

Scalable Continuous Query Processing by Tracking Hotspots

Pankaj K. Agarwal Junyi Xie Jun Yang Hai Yu
Department of Computer Science
Duke University
{pankaj, junyi, junyang, fishhai}@cs.duke.edu

ABSTRACT

This paper considers the problem of scalably processing a large number of continuous queries by tracking common user interests. We propose a flexible framework with novel data structures and algorithms for group-processing and indexing hundreds of thousands of continuous queries. In our approach, we first introduce grouping by *stabbing set* to exploit clusterness of user interests, and propose to track hotspots in queries by partitioning them into groups and accordingly apply different processing strategies. To maintain partitions dynamically we present efficient algorithms which maintain nearly optimal partitions in nearly amortized logarithmic time. We illustrate that hotspots enables us to scalably process large numbers of continuous select-join and band-join queries, which are much more challenging than simple range selection queries. We also show how to use hotspots to build a nearly optimal histogram for intervals in linear time. Finally extensive experiments demonstrate that our approach improves system throughput by orders of magnitudes.

1. INTRODUCTION

Continuous query processing has attracted much interest in the database community recently because of its wide range of traditional and emerging applications, e.g., trigger and production rule processing [20, 12], data monitoring [4], stream processing [19], and publish/subscribe systems [16, 6, 18]. In contrast to traditional query systems, in which each query is run once against a snapshot of the database, continuous query systems support standing queries that continuously generate new results (or changes to results) as data continues to arrive in a stream. In this paper we propose a novel technique for indexing and processing continuous queries, in the hope to address the increasing challenge of scalability in continuous query processing systems.

Challenge of scalability. Formally, a *continuous query* defined by a relational expression Q issued over a database state \mathcal{D}_0 initially returns $Q(\mathcal{D}_0)$; then, for each subsequent database update that changes the database state from \mathcal{D}_{i-1} to \mathcal{D}_i , the query needs to return the changes between $Q(\mathcal{D}_i)$ and $Q(\mathcal{D}_{i-1})$. How can a continuous query processing system handle thousands or even millions

of such continuous queries in a scalable way? For each incoming data tuple, the system needs to identify the subset of continuous queries whose results are affected by the tuple, and compute changes to these results. If there are many continuous queries, a brute-force approach that processes each of them in turn will be inefficient and unable to meet the response-time requirement of most applications.

A powerful observation made by recent work on scalable continuous query processing is the interchangeable roles of queries and data. Continuous queries can be treated as data, while each data tuple can be treated as a query requesting the subset of continuous queries affected by the tuple. Thus, it is natural to apply indexing and query processing techniques traditionally intended for data to continuous queries. For example, many index structures have been applied to continuous queries to support efficient identification of affected queries without scanning through the whole set (e.g., [12] and others). In particular, consider range-selection queries of the form $\sigma_{a_i \leq A \leq b_i} R$, where A is an attribute of relation R and a_i, b_i are query parameters. These queries can be indexed as a set of intervals $\{[a_i, b_i]\}$ using, for example, interval tree [8] or interval skip list [11]. Given an insertion r into R , the set of affected queries are exactly those whose intervals are *stabbed* by $r.A$ (i.e., contain $r.A$). With an appropriate index, a *stabbing query*, which returns the subset of all intervals stabbed by a given point, can be answered in logarithmic time.

However, for complex continuous queries such as continuous joins, the problem of scalable processing becomes a real challenge, because these queries act over two or more data streams instead of a single data stream. As far as we know, most existing work on indexing relational continuous queries has only focused on simple selection condition, or conjunction of selection conditions, and there has been little work on how to scalably index complex continuous queries especially continuous joins, which are an equally important class of queries and building blocks of more complicated stateful subscriptions in large scale publish/subscribe systems.

High-level idea. We propose a novel technique for indexing and processing continuous queries, which is powerful enough to handle complex queries such as continuous joins in a scalable way. The main idea of our approach is to exploit clustering patterns in the set of continuous queries. For example, consider continuous queries issued by stock traders for monitoring the market. Suppose these queries include selections that restrict the stocks of interest to those with price/earning ratio within given ranges. We expect many of these price/earning ratio ranges to overlap significantly (though not necessarily to be identical), perhaps with a high-density cluster at low price/earning ratios because traders tend to be interested in stocks with good value.

Such clustering patterns commonly arise in the continuous query

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, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

setting. Following this observation, suppose that we first cluster the set of continuous queries based on the similarity of their query ranges. Then like in the above stock trader example, one may be able to identify a few number of clusters (or *hotspots*) such that the majority of the continuous queries fall into one of these clusters. Let us call queries in those clusters *hotspot queries*, and the remaining queries *scattered queries*. Our idea is then to index hotspot queries and scattered queries separately. The key is that, because hotspot queries in each cluster share similarity in their query ranges, they are allowed to be indexed in a very structured manner so that they can be processed much faster than usual. For scattered queries, however, one may apply a relatively inefficient processing method because hopefully there will be few such queries. Overall, we gain a significant speedup in processing all continuous queries as the majority of them are processed very efficiently.

Note that our approach naturally leads to faster processing for more clustered query ranges, although in the worst case (i.e., when the query ranges are not clustered, which is unlikely to happen in most continuous query applications) it may degrade into a traditional processing method, which is simply the best we can do because there is no opportunity for share processing at all.

Contributions. To materialize the above idea, we will address two main technical issues: (1) how to identify hotspot queries and their corresponding clusters, and keep track of these clusters when continuous queries are inserted into or deleted from the system; (2) how similarity in query ranges can be exploited so as to index the hotspot queries in each cluster in an efficient manner. The first issue is discussed in Section 2. The second issue depends on specific applications and is illustrated by three representative examples in Section 3.

In particular, the main contributions of this paper are as follows:

- In Section 2, we introduce the notions of *stabbing partition* and *stabbing set index* (SSI for short) as a tool to discover and exploit the clustering patterns of continuous queries. We further introduce the notion of *hotspots* to identify hotspot queries from the clustering, and present efficient algorithms to keep track of the hotspots when continuous queries are constantly inserted into and deleted from the system.
- In Section 3, we show how similarity in the query ranges in each hotspot can be exploited so as to process hotspot queries more efficiently. This is done by materializing the above general SSI scheme for specific types of continuous queries. We give three representative examples:
 - (1) indexing continuous band joins [9] whose join conditions check whether the difference between two join attribute values fall within some range;
 - (2) indexing continuous equality joins with different local range selections; and
 - (3) building a nearly optimal histogram in *linear* time for selectivity estimation.
- Finally in Section 4, we demonstrate through extensive experiments that our new algorithms and processing framework are very effective and deliver *significantly* better performance than traditional approaches for processing a large number of continuous queries.

We believe that our techniques are likely applicable to more general query evaluation contexts. In this paper we focus our attention to continuous query processing and leave its application to other scenarios as future work.

2. THE HOTSPOT-TRACKING SCHEME

Consider a set I of continuous queries whose query ranges are defined over a numerical attribute A . Intuitively, if many query ranges of I contain some value $x \in A$, then x is likely to be a “hotspot” for this set of continuous queries.¹ In general there could be several hotspots for I , depending on the distribution of the query ranges.

As continuous queries are inserted or deleted over time, the hotspots may also evolve. For example, people tend to pay more attention to high temperatures in summer, but more to low temperatures when winter comes. Therefore we need an efficient mechanism to keep track of the evolution of the hotspots. The main body of this section is dedicated to this task.

2.1 Stabbing Partition and Stabbing Set Index

We begin by introducing some tools for discovering and exploiting the clustering patterns of a set of intervals.

DEFINITION 1. Let I be a set of intervals. A *stabbing partition* of I is a partition of the intervals of I into disjoint groups I_1, I_2, \dots, I_τ such that within each group I_j , a common point p_j stabs all intervals in this group (in other words, the common intersection of all intervals in this group is nonempty). We call τ the *stabbing number* (or *size*) of this stabbing partition, and p_j the *stabbing point* of group I_j . The set $P = \{p_1, \dots, p_\tau\}$ is called a *stabbing set* of I .

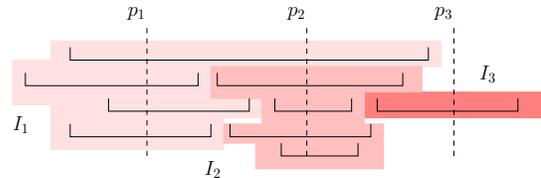


Figure 1: A stabbing partition of 10 intervals. I_1 and I_2 are 0.4-hotspots.

An example of the stabbing partition is shown in Figure 1. It is not hard to see that an optimal stabbing partition of a set of intervals that results in the fewest number of groups (i.e., τ is minimized) can be computed in a greedy manner, as follows. We scan the intervals in increasing order of their left endpoints, while maintaining a list of intervals we have seen. As soon as we encounter an interval that does not overlap with the common intersection of the intervals in our list, we output all intervals in our list as a group, and choose any point in their common intersection as the stabbing point for this group. The process then continues with the list containing only the newly encountered interval. The cost of this procedure is dominated by sorting the intervals by their left endpoints. We refer to the resulting stabbing partition of I as its *canonical stabbing partition*. Note that the canonical stabbing partition has the smallest possible stabbing number, which we shall denote by $\tau(I)$. We state the above fact as a lemma for future use.

