Query-Aware Partitioning for Monitoring Massive Network Data Streams

Theodore Johnson
AT&T Labs-Research
johnsont@research.att.com

S. Muthukrishnan
Rutgers University
muthu@cs.rutgers.edu

Vladislav Shkapenyuk
AT&T Labs-Research
vshkap@research.att.com

Oliver Spatscheck
AT&T Labs-Research
spatsch@research.att.com

ABSTRACT

Data Stream Management Systems (DSMS) are gaining acceptance for applications that need to process very large volumes of data in real time. The load generated by such applications frequently exceeds by far the computation capabilities of a single centralized server. In particular, a single-server instance of our DSMS, Gigascope, cannot keep up with the processing demands of the new OC-786 networks, which can generate more than 100 million packets per second. In this paper, we explore a mechanism for the distributed processing of very high speed data streams.

Existing distributed DSMSs employ two mechanisms for distributing the load across the participating machines: partitioning of the query execution plans and partitioning of the input data stream in a query-independent fashion. However, for a large class of queries, both approaches fail to reduce the load as compared to centralized system, and can even lead to an increase in the load. In this paper we present an alternative approach - query-aware data stream partitioning that allows for more efficient scaling. We present methods for analyzing any given query set and choose the optimal partitioning scheme, and show how to reconcile potentially conflicting requirements that different queries might place on partitioning. We conclude with experiments on a small cluster of processing nodes on high-rate network traffic feed that demonstrates with different query sets that our methods effectively distribute the load across all processing nodes and facilitate efficient scaling whenever more processing nodes becomes available.

Categories and Subject Descriptors

H.2.4 [**DataBase Management**]: Systems, Distributed databases

General Terms: Design, Performance

Keywords: Data streams, partitioning, query optimization

1. INTRODUCTION

Data stream management systems (DSMS) have increasingly become the tool of choice for applications that require sophisticated processing of large volumes of data in real time. Example applications include large scale sensor networks [3], and especially network monitoring [11][21]. The volume of data that needs to be processed in real time for such applications can easily exceed the resources available on a centralized server. For example, dual OC768 network links currently being deployed in the Internet backbone generate up to 2x40 Gbit/sec of traffic, which corresponds to roughly 112 million packets/sec. Even a fast 4GHz server can spend at most 26 cycles processing each tuple, which does not allow it to perform any meaningful processing short of incrementing few counters. Furthermore, this data load exceeds by an order of magnitude the throughput of fastest computer buses such as PCI-X and PCI-Express. Nevertheless, AT&T needs to monitor these links to ensure the health of its network. Given its successful application across the AT&T network, Gigascope is the natural candidate for the OC-768 monitoring platform.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.
Copyright 2008 ACM 978-1-60558-102-6/08/06...$5.00.

Distributed DSMSs attack the performance problem by spreading the load across a number of cooperating machines running independent DSMSs. Two commonly used techniques used to distribute the load across the participating machines are partitioning query plans into subplans to be executed in parallel (*query plan partitioning*) and splitting resource-intensive query nodes into multiple nodes working on subset of data feed (*data stream partitioning)* [9]. However, query plan partitioning fails to generate feasible execution plans if the original query plan contains one or more operator that are too "heavy" for a single machine (and at 100M packets/sec, most non-trivial operators are too heavy - memory copy cost alone reach 10Gbytes/sec). Most of the query plans used in network monitoring application are characterized by highly non-uniform resource consumption of different query nodes, which makes it impossible for query plan partitioning to evenly distribute the load.

The published data stream partitioning as implemented in DSMSs is done in query-independent fashion (e.g. partitioning tuples in random or round robin fashion) [9][20]. However, for a large class of queries such data stream partitioning fails to significantly reduce the load compared to centralized system and can even lead to an increase in the load.

Example. Let us consider an example of a network monitoring query computing traffic flows – summaries of packets between a source and a destination during a period of time. The group-by attributes are the source and destination IP address, the source and destination port, and the protocol, while the aggregates include the number of packets, the number of bytes transferred, start and stop times, and so on. These types of queries are popular in various network monitoring applications – from performance monitoring to detecting network attacks [14]. The SQL version of the query is shown below.

```
SELECT time,srcIP,destIP,srcPort,destPort,
       COUNT(*),SUM(len),
```

```
       MIN(timestamp),MAX(timestamp), …
FROM TCP
GROUP BY time,srcIP,destIP,srcPort,destPort
```

Suppose that data stream partitioning is applied in round robin fashion to evenly distribute input tuples among *n* machines that compute partially aggregated flows and send them to a central node that merges the partials flows and computes a final aggregation. It is easy to see that in the worst case, a single flow will result in *n* partial flows being computed and transmitted over the network to central aggregating node. In a more typical network monitoring scenario, the query is only interested in a subset of all flows. For example, suppose we want to monitor *attack* flows that do not follow TCP protocols and can frequently be differentiated by OR of the flags of the packets in the flow. In SQL we can write this query by adding a corresponding HAVING clause to the flow query (e.g. `HAVING OR_AGGR(flags)= ATTACK_PATTERN`). It is easy to see that none of the nodes performing local aggregation will be able to apply the HAVING clause to filter out regular flows. Due to the significant overhead involved in processing remote tuples as compared to local processing, the CPU and network link load on the final aggregation node can exceed the load on single node in centralized case, rendering the execution strategy infeasible.

For this example, a more reasonable approach for distributing the load among the participating machines is to partition the input data stream based on flows (e.g. evenly distributing entire flows). If such partitioning is utilized, all flows can be computed locally and filtered using the HAVING clause before being transmitted over the network. The problem of determining a good partitioning scheme for certain classes of individual relational queries (aggregations and equijoins) has been studied in the context of parallel relational databases [12][15]; however in the streaming environment existing approaches do not scale to complex query sets and massive data rates. Network monitoring applications generally run a large number of queries simultaneously (one of our applications runs 50 simultaneous queries); queries in turn may contain a number of different subqueries. Each of the subqueries might place different requirements for the way partitioning has to be done. These requirements can easily be in conflict with each other and it would not be always possible to satisfy all of them.

It is also not feasible to dynamically repartition the inputs to suit individual queries as is commonly done in parallel relational databases, since each such repartitioning puts the entire stream back into inter-node network without any data reduction, which greatly increases communication costs. Furthermore, splitting 80 Gbit/sec traffic requires specialized network hardware which is an order of magnitude more expensive than the computational hardware – we can only afford to partition the source once. In general, we need a partitioning mechanism that can automatically analyze an arbitrary complex query set and determine a single optimal initial stream partitioning scheme.

In order to incorporate the results of the analysis into distributed query optimization, we need to make the optimizer fully aware of the partitioning scheme used. However, we cannot make an assumption that the actual partitioning scheme used by the system is identical to the optimal one recommended by the analysis. Monitoring the 80Gbit/sec link requires specialized network equipment that can partition the data at line speeds. The currently available hardware for OC768 monitoring partitions each direction of OC768 stream into four 10Gbit Ethernet substreams that are sent to separate Gigascope servers. The network interface cards (NICs) specialized for 10GEth monitoring are typically capable of further partitioning the network stream into subinterfaces. We note that NICs for monitoring OC768 are not currently available, rendering query plan partitioning infeasible.

