L_2^{(X,\mathbb{Y})}$. (Here, we use different fonts to indicate different abstraction levels, e.g., X for the station abstraction level and $\mathbb{X}$ for the district abstraction level.) For example, assume that district D10 includes two stations Pentagon and Clarendon, then the lists $L_2^{(X,Y)}$[Wheaton,Clarendon] and $L_2^{(X,Y)}$[Wheaton,Pentagon] (see l_7 and l_8 in Figure 10) are unioned to obtain $L_2^{(X,\mathbb{Y})}$[Wheaton,D10]. The result of applying a P-ROLL-UP can then be obtained by counting the number sequences in $L_2^{(X,\mathbb{Y})}$ that satisfy the cell restriction and matching predicate. For instance, the cell [Wheaton,D10] in the resulting S-cuboid has a count of three.

In the above example, symbols in the pattern template (X, Y) are unrestricted. We remark that if symbols are restricted then a P-ROLL-UP may not be processed by simply merging lists. To understand why it is so, let us consider a sequence $s6$: ⟨Pentagon, Wheaton, Wheaton, Clarendon⟩. Clearly, s_6 does not occur in any list of $L_4^{(X,Y,Y,X)}$. However, district D10 includes both Pentagon and Clarendon and so s_6 should be in $M = L_4^{(\mathbb{X},Y,Y,\mathbb{X})}$[D10, Wheaton, Wheaton, D10]. Hence, if we compute M by merging lists in $L_4^{(X,Y,Y,X)}$, s_6 will be missed incorrectly. This example shows that if the pattern template consists of restricted symbols, P-ROLL-UP cannot be implemented by merging

inverted lists at a lower abstraction level. In this case, we compute the result by invoking QUERYINDICES.

5. [P-DRILL-DOWN**]** Consider applying P-DRILL-DOWN on Q_A (i.e., the pattern dimension Y of Q_3 has been rolled-up). If the inverted index $L_2^{(X,Y)}$ for Q_a is available, the cached result can be returned. Otherwise, P-DRILL-DOWN is processed either by invoking QUERYINDICES or by constructing the inverted index $L_2^{(X,Y)}$ from $L_2^{(X,\mathbb{Y})}$. For the latter case, each list $L_2[v_1,\mathrm{v}_2]$ in $L_2^{(X,\mathbb{Y})}$ is *refined* into a number of lists $L_2[v_1, v_2]$ where v_2 is a lower-level concept of v_2. Data sequences are examined to determine the refinement. For example, $L_2^{(X,\mathbb{Y})}$[Wheaton,D10] = $\{s1, s2, s4\}$. It is refined to L_2[Wheaton,Pentagon] = $\{s1, s2\}$ and L_2[Wheaton,Clarendon] = $\{s4\}$. □

The counter-based approach (CB) constructs an S-cuboid by scanning data sequences to determine which cells each sequence is relevant to. All sequences are thus examined in answering a S-OLAP query. On the other hand, the inverted list approach (II) constructs inverted lists and accesses data sequences that are contained in certain lists. In terms of performance, II has the advantage of fewer data sequence accesses if queries are very selective (e.g., point queries or subcube queries), where appropriate lists have already been constructed. This can be seen from our example iterative queries. On the other hand, the construction of inverted indices can be costly. This affects the performance of II, particularly in the start-up cost of iterative queries.

The inverted index approach is not a Swiss army knife for implementing all S-OLAP operations. For example, it cannot efficiently support P-ROLL-UP if the pattern template contains restricted symbols. In these cases, CB could be a competitive option. In fact, this is a sophisticated S-OLAP query optimization problem where many factors such as storage space, memory availability, and execution speed are parts of the formula. Another interesting question concerns "which" inverted indices should be materialized offline. A related problem is thus about how to determine the lists to be built given a set of *frequently asked queries*. All these problems are related to the design of an S-OLAP query optimizer and we regard this as one of our most important future work.

5 Experimental Evaluation

This section shows the results of the experiments we conducted on our prototype S-OLAP system. The prototype was implemented using C++ and all the experiments were conducted on an Intel Pentium-4 2.6GHz PC with 2GB of RAM. The system ran Linux with the 2.6.10 kernel and gcc 3.3.3.

We have performed experiments on both real data and synthetic data. The experiments on real data (Section 5.1) show a use case of performing click stream data analysis using our S-OLAP system. The experiments on synthetic data (Section 5.2) study the performance of our S-OLAP prototype system and evaluate the counter-based and the inverted index approaches.

5.1 Experiments on Real Data

The real sequence data is a clickstream and purchase dataset from Gazelle.com, a legwear and legcare web retailer, who closed their online store on 2000-08-18. It was

prepared by [11] for KDD Cup 2000. The original data file size is 238.9MB. Each tuple in the data file is a visitor click event (sorted by user sessions) and there is a total of 164,364 click events. The details of an event are captured by 215 attributes. Three example attributes are *session-id*, *request-time* and *page* which identify a user session, its first access time, and the accessed page.[8]

To demonstrate the usability of an S-OLAP system and to validate our S-OLAP design, we use our S-OLAP prototype system to answer a KDD Cup 2000 data mining query in an OLAP data exploratory way. The selected query is KDD Cup 2000 Query 1, which looks for page-click patterns of visitors. Since the data was not designed for OLAP analysis, we have performed the following pre-processing steps: (1) We manually inspected the data and filtered out click sequences that were generated from web crawlers (i.e., user sessions with thousands of clicks). After this step, an event database with 148,924 click events was obtained. (2) We manually associated a concept hierarchy raw-page $\rightarrow$ page-category to the *page* attribute such that a page can be categorized by two abstraction levels. page-category is a higher abstraction level and there are 44 categories. Example categories include "Assortment", "Legwear", "Legcare", "Main Pages", etc.

To answer the KDD Cup query, we started with a general S-OLAP query Q_a to look for information about any two-step page accesses at the page-category abstraction level:

1. SELECT COUNT(*) FROM Event
2. CLUSTER BY *session-id*
3. SEQUENCE BY *request-time* ASCENDING
4. CUBOID BY SUBSTRING(X,Y) WITH
5. X AS page AT page-category,
6. Y AS page AT page-category
7. LEFT-MAXIMALITY($x1$,$y1$)

There were 50,524 sequences constructed and they were in a single sequence group. Query Q_a returned a 44$\times$44 2D S-cuboid. From the result, we found out that the cell (Assortment,Legcare) had a count of 150, meaning that there were 150 sessions first visited an Assortment-related page followed by a Legcare-related page. Interestingly, we found that the cell (Assortment,Legwear) had a much larger count of 2,201 sequences (the highest count in the S-cuboid), meaning that there were many sessions first visited an Assortment-related page followed by a Legware-related page. Consequently, we performed a *slice* operation on that cell (i.e., Assortment $\rightarrow$ Legwear) and performed a P-DRILL-DOWN operation to see what Legwear products the visitors actually wanted to browse. This results in a new query Q_b (the cuboid specification is omitted due to lack of space).

Query Q_b returned a 1$\times$279 2D S-cuboid. The cell with the highest count was (Assortment,product-id-null) which had a count of 181, meaning that many sessions visited a product page where the product has no product-id after clicking an Assortment-related page. Another remarkable cell was (Assortment,product-id-34893) which had a count of 172 (the second highest count), meaning that there were many sessions first visited an Assortment-related page followed by a DKNY Skin collection legwear page

[8] The attribute names are renamed here for better exposition.

(product-id=34893). After viewing the result of Q_b, we performed an APPEND operation to see if those sessions who visited an Assortment-related page followed by a Legware-related page would visit one more Legware-related page to perform so-called "comparison shopping". That APPEND operation resulted in a new query Q_c.

Query Q_c returned a $1\times279\times279$ 3D S-cuboid. A remarkable cell was (Assortment,product-id-34885,product-id-34897) which had a count of 14, meaning that there were 14 sessions visited an Assortment-related page, then a DKNY Skin collection legwear page (product-id=34885), and then a DKNY Tanga collection legware page (product-id=34897). At that point, we stopped our S-OLAP exploration because we have collected enough information to answer Query 1 in KDD Cup 2000 indirectly. Altogether, the three queries had inserted 0.3MB of cuboids in the cuboid repository.

In the following we report the performances of iterative queries Q_a, Q_b, and Q_c using both the counter-based approach (CB) and the inverted index approach (II). We repeated each query many times in order that the 90% confidence intervals of the reported numbers are within $\pm$5%. Note that in this experiment we did not precompute any inverted index in advance. Table 1 shows the result.

