Efficient Aggregate Computation over Data Streams

Kanthi Nagaraj, K.V.M. Naidu, Rajeev Rastogi, Scott Satkin †

Bell Laboratories-Research, Bangalore, India
{kanthicn, naidukvm, rastogi}@alcatel-lucent.com
scott@satkin.com

***Abstract*— Cisco's *NetFlow Collector* (NFC) is a powerful example of a real-world product that supports multiple aggregate queries over a continuous stream of IP flow records. NFC enables a plethora of network management tasks like traffic demands estimation, application traffic profiling, etc. In this paper, we investigate two *computation sharing* techniques for enabling streaming applications such as NFC to scale to hundreds of queries. Our first technique instantiates certain *intermediate* aggregates which are then used to generate the final answers for input queries. Our second technique *coalesces* the filter conditions of similar queries and uses the coalesced filter to pre-filter stream data input to these queries. Using these techniques, we propose a heuristic to compute a good query plan and perform extensive simulations to show that our heuristic delivers a factor of over 3 performance improvement compared to a naive approach.**

I. Introduction

Our research primarily aims to improve the scalability of NFC-like applications[1]so that they can process hundreds of queries. In an IP network, a flow is essentially a continuous unidirectional sequence of packets from a source device to a destination device. NetFlow [2] is the most widely used IP flow measurement solution today. Each NetFlow record has a number of attributes that describe the various flow statistics. Individual attributes can be classified into one of two categories:

- *Group-by attributes*: source/destination IP addresses for the flow, source/destination ports, ToS byte, protocol, input and output interfaces, etc.
- *Measure attributes*: number of packets/bytes in the flow, begin/end timestamp, flow duration, etc.

NFC collects the NetFlow records exported by devices in the network, and processes user-specified aggregate queries on the collected NetFlow data. Each aggregate query consists of (1) a subset of group-by attributes – records with matching values for attributes in the subset are aggregated together, (2) an aggregate operator (e.g., SUM, COUNT) on a measure attribute – the measure attribute values for aggregated records are combined using the specified aggregate operator, (3) a boolean filter condition on attributes, and (4) a time period over which aggregation is to be performed – after each successive time period, result tuples for the aggregate query (computed over NetFlow records that arrived during the time period) are output.

† The work was done while the author was as intern with Bell Labs Research India

[1]Our techniques work for any multiple-query streaming applications like financial tickers, retail transactions, Web log records, sensor node readings, and call detail records in telecommunications.

Our Contributions. In this paper, we investigate two *computation sharing* techniques for scalable online processing of hundreds of aggregate queries on rapid-rate data streams, in the setting of availability of sufficient memory. The key idea underlying our techniques is to first identify similarities among the group-by attributes and filter conditions of queries, and then use these commonalities as building blocks to generate the final query answers.

We consider two mechanisms for sharing computation among user queries. The first mechanism involves instantiating a few fine-grained *intermediate* aggregates and then using these to generate coarse-grained query answers. Our second mechanism looks to *coalesce* similar filter conditions into a single filter, that is then used as a pre-filter to reduce the amount of data input to the queries. We formulate the problem in graph theoretic setting and prove that the problem of finding a minimum-cost aggregate tree is NP-hard, and propose a randomized heuristic to find good low-cost aggregate trees. We carry out an extensive empirical study with real-life NetFlow data sets to gauge the effectiveness of our two computation sharing mechanisms. Our experimental results indicate that the query plans generated by our proposed heuristic can boost system performance by a factor of over 3 compared to a naive approach which processes each query separately.

II. Related Work

The idea of sharing resources among multiple queries has been explored before by prior work on multi-query optimization [18], [17], but in the context of a conventional DBMS. Our problem has similarities to the materialized view selection problems studied in [16] and [12] and cube computation problem [3]. However unlike these approaches, we are not looking to compute the entire cube. Also, our stream setting necessitates that we take into account the cost of computing intermediate group-bys as well.