Even though the partition hardware is programmable using FPGAs and TCAMs, the limited number of available gates place restrictions on a type of partitioning can be performed in hardware. For example it is possible to implement partitioning based on TCP fields such as source or destination IP addresses, but accessing fields from higher-level protocols such as HTTP requires regular expression processing that is not currently feasible to do at OC768 speeds. Furthermore, it is not always possible to dynamically reconfigure the once optimal partitioning scheme every time the query workload changes. Therefore, we need a distributed query optimizer that is flexible enough to take advantage of any available partitioning.

The query-aware data stream partitioning mechanism proposed in this paper includes both an analysis framework for determining the optimal partitioning and a partition-aware distributed query optimizer that transforms the unoptimized query plan into a semantically equivalent query plan that takes advantage of existing partitions.

The contributions we make in this paper are as follows. We

1. Develop the concept of query-aware data stream partitioning for distributed stream processing.
2. Design and develop a framework for analyzing a set of queries to determine a partitioning strategy that would satisfy all the queries in a set.
3. Determine a set of query transformation rules to be used by query optimizer to take advantage of existing data stream partitioning.
4. Perform detailed experiments with a live cluster of stream processing nodes and show that our partitioning methods lead to highly efficient distributed query execution plans that scale linearly with the number of nodes.

2. RELATED WORK

The area of data streaming has received a lot of attention in recent years, with research efforts ranging from developing streaming algorithms to designing and building Data Stream Management Systems (DSMS). A number of research DSMSs are currently being developed, including Aurora/Borealis [1], TelegraphCQ [7], Gigascope [11] and many others. A number of currently active research projects focus on extending DSMS to enable scalable distributed stream processing [6][20]. Two main approaches used to distribute the load across the cooperating machines are query plan partitioning and data stream partitioning.

The load distribution mechanism used in Borealis [6] relies on query plan partitioning to balance the load on cooperating DSMSs. As we discussed earlier this approach is not feasible if a query plan contains one or more operators that are too "heavy" for a single machine. In addition to query plan partitioning, Borealis also employ fairly simple data stream partitioning mechanism called *box splitting*. However, partitioning is done in a query-independent manner and requires expensive processing of partial results generated by split query nodes.

The FLUX load partitioning operator used in TelegraphCQ DSMS [20] supports a variety of data stream partitioning schemes

including the hash-based strategy used in our paper. The primary goal of FLUX is to avoid imbalance in the load caused by the data scheme. To address the imbalance problem it uses an adaptive partitioning adjusted at runtime depending on observed data skew. The partitioning itself is still however operator-independent and suffers from excessive load on the node combining partial results.

The Grid Stream Data Manager (GSDM) described in [13] proposes an *operator-dependent windows split* strategy which partitions the input data stream in such a way that partial results can be inexpensively combined. The query writer is expected to manually provide specific *stream distribute/stream merge* routines for all query nodes eligible for optimization. The authors do not address the issue of automatic inference of an optimal splitting strategy for arbitrary query sets.

Hash-based partitioning has been studied in the context of parallel relational databases [12][15] to parallelize the execution of individual aggregation and equijoin queries. However, the main technique used – dynamic stream repartitioning - is not suitable for processing high-rate data streams as it puts the entire stream back into the network and makes the communication cost prohibitively expensive. Furthermore, the problem addressed in our paper is finding optimal partitioning for arbitrary complex query sets rather than individual queries.

Recent work on automating physical database design for relational databases [19] addresses the problem of choosing a database partitioning scheme that is optimal or close to optimal for a given query workload. The main idea of this work is to generate a large number of candidate partitions and perform a heuristic search to find the lowest cost partitioning scheme (using the cost estimates provided by IBM DB2 optimizer). Our approach is more principled, since it uses a much smaller set of possible partitions and then uses a cost model to reconcile the conflicts. We also are not as reliant on the quality of the cost model, which is very important for processing data streams with rapidly changing characteristics. Furthermore, our objective function – minimizing the maximum communication cost - is more appropriate for distributed stream processing.

3. QUERY-AWARE STREAM PARTITIONING OVERVIEW

The goal of the query-aware data stream partitioning mechanism is to distribute input tuples across multiple machines in such a way that maximizes the amount of data reduction that can be performed locally before shipping the intermediate results to a node that produces final results. We would call such partitioning *compatible* with a given query. In this section we will give a formal definition of partition compatibility and show how to infer a compatible partitioning scheme for two major classes of streaming queries – aggregations and joins.

3.1 Tumbling window query semantics

A primary requirement of a DSMS is to provide a way to unblock otherwise blocking operators such as aggregation and join. Different DSMSs take different approaches, but in general they provide a way to define a window on the data stream on which the query evaluation will occur at any moment in time. Two main approaches for defining a window on a stream are sliding windows (both time- and tuple-based) and tumbling windows. In streaming systems that rely on tumbling windows, one or more attributes of a data stream are marked as being ordered. Query evaluation windows are determined by analyzing how a query references the ordered attributes. For example, consider the following schema.

PKT(time *increasing*, srcIP, destIP, len)

The time attribute is marked as being ordered, specifically increasing. Then the following query computes the sum of the length of packets between each source and destination IP address for every minute

```
SELECT tb, srcIP, destIP, sum(len)
FROM PKT
GROUP BY time/60 as tb, srcIP, destIP
```

Similarly a join query on streams R and S must contain a join predicate such as R.tr=S.ts or R.tr/2=S.ts+1: that is, one which relates a timestamp field from R to one in S. An example of join query that combines the length of packets with matching IP addresses is shown below:

```
SELECT time, PKT1.srcIP, PKT1.destIP,
    PKT1.len + PKT2.len
FROM PKT1 JOIN PKT2
WHERE PKT1.time = PKT2.time and
    PKT1.srcIP = PKT2.srcIP and
    PKT1.destIP = PKT2.destIP
```

These kinds of queries use tumbling window semantics in which the window covers only the current epoch. Li et al. [17] show how tumbling windows can be used for the efficient evaluation of sliding window queries using panes. Therefore we will assume tumbling window semantics for our queries (except where otherwise noted) for simplicity.

3.2 Illustrative example

We illustrate the query-aware partitioning mechanism by working through an example query set. The first query (**flows**, denoted γ_1) computes simplified TCP traffic flows for every 60 second time epoch (for each communicating source and destination host it produces a number of packets sent between them). The higher-level aggregation query (**heavy_flows**, denoted γ_2) computes "heaviest" flows for each source (heaviest flows have the largest number of packets). Finally a self-join query (**flow_pairs**, denoted $\bowtie$) correlates heavy flows that span consequent time epochs. The corresponding SQL statements for both queries are shown below:

```
Query flows:
  SELECT tb,srcIP,destIP,COUNT(*) as cnt
  FROM TCP
  GROUP BY time/60 as tb,srcIP,destIP

Query heavy_flows:
  SELECT  tb,srcIP,max(cnt) as max_cnt
  FROM flows
  GROUP BY tb, srcIP

Query flow_pairs:
  SELECT S1.tb, S1.srcIP,
      S1.max_cnt,S2.max_cnt
  FROM heavy_flows S1, heavy_flows S2
  WHERE S1.srcIP = S2.srcIP and S1.tb
      = S2.tb+1
```

A query plan for execution of the queries is shown in Figure 1.