	Counter-Based (CB)		Inverted Index (II)		
Query	Runtime (ms)	Number of sequences scanned	Runtime (ms)	Number of sequences scanned	Size of II (MB)
Q_a	24.3	50,524	46.24	50,524	0.897
Q_b	21.5	50,524	6.26	2,201	0.104
Q_c	23.0	50,524	5.92	842	0
$\sum$	68.8	151,572	58.42	53,567	1.001

Table 1. Real Data Experiment

Table 1 shows that for the first query Q_a, CB had a better performance than II. This is not surprising because we did not precompute any inverted index in advance so that the query processing time of Q_a included the time for building 0.897MB inverted indices. However, for Q_b and Q_c, II outperformed CB because II did not need to scan all sequences with the help of the inverted indices. Table 1 also shows the advantage of using inverted indices to perform S-OLAP operations. From Q_a to Q_b, we had performed a *slice* and a P-DRILL-DOWN operation. After the *slice* operation, the number of sequences related to Q_b was reduced. As a result, the II implementation of the P-DRILL-DOWN operation outperformed the CB implementation because Q_b became more selective. From Q_b to Q_c, we had performed an APPEND operation. Table 1 shows that the II implementation of the APPEND operation also outperformed the CB implementation because II reused the inverted indices to scan fewer sequences than CB.

5.2 Experiments on Synthetic Data

Synthetic sequence databases are synthesized in the following manner. The generator takes 4 parameters: L, I, θ, and D. The generated sequence database has D sequences. Each sequence s in a dataset is generated independently. Its length l, with mean L, is

first determined by a random variable following a Poisson distribution. Then, we repeatedly add events to the sequence until the target length l is reached. The first event symbol is randomly selected according to a pre-determined distribution following Zipf's law with parameter I and θ (I is the number of possible symbols and θ is the skew factor). Subsequent events are generated one after the other using a Markov chain of degree 1. The conditional probabilities are pre-determined and are skewed according to Zipf's law. All the generated sequences form a single sequence group and that is served as the input data to the algorithms.

QuerySet A – (a) Varying D. The objective of this experiment is to study the scalability of the counter-based approach and the inverted index approach under a series of APPEND operations. In this experiment, we executed a set of iterative queries under different numbers of sequences. The query set, namely Q_A, consists of five S-OLAP queries Q_{A1}, Q_{A2}, Q_{A3}, Q_{A4} and Q_{A5}. A query is obtained from a previous one by doing a *slice* followed by an APPEND. The initial query Q_{A1} has a substring pattern template (X, Y) and it looks for size-two patterns in the sequence dataset and counts their occurrences. The second query Q_{A2} is obtained from Q_{A1} by performing a *slice* operation on the cell with the highest count and APPENDing a new pattern symbol Z to the pattern template of Q_{A1}. Therefore, Q_{A2} has a substring pattern template (X, Y, Z) and it looks for size-three patterns (with the first two symbols fixed) in the sequence dataset and counts their occurrences. Query Q_{A3}, Q_{A4} and Q_{A5} are obtained in a similar way and they are queries that look for size-four, size-five and size-six patterns in the sequence dataset, respectively.

Figure 16 shows the running time of query set Q_A under three datasets with different number of sequences ($I100.L20.\theta0.9.Dx$, where x=$100K/500K/1000K$). Three size-two inverted indices at the finest level of abstraction were precomputed for the three datasets. The precomputations took 0.43s, 2.052s and 3.879s, respectively. The sizes of the built indices were 7.3MB, 36.3MB and 72.2MB, respectively. The running time of Q_A is presented as the cumulative running time from the first query Q_{A1} to the last query Q_{A5}. From the figure, we can see that (1) both CB and II scaled linearly w.r.t. the number of sequences; and (2) II outperformed CB in all datasets in this experiment. Figure 17 shows the cumulative number of sequences scanned up to a certain query. We can see that CB scanned the whole dataset every time it executed. For Q_{A1}, II did not scan the dataset because it could be answered by the inverted indices directly. For the successive queries Q_{A2} to Q_{A5}, II took less than 1 second to finish inverted index joins in all cases because Q_{Ai+1} could exploit the inverted indices built by Q_{Ai} and thus not many data sequences were scanned.

QuerySet A – (b) Varying L. In this experiment, we executed query set Q_A on a dataset of 500K sequences and we varied the average length L of the sequences (i.e., $I100.Ly.\theta0.9.D500K$, where y=10/20/30). Figure 18 shows the cumulative running time and Figure 19 shows the cumulative number of sequences scanned, respectively. The following conclusions can be drawn from the results: (1) both CB and II scaled linearly w.r.t. the average sequence length and (2) II outperformed CB in all datasets in this experiment.