LEMMA 1. *Given a set I of n intervals, the canonical stabbing partition of I , whose size is $\tau(I)$, can be computed by the greedy algorithm in $O(n \log n)$ time.*

We next briefly introduce the general framework of *stabbing set index* (SSI for short), which is able to exploit the clustering patterns of continuous queries for more efficient processing. It will later be

¹This is the one-dimensional case. For multi-dimensional query ranges, one can project them to each dimension and talk about hotspots in each dimension.

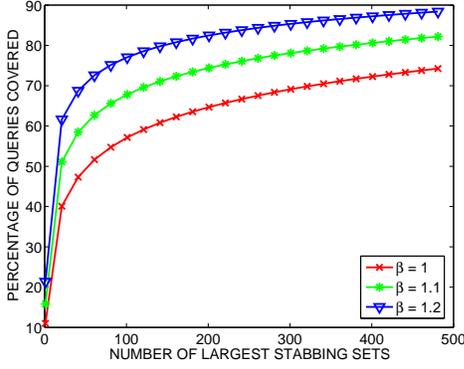


Figure 2: Hotspot coverage in Zipf distribution.

materialized for specific types of continuous queries in Section 3. To index a set of continuous queries, SSI works by first deriving a set I of intervals from these queries, one interval for each query, and computing a stabbing partition \mathcal{J} of I . SSI stores the stabbing points p_1, \dots, p_τ in sorted order in a search tree. Furthermore, for each group $I_j \in \mathcal{J}$, SSI maintains a separate data structure on the set of continuous queries corresponding to the intervals of I_j , which can be as simple as a sorted list, or as complex as an R-tree. Thus SSI is completely agnostic about the underlying data structure used, which enables us to apply SSI to different types of continuous queries. Intuitively, the fact that intervals within the same group are stabbed by a common point enables us to process the set of queries corresponding to these intervals more efficiently by “sharing” processing cost across them.

Note. As mentioned in the introduction, we actually only apply SSI to the subset of large clusters in the stabbing partition (i.e., they correspond to hotspot queries) instead of the entire set, because it attains little benefit on scattered queries and has to otherwise pay extra maintenance cost for them. The scattered queries will be taken care of by traditional algorithms.

2.2 Tracking Hotspots

User interests are not only clustered, but the clustering is also more or less unbalanced. Let us consider the following simple example. Suppose that user interests follow a Zipfian distribution, which is widely recognized to model popularity rankings such as website popularity or city populations. In particular, if we regard each stabbing group as a group of users interested in a common hotspot, Zipf’s law states that the number of queries within a stabbing group is roughly inversely proportional to its rank in popularity. That is, the number n_k of queries in the k -th largest group is proportional to $k^{-\beta}$, where β is a positive constant close to 1. Suppose there are a total number of 5000 groups in a stabbing partition. Figure 2 shows the percentage of queries covered by top- k largest stabbing groups out of all 5000 stabbing groups if the group sizes are governed by a Zipfian distribution with parameter $\beta \in [1.0, 1.2]$. From this figure we can see that top-500 largest stabbing groups (10% of all groups) cover about 70% of all queries when $\beta = 1$, and the coverage increases with a larger β .

Motivated by the above example, we next introduce the notion of α -hotspots.

DEFINITION 2. Let $\alpha > 0$ be a fixed parameter. Suppose $\mathcal{J} = \{I_1, \dots, I_\tau\}$ is a stabbing partition of I . A group $I_i \in \mathcal{J}$ is called an α -hotspot if $|I_i| \geq \alpha|I|$. An interval of I is called a *hotspot*

interval (with respect to \mathcal{J}) if it falls into an α -hotspot, and is called a *scattered interval* otherwise (see Figure 1).

In other words, if we think of the intervals in I as query ranges of the continuous queries, then an α -hotspot I_i corresponds to a collection of at least α fraction of the continuous queries whose query ranges contain the particular value p_i , where p_i is the stabbing point of I_i . Note that the number of α -hotspots is at most $1/\alpha$ by definition.

It is quite easy to identify all the hotspots once a stabbing partition \mathcal{J} of I is given. We next turn our attention to the problem of tracking hotspots as intervals in I are being inserted or deleted over time. When designing such a hotspot-tracking scheme, one needs to keep the following two issues in mind:

- (1) Note that the definition of α -hotspots depends on the specified stabbing partition \mathcal{J} of I . In order to extract meaningful hotspots from I , it is important to require that the size of \mathcal{J} is as small as possible, because intuitively small stabbing partitions provide more accurate pictures on how the intervals in I are clustered. Thus, to keep track of α -hotspots as intervals are inserted into or deleted from I , one needs to maintain a stabbing partition of I of size close to $\tau(I)$.
- (2) Let $S \subseteq I$ denote the set of all scattered intervals, and let $H = I \setminus S$ denote the set of all hotspot intervals. As the hotspots of I evolve over time, intervals may move into S (from H) or out of S (into H) accordingly. Since we will be using different indexing schemes for S and H , it is desirable for efficiency reasons to minimize the number of intervals that move in or out of S at each update.

We next describe an algorithm for tracking hotspots that takes care of both issues. Specifically, let $\varepsilon, \alpha > 0$ be fixed parameters; the algorithm will maintain a stabbing partition \mathcal{J} of I and a partition of \mathcal{J} into two sets \mathcal{J}_H and $\mathcal{J}_S = \mathcal{J} \setminus \mathcal{J}_H$ that satisfy the following three invariants all the time:

- (I1) \mathcal{J}_H contains all α -hotspots of \mathcal{J} , and possibly a few $(\alpha/2)$ -hotspots, but nothing more. Hence, $|\mathcal{J}_H| \leq 2/\alpha$;
- (I2) The size of \mathcal{J} is at most $(1 + \varepsilon)\tau(I) + 2/\alpha$;
- (I3) Let S denote the set of intervals in the groups of \mathcal{J}_S . Then the amortized number of intervals moving into or out of S per update is $O(1)$ (in fact, at most 5).

We need the following lemma, which says that one can maintain a stabbing partition of I of size close to $\tau(I)$ in amortized logarithmic time per update. The proof of this lemma as well as its efficient implementation and practical variant are presented in Section 2.3.

LEMMA 2. Let $\varepsilon > 0$ be a fixed parameter. We can maintain a stabbing partition of I of size at most $(1 + \varepsilon)\tau(I)$ at all times. The amortized cost per insertion and deletion is $O(\varepsilon^{-1} \log |I|)$.

The hotspot-tracking algorithm works as follows. At any time, we implicitly maintain a stabbing partition \mathcal{J} of I by maintaining a partition of \mathcal{J} into two sets \mathcal{J}_H and $\mathcal{J}_S = \mathcal{J} \setminus \mathcal{J}_H$. We use S to denote the set of intervals falling into the groups of \mathcal{J}_S , and $H = I \setminus S$ to denote the set of intervals falling into the groups of \mathcal{J}_H . Hence, \mathcal{J}_S is a stabbing partition of S , and \mathcal{J}_H is a stabbing partition of H . Initially when $I = \emptyset$, we have $\mathcal{J} = \emptyset$, $\mathcal{J}_H = \mathcal{J}_S = \emptyset$, and $S = H = \emptyset$. A schematic view of the algorithm is depicted in Figure 3.

Insertion. When an interval γ is inserted into I , we first check if γ can be added to any group $I_i \in \mathcal{J}_H$, such that the common

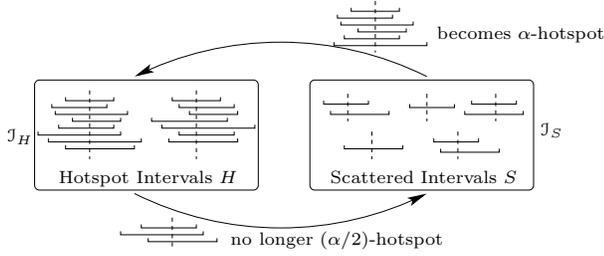


Figure 3: Schematic view of the hotspot-tracking algorithm.

intersection of the intervals in that group remains nonempty after adding γ . (This can be done brute-force in $O(1/\alpha)$ time, by maintaining the common intersection of each group. We omit the details.)

If there indeed exists such a group $I_i \in \mathcal{J}_H$, we simply add γ into I_i and are done. If there is no such group, we add γ into the set S , and then use the algorithm of Lemma 2 to update the stabbing partition of S , i.e., \mathcal{J}_S . As a consequence, the sizes of some groups in \mathcal{J}_S may become $\geq \alpha|I|$. We “promote” all such groups of \mathcal{J}_S into \mathcal{J}_H (because they become α -hotspots). Consequently, intervals in these groups should be moved out of S . We maintain the stabbing partition \mathcal{J}_S of S by deleting these intervals from S one by one and using Lemma 2 to update \mathcal{J}_S . (But in practice, it might be unnecessary to use Lemma 2 to update \mathcal{J}_S , as the intervals are moved out of S in groups.)

Note that after an insertion, the size of I is increased by one. Therefore, the sizes of some groups in \mathcal{J}_H may become $< (\alpha/2)|I|$. We “demote” all such groups of \mathcal{J}_H into \mathcal{J}_S (because they are no longer $(\alpha/2)$ -hotspots). Consequently, the intervals in such groups are moved into S . We again use Lemma 2 to update \mathcal{J}_S by inserting these intervals into S one by one. Note that when these insertions are finished, some groups in \mathcal{J}_S might again become new α -hotspots, in which case we “promote” these groups into \mathcal{J}_H as done in the previous paragraph.

