Available online at www.sciencedirect.com
ScienceDirect
Data & Knowledge Engineering 59 (2006) 700–724
DATA & KNOWLEDGE ENGINEERING
www.elsevier.com/locate/datak

Extending the data warehouse for service provisioning data

Yannis Kotidis *

AT&T Labs-Research, 180 Park Ave., Florham Park, NJ 07932, United States

Received 1 December 2004; received in revised form 14 November 2005; accepted 15 November 2005
Available online 15 December 2005

Abstract

The last few years, there has been an extensive body of literature in data warehousing applications that primarily focuses on basket-type (transactional) data, common in retail industries. In this paper we focus on *service provisioning data*, that is data that is recorded internally in an organization for provisioning certain business related tasks. Coupling the recorded data with the underlying process and business-practice(s) that generate them is crucial for end-to-end analysis. Our framework is based on a graph description of the process (called a *sketch*) that is generating this data. Using this sketch, we formalize a new class of aggregate queries that consolidate data from a part of the process, based on a user defined path expression. We then show how to build a compact, non-redundant collection of summary (aggregate) tables and indices for this new type of queries. We first explore how to select a minimum set of views to answer queries with path-expressions over the given sketch. For queries that also include aggregation, we define two partial orders among the views. The first is used to pick the minimum set of aggregate views to answer any query with no false dismissals, while the second describes an augmented set that allows fewer false positives. Computing a non-materialized aggregate is done through appropriate rewriting of the user query. We describe two indexing schemes that use *phantom* (non-materialized) aggregate values to expedite query processing. Experimental results show these schemes to perform well on synthetic and real datasets.

Keywords: Data warehouse; Workflow; Materialized views

1. Introduction

The astonishing success of information technology has resulted in an explosive growth in the amount of data that is being recorded in daily basis on various domains. This abundance of data has in-turn driven the data warehousing sector into a major segment of the information technology market and, subsequently, has attracted a lot of interest from the academic community. The data warehouse is an integrated informational store that provides stable, point-in-time data for decision support applications. Data-rich industries, like retail and financial services, have been the most typical users of data warehousing technology for the obvious reason that they have large quantities of good quality internal and external data available, to which they need to add value.

* Tel.: +1 9733608347.
E-mail addresses: kotidis@research.att.com, kotidis@cs.umd.edu

0169-023X/$ - see front matter
doi:10.1016/j.datak.2005.11.003

Often data is recorded internally in an organization for provisioning certain business related tasks. During the last few decades there has been an increasing trend to computerize every possible business process and eliminate manual hand-offs. Workflow management and Customer Relationship Management (CRM) software are being used to help manage customer relationships in an efficient and organized manner. For example, in the telecom sector, accepting a new customer for long distance service involves several steps from entry and verification of personal data to third party verification (a process in which a designated third party confirms the customer's intention to change service), and finally placing a new entry in the company's billing database. A process known as *revenue assurance* verifies the beginning-to-end completeness, accuracy, and integrity of the capture, recording, billing, and reporting of all revenue-producing events from customer order entry through collection.

The type of data generated from these processes does not conform to the basket paradigm, regularly used in data warehousing. Informally, a service provisioning database contains a large collection of customer records. Each record describes a sequence of events that captures the interaction between a customer's order and various components of the organization. Presumably, there is a well defined workflow that describes the flow of events for incorporating a customer's order [43]. Each customer record is an instance of some part of this workflow process annotated with timing information and other business related attributes. Each order is then treated as an "entity", which flows through the service's workflow. Network traffic is another example in which the recorded data conforms to an underlying structure: the network topology. Any end-to-end communication of network elements uses a collection of network paths directed by the routing protocols and the physical interconnect among them.

When trying to apply conventional data warehousing techniques for a service provisioning database, we are faced with the problem of mapping the recorded data into a relational schema in a way that allows complex analytical queries over the recorded information with respect to the structural properties of the process that generates these records. In our framework, the user expresses his intention to analyze the data at a particular resolution for some portion of the process by providing what we call a *sketch* of the process. The sketch is a graph representation of the underlying process, containing nodes representing "states" and edges representing "transitions". Given a sketch, we explore how to build a compact, non-redundant collection of summary tables and indices to facilitate flexible decision support analysis.

As we will demonstrate, conventional relational implementations are incapable of providing flexible analysis of the recorded data with respect to a given sketch [16]. In the basket-data paradigm a multi-dimensional approach is used, in which data, representing transactions, is projected and aggregated using a set of dimensions (like products and customers). In a service provisioning database the structural properties of the sketch are the dimensions of interest. In this paper we introduce pair-wise aggregate queries as a means to describe the *scope* of an interesting aggregation. Such a query consists of a *path expression* over the graph (sketch) that collects relevant recorded information from parts of the underlying process and a user defined aggregate function $\mathscr{F}$ that consolidates this data. We first formally define pair-wise aggregate queries and then show how to express more complex expressions and evaluate them as a series of pair-wise queries. In addition, we explore operations that allow us to zoom in and out of certain states of the process, or exclude parts we are not interesting in. As will be explained these operations are naturally mapped into pair-wise queries that consolidate the underlying records.

For large datasets, evaluating pair-wise aggregate queries on the fly can be prohibitively expensive. Often there is no efficient way to simultaneously evaluate path and aggregate expressions over the data. Materialized views can be used to speed up query processing and also shield the user from the details of mapping a complex aggregate expression to the underlying relational schema. We first show how to optimize execution of queries with path expressions using a non-redundant collection of views materialized as bitmapped indexes [10,42,50]. For queries that also contain aggregations we propose the use of pair-wise aggregate views that contain precomputed results of pair-wise queries and discuss different materialization policies. These views can be used to answer arbitrary pair-wise aggregate queries without imposing any false negatives (i.e. omit records that should have been in the answer) and with provably less cost than a regular index scan. We then show how to rewrite queries to use these views based on a partial order that we define among them. This rewriting works efficiently using an indexing scheme that partitions the records of a view on non-materialized *phantom aggregates*, in a way that allows efficient evaluation of subsequent queries against these aggregates.

1.1. Map

The rest of the paper is organized as follows: Section 2 motivates the problem from two practical applications. Section 3 discusses related work. In Section 4 we introduce our framework, define pair-wise queries and materialized pair-wise views and discuss the shortcomings of various relational models for the service provisioning data. In Sections 5 and 6 we show how to choose from, index and query these aggregate views for answering user queries. Finally Section 7 contains the experiments and in Section 8 we draw the conclusions.

2. Motivation

We here present two examples of service provisioning that will help us better motivate the discussion.

2.1. Telephone service provisioning

Telephone service provisioning includes several steps, starting with the reception of customer's order and ending with the establishment (or modification) of service. The workflows related to this process are typically very complicated because orders require processing by many departments. Fig. 1 provides a simplified high level view of this workflow for long distance orders. There are five major states modeled in this example. *Create* involves all actions related to the reception and creation of a new order. *TPV* stands for third-party verification and *LEC* stands for Local Exchange Carrier. The latter state includes all communications with the *LEC* to establish the caller as a new customer. State *Billing* involves all actions related to creation or modification of a customer's billing record. Finally, state *Complete* denotes the successful completion of the whole process. A customer's order is modeled as an entity that flows through this process. States *TPV* and *LEC* are only used for orders that involve creation of a new long distance service. Orders of customers that call to modify their plans (e.g. sign to a new promotion) skip these states. Each of the five main states can be expanded and modeled in more details. Depending on the application, we might want to "drill-down" on a state and analyze the flow of records at a finer granularity.

The user expresses his intention to analyze the data at a particular resolution for some portion of this process by providing what we call a *sketch*. A sketch is a directed acyclic graph (DAG) describing states and transitions that he is interested in (later on we extend our discussion for sketches with cycles). For example if the user is interested in the five depicted states of the workflow of Fig. 1, the sketch is simply a DAG representation of the process in which each state is mapped to a node and each transition is mapped to an edge. In this example the process at this particular level happens to be acyclic too but this is not required in general.

A sketch is an abstraction of the whole process and is used to filter those events that flow through the specified nodes and edges of the DAG. For example if there were a *Failed* state in the process that is not included in the sketch then the analysis will only target records of orders that have successfully completed. Analysis will be based on data attributes collected by the recording mechanisms as well as the structural properties of the given sketch. Examples of such queries include:

1. Retrieve all orders that passed through states *TPV* and *LEC* (e.g. new long distance customers).
2. Find all orders for which an intermediate transition from state *Create* to state *Complete* took more than 8 h, while the order was completed in less than a day (e.g. trace "hidden delays").
3. Find all orders that were modified (possibly due to initial data entry errors) more than once between states *TPV* and *Billing*.

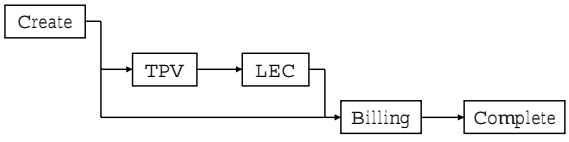


Fig. 1. Telephone service provisioning example.

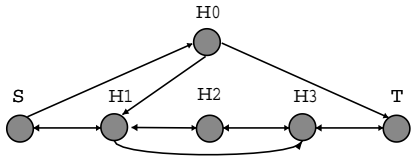


Fig. 2. Delivery service provisioning example.

Query (1) is an example of a path-query, discussed in Section 5. Query (2) inquires timing information recorded as the order flows through the process. This is critical for most service ordering systems as user satisfaction is primarily based on timely processing of his orders [6,17]. Surprisingly, the area of handling time-related issues and detecting potential problems has not received adequate attention in the workflow literature [17]. For query (3) we exploit a version attribute attached to each record that counts all modifications to the order.

2.2. Delivery service provisioning

Fig. 2 depicts a number of "hubs" H_i that are used to interconnect two sites S and T. Examples of such a scenario can be found in different domains. In a network service provisioning system, S and T can be two sub-networks and hubs H_i will be the network elements (e.g. routers) that provide the interconnect among these two networks. In a packet delivery service (like FedEx) the picture of Fig. 2 may describe the network of company's locations and the connectivity among them. Connecting paths may have different capacities, bandwidth, latencies etc. Some connections are bi-directional while others not, as shown in the figure. Assuming that we want to provision delivery of services from site S to site T, our sketch is obtained by making all bi-directional edges in the figure be pointing from left to right. Given this sketch, the queries that we are interested in include:

4. Find all flows from S to T that utilized hub $H0$.
5. Find all flows from S to T that stopped at least at 3 hubs.
6. Find all flows from $H1$ to $H3$ for which each transition required at least 1 day.

Query (4) is a simplified path-query on a single node. In query (5) we "aggregate" multiple paths from S to T by counting the number of intermediate hubs in a flow from S to T. An equivalent, but more cumbersome, way to state the query is to ask for all flows of the form $S \to H1 \to H2 \to H3 \to T$, $S \to H0 \to H1 \to H2 \to H3 \to T$ or $S \to H0 \to H1 \to H3 \to T$. Finally query (6) requests a specific path from the sketch along with additional timing constraints.

3. Related work

Within the past decade we have witnessed renewed excitement in decision support tools and applications. Advances in information technology and globalization of businesses created the right combination of "supply" and "demand" to fuel the data warehousing field.

The primal goal of a data warehouse is to provide an integrated data store for the execution of complex analytical queries. Such queries often involve aggregation. Because of the size of the data at-hand and the plethora of choices for ad hoc grouping and aggregation, data warehouses rely on pre-computation and extensive indexing of the data. Bitmapped indexes and their variations [41,10,42,50] have found their way into most industrial solutions. Another form of pre-computation is the use of materialized views containing frequently asked aggregates. Engineering questions, such as how many and which views to materialize under a space and/or update time constraint and an expected query workload have led to several view selection algorithms [28,47,48,27,4] as well as alternative dynamic organizations [34,45,46]. In this paper, we too rely on pre-computation for speeding up aggregate queries that arise in the process of analyzing service provisioning data. Our definition of pair-wise queries has been motivated from related literature on recursive queries (e.g.

[35,31,44,49]). As will be explained, we use the structural properties of the underlying process that generates the data to define a partial order among the views. Our primary focus is on selecting the minimum number of views that can answer any query of interest without false dismissals. If additional resources are available, using the partial order we define among the views, one can easily modify the algorithms of e.g. [28] for selecting additional views in the materialized set. Implementation of the aggregate views we discuss in this paper can be done using bitmapped indexes and other standard relational tools like B-trees. We further exploit the dependencies among the views (that have been discovered from analyzing the process) to extend traditional indexing schemes and achieve even better query performance.

Dimensional modeling is a specialization of the traditional ER modeling [12,40]. It distinguishes *facts* (like credit-card transactions), from *dimensions* that provide a context for the facts (like time, product, location). Most common examples of dimensional modeling include the stars schema, the snowflake schema and their extensions like the federated star schemas [33,11]. In the service provision example, the underlying process provides the context within which data is modeled and analyzed. In this paper, we assume that the underlying process is given. When the process is unknown, one can infer it using mining techniques such as the ones described in [1,14].

