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. We propose a flexible framework with novel data structures and algorithms for group-processing and indexing continuous queries by exploiting potential overlaps in query predicates. Our approach partitions the collection of continuous queries into groups based on the clustering patterns of the query ranges, and then applies specialized processing strategies to those heavily-clustered groups (or *hotspots*). To maintain the partition dynamically, we present efficient algorithms that maintain a nearly optimal partition in nearly amortized logarithmic time. We show how to use the hotspots to scalably process large numbers of continuous select-join and band-join queries, which are much more challenging than simple range selection queries. Experiments demonstrate that this approach can improve the processing throughput by orders of magnitude. As another application of hotspots, we show how to use them to build a high-quality histogram for intervals in linear time.

1 Introduction

Continuous query processing has attracted much interest in the database community recently because of a wide range of traditional and emerging applications, e.g., trigger and production rule processing [23, 12], data monitoring [4], stream processing [21], and publish/subscribe systems [17, 6, 20, 9]. In contrast to traditional query systems, where each query runs once against a snapshot of the database, continuous query systems support standing queries that continuously generate new results (or changes to results) as new data continues to arrive in a stream. In this paper we propose a novel technique for indexing and processing continuous queries, with the goal of addressing 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 $Q(\mathcal{D}_{i-1})$, if any. 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

*Research by the first and fourth authors is supported by NSF under grants CCR-00-86013, EIA-01-31905, CCR-02-04118, and DEB-04-25465, by ARO grants W911NF-04-1-0278 and DAAD19-03-1-0352, and by a grant from the U.S.–Israel Binational Science Foundation. Research by the second and third authors is supported by NSF CAREER award IIS-0238386 and an IBM Faculty Award.

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 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 [7] 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 conditions or conjunction of selection conditions, and there has been little work on how to scalably index complex continuous queries such as joins, which are not only important in their own right but also essential in building more complex queries.

Opportunity for optimization. We propose a novel technique for indexing and processing continuous queries applicable to joins. The main idea 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 often arise in the continuous query setting. Following this observation, suppose that we cluster the set of continuous queries based on the similarity of their query ranges. Then, like in the above stock trader example, we may be able to identify a number of large clusters (or *hotspots*) containing the majority of all continuous queries. Let us call the queries in these 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 can be indexed in special ways that support much faster processing. For scattered queries, on the other hand, we may use a traditional processing method that is less efficient. The hope is that scatter queries will be the minority, so overall we gain a significant speedup in processing all continuous queries.

Note that our approach naturally leads to faster processing for more clustered query ranges. In the unlikely worst case, i.e., when most query ranges are scattered, it gracefully degrades into a traditional processing method, which is the best we can do because there is no opportunity for clustered processing.

Contributions. To materialize the idea above, we need to 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 of queries inside a hotspot can be exploited to index and process them 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 large clusters from the partition, and present efficient algorithms to maintain 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 within each hotspot can be exploited for more efficient processing. We give three representative examples:
  (1) indexing continuous band joins [8] whose join conditions check whether the difference between two join attribute values falls within some range;
  (2) indexing continuous equality joins with different local range selections; and
  (3) building a high-quality histogram for a set of intervals in linear time for selectivity estimation.
- In Section 4, we demonstrate through 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.

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, the hotspots may also evolve over time. 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 $I_1, I_2, \ldots, I_\tau$ 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 $P = \{p_1, \cdots, p_\tau\}$ is called a *stabbing set* of I.

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,

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

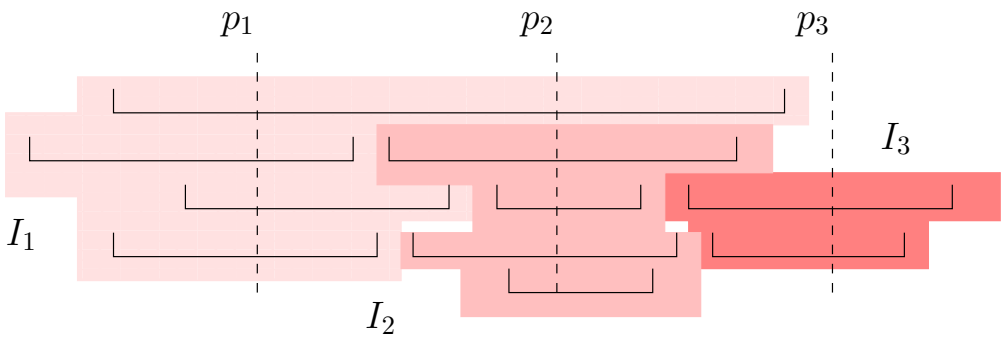


Figure 1: A stabbing partition of 10 intervals. I_1 and I_2 are 0.4-hotspots.

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 instantiated for specific uses in Section 3. Given 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 $\mathcal{I}$ of I. SSI stores the stabbing points $p_1, \ldots, p_\tau$ in sorted order in a search tree. Furthermore, for each group $I_j \in \mathcal{I}$, 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" work among them.

Note that, as mentioned in the introduction, we actually only apply SSI to the subset of large clusters (i.e., hotspots) in the stabbing partition instead of the entire set of clusters. The reason is that scattered queries do not benefit from the specialized techniques designed to exploit clustering; in fact, they incur extra overhead. Therefore, we process scattered queries using traditional algorithms.

2.2 Tracking Hotspots

Clusters in the SSI may be unbalanced, as illustrated by the following simple example. Suppose that user interests follow a Zipfian distribution, 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

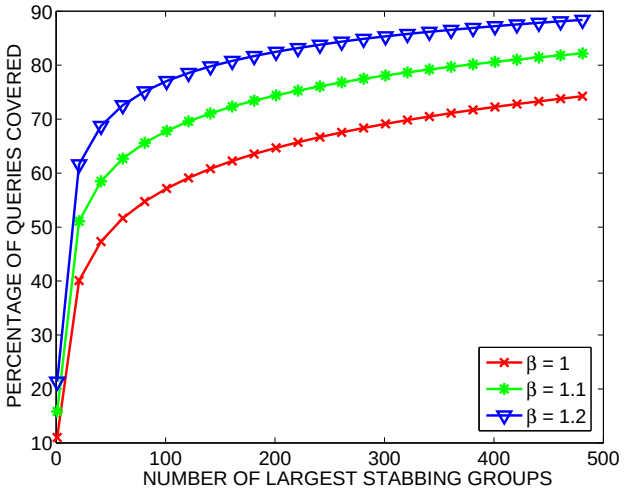


Figure 2: Hotspot coverage in Zipf distribution.

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 the 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 $\mathcal{I} = \{I_1, \cdots, I_\tau\}$ is a stabbing partition of I. A group $I_i \in \mathcal{I}$ is called an α-*hotspot* if $|I_i| \geq \alpha|I|$. An interval of I is called a *hotspot interval* (with respect to $\mathcal{I}$) 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 contains at least α fraction of all continuous queries. 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 $\mathcal{I}$ 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 $\mathcal{I}$ of I. In order to extract meaningful hotspots from I, it is important to require that the size of $\mathcal{I}$ 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 $\mathcal{I}$ of I and a partition of $\mathcal{I}$ into two sets $\mathcal{I}_H$ and $\mathcal{I}_S = \mathcal{I} \setminus \mathcal{I}_H$ that satisfy the following three invariants all the time:

- **(I1)** $\mathcal{I}_H$ contains all α-hotspots of $\mathcal{I}$, and possibly a few $(\alpha/2)$-hotspots, but nothing more. Hence, $|\mathcal{I}_H| \leq 2/\alpha$;
- **(I2)** The size of $\mathcal{I}$ is at most $(1+\varepsilon)\tau(I) + 2/\alpha$;
- **(I3)** Let S denote the set of intervals in the groups of $\mathcal{I}_S$. 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. Katz et al. [15] first proved this result by presenting an algorithm with the claimed performance bound. In Section 2.3 we will describe a slightly better algorithm that is more suitable for real-time applications, as well as simple and practical variants of the algorithm.