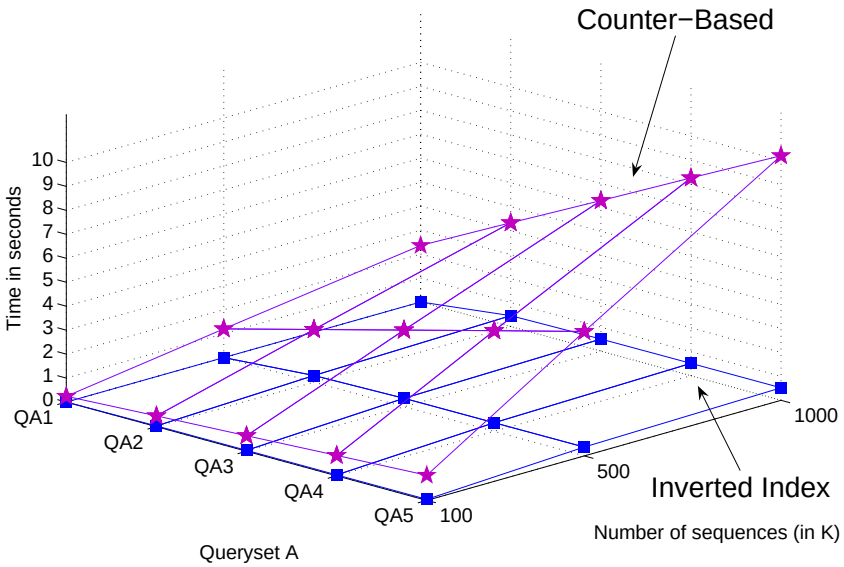

Fig. 16. Cumulative running time of $Q_A.I100.L20.\theta 0.9.Dx$

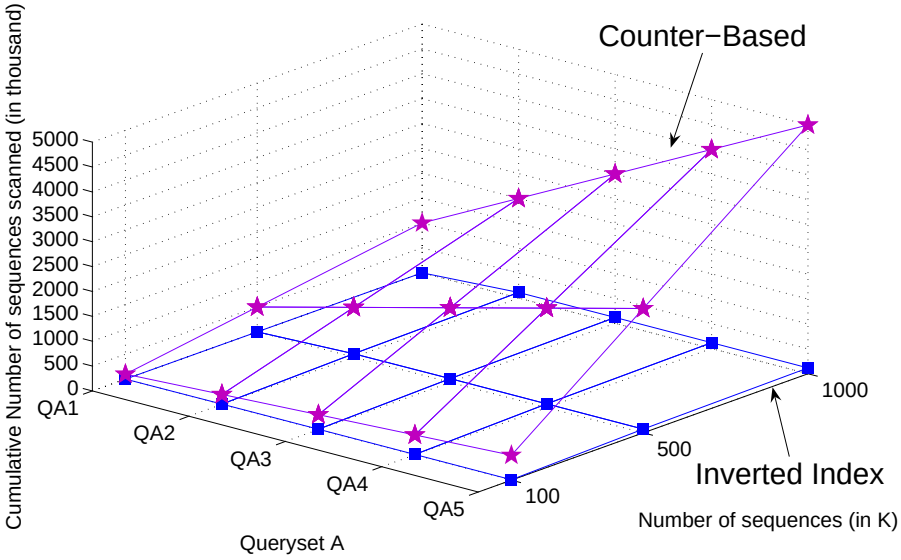

Fig. 17. Cumulative number of sequences scanned of $Q_A.I100.L20.\theta 0.9.Dx$

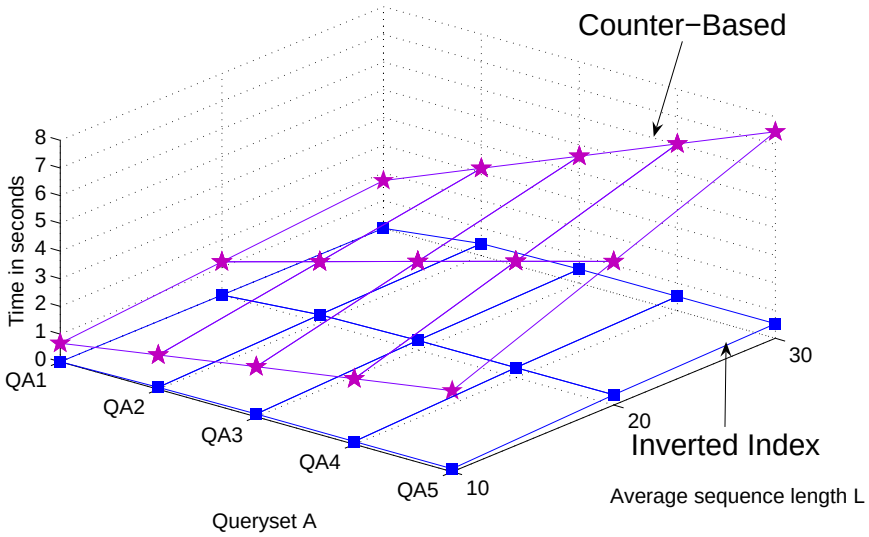

Fig. 18. Cumulative running time of $Q_A.I100.Ly.\theta 0.9.D500\text{K}$

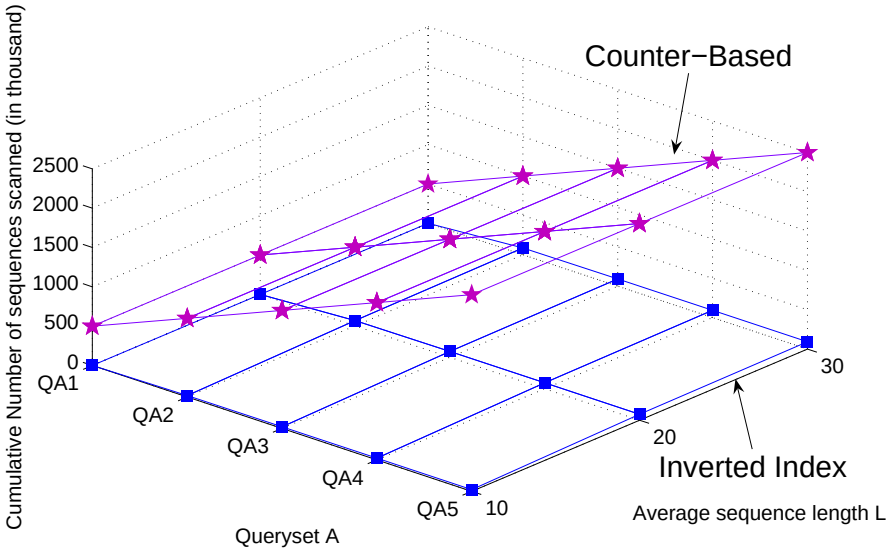

Fig. 19. Cumulative number of sequences scanned of $Q_A.I100.Ly.\theta 0.9.D500\text{K}$

QuerySet A – (b) Varying θ. In this experiment, we executed query set Q_A on a dataset of 500K sequences and we varied the skew factor θ of the sequences (i.e., $I100.L20.\theta z.D500$K, where z=0.7/0.8/0.9). Figure 20 shows the cumulative running time and Figure 21 shows the cumulative number of sequences scanned, respectively. We can see that CB is insensitive to the skew factor because CB processes each iterative query from scratch, the number of sequences scanned is therefore the same for each iterative query, and so as the running time. For II, we see that both the running time and the number of sequences scanned increase slightly with the increase of the skew factor. The reason is that a higher θ values imples a more skewed distribution of events in the dataset, leading to a skewed distiribution of the cuboid cells returned by each iterative query. Since we sliced on the cell with the highest count in each iteratvie query, the value in the sliced cell is therefore larger when θ is higher. As the number of sequences to be processed depends on the cardinality of the inverted indice in the sliced cell, a larger θ value thus increases the number of sequences to be processed and so as the running time of the iterative query.

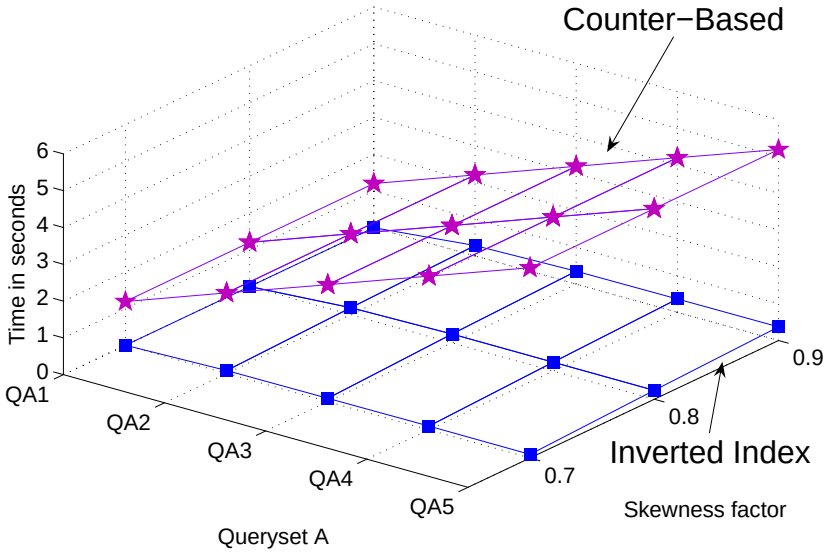

Fig. 20. Cumulative running time of $Q_A.I100.L20.\theta z.D500k$

QuerySet A – (b) Varying domain I. In this experiment, we executed query set Q_A on a dataset of 500K sequences and we varied the domain I in the dataset (i.e., $Id.L20.\theta 0.9.D500$K, where d=100/200/300). Figure 22 shows the cumulative running time and Figure 23 shows the cumulative number of sequences scanned, respectively. From the figures, we see that both CB and II are insensitive to the number of possible symbols in the dataset.
QuerySet B – (a) Varying D (b) Varying L. The objective of this experiment is to study the performance of CB and II under the P-ROLL-UP and P-DRILL-DOWN oper-