Workflow management systems are extensively used for process simulation, in order to identify bottlenecks and analyze execution durations of business tasks [17,38,30]. They are also used in office automation for assignment of tasks and execution monitoring (triggering alarms when deadlines are missed, exemption handling etc.) [32]. In the database community there has been extensive work on extending transactions for workflow applications [13,39,9,5,21,7,3]. In this paper we focus on modeling and organizing the data for post-processing and analysis. Our work aims on utilizing existing data warehousing techniques for the processing and analysis of data generated from real business processes. This is an area that has received little attention in the literature. Grigori et al. in [26] discuss a set of *Business Process Intelligence* tools for cleaning and aggregating workflow logs into a warehouse for analysis, prediction and optimization of business processes. In [18] the authors make a case for using data On-Line Analytical Processing (OLAP) tools in analyzing workflow logs and present a generic data warehouse design. In [8] the authors identify some major challenges in designing data warehouses for managing workflow data such as the presence of multiple related facts that lead to complicated schemas, the complexity of aggregations required due to this fact and the volatility of workflow models that may require frequent substantial changes in the warehouse design. The authors of [36] follow an object-oriented approach driven by use case and object models for modeling business requirements for the data warehouse.

4. A framework for service provisioning data

4.1. Data model

We first provide a formal definition of a sketch. Given a process that is being provisioned, a sketch is a directed acyclic graph $G(V,E)$ with the following properties: each node v in V corresponds to a particular state of the process (seen at the desirable resolution). An edge $e=(v_1,v_2)$ represents a valid transition between corresponding states v_1 and v_2 in the process that we are observing. There is a set $S \in V$ of *starting nodes* in the sketch i.e. nodes that have no incoming edges in $G(V,E)$. Similarly all nodes with no outgoing edges form a set of *terminal nodes* T. In Section 5.3 we extend the definition to graphs that include cycles.

A record r is called *relevant* to a sketch $G(V,E)$ if it describes a transition from some starting node $s \in S$ to a terminal node $t \in T$ through nodes of $V-S-T$ using edges in E. For this model we assume that information is being recorded at every edge traversed in r and stored in attributes called *measures*. For simplicity in the notation we will only refer to examples with a single numeric measure denoted as x. In that case, a record r is an ordered set of pairs:

$$r=\{(e_1,x_1),\ldots,(e_{k_r},x_{k_r})\} \tag{1}$$

$\mathscr{R}$ denotes the whole collection of records that are relevant to the sketch. Sometimes, specific nodes may have measure data recorded too. This is useful in order to trace intra-node processing. In these cases, we replace such a node v with a linked pair v_1, v_2. Edge (v_1,v_2) is then used to store the intra-node measure.

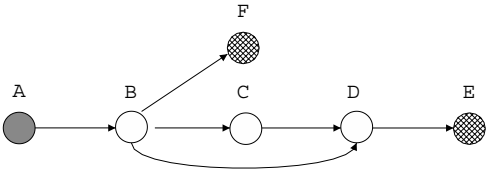


Fig. 3. Simple sketch.

Table 1
Examples of valid records for the sketch of Fig. 3

Original representation	Dual representation
$r_1 = \{(e_{(A,B)}, 5), (e_{(B,C)}, 7), (e_{(C,D)}, 8), (e_{(D,E)}, 2)\}$	$r_1^d = \{(v_A, 5), (v_B, 7), (v_C, 8), (v_D, 2), (v_E, \dashv)\}$
$r_2 = \{(e_{(A,B)}, 6), (e_{(B,D)}, 4), (e_{(D,E)}, 3)\}$	$r_2^d = \{(v_A, 6), (v_B, 8), (v_D, 3), (v_E, \dashv)\}$
$r_3 = \{(e_{(A,B)}, 4), (e_{(B,F)}, 1)\}$	$r_3^d = \{(v_A, 4), (v_B, 1), (v_F, \dashv)\}$

A dual representation of a record, denoted as r^d is derived by using state information [1]:

$$r^d = \{(v_1^{e_1}, x_1), (v_1^{e_2}, x_2), \ldots, (v_1^{e_{k_r}}, x_{k_r}), (v_2^{e_{k_r}}, \dashv)\} \tag{2}$$

As a running example we will be using the sketch of Fig. 3. Node A is a starting node and nodes F, E are terminal nodes. Table 1 shows examples of relevant records for that sketch in their original and dual representation.

When information is only recorded at the nodes, we use notation (1) on the dual graph of the sketch (e.g. by switching nodes and edges).

4.2. Pair-wise queries

In OLAP, a multi-dimensional approach to analysis is used to align the data content with the analyst's mental model. In the basket-data paradigm such dimensions are the products and customers involved in a transaction [24]. The measures are then projected and evaluated over the selected dimensions. In a service provisioning database the structural properties of the sketch are the dimensions of interest.

In order to analyze the data we need to specify parts of the sketch as the *scope* of our analysis and then compute interesting aggregates on the relevant measures. Since G is acyclic, any two distinct nodes u and v for which there is a path $u \rightarrow v = (u = v_1, v_2, \ldots, v_k = v)$ from u to v in G define a selection filter over the stored records.

Definition 4.1. A pair-wise selection filter $S_{u\rightarrow v}(\mathscr{R})$ returns all records that contain a transition from u to v. For each qualifying record $r = \{(e_1, x_1), \ldots, (e_{k_r}, x_{k_r})\}$ there exist i, j such that $1 \leqslant i \leqslant j \leqslant k_r$ and $e_i = (u, ?)$, $e_j = (?, v)$, where ? denotes a "don't care" value.

We further define $S_{u\rightarrow u}(\mathscr{R})$ to return the records that contain node u.

An additional operator is required to gain access to the measures collected along the path $u \rightarrow v$ for each qualifying record in $S_{u\rightarrow v}(\mathscr{R})$:

Definition 4.2. The projection operator $P_{u\rightarrow v}(r)$, when applied on record $r = \{(e_1, x_1), \ldots, (e_{k_r}, x_{k_r})\}$ returns the set of recorded measure data along the path $u \rightarrow v$: $P_{u\rightarrow v}(r) = \{x_i, \ldots, x_j\}$ with $1 \leqslant i \leqslant j \leqslant k_r$, $e_i = (u, ?)$ and $e_j = (?, v)$.

[1] $\dashv$ denotes a sentinel "end-of-record" value. It is necessary when a node is linked to more than one terminal nodes (e.g. the last pair can be omitted in this example).

By utilizing the projection operator, we can then apply any interesting aggregate function like *sum*( ), *count*( ), *min*( ), *max*( ), *median*( ) etc. over the (set of) selected measures. We use $\mathscr{F}(P_{u\to v}(r))$ to denote the result of aggregate function $\mathscr{F}$ when applied to measures $x_i,\ldots,x_j$ collected along a path $u \to v$ for record r. In what follows we also use the binary function *exist*( ), which simply evaluates to 1 (true) when its input is not empty, i.e. $exist(P_{u\to v}(r)) = 1$ when r is in $S_{u\to v}(\mathscr{R})$.

Definition 4.3. A pair-wise aggregation query $l \leqslant \mathscr{F}_{u\to v} \leqslant h$ when applied on a set of records $\mathscr{R}$, retrieves all records $r \in S_{u\to v}(\mathscr{R})$ that satisfy the condition $l \leqslant \mathscr{F}(P_{u\to v}(r)) \leqslant h$.

The definition is also extended to strictly greater-than/less-than operators and single-sided queries. For brevity, the set of records $\mathscr{R}$ is omitted when we discuss a query $l \leqslant \mathscr{F}_{u\to v} \leqslant h$.

Example 4.4. Queries 1–6 of Section 2 are written as

1. $exist_{TPV\to LEC} = 1$
2. $max_{Create\to Complete} > 8\ hours\ AND\ sum_{Create\to Complete} < 1\ day$
3. $sum_{TPV\to Billing} > 1$
4. $exist_{H0\to H0} = 1$
5. $count_{S\to T} \geqslant 3$
6. $min_{H1\to H3} \geqslant 1\ day$

4.3. Process navigation

In a service provisioning database, the underlying process provides the schema context within which data is modeled and analyzed. We here define three operations that allow us to zoom in and out of particular states of the process, or eliminate (hide) parts we are not directly interested at. The first two operations, `Zoom` and `UnZoom` allow a user to examine a business process at different resolutions. The `Zoom` operation replaces a node in the sketch with a sub-process that describes the internal processing happening on the node. `Zoom` is roughly related to a drill-down operation in OLAP, where data is examined at a progressively finer granularity. `UnZoom` provides the reverse functionality. The last operation, namely `Hide`, is used to hide details on pieces of the process. `Hide` relates to `Zoom/UnZoom` in that is allows us to abstract the process, but, unlike `Zoom/UnZoom`, we can freely manipulate the sketch without adhering to a predefined hierarchical decomposition. This is useful for ad hoc type of analysis or when data is presented to a third party and we want to conceal certain details.

Bellow we discuss these operations in more details and give examples.

Zoom/UnZoom: The `Zoom` operation is used to examine a node in the sketch in more details. In particular, `Zoom` replaces a node u with a sketch G_u describing the "internals" of node u in more details. In addition to G_u, `Zoom` requires two mapping functions M_{in} and M_{out}. Function M_{in} maps an incoming edge to u in the original sketch to a starting node in G_u. Function M_{out} maps every terminal node in G_u to a node in G. In Fig. 4 we

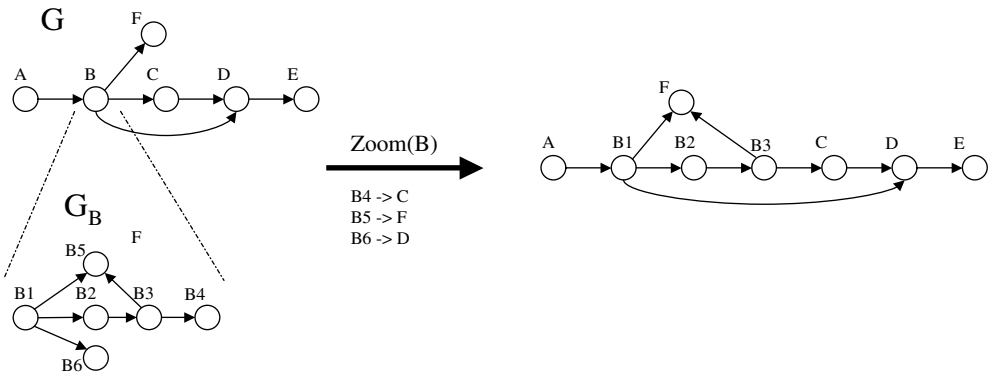


Fig. 4. Zoom operation.

show an example of zooming into node B in the sketch of Fig. 3. Original edge (A, B) is replaced by edge $(A, B1)$, since $B1$ is the starting node in G_B. Similarly, nodes $B4$, $B5$ and $B6$ are mapped to nodes C, F and D as shown in the figure.

`UnZoom` provides the reverse functionality. When we move from a more fine-grained description of the process to a coarser one, we need to provide an aggregation method for coalescing the fine-resolution measurements. The exact operation is application specific. For instance, when measurements describe a cost-related attribute then we can use the *sum*( ) function. In that case, for reverting to the original sketch in Fig. 4, `UnZoom` requires the following aggregations:

$$x_{BC} = sum_{B1\to C} = sum_{B1\to B4}$$

$$x_{BF} = sum_{B1\to F} = sum_{B1\to B5}$$

$$x_{BD} = sum_{B1\to D} = sum_{B1\to B6}$$

We note that all these operations can be described in the context of pair-wise aggregate queries.

Hide(u): this operation results in a simpler sketch by removing node u from consideration. For a sketch $G(V, E)$, `Hide`(u) results in a new sketch $G'(V - \{u\}, E')$ where

$$E' = E \cup \{(a, b) : (a, u), (u, b) \in E\} - (\{(u, ?) \in E\} \cup \{(?, u) \in E\})$$

Fig. 5 shows the resulting sketches from three successive calls: `Hide`(B) followed by `Hide`(D) and `Hide`(C). The first call removes node B from the sketch. Incoming edge (A, B) is replaced by edges (A, F), (A, C) and (A, D). Edge (A, F) "hides" the detailed transition $A \to B \to F$. For records that contain this transition, the two measures x_{AB} and x_{BF} are replaced by a derived measure x_{AF}. Like the case of the `UnZoom` operation, the exact formula depends on the context. For instance, if the measures relay cost information, one can use $x_{AF} = x_{AB} + x_{BF} = sum_{B\to F}$.