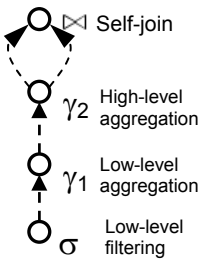

Figure 1: Sample query execution plan

1. *Which partitioning scheme is optimal for each of the queries in an execution plan?*

Intuitively, a lower-level aggregation query (node γ_1) will benefit the most from a partitioning which guarantees that no tuples with identical pair of attributes (srcIP, destIP) will end up in different partitions. Any partitioning that satisfies this properly would allow γ_1 to be evaluated in parallel on all participating hosts with linear scalability. Following a similar intuition, the query nodes γ_2 and self-join node ⋈ will benefit the most if the input stream was partitioning using (srcIP). Later in the section, we will formally define what requirements a partitioning scheme must satisfy and give inference rules to compute an appropriate partitioning for major classes of streaming queries.

2. *How to reconcile potentially conflicting partitioning requirements from different queries in a query set?*

As we have seen previously, query γ_1 will benefit mostly from partitioning based on attributes (srcIP, destIP), while the rest of the queries would prefer partitioning on (srcIP). Since it is (usually) not feasible to partition the input stream simultaneously in multiple ways, we need to reconcile partitioning requirements of different query nodes. It is easy to see that partitioning on (srcIP) can satisfy all queries in our sample query set. More generally, we will need an algorithm for inferring an optimal set of attributes to be used for partitioning for arbitrary complex query set. We will present such an algorithm in Section 4.

3. *How can we use the information about the scheme used for partitioning in distributed query optimizer?*

Assuming the input stream is partitioned as recommended by the query analysis, we can use this information to drive the distributed query optimizer. In our prototype implementation the optimizer works by invoking a set of partition-aware transformation rules on nodes of original query plan in bottom-up fashion.

In many real life applications the query writer does not have complete control over how the partitioning is done. As we mentioned in the introduction, processing capabilities of the hardware used for partitioning can place restrictions on the partitioning scheme. For example we could have hardware that can only split the input stream based on (destIP). The query optimization framework needs to be flexible enough to maximally take advantage of any partitioning, even if it is different from the optimal one. An example distributed query plan produced by the optimizer under the assumption that partitioning is done based on (destIP) is shown in Figure 2.

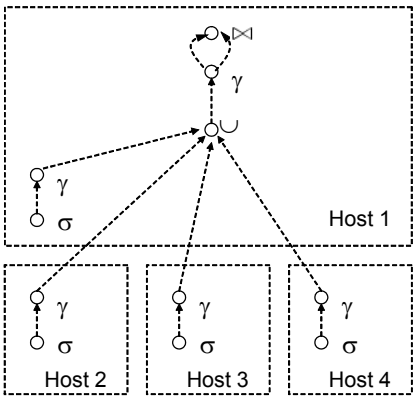

Figure 2: Optimized query execution plan

3.3 Hash-based stream partitioning

The main goal of any stream partitioning scheme is to distribute tuples evenly across multiple distributed nodes in such a way that load is evenly spread across all nodes. There are multiple ways in which such scheme could be implemented, but one of the simplest can be done by hashing selected set of tuple attributes. Let *A* be a set of tuple attributes (the *partitioning set*), *H(A)* a hash function returning 0 .. *R*, and *M* the number of desired partitions. Then, a tuple falls into parititon *i* if

$$i*R/M \le H(A) \le (i+1)*R/M$$

For many query sets, it is beneficial not to restrict ourselves to using singleton tuple attributes and instead allow grouping sets to include arbitrary scalar expression involving tuple attributes. For example one choice of partitioning set for could be (srcIP & 0xFFF0, destIP) which will effectively partition tuples based on subnet that srcIP belongs to. Let $sc\_exp_i(attr_i)$ represent a scalar expression. For the rest of the paper we will only assume more general definition of partitioning set:

$$(sc\_exp_1(attr_1), sc\_exp_2(attr_2), \ldots, sc\_exp_n(attr_n))$$

3.4 Partition compatibility

The choice of the partition set critically impacts the ability of the query optimizer to reorganize the query plans for distributed evaluation. Consider the following aggregation query that computes simple network flows:

```
SELECT tb, srcIP, destIP, sum(len)
FROM PKT
GROUP BY time/60 as tb, srcIP, destIP
```

It is easy to see that partitioning using partitioning set (time/60, srcIP, destIP) allows each host to execute the aggregation query locally on corresponding partition with no further aggregation necessary. A partition-aware query optimizer can replace the aggregation query by stream union of the identical queries running on individual partitions. However, if (srcIP, destIP, srcPort, destPort) is used as partitioning set, this optimization would not be possible. We will capture the notation of "optimizer-friendly" partitioning set in the following definition: Partitioning set P is **compatible** with a query Q if for every time window, the output of the query is equal to a stream union of the output of the Q running on all partitions produced by P.

An example of such compatible partitioning set for the query above is {(time/60)/2, srcIP & 0xFFF0, destIP & 0xFF00). An example of an incompatible grouping set for the query above is

{time, srcIP, destIP} (since tuples belonging to the same 60 second epoch will end up in different partitions).

In the following sections we will give the rules for inferring the compatible partioning sets for two major classes of streaming queries - aggregations and joins. Other types of streaming queries (selection, projection, union) are always compatible with any partitioning sets and therefore we will omit the discussion of these query types.

3.5 Inference of partitioning sets for streaming queries

The definition of partition compatibility given in the previous section is very generic and does not directly tell us how to infer the partitioning set for a given query. In this section, we give equivalent definitions of query compatibility for both aggregation and join queries that can be directly applied to compute the partitions.

3.5.1 Dealing with temporal attributes

One issue that needs to be considered when selecting a partitioning set compatible with a given query is whether to include the temporal attributes. Selecting the temporal attribute in a partitioning set will effectively change the allocation of groups to processors whenever the time epoch changes. This property could be desirable if we want to avoid bad hash functions that fail to uniformly spread the load across the participating machines. However, for sliding window queries that use pane-based evaluation [17], changing the group allocation in the middle of a window will lead to incorrect query results. Even for tumbling window queries a temporal attribute is generally not a good choice for load-balancing partitioning unless it is extremely (nanosecond) fine grained, as tuples correlated in time tend to have very highly correlated values of the temporal attribute. For this reason we will exclude the temporal attributes from further consideration.

3.5.2 Partitioning sets for aggregation queries

In its general form an aggregation query has the following format:

```
SELECT expr1, exp2, ... ,exprn
FROM STREAM_NAME
WHERE tup_predicate
GROUP BY temp_var, gb_var1, ... ,
   gb_varm
HAVING group_predicate
```

We only consider a subset G of these groupby variables ($gb\_var_1, \ldots, gb\_var_m$) that can be expressed as a scalar expression involving an attribute of one of the source input streams (ignoring grouping variables that are, e.g., results of aggregations computed in lower-level queries). Then, any compatible partitioning set for aggregation query Q will have the following form:

$$\{se(gb\_var_1), \ldots, se(gb\_var_n)\}$$

where se(x) is any scalar expression involving x. Given that there is an infinite number of possible scalar expression, every aggregation query has an infinite number of compatible partitioning sets. Furthermore any subset of a compatible partitioning set is also compatible.

