A Data Warehousing Architecture for Enabling Service Provisioning Process

Yannis Kotidis

AT&T Labs-Research
kotidis@research.att.com

Abstract

In this paper we focus on the following problem in information management: given a large collection of recorded information and some knowledge of the process that is generating this data we want to build a compact, non-redundant collection of summary (aggregate) tables and indices to facilitate flexible decision support analysis. The additional knowledge is depicted as a graph called the *sketch* that is supper-imposed over the process and indicates particular parts that we want to analyze. We first show how to select a minimum set of views to answer queries with path-expressions over the given sketch. For queries that also include aggregations, 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* aggregate values and allow us to query a view efficiently even for non-materialized aggregates. Experimental results show these schemes to perform well on synthetic and real datasets.

1 Introduction

The astonishing success of information technology has led to an explosive growth in the amount of data that is being recorded in daily basis on various domains. Often data is recorded internally in an organization for provisioning certain business related tasks. For example 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 chance 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.

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 [14]. Each customer record is an instance of some part of this workflow process annotated with timing information and other business related attributes.

When trying to apply conventional data warehousing techniques for a service provisioning process 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. Our work has been motivated from related literature on recursive queries (e.g. [10, 12, 15, 16]) and workflow systems (e.g. [2, 6, 14]). We assume that 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. Given such a sketch, we explore how can we build a compact, non-redundant collection of summary tables and indices to facilitate flexible decision support analysis. Our contributions are outlined bellow:

- conventional relational implementations are incapable of providing flexible analysis of the recorded data with respect to a given sketch [5]. In this paper we introduce pair-wise aggregate queries as a mean to describe the *scope* of an interesting aggregation. Such a query consists of a *path expression* over the sketch that collects relevant recorded information and a user-defined aggregate function $\mathcal{F}$ that consolidates this data. We then show how to express more complex expressions and evaluate them as a series of pair-wise queries.
- evaluating pair-wise aggregate queries on-the-fly can

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

**Proceedings of the 27th VLDB Conference,
Roma, Italy, 2001**

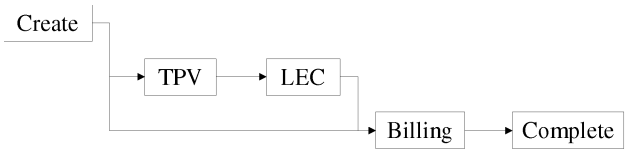


Figure 1: Telephone Service Provisioning example

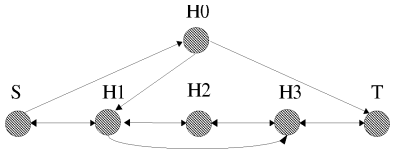


Figure 2: Delivery Service Provisioning example

be prohibitively costly. 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 optimize evaluation of queries with path expressions based on a non-redundant collection of views materialized as bitmapped indexes [3, 13, 17]. For queries that also contain aggregations we propose the use of pair-wise aggregate views that contain pre-computed results of pair-wise queries and discuss different materialization policies. 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.

The rest of the paper is organized as follows: Section 2 motivates the problem from two practical examples. Sections 3, 4 introduce the notion of the sketch and define pair-wise queries. In section 5 we show the shortcomings of various relational models for the service provisioning data and then introduce materialized pair-wise views, while in sections 6 and 7 we show how to choose from, index and query these views for answering user queries. Finally section 8 contains the experiments and in section 9 we draw the conclusions.

2 Motivation

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

Telephone Service Provisioning: this operation 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. Figure 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" 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 Figure 1 his 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 the attributes of the data 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 hours, 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*.

Query (1) is an example of a path-query, discussed in section 6. Query (2) inquires timing information recorded as the the order flows though the process. This is critical for most service ordering systems as user satisfaction is primarily based on timely processing of his orders [2, 6]. For query (3) we exploit a version attribute attached to each record that counts all modifications to the customer's order.