4.4. Processing of pair-wise queries in a relational data store

A natural attempt to store this data in a relational system is to break each record into a list of (record-id, edge-id, measure) triplets using table: $\mathcal{R}(r_{id}, e_{id}, x)$. This vertical representation has an excellent effect when querying on transitions between two adjacent nodes: a pair-wise query $\mathcal{F}_{u\to v}$, where $(u, v) \in E$ is efficiently computed given an index on e_{id}. However, for paths $u \to v$ with one or more "hops" the query requires a number of self-joins of $\mathcal{R}$ equal to the length of the path in order to "collect" all measures along the path. Furthermore, in case there are more that one paths between the selected nodes, the user has to implicitly write an SQL sub-query for each one of them. For example query $sum_{B\to D} \geqslant 5$ returns the union of the following expressions:

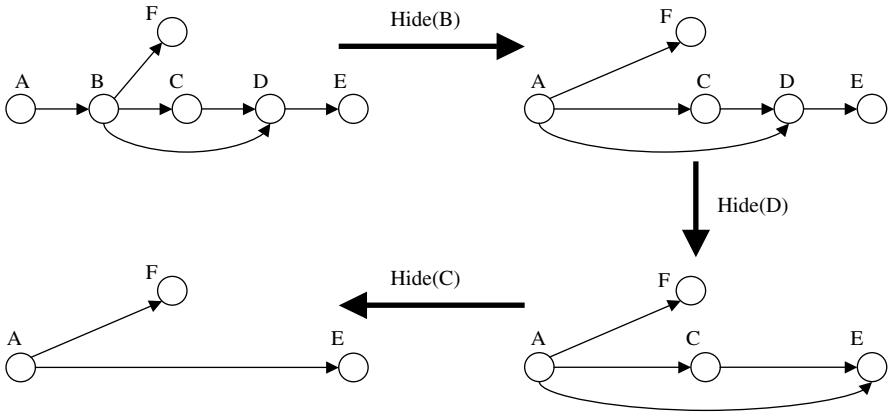


Fig. 5. Hide operation.

q_1: select $R_1.r_{id}$, $R_1.x + R_2.x$
from $\mathscr{R}R_1, \mathscr{R}R_2$
where $R_1.r_{id} = R_2.r_{id}$ and $R_1.e_{id} =$ "BC"
and $R_2.e_{id} =$ "CD" and $R_1.x + R_2.x \geqslant 5$
q_2: select $R.r_{id}$, $R.x$
from $\mathscr{R}R$
where $R.e_{id} =$ "BD" and $R.x \geqslant 5$

An alternative horizontal representation of the data is: $\mathscr{R}^h(r_{id}, x_{e_1}, \ldots, x_{e_{|E|}})$, where x_{e_i} is the measure for edge e_i. If a record does not contain a transition, the corresponding x_{e_i} value is null. Compared to the vertical representation, the horizontal schema avoids the need for self-joins, but the user is still required to implicitly describe all paths between the end-points u, v. The previous query is now written as

q_3: select $\mathscr{R}^h.r_{id}$, $\mathscr{R}^h.x_{BC} + \mathscr{R}^h.x_{CD}$
from $\mathscr{R}^h$
where $\mathscr{R}^h.x_{BC} + R^h.x_{CD} \geqslant 5$
q_4: select $\mathscr{R}^h.r_{id}$, $\mathscr{R}^h.x_{BD}$
from $\mathscr{R}^h$
where $\mathscr{R}^h.x_{BD} \geqslant 5$

A potential problem of the horizontal representation is that most commercial database systems impose a limit on the number of columns in a table that can be reached when $G(V, E)$ is large and multiple measures are being collected along each edge.

For aggregate functions like *sum*( ) and *count*( ) we can exploit a dual prefix-representation [22,29] of the record:

$$r^d_{\mathscr{F}} = \{(v_1^{e_1}, 0), (v_1^{e_2}, \mathscr{F}(x_1)), \ldots, (v_1^{e_{k_r}}, \mathscr{F}(x_1, \ldots, x_{k_r-1})), (v_2^{e_{k_r}}, \mathscr{F}(x_1, \ldots, x_{k_r}))\} \quad (3)$$

The prefix-$\mathscr{F}$ representation allows us to compute the aggregation by subtracting the prefix-representation of the measure at the ending node from the prefix-representation at the starting node using table $\mathscr{R}^d_{\mathscr{F}}(r_{id}, v_{id}, x_{\mathscr{F}})$. Thus, a pair-wise query $sum_{u\to v}/count_{u\to v}$ is expressed as a single self-join that can be optimized using an index on the v_{id} column.

For example $sum_{B\to D} \geqslant 5$ is written as

q_5: select $R_1.r_{id}$, $R_2.x_{sum} - R_1.x_{sum}$
from $\mathscr{R}^d_{sum}R_1, \mathscr{R}^d_{sum}R_2$
where $R_1.r_{id} = R_2.r_{id}$
and $R_1.v_{id} =$ "B" and $R_2.v_{id} =$ "D"
and $R_2.x_{sum} - R_1.x_{sum} \geqslant 5$

A common restriction of all three representations is that, in many cases, we cannot use an index for the predicate on the measures (e.g. queries q_1, q_3 and q_5). If the predicates on the measures based on values l, h are highly selective (which is the case when we are looking for outliers) indexing on the path information through indexes on e_{id} or v_{id} will not be sufficient. Querying the dataset is also cumbersome for the user, as she/he has to compose the query appropriately to reflect the part of the sketch that is interested in. Materialized views can be used to accelerate query performance and also ease navigation through the dataset.

For some aggregate function $\mathscr{F}$ let view $\mathscr{V}_{\mathscr{F}}$ compute all pair-wise aggregates $\mathscr{F}_{u\to v}$ for each u and v for which there is a path $u \to v$ in $G(V, E)$: $\mathscr{V}_{\mathscr{F}} = \{u, v, r_{id}, \mathscr{F}(P_{u\to v}(r))\}$. Using the view it is straightforward to express query $l \leqslant \mathscr{F}_{u\to v} \leqslant h$ with selections on columns u and v. In addition to these attributes, indexes on the derived function values $\mathscr{F}(P_{u\to v}(r))$ can be used to accelerate retrieval of matched records. The view requires, asymptotically, $O(|V|^2 * |\mathscr{R}|)$ space, where $|V|$ is the number of nodes in the sketch and $|\mathscr{R}|$ the number of records. The complete pair-wise collection of values in $\mathscr{V}_{\mathscr{F}}$ will probably be prohibitively large to compute and store. In the following sections we show that there is a lot of redundancy in the values of this view that

we use to reduce the space requirements. For referring to appropriate subsets of view $\mathscr{V}_{\mathscr{F}}$ we use the following notation.

Definition 4.5. Given a pair u, v for which there exists a path $u \rightarrow v$ in $G(V,E)$ we define view $\mathscr{V}_{\mathscr{F}_{u\rightarrow v}}$ as the projection of all records in $\mathscr{V}_{\mathscr{F}}$ that contain these states.

5. Processing path queries

For a start we assume that $\mathscr{F} = exist(\,)$, i.e. we are only interested in the transitions stored in the records and not in the actual measures. A *pair-wise path query* $exist_{u\rightarrow v} = 1$ retrieves all records that include a path from u to v.[2] View $\mathscr{V}_{exist_{u\rightarrow v}}$ lists all records that are returned by that query. The view can be stored as a list of record-ids or even better as a (compressed) bitmap of length $|\mathscr{R}|$.

Definition 5.1. $\mathscr{V}_{exist_{u\rightarrow v}}$ is a bitmap of length $|\mathscr{R}|$ with bits set at position i, for every record $r_i \in S_{u\rightarrow v}(\mathscr{R})$.

Given that $\mathscr{V}_{exist_{u\rightarrow v}}$ is materialized, what other queries benefit from this view? The answer to this question is encoded in the graph representation of the sketch. Assume for instance that view $\mathscr{V}_{exist_{C\rightarrow E}}$ is materialized. This view lists all records that contain a path $C \rightarrow E$. Since this path always includes state D we conclude that $S_{C\rightarrow D}(\mathscr{R}) = S_{C\rightarrow E}(\mathscr{R})$ and therefore we can use this view to answer query $exist_{C\rightarrow D}$. For the previous query, materialized view $\mathscr{V}_{exist_{B\rightarrow D}}$ contains a superset of the required records since $S_{C\rightarrow D}(\mathscr{R}) \subseteq S_{B\rightarrow D}(\mathscr{R})$. As a result, querying the view instead of $\mathscr{R}$ results in generating *false positives*, i.e. it will return additional records that do not belong to $S_{C\rightarrow D}(\mathscr{R})$. However, one might want to use the view, as a dirty filter over the dataset if the number of bits set in $\mathscr{V}_{exist_{B\rightarrow D}}$ is much less than $|\mathscr{R}|$, e.g. when lots of events terminate on state F. A third choice arises for query $exist_{D\rightarrow E}$ and view $\mathscr{V}_{exist_{C\rightarrow E}}$. Because of edge (B,D), using the view to answer this query results in false negatives, i.e. dismissal of records that contain the requested transition. An answering mechanism that requires no false dismissals should always avoid the last case.

5.1. Materialized views for pair-wise path queries

We now investigate the problem of selecting the minimum subset of views $\mathscr{V}_{exist_{u\rightarrow v}}$ to be materialized so that subsequent pair-wise path queries can be answered from these views without accessing the dataset. We first define the notion of equivalence among two views.

Definition 5.2. Given two pairs of nodes (x,y) and (a,b) s.t. there is a path from x to y and from a to b in $G(V,E)$ we say that $\mathscr{V}_{exist_{a\rightarrow b}}$ is equivalent ($\equiv$) to view $\mathscr{V}_{exist_{x\rightarrow y}}$ if they contain the same records for any instance of $\mathscr{R}$.

The definition implies that $\mathscr{V}_{exist_{a\rightarrow b}} \equiv \mathscr{V}_{exist_{x\rightarrow y}}$ if each valid record that contains a path $x \rightarrow y$ also contains a path $a \rightarrow b$ and vice-versa. This means that there is a path $x \rightarrow a$ or $a \rightarrow x$ in $G(V,E)$. Assuming that the first is true (otherwise we swap (x,y) and (a,b)) the following condition verifies that x is always included in a record that contains a path $a \rightarrow b$:

1. For all nodes s in the set of starting nodes of $G(V - \{x\}, E - E_x)$,[3] $\nexists$ a path $s \rightarrow a$.

Depending on the relative position of the remaining nodes in the sketch we have the following cases:
Case (1): $\exists$ a path $y \rightarrow a$ in $G(V,E)$. Because the graph is acyclic, this implies that nodes a and b are reached after departing nodes x and y in the specified order. We denote this as: $x \rightarrow y \rightarrow a \rightarrow b$. In this case $\mathscr{V}_{exist_{a\rightarrow b}} \equiv \mathscr{V}_{exist_{x\rightarrow y}}$ if (i) after leaving node y we always pass through a and b and (ii) any path $s \rightarrow a$ includes y. Thus, the following additional constraints must be met:

2. For all nodes t in the set of terminal nodes of $G(V - \{a\}, E - E_a)$, $\nexists$ a path $y \rightarrow t$.
3. For all nodes t in the set of terminal nodes of $G(V - \{b\}, E - E_b)$, $\nexists$ a path $y \rightarrow t$.

[2] Similarly, $exist_{u\rightarrow v} = 0$ retrieves all records without such a transition. When the predicate is omitted, we assume it is =1.

[3] $G(V - \{v\}, E - E_v)$ is the graph obtained if we remove node v from the sketch and all its incident edges, e.g. $E_v = \{(v_1,v_2) \in E \text{ s.t. } v_1 = v \text{ OR } v_2 = v\}$.

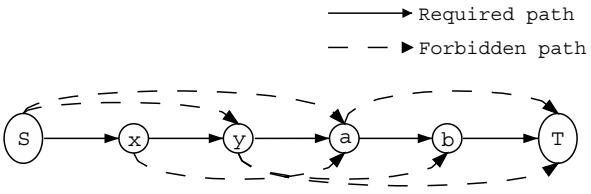


Fig. 6. Required and forbidden paths in a sketch (case 1).

4. For all nodes s in the set of starting nodes of $G(V-\{y\}, E-E_y)$, $\nexists$ a path $s \rightarrow a$.

Fig. 6 depicts the required and forbidden paths in a sketch for case 1 to hold. Super-nodes S and T correspond to all starting and terminal nodes.

Case (2): $\exists$ a path $a \rightarrow y$ in $G(V,E)$. We denote this as: $x \rightarrow a \rightarrow y \rightarrow b$. In this case we verify conditions (1), (3) as well as:

5. $\nexists$ a path $x \rightarrow y$ in $G(V-\{a\}, E-E_a)$.
6. $\nexists$ a path $a \rightarrow b$ in $G(V-\{y\}, E-E_y)$.

Case (3): $\exists$ a path $b \rightarrow y$ in $G(V,E)$. This is denoted as $x \rightarrow a \rightarrow b \rightarrow y$. In this case we test conditions (1), (5) as well as:

7. For all nodes t in the set of terminal nodes of $G(V-\{y\}, E-E_y)$, $\nexists$ a path $b \rightarrow t$.
8. $\nexists$ a path $x \rightarrow y$ in $G(V-\{b\}, E-E_b)$.

These tests require at most $2|S|+2$ Depth-First-Search scans of the graph for every pair (x,y) and (a,b). We stretch here that these tests are only performed once when the sketch is specified. For a sketch with 50 vertices and 100 edges they take 45 seconds in a 600 MHz Pentium III PC.

