Prefilter: Predicate Pushdown at Streaming Speeds

Lukasz Golab, Theodore Johnson, and Oliver Spatscheck
AT&T Labs–Research
180 Park Avenue, Florham Park, NJ, USA 07932
{lgolab,johnsont,spatsch}@research.att.com

ABSTRACT

This paper presents the *prefilter*: a predicate pushdown framework for a Data Stream Management System (DSMS). Though early predicate evaluation is a well-known query optimization strategy, novel problems arise in a high-performance DSMS. In particular, (i) query invocation costs are high as compared to the cost of evaluating simple predicates that are often used in high-speed stream analysis; (ii) selectivity estimates may become inaccurate over time; and (iii) multiple queries, possibly containing common subexpressions, must be processed continuously. The prefilter addresses these issues by constructing appropriate predicates for early evaluation as soon as new data arrive and before any queries are invoked. It also compresses the bit vector representing the outcomes of pushed-down predicates over newly arrived tuples, and uses the compressed bitmap to efficiently check which queries do not have to be invoked. Using a set of network monitoring queries, we show that the performance of AT&T's Gigascope DSMS is significantly improved by the prefilter.

1. INTRODUCTION

This paper describes the *prefilter*: a lightweight mechanism for early predicate evaluation in a high-performance Data Stream Management System (DSMS). Predicate pushdown has been explored in many forms; however, novel issues arise when evaluating multiple continuous queries over very fast data streams, such as IP traffic on a busy link.

For one, merely invoking a query (i.e., polling the scheduler, pushing variables on the stack, etc.) becomes a significant source of overhead. For instance, suppose that a stream arriving at a rate of 200,000 packets/sec (800 Megabits/sec assuming a packet size of 500 bytes) is monitored by 50 queries. Assuming that each query invocation requires roughly 100 instructions, this overhead amounts to a billion instructions per second. While invocation costs are negligible in queries containing bulky operators such as joins, they are significant in lightweight streaming query plans with simple predicates (e.g., scalar comparisons that require several instructions). Additionally, queries in a large stream analysis query set often look for "needles in haystacks", i.e., rare events or unusual patterns. However, we must examine the entire stream in order to detect these valuable rare events.

We thus aim to reduce the number of query invocations, and thereby improve throughput, by "prefiltering" the data stream. That is, we push down a set of inexpensive *prefilter predicates* and evaluate them as soon as a new stream tuple arrives (and before any queries are instantiated). Based on the results of these predicates, we do not invoke queries that cannot be satisfied with the new tuple. Effectively, the prefilter achieves earlier and more aggressive data reduction than traditional predicate pushdown, and incorporates multi-query optimization as prefilter predicates occurring in multiple queries need to be evaluated only once.

We illustrate the challenges behind the prefilter with the following subset of queries that monitor an IP traffic stream:

```
Q1: SELECT t, srcIP, destIP, sum(len), count(*)
    FROM IPStream
    WHERE protocol=UDP
    GROUP BY time/60 as t, srcIP, destIP

Q2: SELECT t, srcIP, destIP, sum(len), count(*)
    FROM IPStream
    WHERE protocol=UDP AND dest_port=53 AND qr=0
    GROUP BY time/60 as t, srcIP, destIP

Q3: SELECT t, srcIP, destIP, sum(len), count(*)
    FROM IPStream
    WHERE protocol=UDP AND src_port=53 AND qr=1
    GROUP BY time/60 as t, srcIP, destIP
```

Q1 computes the bandwidth usage (sum of packet lengths) and packet count of UDP traffic for each source-destination IP pair. Q2 and Q3 compute the same aggregates over DNS requests and responses, respectively (DNS uses the UDP transport protocol on port 53, while `qr` is a boolean field that distinguishes DNS requests from responses). In order to unblock the aggregation, all three queries are evaluated over one-minute time epochs (`GROUP BY time/60`). At the end of each epoch, each `srcIP-destIP` group and its aggregates are flushed to the output stream. Note that all the selection predicates in this query set involve inexpensive scalar comparisons; the aggregation operations are more expensive, but they operate over a reduced data stream, after the respective selection predicates have been applied.

The first technical issue is: *which predicates should be pushed down to the prefilter?* One possibility is the shared predicate `protocol=UDP`, as per classical multi-query optimization techniques that factor out common subexpressions [20]. Using recent DSMS and publish-subscribe multi-query optimizations [4, 6, 10, 14, 15, 22], we could also build predicate indices and push down predicates on common fields (e.g., `qr=0` and `qr=1`). The novelty in our context is that in addition to evaluating shared pred-

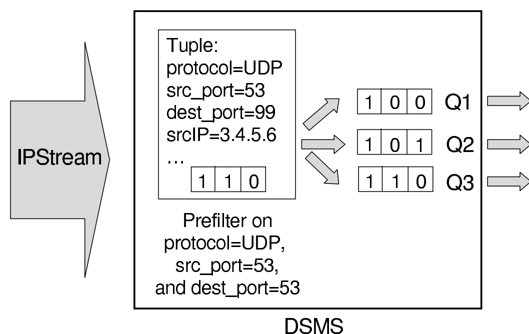

Figure 1: Processing Q1, Q2, and Q3 with the prefilter.

icates only once, we want to reduce the number of query invocations. Therefore, even non-shared inexpensive predicates (e.g., `src_port=53` and `dest_port=53`) are prefilter candidates.

Next, we need to *detect which queries do not have to be executed based on the outcomes of the prefilter predicates*. To do this, we assign a bit signature to each query, specifying which prefilter predicates it contains. The process is illustrated in Figure 1 for the three queries from the motivating example. Suppose that the following predicates are pushed down to the prefilter: `protocol=UDP`, `src_port=53`, and `dest_port=53`. Let S_i be the *bit signature* of query Q_i. For instance, Q3 contains the first and second prefilter predicates, therefore $S_3 = 110$. When a new tuple arrives, the three predicates are evaluated at the prefilter and their outcomes entered into a bit vector B. In Figure 1, the new tuple is assumed to satisfy the first two prefilter predicates, therefore $B = 110$. Assuming that the `WHERE` clause of each query contains a conjunction of predicates, we execute Q_i only if $B \,\&\, S_i = S_i$, where & is the bitwise-AND operation. That is, Q_i is invoked only if each of its pushed-down predicates evaluates to true. The invoked query plans (Q1 and Q3 in our example) then evaluate their remaining operators. Note that:

- the shared predicate `protocol=UDP` is evaluated only once per tuple,
- we avoid the cost of invoking Q2 for this tuple,
- simple bit operations are sufficient to determine which query plans to execute over a new tuple.