Delivery Service Provisioning: Figure 2 depicts a number of "hubs" H_i that are used to interconnect two sites S and T. Connecting paths may have different capacities, bandwidth, latencies etc. Also some connections are

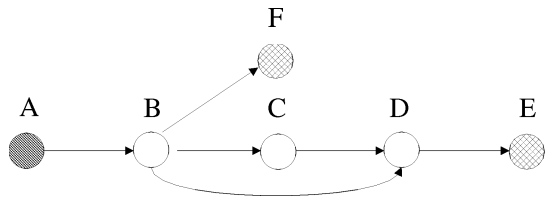


Figure 3: Simple Sketch

bi-directional while others not, as shown in the Figure. Assuming that we want to provision delivery of services from element S to element 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 way to state the query is to ask for all flows of the form $S \rightarrow H1 \rightarrow H2 \rightarrow H3 \rightarrow T$, $S \rightarrow H0 \rightarrow H1 \rightarrow H2 \rightarrow H3 \rightarrow T$ or $S \rightarrow H0 \rightarrow H1 \rightarrow H3 \rightarrow T$. Finally query (6) requests a specific path from the sketch along with additional timing constraints.

3 Data Model

We first give 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 6.2 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:

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

$\mathcal{R}$ 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. 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).

As a running example we will be using the sketch of Figure 3. Node A is a starting node and nodes F,E are terminal nodes.

4 Pair-wise Queries

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 $S_{u \rightarrow v}$ 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 $1 \leq i \leq j \leq k_r$ and $e_i = (u, ?), e_j = (?, v)$, where $?$ denotes a "don't care" value.* ∎

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

Definition 4.2 *The projection operator $P_{u \rightarrow v}(r)$ returns the recored measure data along the path $u \rightarrow v$: $P_{u \rightarrow v}(r) = \{\{x_i, \ldots, x_j\} | \exists i, j \, s.t. \, 1 \leq i \leq j \leq k_r \, and \, e_i = (u, ?), e_j = (?, v)\}$.* ∎

Having subset the input dataset on a portion of the process we can then apply any interesting aggregate function like $sum()$, $count()$, $min()$, $max()$, $median()$ e.t.c over the selected measures.

Definition 4.3 *Let $\mathcal{F}_{u \rightarrow v} = \mathcal{F}(P_{u \rightarrow v}(r)) = \mathcal{F}(x_i, \ldots, x_j)$ be the result of aggregate function $\mathcal{F}$ when applied to measures $x_i, \ldots, x_j$ collected along a path $u \rightarrow v$ for record r. A pair-wise aggregation query $l \leq \mathcal{F}_{u \rightarrow v} \leq h$ retrieves all records $r \in S_{u \rightarrow v}(\mathcal{R})$ for which the evaluated measure data satisfies the inequality's bounds l, h.* ∎

The definition is also extended to strictly greater-than/less-than operators and single-sided queries. For example query 2 of section 2 can be stated as: $max_{Create \rightarrow Complete} > 8$ hours AND $sum_{Create \rightarrow Complete} < 1$ day. We further define $S_{u \rightarrow u}(\mathcal{R})$ to return the records that contain node u. Then the binary function $exist_{u \rightarrow v}$ simply evaluates to 1 (true) for each record returned by $S_{u \rightarrow v}(\mathcal{R})$.

5 Relational Design

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 $(u, v) \in E$ can be easily 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 $sum_{B\to D} \geq 5$ returns the union of the following expressions:

q_1 : `select` $R_1.r_{id}$, $R_1.x + R_2.x$
`from` $R_1\,\mathcal{R}, R_2\,\mathcal{R}$
`where` $R_1.r_{id} = R_2.r_{id}$ `and` $R_1.e_{id}$ = ``BC''
`and` $R_2.e_{id}$ = ``CD'' `and` $R_1.x + R_2.x \geq 5$

q_2 : `select` $R.r_{id}$, $R.x$
`from` $R\,\mathcal{R}$
`where` $R.e_{id}$ = ``BD'' `and` $R.x \geq 5$

An alternative horizontal representation of the data is: $\mathcal{R}^h(r_{id}, x_{e_1}, \ldots, x_{e_{|E|}})$, 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.

For aggregate functions like $sum()$ and $count()$ we can exploit a dual prefix-representation [8, 9] of the record:

$$r^d_{\mathcal{F}} = \{(v_1^{e_1}, 0), (v_1^{e_2}, \mathcal{F}(x_1)), \ldots, (v_1^{e_{k_r}}, \mathcal{F}(x1, \ldots, x_{k_r-1})), (v_2^{e_{k_r}}, \mathcal{F}(x1, \ldots, x_{k_r}))\} \quad (2)$$

The prefix-$\mathcal{F}$ 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 $\mathcal{R}^d_{\mathcal{F}}(r_{id}, v_{id}, x_{\mathcal{F}})$.