3.5.3 Partitioning sets for join queries

We will consider a restricted class of join queries, namely two-way equi-join queries that use the semantics of tumbling windows. The general form of such query has the following format:

```
SELECT expr1, expr2, ... ,exprn
FROM STREAM1 AS S {LEFT|RIGHT|FULL}
 [OUTER] JOIN STREAM2 as R
WHERE STREAM1.ts = STREAM1.ts and
  STREAM1.var11 = STREAM2.var21 and ...
  STREAM1.var1k = STREAM2.var2k and
  other_predicates;
```

For ease of the analysis we will only consider join queries whose WHERE clause is in Conjunctive Normal Form (CNF) in which at least one of the CNF terms is equality predicate between the scalar expressions involving attributes of the source streams. Let J be a set of all such equality predicates $\{ se(R.rattr_1) = se(S.sattr_1), \ldots, se(R.rattr_n) = se(S.sattr_n)\}$. As with aggregation queries, we will only consider scalar expressions involving attributes of the source input streams. Then we can compute the partitioning sets for both streams S and R using

$$Partn\_R = \{ se(R.attr_1), \ldots se(R.attr_n) \}$$
$$Partn\_S = \{ se(S.attr_1), \ldots, se(S.attr_n) \}$$

respectively. It also follows that join query is compatible with any non-empty subset of its partitioning set. Since it is not feasible to partition the input stream simultaneously in multiple ways, *Partn_R* and *Partn_S* will need to be reconciled to compute a single partitioning scheme.

4. PARTITIONING FOR QUERY SETS

Data stream management systems are expected to run a large number of queries simultaneously; queries in turn may contain a number of different subqueries (selections, aggregations, unions, and joins). Each of the subqueries might place different requirements on partitioning set to be compatible with it.

Example: Consider the following query set:

```
Query tcp_flows:
  SELECT tb, srcIP, destIP, srcPort,
destPort, COUNT(*), SUM(len)
  FROM TCP
  GROUP BY time/60 as tb, srcIP, destIP,
   srcPort, destPort

Query flow_cnt:
  SELECT tb, srcIP, destIP, count(*)
  FROM tcp_flows
  GROUP BY tb, srcIP, destIP
```

Query `tcp_flows` computes the number of packets and total number of bytes sent in each flow; query `flow_cnt` computes a number of distinct flows active during the time epoch for each pair of communication hosts.

Based on our analysis for individual queries, `tcp_flows` is compatible with partitioning set of the form of {sc_exp(`srcIP`), sc_exp(`destIP`), sc_exp(`srcPort`), sc_exp(`destPort`)} or any of its non-empty subsets. Query flow_cnt, on other hand, requires the input stream to be partitioned using {sc_exp(`srcIP`), sc_exp(`destIP`)} to be compatible with distributed optimization. Considering both partitioning sets we can infer that partitioning based on {sc_exp(`srcIP`),

sc_exp(destIP)} will be compatible with *both* queries. A similar inference is required for join queries whose child queries have different compatible partitioning sets.

In what follows we present our analysis framework that infers the compatible partitioning set for arbitrary set of streaming queries. Our framework makes a simplifying assumption that all of the source input streams processed by a query set are partitioned using the same partitioning set. Expanding the analysis algorithms to handle different partitioning schemes for different input stream is part of planned future work.

4.1 Reconciling partitioning sets

Previously we discussed the need to reconcile the different requirements two queries might have for a compatible grouping set to generate a new grouping set compatible with both queries. We abstract this issue using Reconcile_Group_Sets(), defined as follows:

> **Def.** *Given two partitioning set definitions PS1 for query Q1 and PS2 for query Q2, Reconcile_Partn_Sets() is defined to return the largest partitioning set Reconciled_PS such that both Q1 and Q2 are compatible with partitioning using a set Reconciled_PS. The empty set is returned if no such Reconciled_PS exists.*

Considering a simple case of partitioning sets consisting of just the stream attributes (no scalar expressions involved), ReconcilePartn_Sets() returns the intersection of the two partitioning sets. For example Reconcile_Partn_Sets({srcIP, destIP}, {srcIP, destIP, srcPort, destPort},) is the set { srcIP, destIP }. For a more general case of partitioning sets involving arbitrary scalar expressions, Reconcile_Partn_Sets uses scalar expression analysis to find "least common denominator". For example

```
Reconcile_Partn_Sets (
  {sc_exp(time/60), sc_exp(srcIP), sc_exp(destIP)},
  {sc_exp(time/90}, sc_exp(srcIP & 0xFFF0)} )
is equal to a set
{sc_exp(time/180, sc_exp(srcIP & 0xFFF0)}.
```

The Reconcile_Partn_Sets function can make use of either simple or complex analysis based on the implementation time that is available. A full discussion is beyond the scope of this paper, but we expect that the simple analyses used in the example will suffice for most cases.

4.2 Algorithm for computing a compatible partitioning set

We represent a set of streaming queries as a Directed Acyclic Graph (DAG) of streaming *query nodes*, where each query node is a basic streaming query (selection/projection, union, aggregation, and join). Even though most real systems also use more complicated streaming operators, we can always express them using a combination of basic query nodes. Note that based on the analysis in Section 3, we know how to compute compatible partitioning sets for all individual query nodes.

Computing a compatible partitioning for an arbitrary query set essentially requires reconciling all the requirements that all nodes in the query graph place on compatible partitioning sets. A simplified implementation of the procedure of computing compatible set *PS* for a DAG with *n* nodes would look the following way:

1. For every query node Q_i in a query DAG, compute the compatible partitioning set $PS(Q_i)$.
2. Set $PS = PS(Q_1)$.
3. For every $i \in [1 \text{ to } n]$, set PS = *Reconcile_Partn_Sets(PS,* $PS(Q_i)$).

Unfortunately, for many realistic query sets we would expect the resulting partitioning set *PS* to be empty due to conflicting requirements of different queries. A more reasonable approach would be to try to satisfy a subset of nodes in a query DAG in order to minimize the total cost of the query execution plan. There are a variety of different cost models that can be used to drive the optimization; in this paper we will use a simple model that approximates a maximum network load on single node.

4.2.1 Cost model for streaming query nodes

The cost model that we are going to use in this paper defines a cost of query execution plan to be the maximum amount of data a single node in query execution plan is expected to receive over the network during one time epoch. The intuition behind this model is trying to avoid query plans that overload a single host with excessive amounts of data sent from query nodes residing on different hosts.

Let *R* be the rate of the input stream on which the query set is operating, and *PS* be a partitioning set. For each query node *Qi* in a potential query execution plan we define the following variables:

- selectivity_factor (*Qi*). The selectivity factor estimates the expected ratio of the number of output tuples to the number of input tuples *Qi* receives during one epoch.
- out_tuple_size (*Qi*). Expected size of the output tuple produced by *Qi*.
- We recursively define input_rate (*Qi*) to be *R* if *Qi* is a leaf node and to be the sum of all output_rate (*Qj*) s.t. *Qj is a child of Qi*.
- output_rate (*Qi*) = (input_rate (*Qi*) / in_tuple_size (*Qi*))* selectivity_factor (*Qi*) * out_tuple_size (*Qi*).

We define the cost(*Qi)* in the following way:

- 0 if it processes only local data
- input_rate (*Qi*) if *Qi* is incompatible with *PS*
- output_rate (*Qi*) if *Qi* is compatible with *PS*

The intuition behind this cost formula is that an operator partitioned using a compatible partitioning set only needs to compute the union of the results produced by remote nodes, and therefore the rate of the remote data it is expected to receive is equal to its output rate.

Finally, we define the cost of the query plan *Qplan* given partitioning *PS* cost(*Qplan, PS*) to be the max cost(*Qi)* for all *i*. The intuition behind this formula is trying to avoid overloading a single node rather than minimizing average load.

