A Framework for Enabling Query Rewrites when Analyzing Workflow Records*

Dritan Bleco [#1], Yannis Kotidis [#2]

[#]*Athens University of Economics and Business*
76 Patission Street, Athens, Greece
[1]dritanbleco@aueb.gr, [2]kotidis@aueb.gr

Abstract. Workflow records are naturally depicted using a graph model in which service points are denoted as nodes in the graph and edges portray the flow of processing. In this paper we consider the problem of enabling aggregation queries over large-scale graph datasets consisting of millions of workflow records. We discuss how to decompose complex, ad-hoc aggregations on graph workflow records into smaller, independent computations via proper query rewriting. Our framework allows reuse of precomputed materialized query views during query evaluation and, thus, enables view selection decisions that are of immense value in optimizing heavy analytical workloads.

1 Introduction

A Workflow Management application in a Customer Support Call Center manages massive collections of different trouble (issue) tickets generated daily by an Issue Tracking System (ITS). Such an application tracks all activities from the creation of the trouble ticket till its completion. A trouble ticket is commonly composed by different flows of tasks that may be serviced in parallel or sequentially by distinct agencies within the company's domain called service points.

A natural way to depict the workflow followed by a trouble ticket is to utilize a graph model in which service points are denoted as nodes and edges depict the flow of processing. Figure 1 depicts the graph instance associated with a particular trouble ticket. Numeric labels on the nodes depict processing time (in days) within the service point. The values on the edges depict delays for propagating the associated task(s) from one service point to another (e.g. handoffs latencies or wait times).

In this paper we consider the problem of enabling analytical queries over large-scale graph datasets related to workflow management applications such as those serving ITS. We describe a comprehensive framework for modeling analytical queries that range over the structure of the graph records. For example, queries like "find the average ticket completion time" in an ITS application are naturally captured by our framework.

As will be explained, in our framework we decompose complex, ad-hoc aggregations on graph workflow records into smaller, independent computations via proper query rewriting. In the context of a large-scale data warehouse, our framework allows

* This work was partially supported by the Basic Research Funding Program, Athens University of Economics and Business.

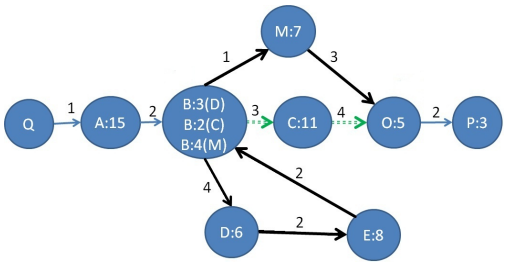

Fig. 1. A sample workflow record

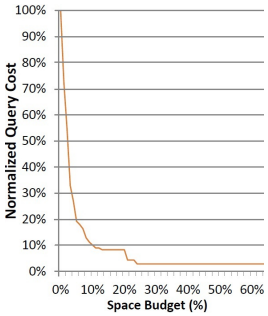

Fig. 2. Benefits of Materialization

re-use of precomputed materialized query views during query evaluation and, thus, enables view selection decisions [1]. Our framework is not an attempt to provide a new algebra for graph analytics. All computations discussed in our work, can be naturally expressed in relational algebra, given a decomposition of the graph records in a relational backend. Our query rewriting framework allows us to optimize ad-hoc aggregations over workflow records by utilizing precomputed views, independently of the technological platform used for storing and querying such records. In our experimental evaluation, we demonstrate that our techniques can be used to expedite costly queries via query rewriting and available precomputations in the data warehouse.

2 Motivation and Basic Concepts

Figure 1 captures information related to a single trouble ticket workflow that we will refer to as a *record* henceforth. Service point B is a special type of node called *splitting node*, where processing of the ticket is split between two tasks that are processed in parallel following different *flows* in particular $[B, C, O]$ (or $[BCO]$ for brevity) and $[BDEBMO]$. These independent flows are merged (synchronized) at node O, which is a *merging node*. From there, the ticket is moved to node P where it is marked as completed. There can be recordings of different cost metrics, in addition to the time we consider in this example. A decision maker would like to analyze data according to all these attributes over different parts of the graph for thousands of such records. In what follows we describe a formal way to analyze such data.

In a workflow management application a flow is simply a sequence of nodes resulting from the concatenation of adjacent edges. When a flow is uniquely identified by its endpoints, for brevity, we omit the internal nodes. For example flow $[ABC]$ is depicted as $[AC]$. In the record of Figure 1, flow $[BDEBMO]$ contains a cycle, $[BDEB]$ in our case. This cycle shows that this flow, for some reason, goes back to node B for further proccessing and after that to node M. On each node there is a cost related with each input/output edge. For example, to proceed from B to M the flow was processed locally on B for four days.

When we analyze the costs on a flow in a trouble ticket often we want to omit the costs on the two side nodes. Borrowing notation from mathematics, we denote this

"open-ended" flow similarly to an interval whose endpoints are excluded. For example (BCO) denotes our intention to look at the processing of the flow excluding cost related to its starting and ending points. Similarly, a flow can be opened in only one of its side nodes, i.e. $[BO)$.

In a large ITS application, a lot of tickets are processed daily, creating a massive collection of such data. Given these primitive data an analyst should be able to answer queries like

- Q_1: What is the total wall-clock time for each trouble ticket from its initialization till its completion?
- Q_2: What is the total waiting time at a certain merging node?
- Q_3: What are the total processing time and total delay time?
- Q_4: What is the wasting time due to a non approved task (a circle during the flow or a flag over a node depicts such a situation)?

