Input-Sensitive Scalable Continuous Join Query Processing

Anonymous

This paper considers the problem of scalably processing a large number of continuous queries. Our approach, consisting of novel data structures and algorithms and a flexible processing framework, advances the state of the art in several ways. First, our approach is query-sensitive in the sense that it exploits potential overlaps in query predicates for efficient group processing. We partition the collection of continuous queries into groups based on the clustering patterns of the query predicates, and apply specialized processing strategies to heavily-clustered groups (or *hotspots*). We show how to maintain the hotspots efficiently, and use them to scalably process continuous select-join, band-join, and window-join queries. Second, our approach is also data-sensitive, in the sense that it makes cost-based decisions on how to processing each incoming tuple based on its characteristics. Experiments demonstrate that our approach can improve the processing throughput by orders of magnitude.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*query processing*

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Continuous queries, data streams, publish/subscribe, event matching

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 [Widom and Ceri 1996; Hanson et al. 1999], data monitoring [Carney et al. 2002], stream processing [Special 2003; Gehrke and Hellerstein 2004] and publish/subscribe systems [Liu et al. 1999; Chen et al. 2000; Pereira et al. 2001; Dittrich et al. 2005]. 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 address the challenge of scalability in a continuous query system by exploiting opportunities for *input-sensitive processing* with techniques that adapt to the characteristics of data and queries.

Challenge of Scalability

Consider a *continuous query* defined by a relational expression Q over a database $\mathcal{D}$. When this query is initially issued, it returns $Q(\mathcal{D}_0)$, where $\mathcal{D}_0$ represents the database state at that time. Then, for each subsequent database modification that changes the database state from $\mathcal{D}_{i-1}$ to $\mathcal{D}_i$, the query needs to return the changes (in the form of additions, deletions, and/or updates) from $Q(\mathcal{D}_{i-1})$ to $Q(\mathcal{D}_i)$, if any.[1] How can a continuous query system handle thousands or even millions of such continuous queries? For each incoming

[1]Note that we base our definition of continuous queries on the semantics of relational model and queries, instead of stream processing [Special 2003; Gehrke and Hellerstein 2004]. Our queries can be regarded as continuously running over database modification streams and returning result modification streams. Unlike in typical stream processing, we do not impose the restriction that queries must be evaluated in main memory, or that (conse-

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

A powerful observation made by recent work on scalable continuous query processing is the interchangeable roles of queries and data. Continuous queries can be treated as data, while each data tuple can be treated as a query requesting the subset of continuous queries affected by the tuple. Thus, it is natural to apply indexing and 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., [Hanson et al. 1999; Chen et al. 2000; Madden et al. 2002] and numerous 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* [de Berg et al. 2000] or *interval skip list* [Hanson and Johnson 1991]. 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 are over more than one input stream. Most existing work on indexing relational continuous queries has only focused on simple selection conditions or conjunction of selection conditions. With notable exceptions in [Chandrasekaran and Franklin 2003; Agarwal et al. 2005; Lim et al. 2006; Agarwal et al. 2006], 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.

Opportunities for Input-Sensitive Processing

We observe that in many practical applications (such as publish/subscribe systems), the set of continuous queries often exhibit clustering patterns that reflect overlapping user interests. The main idea is to exploit such patterns for more efficient group processing. 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.

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

quently) joins must be *windowed* in order to bound execution state. Nonetheless, our techniques can be readily extended to stream settings, and we show how to handle window joins.

Also, note that our definition of continuous queries is by no means the most general possible. For example, we do not consider continuous queries that allow their results to be refreshed periodically by time (as opposed to whenever they change).

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 scattered queries will be the minority, so overall we gain a significant speedup in processing all continuous queries.

Besides the above idea of *query-sensitive processing*, a complementary aspect of input-sensitive processing is *data-sensitive processing*. The observation here is that during the course of continuous query processing, we may encounter incoming data with different characteristics that warrant change in processing strategy. Dynamic adaptation of query execution has been studied extensively by a number of systems (e.g., [Ives et al. 1999; Avnur and Hellerstein 2000; Madden et al. 2002; Markl et al. 2004; Babu et al. 2005]). Most of them process most of the incoming data using a single, current best plan, which adapts over time but typically does not change for every tuple (with the notable exception of [Bizarro et al. 2005], which we discuss further in Section 6). In a system with a large number of continuous queries, the cost of processing each incoming tuple can be substantial. Given the high cost-saving potential, we argue that is beneficial to support more aggressive data-sensitive processing that makes cost-based decisions to switch among alternative query plans for every input tuple. This approach can be regarded as another example of interchanging the roles of queries and data, where each incoming tuple is optimized as a separate query over the set of the continuous queries.