Deletion. When an interval γ is deleted from I , the situation is somewhat symmetric to the case of insertion. We first check whether γ is contained in some group of \mathcal{J}_H . (This can be done in constant time by maintaining appropriate pointers from intervals to groups.)

If there indeed exists such a group $I_i \in \mathcal{J}_H$, we remove γ from this group. The removal might make I_i no longer an $(\alpha/2)$ -hotspot (note, however, the other groups in \mathcal{J}_H remain $(\alpha/2)$ -hotspots because their sizes do not change but the size of $|I|$ decreases by one.) In this case, we “demote” I_i into \mathcal{J}_S by inserting the intervals of I_i into S one by one and updating \mathcal{J}_S using Lemma 2. Otherwise, we know that $\gamma \in S$. We remove γ from S and update \mathcal{J}_S accordingly using Lemma 2.

After that, some groups in \mathcal{J}_S could become α -hotspots. We “promote” these groups into \mathcal{J}_H and remove their intervals from S as before.

THEOREM 1. *The above algorithm maintains the three invariants (I1)–(I3) at all times. Furthermore, the amortized cost for each update is $O(\alpha^{-1} + \varepsilon^{-1} \log |I|)$.*

PROOF. (I1) Obvious from the algorithm. Initially $\mathcal{J}_H = \emptyset$. The algorithm guarantees that: (i) whenever a group in \mathcal{J}_S becomes an α -hotspot, it is promoted to \mathcal{J}_H ; and (ii) when a group in \mathcal{J}_H is no longer an $(\alpha/2)$ -hotspot, it is demoted to \mathcal{J}_S .

(I2) Since we used Lemma 2 to maintain \mathcal{J}_S , we have $|\mathcal{J}_S| \leq (1 + \varepsilon)\tau(S) \leq (1 + \varepsilon)\tau(I)$. By (a), we also have $|\mathcal{J}_H| \leq 2/\alpha$. Hence,

$$|\mathcal{J}| = |\mathcal{J}_H| + |\mathcal{J}_S| \leq (1 + \varepsilon)\tau(I) + 2/\alpha.$$

(I3) We prove this invariant by an accounting argument. Specifically, we show how to deposit credits into the intervals of S and the groups of \mathcal{J}_H , for each insertion and deletion in I , so that the following two invariants hold:

- (i) at any time, each interval in S has one credit;
- (ii) when a group of \mathcal{J}_H is demoted to \mathcal{J}_S , it has at least $\alpha|I|$ credits.

If these two invariants hold, then we can pay the cost of moving intervals into or out of S by the credits associated with the relevant intervals, as follows. When an interval moves out of S (because of a promotion), we simply pay this move-out by the one credit deposited in that interval. When intervals are moved into S because of a demotion of a group $I_i \in \mathcal{J}_H$, note that the number of intervals in this group, $|I_i|$, is at most $(\alpha/2)|I|$. Since I_i has accumulated at least $\alpha|I|$ credits, we use $(\alpha/2)|I|$ credits to pay for each of the $|I_i|$ move-ins, and deposit the remaining $(\alpha/2)|I|$ credits to the intervals of I_i so that each interval has one credit (because they now belong to S and thus have to have one credit each by the first invariant). Overall, since each move-in or move-out can be paid by one credit, hence the total number of intervals moving into and out of S over the entire history is bounded by the total number of deposited credits.

How is the credit deposited for each update in I ? For each insertion γ , we deposit 2α credits to each group in \mathcal{J}_H . Furthermore, if γ does not fall into any group of \mathcal{J}_H (recall that in this case our algorithm inserts γ into S), we deposit another one credit to γ . Since $|\mathcal{J}_H| \leq 2/\alpha$ by (a), an insertion deposits at most $2\alpha \cdot (2/\alpha) + 1 = 5$ credits. For each deletion γ , if γ belongs to a group I_i in \mathcal{J}_H , we deposit two credits to the group I_i ; otherwise we deposit nothing. Clearly, if there are a total number of n insertions and deletions, the total number of credits deposited is $O(n)$. By the discussion of the previous paragraph, we then know that the amortized number of intervals moving into or out of S is $O(1)$ for each update.

It remains to show that (i) and (ii) hold for the above credit-deposit scheme. By the above discussion, we know that (i) is an easy consequence of (ii). So we only have to show (ii). Let $I_i \in \mathcal{J}_H$ be a group to be demoted. We know that I_i was promoted to \mathcal{J}_H at an earlier time.

Let x_0 be the size of I_i and n_0 be the size of I at the time of its promotion. Also let x_1 be the size of I_i and n_1 be the size of I at the time of its demotion. It is clear that $x_0 \geq \alpha n_0$ and $x_1 < (\alpha/2)n_1$. Suppose k insertions and ℓ deletions occur in I between the times of promotion and demotion. Then $n_1 = n_0 + k - \ell$.

Because the size of I_i changes from x_0 to x_1 . At least $x_0 - x_1$ deletions happened to the group I_i ($x_0 - x_1$ might be a negative number, but it does not hurt our argument). Therefore, at least $2(x_0 - x_1)$ credits are deposited into I_i by those deletions. Meanwhile, I_i also receives $2\alpha k$ credits from the k insertions. In total, I_i must have accumulated at least $2(x_0 - x_1) + 2\alpha k$ credits for the time period from its promotion to its demotion. Observe that

$$\begin{aligned} 2(x_0 - x_1) + 2\alpha k &\geq 2(\alpha n_0 - \alpha n_1/2) + 2\alpha k \\ &= 2\alpha n_0 - \alpha(n_0 + k - \ell) + 2\alpha k \\ &= \alpha n_0 + \alpha k + \alpha \ell \\ &\geq \alpha(n_0 + k - \ell) = \alpha n_1. \end{aligned}$$

In other words, I_i has accumulated at least αn_1 credits before its demotion, as desired.

Finally, the bound on the amortized cost is a corollary of (I3) and Lemma 2. Note that the cost for each update is dominated by the cost for updating \mathcal{J}_S using Lemma 2. Since the amortized number of intervals moving into and out of S is $O(1)$ per update, by Lemma 2, we know that the amortized cost for updating the stabbing partition \mathcal{J}_S of S is $O(\varepsilon^{-1} \log n)$. \square

2.3 Dynamic Stabbing Partitions

This section is devoted to an efficient implementation of Lemma 2. Because it is somewhat involved and is not a prerequisite for the subsequent discussions of the paper, this section can be skipped at the reader's discretion.

We first observe that if one were to maintain the smallest stabbing partition of I (such as the canonical stabbing partition) as intervals are inserted or deleted, then the stabbing partition of I may completely change after a very few number of insertions or deletions. (A simple example is omitted for brevity.) This motivates us to resort to a stabbing partition of *approximately* smallest size. More precisely, we want to maintain a partition of size at most $(1 + \varepsilon)\tau(I)$ for some parameter $\varepsilon > 0$, where recall that $\tau(I)$ is the size of the smallest stabbing partition of I . Although the quality of the stabbing partition is compromised, the benefit of resorting to an approximation is that the cost required for maintaining such a relaxed partition is much lower than for maintaining the smallest one.

Typically we choose ε to be a small constant. The value of ε can be used as a tunable parameter to achieve flexible tradeoffs between the quality of the stabbing partition and the maintenance cost: a smaller ε results in a better stabbing partition, but also increases the maintenance cost. Next we describe in detail how to maintain the stabbing partitions.

A simple strategy. We sketch a lazy maintenance strategy that guarantees the quality of the stabbing partition. It is very easy to implement and works reasonably well in practice, but may perform badly in the worst case.

Let I be a set of n intervals, and $\varepsilon > 0$ be a fixed positive parameter. The lazy strategy works as follows. We begin with the canonical stabbing partition \mathcal{J} of I of size $\tau_0 = \tau(I)$ as well as a corresponding stabbing set P . When a new interval γ is inserted into I , we simply pick a point $p_\gamma \in \gamma$ and let $P = P \cup \{p_\gamma\}$; we also create a singleton group $\{\gamma\}$ and add it to \mathcal{J} . When an interval γ is deleted from I , suppose that γ belongs to some group $I_i \in \mathcal{J}$. We then remove γ from I_i , and if I_i becomes empty after the removal of γ , we also remove I_i from \mathcal{J} and the stabbing point of I_i from P . After $\varepsilon\tau_0/h$ number of insertions and deletions, where $h = \varepsilon + 2$ (the reason for this choice of h will become clear shortly), we trigger a *reconstruction stage*: we use Lemma 1 to reconstruct the canonical stabbing partition (whose size is $\tau(I)$) for the current I , which takes $O(n \log n)$ time.

LEMMA 3. *The above procedure maintains a stabbing partition of size at most $(1 + \varepsilon)\tau(I)$ at all times.*

PROOF. Clearly, the procedure maintains a stabbing partition \mathcal{J} of size at most $(1 + \varepsilon/h)\tau_0$ at all times, because each insertion increases the size of \mathcal{J} by at most one, and each deletion does not increase the size of \mathcal{J} . On the other hand, note that inserting an interval into I does not decrease $\tau(I)$, and deleting an interval from I may decrease $\tau(I)$ by at most one. Therefore $\tau(I) \geq (1 - \varepsilon/h)\tau_0$ at any time before the reconstruction stage. Hence,

$$(1 + \varepsilon)\tau(I) \geq (1 + \varepsilon)(1 - \varepsilon/h)\tau_0 = (1 + \varepsilon/h)\tau_0,$$

where the last equality follows from the choice of $h = \varepsilon + 2$. This

implies that the procedure always maintains a stabbing partition of size at most $(1 + \varepsilon)\tau(I)$. \square

The basic strategy can be refined in several ways to improve its efficiency at runtime. For example, for a newly inserted interval γ , if there already exists a point p_i in the current stabbing set that stabs γ , and suppose p_i is the stabbing point for the group I_i , then we can simply add γ into I_i , instead of creating a new singleton group $\{\gamma\}$ in the stabbing partition. A more careful implementation is to maintain the common intersection of each group, instead of just a single stabbing point. For each new insertion γ , we check whether there exists a group whose common intersection overlaps with γ , and if so, add γ to that group.