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 $\mathcal{I}$ of I by maintaining a partition of $\mathcal{I}$ into two sets $\mathcal{I}_H$ and $\mathcal{I}_S = \mathcal{I} \setminus \mathcal{I}_H$. We use S to denote the set of intervals falling into the groups of $\mathcal{I}_S$, and $H = I \setminus S$ to denote the set of intervals falling into the groups of $\mathcal{I}_H$. Hence, $\mathcal{I}_S$ is a stabbing partition of S, and $\mathcal{I}_H$ is a stabbing partition of H. Initially when $I = \emptyset$, we have $\mathcal{I} = \emptyset$, $\mathcal{I}_H = \mathcal{I}_S = \emptyset$, and $S = H = \emptyset$. A schematic view of the algorithm is depicted in Figure 3.

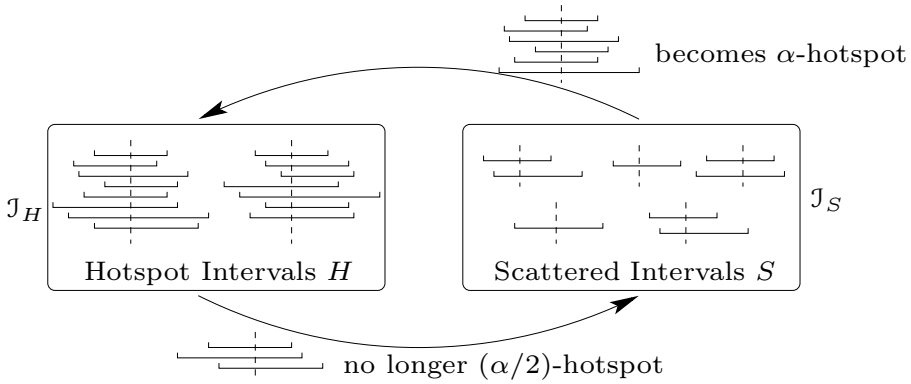


Figure 3: Schematic view of the hotspot-tracking algorithm.

Insertion. When an interval γ is inserted into I, we first check if γ can be added to any group $I_i \in \mathcal{I}_H$, such that the common intersection of the intervals in that group remains nonempty after adding γ. This can be done brute-forcely in $O(1/\alpha)$ time by maintaining the common intersection of each group in $\mathcal{I}_H$, or in $O(\log(1/\alpha))$ time by using a more complicated data structure (e.g., a dynamic priority search tree [19]); we omit the details.

If there indeed exists such a group $I_i \in \mathcal{I}_H$, 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., $\mathcal{I}_S$. As a consequence, the sizes of some groups in $\mathcal{I}_S$ may become $\geq \alpha|I|$. We "promote" all such

groups of $\mathcal{I}_S$ into $\mathcal{I}_H$ (because they become α-hotspots). Consequently, intervals in these groups should be moved out of S. We maintain the stabbing partition $\mathcal{I}_S$ of S by deleting these intervals from S one by one and using Lemma 2 to update $\mathcal{I}_S$. (But in practice, it might be unnecessary to use Lemma 2 to update $\mathcal{I}_S$, 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 $\mathcal{I}_H$ may become $< (\alpha/2)|I|$. We "demote" all such groups of $\mathcal{I}_H$ into $\mathcal{I}_S$ (because they are no longer $(\alpha/2)$-hotspots). Consequently, intervals in these groups are moved into S. We again use Lemma 2 to update $\mathcal{I}_S$ by inserting these intervals into S one by one. Note that when these insertions are finished, some groups in $\mathcal{I}_S$ might again become new α-hotspots, in which case we "promote" these groups into $\mathcal{I}_H$ 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 $\mathcal{I}_H$. This can be done in constant time by maintaining appropriate pointers from intervals to groups.

If there indeed exists such a group $I_i \in \mathcal{I}_H$, we remove γ from this group. The removal might make I_i no longer an $(\alpha/2)$-hotspot (note, however, the other groups in $\mathcal{I}_H$ 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 $\mathcal{I}_S$ by inserting the intervals of I_i into S one by one and updating $\mathcal{I}_S$ using Lemma 2. Otherwise, we know that $\gamma \in S$. We remove γ from S and update $\mathcal{I}_S$ accordingly using Lemma 2.

After that, some groups in $\mathcal{I}_S$ could become α-hotspots. We "promote" these groups into $\mathcal{I}_H$ 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* $O(\varepsilon^{-1} \log |I|)$.

Proof:

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

(I2) Since we used Lemma 2 to maintain $\mathcal{I}_S$, we have $|\mathcal{I}_S| \leq (1+\varepsilon)\tau(S) \leq (1+\varepsilon)\tau(I)$. By (I1), we also have $|\mathcal{I}_H| \leq 2/\alpha$. Hence,

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

(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 $\mathcal{I}_H$, 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 $\mathcal{I}_H$ is demoted to $\mathcal{I}_S$, 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 $I_i \in \mathcal{I}_H$, 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, 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 always deposit 2α credits to each group in $\mathfrak{I}_H$. Furthermore, if γ does not fall into any group of $\mathfrak{I}_H$ (recall that in this case our algorithm inserts γ into S), we deposit another one credit to γ. Since $|\mathfrak{I}_H| \leq 2/\alpha$ by (I1), 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 $\mathfrak{I}_H$, 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 $I_i \in \mathfrak{I}_H$ be a group to be demoted. We know that I_i was promoted to $\mathfrak{I}_H$ 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 $\mathfrak{I}_S$ 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 $\mathfrak{I}_S$ of S is $O(\varepsilon^{-1} \log |I|)$. □

2.3 Dynamic Stabbing Partitions

This section is devoted to an efficient implementation of Lemma 2. Because it is not a prerequisite for the subsequent discussions of this 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 small constant number of insertions or deletions. (A simple example is omitted for brevity.) Thus, we 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 poorly 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 $\mathcal{I}$ 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 $\mathcal{I}$. When an interval γ is deleted from I, suppose that γ belongs to some group $I_i \in \mathcal{I}$. We then remove γ from I_i, and if I_i becomes empty after the removal of γ, we also remove I_i from $\mathcal{I}$ and the stabbing point of I_i from P. After $\varepsilon\tau_0/(\varepsilon + 2)$ number of insertions and deletions, 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: Please see Appendix A for details. □

The above 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 $\overline{I}$ denote the set of intervals after the last reconstruction and $\tau_0 = \tau(\overline{I})$. Suppose that m intervals have been deleted from $\overline{I}$ 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. 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 Appendix B, 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 the above simple strategy. Moreover, each insertion or deletion affects 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. Please refer Appendix B for detailed algorithms and pseudocode.

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: scalable processing of continuous band joins, continuous equality joins with local selections, and building histograms for selectivity estimation. 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.

In particular, we consider the following two types of continuous queries over relations $R(A,B)$ and $S(B,C)$:

- **Equality join with local selections**: $\sigma_{A\in rangeA}R \bowtie_{R.B=S.B} \sigma_{C\in rangeC}S$
- **Band join**: $R \bowtie_{S.B-R.B\in rangeB} S$

In equality join with local selections, the query parameters $rangeA$ and $rangeC$ in the local selection conditions are ranges over numeric domains of $R.A$ and $S.C$, respectively. In band join, $rangeB$ in the join condition is a range over the numeric domain of $R.B$ and $S.B$. These two types of queries are important in their own right, and also essential as building blocks of more complex queries. We give two examples of these queries below.