The $\equiv$ relation partitions the views in equivalent classes $\tilde{\mathscr{V}}_1, \tilde{\mathscr{V}}_2, \ldots$ In the graph of Fig. 3 we have the following four classes:

$$\tilde{\mathscr{V}}_1 = \{\mathscr{V}_{exist_{A\rightarrow C}}, \mathscr{V}_{exist_{B\rightarrow C}}, \mathscr{V}_{exist_{C\rightarrow D}}, \mathscr{V}_{exist_{C\rightarrow E}}\}$$
$$\tilde{\mathscr{V}}_2 = \{\mathscr{V}_{exist_{A\rightarrow F}}, \mathscr{V}_{exist_{B\rightarrow F}}\}$$
$$\tilde{\mathscr{V}}_3 = \{\mathscr{V}_{exist_{A\rightarrow D}}, \mathscr{V}_{exist_{A\rightarrow E}}, \mathscr{V}_{exist_{B\rightarrow D}}, \mathscr{V}_{exist_{B\rightarrow E}}, \mathscr{V}_{exist_{D\rightarrow E}}\}$$
$$\tilde{\mathscr{V}}_4 = \{\mathscr{V}_{exist_{A\rightarrow B}}\}$$

All views belonging to the same class $\tilde{\mathscr{V}}_i$ contain exactly the same bitmap for any instance of $\mathscr{R}$. Thus, only one of these views is needed to be materialized. For a view $\mathscr{V}_{exist_{u\rightarrow v}}$, we denote as $\tilde{\mathscr{V}}_{exist_{u\rightarrow v}}$ the materialized representative of its class. We also denote the number of classes of equivalent views in $G(V,E)$ as $|\tilde{\mathscr{V}}|$. In the graph of Fig. 3, $|\tilde{\mathscr{V}}| = 4$.

In the previous discussion we assumed that there is a path from u to v in $G(V,E)$. If this is not true then $\mathscr{V}_{exist_{u\rightarrow v}}$ is empty by default. These views belong to a virtual class $\tilde{\mathscr{V}}_0$. The representative of this class contains no records. On the opposite side, when transition $u \rightarrow v$ exists in all records for any instance of $\mathscr{R}$ (like $A \rightarrow B$ in our example), the corresponding view indexes the whole dataset and $\tilde{\mathscr{V}}_{exist_{A\rightarrow B}}$ is not materialized. In order to see whether the representative $\tilde{\mathscr{V}}_{exist_{u\rightarrow v}}$ of a class trivially indexes all records in $\mathscr{R}$ the following condition is tested:

9. For all pairs s,t from the sets of starting and terminal nodes of $G(V-\{u,v\}, E-E_u-E_v)$, $\nexists$ path $s \rightarrow t$.

For the sketch of our example just three representative views are needed whose combined size is $3|\mathscr{R}|$ (uncompressed) bits. In practice $|\tilde{\mathscr{V}}|$ depends on the complexity of the sketch. Business processes often have parts with sequential actions $v_1 \rightarrow v_2 \rightarrow \cdots$ As an example processing of a customer's order spawns several processes (possibly on different departments) that are executed in parallel. We model this scenario in the following way: starting with a set of k nodes that are lined in a chain, we pick a random non-terminal node, generate a new branch from that node of length $\leqslant k$ and repeat several times. Fig. 7 shows a possible result for

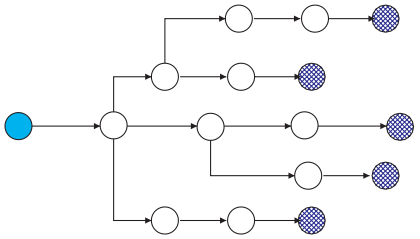


Fig. 7. A service provisioning tree.

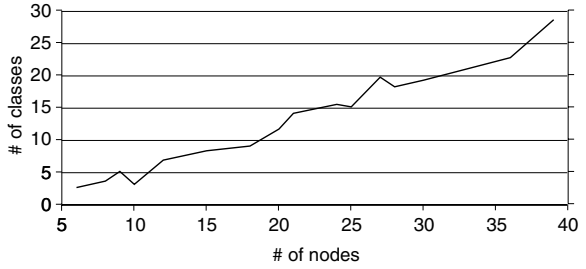


Fig. 8. Number of classes.

$k = 5$ and four iterations. Processes of this form are common in the telephone service provisioning domain. In Fig. 8 we plot the number of classes $|\tilde{\mathscr{V}}|$ versus the number of nodes in a sketch that we generate this way. We varied k between 3 and 6, generated a sample set of 100 graphs and averaged the number of classes for sketches with the same number of nodes. Clearly for this case the number of classes is linear in the number of nodes and the combined size of all views is about the same as the size of a single index on the relation column storing the edge identifiers e_{id} (see Section 4.4). We believe this to be true in many practical cases.

5.2. Evaluating more complex path queries

We now define the selection operator $S_{v_1 \to \cdots \to v_n}(\mathscr{R})$ that returns all records that visit nodes $v_1, \ldots, v_n$ in the specified order. A multi-node path query $exist_{v_1 \to \cdots \to v_n}$ evaluates to 1 for all records in $S_{v_1 \to \cdots \to v_n}(\mathscr{R})$ and is computed as follows:

$n = 1$: We answer the query by ORing the bitmaps of all views $\tilde{\mathscr{V}}_{exist_{s \to v_1}}$, $s \in S$. Alternatively we could use views $\tilde{\mathscr{V}}_{exist_{v_1 \to t}}$, $t \in T$. Overall, we need to OR at-most $min(|S|, |T|, |\tilde{\mathscr{V}}|)$ bitmaps.
$n = 2$: We directly answer the query using view $\tilde{\mathscr{V}}_{exist_{v_1 \to v_2}}$.
$n > 2$: We AND bitmaps of views $\tilde{\mathscr{V}}_{exist_{v_i \to v_{i+1}}}, i = 1, \ldots, n-1$. Up to $min(n-1, |\tilde{\mathscr{V}}|)$ bitmaps are read. This is a crude upper bound as many of these views belong to the same class.

More complex path queries can be expressed as series of multi-node expressions. For example, if we want all records that pass through nodes A, B, D, E but not from C we compute: $exist_{A \to B \to D \to E}$ AND NOT $exist_{C \to C} = \tilde{\mathscr{V}}_{exist_{A \to B}}$ AND $\tilde{\mathscr{V}}_{exist_{B \to D}}$ AND $\tilde{\mathscr{V}}_{exist_{D \to E}}$ AND NOT $\tilde{\mathscr{V}}_{exist_{A \to C}} = \tilde{\mathscr{V}}_{exist_{A \to E}}$ AND NOT $\tilde{\mathscr{V}}_{exist_{A \to C}}$.

A path query can be answered using bit-mapped indices on the nodes ($B(u_i)$) based on the dual representation of a record. There is a direct way to translate the optimized view expression to an optimized expression of bitmaps on the nodes. For the sketch of our running example there are the following four equivalent classes of bitmaps: $B(A) \equiv B(B)$ (with all bits set), $B(C)$, $B(D) \equiv B(E)$ and $B(F)$. The query is now expressed as $B(A)$ AND $B(B)$ AND $B(D)$ AND $B(E)$ AND NOT $B(C) = \tilde{B}(D)$ AND NOT $\tilde{B}(C)$.

While, for a DAG, the translation to the dual representation is straightforward, in the next subsection we demonstrate that when the sketch contains cycles, pair-wise views are more expressive than using bitmapped indexes on the state information.

5.3. Processing path queries in a digraph

A digraph $G(V,E)$ is used as a sketch if each weakly connected component has a non-empty set $S' \in V$ of nodes with no incoming edges and a non-empty set $T' \in V$ of nodes with no out-going edges. The definition of a record is now changed to be an ordered multi-set of (edge,measure) values. Query $\mathscr{F}_{u\to v}$ is defined to aggregate all measures between the first occurrence of node u and the last occurrence of node v in the record.[4]

Since the sketch may contain cycles, we need to add additional constraints in the evaluation of pairs (x,y) and (a,b) for computing the $\equiv$ relation.

Case (1): Constraints (1)–(4) ensure that nodes a and b are visited after nodes x, y in a record. In an acyclic graph this is enough to guarantee a path $x \to y \to a \to b$ in the record. In a digraph we may also see paths: $x \to y \to b \to a$, $y \to x \to a \to b$ and $y \to x \to b \to a$ that should be excluded. For that we add the following two tests:

10. $\nexists$ a path $y \to b$ in $G(V - \{a\}, E - E_a)$.
11. For all nodes s in the set of starting nodes of $G(V - \{x\}, E - E_x)$, $\nexists$ a path $s \to y$.

Case (2): The relative order of the nodes cannot change if all four constraints are met.

Case (3): We need to secure the order of nodes x and y with the following test:

12. $\nexists$ a path $x \to b$ in $G(V - \{a\}, E - E_a)$

When evaluating a multi-node path query as described in Section 5.2 the resulting bitmap describes more records that are actually in $S_{v_1\to\cdots\to v_n}(\mathscr{R})$ as the evaluation process does not guarantee an order between paths $v_i \to v_{i+1}$. For example record $r = \{\ldots,((v_2,v_3),x_{(v_2,v_3)}),\ldots,((v_1,v_2),x_{(v_1,v_2)}),\ldots\}$ contains both a path $v_1 \to v_2$ and $v_2 \to v_3$ but not path $v_1 \to v_2 \to v_3$. Such a record is possible in a (complex) digraph but not in a DAG. Thus, we use the optimized expression as a dirty filter and evaluate the retrieved records at a latter step to eliminate possible false positives.

Example 5.3. For the graph of Fig. 9, the equivalent classes are:

$$\tilde{\mathscr{V}}_1 = \{\mathscr{V}_{exist_{A\to B}}, \mathscr{V}_{exist_{A\to C}}, \mathscr{V}_{exist_{A\to D}}, \mathscr{V}_{exist_{B\to C}}, \mathscr{V}_{exist_{B\to D}}, \mathscr{V}_{exist_{C\to D}}\}$$

$$\tilde{\mathscr{V}}_2 = \{\mathscr{V}_{exist_{A\to E}}, \mathscr{V}_{exist_{B\to E}}, \mathscr{V}_{exist_{C\to E}}, \mathscr{V}_{exist_{D\to E}}\}$$

$$\tilde{\mathscr{V}}_3 = \{\mathscr{V}_{exist_{A\to F}}, \mathscr{V}_{exist_{B\to F}}, \mathscr{V}_{exist_{C\to F}}, \mathscr{V}_{exist_{D\to F}}\}$$

$$\tilde{\mathscr{V}}_4 = \{\mathscr{V}_{exist_{D\to B}}, \mathscr{V}_{exist_{D\to C}}, \mathscr{V}_{exist_{C\to B}}\}$$

The representative of class $\tilde{\mathscr{V}}_1$ is omitted because of condition (9). Suppose that we want to find all records that used the back-edge (D,B) and terminate at node E. The query is formulated as: $exist_{A\to D\to B\to E}$. We compute $\tilde{\mathscr{V}}_{exist_{A\to D}}$ AND $\tilde{\mathscr{V}}_{exist_{D\to B}}$ AND $\tilde{\mathscr{V}}_{exist_{B\to E}} = \tilde{\mathscr{V}}_{exist_{D\to B}}$ AND $\tilde{\mathscr{V}}_{exist_{B\to E}}$. It is easy to see that for the given sketch, the result describes exactly the records that we want, i.e. there are no false positives. Processing requires retrieving exactly 2 bitmaps.

If for the same query we were to use bitmaps $B(u_i)$ on the nodes we get: $B(A)$ AND $B(B)$ AND $B(D)$ AND $B(E) = B(E)$. This expressions describes all records that terminate on node E whether or not they used the back-edge (B,D) and there is no way to express this property using indexes on the nodes. Pair-wise views on the other hand allow us to describe arbitrary transitions between two nodes but evaluation does not guarantee the order of consecutive pairs in the expression. Compared to using bitmapped indices on the nodes, we can show the following.

[4] Other definitions are possible depending on the context.

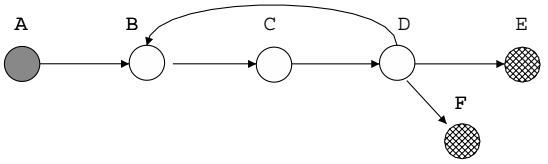


Fig. 9. Sketch with cycles.

Lemma 5.4. *Evaluation of a multi-node path query* $exist_{v_1 \to \cdots \to v_n}$ *in a digraph using bitmap indexes on the nodes* v_i *results in at least as many false positives as the optimized expression of pair-wise path views.*

In many practical scenarios pair-wise views do not add false positives in evaluation of path-expressions over digraphs. In-fact, we can detect all pathological cases by analyzing the sketch in a similar manner. That is, we can tell if there is a need to make a second pass over the records to eliminate false positives by looking at the sketch.

6. Processing pair-wise aggregate queries

We now address the problem of using materialized views for answering pair-wise aggregate queries of the form:

$$l \leqslant \mathscr{F}_{a \to b} \leqslant h \tag{4}$$

when $\mathscr{F} = sum(), count(), max(), min()$. A pair-wise aggregate view $\mathscr{V}_{\mathscr{F}_{x \to y}}$ contains pairs of $(r_{id}, \mathscr{F}(P_{x \to y}(r)))$ values. We can implement this view as a B-tree with the second value used as a key and the first value used to point to the appropriate records in $\mathscr{R}$.