The condition for triggering a reconstruction stage (i.e., when the total number of insertions and deletions reaches $\varepsilon\tau_0/(\varepsilon + 2)$) can also be relaxed. Let \bar{I} denote the set of intervals after the last reconstruction and $\tau_0 = \tau(\bar{I})$. Suppose that m intervals have been deleted from \bar{I} so far since the last reconstruction (the total number of deletions so far could be larger because some intervals may be inserted and subsequently deleted), then we invoke a reconstruction stage only if $|P| \geq (1 + \varepsilon)(\tau_0 - m)$, where $|P|$ is the size of the maintained stabbing set at that time. The proof of correctness follows the same argumentation as in the proof of Lemma 3. Note that it is weaker than the old trigger condition, and hence leads to less frequent invocations of reconstruction stages.

A refined algorithm. The amortized cost per insertion and deletion in the above simple strategy is $O(n \log n / (\varepsilon\tau_0))$. In the full version of the paper [1], we describe a refined algorithm for maintaining the stabbing partition in $O(\varepsilon^{-1} \log n)$ amortized time per update, by a careful implementation of the reconstruction stage in above simple strategy. Moreover, each insertion and deletion update only *one* group in the stabbing partition. In the general SSI scheme, changes in the stabbing partition often need to be propagated to the data structures associated with the groups of the stabbing partition. Our algorithm therefore requires infrequent propagations and is suitable for real-time applications. Due to space constraint we only state the main result and leave the detailed algorithm and its pseudocode to the full version of this paper [1].

THEOREM 2. *Let $\varepsilon > 0$ be a fixed parameter. The above algorithm maintains a stabbing partition of I of size at most $(1 + \varepsilon)\tau(I)$ at all times. The amortized cost per insertion and deletion is $O(\varepsilon^{-1} \log |I|)$. Before the reconstruction stage, each insertion or deletion affects at most one group in the stabbing partition.*

3. APPLICATIONS

In this section we give three representative applications of our stabbing set index (SSI) and hotspot-tracking schemes in speeding up continuous query processing. Each of these applications has a somewhat different flavor, and achieves notable performance improvement over traditional processing techniques. This list of applications is not meant to be exhaustive, but should help illustrate the main idea of our techniques.

The first two subsections demonstrate how SSI enables scalably processing a set of continuous select-join queries by exploiting common interests. In particular, we consider the following two types of continuous queries over relations $R(A, B)$ and $S(B, C)$:

Equality join: $\sigma_{A \in \text{range } A} R \bowtie_{R.B=S.B} \sigma_{C \in \text{range } C} S$

Band join: $R \bowtie_{S.B-R.B \in \text{range } B} S$

In equality join, the query parameters *range* A and *range* C are local range selections over numeric domains of $R.A$ and $S.C$, respectively; in band join, *range* B is a join condition between $R.B$

and $S.B$. These two types of queries are widely used in large scale pub/sub systems, and are building blocks of more complicated queries. We give two illustrative examples of both queries.

Example 1. Consider a listing database for merchants with the following two relations: $\text{Supply}(\text{suppId}, \text{prodId}, \text{quantity}, \dots)$, and $\text{Demand}(\text{custId}, \text{prodId}, \text{quantity}, \dots)$. Merchants are interested in tracking supply and demand for products. Each merchant, depending on its size and business model, may be interested in different ranges of supply and demand quantities. For example, wholesalers may be interested in supply and demand with large quantities, while small retailers may be interested in supply and demand with small quantities. Thus, each merchant defines a continuous query

$$\sigma_{\text{quantity} \in \text{range} S_i} \text{Supply} \bowtie \sigma_{\text{quantity} \in \text{range} D_i} \text{Demand},$$

which is an equality join (with equality imposed on prodId) with local range selections.

Example 2. For an example of band joins, consider a monitoring system for coastal defense with relations $\text{Units}(\text{id}, \text{model}, \text{pos}, \dots)$ and $\text{Targets}(\text{id}, \text{type}, \text{pos}, \dots)$, where pos specifies points on the one-dimensional coast line. We want to get alerted when a target appears within the effective range of a unit. For each class of units, e.g., gun batteries, a continuous query can be defined for this purpose: e.g.,

$$\sigma_{\text{model} = \text{'BB'}} \text{Units} \bowtie_{\text{Units.pos} - \text{Targets.pos} \in \text{range}} \sigma_{\text{type} = \text{'surface'}} \text{Targets}.$$

where BB is a fictitious model of gun batteries, range is the firing range of this model, and the selection condition on Targets captures the fact that this model is only effective against surface targets. This continuous query is a band join with local selections. Note that for different classes of units, the band join conditions are different because of different firing ranges.

3.1 Band Joins

We first consider the problem of processing a group of continuous band joins, each of the form $R \bowtie_{S.B - R.B \in \text{range} B_i} S$. When a new R -tuple r arrives, we need to identify the subset of continuous queries whose query results are affected by r and compute changes to these results. The case in which a new S -tuple arrives is symmetric.

Previous approaches. We first note that existing techniques based on sharing identical join operations [6] do not apply to band joins because each $\text{range} B_i$ can be different. The state-of-art approach to handle continuous queries with different join conditions is proposed by PSoup [5], where multiple “hybrid structures” (i.e., data-carrying, partially processed join queries) are applied to a database relation together as a group, by treating these structures as a relation to be joined with the database relation.

Following the PSoup approach, we can process each new R -tuple r as follows. First, we “instantiate” the band join conditions by the actual value of $r.B$, resulting in a set of selection conditions $\{S.B \in \text{range} B_i + r.B\}$ local to S . Then, this set of selections can be treated as a relation of intervals $\{\text{range} B_i + r.B\}$ and joined with S ; each S -tuple s such that $s.B$ stabs the interval $\text{range} B_i + r.B$ corresponds to a new result tuple rs for the i -th band join. Depending on which join algorithms to use, we have several possible strategies.

- **BJ-QOuter** (band join processing with queries as the outer relation) processes each interval $\text{range} B_i + r.B$ in turn, and uses an ordered index on $S(B)$ (e.g., B-tree) to search for S -tuples within the interval.

- **BJ-DOuter** (band join processing with data as the outer relation) utilizes an index on ranges $\{\text{range} B_i\}$ (e.g., priority search tree or external interval tree). For each S -tuple s , BJ-DOuter probes the index for ranges containing $s.B - r.B$.
- **BJ-MJ** (band join processing with merge join) uses the merge join algorithm to join the intervals $\{\text{range} B_i + r.B\}$ with S . This strategies requires that we maintain the intervals $\{\text{range} B_i\}$ in sorted order of their left endpoints (note that addition of $r.B$ does not alter this order), and that we also maintain S in sorted $S.B$ order (which can be done by an ordered index, e.g., B-tree, on $S(B)$). Otherwise, BJ-MJ requires additional sorting.

Clearly, all three strategies have processing times at least linear in the size of S or in the number of band joins (the detailed bounds are provided in Theorem 3 below), which may be unable to meet the response-time requirement of critical applications. The difficulty comes in part from the fact that each continuous band join has its own join condition, and at first glance it is not clear at all how to share the processing cost across different band joins. Our SSI-based approach overcomes this problem.

The SSI approach. We now present an algorithm, **BJ-SSI** (band join processing with SSI), based on an SSI for the continuous queries constructed on the band join ranges $\{\text{range} B_i\}$. The index structure is rather simple. Each group I_j in the SSI is stored in two sequences I_j^l and I_j^r : I_j^l stores all ranges in I_j in increasing order of their left endpoints, while I_j^r stores all ranges in I_j in decreasing order of their right endpoints. The total space of these sorted sequences is clearly linear in the number of queries. We also build a B-tree index on $S(B)$.

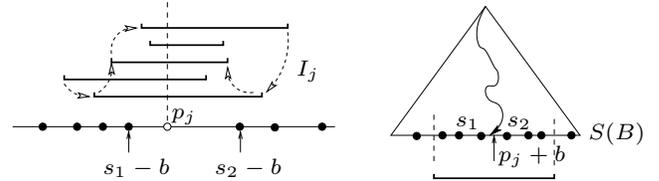


Figure 4: The SSI algorithm for band join processing. Arrows indicate the order in which the intervals are visited.

When a new R -tuple $r(a, b)$ is inserted, the problem is to identify all band joins that are affected and compute results for them. In terms of the ranges that we index in the SSI, we are looking for the set of all ranges $\text{range} B_i$ that are stabbed by some point $s.B - b$ where $s \in S$.

BJ-SSI processes the new R -tuple $r(a, b)$ in two steps: in the first step it finds all queries that are affected by r , and in the second step it returns the new results for each affected query.