Example 1. Consider a listing database for merchants with the following two relations:

$$\texttt{Supply}(\texttt{suppId},\texttt{prodId},\texttt{quantity},\ldots),$$

and

$$\texttt{Demand}(\texttt{custId},\texttt{prodId},\texttt{quantity},\ldots)$$

. 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

$$\sigma_{\texttt{quantity}\in rangeS_i}\texttt{Supply} \bowtie \sigma_{\texttt{quantity}\in rangeD_i}\texttt{Demand},$$

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 $\texttt{Unit}(\texttt{id},\texttt{model},\texttt{pos},\ldots)$ and $\texttt{Target}(\texttt{id},\texttt{type},\texttt{pos},\ldots)$, 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.,

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

where `BB` is a fictitious model of gun batteries, $range$ is the firing range of this model, and the selection condition on `Target` 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

$$R \bowtie_{S.B-R.B \in rangeB_i} S.$$

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 $rangeB_i$ 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 $\{S.B \in rangeB_i + r.B\}$ local to S. Then, this set of selections can be treated as a relation of intervals $\{rangeB_i + r.B\}$ and joined with S; each S-tuple s such that $s.B$ stabs the interval $rangeB_i + r.B$ 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 $rangeB_i + r.B$ 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 $\{rangeB_i\}$ (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 $\{rangeB_i + r.B\}$ with S. This strategies requires that we maintain the intervals $\{rangeB_i\}$ 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 difficulty.

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 $\{rangeB_i\}$. 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 $S(B)$.

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 $rangeB_i$ that are stabbed by some point $s.B - b$ where $s \in S$.

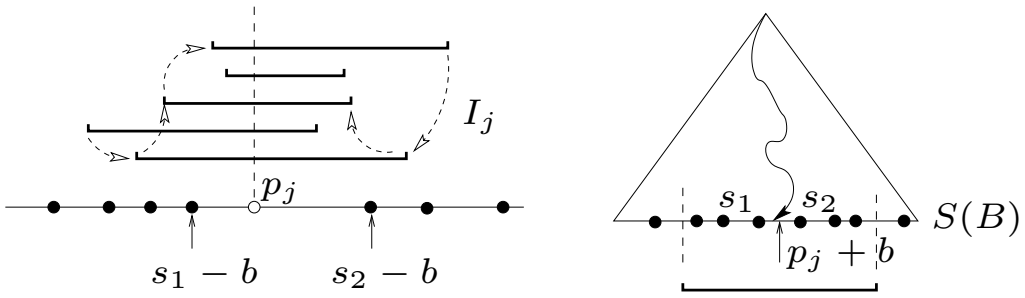


Figure 4: The SSI algorithm for band join processing. Arrows indicate the order in which the intervals are visited.

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 the 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.

To see that the above procedure correctly returns the set of all affected continuous band joins in I_j, 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 $s.B \in (s_1, s_2)$.

(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 $\mathcal{I}_H$ of Theorem 1 (i.e., the collection of hotspots), we immediately obtain an efficient algorithm for processing the subset of hotspot queries. Note that $|\mathcal{I}_H| \leq 2/\alpha$, 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

$$\sigma_{A \in rangeA_i} R \bowtie_{R.B=S.B} \sigma_{C \in rangeC_i} S.$$

Each such query can be represented by a rectangle spanned by two ranges $rangeC_i$ and $rangeA_i$ 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 $\{rangeC_i \times rangeA_i\}$ 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 $\{rangeA_i\}$ (e.g., a priority search tree [7] or external interval tree [2]). To compute the new result tuples for each identified query with query range $rangeC_i$ 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 $S.B = r.B \wedge S.C \in rangeC_i$.

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 circumvents 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.

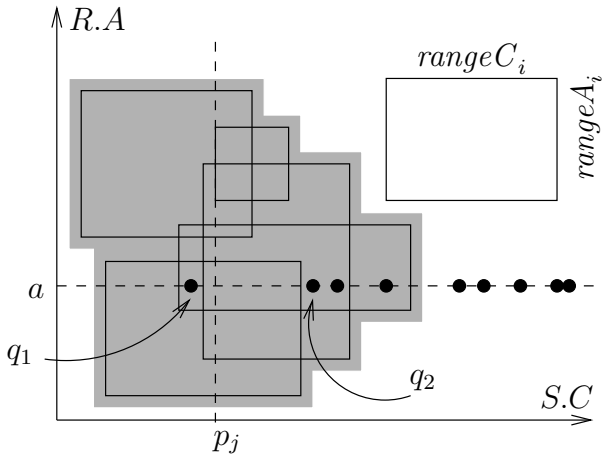


Figure 5: The SSI algorithm for processing equality joins with local selections.

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.

We claim that the query rectangles returned by the R-tree lookup constitute precisely the set of continuous queries in I_j that are affected by r. To see this, recall that by our construction, 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.

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 $\mathcal{I}_H$ of Theorem 1 (i.e., the collection of hotspots), we immediately obtain an efficient algorithm for processing the subset of hotspot queries. Since $|\mathcal{I}_H| \leq 2/\alpha$, 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 Intervals 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 relative error between $h(x)$ and $f_I(x)$ is

$$E^2(h, f_I) = \int \frac{|h(x) - f_I(x)|^2}{|f_I(x)|^2} \phi(x)\, \mathrm{d}x.$$

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, 16]. In contrast, our new algorithm below is simple, runs in nearly linear time, and often provides a high-quality histogram. 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 does not. 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 observation: Computing an optimal histogram for each group of a stabbing partition of I can be reduced to a simple geometric 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 $\mathcal{I} = \{I_1, \cdots, I_\tau\}$ for I as in Lemma 1, and then build a histogram for each group of $\mathcal{I}$. The final histogram is obtained by summing up these histograms. Let p_i be the stabbing point of an α-hotspot $I_i \in \mathcal{I}$, and let $f^l_{I_i}$ (resp. $f^r_{I_i}$) 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^l_i and h^r_i to approximate $f^l_{I_i}$ and $f^r_{I_i}$ respectively, and then let $h_i(x) = h^l_i(x) + h^r_i(x)$.

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

Lemma 4 *There is an optimal histogram with at most k buckets such that each bucket boundary passes through one of the break points $x_1, \cdots, x_m$.*

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. □

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

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

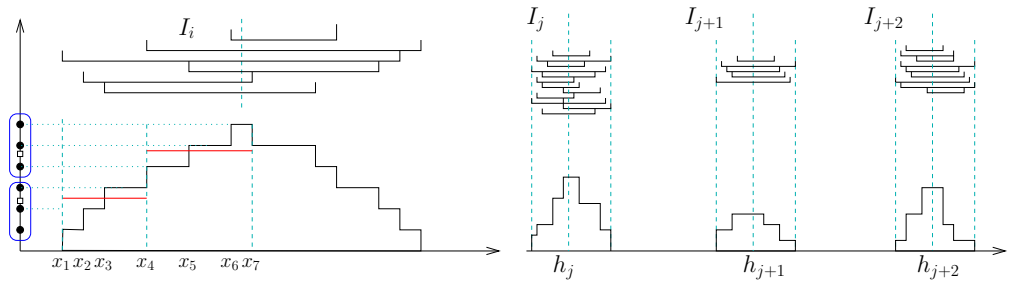


Figure 6: Reducing to a one-dimensional weighted k-means clustering problem.

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

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

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

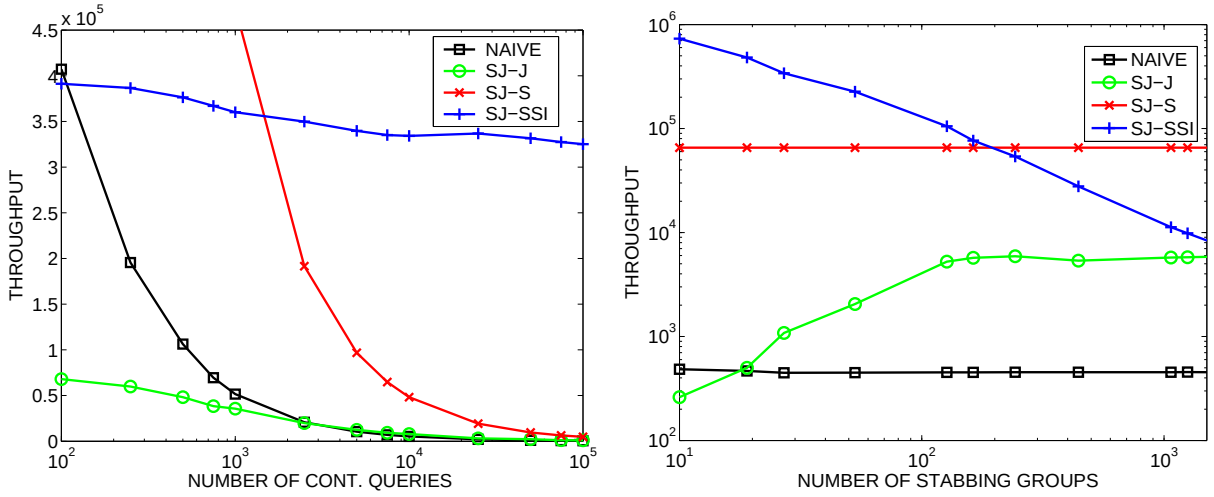
To find a histogram $h_i^l(x)$ that minimizes (1), we solve the following weighted k-means clustering problem in one dimension: Given a set of m points $y_1 = f_{I_i}^l(x_1), \cdots, y_m = f_{I_i}^l(x_m)$, and a weight $w_\ell = \int_{x_\ell}^{x_{\ell+1}} \phi(x)\, \mathrm{d}x / |y_\ell|^2$ for each point y_ℓ, find k centers $c_1, \cdots, c_k$ and an assignment of each y_ℓ to one of the centers so that the weighted k-means clustering cost is minimized (see the left part of Figure 6). We have the following lemma to establish the correctness of our algorithm.