More recently, many systems [9], [4], [6], [7], [14], [10], [5], [13] for processing continuous queries over data streams have employed resource sharing to achieve better scalability. Dobra et al. [11] consider the problem of sharing sketches for approximating the sizes of multiple join queries. Our work is most closely related to the research of Zhang et al. [20], which was done in the context of Gigascope [10]. Although our intermediate aggregates are conceptually similar to the phantoms of [20], we differ significantly from them in basic assumptions. We assume availibility of sufficient memory, which leads to fundamentally different cost models. Unlike

our heuristic, their query plan generation heuristics enumerate all possible phantoms (exponential in the number of attributes). Also, our heuristics allows queries with filters.

III. System Model

We consider a single stream consisting of an infinite sequence of tuples, each with *group-by* attributes $a_1, \ldots, a_m$ and a *measure* attribute a_0 . We are interested in answering a set of aggregate queries $\mathcal{Q} = \{Q_1, \ldots, Q_n\}$ defined over the stream of tuples. A typical aggregate query Q_i has 3 main components, listed below:

- *Aggregation.* This includes (1) the subset of group-by attributes on which aggregation is performed, and (2) the *aggregation operator* that is applied to the measure attribute values of aggregated tuples.
- *Filter.* This is essentially a boolean expression (containing boolean operators $\vee$ and $\wedge$) over attribute range conditions.
- *Period.* This is the time interval over which aggregation is performed.

Without loss of generality, we will assume that the measure attribute, the aggregation operator and the time period is same for all the aggregate queries. In the remainder of this paper, we will use the following notation: $\mathcal{A}$ to denote the collection of grouping attributes A_i for the queries; $\mathcal{F}$ to denote set of filters F_i; N to denote the number of stream tuples that arrive in time period T; σ_{F_i} to denote the selectivity of the filter condition F_i; $sz(A_i, F_i)$ to denote the size of the result after tuples filtered through F_i are aggregated on attributes in A_i [2]; $C_H(A_i)$ to denote the cost of hashing a tuple on its group-by attributes A_i and $C_F(F_i)$ to denote the cost of checking the filter condition F_i for the tuple. Now, the CPU cost for processing Q_i over time interval T is $N \cdot C_F(F_i) + N \cdot \sigma_{F_i} \cdot C_H(A_i)$ [3].

IV. Processing Aggregate Queries without Filters

We begin by considering queries without filters. Thus, each query $Q_i \in \mathcal{Q}$ is simply the group-by attributes A_i on which tuples are aggregated, and query processing costs are completely dominated by the hash function computation costs.

To reduce the number of hash operations, our scheme instantiates a few *intermediate* aggregates $B_1, \ldots, B_q$ and then uses them to compute the various A_is. These intermediate aggregates share the aggregate computations leading to an overall reduction in CPU cycles. Before we discuss what is the best set of intermediate aggregates to instantiate, we introduce the notion of aggregate trees. **Aggregate Trees.** An aggregate tree is a directed tree with (1) a special *root* node corresponding to the input stream, and (2) other nodes corresponding to aggregates. The aggregate for vertex v_i is denoted by $A(v_i)$. We use the special symbol $\top$ for $A(root)$ and require that all nodes in the tree are reachable from root.

A directed edge $\langle v_1, v_2 \rangle$ from vertex v_1 to vertex v_2 can be present in the tree only if the aggregate for v_1 covers the aggregate for v_2 (that is, $A(v_2) \subseteq A(v_1)$). Also, we assume that the root covers every other aggregate (since root corresponds to input stream). Each edge $\langle v_1, v_2 \rangle$ in the tree has an associated cost given by $sz(A(v_1)) \cdot C_H(A(v_2))$, which corresponds to the real cost of computing aggregates at node v_2. The cost of a tree is simply the sum of the costs of all its edges, which again corresponds to the real cost of computing aggregates associated with all nodes in the tree. The plan for a tree generates aggregates in two phases:

- *Real-time streaming phase.* Only the child aggregates of the root node are maintained as tuples are streaming in. Each streaming tuple is inserted into the hash tables of each of the root's children.
- *Periodic results output phase.* At time intervals of period T, the root's children are used to generate the remaining aggregates in the tree by performing a depth first traversal of the tree.

