Multi-Query Optimization of Sliding Window Aggregates by Schedule Synchronization*

Lukasz Golab†
University of Waterloo
lgolab@uwaterloo.ca

Kumar Gaurav Bijay
Indian Inst. of Tech., Bombay
gauravk@cse.iitb.ac.in

M. Tamer Özsu
University of Waterloo
tozsu@uwaterloo.ca

Categories and Subject Descriptors: H.2.4 [Database Management]: Systems—Query Processing

General Terms: Algorithms, Design

Keywords: data streams, sliding windows, persistent queries, multi-query optimization

1. INTRODUCTION

Data stream management systems process on-line data such as sensor measurements, IP packet headers, stock quotes, and transaction logs. Usually, only a sliding window of recently arrived data is available at any given time for two reasons: to avoid memory overflow and to emphasize recent data that are likely to be more useful. Users issue persistent queries that monitor the streaming data over some period of time. As such, persistent queries often compute sliding window aggregates, such as road traffic volume via sensors embedded in a motorway, network bandwidth usage statistics, or recent behaviour of stock prices. For efficiency, answers of persistent queries are typically assumed to be updated periodically with some user-specified frequency or re-execution interval.

Similar monitoring queries may run in parallel at any given time; for example, many queries may compute the same aggregate function on the same attribute, but over different window lengths and with different frequencies. Therefore, multi-query optimization is particularly important. One of the primary goals of traditional multi-query optimization is to detect common parts across multiple queries issued at the same time and perform the common task only once. In this paper, we argue that in the context of periodically re-evaluated persistent queries, multi-query optimization requires an additional step This is because queries that have been identified as similar may be re-evaluated with different frequencies and therefore may be scheduled at different times. Thus, the purpose of the additional step is to synchronize the re-execution times of similar queries.

The solution presented in this paper assumes that users specify an upper bound on the interval between re-evaluations of their queries and is based upon the following insight: it may be cheaper to re-execute some queries more often if their re-execution schedules can be synchronized with those of similar queries, thereby amortizing the computation costs. We also show that additional schedule synchronization is possible when the system is forced to lengthen the desired re-execution intervals during periods of overload. What follows is a brief overview of the proposed solution using a simple running example; more details may be found in the extended version of this paper [1].

*This research is partially supported by Bell Canada, Natural Sciences and Engineering Research Council of Canada (NSERC), and Communications and Information Technology Ontario (CITO).
†Now at AT&T Labs, Florham Park, New Jersey, USA.

Copyright is held by the author/owner(s).
CIKM'06, November 5–11, 2006, Arlington, Virginia, USA.
ACM 1-59593-433-2/06/0011.

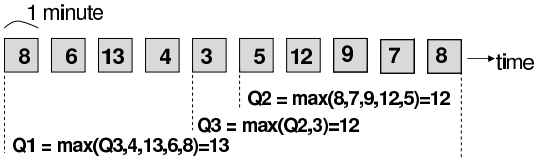

Figure 1: Shared evaluation of MAX aggregates.

2. QUERY EVALUATION

Assume that the current workload contains queries Q1 through Q7, each of which computes the `MAX` aggregate on a numerical attribute `a` of stream `S`. Suppose that the queries have the following window lengths (specified via the `WINDOW` clause) and frequencies (specified via the `SLIDE` clause).

```
Q1 SELECT MAX(a) FROM S [WINDOW 10 min SLIDE 2 min]
Q2 SELECT MAX(a) FROM S [WINDOW  5 min SLIDE 2 min]
Q3 SELECT MAX(a) FROM S [WINDOW  6 min SLIDE 2 min]
Q4 SELECT MAX(a) FROM S [WINDOW 15 min SLIDE 3 min]
Q5 SELECT MAX(a) FROM S [WINDOW 12 min SLIDE 3 min]
Q6 SELECT MAX(a) FROM S [WINDOW 20 min SLIDE 5 min]
Q7 SELECT MAX(a) FROM S [WINDOW 30 min SLIDE 5 min]
```

Suppose that Q1, Q2, and Q3 are all due for re-evaluation. They can be answered using a sliding window synopsis illustrated in Figure 1. The window is partitioned into non-overlapping intervals of one minute each and each interval stores only its maximum value. Every minute, the synopsis is updated by deleting the oldest interval and appending the maximum value that has arrived within the last minute. To obtain the maximum over a sliding window of n minutes, we take the maximum of the first n intervals. As illustrated, the answers of Q2 and Q3 may effectively be computed for free during the computation Q1—we stop after reading the first five intervals and return an answer of Q2, then read the next interval and return an answer of Q3, and then read the remaining four intervals in order to answer Q1.

In general, we assume the existence of a set of rules specifying which queries may be efficiently executed together if their re-execution times happen to coincide. In the above example, all seven queries may share computation, provided that the synopsis from Figure 1 contains 30 one-minute intervals to cover the longest window referenced by Q7. Furthermore, we assume the existence of a query scheduler that determines when to re-execute queries based upon the frequencies specified in their `SLIDE` clauses (the classical earliest-deadline-first algorithm may be used as a starting point for designing such a scheduler [1]).