Lemma 5 *Minimizing (1) is equivalent to solving the above weighted k-means clustering problem.*

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 group 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 of [13] on the whole point set to compute an ε-approximate optimal histogram in nearly linear time $O(n) + \mathrm{poly}(k, 1/\varepsilon, \log n)$, which automatically assigns an appropriate number of buckets to each I_i. In practice, one may wish to use the simpler iterative k-means clustering algorithm [10] instead. Since the iterative k-means algorithm is sensitive to the initial assignment of clusters, we can heuristically assign each group a number of buckets proportional to the cardinality of the group. We then run the iterative k-means algorithm on each group separately.

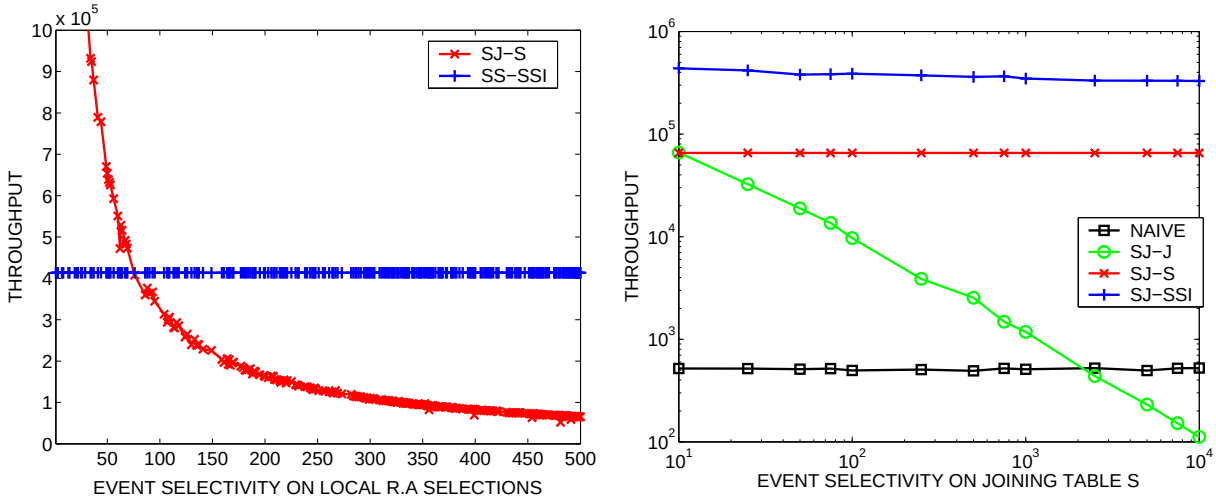
4 Experiments

To compare our techniques against traditional processing techniques in terms of their scalability with a large number of continuous queries, we have implemented various algorithms discussed in previous sections in Java SDK 1.5.0. Unless otherwise noted, all experiments were conducted on a Sun Blade 150 with a 650MHz UltraSPARC-III processor and 512 MB of memory. We measured the processing throughput, i.e., the number of data update events that each approach is able to process per second. We excluded the output time from measurement since it is application-dependent and common to all approaches. We also measured the cost of maintaining associated data structures in all approaches.



(i) (ii)

Figure 7: Throughput of equality joins with local selections (i) over number of continuous queries, (ii) over size of stabbing partition.



(iii) (iv)

Figure 8: Throughput of equality joins with local selections (iii) over event selectivity with respect to local selections on $R.A$, and (iv) over event selectivity with respect to the joining table S.

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 respective domains. 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 continuous queries, each with $100,000$ queries initially. The first set consists of equality joins with local selections discussed in Section 3.2, and the second set consists of band joins discussed in Section 3.1. The midpoints of $rangeA_i$ follow a normal distribution, and the midpoints of

$rangeB_i$ and $rangeC_i$ are uniformly distributed. The lengths of all ranges are normally distributed. At runtime, users may insert new continuous queries, and delete or update existing ones. Table 1 summarizes the data and workload parameters, where μ_i's and σ_i's are used to adjust various input characteristics that affect performance, such as selectivities of incoming events against continuous queries as well as the degree of overlap among continuous queries.

Parameter	Value
Size of each base table	$100,000$
Initial number of continuous queries	$100,000$
Join attribute $R.B$	$\mathrm{Uni}(0, 10000)$
Local selection attribute $R.A$, $S.C$	$\mathrm{Uni}(0, 10000)$
Join attribute $S.B$	$\mathrm{Normal}(5000, 1000)$
Domain of $S.B$	$[0, 10000]$
Midpoint of $rangeA_i$	$\mathrm{Normal}(\mu_1, \sigma_1^2)$
Length of $rangeA_i$,$rangeC_i$	$\mathrm{Normal}(\mu_2, \sigma_2^2)$
Midpoint of $rangeB_i$, $rangeC_i$	$\mathrm{Uni}(0, 10000)$
Length of $rangeB_i$	$\mathrm{Normal}(\mu_3, \sigma_3^2)$

Table 1: Experimental parameters.

Equality joins with local selections. In addition to the algorithms SJ-SSI, SJ-J(oinFirst), and SJ-S(electFirst) discussed in Section 3.2, we have also implemented an algorithm called NAIVE, which first joins the new R tuple with S to generate a list of intermediate result tuples ordered by $S.C$, and then evaluates the local selections of each continuous query 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.

To focus our attention on the effectiveness of SJ-SSI itself, we first present a series of results obtained by applying SJ-SSI to *all* stabbing groups (regardless of whether they are hotspots). We then present results for combining SSI and hotspots afterwards.

Figure 7(i) compares the throughput of various approaches as the number of continuous queries increases from 10 to $100,000$. In this set of experiments, the stabbing number for $\{rangeC_i\}$ 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 continuous queries and therefore is completely unscalable. The average selectivity of an event on the local selection ranges $\{rangeA_i\}$ is 0.1; that is, an incoming event satisfies the $R.A$ selection for 10% of all continuous queries. Consequently, SJ-S, which works by iterating through queries whose $R.A$ selection is satisfied, performs well only when the number of queries is small. Similar to NAIVE, it degrades linearly with the number of queries and thus is not scalable either. The performance decrease of SJ-J can be attributed to 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 queries.

Compared with the other approaches, SJ-SSI demonstrates excellent scalability. Its throughput only drops by less than 20% when the number of queries increases from 100 to $100,000$. The reason is that SJ-SSI depends primarily on the number of stabbing groups rather than the number of queries. As long as the number of groups is stable (roughly 30 for these experiments), SJ-SSI's performance is relatively stable. The slight performance drop comes from the increasing cost of the point stabbing query within each stabbing group, because each group on average contains more queries.