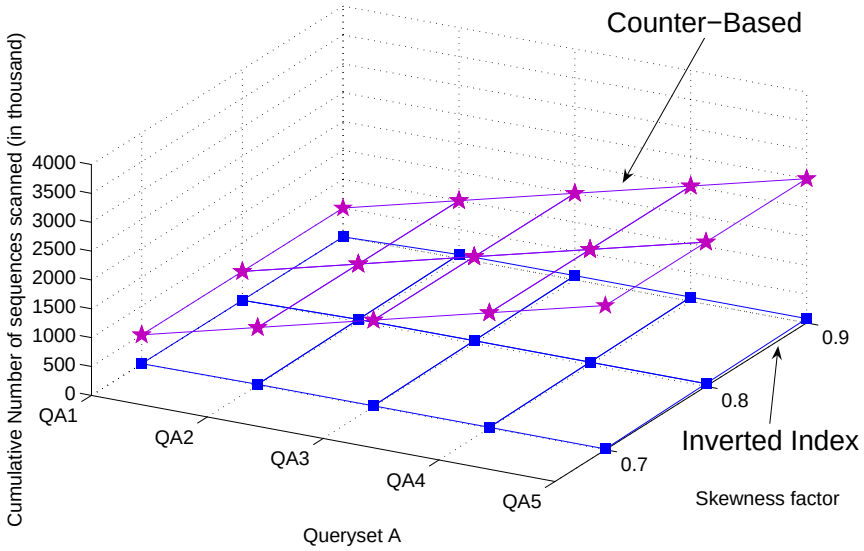

Fig. 21. Cumulative number of sequences scanned of $Q_A.I100.L20.\theta z.D500k$

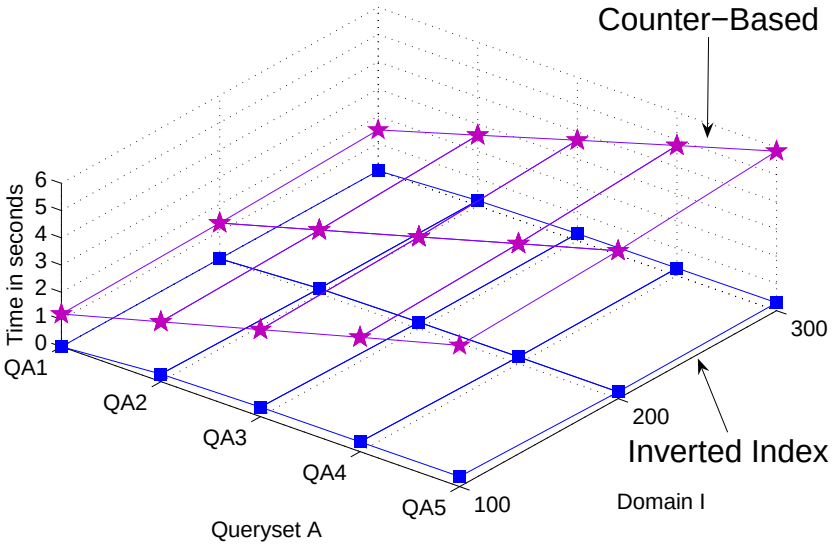

Fig. 22. Cumulative running time of $Q_A.Id.L20.\theta 0.9.D500k$

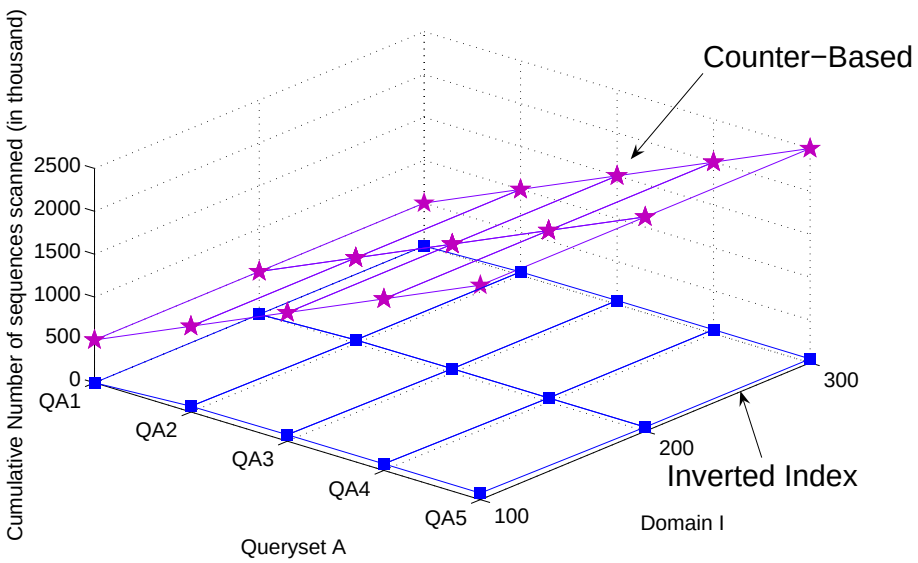

Fig. 23. Cumulative number of sequences scanned of $Q_A.Id.L20.\theta 0.9.D500k$

ations. In this experiment, the dataset was $I100.Lx.\theta 0.9.Dy$. We hierarchically organized the events into 3 concept levels. The 100 event symbols are divided into 20 groups, with group sizes following Zipf's law (I=20, θ=0.9). Similarly, the 20 groups are divided into 5 super-groups, with super-group sizes following Zipf's law (I=5, θ=0.9).

We used another query set Q_B in this experiment. Q_B consists of three queries Q_{B1}, Q_{B2}, and Q_{B3}. The first query Q_{B1} has a substring pattern templates of $(\mathbb{X}, \mathbb{Y}, \mathbb{Z})$ ($\mathbb{X}$ is the middle abstraction level). The second query Q_{B2} is obtained from Q_{B1} by performing a subcube operation to select the subcube with the same $\mathbb{X}$ value where its total count is the highest among different subcubes and then P-DRILL-DOWN into $\mathbb{X}$, i.e., the pattern template is $(X, \mathbb{Y}, \mathbb{Z})$ (X is the finest abstraction level). Similarly, the third query Q_{B3} is obtained from Q_{B1} by performing the same subcube operation and then P-ROLL-UP on $\mathbb{Y}$, i.e., the pattern template is $(\mathbb{X}, \mathcal{Y}, \mathbb{Z})$ (we did not P-ROLL-UP on $\mathbb{X}$ because it was sliced; $\mathcal{Y}$ is the highest abstraction level).

Similar to the experiments conducted in query set A (see above), we executed Q_B on datasets with different D (Figure 24, 25) and L (Figure 26, 27) values. In this experiment, an inverted index $L_3^{(\mathbb{X},\mathbb{Y},\mathbb{Z})}$ was precomputed in advance. The experimental results draw the following conclusions: (1) For P-DRILL-DOWN (i.e., Q_{B2}), CB and II had comparable performance because we sliced on the subcube with the highest count and the query was not selective. Therefore, II also needed to scan a lot of sequences in order to compute the inverted list $L^{(X,\mathbb{Y},\mathbb{Z})}$. (We found that if we sliced on cells with moderate counts then II outperformed CB. Figure 28 - Figure 31 show the experimental results.) (2) For P-ROLL-UP (i.e., Q_{B3}), II outperformed CB in all datasets because II

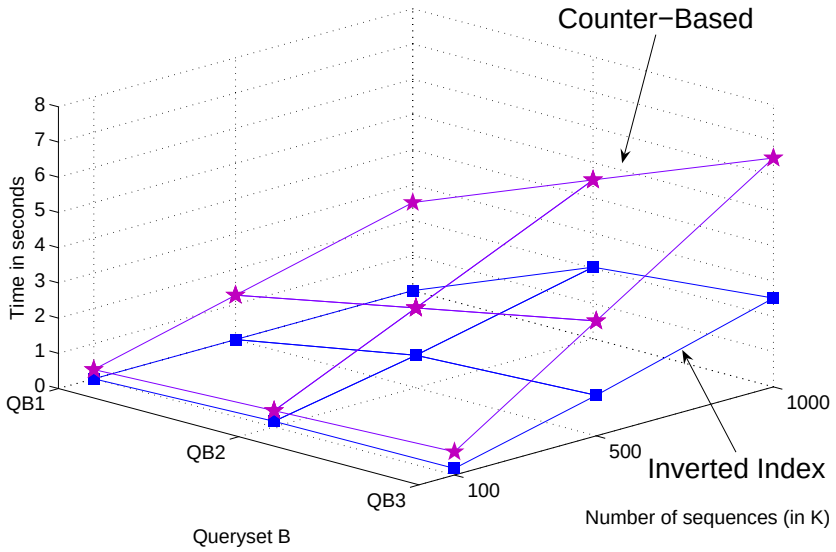

Fig. 24. Cumulative running time of $Q_B.I100.L20.\theta 0.9.Dy$, slice on the subcube with highest count in Q_{B2}.

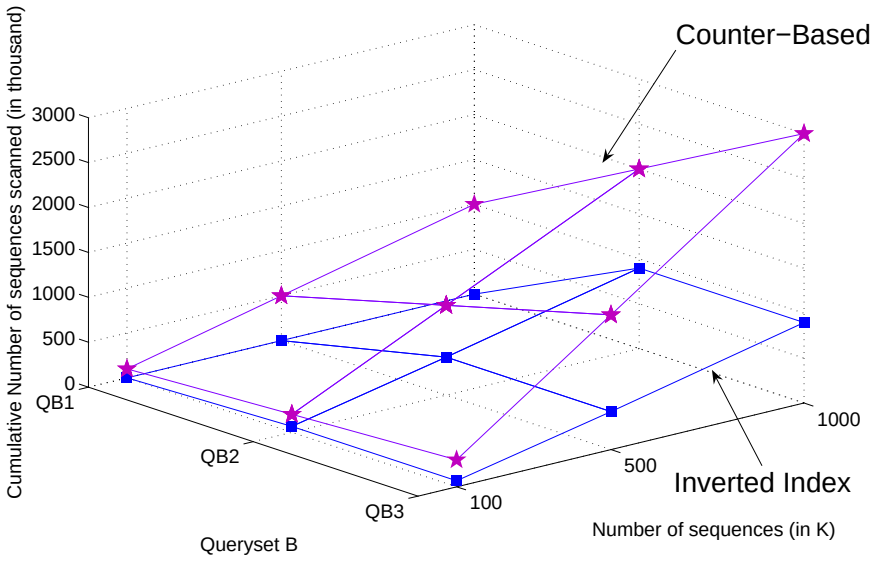

Fig. 25. Cumulative number of sequences scanned of $Q_B.I100.L20.\theta 0.9.Dy$, slice on the subcube with highest count in Q_{B2}.

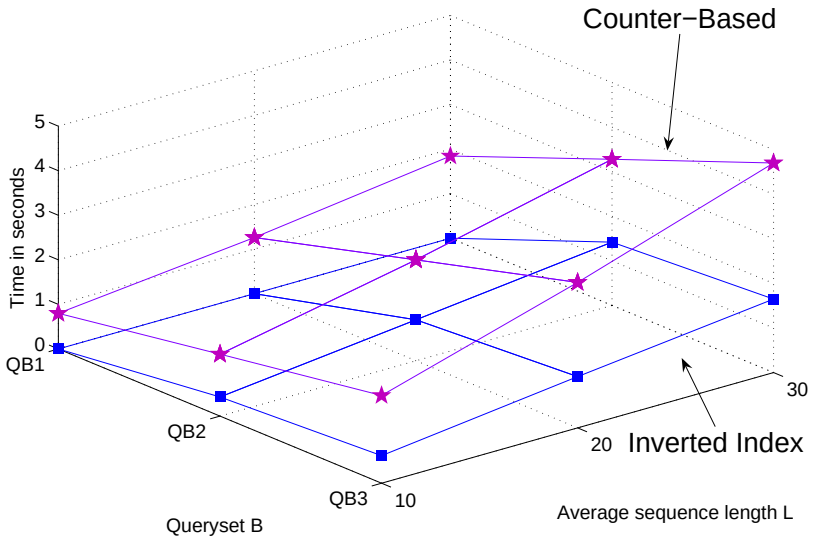

Fig. 26. Cumulative running time of $Q_B.I100.Lx.\theta 0.9.D500k$, slice on the subcube with highest count in Q_{B2}.