Since the prefilter is evaluated for each new tuple, its overhead must be minimal in order to keep up with a massive data stream. One way to guarantee efficiency of bit operations is via a hardware-dependent cap on the number of bits in the prefilter and the query signatures. For instance, a 64-bit processor can perform efficient operations on up to 64 bits using one register-compare. Furthermore, we may want to evaluate the prefilter on specialized hardware directly at the data source, e.g., a network interface card. In this case, the bit budget may be even smaller to reflect the limited processing capabilities of network hardware and to minimize the overhead of transmitting a bitmap with every tuple (that satisfies at least one prefilter predicate).

If the number of prefilter candidates exceeds the bit budget b, then one solution is to choose the "best" b predicates with respect to the expected query processing cost. However, making a reasonable choice requires accurate estimates of predicate selectivity [17]. Unfortunately, the selectivity estimates available to a DSMS may become inaccurate over time due to the time-evolving nature of streaming data and the long-running nature of streaming queries.

The main technical contribution of this paper is a solution to the above problem. Rather than choosing predicates based on unreliable statistics, we retain all the candidates and attempt to represent them using a small number of bits. Recall Figure 1 and suppose that we want to evaluate all five unique predicates in the prefilter, i.e., `protocol=UDP`, `dest_port=53`, `src_port=53`, `qr=0`, and `qr=1`. A straightforward solution requires a prefilter bit vector and query signatures of size five. However, instead of spending one bit on each distinct predicate, we may assign a conjunction of several predicates to a single bit, so long as we can still avoid invoking a query if any one of its prefilter predicates fails. In Figure 1, we can leave the first bit unchanged, change the second bit to `(src_port=53 AND qr=1)`, and change the third bit to `(dest_port=53 AND qr=0)`. This way, we can push two more predicates into the prefilter without increasing the bit budget. We will show that using the fewest possible bits to represent a set of predicates can be reduced to a bipartite graph covering problem. We will prove that the problem is NP-hard and propose efficient heuristics.

The second contribution of this paper is an experimental evaluation of the prefilter within AT&T's Gigascope DSMS [7, 8]. Using a network monitoring query set, we show that the prefilter reduces the CPU utilization by nearly one half, even without accurate selectivity estimates.

The remainder of this paper is organized as follows. Section 2 describes the design of the prefilter, Section 3 presents experimental results, Section 4 compares the prefilter with previous work, and Section 5 concludes the paper with suggestions for future research.

2. PREFILTER DETAILS

We now present the technical details behind the prefilter. Choosing prefilter predicates is discussed in Section 2.1, followed by representing the predicates using a small number of bits in Section 2.2. Section 2.3 then discusses cases where the number of candidate predicates exceeds the bit budget, and Section 2.4 presents optimization techniques for efficient evaluation of prefilter predicates.

For the remainder of this paper, assume without loss of generality that a single data stream is being processed. Multiple inputs are handled by creating independent prefilters with predicates over their respective streams, whereas join predicates over multiple streams are computed by the join operators further up in the processing tree. Each query is assumed to contain a conjunction of zero or more *base predicates*. Two base predicates are said to be equivalent if they are syntactically equal (after normalization). Each base predicate is associated with a cost and, optionally, a selectivity estimate, with the caveat that the latter may not be accurate throughout the lifetime of the query set.

2.1 Choosing Prefilter Predicates

The first step in creating the prefilter is to choose which predicates to push down. An exhaustive solution considers pushing down each subset of the base predicates. However, in addition to being prohibitively expensive to compute, the exhaustive technique requires accurate selectivity estimates in order to compute the expected cost of each alternative. Instead, we use a simple and robust heuristic. First, we set c to be the maximum cost of a base predicate that may be considered "cheap"; e.g., a one-cycle operation such as comparison of an attribute value to a constant. The value of c should be much smaller than the cost of query invocation. The remaining base predicates are labeled "expensive". Then, we place all the cheap base predicates (shared or otherwise) in the prefilter candidate set.

An example is shown in Figure 2, illustrating plans for two

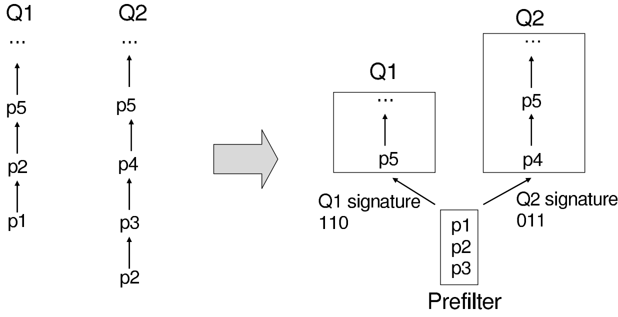

Figure 2: Translating individual query plans into a shared plan.

queries, Q1 and Q2 (only their predicates are shown). Suppose that `p1` through `p3` are cheap and that the illustrated plans are locally optimal, i.e., the base predicates of both queries are ordered in an optimal way. The right side of the illustration shows the corresponding prefilter with all the cheap base predicates pushed down.

The justification behind the proposed heuristic is as follows. First, the cost of evaluating a predicate is expected to be more stable over time than its selectivity. Additionally, even if selectivities are known to be accurate and are used to generate optimal local plans, cheap predicates are still ordered early in these optimal plans, unless they are very non-selective. Therefore, pushing down cheap base predicates is likely to create an efficient and robust global plan. Second, pushing down all the cheap base predicates induces common sub-expressions that would not exist if only the locally optimal plans were considered. For instance, the two queries in Figure 2 share the cheap base predicate `p2`, but this predicate could not be "factored out" unless we flipped the execution order of `p1` and `p2` in Q1's plan. In other words, the heuristic implicitly considers locally non-optimal plans when building the global plan.