Figure 7(ii) compares the performance of various approaches over a range of clusteredness among $rangeC_i$'s. The number of continuous queries stays at $100,000$, but we increase the number of stabbing groups by decreasing mean and variance of interval lengths. As can be seen, NAIVE and SJ-S are completely indifferent about the clusteredness of queries, while SJ-SSI benefits from smaller numbers of stabbing groups. SJ-S outperforms SJ-SSI when there are more than 250 stabbing groups, as the event selectivity on $R.A$ selections is roughly 250 in these experiments. In the worse case, when all query ranges are disjoint, SJ-SSI degenerates to NAIVE. As a side node, it is interesting that SJ-J performs better on less clustered queries. The reason is that the cost of querying an R-tree tends to be lower if the indexed objects overlap less.

Figure 8(iii) shows the throughput of SJ-S and SJ-SSI when we decrease the average event selectivity on local $R.A$ selections (SJ-J and NAIVE are unaffected by this parameter). We control this selectivity by fine-tuning the distribution of $rangeA_i$'s. From this figure, we see that SJ-S is very sensitive to this selectivity, over which its throughput deteriorates linearly (since this selectivity directly controls how large n' is in Theorem 4). On the other hand, SJ-SSI is unaffected by this selectivity.

Figure 8(iv) studies the impact of event selectivity on joining table S, i.e., how many S tuples join with the incoming event, controlled by fine-tuning the distribution of $S.B$. Except for SJ-J, all other approaches are immune to increase in this selectivity. SJ-J's performance degrades linearly as the number of intermediate join result tuples increases.

SSI + hotspot-tracking. In all previous experiments we applied SJ-SSI to every group in the stabbing partition, ignoring the hotspot optimization. Now, we conduct experiments to demonstrate the effect of hotspot-tracking in group processing equality joins with local selections. For each experiment in this set, we generate a workload of $500,000$ queries, with varying degrees of clusteredness across these workloads. We choose a fairly small α value (on the order of 0.1%) for each workload, such that no more than 500 groups are chosen as α-hotspots. In Figure 9, the horizontal axis shows the ten workloads in increasing degree of clusteredness, labeled by the percentage of intervals in hotspots. 100% means that queries are highly clustered since top 500 stabbing groups cover all $500,000$ intervals, while 10% means the queries are relatively scattered and consequently top 500 stabbing groups can only cover 10% of all intervals.

The vertical axis of Figure 9 plots the average processing time per event, measured over $10,000$ events for each experiment. We compare two approaches here: TRADITIONAL simply use SJ-S(electFirst); HOTSPOT-BASED uses SJ-SSI on the hotspots, and SJ-S on the remaining, scattered intervals. The traditional approach, unable to exploit the clusteredness, behaves identically across workloads. On the other hand, the performance of the hotspot-based approach improves linearly with the increasing coverage by hotspots, as it benefits from the ability of SSI in exploiting clusteredness for efficient group processing. Moreover, this hotspot-based approach offers an additional advantage over the "purist" approach of applying SJ-SSI to every stabbing group. By restricting SJ-SSI to hotspots, the hotspot-based approach is able to focus on large stabbing groups where SJ-SSI really shines, while avoiding the overhead of going over a large number of small groups for which SJ-SSI may be outperformed by more traditional approaches.

Band joins. In this set of experiments, we study the performance of our SSI-based algorithm for band-join queries. We compare BJ-SSI with BJ-D(Outer), BJ-Q(Outer), and BJ-MJ, discussed in Section 3.1. Again, to focus our attention on the effectiveness of BJ-SSI itself, we show results of applying BJ-SSI to all stabbing groups without the hotspot optimization.

Figure 10(i) shows the throughput of various approaches over an increasing number of continuous queries from 50 to $500,000$. As the number of queries increases, the number of stabbing groups also

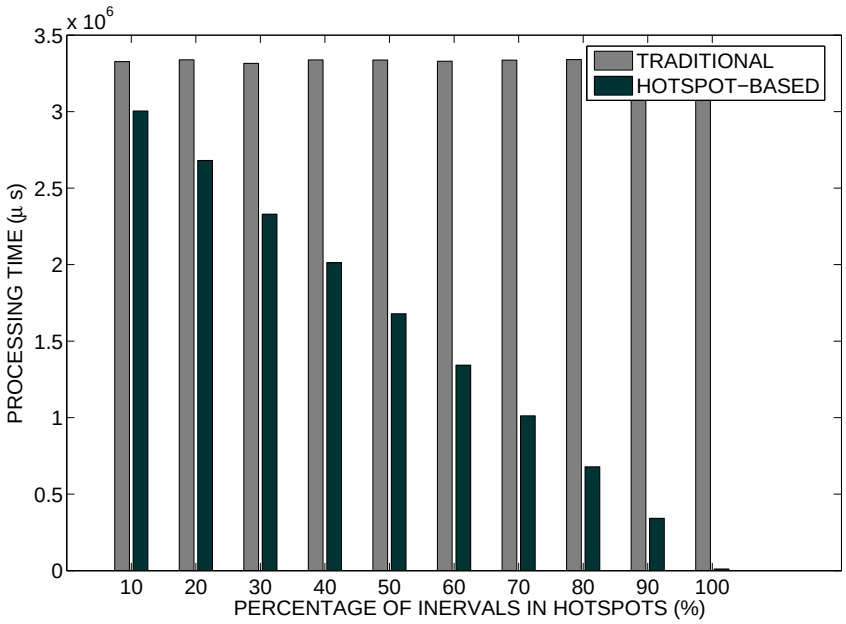


Figure 9: SSI with hotspot-tracking.

increases from about 10 to 60 accordingly. In BJ-D, for each tuple in base table S, an offset is added and used to probe the index of all band join windows. Although BJ-D is not very sensitive to the number of queries, it is inefficient because a large base table will easily destroy the throughput. BJ-Q, similar to NAIVE, completely breaks down on a large number of queries. Its throughput drops below 100 when there are more than 1000 queries. The processing time of BJ-MJ is linear both in the size of the base table and in the number of queries. As shown in the figure, BJ-MJ enjoys a stable throughput when the number of queries is small, because the cost of traversing the sorted base table dominates the total query time. However, once the number of queries reaches $50,000$, the throughput of BJ-MJ starts to decrease quickly. In sharp contrast, BJ-SSI always outperforms the other approaches by orders of magnitudes, and is very stable over an increasing number of queries. Its performance drops to roughly $1/3$ when the number of queries has increased by a factor of 10^4.

Figure 10(ii) shows the throughput over an increasing number of stabbing groups, while the total number of continuous queries is kept constant at $100,000$. We have omitted BJ-Q in this figure due to its extremely poor performance on a large number of queries. BJ-MJ and BJ-D are insensitive to the number of the stabbing groups, while the performance of BJ-SSI deteriorates linearly as this number increases. Nevertheless, BJ-SSI outperforms the other two approaches even when there are as many as 5000 groups in the partition, which is a fairly large number in practice.

Dynamic maintenance. In the previous experiments, we have demonstrated that our SSI-based approaches offer excellent scalability over a large number of continuous queries. We now compare the dynamic maintenance cost of SSI-based approaches with other alternatives. For this purpose, starting from the initial set of $100,000$ queries, we generate $100,000$ updates to this set at run time. The update is either an insertion of a new query or a deletion of an existing query, each with probability 0.5.