A common restriction of all three representations is that, in many cases, we can not use an index for the predicate on the measures (e.g. query q_1). 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 he has to compose his query appropriately to reflect the part of the sketch that he is interested in. Materialized views can be used to accelerate query performance and also ease navigation through the dataset.

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

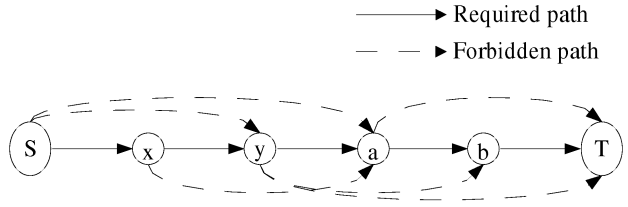


Figure 4: Required and forbidden paths in a sketch (case 1)

Definition 5.1 *Given a pair u, v for which there exists a path $u \to v$ in $G(V, E)$ we define view $\mathcal{V}_{\mathcal{F}_{u\to v}}$ as the projection of all records in $\mathcal{V}_{\mathcal{F}}$ that contain these states.* ∎

6 Processing Path Queries

For a start we assume that $\mathcal{F} = exist()$, i.e. we are only interested in the transitions stored in the records and not in the actual measures. A *pair-wise path query* $exist_{u\to v} = 1$ retrieves all records that include a path from u to v.[1] View $\mathcal{V}_{exist_{u\to v}}$ 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 $|\mathcal{R}|$:

Definition 6.1 *$\mathcal{V}_{exist_{u\to v}}$ is a bitmap of length $|\mathcal{R}|$ with bits set at position i, for every record $r_i \in S_{u\to v}(\mathcal{R})$.* ∎

We now investigate the problem of selecting the minimum subset of views $\mathcal{V}_{exist_{u\to v}}$ 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 6.2 *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 $\mathcal{V}_{exist_{a\to b}}$ is equivalent ($\equiv$) to view $\mathcal{V}_{exist_{x\to y}}$ if they contain the same records for any instance of $\mathcal{R}$.* ∎

The definition implies that $\mathcal{V}_{exist_{a\to b}} \equiv \mathcal{V}_{exist_{x\to y}}$ if each valid record that contains a path $x \to y$ also contains a path $a \to b$ and vice-versa. This means that there is a path $x \to a$ or $a \to x$ 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 $a \to b$:

1. $\nexists$ a path $s \to a$ for all $s \in S$ in $G(V - \{x\}, E - E_x)$[2]

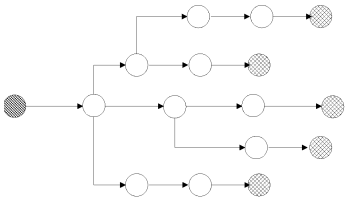


Figure 5: A Service Provisioning Tree

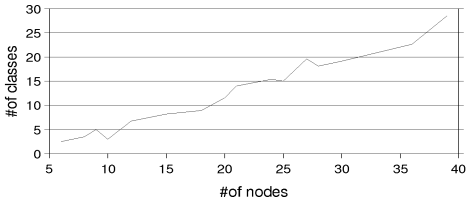


Figure 6: Number of classes

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 $\mathcal{V}_{exist_{a\rightarrow b}} \equiv \mathcal{V}_{exist_{x\rightarrow y}}$ 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. $\nexists$ a path $y \rightarrow t$ for all $t \in T$ in $G(V - \{a\}, E - E_a)$
3. $\nexists$ a path $y \rightarrow t$ for all $t \in T$ in $G(V - \{b\}, E - E_b)$
4. $\nexists$ a path $s \rightarrow a$ for all $s \in S$ in $G(V - \{y\}, E - E_y)$

Figure 4 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. $\nexists$ a path $b \rightarrow t$ for all $t \in T$ in $G(V - \{y\}, E - E_y)$
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 secs in a 600Mhz Pentium III PC.

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

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

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

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

9. $\nexists$ path $s \rightarrow t$ for all $s \in S$ and $t \in T$ in $G(V - \{u, v\}, E - E_u - E_v)$

For the sketch of our example just 3 representative views are needed whose combined size is $3|\mathcal{R}|$ (uncompressed) bits. In practice $|\tilde{\mathcal{V}}|$ depends on the complexity of the sketch. Business processes often have parts with sequential actions $v_1 \rightarrow v_2 \rightarrow \ldots$. 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 $\leq k$ and repeat several times. Figure 5 shows a possible result for $k = 5$ and four iterations. Processes of this form are common in the telephone service provisioning domain. In Figure 6 we plot the number of classes $|\tilde{\mathcal{V}}|$ 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 e_{id}. We believe this to be true in many practical cases.