A disadvantage of preventing expensive predicates from being evaluated at the prefilter is that shared expensive predicates are re-executed redundantly; e.g., `p5` in Figure 2 is computed by Q1 and Q2. However, adding `p5` to the prefilter may not be optimal as it would reverse the order of evaluation of `p4` and `p5` in Q2. If `p5` is much more expensive and/or much less selective than `p4`, then the resulting global plan could be inefficient despite the shared evaluation of `p5`. One alternative is to push down `p4` as well and evaluate it in the prefilter before `p5`. However, this approach suffers from two problems. First, in the worst case, all the base predicates would have to be pushed down. This defeats the goals of keeping the prefilter bit vector short and evaluating inexpensive predicates at the prefilter. Second, the prefilter evaluation logic would have to be more complex in order to avoid unnecessary evaluation of expensive predicates; e.g., in Figure 2, first we compute the cheap base predicates, then `p4`, then `p5`, but only if either `p4` evaluated to true, or `p1` or `p2` evaluated to true.

Rather than computing expensive predicates at the prefilter, our solution is to cache the outcomes of shared expensive predicates in a separate data structure. This way, if Q1 in Figure 2 is invoked and computes `p5`, then Q2 can look up the result of `p5` in the predicate cache.

2.2 Compact Representation of Prefilter Predicates

The next step in prefilter design is to assign a small number of bits to represent the pushed-down predicates, while still being able to avoid unnecessary query invocations. We define a *composite predicate* as a conjunction of two or more base predicates. The idea is to assign bits to composite, rather than base, predicates.

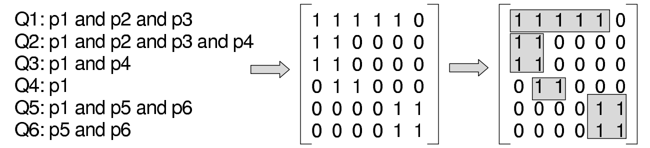

Figure 3: A minimum-sized rectangle covering of Q1 through Q6.

Let n be the number of queries and p be the number of unique base predicates that are candidates for the prefilter. Let M be a p-by-n boolean matrix and $M(i, j)$ be the entry in its ith row and jth column. Define $M(i, j)$ to be 1 if the ith base predicate is referenced in the query corresponding to the jth query. Otherwise, $M(i, j) = 0$. The following definitions will be used in our formalization.

Definition 1 *Let P and Q be subsets of the rows and columns of M, respectively. P and Q define a rectangle $r(P, Q)$ if for each $i \in P$ and $j \in Q$, $M(i, j) = 1$.*

Definition 2 *A rectangle covering of M is a set of rectangles defined over M such that each non-zero entry in M is in at least one rectangle.*

We express the problem of minimizing the length of the prefilter bit vector (and avoiding all the query invocations that would be avoided if each base predicate had a separate bit) as finding a minimum-sized rectangle covering of M. An example is illustrated in Figure 3, showing (the predicates of) six queries, the corresponding matrix, and a minimum-sized rectangle covering. The four rectangles denote the following prefilter predicates: `p1`, `(p2 AND p3)`, `p4`, and `(p5 AND p6)`[1]. The corresponding bit signatures L_i are: $L_1 = 1100$, $L_2 = 1110$, $L_3 = 1010$, $L_4 = 1000$, $L_5 = 1001$, and $L_6 = 0001$. Thus, we require only four bits (i.e., four composite predicates) to represent the six unique base predicates. Furthermore, it is easy to show that this configuration is equivalent, in terms of avoiding query invocations, to assigning one bit per base predicate. Note that for simplicity, we have illustrated a minimum-sized rectangle covering that does not contain overlapping rectangles; we will deal with overlap in Section 2.4.

Finding a minimum-sized rectangle covering of a boolean matrix is NP-hard as it can be reduced to finding a minimum-sized bipartite graph covering using complete subgraphs (see Appendix for proof). Below, we present a heuristic for finding a near-optimal solution; its efficiency and effectiveness are experimentally evaluated in Section 3. The heuristic consists of two steps: finding rectangles embedded in M and using them to create a covering of M.

Finding rectangles in M is accomplished by the algorithm shown in Figure 4; recall that p be the number of prefilter candidate predicates. The idea is to use rectangles representing i base predicates to generate new rectangles with $i + 1$ base predicates. In steps 2 through 6, we initialize a set *BASE* corresponding to the base predicates, as well as the target set of rectangles *RECT*. The latter initially contains all the rows and columns of M. The loop in lines 7 through 12 creates rectangles of size $i + 1$ by attempting to add every possible base predicate (i.e., every rectangle in *BASE*) to each

[1]Note that the rows and/or columns of a rectangle need not be contiguous. For instance, the first and fourth row and the second and third column form a rectangle defined by $P = \{1, 4\}$ and $Q = \{2, 3\}$.

```
1  BASE=RECT=∅
2  For each row i of M
3     Add rectangle corresponding to i to BASE
4     Add rectangle corresponding to i to RECT
5  For each column j of M
6     Add rectangle corresponding to j to RECT
7  For k = 1 to p − 1
8     For each rect. r(P, Q) ∈ RECT with |P| = k
9        For each rectangle b(P′, Q′) ∈ BASE
10          If |Q ∩ Q′| > 1
11             Create rectangle r′(P ∪ P′, Q ∩ Q′)
12             Add r′ to RECT
13 Return RECT
```

Figure 4: Rectangle finding algorithm.

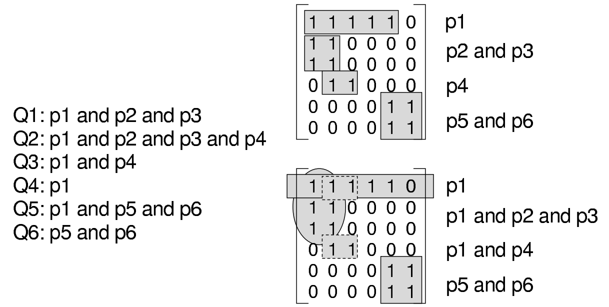

Figure 5: Two minimum-sized rectangle coverings of Q1 through Q6.

rectangle of size i. Line 10 tests if each attempt actually creates a new rectangle—all of its $i + 1$ base predicates must occur in more than one query (otherwise, the "new" rectangle is contained in an individual column of M already added to *RECT* in line 6). If so, then we add the new rectangle r' to *RECT*. Its base predicate set is the union of the "old" rectangle's base predicate set and the new base predicate used in line 9. The query set of r' consists of queries that reference all of its predicates.