Figure 11 shows the amortized maintenance cost for each of the algorithms BJ-D, BJ-Q, BJ-MJ, and

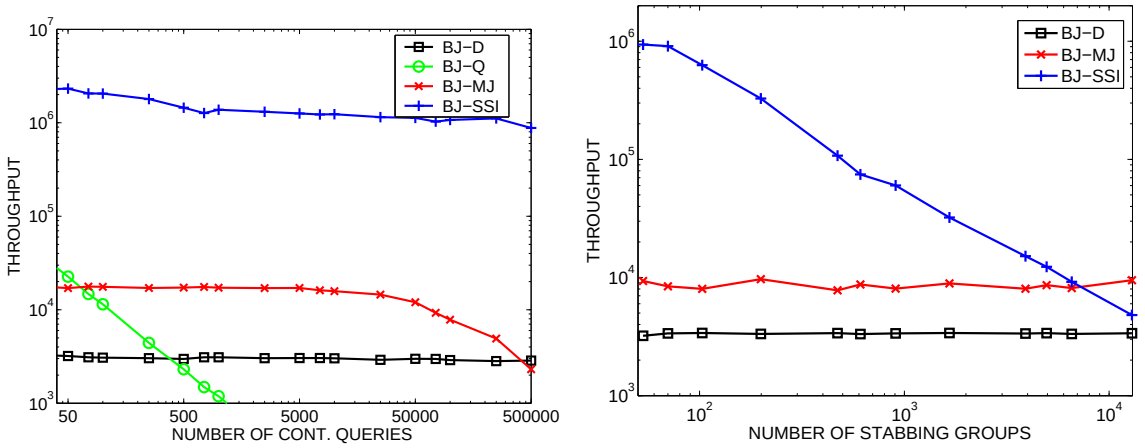


Figure 10: Throughput of band-joins (i) over the number of continuous queries and (ii) over the number of stabbing groups.

BJ-SSI. Since BJ-Q does not maintain any index structure on the queries, 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, for which have chosen $\varepsilon = 3$. Consequently, the query time of BJ-SSI is increased by a factor of $1 + \varepsilon = 4$ 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 occurs fairly infrequently because all subscriptions are from the same distribution and naturally clustered, and therefore with high probability a new subscription will be inserted into an existent stabbing group without increasing the number of of the stabbing groups. As shown in Figure 11, the amortized maintenance cost of BJ-SSI is only 20% more than that of BJ-MJ, which is well justified by a substantial improvement in event processing scalability.

SSI-based histogram construction. We compare SSI-HIST, the histogram constructed by our SSI-based algorithm in Section 3.3, with EQW-HIST, the standard equal-width histogram, and with OPTIMAL, the optimal histogram constructed using dynamic programming. We create $100,000$ intervals in the range $[0, 10000]$. Their midpoints and lengths are governed by $\mathrm{Normal}(5000, 1500)$ and $\mathrm{Normal}(1000, 2000)$, and they happen to form 18 stabbing groups. Given a fixed number of buckets, we build SSI-HIST using the k-mean algorithm and the heuristics to assign the number of buckets to each stabbing group based on its cardinality, as described in Section 3.3. Construction of SSI-HIST completes within one minute. However, construction of OPTIMAL using dynamic programming for $100,000$ intervals proved to be unacceptably slow on our computing platform. Instead, we built OPTIMAL on just a sample of $10,000$ intervals and ran experiments multiple times until a stable estimation is reached. Even with one-tenth of the original data, OPTIMAL took roughly 6.5 hours on a computer with 3GHz processor and 2GB memory, in sharp contrast to the ease of constructing SSI-HIST.

Figure 12 compares the performance of SSI-HIST, EQW-HIST, and OPTIMAL, as we increase the size of the histogram from 20 to 70 buckets. Each data point is obtained by running 5000 uniformly distributed

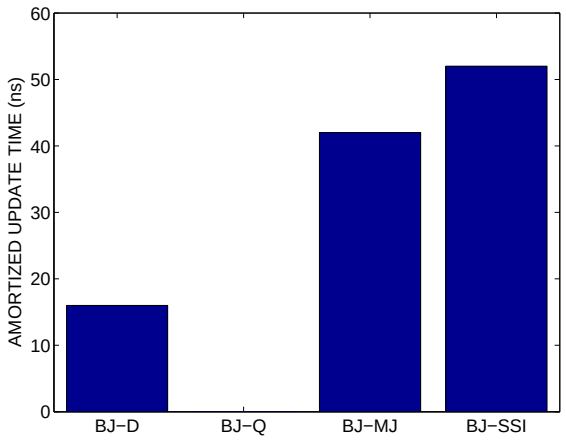


Figure 11: Maintenance cost for band-join algorithms.

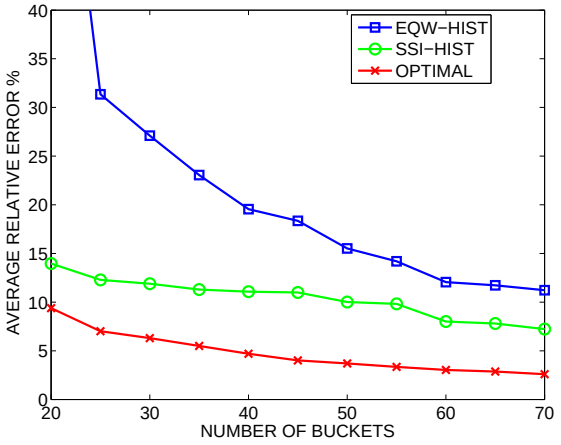


Figure 12: Quality of various histograms for intervals.

stabbing queries; we compute the relative error between true and estimated result sizes, and then report the average of these errors. As expected, OPTIMAL consistently wins; however, this advantage is greatly offset by its impracticality in terms of construction cost. On the other hand, SSI outperforms EQW-HIST all the time and dramatically reduces the gap between EQW-HIST and OPTIMAL. Specifically, given only 20 buckets, SSI-HIST achieves an error rate as small as 14.9%, while that of EQW-HIST is more than 70%. In fact, EQW-HIST would require 50 buckets to reach the same error rate as that of SSI-HIST with 20 buckets.

5 Related Work

As mentioned in Section 1, scalable continuous query processing plays a pivotal role in many applications (e.g., [23, 12, 4, 21]). For example, publish/subscribe systems [17, 6, 20, 9] by definition need to handle a huge number of subscriptions (continuous queries) efficiently. Our earlier work [1] considered the problem of indexing continuous band-join queries, and presented an indexing structure with subquadratic space and sublinear query time. However, the structure is mainly of theoretical interest. On the practical side, several continuous query and stream processing systems (e.g., [6, 18, 5, 21]) 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 [18] 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 reordering 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. 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 technique for handling a large number of continuous queries by tracking the hotspots in user interests. Our technique has a number of applications including scalable continuous join processing and histogram construction for interval data. Our work opens the door for many interesting problems. First, it would be interesting to extend the idea of clustering by stabbing partition to multidimensional spaces, so that we can handle multi-attribute selection conditions. More generally, we plan to investigate group processing for more complex queries, e.g., those combining both band-join and local selection conditions, as well as possible aggregation. Although we have taken the first step with this paper, it remains a challenging problem to develop methods for composing group-processing techniques for more complex queries. Finally, we are developing a general cost-based optimization framework for identifying the best processing strategy. A good starting point is the previous work on group optimization in NiagaraCQ [6]. However, our space of alternatives is considerably richer. In addition, we are making our system adaptive at much finer granularity—every incoming data update event can potentially be processed using a different strategy.

APPENDICES

A Proof of Lemma 3

Proof: Suppose this procedure maintains a stabbing partition $\mathcal{I}$ with size P. Then $P \leq (1 + \frac{\varepsilon}{\varepsilon+2})\tau_0$ at all times, because each insertion increases the size of $\mathcal{I}$ by at most one, and each deletion does not increase the size of $\mathcal{I}$. 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 $\tau(I) \geq (1 - \frac{\varepsilon}{\varepsilon+2})\tau_0$ at any time before the reconstruction stage. Hence,

$$P \leq (1 + \frac{\varepsilon}{\varepsilon + 2})\tau_0 = (1 + \varepsilon)(1 - \frac{\varepsilon}{\varepsilon + 2})\tau_0 \leq (1 + \varepsilon)\tau(I)$$

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

B A Refined Algorithm

The amortized cost per insertion and deletion in the simple strategy (Section 2.3) is $O(n \log n/(\varepsilon\tau_0))$. Katz et al. [15] presented a faster algorithm for maintaining the stabbing partition with $O((1/\varepsilon) \log n)$ worst-case amortized cost per insertion and deletion. The drawback of their algorithm is that each insertion and deletion requires to update $\Omega(1/\varepsilon)$ groups in the stabbing partition. We present a similar algorithm with the same amortized cost, but each insertion and deletion only requires to update one group in the stabbing partition. Recall that changes in the stabbing partition often need to be propagated to other data structures associated with SSI at runtime. Our implementation therefore requires much less frequent propagations and is more suitable for real-time applications.