In short, an aggregate tree corresponds to a query plan capable of generating answers for every aggregate contained in the tree. Thus, our problem of finding a good query plan (with low hash computation costs) to process the aggregate queries in $\mathcal{A}$ reduces to the following:

Problem Statement. Given an aggregate set $\mathcal{A}$, compute the minimum-cost aggregate tree $\mathcal{T}$ that contains all the aggregates in $\mathcal{A}$. □

Theorem 1: The following decision problem is NP-hard[15]: Given an aggregate set $\mathcal{A}$ and a constant τ, is there an aggregate tree $\mathcal{T}$ with cost at most τ that also contains all the aggregates in $\mathcal{A}$? □

Randomized Heuristic. We now propose a randomized heuristic that adopts a global approach to compute a minimum-cost aggregate tree. Consider the graph containing a node for every possible aggregate (that is, every possible subset of group-by attributes), and also $\top$ for the input stream. In the aggregate graph, there is a directed edge from aggregate A to aggregate B iff A covers B, and the cost of the edge is $sz(A) \cdot C_H(B)$. Now, it is easy to see that computing the optimal aggregate tree $\mathcal{T}$ is nothing but computing a directed steiner tree (in the graph) that connects the root $\top$ to the set of aggregates $\mathcal{A}$. But, such a full aggregate graph contains exponential number of nodes 2^m nodes (a node for every subset of group-by attributes).

Our randomized heuristic (Algorithm 1) circumvents this exponential problem by employing randomization in successive iterations to construct a sequence of *partial* (instead of full) aggregate graphs. At the end of each iteration, variables T_{best} and S keep track of the current best aggregate tree and the aggregates contained in it, respectively. In each iteration, we pick a set random intermediate aggregates R (Steps 4–8), and construct a partial aggregate graph G on $S \cup R$. (G contains edges from an aggregate to every other aggregate that it covers.) We then invoke the dual-ascent directed steiner

[2]Both σ_{F_i} and $sz(A_i, F_i)$ can be estimated by maintaining random samples of past stream tuples and applying sampling-based techniques from [8]

[3]The computation cost for query Q_i on each stream tuple includes the cost of applying the filter F_i to the tuple, and then inserting the tuple into the hash table on attributes A_i if it satisfies F_i.

Algorithm 1 Randomized($\mathcal{A}$): Randomized heuristic for finding aggregate tree.

```
1: Initialize S = A ∪ {⊤};
2: for c1 iterations do
3:    R = ∅;
4:    for c2 iterations do
5:       Pick a random number r between 1 and n;
6:       Pick r aggregates at random from A and let B be their
         union;
7:       R = R ∪ {B};
8:    end for
9:    Let G be the partial aggregate graph on S ∪ R;
10:   T_best = Steiner(G, ⊤, A);
11:   Set S to the set of aggregates that appear in T_best;
12: end for
13: Return T_best;
```

heuristic of [19] to compute a minimum-cost tree connecting root $\top$ to aggregates in $\mathcal{A}$ in graph G. The user-defined parameters c_1 and c_2 determine the number of iterations and the number of random aggregates selected in each iteration, respectively.

V. Processing Aggregate Queries with Filters

We now turn our attention to aggregate queries with filters. Each query Q_i consists of a set A_i of grouping attributes and a filter F_i. In the presence of filters, we can reduce computational overhead by sharing filter evaluation among the various queries along with sharing of the aggregate computation.

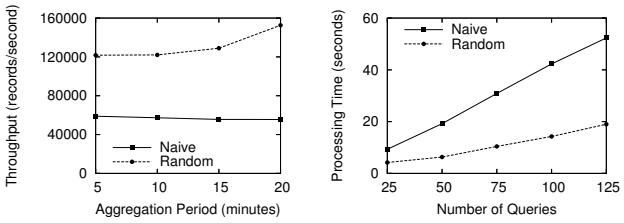
Aggregate Trees. In the presence of filters, each node of the aggregate tree is a (filter, grouping attributes) pair. The root node is special with a (filter, attributes) pair equal to $(\top, \top)$, and corresponds to the input stream. Here, $\top$ is a special symbol that contains all other filters and grouping attributes. In the aggregate tree, there can be an edge from a vertex v_1 to a vertex v_2 only if v_1 covers v_2, that is, the filter and group-by attributes of v_1 contain the filter and group-by attributes, respectively, of v_2. We assign a cost to each tree edge $\langle v_1, v_2 \rangle$ equal to the CPU cost of materializing the result tuples for v_2 using the tuples of v_1.