The number of rectangles in M may be large, but a variety of pruning rules may be applied. For instance, we can remove rectangles contained in a newly created rectangle. Recall the rectangle in the bottom-right corner of Figure 3, defined as: $P = \{5, 6\}$, $Q = \{5, 6\}$ and corresponding to the composite predicate `(p5 AND p6)`. This rectangle contains two smaller rectangles, corresponding to base predicates `p5` and `p6`, respectively, i.e., $P = \{5\}$, $Q = \{5, 6\}$ and $P = \{6\}$, $Q = \{5, 6\}$. Once the large rectangle is found, the smaller ones may be removed.

Another simple optimization is to only consider rectangles containing a small number of base predicates, say up to j (i.e., modify line 7 to iterate from 1 to j). The reasoning behind this rule is that we do not expect a very large number of base predicates to be shared across a group of queries.

Finally, having generated a set of rectangles embedded in M, we apply the greedy heuristic for set covering problems in order to find an approximate solution for the minimum-sized rectangle covering. That is, at each step, we choose the rectangle which covers the most uncovered "ones" in M. Each rectangle in the covering is then translated into the predicate that it represents.

2.3 Handling a Large Number of Predicates

Recall that the number of prefilter bits is limited in order to reduce overhead. For workloads containing many queries and unique predicates, there may be more predicates than available bits, even after compressing the predicates using the rectangle covering heuristics. Suppose the number of available bits is k. In this situation, we use one of the following two solutions. The first is to push down the first k rectangles returned by the rectangle covering heuristic. The second solution is used when accurate predicate selectivity estimates are available; e.g., if statistics are collected periodically and selectivities are known not to change over time. In this case, rather than building the covering by always choosing the rectangle which covers the most uncovered "ones" in M, we choose the rectangle (i.e., composite predicate) which yields the biggest decrease in the expected number of query invocations. Assuming that all the predicates are independent, we can calculate the expected number of invocations of a particular query as the product of the selectivities of all of its predicates evaluated at the prefilter. We then take the first k rectangles returned by the modified heuristic, eliminate rectangle overlap, and place the resulting k predicates in the prefilter.

Note that another way to reduce the number of bits used is to assign bits to disjunctions of (base or composite) predicates. For simplicity, we do not consider this possibility in this paper. Intuitively, a disjunction of two or more predicates is more likely to evaluate to true than a conjunction, in which case no query invocations can be avoided.

2.4 Efficient Evaluation of Prefilter Predicates

At this point, we have a set of (base and/or composite) predicates marked for evaluation at the prefilter. The next issue we discuss is how to efficiently evaluate the predicates. A "default" strategy computes the prefilter predicates sequentially and in arbitrary order whenever a new tuple arrives. Then, the prefilter bit vector is compared with each query signature (again, sequentially and in arbitrary order). Additionally, the prefilter framework is compatible with many orthogonal techniques for speeding up predicate evaluation, among them predicate indexing (we will discuss these in more detail in Section 4). However, such techniques may complicate the implementation of the prefilter and incur additional overhead. Instead, in what follows, we discuss two "compile-time" optimizations that do not add any overhead.

The first optimization deals with base predicates repeated in several bits (composite predicates) in the prefilter. This occurs if the covering produced in the previous step contains overlapping rectangles. For example, Figure 5 shows two minimum-sized coverings for the workload from Figure 3; the first is the non-overlapping covering already shown in Figure 3 and the second contains overlapping rectangles. The latter consists of four rectangles: the first row of M corresponding to `p1`, the circled region corresponding to `(p1 AND p2 AND p3)`, the union of the two dotted regions giving `(p1 AND p4)`, and the bottom-right rectangle corresponding to `(p5 AND p6)`. Observe that the overlapping covering induces three prefilter predicates containing `p1`. As a result, `p1` is evaluated redundantly in the prefilter.

We solve this problem by adding a post-processing step that eliminates overlap whenever possible. The idea is to remove a set of base predicates from a composite predicate if a conjunction of those base predicates already has its own bit. In Figure 5, `p1` has its own bit and occurs inside two composite predicates. With `p1` removed, these two predicates simplify to `(p2 AND p3)`, and `p4`, respectively. At this point, all the rectangles in the covering are non-overlapping, though in the general case, more than one iter-

ation of this procedure may be required to make all the possible simplifications. Finally, we revise the query signatures to account for the changes in predicate definitions.

The second improvement concerns efficient attribute "unpacking", i.e., extracting attribute values from raw stream tuples. This process may be relatively expensive for variable-offset and variable-length fields. However, a set of attributes can often be unpacked more efficiently as a group (as compared to on-demand unpacking of individual fields done separately by each query). For instance, in the context of IP traffic streams, it is easy to unpack all the TCP header attributes from an IP packet once the beginning of the header is found. Since the prefilter needs to extract all the fields referenced in all of its predicates prior to evaluating them, it can take advantage of group unpacking. To do this, we compute two parameters for each attribute of the stream: the cost of unpacking it separately and the cost of unpacking it along with a set of other attributes (e.g., those at the same protocol layer in the context of network traffic streams). These parameters are straightforward to estimate; e.g., extracting a fixed-offset field costs one operation, given random access into the packet. We can now model the problem of group unpacking in terms of weighted set covering and use a greedy heuristic to obtain the answer: at each step, we choose the group of fields which gives the cheapest overall unpacking cost per field.

3. EXPERIMENTAL EVALUATION

3.1 Setting

We implemented the default version of the prefilter (which evaluates the predicates sequentially in arbitrary order, without any predicate indices) in the Gigascope DSMS [7, 8]. Gigascope divides each query plan into a low-level and high-level component, denoted LFTA and HFTA, respectively. An LFTA evaluates fast operators over the raw stream, such as projection, simple selection, and partial group-by-aggregation using a fixed-size hash table. Early filtering and pre-aggregation by the LFTAs are crucial in reducing the data volume fed to the HFTAs, which are scheduled separately and execute complex operators (e.g., user-defined functions, joins, and final aggregation).