4.2.2 Computing an optimal compatible partitioning set

We now describe an algorithm for computing an optimal partitioning set for arbitrary query sets. The algorithm takes a query DAG as an input and produces a partitioning set that minimizes the cost of the query execution plan. The basic idea is to enumerate all possible compatible partitioning sets using dynamic programming to reduce the search space. The outline of the algorithm is given below:

1. For every query node Q_i in a query DAG, compute its compatible partitioning set *PS(i)* and cost(*Qplan, PS(i)*). Add non-empty *PS(i)* to a set of partitioning candidates.
2. Set *PS* to be *PS(i)* with minimum cost(*Qplan, PS(i)*).
3. For every candidate pair of partitioning sets *PS(i)* and *PS(j)* compute compatible partitioning set *PS(i, j) = Reconcile_Partn_Sets(PS(i), PS(j))* and cost(*Qplan, PS(i,j)*). Add non-empty *PS(i, j)* to a set of candidate pairs.
4. Set *PS* to be *PS (i, j)* with minimum cost(*Qplan, PS(I, j)*).
5. Similarly to previous step, expand candidate pairs of partitioning sets to candidate triples and compute corresponding reconciled partitioning sets and minimum cost.
6. Continue the iterative process until we exhaust the search space or end up with an empty list of candidates for the next iteration.

Since it is impossible for a partitioning set to be compatible with a node and not to be compatible with one of the node predecessors, we can use the following heuristics to further reduce the search space:

- Only consider leaf nodes for a set of initial candidates
- When expanding candidate sets only consider adding a node that is either an immediate parent of a node already in the set or is a leaf node.

5. QUERY PLAN TRANSFORMATION FOR A GIVEN PARTITIONING

The query analysis framework presented in Section 4 provides a way to automatically infer the optimal partitioning scheme for a given set of streaming queries. In order to incorporate the results of the analysis into distributed query optimization, we need to make the optimizer fully aware of the partitioning scheme used. We implemented all partition-related optimizations as a set of transformation rules invoked by the query optimizer on compatible query nodes. All query transformation rules that we use work by replacing a qualifying subtree in query execution plan by equivalent optimized version (under the assumption that the input stream was partitioned using a compatible partitioning method).

As discussed earlier, we cannot assume that the partitioning scheme used by the actual system is identical to the optimal one recommended by the query analyzer. Therefore, the distributed query optimizer needs to take advantage of any partitioning that used by the system, even if it differs from the optimal one.

5.1 Algorithm for performing partition-related query plan transformations

Our algorithm for transforming query execution plans based on available partitioning information consists of the following two phases:

Build partition-agnostic query execution plan

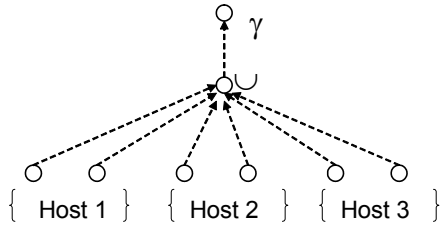

Figure 3: Partition-agnostic query execution plan

Let *S* be the partitioned source input stream consumed by a query set, $S = \cup Partn_i$. We construct a partition-agnostic query plan by creating an additional merge query node that computes a stream union of all the partitions and making all query nodes that consume *S* read from the merge node. Since each host might have multiple CPUs/Cores, we can allocate multiple partitions to each participating host depending on the host capabilities. An example of a partition-agnostic plan for an aggregation query is shown in Figure 3. In this example an input stream *S* is split into 6 different partitions, with 2 partitions assigned to each host.

Even though such a query execution plan is clearly inefficient since it forces all the partitioned streams to be shipped to a single host before performing any processing, in the absence of any information about partitioning scheme used it is often the only feasible plan.

Perform query plan transformation in bottom-up fashion

All transformation rules that we use for partition-related query optimization consist of two procedures: *Opt_Eligible()* and *Transform()*. *Opt_Eligible()* is a Boolean test that takes a query node and returns true if it is eligible for partition-related optimization. *Transform()* replaces the node that passed *Opt_Eligible()* test by equivalent optimized plan. The pseudo code for query optimizer is given below:

1. Compute a topologically sorted list of nodes in the query DAG $Q_1, Q_2, \ldots, Q_n$ starting with the leaf nodes.
2. For every $i \in [1 \text{ to } n]$
   If Opt_Eligible(Q_i)
   Transform(Q_i ,Partitiong_Info)

Performing the transformation in a bottom-up fashion allows us to easily propagate the transformation compatible leaf nodes through the chain of compatible parent nodes. In the following section we will give a detailed description of the implementation of *Opt_Eligible()* and *Transform()* for all major classes of query nodes – aggregations, joins and selection/projection.

5.2 Transformation for aggregation queries

The *Opt_Eligible()* procedure for an aggregation query *Q* and partitioning set *PS* returns true if the following conditions are met:

- query *Q* has a single child node *M* of type merge (stream union)
- each child node of *M* is operating on single partition consistent with *PS*
- *Q* is compatible with *PS*
- *Q* is the only parent of *M*

The last requirement is important to prevent the optimizer from removing the merge nodes that are used by multiple consumers. An example of a query node that stultifies all of the conditions required by *Opt_Eligible()* is shown Figure 4.

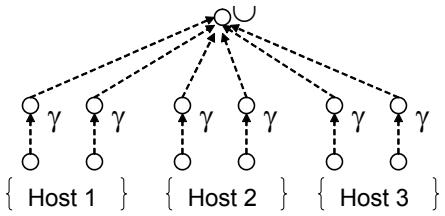

Figure 4: Aggregation transformation for compatible nodes

5.2.1 Transformation for compatible aggregation queries nodes

The main idea behind the *Transform()* procedure for eligible aggregation query *Q* is to push the aggregation operator below the merge *M* and allow it to execute independently on each of the partitions. For each of the inputs of *M* we create a copy of *Q* and push it below the merge operator. The resulting optimized query execution plan is shown in Figure 4.

The correctness of the transformation follows directly from out definition of partition compatibility. Note, that data is fully aggregated before being sent to central node and does not require any additional processing.

5.2.2 Transformation for incompatible aggregation queries

For many aggregation queries that fail the *Opt_Eligible()* test we can still do better than use the default partition-agnostic query execution plan. The main idea behind the proposed optimization is the concept of partial aggregates. This idea is widely used in a number of streaming database engines [9][10], sensor networks [3][8] and traditional relational databases [16]. We illustrate this idea on a query that computes a count of number of packets sent between pairs of hosts:

```
Query tcp_count:
   SELECT time, srcIP, destIP, srcPort,
COUNT(*)
   FROM TCP
   GROUP BY time, srcIP, destIP, srcPort
```

We can split **tcp_count** into two queries called sub- and super-aggregate:

```
Query super_tcp_count:
   SELECT time, srcIP, destIP, srcPort,
SUM(cnt)
   FROM sub_tcp_count
   GROUP BY time, srcIP, destIP, srcPort

Query sub_tcp_count:
   SELECT time, srcIP, destIP, srcPort,
COUNT(*) as cnt
   FROM TCP
   GROUP BY time, srcIP, destIP, srcPort
```

All the SQL's built-in aggregates can be trivially split in a similar fashion. Many commonly used User Defined Aggregate Functions (UDAFs) can also be easily split into two components as was suggested in [10]. Note that we can push all the predicates in the query's WHERE clause to sub-aggregates, but all predicates in HAVING clause need complete aggregate values and therefore must be evaluated in super-aggregate. The query execution plan produced by this optimization is shown is Figure 5.