3. SCHEDULE SYNCHRONIZATION

Suppose that queries Q1 through Q7 are partitioned into three groups according to their frequencies: group G_1 contains Q1, Q2, and Q3; group G_2 contains Q4 and Q5; and group G_3 contains Q6 and Q7. An execution sequence of Q1 through Q7 is illustrated on a time axis in Figure 2 (a). Queries in G_1 are re-executed jointly (as described in Section 2) every two minutes, queries in G_2 every three minutes, and queries in G_3 every five minutes. Furthermore, every six minutes (the least common multiple of 2 and 3), queries in G_1 and G_2 are all due for a re-execution and all five of them may be re-executed together. Similarly, queries in G_1 and G_3 are all due for a re-execution every ten minutes, and so on. We call this technique *conservative scheduling* as it performs no schedule synchronization—similar queries are executed together only if they happen to be due for re-execution at the same time.

Another possibility is to synchronize the schedules of all similar queries that may be executed together in order to take full advantage of overlapping computation. This means that all seven queries from above would have to be scheduled every two minutes, as illustrated in Figure 2 (b) (this is acceptable due to the assumption that users specify upper bounds on the re-execution intervals of their queries). That is, every two minutes, all 30 intervals of the synopsis are scanned in order to answer Q7 and the remaining six queries are answered at the same time. We refer to this technique as *aggressive scheduling* and point out that it re-executes Q4 through Q7 more often than necessary.

We propose a technique called *hybrid scheduling* that is a compromise between conservative and aggressive scheduling. Using a relative cost model, hybrid scheduling determines whether it is more efficient for some groups of similar queries to be re-executed more often than necessary in order to synchronize their schedules with other groups. In the context of our running example, the first step is to calculate the cost of a single re-execution of each group. We can compute all queries in G_1 by scanning the ten youngest intervals of the synopsis (recall that answers over shorter windows, as needed by $Q2$ and $Q3$, will be computed along the way). The cost is 9 (comparisons to determine the maximum of ten maximum values stored in each interval). By similar reasoning, the cost of G_2 is 14 and the cost of G_3 is 29. Next, we want to compute the execution cost of G_1 and G_2 incurred by conservative scheduling (recall Figure 2 (a)). Every six minutes, queries in both groups are executed together for a cost of 14. Again, while scanning the synopsis to compute the maximum over the 15-minute window needed by $Q4$, the answers of other queries are computed along the way. In the interim, queries in G_1 are executed separately twice, for a cost of $9 * 2 = 18$, and G_2 once for a cost of 14.

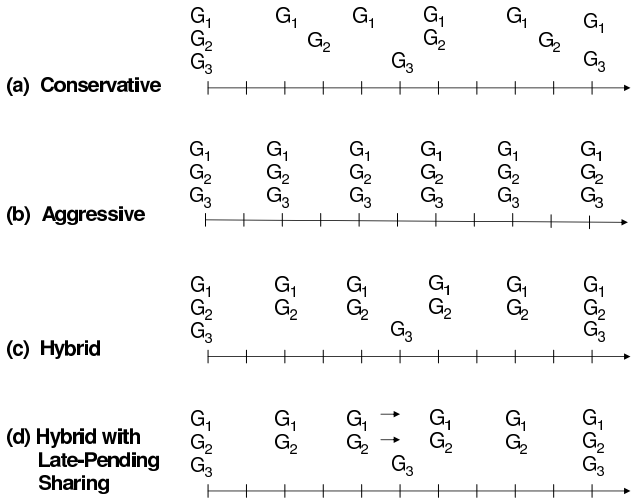

Figure 2: Execution sequences using various scheduling techniques.

Therefore, the total cost of executing queries in G_1 and G_2 is 46 per six minutes, or 7.67 per minute.

Now suppose that G_2 is executed whenever G_1 is due for a refresh. In this case, both groups of queries are executed every two minutes and their cost per minute is $\frac{14}{2} = 7$. Therefore, the best way to execute queries in G_1 and G_2 is in fact to schedule them both with a frequency of two minutes. In total, there are five possibilities: none of the groups change their frequencies (which corresponds to conservative scheduling and costs 11.63 per minute), G_2 changes its frequency to two minutes (11.4 per minute), both G_2 and G_3 change their frequency to two minutes (this corresponds to aggressive scheduling and costs 14.5 per minute), G_3 changes its frequency to three minutes (12.67 per minute), and G_3 changes its frequency to two minutes (16.83 per minute). Hybrid scheduling chooses the most efficient of these five possibilities, namely increasing the frequency of G_2 to match that of G_1 and always executing queries in these two groups together, as illustrated in Figure 2 (c).

We now show that additional computation sharing is possible during overload, when hybrid scheduling is unable to execute all queries with the desired frequencies. A technique called *late-pending sharing* is shown in Figure 2 (d). As indicated by the arrows, suppose that queries in G_1 and G_2 are late by one minute. At this time, queries in G_3 are now pending. One possibility is to execute the late queries first and then move on to the pending queries in G_3. In contrast, late-pending sharing recognizes that queries in all three groups are similar and executes the three groups together. Although this shifts some of the system resources away from clearing the backlog of late queries, it is beneficial in terms of system throughput due to shared computation (rather than scanning the synopsis twice, one scan suffices to answer all seven queries).

4. REFERENCES

[1] L. Golab, K. G. Bijay, and M. T. Özsu. Multi-query optimization of sliding window aggregates by schedule synchronization. University of Waterloo Technical Report CS-2006-26 (`www.cs.uwaterloo.ca/research/tr/2006`).