Tests were performed on a live network data feed from an AT&T data center tap. All of our experiments monitor a high-speed DAG4.3GE Gigabit Ethernet interface, which receives approximately 105,000 packets per second (about 400 Mbits per second). Experiments were conducted on dual-processor 2.8 GHz P4 server with 4 GB of RAM running FreeBSD 4.10.

We tested the prefilter on a network monitoring query set developed for an AT&T application. The set contains 24 complex queries, i.e., 24 output streams to which applications may subscribe. This gives rise to 50 LFTAs (some queries merge the output of many LFTAs). The LFTAs contain 47 unique cheap predicates, where a predicate is considered cheap if its estimated cost is less than 10 instructions (raising this threshold up to 100 did not appear to have any performance effect on the available query sets). Thus, the prefilter contains 47 predicates and bit signatures for 50 queries (LFTAs).

Though we are unable to reveal the details of the query set, we illustrate the corresponding matrix M (of size 47x50) in Figure 6. The first 14 rows correspond to shared predicates, e.g., referencing common protocols such as TCP or UDP. This motivates the multi-query optimization goal of the prefilter. The remaining 33 prefilter predicates occur in one query each and are used to find specific packets, e.g., DNS requests and responses, as in the example from Section 1. This reflects the data reduction goal of the prefilter as

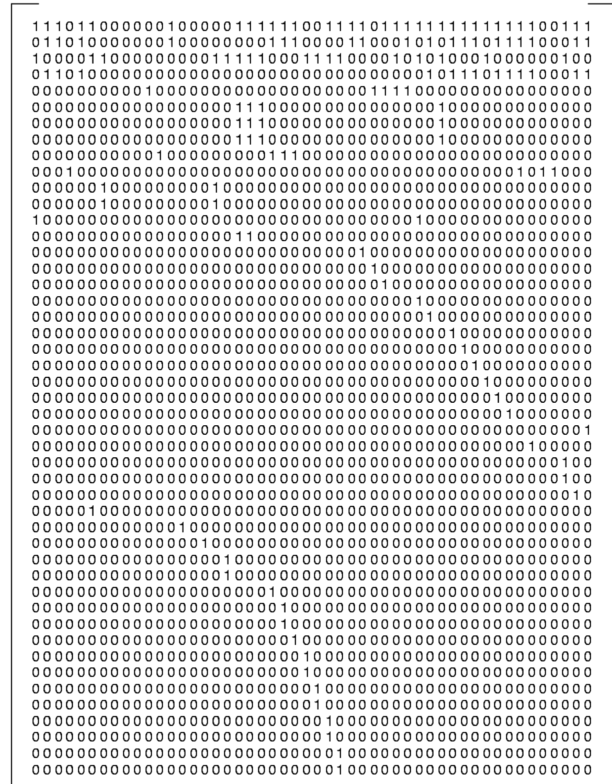

Figure 6: Matrix representation of the query set used in the experiments.

these more specific predicates may be highly selective. We remark that our query set is the largest among the Gigascope applications we have, and is similar to our other query sets (having shared inexpensive predicates for selecting data of common interest, as well as query-specific predicates for selecting rate packets or events).

Also note that four columns of M are all-zeros, therefore of the 50 LFTAs, only 46 contain at least one inexpensive predicate. Moreover, observe that the matrix is quite sparse. It contains 116 rectangles, which can be reduced to 44 if the pruning rule from Section 2.2 is used to remove smaller rectangles contained in larger ones. The height of the tallest rectangles is six (i.e., corresponding to rows 1, 3, 6, 7, 8, and 14, and columns 19 and 20).

Finally, we note that our query set is expected to grow over time. We anticipate that newly added queries will contain some existing predicates (e.g., on common protocols) as well as some new ones. Since the current query set already exceeds the 16 and 32-bit budgets, we expect to reach the 64-bit limit in the near future (of course, another way to deal with a very large query set is to partition it across several Gigascope machines, but that requires making multiple copies of a high-volume data stream).

3.2 Performance of Rectangle Covering Heuristic

We begin by showing the efficiency and effectiveness of the rectangle covering heuristic from Section 2.2. Recall that the cost of finding a covering consists of two parts: finding rectangles in M and generating the covering. Figure 7 plots the time taken by our heuristic and an exhaustive approach as a function of the number of rectangles in the matrix representation of the query workload. The exhaustive approach examines every permutation of the rect-

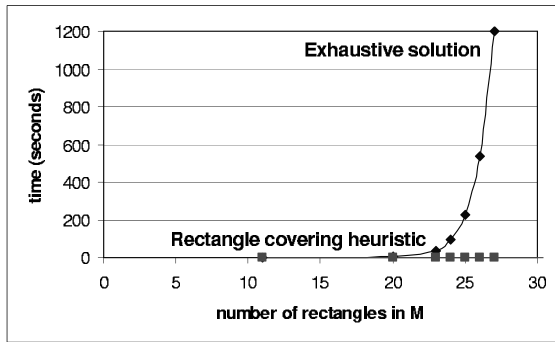

Figure 7: Running times of the rectangle covering heuristic and the exhaustive solution.

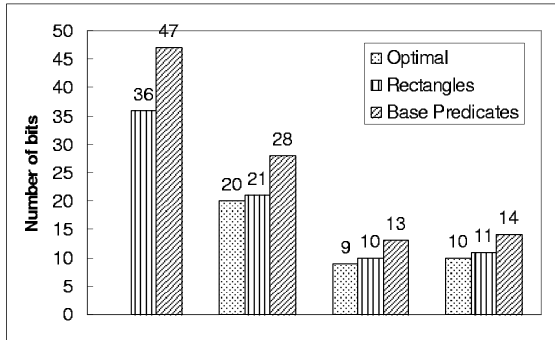

Figure 8: Effectiveness of the rectangle covering heuristic.

angles, starting with all sets of one rectangle each and working upwards. Therefore, its time complexity is exponential in the number of rectangles. As shown on the graph, the exhaustive technique requires over 1000 seconds (over 20 minutes) to find a covering when M contains 27 rectangles. Therefore, the optimal algorithm is intractable over our query set, even if rectangle pruning is used. In contrast, our heuristic can handle hundreds of rectangles in a fraction of a second [2].