If another view $\mathscr{V}_{\mathscr{F}_{x \to y}}$ with $(x,y) \neq (a,b)$ and $\mathscr{V}_{exist_{x \to y}} \equiv \mathscr{V}_{exist_{a \to b}}$ is materialized it can be used to locate all records with a transition from a to b. This however is inefficient if the numeric predicates on the aggregate are highly selective, e.g. when many records contain a path from a to b but few of them satisfy $l \leqslant \mathscr{F}(P_{a \to b}(r)) \leqslant h$.

Unfortunately, there is no way to relate the aggregates stored in the view with the aggregates requested by the query since pairs (x,y) and (a,b) might be on (connected but) unrelated parts of the sketch. As an example consider query $max_{B \to D} > 100$ that retrieves all records that either have (i) an edge (B,D) with a measure greater than 100 or (ii) edges (B,C) and (C,D) with associated measures that at least one is greater than 100. If view $\mathscr{V}_{max_{D \to E}}$ was materialized it provides filtering on the path expression only, since $\mathscr{V}_{exist_{D \to E}} \equiv \mathscr{V}_{exist_{B \to D}}$. On the other hand, view $\mathscr{V}_{max_{A \to E}}$ provides more leverage since a candidate record r must have $max(P_{A \to E}(r)) > 100$. Therefore, we execute query $max_{A \to E} > 100$ on view $\mathscr{V}_{max_{A \to E}}$ and then check the retrieved records whether $max(P_{B \to D}(r))$ is indeed greater than 100. This process is called query rewriting and is discussed in details in the forthcoming sections.

In general we may be able to exploit the aggregates stored in view $\mathscr{V}_{\mathscr{F}_{x \to y}}$ when evaluating query (4) if nodes a and b are *contained* in a path from $x \to y$. Containment is not a mandatory condition for view equivalence as defined in the previous section. It is described as $x \to a \to b \to y$ in case (3). When all four conditions for this case are met then for distributive aggregate functions, the stored aggregate value along path $x \to y$ can be expressed as[5]:

$$\mathscr{F}(P_{x \to y}(r)) = \mathscr{F}'(\mathscr{F}(P_{x \to a}(r)), \mathscr{F}(P_{a \to b}(r)), \mathscr{F}(P_{b \to y}(r))) \tag{5}$$

where $\mathscr{F}' = \mathscr{F}$ for $\mathscr{F} = max(), min(), sum()$ and $\mathscr{F}' = sum()$ for $\mathscr{F} = count()$. This well-known property of a distributive function is frequently exploited to share computation of data cube aggregates [2].

[5] Function $\mathscr{F}(x_1, \ldots, x_n)$ is distributive if there exists function $\mathscr{G}$ such that $\mathscr{F}(x_1, \ldots, x_n) = \mathscr{G}(F(x_1, \ldots, x_i), F(x_{i+1}, \ldots, x_n))$ for any i: $1 < i < n$. The definition implies that the input of $\mathscr{F}$ can be partitioned into an arbitrary number of sets (sub-problems) and the result of the function can always be computed by composing (using $\mathscr{G}$) the results of the sub-problems without any "state" information on how partitioning were obtained.

For any two pairs of nodes (x,y) and (a,b) for which the conditions of case (3) are met we denote that $\mathscr{V}_{\mathscr{F}_{a\to b}} \preceq \mathscr{V}_{\mathscr{F}_{x\to y}}$. The $\preceq$ relation is reflexive, transitive and antisymmetric. Thus, it defines a partial order on the views. The antisymmetry comes from the containment requirement. For the sketch of Fig. 3 the $\preceq$ relation is shown in Fig. 10. A node in the figure represents an aggregate view on the corresponding pairs. An arrow between two views depicts that the pointed view $\preceq$ the other. We also include the whole dataset $\mathscr{R}$ to depict that all these views can be computed from the raw records.

For a view $\mathscr{V}_{\mathscr{F}_{u\to v}}$ the top-level ancestor, i.e. the higher view $\mathscr{V}$ in the hierarchy s.t. $\mathscr{V}_{\mathscr{F}_{u\to v}} \preceq \mathscr{V}$ is denoted as $\hat{\mathscr{V}}_{\mathscr{F}_{u\to v}}$. For example $\hat{\mathscr{V}}_{\mathscr{F}_{D\to E}} = \mathscr{V}_{\mathscr{F}_{A\to E}}$. The number of top-level views is denoted as $|\hat{\mathscr{V}}|$, and is 5 in our example.

6.1. Weaker condition

When view $\mathscr{V}_{\mathscr{F}_{x\to y}}$ such that $\mathscr{V}_{\mathscr{F}_{a\to b}} \preceq \mathscr{V}_{\mathscr{F}_{x\to y}}$ is used for query (4), it retrieves all records with a transition $a \to b$ with some additional filtering based on a rewriting for the measures along $x \to y$. The details of this rewriting are included in the forthcoming sections. The rewriting introduces false positives (but no false negatives) for the aggregate values but is exact on the path requirement: all records retrieved from view $\mathscr{V}_{\mathscr{F}_{x\to y}}$ contain a path $a \to b$.

We can relax the path equivalence requirement at the expense of getting more false positives. We define that $\mathscr{V}_{\mathscr{F}_{a\to b}} \prec \mathscr{V}_{\mathscr{F}_{x\to y}}$ if (i) x is required to reach a and (ii) any path from b to a terminal node includes y. Thus, only tests (1) and (7) are required.

The $\prec$ relation is also a partial order and implies that $S_{a\to b}(\mathscr{R}) \subseteq S_{x\to y}(\mathscr{R})$. For the sketch of Fig. 3 the $\prec$ partial order is depicted in Fig. 11. The top-level views of the order are denoted as $\dot{\mathscr{V}}$ and their number as $|\dot{\mathscr{V}}|$. In this example $|\dot{\mathscr{V}}| = 3$.

Based of the definition of relations $\equiv$, $\preceq$ and $\prec$ the following observations are made:

- $\mathscr{V}_{\mathscr{F}_{a\to b}} \preceq \mathscr{V}_{\mathscr{F}_{x\to y}}$ implies that $\mathscr{V}_{exist_{a\to b}} \equiv \mathscr{V}_{exist_{x\to y}}$ and therefore $|\tilde{\mathscr{V}}| \leqslant |\hat{\mathscr{V}}|$.
- Views $\hat{\mathscr{V}}_{\mathscr{F}_{a\to b}}$ answer exactly path queries in a DAG and with possible false positives in a digraph.

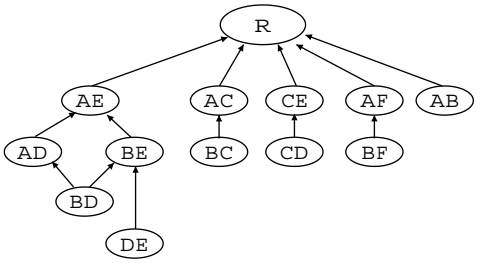


Fig. 10. The $\preceq$ partial order.

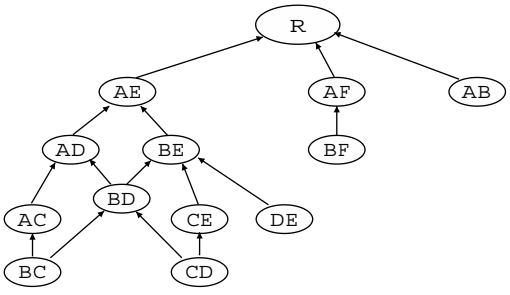


Fig. 11. The $\prec$ partial order.

- $\mathscr{V}_{\mathscr{F}_{a\to b}} \preceq \mathscr{V}_{\mathscr{F}_{x\to y}}$ implies that $\mathscr{V}_{\mathscr{F}_{a\to b}} \prec \mathscr{V}_{\mathscr{F}_{x\to y}}$ and therefore $|\dot{\mathscr{V}}| \leqslant |\hat{\mathscr{V}}|$.
- Views $\dot{\mathscr{V}}_{\mathscr{F}_{a\to b}}$ is the smallest set of pair-wise views that answer any pair-wise path/aggregate query with no false negatives without looking at the data. However, these views may introduce false positives in path expressions both in a DAG and a digraph.

6.2. Pair-wise sum-queries

Pair-wise sum-queries are queries of the form: $l \leqslant sum_{a\to b} \leqslant h$. We assume that at least the top-level views $\hat{\mathscr{V}}_{sum_{u\to v}}$ of the $\preceq$ hierarchy of Fig. 10 are materialized and we explore how queries on the remaining pairs can be translated and executed efficiently using these views.

If view $\mathscr{V}_{sum_{a\to b}}$ is not computed we are using materialized view $\mathscr{V}_{sum_{x\to y}} = \hat{\mathscr{V}}_{sum_{a\to b}}$ as a dirty filter to find candidate records that we retrieve and evaluate from $\mathscr{R}$ in a latter step. Along with the views, we store in the database the dependency graph of Fig. 10. This graph has a size of $O(|V|^2)$, which we consider insignificant in a data warehouse context. For each node in the graph we maintain the minimum and maximum value of the $sum()$ function evaluated over the stored records that contain the specified transition. For instance, node DE will store the following two numbers: $min\_sum_{D\to E} = min(sum(P_{D\to E}(r)))$ and $max\_sum_{D\to E} = max(sum(P_{D\to E}(r)))$ for all $r \in S_{D\to E}(\mathscr{R})$. These statistics are easy to maintain in an append-only scenario, while we load new records in the data warehouse. In fact, many service provisioning datasets are obtained from recording tools and data is indeed append only. Using Eq. (5), query $l \leqslant sum_{a\to b} \leqslant h$ is re-written as[6]:

$$l' = l + min\_sum_{x\to a} + min\_sum_{b\to y} \leqslant sum_{x\to y} \leqslant h + max\_sum_{x\to a} + max\_sum_{b\to y} = h' \tag{6}$$

We therefore query view $\mathscr{V}_{sum_{x\to y}}$ using constants l' and h' and get a list of r_{id}s that satisfy formula (6). Each candidate record r is retrieved from $\mathscr{R}$ in order to evaluate whether $l \leqslant sum(P_{a\to b}(r)) \leqslant h$. Because of the $\preceq$ relation using the view is equivalent for the path-requirement $a \to b$ and is therefore at least as good as using any type of indices on the node/edge-ids of the records.

Lemma 6.1. *A query rewrite of formula (6) results in accessing no-more records than what is required because of the path-expression $a \to b$.*

We now calculate the number of false positives introduced by the rewriting of formula (6). The probability of a record being a false positive is the conditional probability:

$$Prob_{false\_positive} = Prob(sum(P_{a\to b}(r)) \notin [l,h] | sum(P_{x\to y}(r)) \in [l',h'])$$

If the distribution of values of the stored measures are known, one can analytically compute the above probability for any combination of pairs a,b and x,y. As an exercise, we here consider the following example. Assume that the values collected along edges (C,D) and (D,E) form two independent uniform random variables X and Y respectively. Assume that view $\mathscr{V}_{sum_{C\to E}}$ is materialized and used for evaluating query: $a \leqslant sum_{C\to D} \leqslant 1$, i.e. the query requests all records with $X \geqslant a$ (with $a \leqslant 1$). Using formula (6) we retrieve all records with $a + 0 \leqslant X + Y \leqslant 1 + 1$, since $min\_sum_{D\to E} = min(Y) = 0$, $max\_sum_{D\to E} = max(Y) = 1$. The probability of getting a false positive is then:

$$Prob_{false\_positive} = Prob(X < a | X + Y \geqslant a) = \frac{Prob(X < a \wedge X + Y \geqslant a)}{Prob(X + Y \geqslant a)}$$

This probability is computed based on Fig. 12. All records retrieved from the rewriting $X + Y \geqslant a$ are above line DC. Thus, we must compute the integral of the join-density distribution for the trapezoid $ABCD$ over the integral over the whole area except the lower triangle OCD. For independent uniform random variables the integrals simply compute the respective areas:

$$Area_{ABCD} = \frac{(1+1-a)*a}{2}, Area_{OCD} = \frac{a^2}{2} \Rightarrow Prob_{false\_positive} = \frac{(2-a)a}{2-a^2}$$

[6] we assume that $min\_sum_{u\to v} = max\_sum_{u\to v} = 0$ if $u = v$.

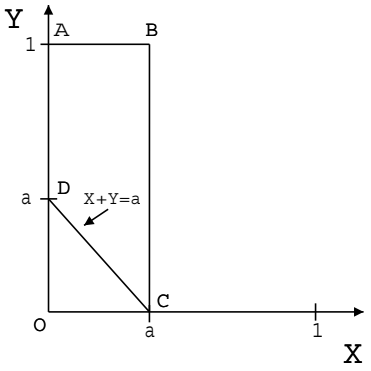


Fig. 12. Computation of false positives.

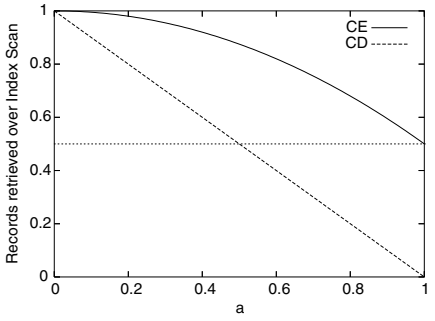


Fig. 13. Records retrieved over index scans.