(STEP 1) BJ-SSI proceeds for each group I_j in the SSI as follows. Using the B-tree index on $S(B)$, we look up the search key $p_j + b$, where p_j is the stabbing point for I_j . This lookup locates the two adjacent entries in the B-tree whose $S.B$ values s_1 and s_2 surround the point $p_j + b$ (or equivalently, $s_1 - b$ and $s_2 - b$ surround p_j , as illustrated in Figure 4). If either s_1 or s_2 coincides with $p_j + b$, then it is obvious that all queries in I_j are affected by the incoming update (at the very least the S -tuple with $B = p_j + b$ joins with r for all these queries). Otherwise, the exact subset of queries in I_j affected by the incoming tuple can be identified as follows (see left part of Figure 4): (1) We scan I_j^l in order up to the first query range with left endpoint greater than $s_1 - b$; all queries encountered

before this one are affected. (2) Similarly, we scan I_j^r in order up to the first query range with right endpoint less than $s_2 - b$; again, all queries encountered before this one are affected. The correctness of the above approach is guaranteed by the following lemma.

LEMMA 4. *The above procedure correctly returns the set of all affected continuous band joins in I_j .*

PROOF. Recall that all query ranges in I_j are stabbed by the point p_j . Any query range whose left endpoint is less than or equal to $s_1 - b$ must contain $s_1 - b$ (because it contains p_j); similarly, any query range whose right endpoint is greater than or equal to $s_2 - b$ must contain $s_2 - b$. On the other hand, query ranges whose left and right endpoints fall in the gap between $s_1 - b$ and $s_2 - b$ produce no new join result tuples, because s_1 and s_2 are adjacent in the B-tree on $S(B)$ and hence there is no S -tuple s such that $s.B \in (s_1, s_2)$. \square

(STEP 2) Once we have found the set of all affected queries in I_j , we can compute changes to the results of these queries as follows (see right part of Figure 4). Observe that the query interval of each affected continuous query in the group I_j covers a consecutive sequence of S -tuples, including either s_1 or s_2 . Therefore, to compute the new result tuples for each affected query, we can simply traverse the leaves of the B-tree index on $S(B)$, in both directions starting from the point $p_j + b$ (which we have already found earlier), to produce result tuples for this query. We stop as soon as we encounter a $S.B$ value outside the query range.

In summary, BJ-SSI has the following nice properties:

- (1) BJ-SSI never considers a tuple in S unless it contributes to some join result or happens to be closest to some stabbing point offset by b (there are at most two such tuples per group);
- (2) BJ-SSI never considers a band join query unless it will generate some new result tuple or it terminates the scanning of some I_j^l or I_j^r (again, there are at most two such queries per group).

In contrast, BJ-QOuter, BJ-DOuter, and BJ-MJ must scan either all queries or all tuples in S , many of which may not actually contribute any result. We conclude with the following theorem.

THEOREM 3. *Let n denote the number of continuous band joins, τ denote the stabbing number, m denote the size of S , and k denote the output size. The worst-case running times to process an incoming R -tuple are as follows:*

- *BJ-QOuter:* $O(n \log m + k)$;
- *BJ-DOuter:* $O(m \log n + k)$;
- *BJ-MJ:* $O(m + n + k)$.
- *BJ-SSI:* $O(\tau \log m + k)$;

SSI + Hotspot-tracking. Applying BJ-SSI to the set \mathcal{J}_H of Theorem 1 (i.e., the collection of hotspots), we immediately get an efficient algorithm for processing the subset of hotspot queries. Note that Theorem 1 (a) guarantees $|\mathcal{J}_H| \leq 2/\alpha$, hence by Theorem 3 (with $\tau \leq 2/\alpha$), we can then process all hotspot queries in $O(\alpha^{-1} \log m + k)$ time, which is a huge speedup in comparison to the other processing strategies.

3.2 Equality Joins with Local Selections

We now turn our attention to the problem of processing continuous equality joins with local selections, each of the form

$$\sigma_{A \in \text{range}A_i} R \bowtie_{R.B=S.B} \sigma_{C \in \text{range}C_i} S.$$

Each such query can be represented by a rectangle spanned by two ranges $\text{range}C_i$ and $\text{range}A_i$ in the two-dimensional product space $S.C \times R.A$, as illustrated in Figure 5. Suppose that a new R -tuple $r(a, b)$ has been inserted. In the product space $S.C \times R.A$, each tuple rs resulted from joining r with S can be viewed as a point on the line $R.A = a$ because these tuples have the same $R.A$ value (from r) but different $S.C$ values (from different S -tuple that join with r). We call these points *join result points*. To identify the subset of affected queries and compute changes to the results of these queries, our task reduces to reporting which query rectangles cover which join result points.

Previous approaches. When a new R -tuple r arrives, there are two basic strategies depending on the order in which we process joins and selections.

- *SJ-JoinFirst (select-join processing with join first)* proceeds as follows: (1) it first joins r with S ; (2) for each join result tuple, it checks the local selection conditions to see which continuous queries are affected. In more detail, the join between r and S can be done efficiently by probing an index on $S(B)$ (e.g., a B-tree) using $r.B$. For each join result tuple rs with $r.B = s.B$, we then probe a two-dimensional index (e.g., an R-tree) constructed on the set of query rectangles $\{\text{range}C_i \times \text{range}A_i\}$ with the point $(s.C, r.A)$. The subset of continuous queries that need to return rs as a new result tuple are exactly those whose query rectangles contain the point $(s.C, r.A)$.
- *SJ-SelectFirst (select-join processing with selection first)* proceeds as follows: (1) it first identifies the subset of continuous queries whose local selections on R are satisfied by the incoming tuple r ; (2) for each such query, it computes new result tuples by joining r with S and applying the local selection on S . In more detail, to identify the subset of continuous queries whose local selections on R are satisfied by r , we can use $r.A$ to probe an index on query ranges $\{\text{range}A_i\}$ (e.g., a priority search tree [8] or external interval tree [2]). To compute the new result tuples for each identified query with query range $\text{range}C_i$ on S , we can use an ordered index for S with composite search key $S(B, C)$ (e.g., a B-tree). We search the index for S -tuples satisfying $S.B = r.B \wedge S.C \in \text{range}C_i$.

Both SJ-JoinFirst and SJ-SelectFirst are prone to the problem of large intermediate results generated at step (1) of each algorithm. Consider the supply/demand example again. Suppose that our merchants are not interested in matching low-quantity supply with high-quantity demand (though many are interested in matching supply and demand that are both low in quantity). Further suppose that a particular product is in popular demand and mostly with high quantities. When a low-quantity supply source for this product appears, it will generate lots of joins (in the SJ-JoinFirst case) and satisfy local selections of many continuous queries (in the SJ-SelectFirst case), but very few continuous queries will actually be affected in the end. Therefore in this case, neither SJ-JoinFirst nor SJ-SelectFirst is efficient because of the large intermediate results generated at step (1).

The SSI approach. We now present our algorithm, *SJ-SSI (select-join processing with SSI)*, which gets around the aforementioned

problems of SJ-JoinFirst and SJ-SelectFirst by using an SSI for the continuous queries constructed on the local selection ranges $\{rangeC_i\}$, i.e., projections of the query rectangles onto the $S.C$ axis. (Here we focus on processing incoming R -tuples; to process incoming S -tuples, we would need a corresponding SSI constructed on $\{rangeA_i\}$.) Each group in the SSI is stored as an R-tree that indexes the member queries by their query rectangles. The total space of these data structures is linear in the number of queries since each query is stored only once in some group.

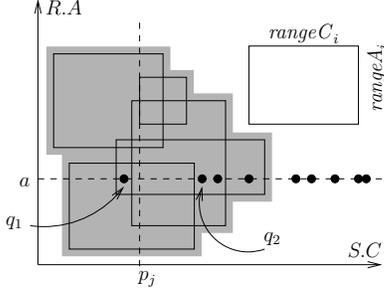


Figure 5: The SSI algorithm for processing equality joins with local selections.

To process an insertion r into R , for each group I_j with stabbing point p_j , we look for the search key $(r.B, p_j)$ in a B-tree index of table S on $S(B, C)$. This lookup locates the two joining S -tuples whose C values q_1 and q_2 are closest (or identical) to p_j from left and from right, respectively. Looking at Figure 5, they correspond to the two join result points (q_1, a) and (q_2, a) closest to (p_j, a) in the product space $S.C \times R.A$. We use these two join result points to probe the R-tree for group I_j . In the event that either q_1 or q_2 coincides with p_j , only one probe is needed. The following lemma establishes correctness of the above procedure.

LEMMA 5. *The query rectangles returned by the R-tree lookup constitute precisely the set of continuous queries in I_j that are affected by r .*

PROOF. Recall that by construction of the SSI, all queries in the group I_j intersects the line $S.C = p_j$. Any query in I_j that contains neither (q_1, a) nor (q_2, a) cannot possibly contain any join result point at all — such queries either do not intersect the line $R.A = a$ or happen to fall in the gap between q_1 and q_2 .

On the other hand, any query that contains either (q_1, a) or (q_2, a) is clearly affected and produces at least one of the two join result points. \square

Finally, observe that the query rectangle of each affected continuous query in the group I_j covers a consecutive sequence of join result points on the line $R.A = a$, including either q_1 or q_2 (see Figure 5). Therefore, to compute the new result tuples for each affected query, we can proceed as follows. For each query rectangle returned, we traverse the leaves of the B-tree on $S(B, C)$, in both directions starting from the entries for q_1 and q_2 , to produce all result tuples for this query. We stop as soon as we encounter a different $S.B$ value or a $S.C$ value outside the query range. This is similar to what we have done for band joins in the previous section.