6.1 Evaluating more Complex Path Queries

We now define the selection operator $S_{v_1\rightarrow\cdots\rightarrow v_n}(\mathcal{R})$ that returns all records that visit nodes $v_1 \ldots, v_n$ in the specified order. A multi-node path query $exist_{v_1\rightarrow\cdots\rightarrow v_n}$

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

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

evaluates to 1 for all records in $S_{v_1 \to \cdots \to v_n}(\mathcal{R})$ and is computed as follows:

$n = 1$: We answer the query by ORing the bitmaps of all views $\tilde{\mathcal{V}}_{exist_{s \to v_1}}, s \in S$. Alternatively we could use views $\tilde{\mathcal{V}}_{exist_{v_1 \to t}}, t \in T$. Overall, we need to OR at-most $min(|S|, |T|, |\tilde{\mathcal{V}}|)$ bitmaps.

$n = 2$: We simply answer the query using view $\tilde{\mathcal{V}}_{exist_{v_1 \to v_2}}$.

$n > 2$: We AND bitmaps of views $\tilde{\mathcal{V}}_{exist_{v_i \to v_{i+1}}}, i = 1, \ldots, n-1$. Up to $min(n-1, |\tilde{\mathcal{V}}|)$ 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 $exist_{C \to C} = \tilde{\mathcal{V}}_{exist_{A \to B}}$ AND $\tilde{\mathcal{V}}_{exist_{B \to D}}$ AND $\tilde{\mathcal{V}}_{exist_{D \to E}}$ AND NOT $\tilde{\mathcal{V}}_{exist_{A \to C}} = \tilde{\mathcal{V}}_{exist_{A \to E}}$ AND NOT $\tilde{\mathcal{V}}_{exist_{A \to C}}$.

A path query can be answered using bit-mapped indices on the nodes present in a 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. The details of this reduction are omitted due to space limitations.

6.2 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 $\mathcal{F}_{u \to u}$ is defined to aggregate all measures between the first occurrence of node u and the last occurrence of node v in the record.[3]

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. $\nexists$ a path $s \to y$ for all $s \in S$ in $G(V - \{x\}, E - E_x)$

Case (2): The relative order of the nodes can not 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 6.1 the resulting bitmap describes more records that are actually in $S_{v_1 \to \cdots \to v_n}(\mathcal{R})$ as the evaluation process does not guarantee an order between paths $v_i \to v_{i+1}$. Thus, we use the optimized expression as a dirty filter and evaluate the retrieved records at a latter step to eliminate possible false positives. Compared to using bitmapped indices on the nodes, we can prove the following:[4]

Lemma 6.3 *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.* ∎

7 Processing Pair-wise Aggregate Queries

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

$$l \leq \mathcal{F}_{a \to b} \leq h \tag{3}$$

when $\mathcal{F}$=$sum(), count(), max(), min()$. A pair-wise aggregate view $\mathcal{V}_{\mathcal{F}_{x \to y}}$ contains pairs of $(r_{id}, \mathcal{F}(P_{x \to y}(r)))$ 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 $\mathcal{R}$.

If another view $\mathcal{V}_{\mathcal{F}_{x \to y}}$ with $(x, y) \neq (a, b)$ and $\mathcal{V}_{exist_{x \to y}} \equiv \mathcal{V}_{exist_{a \to b}}$ 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 $l \leq \mathcal{F}(P_{a \to b}(r)) \leq h$.

In general we may be able to exploit the aggregates stored in view $\mathcal{V}_{\mathcal{F}_{x \to y}}$ when evaluating query (3) 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:

$$\mathcal{F}(P_{x \to y}(r)) = \mathcal{F}'(\mathcal{F}(P_{x \to a}(r)), \mathcal{F}(P_{a \to b}(r)), \mathcal{F}(P_{b \to y}(r))) \tag{4}$$

where $\mathcal{F}'$=$\mathcal{F}$ for $\mathcal{F} = max(), min(), sum()$ and $\mathcal{F}$'=$sum()$ for $\mathcal{F}$=$count()$. This well know property of a distributive function is frequently exploited to share computation of data cube aggregates [1].