In practice we evaluate query speed-up when using the view in comparison with the use on an index I that retrieves all records that contain an edge (C, D). Let $|I|$ be the number of records retrieved using this index. The number of records retrieved from view $\mathscr{V}_{sum_{C\to E}}$ is expected to be: $|\mathscr{V}_{CE}| = (1 - Area_{OCD})|I| = \frac{2-a^2}{2}|I|$. Similarly, the number of records contained in the query result that would have retrieved from view $\mathscr{V}_{sum_{C\to D}}$ if it were materialized is: $|\mathscr{V}_{CD}| = (1 - a)|I|$. Fig. 13 plots these values with respect to a. The graph demonstrates that using view $\mathscr{V}_{sum_{C\to E}}$ is always better than using an index, as expected because of Lemma 6.1. The maximum speed-up that we obtain compared to the index is 2-1 since at least $|I|/2$ records are accessed. In the following subsection we show how to obtain even better speed-ups.

6.2.1. Using phantom aggregates to speed-up queries on non-materialized views

We now show how to calibrate indexing of the records of a materialized view $\mathscr{V}$ to achieve better performance when querying for another view $\mathscr{V}' \preceq \mathscr{V}$ that is not materialized.

Each top-level view $\mathscr{V}$ of Fig. 10 defines a poset $D_{\mathscr{V}}$ with all views $\mathscr{V}' \preceq \mathscr{V}$. Let $\mathscr{L}_{\mathscr{V}}$ be the set of lower bounds in $D_{\mathscr{V}}$. For example $\mathscr{L}_{\mathscr{V}_{sum_{A\to E}}} = \{\mathscr{V}_{sum_{B\to D}}, \mathscr{V}_{sum_{D\to E}}\}$.

Lemma 6.2. *For each top-level view $\mathscr{V}$ in the $\preceq$ order, $\mathscr{L}_{\mathscr{V}}$ contains views on non-overlapping paths in $G(V, E)$ (with the exception of the end-points).*

Proof. Let $\mathscr{V}_{sum_{a_1\to b_1}}$, $\mathscr{V}_{sum_{a_2\to b_2}}$ be two lower bounds in $D_{\mathscr{V}}$. We assume that paths $a_1 \to b_1$ and $a_2 \to b_2$ overlap and without loss of generality that the arrangement of the nodes is $a_1 \to a_2 \to b_1 \to b_2$, otherwise we switch notation.[7]

[7] The case where $a_1 \to a_2 \to b_2 \to b_1$ is not permissible because of the assumption that $\mathscr{V}_{sum_{a_1\to b_1}}$ is a lower-bound in $D_{\mathscr{V}}$.

Because of conditions (1), (5), (7) and (8) of case (3) for $a_1 \rightarrow b_1$ and $a_2 \rightarrow b_2$, we conclude that $\mathscr{V}_{sum_{a_2 \rightarrow b_1}} \preceq \mathscr{V}$; that is $\mathscr{V}_{sum_{a_2 \rightarrow b_1}} \in D_{\mathscr{V}}$. With similar arguments we have that $\mathscr{V}_{sum_{a_2 \rightarrow b_1}} \preceq \mathscr{V}_{sum_{a_1 \rightarrow b_1}}$. The latter contradicts the assumption that $\mathscr{V}_{sum_{a_1 \rightarrow b_1}}$ is a lower bound in $D_{\mathscr{V}}$, unless $a_1 = a_2$. However, in case ($a_1 = a_2$) then $\mathscr{V}_{sum_{a_1 \rightarrow b_1}} \preceq \mathscr{V}_{sum_{a_2 \rightarrow b_2}}$, which again contradicts the assumption that $\mathscr{V}_{sum_{a_2 \rightarrow b_2}}$ is a lower bound, unless $a_1 = a_2$ and $b_1 = b_2$. That is, the two paths cannot overlap, unless we are talking about the same pair. □

Our key-idea is to use the values of the views in $\mathscr{L}_{\mathscr{V}}$ to cluster the records of $\mathscr{V}$ in a way that will allow fewer false positives due to the rewriting. We propose two methods for storing the view based on partitioning its records on the values of the aggregates in $\mathscr{L}_{\mathscr{V}}$. We call these values *phantom aggregates* as they do not appear in $\mathscr{V}$. Both methods maintain a hybrid data-structure, in which the upper part describes a partitioning scheme based on the phantom aggregates and the lower part implements a collection of B-trees on the values of $\mathscr{V}$, using one tree per partition. The upper partitioning scheme is fixed, i.e. we make no attempt to modify it during updates. This is not a problem as phantom aggregates have a suggestive value during query rewriting. In practice we can periodically modify the partitions when the dataset or the query workload change.

kd-tree method. For some small value B, we generate B partitions for the values of $V = V_{sum_{u \rightarrow v}}$ in the following manner: we treat each value of V as an $|L_V|$-dimensional point with coordinates defined by the phantom *sum*( ) aggregates of views in L_V. We then build the first $\log_{|L_V|}(B)$ levels of a kd-tree for these values. The kd-tree is an extension of a binary search tree in more dimensions. The main difference is that levels of the tree are split along successive dimensions, i.e level 0 is split on the first dimension, level 1 on the second etc. More details on the kd-tree can be found in [20]. In our framework, each node at level $\log_{|L_V|}(B)$ contains a pointer to a partition of the original values in V with all records whose phantom aggregates fall in the sub-space specified by the path from the root to that node in the kd-tree. Each partition is organized as a B-tree having $sum(P_{u \rightarrow v(r)})$ as the key and the corresponding r_{ids} as values. At the root of each tree we store the *min_sum*( ) and *max_sum*( ) values for the partition, for each phantom aggregate.

Querying this structure for any view $\mathscr{V}' \preceq \mathscr{V}$ is done using the top-level kd-tree nodes for pruning the search. Queries on values of $\mathscr{V}$ ignore these levels and access all the underlying B-trees. In the presence of multiple disks, all these trees can be efficiently searched in parallel. The space requirement for the first $\log_{|\mathscr{L}_{\mathscr{V}'}|}(B)$ levels of the kd-tree is $2 * (B - 1)$, which fits in a single data page for small Bs.

Example 6.3. We demonstrate the creation of the hybrid structure of Fig. 14 for view $\mathscr{V}_{sum_{A \rightarrow E}}$. Each record of this view is a pair of $(r_{id}, sum(P_{A \rightarrow E}(r)))$ values. We first attach the phantom aggregates of views $\mathscr{L}_{\mathscr{V}_{sum_{A \rightarrow E}}} = \{\mathscr{V}_{sum_{B \rightarrow D}}, \mathscr{V}_{sum_{D \rightarrow E}}\}$ and generate a temporary table $\mathscr{V}_{temp}$ with attributes: r_{id}, $sum(P_{A \rightarrow E}(r))$, $sum(P_{B \rightarrow D}(r))$, $sum(P_{D \rightarrow E}(r))$. We then scan $\mathscr{V}_{temp}$ and generate a kd-tree on the $sum(P_{B \rightarrow D}(r))$ and $sum(P_{D \rightarrow E}(r))$ phantom aggregates. During this phase we also compute $min\_sum_{B \rightarrow D}$, $max\_sum_{B \rightarrow D}$, $min\_sum_{D \rightarrow E}$, $max\_sum_{D \rightarrow E}$. For $B = 16$, we keep the first $\log_{|\mathscr{L}_{\mathscr{V}_{sum_{A \rightarrow E}}}|=2}(B) = 4$ levels of the tree and use them to partition $\mathscr{V}_{temp}$ in a second scan. Each partition projects the first two attributes of $\mathscr{V}_{temp}$ and is implemented as a separate B-tree using $sum(P_{A \rightarrow E})$ as the key and r_{id} as the values indexed. Nodes of the kd-tree that split the records on the phantom

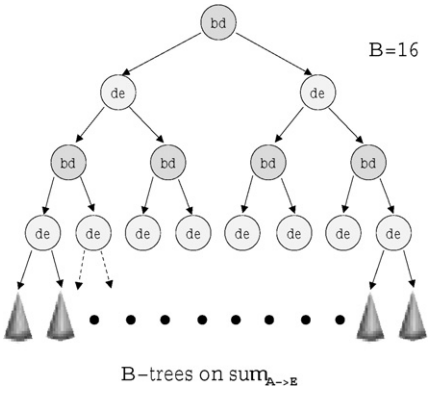


Fig. 14. Implementation of view $\mathscr{V}_{sum_{A \rightarrow E}}$.

$sum(P_{B\to D}(r))$ aggregates are used to prune the search space when evaluating queries: $sum_{B\to D}$, $sum_{B\to E}$ and $sum_{A\to D}$, while nodes that split on $sum(P_{D\to E}(r))$ aggregates are used during $sum_{B\to E}$ and $sum_{D\to E}$ searches.

Grid-based method. We create a hybrid data-structure, which partitions the records of the view by superimposing a $|\mathscr{L}_{\mathscr{V}}|$-dimensional grid on its values. The simplest way to achieve this is to partition the aggregates of each view $\mathscr{V}' \in \mathscr{L}_{\mathscr{V}}$ by computing appropriate quantiles [19,37,23,25]. Multi-dimensional index loading techniques like [15] are also applicable. Notice that the goal is not to equi-split the tuples of the view, but to impose a partitioning scheme that will benefit the expected workload on the phantom aggregates. After the grid is decided, a single B-tree on the records of $\mathscr{V}$ is built for each cell. Storage requirements for the grid is $|\mathscr{L}_{\mathscr{V}}| * (B^{\frac{1}{|\mathscr{L}_{\mathscr{V}}|}} - 1) + B \leqslant 2 * B$, including the pointers to the underlying B-trees.

Example 6.4. View $\mathscr{V}_{sum_{C\to E}}$ is used for queries $sum_{C\to E}$ as well as queries $sum_{C\to D}$, when the latter one is not materialized as a view. To create the grid-based hybrid-structure, we partition the records of the view based on the phantom $sum_{C\to D}$ values (which we know at creation time) into $B \geqslant 2$ sets. This is achieved using known statistics on the values x_{CD} along edge (C, D) to split the records in a way that better suits the expected $sum_{C\to D}$ queries. Fig. 15 shows an example where $B = 4$. The left sub-tree contains record-ids indexed by $sum_{C\to E}$ values, for which the phantom x_{CD} measure (that is not stored in the structure) is less or equal to cd_1. The second tree from the left has records with x_{CD} values in the range $(cd_1, cd_2]$, while the last tree contains records with x_{CD} values greater than cd_3. In the header-page of each B-tree we also store the minimum and maximum values along edge (D, E) for the corresponding partition, which again are known when the values are loaded.

Given a query $l \leqslant sum_{C\to D} \leqslant h$, we access the corresponding tree for each partition that intersects the $[l, h]$ range. Each one of these trees we query using the re-writing of formula (6) where $min\_sum_{D\to E}$ and $max\_sum_{D\to E}$ are defined per tree. In case l (resp. h) is lower (resp. higher) than the values of the grid-points of the partition they are substituted accordingly.

6.3. Pair-wise count-queries

For evaluating query $l \leqslant count_{a\to b} \leqslant h$ we first list all possible paths $p_1, \ldots, p_k$ from a to b in $G(V, E)$ with the appropriate number of transitions. As described in Section 5.2 we can find all records that contain path p_i by accessing up to $|\tilde{\mathscr{V}}|$ appropriate bitmaps. This results in a bitmap B_i for each path. The answer to the query is then computed by ORing all bitmaps B_i. This method requires no access to the dataset $\mathscr{R}$ and generates no false positives/dismissals. The method is applicable when views $\hat{\mathscr{V}}_{\mathscr{F}_{a\to b}}$ (for any function $\mathscr{F}$) are available. The answer is similarly computed by merging appropriate lists of records ids. If views $\dot{\mathscr{V}}_{\mathscr{F}_{a\to b}}$ are used then the answer may include false positives.

6.4. Pair-wise min/max-queries

Pair-wise max/min queries are treated similarly to the pair-wise sum queries. The rewriting in this case will be:

$$l' = \mathscr{F}(l, min\_\mathscr{F}_{x\to a}, min\_\mathscr{F}_{b\to y}) \leqslant \mathscr{F}_{x\to y} \leqslant \mathscr{F}(h, max\_\mathscr{F}_{x\to a}, max\_\mathscr{F}_{b\to y}) = h'$$

for $\mathscr{F} = min()/max()$.

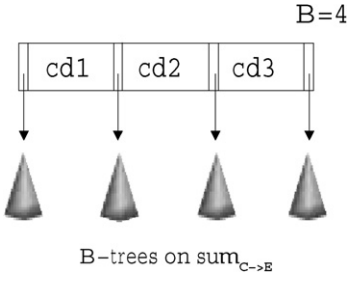


Fig. 15. Implementation of view $\mathscr{V}_{sum_{C\to E}}$.