Next we describe the algorithm in detail. Let $P = \{p_1, \ldots, p_{\tau_0}\}$ be the stabbing set of I returned by the greedy algorithm (Lemma 1), and let $\mathcal{I} = \{I_i \mid 1 \leq i \leq \tau_0\}$ be the stabbing partition of I corresponding to P. For each $I_i \in \mathcal{I}$, we construct a height-balanced binary tree $\mathcal{T}_i$ that supports each of INSERT, DELETE, SPLIT, and JOIN operations in $O(\log n)$ time [22]. The leaves of $\mathcal{T}_i$ store the intervals of I_i from left to right in increasing order of the left endpoints of intervals in I_i. Each internal node v of $\mathcal{T}_i$ with w and z as its two children stores an interval $\gamma_v = \gamma_w \cap \gamma_z$. Note that $\bigcap I_i$, the common intersection of all the intervals in I_i, is stored at the root of $\mathcal{T}_i$.

We handle each newly inserted interval γ by picking a point $p_\gamma \in \gamma$ and letting $P = P \cup \{p_\gamma\}$, and adding a singleton group $I_\gamma = \{\gamma\}$ into $\mathcal{I}$.[2] Let J be the set of intervals inserted since the last reconstruction of the stabbing partition. Note that these intervals appear as singleton sets in $\mathcal{I}$. When an interval γ is deleted, we first check whether $\gamma \in J$. If $\gamma \in J$, we simply remove p_γ from P, γ from J, and $\{\gamma\}$ from $\mathcal{I}$. Otherwise we find I_i such that $\gamma \in I_i$, and delete γ from I_i and the tree $\mathcal{T}_i$; we discard I_i and $\mathcal{T}_i$ if I_i becomes empty. Thus each insertion or deletion takes $O(\log n)$ time, and affects at most one group in the stabbing partition.

After we have performed $\varepsilon\tau_0/(\varepsilon + 2)$ updates, we recompute the optimal stabbing set of I, as well as a corresponding partition of I and the tree structure for each group in the partition, by a *reconstruction stage* that emulates the greedy algorithm for computing an optimal stabbing partition (Lemma 1). It outputs

[2] Again, this can be refined as in the basic strategy: if γ is already stabbed by some point $p \in P$, and suppose that p is the leftmost such point in P, we can simply add γ to the group whose stabbing point is p. The choice of such p is for maintaining the invariant ($\star$) to be stated shortly. For simplicity, we refrain from involving this refinement in our subsequent discussions.

exactly the same stabbing set and partition of I as the greedy algorithm. However, instead of examining each interval of I explicitly, which would require $O(n)$ time after sorting, it examines each of the $O(\tau_0)$ groups in $\mathcal{I}$, in $O(\log n)$ time per group, thereby reducing the running time to $O(\tau_0 \log n)$.

Algorithm RECONSTRUCTIONSTAGE

Input: $\{I_i \mid i \geq 1\} \cup \{I_\gamma \mid \gamma \in A\}$ (sorted), $\mathcal{T}_i$ for each I_i.
Output: smallest stabbing set P, a corresponding stabbing partition $\mathcal{I}$ of I, the tree structure $\mathcal{T}_J$ for each $J \in \mathcal{I}$.

$P, \mathcal{I} \leftarrow \emptyset$; $U, V, \mathcal{T}_U \leftarrow \emptyset$;
  {The collection of all intervals in the sets of U or V is denoted by J}
while $\exists$ unprocessed nonempty groups **do**
  $K \leftarrow$ next unprocessed nonempty group;
  Mark K processed;
  if $l(K) \leq r(\bigcap J)$ **then**
    if $K = I_\gamma$ for some $\gamma \in A$ **then**
      $V \leftarrow V \cup \{\gamma\}$;
    else $\{K = I_i$ for some $1 \leq i \leq \tau_0\}$
      $I_i', I_i \leftarrow$ SPLIT$(I_i, r(\bigcap J))$;
      $\mathcal{T}_i', \mathcal{T}_i \leftarrow$ SPLIT$(\mathcal{T}_i, r(\bigcap J))$;
      $U \leftarrow U \cup \{I_i'\}$, $\mathcal{T}_U \leftarrow$ JOIN$(\mathcal{T}_U, \mathcal{T}_i')$;
      Re-mark I_i unprocessed if $I_i \neq \emptyset$;
  else $\{l(K) > r(\bigcap J)\}$
    if $K = I_\gamma$ for some $\gamma \in A$ **then**
      $I_i \leftarrow$ next unprocessed nonempty group in $\{I_j \mid 1 \leq j \leq \tau_0\}$;
      $I_i', I_i \leftarrow$ SPLIT$(I_i, r(\bigcap J))$;
      $\mathcal{T}_i', \mathcal{T}_i \leftarrow$ SPLIT$(\mathcal{T}_i, r(\bigcap J))$;
      $U \leftarrow U \cup \{I_i'\}$, $\mathcal{T}_U \leftarrow$ JOIN$(\mathcal{T}_U, \mathcal{T}_i')$;
    $\mathcal{T}_J \leftarrow \mathcal{T}_U$;
    for every $\gamma \in V$ **do**
      INSERT$(\mathcal{T}_J, \gamma)$;
    $\mathcal{I} \leftarrow \mathcal{I} \cup \{J\}$, $P \leftarrow P \cup \{r(\bigcap J)\}$;
    if $K = I_\gamma$ for some $\gamma \in A$ **then**
      $U, \mathcal{T}_U \leftarrow \emptyset$, $V \leftarrow \{\gamma\}$;
    else $\{K = I_i$ for some $1 \leq i \leq \tau_0\}$
      $U \leftarrow \{I_i\}$, $\mathcal{T}_U \leftarrow \mathcal{T}_i$, $V \leftarrow \emptyset$;

Figure 13: Pseudo-code for the reconstruction stage.

Throughout the reconstruction, we have the following invariant:

($\star$) *For any $\gamma \in I_i$ and $\xi \in I_j$ with $i < j$, the left endpoint of ξ lies to the right of the left endpoint of γ.*

This invariant holds at the start of the reconstruction stage, because it clearly holds for the initial partition

of I computed by the greedy algorithm, and during subsequent updates, intervals are only removed from but never added into each I_i. During the reconstruction stage, the invariant will remain valid because of the same reason.

Let $L_1, L_2, \ldots$ be the sets in $\mathcal{I} = \{I_i \mid 1 \le i \le \tau_0\} \cup \{I_\gamma \mid \gamma \in J\}$ in increasing order of the left endpoints of their respective common intersections. The reconstruction procedure processes the sets $L_1, L_2, \ldots$ one by one, and outputs a new stabbing set and the corresponding new group of I as well as the binary tree structure for each group in the partition. During the process, it maintains a set A of *active* intervals, whose common intersection $\gamma_A = \bigcap A$ is nonempty. Intuitively, the active set A consists of a set of intervals on the frontier of this process, but has not yet formed a complete group because more intervals might be added into it later. Note that for a reason that will become clear shortly, we do not maintain A explicitly, but represent A as a family of subsets of sets in $\mathcal{I}$. Initially we set $A = L_1$ and start processing L_2. Suppose we have processed $L_1, \ldots, L_u$, and returned $q_1, \ldots, q_x$ as stabbing points and the corresponding groups $Q_1, \ldots, Q_x$. We now process L_{u+1}. Let $\gamma_{u+1} = \bigcap L_{u+1}$, which is stored at the root of the binary tree for L_{u+1}. There are two cases to consider.

Case 1: $\gamma_A \cap \gamma_{u+1} \neq \emptyset$, i.e., all intervals in L_{u+1} intersect r_A. We add L_{u+1} to A, set $\gamma_A = \gamma_A \cap \gamma_{u+1}$, and process L_{u+2}; see Figure 14.