For any two pairs of nodes (x, y) and (a, b) for which case (3) is true we denote that $\mathcal{V}_{\mathcal{F}_{a \to b}} \preceq \mathcal{V}_{\mathcal{F}_{x \to y}}$. 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 Figure 3 the $\preceq$ relation is shown in Figure 7.

[3] other definitions are possible depending on the context.

[4] 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.

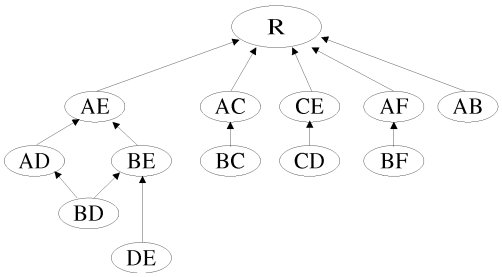


Figure 7: The $\preceq$ partial order

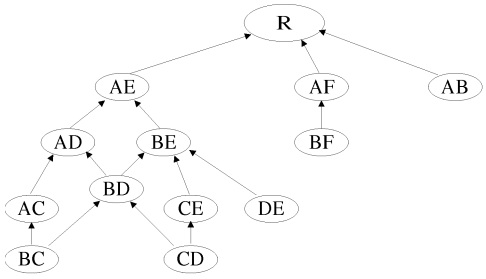


Figure 8: The $\prec$ partial order

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

7.1 Weaker Condition

We can relax the path equivalence requirement at the expense of getting more false positives. We define that $\mathcal{V}_{\mathcal{F}_{a\to b}} \prec \mathcal{V}_{\mathcal{F}_{x\to y}}$ 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 $S_{a\to b}(\mathcal{R}) \subseteq S_{x\to y}(\mathcal{R})$. For the sketch of Figure 3 the $\prec$ partial order is depicted in Figure 8. The top-level views of the order are denoted as $\dot{\mathcal{V}}$ and their number as $|\dot{\mathcal{V}}|$. In this example $|\dot{\mathcal{V}}| = 3$.

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

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

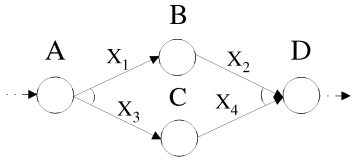


Figure 9: Sketch with Parallel Parts

7.2 Pair-wise sum-Queries

Pair-wise sum-queries are queries of the form: $l \leq sum_{a\to b} \leq h$. We assume that at least the top-level views $\hat{\mathcal{V}}_{sum_{u\to v}}$ of the $\preceq$ hierarchy of Figure 7 are materialized and we explore how queries on the remaining pairs can be translated and executed efficiently using these views.

If view $\mathcal{V}_{sum_{a\to b}}$ is not computed we are using materialized view $\mathcal{V}_{sum_{x\to y}} = \hat{\mathcal{V}}_{sum_{a\to b}}$ as a dirty filter to find candidate records that we retrieve and evaluate from $\mathcal{R}$ in a latter step. Along with the views we store in the database the dependency graph of Figure 7. This graph has a size of $O(|V|^2)$, which we consider insignificant. 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 $r \in S_{D\to E}(\mathcal{R})$. 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 equation (4) query $l \leq sum_{a\to b} \leq h$ is re-written as:[5]

$$l' = l + min_sum_{x\to a} + min_sum_{b\to y} \leq sum_{x\to y} \leq h + max_sum_{x\to a} + max_sum_{b\to y} = h' \quad (5)$$

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.

7.2.1 Handling Parallel Paths

In business workflows it is common to have multiple subtasks that are spawn from a node. Consider for example the sketch of Figure 9. 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 [15]). 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.

[5] we assume that $min_sum_{u\to v} = max_sum_{u\to v} = 0$ if $u = v$.