SJ-SSI avoids the problems of SJ-JoinFirst and SJ-SelectFirst because of the following nice properties:

- (1) SJ-SSI never considers a join result point unless it is covered by some query rectangle or is closest to some stabbing point;
- (2) SJ-SSI never considers a query rectangle unless it covers some join result point.

To summarize, we give the complexity of SJ-JoinFirst, SJ-SelectFirst, and SJ-SSI in the following theorem.

THEOREM 4. *Let n denote the number of continuous equality joins, τ denote the stabbing number, m denote the size of S , and k denote the output size. Furthermore, let $g(n)$ denote the complexity of answering a stabbing query on an index of n two-dimensional ranges. The worst-case running times to process an incoming R -tuple are as follows:*

- *SJ-JoinFirst: $O(\log m + m'g(n) + k)$, where $m' \leq m$ is the number of S -tuples that join with the incoming tuple;*
- *SJ-SelectFirst: $O(\log n + n' \log m + k)$, where $n' \leq n$ is the number of queries whose local selections on R are satisfied by the incoming tuple;*
- *SJ-SSI: $O(\tau(\log m + g(n)) + k)$.*

SSI + Hotspot-tracking. Applying SJ-SSI to the set \mathcal{J}_H of Theorem 1 (i.e., the collection of hotspots), we immediately get an efficient algorithm for processing the subset of hotspot queries. Note that since $|\mathcal{J}_H| \leq 2/\alpha$, by Theorem 3 (with $\tau \leq 2/\alpha$), we can then process all hotspot queries in $O(\alpha^{-1}(\log m + g(n)) + k)$ time, which is in sharp contrast to the other two algorithms, whose running times are at the mercy of the size of the intermediate results m' or n' .

3.3 Histograms for Hotspots in Linear Time

In this section we consider the following problem, which can be used for estimating the number of continuous join queries whose local selection conditions are satisfied by an incoming tuple. Let I be a set of intervals. Given an $x \in \mathbb{R}$, we want to estimate how many intervals of I are stabbed by x . We denote by $f_I(x)$ be the number of intervals stabbed by x in I . The basic idea is clearly to build a histogram $h(x)$ (i.e., a step function) that approximates the function $f_I(x)$. Assuming that the distribution of the incoming tuple x is governed by a probability density function $\phi(x)$, then the mean-squared error between $h(x)$ and $f_I(x)$ can be written as

$$E^2(h, f_I) = \int |h(x) - f_I(x)|^2 \phi(x) dx.$$

Our goal is to find a histogram $h(x)$ with few break points that minimizes the above error. We assume that $\phi(x)$ is given: it can be acquired by standard statistical methods at runtime.

Previous approaches. Most known algorithms for the above problem or similar problems use dynamic programming, whose running time is polynomial but rather large [14, 15]. In contrast, our new algorithm below is simple and runs in nearly linear time. To be fair though, the dynamic-programming approaches usually guarantee to find an optimal solution (i.e., minimizing the error), while the histogram returned by our algorithm is nearly optimal. Nonetheless, since histograms are primarily for estimation purposes, an optimal histogram is not really necessary in practice.

Our approach. Our new approach radically differs from those dynamic-programming approaches, by taking advantage of the following main observations:

- (1) Scattered intervals in I do not contribute much to $f_I(x)$ and thus can be ignored.
- (2) Computing an optimal histogram for each α -hotspot in I can be reduced to a simple geometric clustering problem;

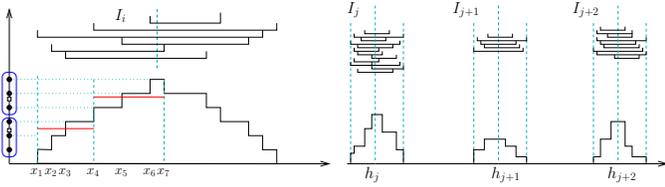


Figure 6: Reducing to a one-dimensional weighted k -means clustering problem.

The algorithm is simple to implement, modulo a standard one-dimensional k -means clustering subroutine.

In more detail, we first compute the canonical stabbing partition $\mathcal{J} = \{I_1, \dots, I_\tau\}$ for I as in Lemma 1, and then build a histogram for each of the α -hotspots of \mathcal{J} . The final histogram is obtained by summing up these (at most $1/\alpha$) histograms. Let p_i be the stabbing point of an α -hotspot $I_i \in \mathcal{J}$, and let $f_{I_i}^l$ (resp. $f_{I_i}^r$) be the part of the function f_{I_i} to the left (resp. right) of p_i . To compute the histogram $h_i(x)$ for a hotspot I_i , we compute two functions h_i^l and h_i^r to approximate $f_{I_i}^l$ and $f_{I_i}^r$ respectively, and then let $h_i(x) = h_i^l(x) + h_i^r(x)$.

We now focus on how to compute a histogram $h_i^l(x)$ with at most k buckets to minimize the error $E^2(h_i^l, f_{I_i}^l)$, where k is a given fixed parameter; the case for computing h_i^r is symmetric. Clearly, $f_{I_i}^l$ is a monotonically increasing step function (see Figure 6); let x_1, \dots, x_m be the break points of $f_{I_i}^l$. Assume without loss of generality that $k < m$.

LEMMA 6. *There is an optimal histogram with at most k buckets such that each bucket boundary passes through one of the break points x_1, \dots, x_m .*

PROOF. Take any optimal histogram whose bucket boundaries do not necessarily pass through those break points. Observe that no bucket completely lies between any two consecutive break points x_j and x_{j+1} ; otherwise one can expand the bucket to the entire interval $[x_j, x_{j+1}]$ and decrease the error. As such, there is at most one bucket boundary between x_j and x_{j+1} . This boundary can be moved to either x_j or x_{j+1} without increasing the error. Repeat this process for all such boundaries and we obtain a desired optimal histogram. \square

By the above lemma, it is sufficient to consider those histograms whose bucket boundaries pass through the break points x_1, \dots, x_m . For such a histogram h_i^l , suppose its bucket boundaries divide the break points into k groups:

$$\{x_{z_0+1}, \dots, x_{z_1}\}; \{x_{z_1+1}, \dots, x_{z_2}\}; \dots; \{x_{z_{k-1}+1}, \dots, x_{z_k}\},$$

where $z_0 = 0$ and $z_k = m$. Furthermore, let the value of h_i^l within the j -th bucket be a constant c_j , for $0 \leq j < k$. Then the error $E(h_i^l, f_{I_i}^l)$ can be written as

$$E^2(h_i^l, f_{I_i}^l) = \sum_{j=0}^{k-1} \sum_{\ell=z_{j+1}}^{z_{j+1}} |y_\ell - c_j|^2 \int_{x_\ell}^{x_{\ell+1}} \phi(x) dx, \quad (1)$$

where $y_\ell = f_{I_i}^l(x_\ell)$.

To find a histogram $h_i^l(x)$ that minimizes (1), we solve the following weighted k -means clustering problem in one dimension: Given a set of m points $y_1 = f_{I_i}^l(x_1), \dots, y_m = f_{I_i}^l(x_m)$, and a weight $w_\ell = \int_{x_\ell}^{x_{\ell+1}} \phi(x) dx$ for each point y_ℓ , find k centers c_1, \dots, c_k and an assignment of each y_ℓ to one of the centers so that the weighted k -means clustering cost is minimized (see left

part of Figure 6). We have the following lemma to establish the correctness of our algorithm.

LEMMA 7. *Minimizing (1) is equivalent to solving the above weighted k -means clustering problem.*

PROOF. Omitted for brevity. \square

Since typically the total amount of buckets allocated to the whole histogram is fixed, the remaining issue is how to assign available buckets to each α -hotspot I_i . One way to completely get around this problem is to map all points in each I_i into a one-dimensional space such that the points within each group are sufficiently far away from the points in other groups, as shown in the right part of Figure 6. Then we can run the k -means algorithm [13] on the whole point set to compute an ε -approximate optimal histogram in nearly linear time $O(n) + \text{poly}(k, 1/\varepsilon, \log n)$, which automatically assigns an appropriate number of buckets to each I_i . In practice we may wish to use the simpler iterative k -means clustering algorithm [10].

4. EXPERIMENTS

To compare our techniques against traditional processing techniques in terms of their scalability with a large number of continuous queries (hereafter also referred to as subscriptions), we built a query engine incorporating various algorithms discussed in previous sections. The algorithms are all implemented in Java SDK 1.4.2. All experiments were conducted on a Sun Blade 150 with a 650-MHz UltraSPARC-III processor and 512 MB of memory. We measured the query engine throughput, i.e., the number of update events that the engine is able to process per second, of different approaches. We excluded the output time from measurement since it is application-dependent and common to all approaches. We also measured the cost of updating associated data structures in all approaches.

Workload generation. We generated two synthetic tables $R(A, B)$ and $S(B, C)$, where B is the join attribute and A, C are the local selection attributes, all integer-valued. Each table contains 100,000 tuples indexed by standard B-trees. R is updated by an incoming stream of insertion events, whose A and B values are drawn uniformly at random from the domain. For tuples in S , their C values are uniformly distributed, while their B values follow a discretized normal distribution, in order to model varying join selectivity.