In Figure 8, we show the effectiveness of the rectangle covering heuristic by comparing the number of bits it requires versus the optimal solution and the number of base predicates. The first set of bars on the left corresponds to our query set; note that we were unable to obtain the optimal solution in a reasonable time. As shown, the rectangle covering heuristic can represent the 47 prefilter predicates using only 36 bits. The remaining three sets of bars correspond to subsets of our query set that monitor different properties of the IP stream; e.g., the second set of bars from the left reflects a subset of 29 queries with 28 base predicates, 96 rectangles, and 27 main rectangles (which is small enough to compute the optimal solution in under one hour). In all cases, our heuristic is only one bit away from the optimal solution.

3.3 Performance of the Prefilter

Next, we report the performance of Gigascope, with and without the prefilter. Our experiments consisted of two stages. First, we obtained selectivity estimates of the 47 base predicates by creating 47 `COUNT(*)` queries, each with one of the base predicates in its

[2]Given that the rectangle covering heuristic is very fast, it may be feasible in many circumstances to simply rebuild the prefilter whenever new queries are added or existing queries removed over time.

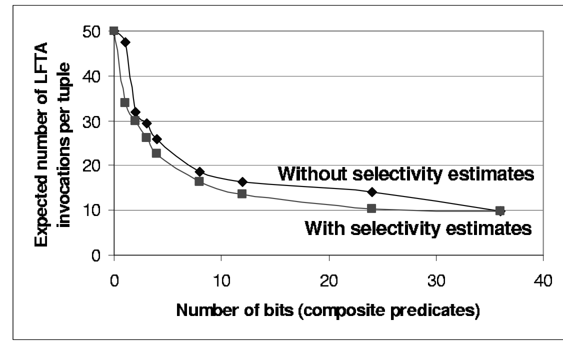

Figure 9: Expected performance of the prefilter (with and without selectivity estimates).

`WHERE` clause. Next, we implemented two versions of the prefilter: one that chooses the rectangle covering without considering selectivities, and one that chooses rectangles according to the expected number of LFTA invocations (as described in Section 2.3). For each version, we experimented with several bit budgets, ranging from one to 36 (which is enough to represent all 47 base predicates).

Prior to discussing the observed performance, we show the expected performance of the two versions of the prefilter in terms of the expected number of LFTA invocations per tuple, assuming that our selectivity estimates remain accurate. Results are plotted in Figure 9 for various numbers of bits in the prefilter, up to 36. When the number of bits is zero, the prefilter is disabled, therefore all 50 LFTAs are invoked for each new tuple. Using 36 bits, fewer than ten LFTAs are expected to be invoked; at this point, using predicate selectivities does not matter as all 36 composite predicates fit in the prefilter anyway. If fewer than 36 bits are available, then the knowledge of (accurate) selectivities can potentially improve performance, but not by a significant margin. Moreover, note that even using as few as ten bits is expected to yield a noticeable performance improvement.

After gathering the selectivity estimates, we immediately ran the experiments with the two versions of the prefilter. Each experiment was performed serially on live traffic data, and hence there is a significant amount of noise error in our results. However, the network feed represents an aggregation of many users, and tends to be stable over short periods of time (but not over the long run; e.g., mornings vs. evenings or weekdays vs. weekends). As a result, the selectivity estimates obtained just before running the experiments were still accurate, aside from ignoring correlations across base predicates due to the independence assumption.

For each experiment, we report the CPU utilization of the Gigascope runtime system, which executes the prefilter and the LFTAs; the CPU consumption of all the HFTAs combined amounted to less than 25 percent and is not affected by the prefilter. We do not report response time because it is not affected by the prefilter (similar to the example queries in Section 1, all the queries in our set flush their results at the end of each time epoch). For each data point, we collected the average packet rate as well as the CPU utilization. We then normalized the CPU utilization by the average packet rate to obtain the equivalent utilization at 105,000 packets/sec (the most common packet rate over the course of the experiments). We observed that the CPU utilization of the runtime system alone (i.e., processing every packet, but not running any queries) was 8.8 percent with the prefilter, and 8.7 percent with the prefilter turned off. Thus, the prefilter is not a source of overhead.

Figure 10 shows the CPU utilization using the two versions of

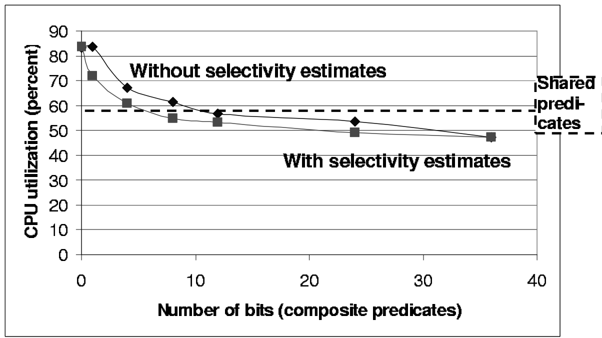

Figure 10: Observed performance of the prefilter (with and without selectivity estimates).

the prefilter described above and with various bit budgets. Without the prefilter (i.e., when the number of bits is zero), the CPU usage is over 80 percent. However, we noticed that there was packet loss at the LFTAs during periods of bursty arrivals (i.e., reduced throughput). There was also packet loss with one bit (with or without selectivity estimates). However, increasing the bit budget to four already brought the CPU utilization down below 70 percent and eliminated packet loss. Increasing the bit budget further caused a gradual decrease of CPU utilization, down to 47 percent when all 36 composite predicates were included. As expected, selectivity knowledge yielded moderate improvement of less than ten percent. Again, this improvement is likely to vanish (or even become negative) as time goes on and the selectivity estimates become stale.

The dotted line in Figure 10 represents the CPU utilization (roughly 58 percent) achieved by evaluating only the 14 shared cheap predicates at the prefilter. This technique may be thought of as corresponding to traditional multi-query optimization. However, further reduction in CPU usage is obtained if the prefilter also includes non-shared cheap predicates. This is because even more LFTA invocations can be avoided and more opportunities for efficient attribute unpacking may arise (recall Section 2.4).