6.5. Handling parallel paths

In business workflows it is common to have multiple sub-tasks that are spawn from a node. Consider for example the sketch of Fig. 16. State A spawns two parallel processes that are being synchronized later at state D. An event that leaves state A collects measures x_i along all four edges (A,B), (B,D), (A,C) and (C,D). In order for our framework to apply for this case we need to define the meaning of an aggregation for pair (A,D) (similar to *path-aggregation* in [44]). If timing information is of use then $sum(P_{A\to D}(r))$ can be defined as $max(x_1+x_2, x_3+x_4)$. If on the other hand some cost-related weights are stored in the edges we may define $sum(P_{A\to D}(r))$ to be $x_1+x_2+x_3+x_4$. As long as we provide a succinct way to describe theses aggregations, the same framework is directly applicable for this data.

7. Experiments

7.1. Evaluate rewriting using the $\preceq$ partial order

In this set of experiments we focus on a specific portion of a sketch that contains $n+1$ nodes: $v_1,\ldots,v_{n+1}$ forming a chain (Fig. 17). Business workflows frequently contain parts with such local sequential actions. We denote as x_i the measure value collected on edge (v_i, v_{i+1}). Each measure describes arrival times that are following an independent exponential distribution. We assume that $\mathscr{V}_{sum_{v_1\to v_{n+1}}}$ is the only view materialized.

The first two experiments evaluate how efficient the view can be for answering pair-wise sum queries of the following two practical classes:

- **queries for outliers:** these are queries of the form:

$$sum_{v_i\to v_{j+1}} \geqslant E[x_i+\cdots+x_j] + kDEV[x_i+\cdots+x_j] \tag{7}$$

this query can be stated as: "find all records where a transition from node i to node $j+1$ took more that $\mu + k * s$", where μ, s are the expected time and standard deviation for transitions between these nodes and k is a user defined parameter that describes how selective the query is. Conceptually, this type of queries access "sparse" areas of the n-dimensional space formed by the measures, looking for outliers.

- **queries on hot-spots:** these are queries of the form:

$$E[x_i+\cdots x_j] - kDEV[x_i+\cdots x_j] \leqslant sum_{v_i\to v_{j+1}} \leqslant E[x_i+\cdots x_j] + kDEV[x_i+\cdots x_j] \tag{8}$$

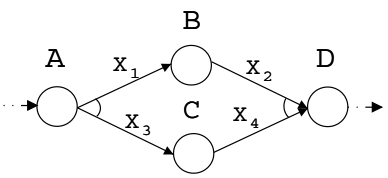


Fig. 16. Sketch with parallel parts.

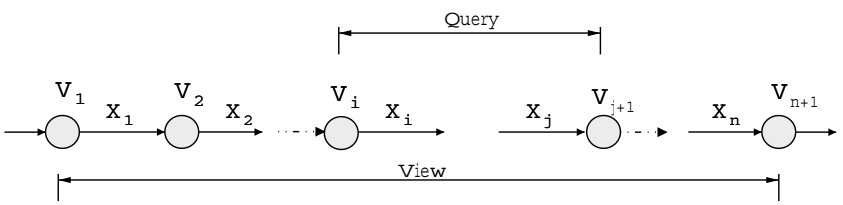


Fig. 17. Chain of n nodes.

These queries ask for aggregates close to the expected value of the combined distribution and evaluate the rewriting when querying a "dense" area of the data.

For the first experiment we varied n and tested querying view $\mathscr{V}_{sum_{v_1 \to v_{n+1}}}$ for computing outliers on the first two transitions: $sum_{v_1 \to v_3} = x_1 + x_2 \geqslant E[x_1 + x_2] + kDEV[x_1 + x_2]$. We used a synthetic dataset of 1,000,000 records with a transition $v_1 \to \cdots \to v_{n+1}$. In Fig. 18 we report the number of records returned from the view varying parameters k and n. The flat line represents the number of records returned if instead of the view we do index look-ups in $\mathscr{R}$ for the two edges. This number is constant as all 1,000,000 records qualify. Two observations are made from this graph: (i) performance of the view degrades with the length of the path due to "noise" from measures $x_3, \ldots, x_n$ and (ii) performance gets better when we are looking for extreme outliers (e.g. as k increases).

In Fig. 19 we experiment with queries on hot-spots using as an example query $E[x_1] - kDEV[x_1] \leqslant sum_{v_1 \to v_2} \leqslant E[x_1] + kDEV[x_1]$, varying k from 0.01 to 0.30 and n from 1 (exact view) to 6. For these queries,

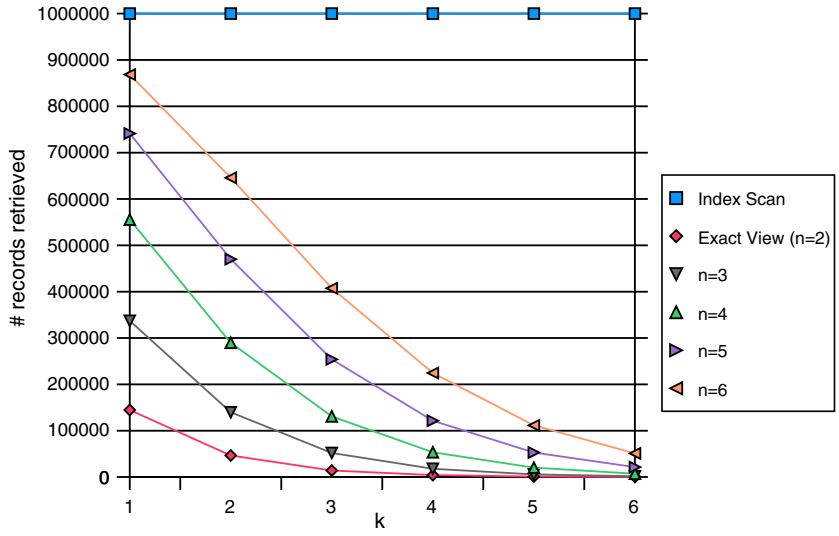


Fig. 18. Querying for outliers.

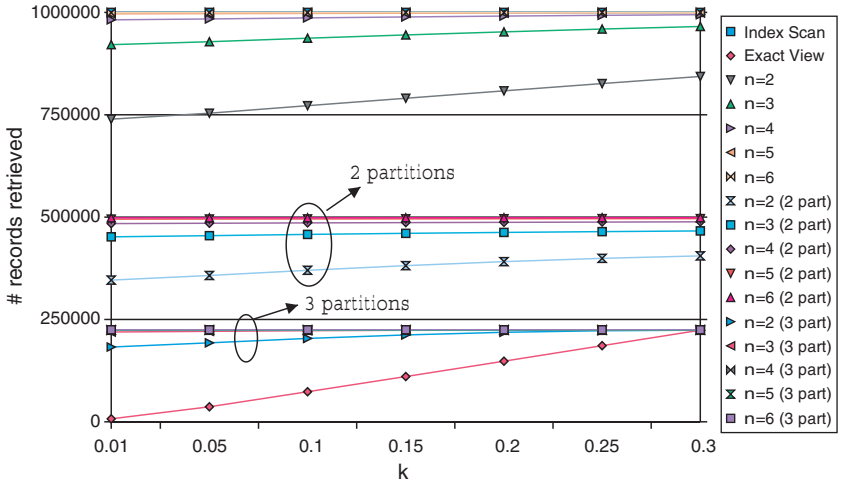


Fig. 19. Querying on hot-spots.

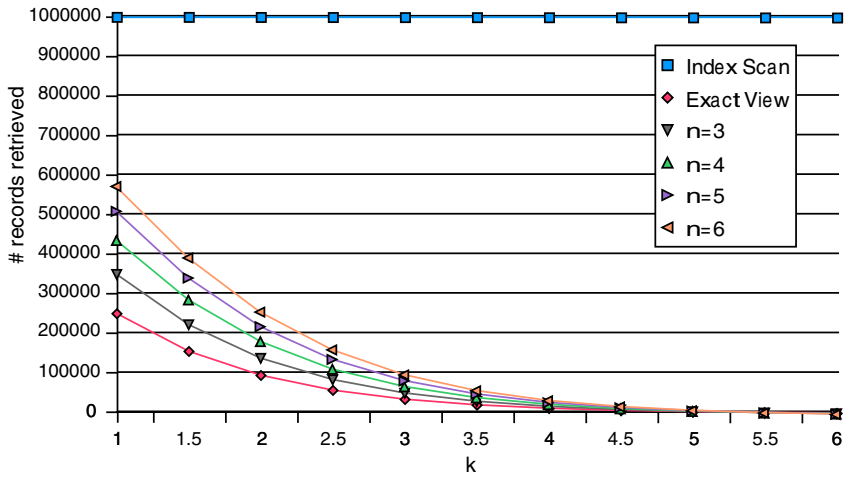


Fig. 20. Pair-wise max-queries.

view $\mathscr{V}_{sum_{v_1 \to v_{n+1}}}$ is not as effective as when querying for outliers. For $n > 2$, the view is about as bad as using the index. In a second run, we used the grid-based index of Section 6.2.1 with *2* and *3* partitions per phantom aggregate $x_2, \ldots, x_n$. For the first case we partitioned on the expected median of each x_i, that is 0.693 for this distribution, and for the latter the break-points where set to 0.7 and 1.3 to better reflect the query pattern. The size of view $\mathscr{V}_{sum_{v_1 \to v_{n+1}}}$ was 11.61 MB when stored as a single B-tree, 11.73 MB when stored using a $2 \times 2 \times 2 \times 2 \times 2 \times 2$ grid and 14.31 MB for the $3 \times 3 \times 3 \times 3 \times 3 \times 3$ grid.[8] This is comparable to the size of a B-tree on e_{id} (11.6 MB).

In the next experiment we evaluate processing of pair-wise max queries. Assume that we would like to find all records for which at least one transition between states v_1 and v_3 was delayed. Thus, we want to retrieve all records with an instance of x_1 or x_2 that was much higher than the expected value $E[x_i] = a$. The query is stated as: query $max_{v_1 \to v_3} \geqslant E[x_i] + kDEV[x_i]$. Notice how this is different than query $sum_{v_1 \to v_3} \geqslant E[x_1 + x_2] + kDEV[x_1 + x_2]$ that retrieves records for which the overall transition was delayed. We evaluated the query using view $\mathscr{V}_{max_{v_1 \to v_{n+1}}}$, varying n from 2 (exact case) to 6. Fig. 20 plots the number of records retrieved. Again performance gets better when we search for extreme outliers (higher value of k).

7.2. Evaluating rewriting using the $\prec$ partial order

We now modify the initial sketch by adding a *cross-over* edge (v_1, v_{n+1}). As a result now $\mathscr{V}_{sum_{v_i \to v_{j+1}}} \prec \mathscr{V}_{sum_{v_1 \to v_{n+1}}}$. We generated 1,000,000 new records varying the probability P of a record using the edge (v_1, v_{n+1}). The measure along the new edge was following the same exponential distribution. We executed query $sum_{v_1 \to v_3} \geqslant E[x_1 + x_2] + 6 * DEV[x_1 + x_2]$ (for $n = 6$) using the view or an index on e_{id}s as shown in Fig. 21. For $P = 0$ no record uses the cross-over path and the index returns all records. With P increasing the index scan becomes more selective but the same is happening for the view.

7.3. Experiment with real data

For the next experiment we used real traces from a business workflow with 6 states, forming a chain. The dataset had 42,754 records. Measures x_i record timing information. In Fig. 22 we normalize the number of records retrieved from view $\mathscr{V}_{sum_{v_1 \to v_6}}$, those retrieved from an index on e_{id} as well as the exact answer size, over the dataset size for all possible queries of formula (7) with $i \in [1, 5]$, $j \in [i, 5]$ and $k = 4$. The x-axis in the figure represents pairs i,j. For the view we used a grid with 2 partitions per measure. We experimented with two

[8] We also tested a grid of the form $3 \times 3 \times 1 \times 1 \times 1 \times 1$ that was slightly worse on queries on x_1 but used only 11.62 MB of disk space.

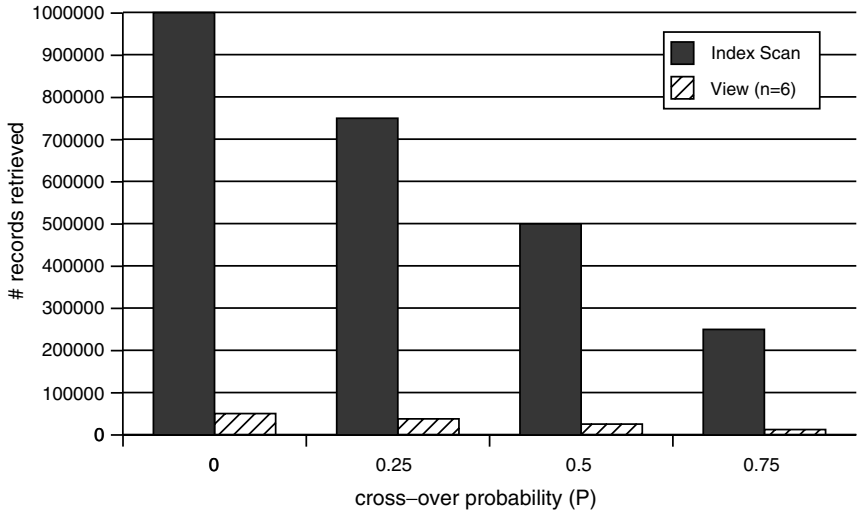


Fig. 21. Testing the $\prec$ partial order.