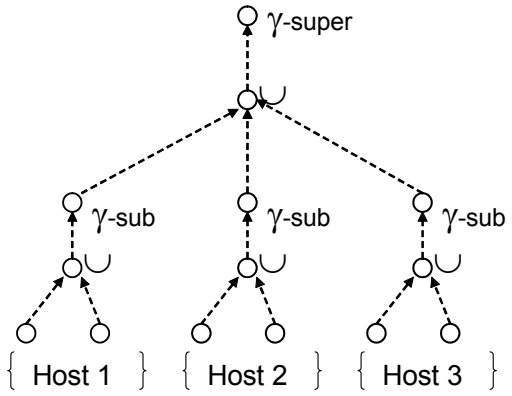

Figure 5: Aggregation transformation for incompatible nodes

5.3 Transformation for join queries

In this section we will only consider two-way join queries, since all multi-way joins can be easily expressed by combination of two-way joins. The *Opt_Eligible()* procedure for a join query *Q* and partitioning set *PS* returns true if the following conditions are met:

- query *Q* has a two children nodes *M1* and *M2* of type merge (stream union)
- each child node of *M1 and M2* is operating on single partition consistent with *PS*
- *Q* is compatible with *PS*
- *Q* is the only parent of *M1 and M2*

An example query execution plan that satisfies *Opt_Eligible()* test is shown in Figure 6.

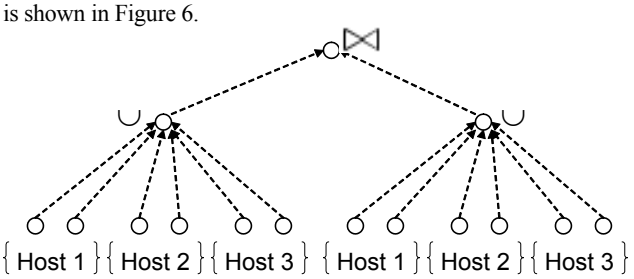

Figure 6: Original query execution plan

The main idea behind the *Transform()* procedure for an eligible join query *Q* is to perform pair-wise joins for each of partition of input stream. This is accomplished by creating a copy of join operator and pushing it below the child merges. The left side partitions that do not have matching right side partitions and similarly unmatched right side partitions are ignored for inner join computations. For outer join computations, unmatched partitions are passed through special projection operator that adds appropriate NULL values needed by outer join. The output tuples produced by the projection operator are then merged with the rest of the final results. The resulting optimized query execution plan for inner-join query is shown in Figure 7.

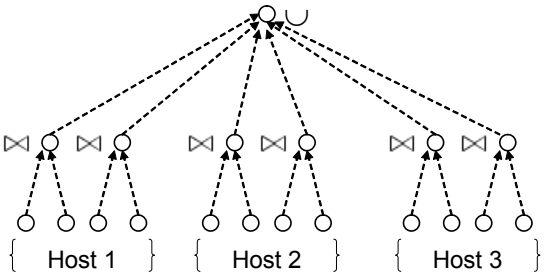

Figure 7: Join transformation for compatible nodes

5.4 Transformations for selection/projection queries

Selection/projection queries are always compatible with partition optimization and can be trivially pushed below child merge operators. Even though this transformation does not necessarily provides significant performance improvements, it is critical to ensure that partition-related optimization propagate further up the query tree.

6. EXPERIMENTAL EVALUATION

In this section we present the result of experimental evaluation of query-aware partitioning in the context of the AT&T Gigascope streaming database [11]. We augmented Gigascope's query analysis framework to add support for stream partitions. We also modified the query optimizer to fully implement all query transformation rules and thus support partitioned evaluation of distributed queries.

All the experiments were conducted by replaying a one-hour trace of network packets and feeding it to a cluster of Gigascope nodes (the OC-768 monitor being under construction at the time of the writing). The trace was obtained by combining four different one-hour traces captured concurrently using four data center taps. Each network tap captured two separate streams of packets for each traffic direction, each direction receiving approximately 100,000 packets/sec (about 400 Mbits/sec). We used a cluster of four dual core 3.0GHz Intel Xeon servers (2 cores per/CPU) with 4 GB of RAM running Linux 2.4.21. Servers were equipped with dual Intel(R) PRO/1000 network interface cards and were connected via Gigabit Ethernet LAN.

The goal of the experiments was to compare the performance of partition-agnostic query evaluation strategy with alternative strategies that take advantage of stream partitioning.

6.1 Partitioning for simple aggregation queries

In this experiment, we observe how the performance of an aggregation query is affected by the choice of partitioning strategy. The query used in the experiment computes network traffic flows returning only suspicious flows that do not follow the TCP protocol (i.e. have an abnormal value of OR aggregate of TCP flags). In our packet trace, suspicious flows accounted for about 5% of the total number of flows. The corresponding GSQL statement for the query is shown below.

```
SELECT tb, srcIP, destIP, srcPort,
   destPort, OR_AGGR(flags) as orflag,
   COUNT(*), SUM(len)
FROM TCP
GROUP BY time as tb, srcIP, destIP,
   srcPort, destPort
HAVING OR_AGGR(flags) = #PATTERN#
```

We varied the number of machines in the cluster from 1 to 4 while varying the number of stream partitions from 2 to 8 respectively. In each experiment, we assign two partitions to each host to make better use of multiple processing cores. We will denote the host assigned to execute a root of the query tree as the *aggregator* node and to the rest of the nodes as leaf nodes.

We compared three different system configurations:

a) Naïve – data stream is partitioned in a round robin fashion, hosts pre-aggregate the data within each partition before sending it for final aggregation.
b) Optimized – data stream is partitioned round robin, but all the host's data (from multiple partitions) is partially aggregated before being sent for final aggregation
c) Partitioned – data stream is partitioned using optimal compatible partitioning set (`srcIP, destIP, srcPort, destPort`)

Note that naïve configuration matches query-independent partitioning that is performed by current state of the art DSMS. In a course of the experiments we observed that all three configurations are very effective at reducing the CPU load on leaf nodes. The load on each host drops from 80.4% to 23.9% (combined CPU utilizations of the leaf nodes) as the number of hosts grows from 1 to 4. However, the load on the aggregator node shows completely opposite behavior. The results of the measuring the load on aggregator node are shown in Figure 8.

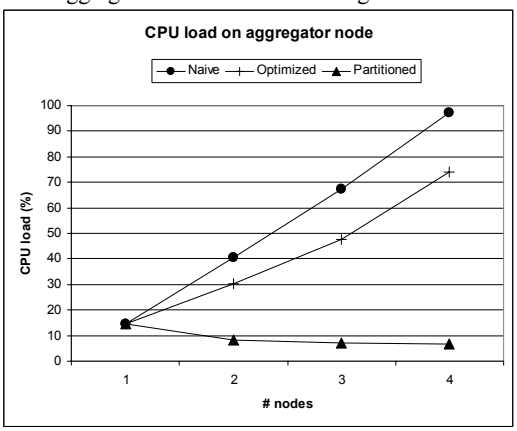

Figure 8: CPU load on aggregator node

As we can observe from the graphs for Naïve configuration, the load grows linearly with a number of hosts and reaches almost 100% CPU utilization for 4 machines. At this point the system is clearly overloaded and starts dropping input tuples. Enabling partial aggregation helps reduce the load by 20-22% but overall trend of linear growth continues. The configuration using partitioning set recommended by the query analyser, on other hand, reduces the load on both aggregator and leaf nodes and enables true linear scaling.