3.4 Lessons Learned

Based upon the results of our experiments, we draw the following conclusions about the performance of the prefilter in Gigascope:

- The rectangle covering heuristic finds near-optimal solutions in terms of the number of bits needed to represent a set of prefilter predicates.
- The prefilter significantly reduces the CPU utilization of the LFTAs, even if only a subset of candidate predicates is pushed down. This means that 1) the prefilter may be evaluated efficiently on network hardware, where the bit budget is smaller, and 2) even if the query set is very large, we are likely to find a small set of (composite) predicates that greatly reduce the number of LFTA invocations.
- The prefilter is likely to be effective even if the query set does not contain shared predicates, as evidenced by the additional drop in CPU usage when non-shared cheap predicates were used in Figure 10.

4. RELATED WORK

We now describe previous work related to the two goals of the prefilter: early data reduction and multi-query optimization.

In terms of early data reduction, the prefilter is based upon the classical idea of predicate pushdown. The novelty of the prefilter is that inexpensive predicates are pushed down "outside" the query plan in order to avoid the overhead of instantiating query operators. This overhead is negligible in complex relational queries, but noticeable in lightweight streaming query plans.

In addition to predicate migration, early data reduction may be accomplished by dropping tuples (random sampling or semantic load shedding [3, 21]). Alternatively, approximate query answers may be obtained by summarizing the stream using limited space (via sketching, histograms, etc. [18]). All of these techniques are orthogonal to the prefilter.

We evaluated the prefilter in Gigascope, which divides each query plan into an LFTA and an HFTA, and schedules the two components separately. However, the prefilter is applicable to any DSMS that needs to process lightweight queries over massive streams, irrespective of the query processing and scheduling architectures. The only requirement is that for each new tuple, the prefilter must be executed before the query processor and scheduler are invoked.

In the context of multi-query optimization, a fundamental problem is that the optimal global query plan is typically different from the union of locally optimal plans for individual queries due to the presence of common sub-expressions [19, 20]. Our heuristic for pushing down cheap predicates is similar to traditional multi-query optimization heuristics that perturb locally optimal plans in order to induce common subexpressions. On a related note, our heuristic also resembles efforts in robust query optimization that attempt to produce query plans which are resistant to changes in estimated parameters such as selectivities [2].

Closely related to the prefilter is the work on shared execution of many selection predicates in DSMSs and publish-subscribe systems. Most of this work proposes variants of predicate indexing [4, 6, 10, 14, 15, 22] and can be easily incorporated into the prefilter framework. Other orthogonal techniques for efficient predicate execution include caching expensive predicates [9] and adaptive execution [17], whereby the outcome of the currently executed predicate determines which predicate to evaluate next.

In particular, we note that predicate indexing is effective if queries contain many similar predicates over the same attribute, which has a large domain, e.g., simple predicates of the form *attribute op constant*, with *op* $\in \{=, <, >\}$ and $constant \in \mathbb{N}$. This is not the case in the network analysis workloads currently handled by Gigascope. First, many attributes in an IP data packet are boolean—recall queries Q2 and Q3 from the Introduction, which reference the boolean attribute `qr` that distinguishes DNS requests from resposes. There are at most two unique predicates over a boolean attribute, and they are both very cheap to evaluate. Second, even those attributes which have very large domains—the source and destination IP addresses—are not referenced by predicates of the form *IP-address op constant*. Instead, these types of predicates are similar to

```
longest_prefix_match(IP, table_name, id)
```

which denotes that the source or destination IP address must contain a prefix of at least one entry in table `table_name` with a given `id`. Thus, a longest prefix match table is a form of a predicate index for hierarchical domains such as IP addresses. In light of these observations, we leave the issue of incorporating index-like optimizations into the prefilter for future work.

[5, 13] discuss predicate pushdown versus pullup for continuous queries with joins. For example, suppose that n queries have the same join predicate, but each has a different single-input selection

predicate on one of the input streams. Maintaining a shared join result may be beneficial, but single-input predicates would have to be pulled above the join. It may be more efficient to push down a predicate that contains a disjunction of the n single-input predicates so that unnecessary join tuples are not materialized. This strategy is compatible with the prefilter—we remove the n single-stream predicates from consideration by the prefilter and instead evaluate the disjunction at the prefilter if its cost is below the threshold. Then, the shared join operator (followed by the n query plans) is invoked only if the disjunction evaluates to true.

We also note that multi-query optimization in Gigascope was specifically addressed in [23]. However, that work only considered overlapping sets of group-by columns and is orthogonal to the prefilter.

Finally, we remark that the rectangle representation of prefilter predicates is similar to Karnaugh maps used to find minimal expressions of Boolean functions [12]. The main difference is that a Karnaugh map refers to a single predicate, while our rectangle covering heuristic finds overlap among composite predicates found in a set of queries. Furthermore, the rectangle finding algorithm from Figure 4 is related to frequent itemset mining algorithms [1], which build large itemsets by extending smaller ones. The fundamental difference is that itemsets must be found by scanning disk-resident data, therefore the goal is to reduce the number of passes over the database. Our algorithm assumes that the rectangle set *RECT* fits in main memory, therefore making p "passes" over memory-resident data (recall line 7 of Figure 4), where p is the number of unique predicates in the prefilter, is not a concern.

5. CONCLUSIONS

This paper presented the prefilter—a lightweight predicate push-down mechanism for a high-performance Data Stream Management System. We motivated two goals of the prefilter in the context of a set of stream analysis queries: early data reduction and multi-query optimization. We accomplished the first goal by evaluating inexpensive predicates as soon as a new tuple arrives and before any queries are invoked. We also formalized the problem of representing a set of predicates using the fewest possible bits, proved its NP hardness, and designed a heuristic that finds a near-optimal solution. Additionally, we showed that efficient extraction of attributes from data stream packets may be performed at the prefilter. The second goal was accomplished by pushing down inexpensive shared predicates and evaluating them only once, as well as caching the results of shared expensive predicates. Experimental results showed that the prefilter improves the performance of AT&T's Gigascope DSMS on a network monitoring query set, without the need to maintain selectivity estimates.

Future work includes the following two directions. Having studied the optimization of simple predicates in this paper, we now want to investigate multi-query optimization in the context of expensive streaming operators, such as user-defined functions. Second, we want to study the capabilities of network interface cards and content-addressable memories in order to determine which operators (in addition to simple predicates) may be pushed down all the way to the network hardware level.

6. REFERENCES

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. *SIGMOD* 1993, 207–216.