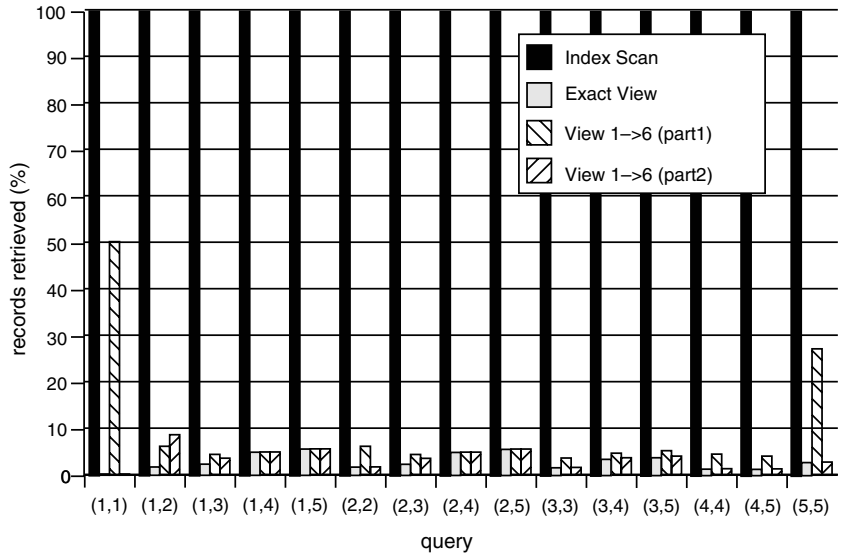


Fig. 22. Querying for outliers, real data.

setups for the grid: the first indicated as *part-1* used as break-points the average value of each measure, while for *part-2* we used value $E[x_i] + k * DEV[x_i]$. *part-2* did better in most cases, as expected, but not always as the optimal break-point per measure is not necessarily optimal for multi-measure queries (e.g. when $j > i$). Overall, the view in all but 2 cases managed to filter-out more than 90% of the dataset. Its size was 652KB while the size of the B-tree index on e_{id}s 521KB.

8. Conclusions

In this paper we showed how to apply data warehousing techniques for organizing service provisioning data. Our solution is based on the notion of a sketch as an abstract description of the underlying process that generates the records. We explored the implications of storing, indexing and querying this data in a relational engine. We introduced pair-wise queries, a new class of aggregate queries that consolidate data from a part of

the process, based on a user defined path expression, and explored their use for multi-resolution analysis of a complex process through appropriate navigation operations.

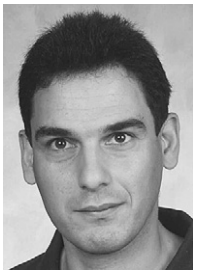
We further explored the use of materialized views for speeding up frequent computations. We first showed how to select a minimum set of views to answer any pair-wise path query and then how to optimize the evaluation of more complex path-expression over the given sketch from the views. For pair-wise queries with aggregations, we defined two partial orders among the views: $\prec$ is used to find the minimum set of aggregate views to answer any query with no false dismissals while $\preceq$ describes an augmented set that allows false positives on the aggregates but not on the path requirement. Computing a non-materialized aggregate is done through appropriate rewriting of the user query. We described two hybrid-indexing schemes that use phantom aggregate values and allow us to efficiently query the view, even for non-materialized aggregates. Experimental results show these schemes to perform well on the synthetic and real datasets that we used.

While our techniques have been motivated by the service provisioning scenario, our framework can be applied to the more general problem of analyzing workflow log records generated by process management software. Post-analysis of such records can benefit from the class of aggregate queries we introduced in this paper and our indexing and view selection techniques can further help in managing the large collections of record logs generated by such systems.

References

[1] R. Agrawal, D. Gunopulos, F. Leymann, Mining process models from workflow logs, in: Proceedings of International Conference on Extending Database Technology (EDBT), March 1998, pp. 469–483.
[2] S. Agrawal, R. Agrawal, P.M. Deshpande, A. Gupta, J.F. Naughton, R. Ramakrishnan, S. Sarawagi, On the Computation of Multidimensional Aggregates, in: Proceedings of the 22nd VLDB conference, Bombay, India, August 1996, pp. 506–521.
[3] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Günthör, C. Mohan, Advanced transaction models in workflow contexts, in: Proceedings of the Twelfth International Conference on Data Engineering, February 1996, pp. 574–581.
[4] E. Baralis, S. Paraboschi, E. Teniente, Materialized view selection in a multidimensional database, in: Proceedings of the 23th International Conference on VLDB, Athens, Greece, August 1997, pp. 156–165.
[5] C. Beeri, P.A. Bernstein, N. Goodman, A model for concurrency in nested transactions systems, J. ACM 36 (2) (1989) 230–269.
[6] C. Bettini, X.S. Wang, S. Jajodia, Free schedules for free agents in workflow systems, in: Proceedings of the TIME, Nova Scotia, Canada, July 2000, pp. 31–38.
[7] A. Biliris, S. Dar, N.H. Gehani, H.V. Jagadish, K. Ramamritham, Asset: a system for supporting extended transactions, in: Proceedings of ACM SIGMOD, May 1994, pp. 44–54.
[8] A. Bonifati, F. Casati, U. Dayal, M.-C. Shan, Warehousing workflow data: challenges and opportunities, in: Proceedings of VLDB, Rome, Italy, 2001, pp. 649–652.
[9] O.A. Bukhres, A.K. Elmagarmid, E. Kühn, Implementation of the flex transaction model, IEEE Data Eng. Bull. 16 (2) (1993) 28–32.
[10] C.Y. Chan, Y. Ioannidis, Bitmap index design and evaluation, in: Proceedings of ACM SIGMOD International Conference on Management of Data, Seattle, Washington, USA, June 1998, pp. 355–366.
[11] S. Chaudhuri, U. Dayal, An overview of data warehousing and OLAP technology, SIGMOD Record 26 (1) (1997).
[12] P. Pin-Shan Chen, The entity-relationship model—towards a unified view of data, ACM Trans. Database Syst. (TODS) 1 (1) (1976) 9–36.
[13] P.K. Chrysanthis, K. Ramamritham, ACTA: a framework for specifying and reasoning about transaction structure and behavior, in: Proceedings ACM SIGMOD, May 1990, pp. 194–203.
[14] J.E. Cook, A.L. Wolf, Discovering models of software processes from event-based data, ACM Trans. Softw. Eng. Methodol. 7 (3) (1998) 215–249.
[15] J. Van den Bercken, B. Seeger, P. Widmayer, A generic approach to bulk loading multidimensional index structures, in: Proceedings of the 23rd International Conference on VLDB, Athens, Greece, August 1997, pp. 406–415.
[16] J. Eder, Extending SQL with general transitive closure and extreme value selections, TKDE 2 (4) (1990).
[17] J. Eder, E. Panagos, M. Rabinovich, Time constraints in workflow systems, in: CAiSE, June 1999, pp. 286–300.
[18] J. Eder, G.E. Olivotto, W. Gruber, A data warehouse for workflow logs, in: EDCIS, 2002, pp. 1–15.
[19] T. Feder, R. Motwani, R. Panigrahy, C. Olston, J. Widom. Computing the median with uncertainty, in: STOC, Portland, Oregon, May 2000, pp. 602–607.
[20] V. Gaede, O. Günther, Multidimensional access methods, ACM Comput. Surveys 30 (2) (1998) 170–231.
[21] H. Garcia-Molina, K. Salem, Sagas, in: Proceedings of ACM SIGMOD, May 1987, pp. 249–259.
[22] S. Geffner, D. Agrawal, A. El Abbadi, T.R. Smith, Relative prefix sums: an efficient approach for querying dynamic OLAP data cubes, in: Proceedings ICDE Conference, Sydney, Australia, March 1999, pp. 328–335.
[23] A. Gilbert, Y. Kotidis, S. Muthukrishnan, M. Strauss, How to summarize the universe: dynamic maintenance of quantiles, in: Proceedings of VLDB, 2002, pp. 454–465.

[24] J. Gray, The Benchmark Handbook for Database and Transaction Processing Systems, second ed., Morgan Kaufmann, San Franscisco, 1993.
[25] M. Greenwald, S. Khanna, Space-efficient online computation of quantile summaries, in: Proceedings of ACM SIGMOD, 2001, pp. 58–66.
[26] D. Grigori, F. Casati, M. Castellanos, U. Dayal, M. Sayal, M.-C. Shan, Business process intelligence, Comput. Industry 53 (3) (2004) 321–343.
[27] H. Gupta, Selections of views to materialize in a data warehouse, in: Proceedings of ICDT, Delphi, January 1997, pp. 98–112.
[28] V. Harinarayan, A. Rajaraman, J. Ullman, Implementing data cubes efficiently, in: Proceedings of ACM SIGMOD, Montreal, Canada, June 1996, pp. 205–216.
[29] C.-T. Ho, R. Agrawal, N. Megiddo, R. Srikant, Range queries in OLAP data cubes, in: Proceedings ACM SIGMOD, Tucson, Arizona, May 1997, pp. 73–88.
[30] R. Hull, F. Llirbat, E. Siman, J. Su, G. Dong, B. Kumar, G. Zhou, Declarative workflows that support easy modification and dynamic browsing, in: Proceedings of the International Joint Conference on Work Activities Coordination and Collaboration, 1999, pp. 69–78.
[31] H. Pirahesh, I.S. Mumick, R. Ramakrishnan, The magic of duplicates and aggregates, in: Proceedings of VLDB, August 1990, pp. 264–277.
[32] M. Kamath, K. Ramamritham, Correctness issues in workflow management, Distrib. Syst. Eng. J. 3 (4) (1996) 213–221.
[33] R. Kimball, The Data Warehouse Toolkit, John Wiley & Sons, 1996.
[34] Y. Kotidis, N. Roussopoulos, A case for dynamic view management, ACM Trans. Database Syst. 26 (4) (2001) 388–423.
[35] P. Larson, V. Deshpande, A file structure supporting traversal recursion, in: Proceedings of ACM SIGMOD, June 1989.
[36] B. List, J. Schiefer, M. Tjoa, Process-oriented requirement analysis supporting the data warehouse design process—a use case driven approach, in: Proceedings of DEXA, September 2000, pp. 593–603.
[37] G.S. Manku, S. Rajagopalan, B.G. Lindsay, Approximate medians and other quantiles in one pass and with limited memory, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, Seattle, Washington, June 1998, pp. 426–435.
[38] C. Mohan, G. Alonso, R. Günthör, M. Kamath, Exotica: a research perspective on workflow management systems, IEEE Data Eng. Bull. 18 (1) (1995) 19–26.
[39] J. Moss, Nested transactions: an approach to reliable computing. Ph.D. thesis, MIT, Cambridge, MA, 1981.
[40] S.B. Navathe, Evolution of data modeling for databases, Commun. ACM 35 (9) (1992) 112–123.
[41] P. O'Neil, G. Graefe, Multi-table joins through bitmapped join indices, SIGMOD Record 24 (3) (1995) 8–11.
[42] P. O'Neil, D. Quass, Improved query performance with variant indexes, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, Tucson, Arizona, May 1997, pp. 38–49.
[43] S. Ozeki, N. Ikeuchi, Customer service evaluation in the telephone service provisioning process, in: Proceedings of Winter Simulation Conference, Arlington, Virginia, December 1992, pp. 1341–1348.
[44] A. Rosenthal, S. Heiler, U. Dayal, F. Manola, Traversal recursion: a practical approach to supporting recursive applications, in: Proceedings of ACM SIGMOD, 1986.
[45] P. Scheuermann, J. Shim, R. Vingralek, WATCHMAN: A data warehouse intelligent cache manager, in: Proceedings of the 22nd VLDB Conference, Bombay, India, September 1996, pp. 51–62.
[46] J. Shim, P. Scheuermann, R. Vingralek, Dynamic caching of query results for decision support systems, in: it SSDBM, Cleveland, Ohio, July 1999, pp. 254–263.
[47] A. Shukla, P.M. Deshpande, J.F. Naughton, Materialized view selection for multidimensional datasets, in: Proceedings of the 24th VLDB Conference, New York City, New York, August 1998, pp. 488–499.
[48] J.R. Smith, C. Li, V. Castelli, A. Jhingran, Dynamic assembly of views in data cubes. in: Proceedings of the Symposium on Principles of Database Systems (PODS), Seattle, Washington, June 1998, pp. 274–283.
[49] S. Sudarshan, R. Ramakrishnan, Aggregation and relevance in deductive databases, in: Proceedings of VLDB, Barcelona, Spain, 1991, pp. 501–511.
[50] M. Wu, A.P. Buchmann, Encoded bitmap indexing for data warehouses, in: Proceedings of the Fourteenth ICDE Conference, Orlando, Florida, February 1998, pp. 220–230.



Yannis Kotidis is a Senior Technical Specialist at the Database Research Department of AT&T Labs-Research in Florham Park, New Jersey. He holds an B.Sc. degree in Electrical Engineering and Computer Science from the National Technical University of Athens and a Ph.D in Computer Science from the University of Maryland. His interests include data warehousing, approximate query processing, data quality in large datasets and sensor networks.