We created two sets of subscriptions, each with 100,000 subscriptions initially. The first set consists of equality joins with local selections discussed in Section 3.2 and the second set consists of band joins in Section 3.1. The midpoints of $\text{range}A_i$ follow a normal distribution that models varying event selectivity against local filters, and the midpoints of $\text{range}B_i$ and $\text{range}C_i$ are uniformly distributed. The lengths of all ranges are normally distributed. Users may insert new subscriptions, delete old subscriptions, and update existing subscriptions at runtime. Table 1 summarizes the data and workload parameters, where μ_i 's and σ_i 's can be used to adjust the selectivity of incoming updates against local selections and the clusteredness of subscription ranges.

Equality joins with local selections. We have implemented SJ-SSI (Section 3.2) and three alternative approaches: SJ-J(oinFirst), SJ-S(electFirst) (Section 3.2), and NAIVE, which first joins the new R tuple with S to generate an ordered list of result tuples, and then evaluates the local selections of each subscription over this intermediate result. NAIVE serves as a baseline for comparison; its cost is $O(\log m + n \log |S'| + k)$, where S' is the subset of S that joins with the new R tuple.

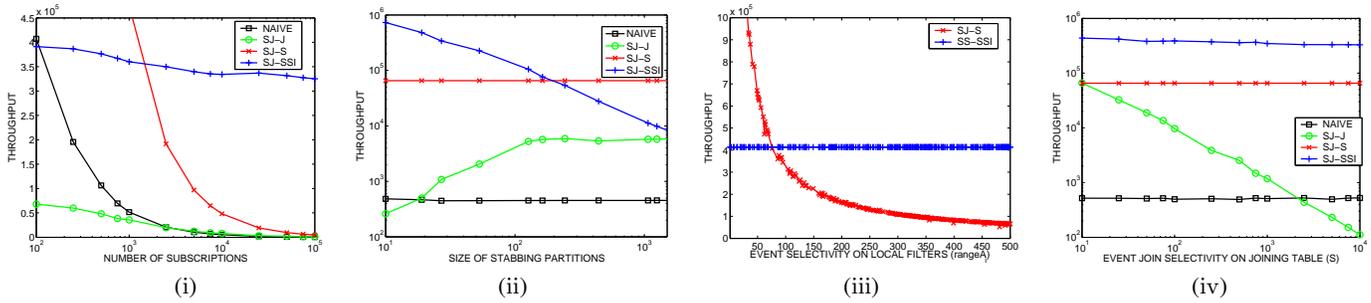


Figure 7: Throughput of equality joins with local selection conditions, (i) over number of subscriptions, (ii) over size of stabbing partitions, (iii) over local filter’s selectivity, and (iv) over join selectivity.

parameter	value(meaning)
size of each base table	100,000
number of initial subscriptions	100,000
join attribute $R.B$	Uni(0, 10000)
local selection attribute $R.A, S.C$	Uni(0, 10000)
join attribute $S.B$	Normal(5000, 1000)
domain of $S.B$	$S.B \in [0, 10000]$
midpoint of $range A_i$	Normal(μ_1, σ_1^2)
length of $range A_i, range C_i$	Normal(μ_2, σ_2^2)
midpoint of $range B_i, range C_i$	Uni(0, 10000)
length of $range B_i$	Normal(μ_3, σ_3^2)

Table 1: Experimental parameters.

Figure 7 (i) compares the throughput of various approaches as the number of subscriptions increases from 10 to 100,000. In this set of experiments, the stabbing number for $\{range C_i\}$ is roughly 30; each incoming R tuple on average joins with 1000 S tuples. In this figure, we see that NAIVE’s performance degrades linearly with the number of subscriptions and therefore is completely unscalable. In our experiments the event selectivity on local filters is 0.1, that is, 10% of all subscriptions can pass through the early selection. Consequently, SJ-S, which first filters out all subscriptions that do not match the incoming event on the local selection condition, performs well only when the number of subscriptions is small. Similarly to NAIVE, it degrades linearly with the number of subscriptions thus is not scalable either. The performance decrease of SJ-J can be attributed to a higher cost in two-dimensional point stabbing queries; in our experiments we used R-trees to support these queries. Although the performance of SJ-J does not drop as drastically as SJ-S and NAIVE, its throughput is less than 5% of SJ-SSI in the case of 100k subscriptions.

Compared with the other approaches, SJ-SSI demonstrates excellent scalability. Its throughput only drops by less than 20% when the number of subscriptions increases from 100 to 100,000. The reason is that SJ-SSI relies on the size of the stabbing partition rather than the number of subscriptions. As long as the size of the partition is stable, which is roughly 30 for all sets of subscriptions of increasing sizes, SJ-SSI performs relatively stable. The slight performance drop is due to the increasing point stabbing query cost within each stabbing group.

Figure 7 (ii) compares the performance of various approaches over the size of the stabbing partition given 100,000 subscriptions. Note the logarithmic scale on throughput. We increase the size of stabbing partitions by increasing mean and variance of interval length. As can be seen, NAIVE and SJ-S are completely independent of the clusteredness of subscriptions, and SJ-SSI drops linearly over an increasing number of stabbing groups. SJ-S out-

performs SJ-SSI when the size of the stabbing partition becomes larger than 250, as the event selectivity of local filters is roughly 250 in our experiments. In the worse case, when all subscriptions are disjoint, SJ-SSI degenerates to NAIVE. It is interesting to see the increasing performance of SJ-J on less clustered subscriptions. This is because when querying an R-tree, we traverse less nodes if the indexed rectangles are less overlapped.

Figure 7 (iii) shows the throughput of SJ-S and SJ-SSI over decreasing event selectivity on local filters (the other two approaches are independent of this parameter). We vary the clusteredness of local filters by fine-tuning the variance of $range A_i$. SJ-S is very sensitive to the local filter selectivity, over which its throughput deteriorates linearly, while SJ-SSI is totally independent of the local filter selectivity. Figure 7 (iv) studies the impact of event join selectivity. Note the logarithmic scale on throughput. Except for SJ-J, all other approaches are immune to event join selectivity. SJ-J drops linearly over the increasing size of intermediate join tuples. SJ-SSI outperforms SJ-S and NAIVE all the time and beats SJ-J even when on average only 10 join tuples are generated as intermediate results.

Band joins. In the second set of experiments, we study the performance of our SSI-based algorithm for band-join queries. We implemented BJ-SSI as well as BJ-D(Outer), BJ-Q(Outer) and BJ-MJ of Section 3.1 and compared their performances. Figure 8 (i) shows the throughput of various approaches over an increasing number of subscriptions from 50 to 500,000. As the number of subscriptions increases, the size of the stabbing partition also increases from about 10 to 60 accordingly. In BJ-D, for each event in the base table, an offset is computed and used to probe the index of all band join windows. Although BJ-D is not sensitive to the number of subscriptions, it is inefficient because typically a large base table will suffocate the throughput. BJ-Q, similarly to NAIVE, completely breaks down on a large number of subscriptions. Its throughput drops below 100 when the number of subscriptions exceeds 1000. The processing time of BJ-MJ is linear both in the size of the base table and in the number of subscriptions. As shown in Figure 8 (i), BJ-MJ enjoys a stable throughput when the number of subscriptions is small, because the cost of traversing the sorted base table dominates the total query time. When the number of subscriptions exceeds 50,000, the throughput of BJ-MJ starts to decrease quickly due to a large number of subscriptions introduced. In sharp contrast, BJ-SSI always outperforms the other approaches by orders of magnitudes, and is very stable over an increasing number of subscriptions. Its performance drops to roughly 1/3 when the number of subscriptions has increased by a factor of 10^4 .

Figure 8 (ii) shows the throughput over an increasing size of stabbing partitions. A total number of 100,000 subscriptions are gen-

erated across different stabbing groups. We have omitted BJ-Q in this figure due to its extremely poor performance on a large number of subscriptions. BJ-MJ and BJ-D are insensitive to the size of the stabbing partition, while the performance of BJ-SSI deteriorates linearly as the partition size increases. Nevertheless, BJ-SSI outperforms the other two approaches even when there are as many as 5000 groups in the partition, which is considered to be a fairly large number in practice.

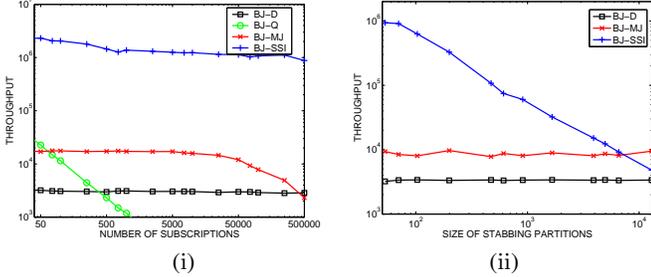


Figure 8: Throughput of band-joins, (i) over the number of subscriptions and (ii) over the size of stabbing partitions.

Dynamic maintenance. In the previous two experiments, we have demonstrated that our SSI-based approaches offer excellent scalability over a large number of subscriptions. In addition, SSI is insensitive to event selectivity on local filters and event join selectivity. We now study the dynamic maintenance cost of SSI-based approaches and other alternatives. For this purpose, besides the initial 100,000 subscriptions, we generated 100,000 subscription updates at run time. The update is either an insertion of a new subscription or a deletion of an existing subscription, each with probability 0.5.