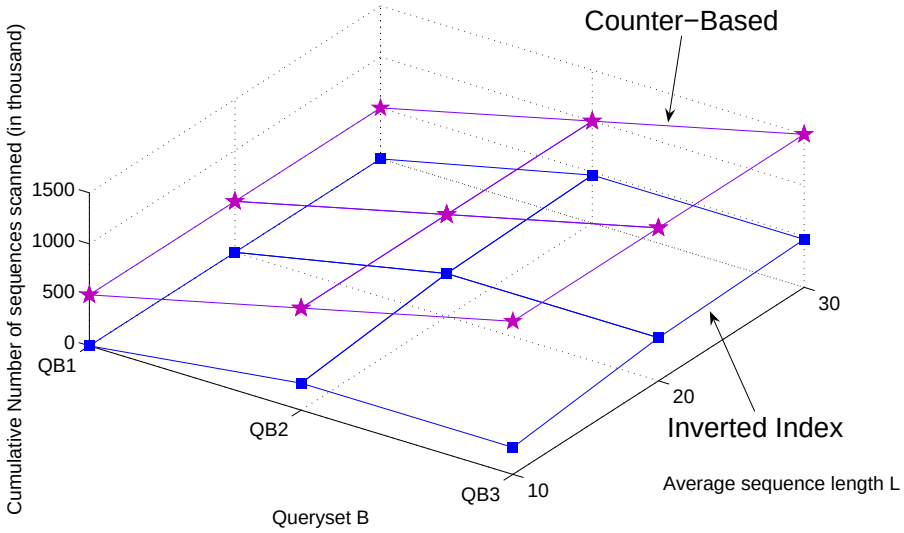

Fig. 27. Cumulative number of sequences scanned of $Q_B.I100.Lx.\theta 0.9.D500k$, slice on the subcube with highest count in Q_{B2}.

computed the answer just by merging the inverted index without scanning the dataset but CB did scan the whole dataset.

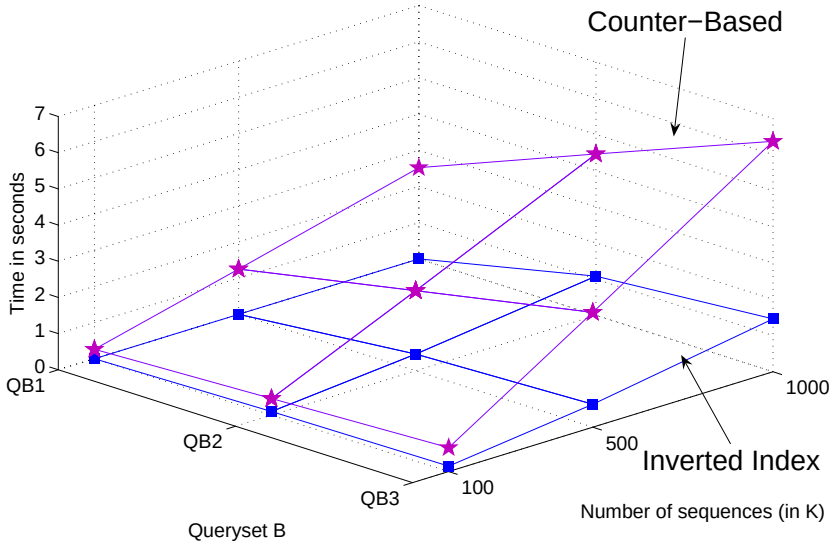

Fig. 28. Cumulative running time of $Q_B.I100.L20.\theta 0.9.Dy$, slice on the subcube with moderate count in Q_{B2}.

QuerySet C – (a) Varying D (b) Varying L. In this experiment we use the substring pattern (X, Y, Y, X) in the query. The dataset in this experiment is the same as the dataset used in Quesy set B (i.e. $I100.Lx.\theta 0.9.Dy$). The query set Q_C consists of two queries Q_{C1}, Q_{C2}. The first query Q_{C1} has a substring pattern templates of $(\mathbb{X}, \mathbb{Y}, \mathbb{Y}, \mathbb{X})$ ($\mathbb{X}$ is the middle abstraction level). The second query Q_{C2} is obtained from Q_{C1} by performing a subcube operation to select the subcube with the same $\mathbb{X}$ value where its total count is the highest among different subcubes and then P-DRILL-DOWN into $\mathbb{X}$, i.e., the pattern template is $(X, \mathbb{Y}, \mathbb{Y}, X)$ (X is the finest abstraction level).

We executed Q_C on datasets with different D (Figure 32, 33) and L (Figure 34, 35) values. In this experiment, the inverted index $L_2^{(\mathbb{X},\mathbb{Y})}$ is precomputed in advance. Therefore in Q_{C1}, II has the following two steps inverted indices joining process: (1) Join $L_2^{(\mathbb{X},\mathbb{Y})}$ and $L_2^{(\mathbb{Y},\mathbb{Y})}$ to obtain $L_3^{(\mathbb{X},\mathbb{Y},\mathbb{Y})}$. (2) Join $L_3^{(\mathbb{X},\mathbb{Y},\mathbb{Y})}$ and $L_2^{(\mathbb{Y},\mathbb{X})}$ to obtain $L_3^{(\mathbb{X},\mathbb{Y},\mathbb{Y},\mathbb{X})}$.

From the experimental results, we see that CB outperformed II in Q_{C1}. The reason is that the inverted indices approach has an extra list joining step. Recall that II has to rescan the dataset in order to eliminate invalid entries in the construction of the inverted indices $L_3^{(\mathbb{X},\mathbb{Y},\mathbb{Y})}$ and $L_3^{(\mathbb{X},\mathbb{Y},\mathbb{Y},\mathbb{X})}$ (Figure 15 line 9). On the other hand, CB doesn't have the overhead and obtain the result in a single dataset scan. For P-DRILL-DOWN (i.e., Q_{C2}), II outperformed CB in all datasets because we sliced on the subcube with the

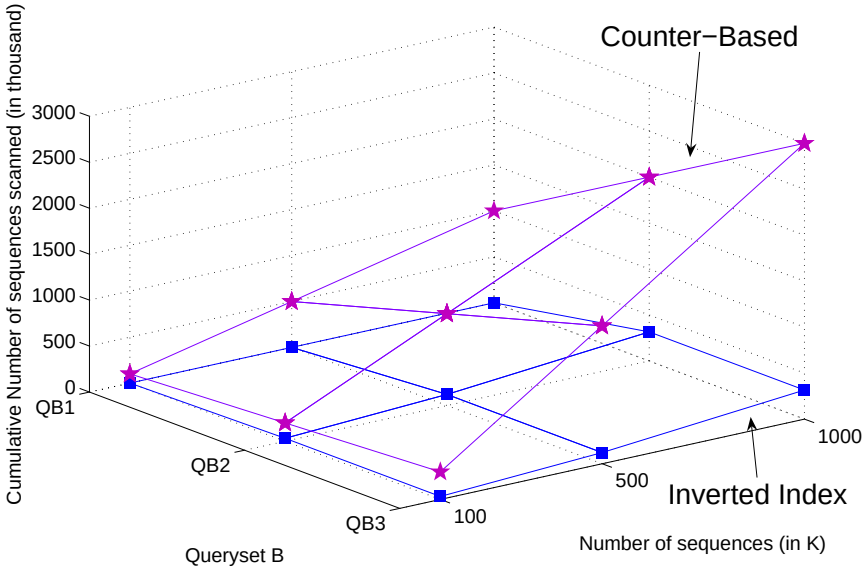

Fig. 29. Cumulative number of sequences scanned of $Q_B.I100.L20.\theta 0.9.Dy$, slice on the subcube with moderate count in Q_{B2}.