7.2.2 Using Phantom Aggregates to Optimize Performance on non-Materialized Views

Each top-level view $\mathcal{V}$ of Figure 7 defines a poset $D_{\mathcal{V}}$ with all views $\mathcal{V}' \preceq \mathcal{V}$. Let $\mathcal{L}_{\mathcal{V}}$ be the set of lower bounds in $D_{\mathcal{V}}$. For example $\mathcal{L}_{\mathcal{V}_{sum_{A\to E}}} = \{\mathcal{V}_{sum_{B\to D}}, \mathcal{V}_{sum_{D\to E}}\}$. One can prove that for each top-level view $\mathcal{V}$ in the $\preceq$ order, $\mathcal{L}_{\mathcal{V}}$ contains views on non-overlapping paths in $G(V, E)$. Our key-idea is to use the values of these views to cluster the records of $\mathcal{V}$ 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 $\mathcal{L}_{\mathcal{V}}$. We call these values *phantom aggregates* as they don't appear in $\mathcal{V}$. 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 $\mathcal{V}$, using one tree per partition. The upper partitioning scheme is fixed (e.g. 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 B, we generate B partitions for the values of $\mathcal{V} = \mathcal{V}_{sum_{u\to v}}$ in the following manner: we treat each value of $\mathcal{V}$ as an $|\mathcal{L}_{\mathcal{V}}|$-dimensional point with coordinates defined by the phantom $sum()$ aggregates of views in $\mathcal{L}_{\mathcal{V}}$. We then build the first $log_{|\mathcal{L}_{\mathcal{V}}|}(B)$ levels of a kd-tree tree for these values. Each node at level $log_{|\mathcal{L}_{\mathcal{V}}|}(B)$ contains a pointer to a partition of the original values in $\mathcal{V}$ 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 $sum(P_{u\to v}(r))$ as the key and the corresponding r_{id}s 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 $\mathcal{V}' \preceq \mathcal{V}$ is done using the top-level kd-tree nodes for pruning the search. Queries on values of $\mathcal{V}$ 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 $log_{|\mathcal{L}_{\mathcal{V}}|}(B)$ levels of the kd-tree is $2 * (B - 1)$, which fits in a single data page for small Bs. Figure 10 shows the hybrid structure for view $\mathcal{V}_{sum_{A\to E}}$.

Grid-based method. We create a hybrid data-structure, which partitions the records of the view by super-imposing a $|\mathcal{L}_{\mathcal{V}}|$-dimensional grid on its values. The simplest way to do this is to partition the aggregates of each view $\mathcal{V}' \in \mathcal{L}_{\mathcal{V}}$ by computing appropriate quantiles [7, 11]. Multidimensional index loading techniques like [4] 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 $\mathcal{V}$ is built for each cell.

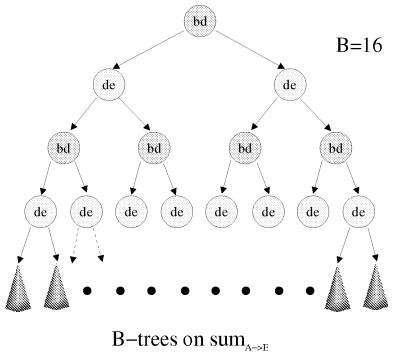


Figure 10: Implementation of view $\mathcal{V}_{sum_{A\to E}}$

7.3 Pair-wise count-Queries

For evaluating query $l \leq count_{a\to b} \leq h$ 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 6.1 we can find all records that contain path p_i by accessing up to $|\tilde{\mathcal{V}}|$ 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 $\mathcal{R}$ and generates no false positives/dismissals. The method is applicable when views $\hat{\mathcal{V}}_{\mathcal{F}_{a\to b}}$ (for any function $\mathcal{F}$) are available. The answer is similarly computed by merging appropriate lists of records ids. If views $\mathcal{V}_{\mathcal{F}_{a\to b}}$ are used then the answer may include false positives.

7.4 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: $l'=\mathcal{F}(l, min_\mathcal{F}_{x\to a}, min_\mathcal{F}_{b\to y}) \leq \mathcal{F}_{x\to y} \leq \mathcal{F}(h, max_\mathcal{F}_{x\to a}, max_\mathcal{F}_{b\to y})=h'$ for $\mathcal{F}=min()/max()$.

8 Experiments

8.1 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. 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 $\mathcal{V}_{sum_{v_1\to v_{n+1}}}$ is the only view materialized. Due to space limitations we defer experiments with other aggregate functions to the full version of this paper.