[2] B. Babcock and S. Chaudhuri. Towards a robust query optimizer: a principled and practical approach. *SIGMOD* 2005, 119–130.

[3] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. *ICDE* 2004, 350–361.

[4] S. Chandrasekaran and M. J. Franklin. PSoup: a system for streaming queries over streaming data. *The VLDB Journal*, 12(2):140–156, 2003.

[5] J. Chen, D. DeWitt, and J. Naughton. Design and Evaluation of Alternative Selection Placement Strategies in Optimizing Continuous Queries. *ICDE* 2002, 345–357.

[6] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for Internet databases. *SIGMOD* 2000, 379–390.

[7] G. Cormode et al. Holistic UDAFs at streaming speeds. *SIGMOD* 2004, 35–46.

[8] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: High performance network monitoring with an SQL interface. *SIGMOD* 2003, 647–651.

[9] M. Denny and M. Franklin. Predicate result range caching for continuous queries. *SIGMOD* 2005, 646–657.

[10] F. Fabret et al. Filtering algorithms and implementation for very fast publish/subscribe systems. *SIGMOD* 2001, 115–126.

[11] M. Garey and D. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[12] M. Karnaugh. The map method for synthesis of combinational logic circuits. *Journal of Symbolic Logic*, 20(2):197, 1955.

[13] S. Krishnamurthy, M. Franklin, J. Hellerstein, and G. Jacobson. The case for precision sharing. *VLDB* 2004, 972–986.

[14] H.-S. Lim et al. Continuous query processing in data streams using duality of data and queries. *SIGMOD* 2006, 313–324.

[15] S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. *SIGMOD* 2002, 49–60.

[16] R. Motwani et al. Query processing, approximation, and resource management in a data stream management system. *CIDR* 2003, 245–256.

[17] K. Munagala, U. Srivastava, and J. Widom. Optimization of continuous queries with shared expensive filters. *PODS* 2007, 215–224.

[18] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2):1–67, 2005.

[19] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. *SIGMOD* 2000, 249–260.

[20] T. Sellis. Multiple-query optimization. *ACM Trans. Database Sys.*, 13(1):23–52, 1988.

[21] N. Tatbul et al. Load shedding in a data stream manager. *VLDB* 2003, 309–320.

[22] K.-L. Wu, S.-K. Chen, and P. Yu. Interval query indexing for efficient stream processing. *CIKM* 2004, 88–97.

[23] R. Zhang, N. Koudas, B. C. Ooi, and D. Srivastava. Multiple aggregations over data streams. *SIGMOD* 2005, 299–310.

Appendix

We now prove the NP-hardness of minimizing the number of bits required to represent the prefilter predicates. This problem may be reduced to finding a bipartite graph covering by complete bipartite subgraphs, abbreviated CCBS, defined as follows.

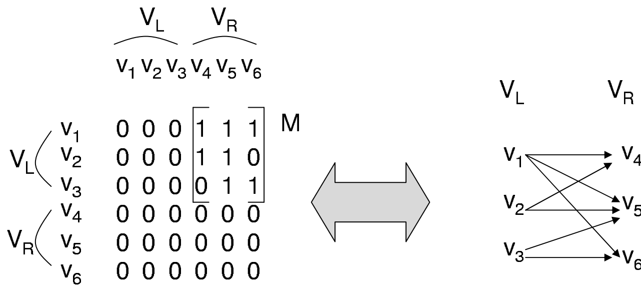

Figure 11: Transforming an instance of CCBS to an instance of rectangle covering.

Definition 3 *Let $G(V, E)$ be a bipartite graph and K be a positive integer such that $K \leq |E|$. We define the decision problem of "covering by complete bipartite subgraphs" (CCBS) as follows: are there $k \leq K$ subsets $V_1, V_2, \ldots, V_k$ of V such that each V_i induces a complete subgraph of G and such that for each edge $\{u, v\} \in E$ there is some V_i that contains both u and v?*

Theorem 1 *CCBS is NP-complete [11].*

We prove the following theorem, from which it follows that our original problem is NP-hard.

Theorem 2 *Let M be a Boolean matrix with p rows and q columns. The following "rectangle covering" decision problem is NP-complete: given a positive integer K such that $K \leq p$, is there a rectangle covering (as defined in Section 2.2) of M with size less than or equal to K?*

First, we observe that the rectangle covering problem is in NP as we can verify if a given set of rectangles is a covering of M in polynomial time. Next, let $\{G(V, E), K\}$ be an instance of CCBS and assume that G is represented by an adjacency matrix; i.e., $G(i, j) = 1$ if there exists an edge from vertex i to vertex j, and zero otherwise. Given that G is bipartite, we can divide V into two disjoint sets, call them V_L and V_R. Without loss of generality, assume that G is a directed graph such that all the edges originate at a vertex in V_L and terminate at a vertex in V_R. Furthermore, assume that all the vertices in V_L are ordered first, followed by the vertices in V_R. This way, all the non-zero entries of G are in its upper-right quadrant. The transformation from CCBS to rectangle covering is simple (and clearly polynomial-time): we assign M to be the upper-right quadrant of G. An example is shown in Figure 11.

To complete the proof, we show that G has a complete bipartite subgraph covering of size K if and only if M has a rectangle covering of size K. First, suppose that G has a complete bipartite subgraph covering of size K and let G' be any one of these K subgraphs. Since G' is complete and bipartite, it must link every vertex in some non-empty set $V'_L \in V_L$ with every vertex in some non-empty set $V'_R \in V_R$. Hence, G' induces a rectangle in G with V'_L and V'_R as its sets of rows and columns, respectively. Since every edge of G is contained in at least one subgraph forming the covering, every non-zero entry in G is contained in at least one corresponding rectangle. Since M is contained in G, the K subgraphs that cover G correspond to K rectangles that also form a rectangle covering of M.

Conversely, assume that M has a rectangle covering of size K. As per our transformation, every rectangle in M is defined by sets of rows and columns corresponding to subsets of V_L and V_R, respectively. Hence, every rectangle in M induces a complete bipartite subgraph of G. Since there exist K rectangles that cover every non-zero entry of M (and also of G), it follows that the K corresponding (complete bipartite) subgraphs cover every edge of G.