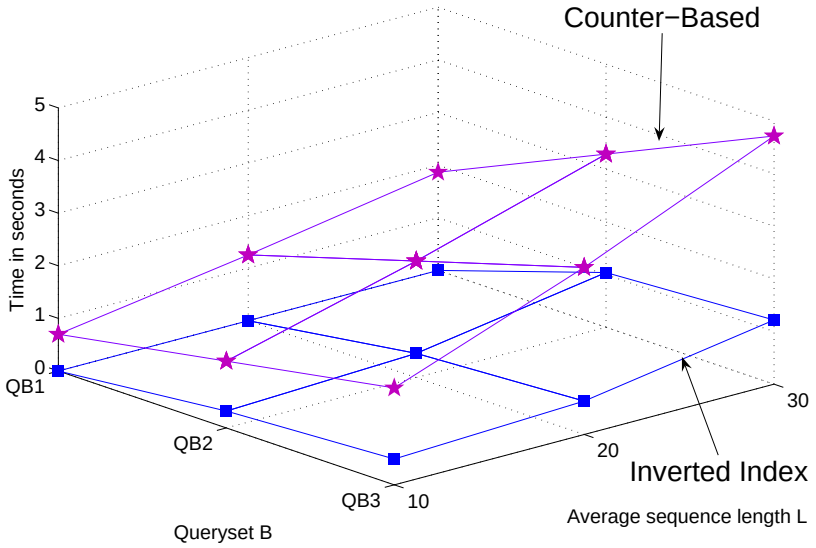

Fig. 30. Cumulative running time of $Q_B.I100.Lx.\theta 0.9.D500k$, slice on the subcube with moderate count in Q_{B2}.

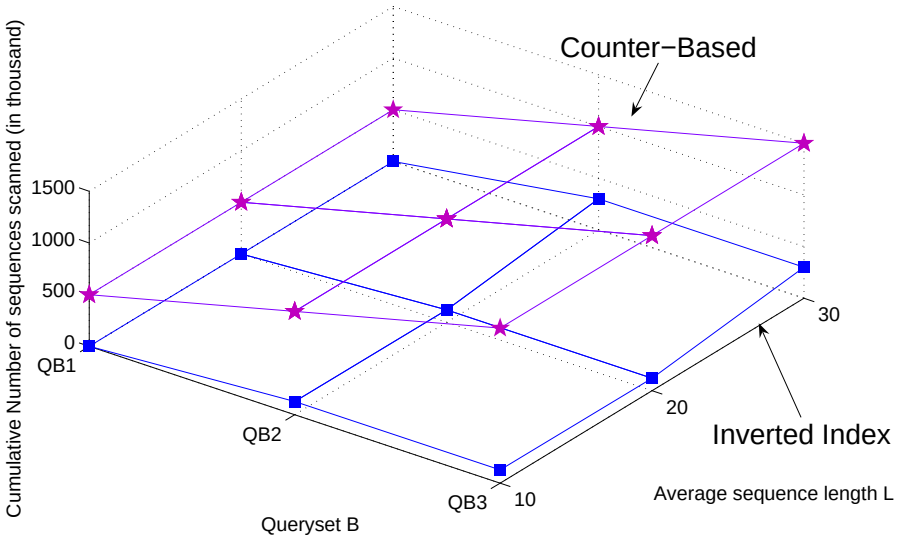

Fig. 31. Cumulative number of sequences scanned of $Q_B.I100.Lx.\theta 0.9.D500k$, slice on the subcube with moderate count in Q_{B2}.

highest count and the substring pattern $(\mathbb{X}, \mathbb{Y}, \mathbb{Y}, \mathbb{X})$ is very selective. Therefore, II only scanned very few number of dataset sequences but CB did scanned the whole dataset.

QuerySet D – (a) Varying D (b) Varying L. In this experiment we repeat Query set C and replace the queries with subsequence patterns. The dataset in this experiment is the same as the dataset used in Quesy set C (i.e. $I100.Lx.\theta 0.9.Dy$). The query set Q_D consists of two queries Q_{D1}, Q_{D2}. The first query Q_{D1} has a subsequence pattern templates of $(\mathbb{X}, \mathbb{Y}, \mathbb{Y}, \mathbb{X})$ ($\mathbb{X}$ is the middle abstraction level). The second query Q_{C2} is obtained from Q_{C1} by performing a subcube operation to select the subcube with the same $\mathbb{X}$ value where its total count is the highest among different subcubes and then P-DRILL-DOWN into $\mathbb{X}$, i.e., the pattern template is $(X, \mathbb{Y}, \mathbb{Y}, X)$ (X is the finest abstraction level).

We executed Q_D on datasets with different D (Figure 36, 37) and L (Figure 38, 39) values. The experimental results draw the following conclusions: (1) CB outperformed II in Q_{D1} because the inverted indices approach has an extra list joining step. (2) The running time of both CB and II scaled linearly w.r.t. the number of sequences in the dataset. (3) The running time of both CB and II scaled exponentially w.r.t. the average sequence length. (4) For P-DRILL-DOWN (i.e., Q_{C2}), II outperformed CB in all datasets because we sliced on the subcube with the highest count and the subsequence pattern $(\mathbb{X}, \mathbb{Y}, \mathbb{Y}, \mathbb{X})$ is very selective. Therefore, II only scanned very few number of dataset sequences but CB did scanned the whole dataset.

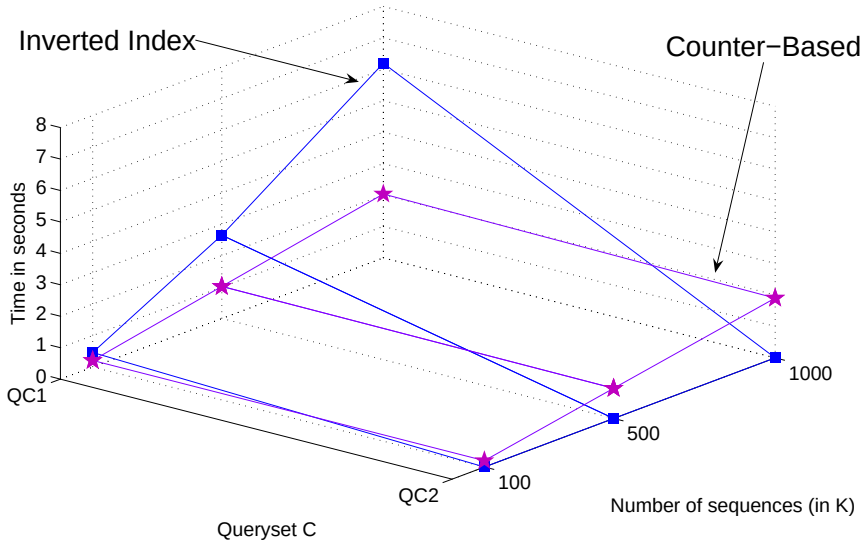

Fig. 32. Running time of $Q_C.I100.L20.\theta 0.9.Dy$.

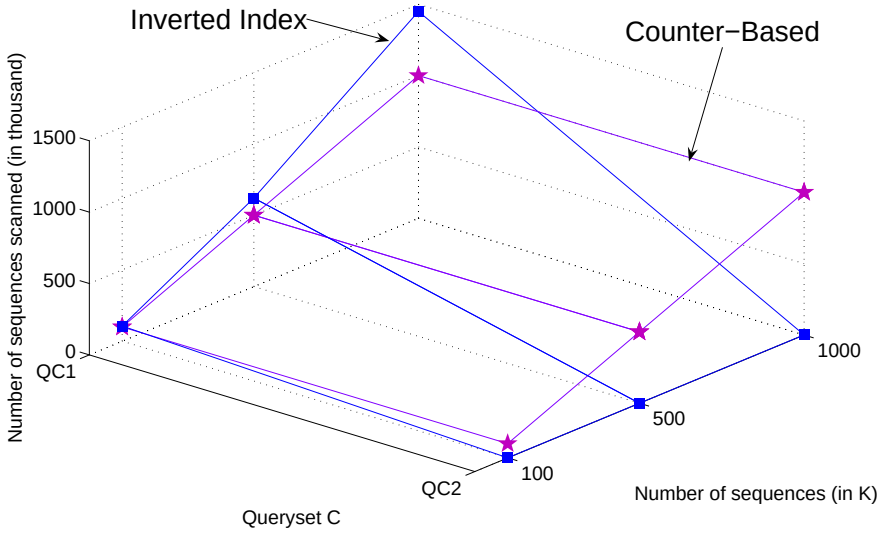

Fig. 33. Number of sequences scanned of $Q_C.I100.L20.\theta 0.9.Dy$.