Randomized Heuristic. This is very similar to Algorithm 1, except that the steps 5-7 are replaced by the following:

1) Randomly select a subset of input query nodes from $\mathcal{Q}$.
2) Let v denote the union of (filters and group-by attributes of) the nodes selected above. Add v to R.
3) For every other node u in S that covers v, we add the following two additional nodes x and y to R: (a) Node x with v's filter, but u's group-by attributes. (b) Node y with v's group-by attributes, but u's filter.

VI. Performance Study

In order to gauge the efficacy of the tree-structured query plan generated by our randomized heuristic, we conducted experiments comparing our computation sharing approach with the traditional naive method of processing each query separately, on real-life NetFlow data sets (obtained from Abeline repository [1]). In all the cases, we observed that the plans generated by our randomized algorithm were able to process the records 2-3 times faster than the naive scheme (in which each agregation query is performed independently for each incoming stream tuple). Here, we show the results of two sets of experiments. Figure 1(a) shows the effect of varying the aggregation period on throughput(number of queries processed per second), when using the set of real-world queries from Cisco's Flow-Collector 3.0 [2]. Figure 1(b) shows the effect of varying the number of input queries on the throughput, when using the set of randomly generated queries. The latter experiment demonstrates the scalability of our heuristic as compared to the naive scheme.



(a) (b)

Fig. 1. Comparisons on Cisco FlowCollector queries and Random Queries

References

[1] Abilene Observatory Data Collections. http://abilene.internet2.edu/observatory/data-collections.html.

[2] Cisco CNS NetFlow Collection Engine Installation and Configuration Guide, 3.0. http://www.cisco.com/univercd/cc/td/doc/product/rtrmgmt/nfc/nfc_3_0/nfc_ug/index.htm.

[3] Sameet Agarwal et al. On the Computation of Multidimensional Aggregates. In *VLDB*, 1996.

[4] A. Arasu et al. STREAM: The Stanford Stream Data Manager. In *IEEE Data Engineering Bulletin*, 2003.

[5] A. Arasu et al. Resource Sharing in Continuous Sliding-Window Aggregates. In *VLDB*, 2004.

[6] Donald Carney et al. Monitoring Streams - A New Class of Data Management Applications. In *VLDB*, 2002.

[7] S. Chandrasekaran et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, 2003.

[8] Moses Charikar et al. Towards Estimation Error Guarantees for Distinct Values. In *PODS*, 2000.

[9] Jianjun Chen et al. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD*, 2000.

[10] C. Cranor et al. Gigascope: A Stream Database for Network Applications. In *SIGMOD*, 2003.

[11] A. Dobra et al. Sketch-Based Multi-query Processing over Data Streams. In *EDBT*, 2004.

[12] V. Harinarayan et al. Implementing Data Cubes Efficiently. In *SIGMOD*, 1996.

[13] Sailesh Krishnamurthy et al. On-the-Fly Sharing for Streamed Aggregation. In *SIGMOD*, 2006.

[14] S. Madden et al. Continuously Adaptive Continuous Queries over Data Streams. In *SIGMOD*, 2002.

[15] Kanthi Nagaraj et al. Efficient Aggregate Computation over Data Streams. Technical Report ITD-06-47360D, Bell Laboratories, Dec 2006.

[16] K.A. Ross et al. Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time . In *SIGMOD*, 1996.

[17] Prasan Roy et al. Efficient and Extensible Algorithms for Multi Query Optimization. In *SIGMOD*, 2000.

[18] T Sellis. Multiple Query Optimization. In *ACM TODS*, 1988.

[19] R. Wong. A Dual Ascent Approach for Steiner Tree Problems on a Directed Graph. In *Mathematical Programming*, 1984.

[20] Rui Zhang et al. Multiple Aggregations over Data Streams. In *SIGMOD*, 2005.