These queries will be executed over a large collection of tickets. Primitive statistics computed at a per-ticket basis can then be combined to compute aggregate statistcs like average completion time per type of ticket etc. Query Q_1 in this example requires us to find the *longest flow* between nodes Q and P. During the computation, attention should be paid to merging nodes that synchronize execution of parallel flows. In our running example two flows reach merging node O; the fastest one waits for the other flow. For the sample record of Figure 1 the longest flow is $[QABDEBMOP]$ with a total time of 68 days. In this example, query Q_1 spans over the whole record. Depending on the scenario being analyzed, a user may restrict the analysis over parts of the input records (e.g. between two specific service points). Query Q_2 calculates the *longest and shortest flow* among tasks starting at spliting node B (including the cost(s) on this node) and ending at the merging node O. The waiting time is the difference between the returned values of the two subqueries. For our example this is 40-20=20 days. Query Q_3 sums up the measures on the edges (the delay time) and the values on the nodes (the processing time). In this record, the processing is 64 days and delay time 24 days. Query Q_4 first needs to identify the non approved - circles on the ticket and then sum up the total time of the flow related to these circles. For the depicted record the wasting time is related with the cirlce B, D, E, B and equals to 25. Obviously the cost of the second proccess on B is not included because this value is related with the new flow from B to M.

Our framework allows us to model such queries in an intuitive manner via a decomposition of a record into flows and the use of two basic operators we introduce next.

3 Query Rewriting and Evaluation

In order to allow composition of flows we introduce the *merge-join* operator ($\triangleright$) that concatenates two flows f_1 and f_2 that run in parallel when they have the same starting and ending nodes. No merging node should be present in the two flows except the starting or/and the ending node. For flows that are running sequentially, we use a second operator called *union* operator ($\oplus$) that concatenates two flows f_1 and f_2 when the ending node of f_1 is the same as the starting node of f_2 and one of the two flows is open-ended at the common end-point. Using the two operators the ticket depicted in Figure 1 can be rewritten as $[QP]$= $[QAB) \oplus \{[BDEBMO] \triangleright [BCO]\} \oplus (OP]$.

To compute different statistics on these measures we can use a *Flow Level Aggregation* function $F_f(r)$ which takes as input a flow f and a record r. The function F is applied on the measures of the flow and returns f along with the computed aggregate. As an example, in the record r of Figure 1, $SUM_{[QABD)}(r)$ returns flow $[QABD)$ and its duration, i.e. 25 days (denoted as $[QABD)$:25). In case more than one flows are given, the function is computed over the (existing) individual flows and the result is returned along with each respective flow.

In a subsequent step a *Flow Set Level Aggregation* function can be used in order to aggregate the results of the previous step and to return a unique value for a set of flows. As an example, function $MAX(SUM_{[QP]}(r))$ computes the total wall-clock time for the ticket depicted in record r along with the longest flow. The aggregate function over a union among two or more flows can be written also as a union among the aggregate function results of these flows. The aggregate function over the merge join operator among flows can be written as the aggregate function over the set of flows that were related with the merge join. Thus, for our record r we have

$$MAX(SUM_{[QAB)\oplus\{[BDEBMO]\triangleright[BCO]\}\oplus(OP]}(r)) =$$

$$MAX(SUM_{[QAB)}(r) \oplus_{SUM} SUM_{\{[BDEBMO],[BCO]\}}(r) \oplus_{SUM} SUM_{(OP]}(r))$$

The union operator concatenates flows with common ending and starting nodes and, additionally consolidates their measures. In this example, we need to add their measures, and this is indicated with the use of function SUM underneath the operator. In general, the rewrites for pushing flow level aggregation (for the union and merge-join respectively) on a flow are of the form (F,H are appropriate aggregate functions)

$$F_{f=f_1\oplus f_2}(r) = F_{f_1}(r) \oplus_H F_{f_2}(r) \text{ and } F_{f=f_1\triangleright f_2}(r) = F_{f_1,f_2}(r) = \{F_{f_1}(r), F_{f_2}(r)\}$$

Furthermore the Flow Set Level Aggregation can be pushed inside each Flow Level Aggregation Function and can be omitted in case the latter is executed over a simple flow. So continuing the above example we have $MAX(SUM_{[QP]}(r))$=$[QAB) : 18 \oplus_{SUM} [BDEBMO] : 45 \oplus_{SUM} (OP] : 5$=$[QABDEBMOP] : 68$.

In order to demonstrate the effectiveness of our rewrite techniques, we synthesized a random graph consisting of 10000 nodes and 15000 edges and generated 120 million workflow records by selecting random subgraphs from it. The records were stored in a relational backend using a single table storing their edges and appropriate indexes. We created 100 random queries on these records. Half of the queries used the SUM flow level aggregation function. The rest performed, additionally, the MAX flow set level aggregation. We used the Pick By Size (PBS) algorithm [2] for selecting materialized views. In Figure 2 we depict the reduction in query execution cost (compared to running these queries without rewrites) with respect to the available space budget of the views. The results demonstrate that even a modest materialization of 10%, via the use of our rewrites provides substantial savings (up to 90%).

References

1. Kotidis, Y., Roussopoulos, N.: A Case for Dynamic View Management. ACM Transactions on Database Systems **26**(4) (2001)
2. Shukla, A., Deshpande, P., Naughton, J.F.: Materialized View Selection for Multidimensional Datasets. In: VLDB. (1998) 488–499