Contributions

To materialize the ideas above, we need to address three 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; 3) how to combine hotspot-based processing and other processing techniques into a flexible, data-sensitive framework that makes cost-based decisions on how to process each incoming tuple. This paper addresses these issues and makes the following contributions:

—In Section 2, we introduce the notions of *stabbing partition* and *stabbing set index* (*SSI* for short) as a principled method for discovering and exploiting the clustering patterns in query predicates. We further introduce the notion of *hotspots* to capture large clusters, 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 group processing. We study three types of continuous join queries:
  —Continuous *band joins* [DeWitt et al. 1991], whose join conditions test whether the difference between two join attribute values falls within some range. Traditional approaches are based on sharing processing of *identical* join operations, and therefore do not apply in this case. Our technique, however, is able to group-process different band join conditions efficiently.
  —Continuous equality joins with different local range selections, or *select joins* for short. Traditional approaches group-process join and selection operations separately; therefore, they are prone to the problem of large intermediate results generated by

applying one operation earlier than the other, which hurts overall performance. In contrast, our technique is able to group-process select joins as a whole, thereby avoiding this problem.

—Continuous *window joins*, or select joins extended with additional window predicates, where joining tuples' timestamps must fall within a window of prescribed length. Different window joins may specify different local range selections and join window lengths. For each incoming tuple, our technique can quickly identify affected window joins without enumerating all joining tuples in the largest window.

—In Section 4, we present a flexible, cost-based processing framework which is able to dynamically route incoming event to the most promising query plan based on runtime data and query characteristics. We identify the statistics we need to monitor and build cost models for alternative processing strategies.

—In Section 5, 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.

—As another application of stabbing partition, we show in Appendix B how to build a high-quality histogram for a set of intervals in linear time.

2. TRACKING HOTSPOTS

Consider a set of continuous queries whose query ranges are defined over a numerical attribute A. Informally, if many query ranges contain a common value x, then these queries form a "hotspot" around x.[2] In general there could be a number of hotspots for a set of queries, depending on the distribution of their query ranges.

How do hotspots help with group-processing of continuous queries? We offer some high-level intuition here. For an arbitrary collection of query ranges, it is difficult to impose a total order among them that facilitates processing. However, if a group of query ranges share a common point p, we can regard each range as the disjoint union of two subranges, one to the left of p and the other to the right. All subranges to the left of p can be naturally ordered by their left endpoints; this ordering also reflects the containment relationships among these subranges, opening up efficient group-processing opportunities. Ordering all subranges to the right of p by their right endpoints offers similar benefits. Section 3 will discuss in detail how this idea is applied to three types of continuous join queries. In this section, we focus on techniques for grouping query ranges into hotspots and maintaining this grouping.

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*

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

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.

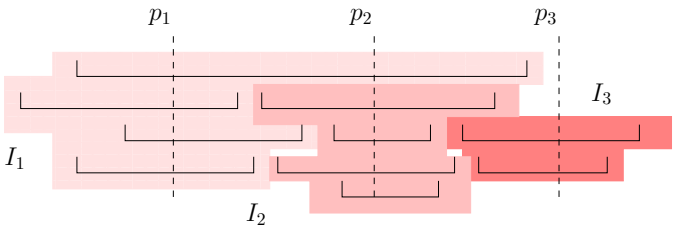


Fig. 1. A stabbing partition of 10 intervals.

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

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

We next briefly introduce the general concept of *stabbing set index* (*SSI*), 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.

2.2 Hotspots

The basic idea of using SSI to exploit the clustering patterns in continuous queries is to efficiently group-process queries from each stabbing group. However, a stabbing group with a small number of queries does not benefit from specialized processing techniques aimed at a large group of queries; such techniques would only incur extra overhead. In practice, groups in the SSI are often unbalanced, as illustrated by the following simple

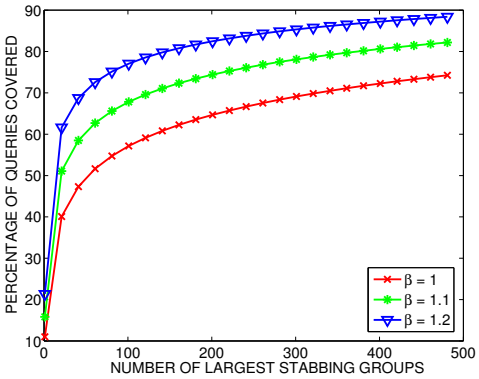


Fig. 2. Coverage by large k stabbing groups in Zipfian distributions.