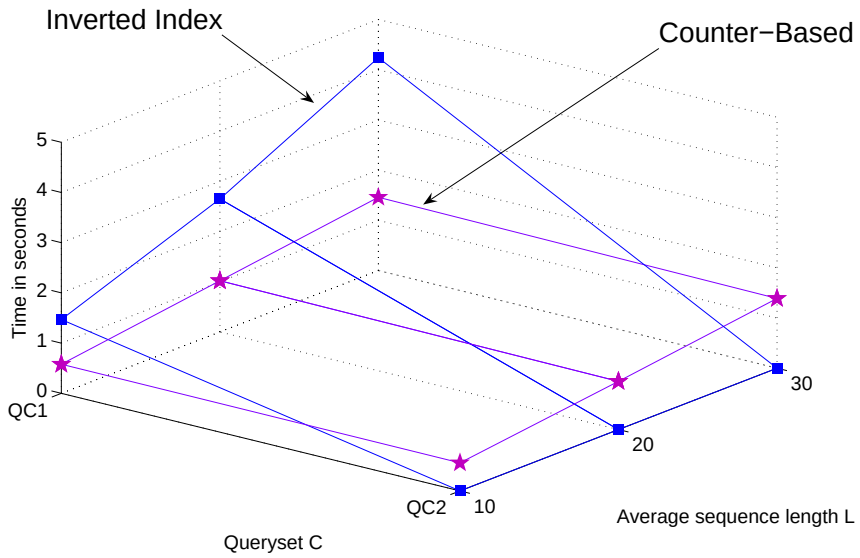

Fig. 34. Running time of $Q_C.I100.Lx.\theta 0.9.D500k$.

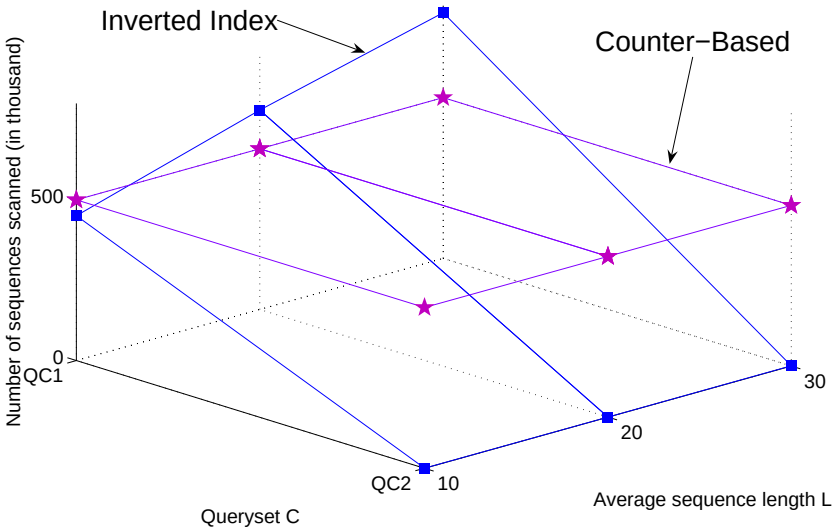

Fig. 35. Number of sequences scanned of $Q_C.I100.Lx.\theta 0.9.D500k$.

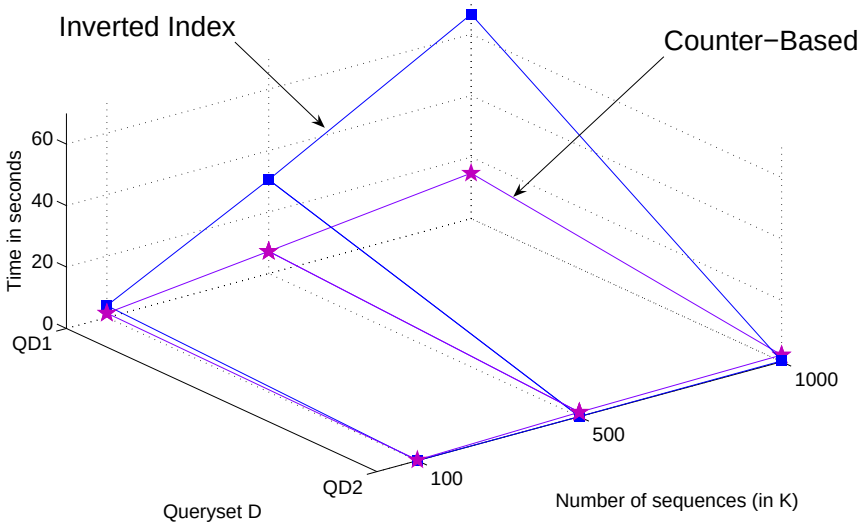

Fig. 36. Running time of $Q_D.I100.L20.\theta 0.9.Dy$.

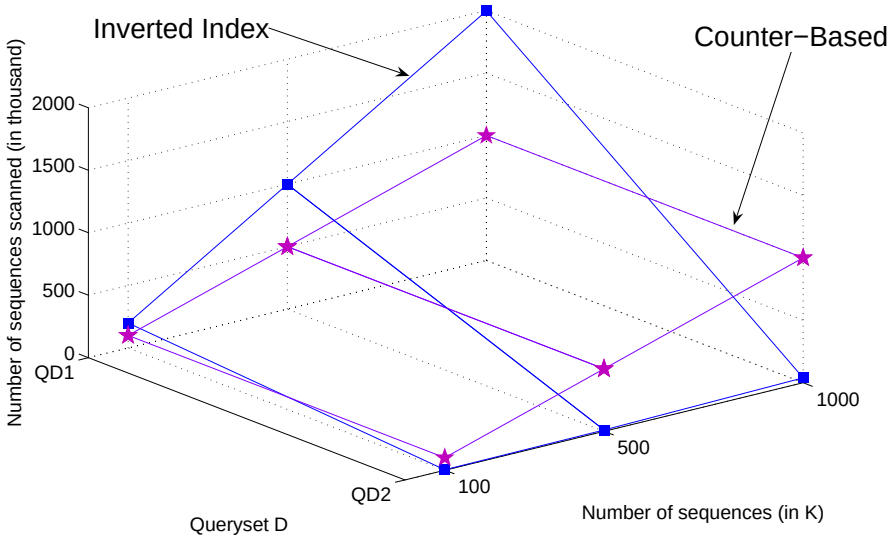

Fig. 37. Number of sequences scanned of $Q_D.I100.L20.\theta 0.9.Dy$.

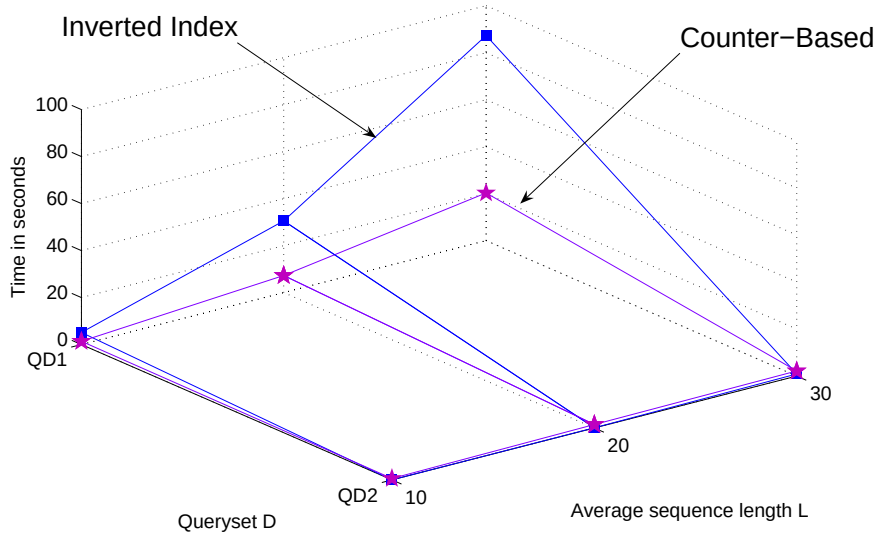

Fig. 38. Running time of $Q_D.I100.Lx.\theta 0.9.D500k$.

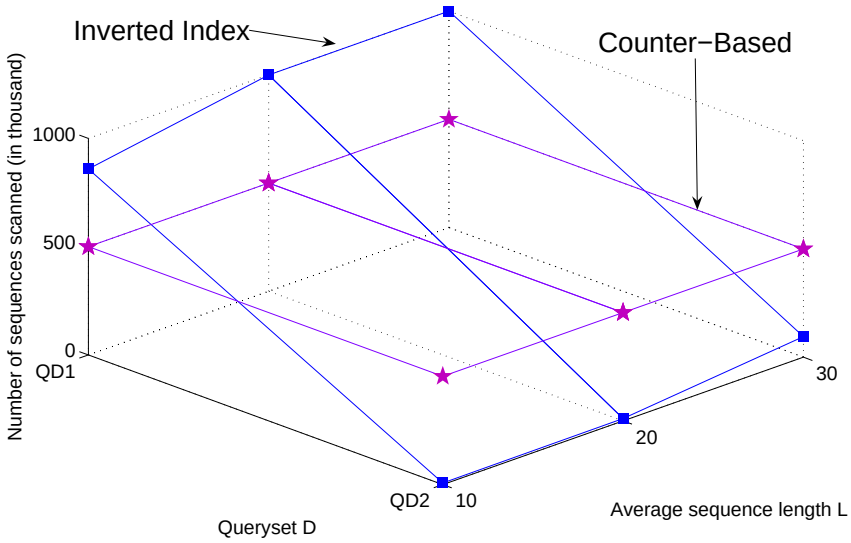

Fig. 39. Number of sequences scanned of $Q_D.I100.Lx.\theta 0.9.D500k$.