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:

$$sum_{v_i\to v_{j+1}} \geq E[x_i + \cdots + x_j] + kDEV[x_i + \cdots + x_j] \quad (6)$$

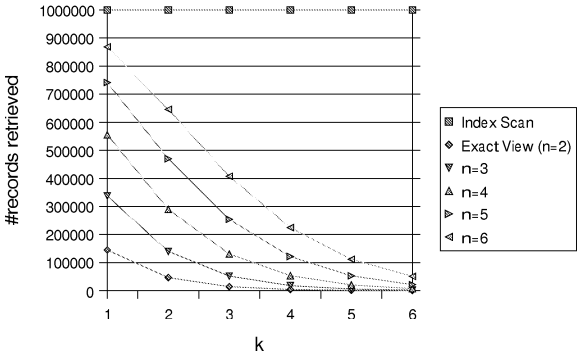


Figure 11: Querying for outliers

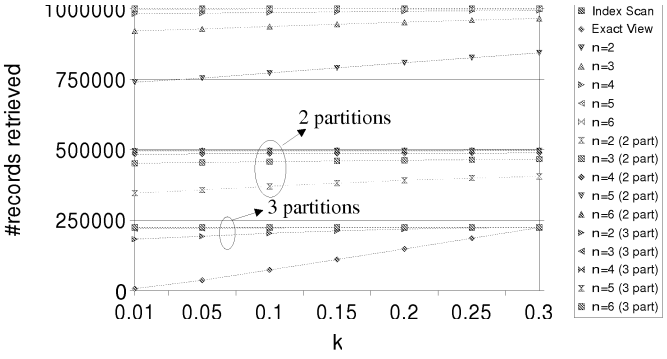


Figure 12: Querying on hot-spots

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:

$$E[x_i + \ldots x_j] - kDEV[x_i + \ldots x_j] \leq sum_{v_i \to v_{j+1}} \leq E[x_i + \ldots x_j] + kDEV[x_i + \ldots x_j] \quad (7)$$

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 $\mathcal{V}_{sum_{v_1 \to v_{n+1}}}$ for computing outliers on the first two transitions: $sum_{v_1 \to v_3} = x_1 + x_2 \geq E[x_1 + x_2] + kDEV[x_1 + x_2]$. We used a synthetic dataset of 1,000,000 records with a transition $v_1 \to \cdots \to v_{n+1}$. In Figure 11 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 $\mathcal{R}$ 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 Figure 12 we experiment with queries on hot-spots using as an example query $E[x_1] - kDEV[x_1] \leq sum_{v_1 \to v_2} \leq E[x_1] + kDEV[x_1]$, varying k from 0.01 to 0.30 and n from 1 (exact view) to 6. For these queries, view $\mathcal{V}_{sum_{v_1 \to v_{n+1}}}$ 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 7.2.2 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 $\mathcal{V}_{sum_{v_1 \to v_{n+1}}}$ was 11.61MB when stored as a single B-tree, 11.73MB when stored using a 2x2x2x2x2x2 grid and 14.31MB for the 3x3x3x3x3x3 grid[6]. This is comparable to the size of a B-tree on e_{id} (11.6MB).

8.2 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 $\mathcal{V}_{sum_{v_i \to v_{j+1}}} \prec \mathcal{V}_{sum_{v_1 \to v_{n+1}}}$. 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 $sum_{v_1 \to v_3} \geq E[x_1 + x_2] + 6 * DEV[x_1 + x_2]$ (for n=6) using the view or an index on e_{id}s as shown in Figure 13. 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.

8.3 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 42754 records. Measures x_i record timing information. In Figure 14 we normalize the number of records retrieved from view $\mathcal{V}_{sum_{v_1 \to v_6}}$, 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 (6) 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 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.

[6] we also tested a grid of the form 3x3x1x1x1x1 that was slightly worse on queries on x_1 but used only 11.62MB of disk space