In addition to the CPU load on aggregator nodes, we also measured network load that query evaluation places on aggregator node. The results of the experiments are shown in Figure 9.

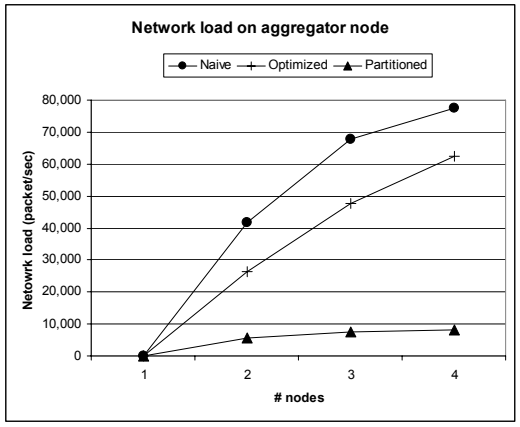

Figure 9: Network load on aggregator node

As we can see from the graph, both partition-agnostic configurations suffer from transmitting the same partial flows to aggregator multiple times and exhibit linear growth in the network load. The slope of the Partitioned configuration is nearly flat with maximum network load limited by the cardinality of the query output.

We observed similar scaling behavior while studying the performance of join queries on naively and optimally partitioned configurations. We observed similar scaling behavior while studying the performance of join queries on naive and optimally partitioned configurations.

6.2 Partitioning for query sets

In our second set of experiments, we study the performance of a query set consisting of independent aggregation and self-join queries. The aggregation query computes the statistics for packets sent between the source subnets and destination hosts (grouping attributes are (srcIP & 0xFFF0, destIP)). The self-join query computes delays between consecutive TCP packets within the same traffic flow. This particular query is often used by network analysis for monitoring TCP session jitter. The optimal partitioning set for aggregation query is (srcIP & 0xFFF0, destIP), while for the join query it is (srcIP, destIP, srcPort. destPoirt). We model a scenario where the restrictions of the partitioning hardware do not allow us to partition the data in a way that is compatible with both queries. According to the cost model presented in Section 4, the optimal partitioning set is (srcIP & 0xFFF0, `destIP)`, which is compatible only with the aggregation query.

We compared three different system configurations:

a) Naïve – data stream is partitioned in a round robin fashion
b) Partitioned (suboptimal) – the data stream is partitioned using the suboptimal partitioning set (srcIP, destIP, srcPort. destPoirt) compatible with the join query
c) Partitioned (optimal) – the data stream is partitioned using the optimal compatible partitioning set (srcIP & 0xFFF0, `destIP)`.

We varied the number of machines in the cluster in the cluster from 1 to 4 with 2 partitions assigned to each host. The results of the measuring the load on aggregator (root of the query tree) node are shown in Figure 10.

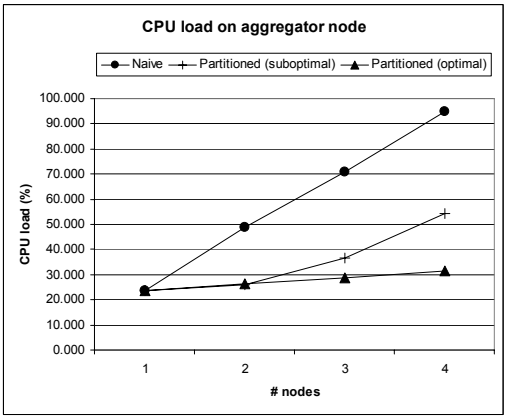

Figure 10: CPU load on aggregator node

As we can see from the graph, the load rises rapidly for the partitioning-agnostic scheme and reaches 95% CPU utilization for 4 participating hosts. Suboptimally partitioned configuration compatible with the join query reduces the load by 43-47% reaching 54% utilization for a 4 host configuration. However, the linear load growth trend is still present due the fact since the workload is dominated by incompatible aggregation query. The load growth curve for the optimal partitioning scheme is much flatter, reducing the load to 31% for 4 host configuration.

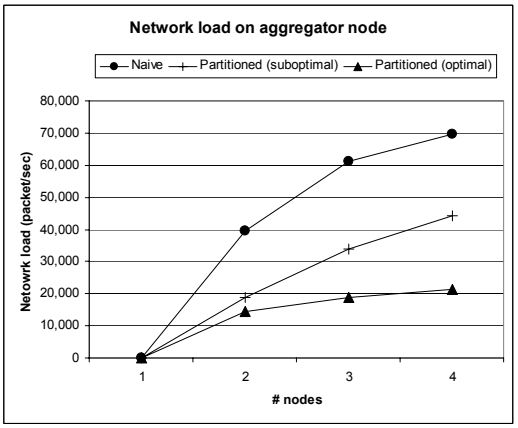

Figure 11: Network load on aggregator node

Figure 11 shows the results of the experiments measuring the network load on the aggregator node. Unable to perform any significant load reduction, the partition-agnostic configuration exhibits an almost linear increase in the network load. Suboptimal configuration, on other hand, evaluates all the joins locally and reduces the network load on aggregator node by 36-52% as the number of participating nodes increases to 4. The optimal configuration has an almost flat growth and effectively reduces the network load by 64-70% depending on number of hosts. These experiments demonstrate that our cost model correctly identifies the dominant queries in a query set and computes the globally optimal partitioning.

6.3 Partitioning for complex queries

In the final set of experiments, we use a more complex query set involving multiple related aggregation and join queries. This query set is identical to the one we used in Section 3 to illustrate query-aware partitioning framework. The corresponding GSQL statements for the queries are shown below.

`Query` **`flows:`**

```
SELECT tb,srcIP,destIP,COUNT(*) as cnt
FROM TCP
GROUP BY time/60 as tb,srcIP,destIP
```

```
Query heavy_flows:
SELECT  tb,srcIP,max(cnt) as max_cnt
FROM flows
GROUP BY tb, srcIP
```

```
Query flow_pairs:
SELECT S1.tb, S1.srcIP,
    S1.max_cnt,S2.max_cnt
FROM heavy_flows S1, heavy_flows S2
WHERE S1.srcIP = S2.srcIP and S1.tb
    = S2.tb+1
```

We compared four different system configurations:

d) Naïve – data stream is partitioned in a round robin fashion
e) Optimized – data stream is partitioned round robin, all the host's data is partially aggregated before being sent for final aggregation
f) Partitioned (partial) – the data stream is partitioned using the suboptimal partitioning set (`srcIP, destIP`)
g) Partitioned (full) –the data stream is partitioned using the optimal compatible partitioning set (`srcIP`)

Note that in the Partitioned (partial) configuration, only query **flow** is compatible with partitioning set while the rest of the queries are incompatible. The query plan generated by the optimizer for suboptimal partitioning is shown in Figure 12.

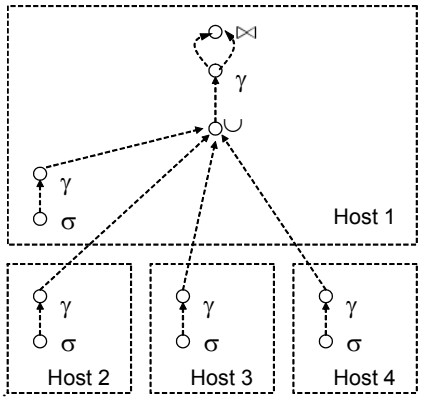

Figure 12: Plan for partially compatible partitioning set

As in previous experiments we varied the number of machines in the cluster from 1 to 4 with 2 partitions assigned to each host.
Since the CPU load on leaf nodes followed the same patterns as in previously shown experiments, we concentrate on discussing the load on aggregator node. The results of the measuring the CPU load on aggregator (root of the query tree) node are shown in Figure 13.