Figure 9 shows the amortized maintenance cost in each of the algorithms BJ-D, BJ-Q, BJ-MJ and BJ-SSI. Since BJ-Q does not maintain any index structure on the subscriptions, its maintenance cost is constantly 0. For BJ-D, the maintenance involves updating the dynamic priority search tree that indexes all band join windows. For BJ-MJ, the maintenance involves updating a sorted list of band join windows. The dynamic maintenance algorithm for BJ-SSI is described in Section 2.3, in which we have chosen $\epsilon = 3$. Consequently, the query time of BJ-SSI is increased by a factor of $1 + \epsilon = 4$ compared to that of BJ-SSI based on an optimal stabbing partition. This approximation factor is acceptable as BJ-SSI outperforms the other approaches by orders of magnitudes in the previous experiments. Note that the reconstruction stage takes place fairly infrequently because all subscriptions are from the same distribution and naturally clustered, and therefore a new subscription is most probably able to be inserted into an existent stabbing group without increasing the size of the stabbing partition. As shown in Figure 9, the amortized maintenance cost of BJ-SSI is only 20% more than that of BJ-MJ, which can be easily paid back by a substantial improvement in scalability.

SSI-based histograms. To demonstrate the advantage of our SSI-based histogram (SSI-HIST) introduced in Section 3.3 over other standard approaches such as the equal-width histogram (EQW-HIST), we compare their performance in terms of the estimation error. We create 100,000 intervals in the integer domain $[0, 10000]$, whose midpoints and lengths are governed by $\text{Normal}(5000, 1500)$ and $\text{Normal}(1000, 2000)$ respectively. As a result there are 18 stabbing groups created from all intervals. Given a fixed total number of buckets, we build SSI-HIST by using the heuristics described

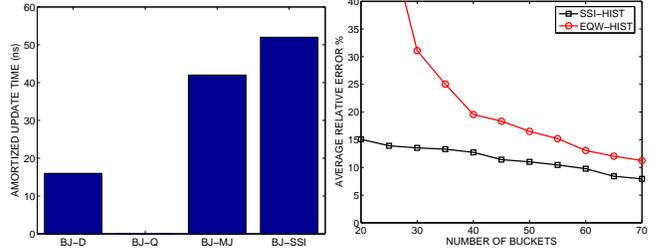


Figure 9: Amortized maintenance cost in various band-join plans.

Figure 10: Histograms.

in Section 3.3 to assign buckets to each stabbing group and using the iterative k-mean algorithm to perform clustering. The construction time of the two methods are comparable and both within half a minute. For each of 5000 uniformly distributed queries, we compute their respective relative errors and then compute the average of these errors. Figure 10 shows their average relative errors when the total number of buckets is increased from 20 to 70. As can be seen, SSI outperforms EQW-HIST all the time. Specifically, given 20 buckets only, SSI-HIST achieves an error rate as small as 15.1%, while that of EQW-HIST is more than 70%. In fact, EQW-HIST requires 50 buckets to reach the same error rate as SSI-HIST with 20 buckets.

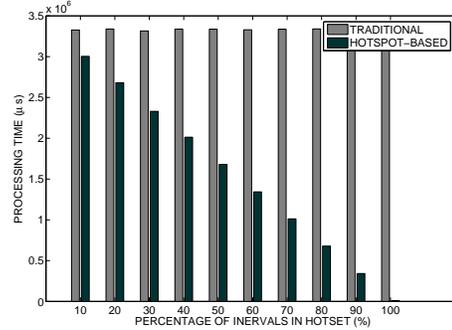


Figure 11: Hotspot-tracking schemes.

Hotspot-tracking schemes. Finally we conduct experiments to compare hotspot tracking schemes in Section 2 with a traditional approach which does not track common interests in subscriptions. For each workload set, we generate 500,000 subscriptions in the form of equality join with local selection conditions 3.2. We group top 500 largest stabbing groups as *hot set* and the coverage percentage of intervals in the hot set indicates the clusterness of user interests, e.g., 100% means subscriptions are highly clustered since top 500 stabbing sets cover all 500,000 intervals, while 10% means the subscriptions are relatively scattered and consequently top 500 stabbing groups can only cover 10% of all intervals. We generate 10 such workload sets with coverage percentage varies from 10% to 100%, as shown in X-axis of Figure 11. As described in Section 2 and 3.2, we use SSI-based approach to process all subscriptions in hot set, and for remaining scattered intervals, we turn to traditional SJ-SelectFirst approach. Figure 11 compares the average processing time of 10,000 incoming events using traditional schemes (SJ-SelectFirst) and our hotspot based approach. Clearly, the traditional scheme is unable to exploit the clusterness in sub-

scriptions, thus its performance is independent of varying coverage percentage of the hot set. On the other hand, the processing time of our hotspot scheme drops linearly over increasing hot set coverage percentage because it adopts SSI processing within hot set to capture clusterness of user interests.

5. RELATED WORK

Scalable continuous query processing plays a pivotal role in large-scale information dissemination systems such as publish/subscribe. By definition, such systems need to handle a huge number of subscriptions (continuous queries) efficiently. Examples include most recent [7], and many more beyond enumeration. Several continuous query and stream processing systems (e.g., [6, 17, 5, 19], etc) have been proposed recently. NiagaraCQ [6] is able to group-process selections and share processing of identical join operations. However, it cannot group process joins with different join conditions (such as band joins). Moreover, NiagaraCQ groups selections and joins separately, resulting in strategies similar to SJ-JoinFirst and SJ-SelectFirst, whose limitations were already discussed in Section 3.2. Our work is able to overcome these limitations. CACQ [17] is a continuous query engine that leverages Eddies [3] to route tuples adaptively to different operators on the fly. It is able to group-process filters, and supports dynamic re-ordering of joins and filters. However, like NiagaraCQ, it still does not support group processing of joins with different join conditions, and processes selections and joins separately. PSoup [5] treats data and queries analogously, thereby making it possible to exploit set-oriented processing on group of joins with arbitrary join conditions. However, PSoup is not specific on what efficient techniques to use for different types of join conditions. Without the join indexing techniques developed in this paper, the alternatives are limited to strategies such as BJ-QOuter, BJ-DOuter, and BJ-MJ (discussed in Section 3.1). Its approach of instantiating partially completed join queries implies time complexity linear in the number of queries. In contrast, our new approach can exploit clustering of queries to achieve sublinear complexity.

6. CONCLUSION AND FUTURE WORK

In this paper we presented a novel and flexible execution framework for scalable processing of continuous joins by tracking the hotspots in user interests. Our work opens the door for many interesting problems. Group processing in more general query processing context, e.g., existence of both band-join windows and local selection conditions, are more complicated yet more tempting because intuitively there exist more opportunities for sharing. We believe our framework makes the first step and server as fundamental building blocks for more complicated sharing processing.

Since group processing demonstrates substantial performance improvement over non-group processing, it is interesting and well motivated to extend clustering by stabbing partitions to multidimensional spaces. We also plan to investigate group processing for aggregation subscriptions. A good starting point is the previous work on group optimization in NiagaraCQ. Our space of alternatives is considerably richer. We will work on developing an optimization framework to support cost-based choices of these alternatives.

7. REFERENCES

- [1] P. K. Agarwal, J. Xie, J. Yang, and H. Yu. Scalable continuous queries processing by tracking hotspots. Technical report, Department of Computer Science, Duke University, March 2006, url: <http://www.cs.duke.edu/~junyi/papers/joincq/vldb06-full.pdf>.
- [2] L. Arge and J. Vitter. Optimal external memory interval management. *SIAM Journal on Computing*, 2003.
- [3] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *2000 SIGMOD*, pages 261–272, Dallas, Texas, USA, May 2000.
- [4] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Monitoring streams - a new class of data management applications. In *2002 VLDB*, pages 215–226, 2002.
- [5] S. Chandrasekaran and M. J. Franklin. Psoup: a system for streaming queries over streaming data. *VLDB J.*, 2003.
- [6] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *2000 SIGMOD*, Dallas, Texas, USA, May 2000.
- [7] J. D., P. M. Fischer, and D. Kossmann. Agile: adaptive indexing for context-aware information filters. In *2005 SIGMOD*, Baltimore, Maryland, USA, June 2005.
- [8] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2nd edition, 2000.
- [9] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An evaluation of non-equi-join algorithms. In *1991 VLDB*, Barcelona, Catalonia, Spain, 1991.
- [10] Q. Du, V. Faber, and M. Gunzburger. Centroidal Voronoi tessellations: Applications and algorithms. *SIAM Reviews*, 41:637–676, 1999.
- [11] E. Hanson and T. Johnson. The interval skip list: A data structure for finding all intervals that overlap a point. In *1991 WADS*, pages 153–164, 1991.
- [12] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park, and A. Vernon. Scalable trigger processing. In *1999 ICDE*, 1999.
- [13] S. Har-Peled and S. Mazumdar. Coresets for k -means and k -median clustering and their applications. In *36th STOC*, pages 291–300, 2004.
- [14] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. Sevcik, and T. Suel. Optimal histograms with quality guarantees. In *1998 VLDB*, pages 275–286, 1998.
- [15] N. Koudas, S. Muthukrishnan, and D. Srivastava. Optimal histograms for hierarchical range queries. In *2000 PODS*, pages 196–204, 2000.
- [16] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *TKDE*, 1999.
- [17] S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *2002 SIGMOD*, Madison, Wisconsin, USA, June 2002.
- [18] J. Pereira, F. Fabret, H. A. Jacobsen, F. Llirbat, and D. Shasha. Webfilter: A high-throughput XML-based publish and subscribe system. In *2001 VLDB*, 2001.
- [19] Special issue on data stream processing. *IEEE Data Engineering Bulletin*, 26(1), March 2003.
- [20] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, 1996.