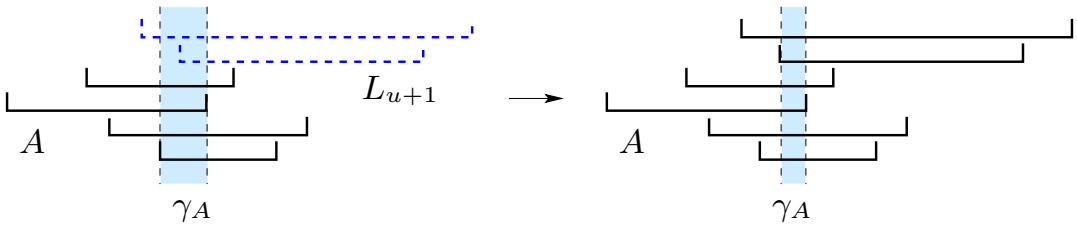


Figure 14: Case 1: $\gamma_A \cap \gamma_{u+1} \neq \emptyset$. Intervals in A are solid, and intervals in L_{u+1} are dashed. We add L_{u+1} to A.

Case 2: $\gamma_A \cap \gamma_{u+1} = \emptyset$. If $L_{u+1} \in \{I_\gamma \mid \gamma \in J\}$, we let L be the leftmost unprocessed group in $\{I_i \mid 1 \le i \le \tau_0\}$ (Figure 15 (a)); otherwise we set $L = L_{u+1}$ (Figure 15 (b)). Let L' be the subset of intervals in L whose left endpoints lie to the left of the right endpoint of γ_A. We split L into two sets L' and (new) $L = L \setminus L'$; we also split the binary tree structure $\mathcal{T}_L$ for L into $\mathcal{T}_{L'}$ and (new) $\mathcal{T}_L$ accordingly. Note that the left endpoint of $\bigcap L$ remains the same. We add L' to A and set $\gamma_A = \gamma_A \cap \gamma'$. Note that by ($\star$), no other interval from $\bigcup_{i \ge u+1} L_i$ can be added into A such that $\bigcap A$ remains nonempty. We then output a new stabbing point q_{x+1} to be the right endpoint of γ_A and output a new group $Q_{x+1} = A$.

We need to construct the tree $\mathcal{T}_{Q_{x+1}}$ for Q_{x+1}. Let $A^{old} = A \setminus J$ and $A^{new} = A \cap J$. Suppose A^{old} is represented as $L'_1, \ldots, L'_k$, where each L'_i is a subset of a group $I_j \in \mathcal{I}$, and they are sorted in increasing order of the indices of corresponding groups in $\mathcal{I}$. By ($\star$), the left endpoints of intervals in L'_i lie to the left of those in L'_{i+1}, so we can merge the trees of $L'_1, \ldots, L'_k$ to construct $\mathcal{T}_{A^{old}}$. Next, we insert the intervals of A^{new} into this tree. We thus have $\mathcal{T}_{Q_{x+1}}$. Finally, we set $A = L_{u+1}$ and start processing L_{u+2}.

In summary, at most $\varepsilon\tau_0/3$ INSERTIONS, $(1+\varepsilon/3)\tau_0$ SPLITS and $(1+\varepsilon/3)\tau_0$ JOINS are invoked in the reconstruction stage. Therefore the total running time is $O(\tau_0 \log n)$. Since the procedure is invoked only after $\varepsilon\tau_0/(\varepsilon+2)$ update operations, the amortized time for each update is thus $O((1+1/\varepsilon)\log n)$. One can verify that the procedure emulates the behavior of the greedy algorithm (Lemma 1) (although in a batched manner), and thus returns the same optimal stabbing partition as the greedy algorithm does.

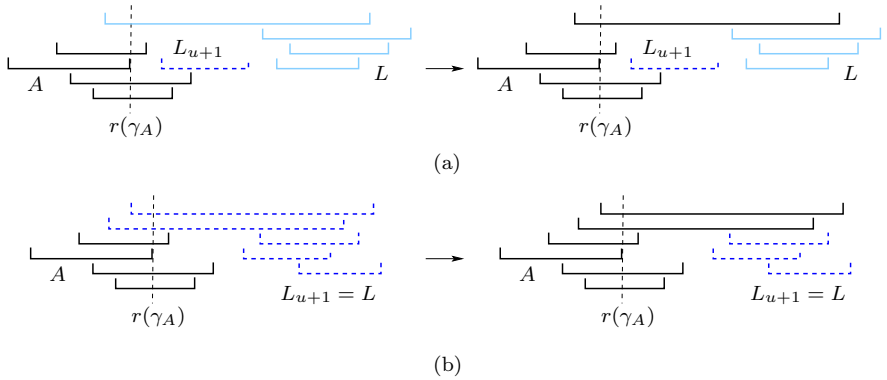


Figure 15: Case 2: $\gamma_A \cap \gamma_{u+1} = \emptyset$. Intervals in A are solid, and intervals in L_{u+1} are dashed. $r(\gamma_A)$ denotes the right endpoint of γ_A. (a) L is set to be the leftmost unprocessed group in $\{I_i \mid 1 \leq i \leq \tau_0\}$; (b) $L = L_{u+1}$.

References

[1] P. K. Agarwal, J. Xie, J. Yang, and H. Yu. Monitoring continuous band-join queries over dynamic data. In *Proc. of the 16th Intl. Sympos. Algorithms and Computation*, pages 349–359, 2005.

[2] L. Arge and J. Vitter. Optimal external memory interval management. *SIAM J. Comput.*, 32(6):1488–1508, 2003.

[3] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proc. of the 19th ACM SIGMOD Intl. Conf. on Management of Data*, pages 261–272, 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 *Proc. of the 28th Intl. Conf. on Very Large Data Bases*, pages 215–226, 2002.

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

[6] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagraCQ: A scalable continuous query system for internet databases. In *Proc. of the 19th ACM SIGMOD Intl. Conf. on Management of Data*, pages 379–390, 2000.

[7] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2nd edition, 2000.

[8] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An evaluation of non-equijoin algorithms. In *Proc. of the 17th Intl. Conf. on Very Large Data Bases*, pages 443–452, 1991.

[9] J.-P. Dittrich, P. M. Fischer, and D. Kossmann. Agile: adaptive indexing for context-aware information filters. In *Proc. of the 24th ACM SIGMOD Intl. Conf. on Management of Data*, pages 215–226, 2005.

[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 *Proc. of the 2nd Workshop on Algorithms and Data Structures*, 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 *Proc. of the 15th Intl. Conf. on Data Engineering*, pages 266–275, 1999.

[13] S. Har-Peled and S. Mazumdar. Coresets for k-means and k-median clustering and their applications. In *Proc. of the 36th Annu. Sympos. Theory of Computing*, 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 *Proc. of the 24th Intl. Conf. on Very Large Data Bases*, pages 275–286, 1998.

[15] M. Katz, F. Nielsen, and M. Segal. Maintenance of a piercing set for intervals with applications. *Algorithmica*, 36(1):59–73, 2003.

[16] N. Koudas, S. Muthukrishnan, and D. Srivastava. Optimal histograms for hierarchical range queries. In *Proc. of the 19th ACM Sympos. on Principles of Database Systems*, pages 196–204, 2000.

[17] L. Liu, C. Pu, and W. Tang. Continual queries for Internet scale event-driven information delivery. *IEEE Trans. on Knowledge and Data Engineering*, 11(4):610–628, 1999.

[18] S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. of the 21st ACM SIGMOD Intl. Conf. on Management of Data*, pages 49–60, 2002.

[19] E. M. McCreight. Priority search trees. *SIAM J. Comput.*, 14:257–276, 1985.

[20] J. Pereira, F. Fabret, H. A. Jacobsen, F. Llirbat, and D. Shasha. Webfilter: A high-throughput XML-based publish and subscribe system. In *Proc. of the 27th Intl. Conf. on Very Large Data Bases*, pages 723–724, 2001.

[21] Special issue on data stream processing. *IEEE Data Eng. Bull.*, 26(1), 2003.

[22] R. E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.

[23] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules For Advanced Database Processing.* Morgan Kaufmann, 1996.