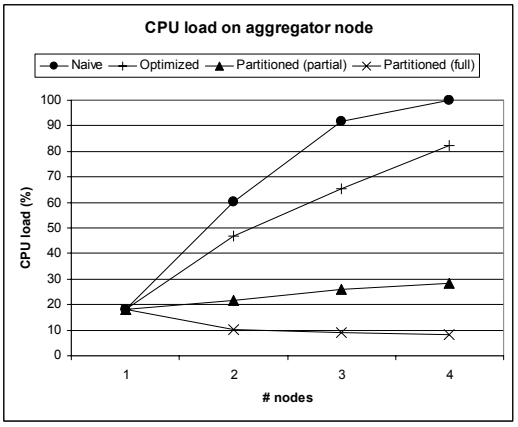

Figure 13: CPU load on aggregator node

As we can observe from the graphs for the Naïve configuration, the load on aggregator node grows linearly with a number of hosts. For a four machine configuration, the system overloaded and is forced to drop tuples from the input stream. The optimized configuration with partial aggregation enabled reduces the load by 23-24% reaching 82% utilization for a 4 host configuration. However, the linear load growth trend is still present and adding one more machine to the cluster will lead to the aggregator overload.

The load for the partially compatible configuration exhibits a nearly flat growth curve, primarily due to the fact that the most expensive query in a query set **flows** fully takes advantage of the compatible partitioning set. The load on aggregator node reaches only 18.4% which leaves a lot of room for further increase in the number of hosts. Finally, the fully compatible configuration exhibits true linear scaling, with the load on the aggregator node reaching 8.4% for a 4 machine setup.

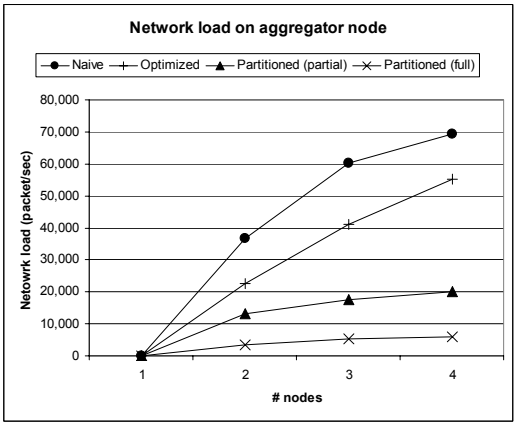

Figure 14: Network load on aggregator node.

Figure 14 shows the results of the experiments measuring the network load on the aggregator node. Here we observe the trends similar to previous experiments. Both Naïve and Optimized configuration with partial aggregates suffer from transmitting duplicate partial flows to the aggregator node and exhibit linear load growth. The partially and fully compatible configurations, on other hand, have flat growth curve with the maximum load approaching the cardinalities of **flows** and **flow_pairs** respectively.

7. CONCLUSIONS

New deployments of very high speed (OC768) networks place unprecedented demands on network monitoring systems, requiring the use parallel and distributed stream processing. Two main approaches used to distribute the load across the cooperating machines are query plan partitioning and query-independent data stream partitioning. However, for a large class of queries both approaches fail to reduce the load compared to centralized system, and can even lead to increase in the load.

In this paper, we introduce the idea of query-aware data stream partitioning that allows us to scale the performance of streaming queries in close to linear fashion. Our stream partitioning mechanism consists of two main components. The first component is a query analysis framework for determining the optimal partitioning for a given set of queries. The second component is a partition-aware distributed query optimizer that transforms an unoptimized query plan into a semantically equivalent query plan that takes advantage of existing partitions. These components operate within the limitations of currently available networking hardware, while being able to take advantage of new capabilities as they become available.

We evaluate our query-aware partitioning approach by running sets of streaming queries of various complexities on a small cluster of processing nodes using high-rate network data streams. The results of our experiments confirm that the partitioning mechanism leads to highly efficient distributed query execution plans that scale linearly with the number of cooperating processing hosts. We also demonstrate that even suboptimal query-aware partitions offer significantly better performance that conventionally used query-independent partitioning. The techniques described in this paper make OC-768 monitoring feasible using a DSMS.

8. REFERENCES

[1] D. J. Abadi et al. The Design of the Borealis Stream Processing Engine, CIDR 2005.

[2] D. J. Abadi et al.. Aurora: A new model and architecture for data stream management. VLDB Journal, 12(2):120-139, 2003.

[3] D. J. Abadi W. Lindner, S. Madden, and J. Schuler. An Integration Framework for Sensor Networks and Data Stream Management Systems. Demonstration. VLDB 2004

[4] A. Arasu et al. STREAM: The Stanford stream data manager. IEEE Data Engineering Bulletin, 26(1):19–26, 2003.

[5] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In Proc. ACM PODS, pages 1–16, 2002.

[6] M. Balazinska, H. Balakrishnan, M. Stonebraker. Contract-Based Load Management in Federated Distributed Systems. NSDI 2004, San Francisco, CA, March 2004.

[7] S. Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. CIDR 2003.

[8] J. Chen, D.J. DeWitt, F. Tian and Y. Wang, NiagaraCQ: A Scalable Continuous Query System for Internet Databases. SIGMOD 2000 pg. 379-390.

[9] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable Distributed Stream Processing. CIDR 2003.

[10] G. Cormode, T. Johnson, F. Korn, S. Muthukrishnan, O. Spatscheck, and D. Srivastava. Holistic UDAFs at streaming speeds. SIGMOD Conference, pages 35–46. ACM, 2004.

[11] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A stream database for network applications. ACM SIGMOD, pages 647–651, 2003.

[12] D. DeWitt, J. Gray. Parallel database systems: the future of high performance database systems. Communications of the ACM, v.35 n.6, p.85-98, June 1992.

[13] M. Ivanova and T. Risch. Customizable Parallel Execution of Scientific Stream Queries. VLDB 2005.

[14] R. R. Kompella, S. Singh, and G. Varghese. On scalable attack detection in the network. In ACM Internet Measurement Conference IMC 2004, pages 187 - 200.

[15] D. Kossmann. The state of the art in distributed query processing. ACM Computing Surveys, 32(4):422--469, 2000.

[16] Per-Ake Larson. Data Reduction by Partial Preaggregation. 18th International Conference on Data Engineering (ICDE'02), 2002.

[17] J. Li, D. Maier, K. Tufte, V. Papadimos, P. A. Tucker: No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. SIGMOD Record 34(1): 39-44 (2005)

[18] S. Muthukrishnan. Data Streams: Algorithms and Applications. Foundations and Trends in Theoretical Computer Science, Vol 2, 2005.

[19] J. Rao, C. Zhang, N. Megiddo, G. M. Lohman: Automating physical database design in a parallel database. SIGMOD Conference 2002: 558-569

[20] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, M. J. Franklin. Flux: An Adaptive Partitioning Operator for Continuous Query Systems. ICDE 2003

[21] M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. In Proc.USENIX Annual Technical Conf., 1998