6 Discussion

This Sequence OLAP project is initiated by a local subway company which has deployed an RFID-based electronic payment system. Every day, the IT department of the company processes the RFID-logged transactions and generates a so-called "OD-matrix" (stands for Origin-Destination Matrix). The OD-matrix is a 2D-matrix which reports the number of passengers traveled from one station to another within the same day (i.e., representing the single-trip information). The OD-matrix is then sent to various departments for different purposes. For example, the engineering department may refer to the OD-matrix in order to schedule their resources. Occasionally, the management of the company requests more sophisticated reports about the passenger distributions. For example, the management was once considering to offer round-trip discounts to passengers. Consequently, they wanted to know the statistics of various passenger traveling patterns, at different level of summarizations. Our example queries Q_1, Q_2, and Q_3 in this paper were parts of their business queries.

However, since there are no OLAP systems that are capable of performing sequence data analysis, the management has to request the IT department to write customized programs whenever they come up with some business sequence queries. Given the huge volume of data and the administrative overhead, the turnaround time is usually one to two weeks. This inefficient way of sequence data analysis severely discourages data exploration and this problem motivates our project.

The current S-OLAP prototype system is now being reviewed by the subway company. Unfortunately, due to their extremely tight data privacy policy, we cannot report any data-related information here until we have resolved all related legal issues. Nonetheless, throughout this project, we have discovered a lot of interesting research issues and we share our findings with the readers in the remaining of this section. We classify the research issues into different areas: (1) Performance, (2) Incremental Update, and (3) Data Integration and Privacy.

1. Performance. As discussed in Section 4, we regard our two proposed S-cuboid construction approaches as a starting point to more sophisticated solutions to implementing an S-OLAP system. In fact, we realize that many S-cuboid cells are often sparsely distributed within the S-cuboid space (i.e., many S-cuboid cells are empty with zero count). In such a case, introducing an iceberg condition [4] (i.e., a minimum support threshold) to filter out cells with low-support count would increase both S-OLAP performance and usability as well as reduce space. How to determine the minimum support threshold is, however, always an interesting but difficult question.

Another interesting direction is to introduce the online aggregation [10] feature into an S-OLAP system. The online aggregation feature would allow an S-OLAP system to report "what it knows so far" instead of waiting until the S-OLAP query is fully processed. Such an approximate answer to the given query is periodically refreshed and refined as the computation continues. This online feature is especially useful for S-OLAP systems because of the non-summarizable restriction of S-cube. Moreover, an approximate query answer is often adequate for many sequence analysis queries. For example, rather than presenting the exact number of round-trip passengers in Figure

2, approximate numbers like 200,000 for the Pengaton-Wheaton round-trip would be informative enough.

We can also consider improving the performance by exploiting some other indices. For example, if the domain of a pattern dimension is small, we can encode both the base data and the inverted indices as bitmap indices. Consequently, the intersection operation and the post-filtering step can be performed much faster using the bitwise-AND operation rather than using the list-intersect operation. Furthermore, if the domain is really small, the saving in storage space could be very high.

2. Incremental Update. Incremental update is another interesting and practical question for OLAP systems. In many applications like the subway company we are supporting, there is a huge amount of new data being generated every day. When a day of new transactions (events) are added to the event database, we could create a new sequence group and precompute the corresponding inverted indices for that day. However, that new set of transactions (events) may also invalidate the cached sequence groups and the corresponding inverted indices of the same week. As a result, it is necessary to devise methods to incrementally update the precomputed inverted indices.

3. Data Integration and Privacy. Smart-card systems, in addition to paying for subway rides, could be easily extended to new application areas. For instance, in Hong Kong, the Octopus Card can also be used to pay for other modes of public transport, to purchase groceries at supermarkets and convenient stores, and even to pay bills at restaurants [1]. Each month, all vendors who have joined this electronic payment network upload their transactions to a centralized server maintained by an independent company for accounting purposes. Each vendor still owns its uploaded data and the data is not accessible by the others.

However, sometimes, a few vendors may share portions of their data to perform sequence data analysis together. For example, assume that the subway company collaborates with a local bus company and offer a subway-bus-transit package with which passengers who first take the subway and then transfer to a bus would get a 20% discount off the second trip. In order to evaluate the effectiveness of that cross-vendors campaign, lots of sequence OLAP queries would be posed on the passengers traveling history. However, how to integrate the two separately-owned sequence databases (the subway passenger traveling history and the bus passenger traveling history) in order to perform such a high-level sequence data analysis (without disclosing the base data to each other) is a challenging research topic.

7 Conclusions

This paper presented the concept of Sequence OLAP (S-OLAP). The concepts of Sequence Cuboid and Sequence Data Cube are introduced. A prototype S-OLAP system is built and it is able to support pattern-based grouping and aggregation, which is currently not supported by any OLAP system. The implementation details of the prototype system as well as the experimental results of evaluating the system are presented.

Acknowledgment. We thank Jiawei Han, Foris Lee and the anonymous reviewers for their valuable comments.

References

1. Contactless payment and the retail point of sale: Applications, technologies and transaction models. *Smart Card Alliance White Paper*. Accessible at http://www.ntru.com/products/Contactless_Pmt_WP_Final.pdf.
2. K. S. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *SIGMOD*, pages 359–370, 1999.
3. Y. Chen, G. Dong, J. Han, B. W. Wah, and J. Wang. Multi-dimensional regression analysis of time-series data streams. In *VLDB*, pages 323–334, 2002.
4. M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing iceberg queries efficiently. In *VLDB*, pages 299–310, 1998.
5. K. Finkenzeller. *RFID Handbook: Fundamental and Applications in Contactless Smart Cards and Identification*. Wiley, 2003.
6. H. Gonzalez, J. Han, and X. Li. FlowCube: Constructuing RFID FlowCubes for Multi-Dimensional Analysis of Commodity Flows. In *VLDB*, pages 834–845, 2006.
7. H. Gonzalez, J. Han, X. Li, and D. Klabjan. Warehousing and Analyzing Massive RFID Data Sets. In *ICDE*, page 83, 2006.
8. J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. Technical report, Microsoft Research, 1995.
9. A. Gupta and I. S. Mumick, editors. *Materialized views: techniques, implementations, and applications*. MIT Press, 1999.
10. J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, pages 171–182, 1997.
11. R. Kohavi, C. Brodley, B. Frasca, L. Mason, and Z. Zheng. KDD-Cup 2000 organizers' report: Peeling the onion. *SIGKDD Explorations*, 2(2):86–98, 2000.
12. H.-J. Lenz and A. Shoshani. Summarizability in OLAP and Statistical Data Bases. In *SSDBM*, 1997.
13. X. Li, J. Han, and H. Gonzalez. High-Dimensional OLAP: A Minimal Cubing Approach. In *VLDB*, pages 528–539, 2004.
14. E. Lo, B. Kao, W.-S. Ho, S. D. Lee, C. K. Chui, and D. W. Cheung. OLAP on Sequence Data. Technical report, Accessible at www.comp.polyu.edu.hk/∼cscllo/solap.pdf, 2008.
15. P. E. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Record*, 24(3):8–11, 1995.
16. R. Ramakrishnan, D. Donjerkovic, A. Ranganathan, K. S. Beyer, and M. Krishnaprasad. SRQL: Sorted Relational Query Language. In *SSDBM*, pages 84–95, 1998.
17. K. A. Ross and D. Srivastava. Fast computation of sparse datacubes. In *VLDB*, pages 116–125, 1997.
18. R. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi. Optimization of sequence queries in database systems. In *PODS*, pages 71–81, 2001.
19. P. Seshadri, M. Livny, and R. Ramakrishnan. Sequence query processing. In *SIGMOD*, pages 430–441, 1994.
20. P. Seshadri, M. Livny, and R. Ramakrishnan. The design and implementation of a sequence database system. In *VLDB*, pages 99–110, 1996.
21. P. Valduriez. Join indices. *TODS*, 12(2):218–246, 1987.

22. N. Wiwatwattana, H. V. Jagadish, L. V. S. Lakshmanan, and D. Srivastava. X^3: A Cube Operator for XML OLAP. In *ICDE*, pages 916–925, 2007.