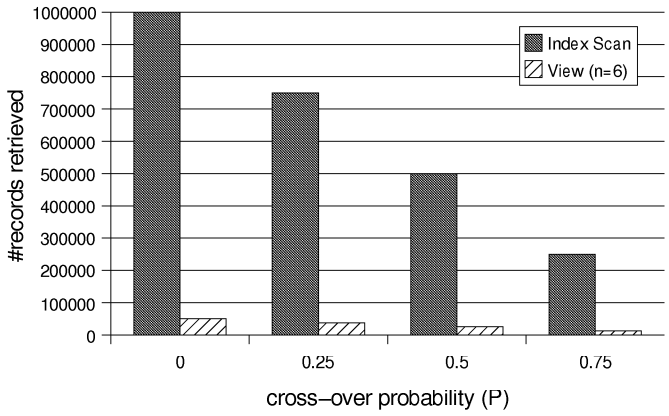


Figure 13: Testing the $\prec$ Partial Order

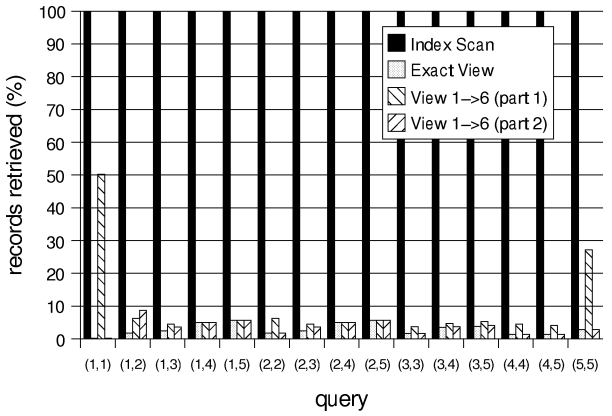


Figure 14: Querying for Outliers, Real Data

9 Conclusions

In this paper we showed how to apply data warehousing techniques for organizing service provisioning data. Our solution was based on the notion of a sketch as a used-defined description of the underlying process that generates the records. We explored the implications of storing, indexing and querying this data in a relational engine. Our observation was that materialized (aggregate) views can be effectively used to query and index such records. We investigated the problem of selecting a minimum (under different requirements) set of aggregate views for answering user queries and also discussed effective indexing mechanisms for their records that achieve superior performance over index scans.

10 Acknowledgements

The author would like to thank Ken Church, Divesh Shrivastava as well as the VLDB referees for their helpful comments and suggestions.

References

[1] S. Agrawal, R. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and S. Sarawagi. On the Computation of Multidimensional Aggregates. In *Proc. of VLDB*, pages 506–521, Bombay, India, 1996.

[2] C. Bettini, X.S. Wang, and S. Jajodia. Free Schedules for Free agents in Workflow Systems. In *Proc. of TIME*, pages 31–38, Nova Scotia, Canada, July 2000.

[3] C. Y. Chan and Y. Ioannidis. Bitmap Index Design and Evaluation. In *Proceedings of SIGMOD*, pages 355–366, Seattle, Washington, June 1998.

[4] J. Van den Bercken, B. Seeger, and P. Widmayer. A Generic Approach to Bulk Loading Multidimensional Index Structures. In *Proc. of VLDB*, August 1997.

[5] J. Eder. Extending SQL with General Transitive Closure and Extreme Value Selections. *TKDE 2(4)*,1990.

[6] J. Eder, E. Panagos and M. Rabinovich. Time Constraints in Workflow Systems. In *CAiSE*, June 1999.

[7] T. Feder, R. Motwani, R. Panigrahy, C. Olston, and J. Widom. Computing the Median with Uncertainty. In *STOC*, pages 602–607, Portland, Oregon, May 2000.

[8] S. Geffner, D. Agrawal, A. El Abbadi, and T. R. Smith. Relative Prefix Sums: An Efficient Approach for Querying Dynamic OLAP Data Cubes. In ICDE, pages 328–335, Sydney, Australia, March 1999.

[9] C. T. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range Queries in OLAP Data Cubes. In *Proceedings SIGMOD*, pages 73–88, Tucson, Arizona, May 1997.

[10] P. Larson and V. Deshpande. A File Structure Supporting Traversal Recursion. In *SIGMOD*, June 1989.

[11] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Approximate Medians and other Quantiles in One Pass and with Limited memory. In *Proc. of SIGMOD*, pages 426–435, Seattle, Washington, June 1998.

[12] I. S. Mumick, H. Pirahesh and R. Ramakrishnan. The Magic of Duplicates and Aggregates. In *Proceedings of VLDB*, pages 264–277, Australia, August 1990.

[13] P. O'Neil and D. Quass. Improved Query Performance with Variant Indexes. In *SIGMOD*, May 1997.

[14] S. Ozeki and N. Ikeuchi. Customer Service Evaluation in the Telephone Service Provisioning Process. In *Proceedings of Winter Simulation Conference*, pages 1341–1348, Arlington, Virginia, December 1992.

[15] A. Rosenthal, S. Heiler, U. Dayal and F. Manola. Traversal Recursion: A Practical Approach to Supporting Recursive Applications. In *SIGMOD*,1986.

[16] S. Sudarshan and R. Ramakrishnan. Aggregation and Relevance in Deductive Databases. In *Proccedings of VLDB*, pages 501–511, Barcelona, Spain, 1991.

[17] M. Wu and A. P. Buchmann. Encoded Bitmap Indexing for Data Warehouses. In *Proc. of ICDE*, 1998.