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 same item, Zipf's law states that the number of queries within a stabbing group is roughly inversely proportional to its rank in popularity. That is, the number n_k of queries in the k-th largest group is proportional to $k^{-\beta}$, where β is a positive constant close to 1. Suppose there are a total number of 5000 stabbing groups. Figure 2 shows the percentage of queries covered by the top-k largest stabbing groups out of all 5000 stabbing groups, where 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*, aimed at identifying large stabbing groups for efficient group processing.

DEFINITION 2. *Let $\alpha > 0$ be a fixed parameter. Suppose $\mathcal{I} = \{I_1, I_2, \ldots\}$ 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.*

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. For example, in Figure 1, I_1 and I_2 are 0.4-hotspots. Note that the number of α-hotspots is at most $1/\alpha$ by definition.

It is quite easy to identify all hotspots once a stabbing partition $\mathcal{I}$ of I is given. As continuous queries are inserted or deleted, however, the hotspots may evolve over time. Therefore, we need an efficient mechanism to keep track of the evolution of the hotspots. When designing such a hotspot-tracking scheme, one needs to keep the following two issues in mind:

—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 that the size of $\mathcal{I}$ is as small as possible, because fewer stabbing groups capture more clustering and lead to more efficient processing. 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 $\tau(I)$.

—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 indexes 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 properties at all times:

(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).

(I1) ensures that $\mathcal{I}_H$ contains only groups that are "hot enough"; the leeway between $\alpha/2$ and α lowers maintenance cost and makes (I3) possible. (I2) satisfies our design requirement that the size of the stabbing partition we maintain is not far from the optimum. (I3) satisfies the other design requirement that, with each update, only a small number of intervals can change from hotspot to scattered, and vice versa.

We need the following lemma, which states that one can maintain a stabbing partition of I of size close to $\tau(I)$ in amortized logarithmic time per update. Katz et al. [Katz et al. 2003] first proved this result by presenting an algorithm with the claimed performance bound. In Appendix A we will describe a slightly better algorithm that is more suitable for real-time applications, as well as simpler and more 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.

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 [McCreight 1985]); 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.

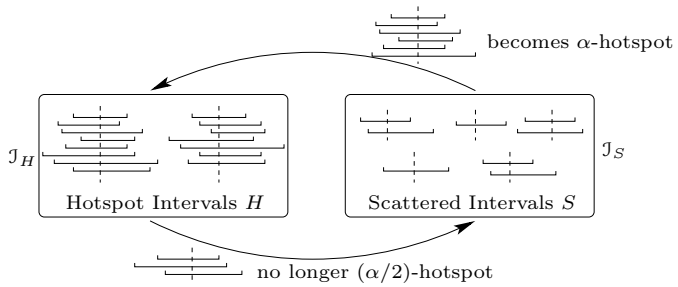


Fig. 3. Schematic view of the hotspot-tracking algorithm.

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 properties (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 $|\mathcal{I}_S| \le (1+\varepsilon)\tau(S) \le (1+\varepsilon)\tau(I)$. By (I1), we also have $|\mathcal{I}_H| \le 2/\alpha$. Hence,

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

(I3) We prove this property 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 $\mathcal{I}_H$. Furthermore, if γ does not fall into any group of $\mathcal{I}_H$ (recall that in this case our algorithm inserts γ into S), we deposit another one credit to γ. Since $|\mathcal{I}_H| \leq 2/\alpha$ 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 $\mathcal{I}_H$, 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) follows easily from (ii). So we only have to show (ii).

Let $I_i \in \mathcal{I}_H$ be a group to be demoted. We know that I_i was promoted to $\mathcal{I}_H$ 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 $\mathcal{I}_S$ using Lemma 2. Since the amortized number of intervals moving in and out of S is $O(1)$ per update, by Lemma 2, we know that the amortized cost for updating $\mathcal{I}_S$ is $O(\varepsilon^{-1} \log |I|)$. □

3. PROCESSING CONTINUOUS JOINS WITH HOTSPOTS

In this section we present three applications of our stabbing set index (SSI) and hotspot-tracking scheme to scalable processing of continuous joins. We first consider two types of continuous queries over relations $R(A, B)$ and $S(B, C)$:

—**Equality join with local selections (select join)**: $\sigma_{A \in rangeA} R \bowtie_{R.B=S.B} \sigma_{C \in rangeC} S$

—**Band join**: $R \bowtie_{S.B-R.B \in rangeB} S$

In a select-join, the query parameters $rangeA$ and $rangeC$ in the local selection conditions are ranges over numeric domains of $R.A$ and $S.C$, respectively. In a 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.

Select-join example. Consider a listing database for merchants with two relations:

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

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.

Band-join example. 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.

Besides select joins and band joins, we also consider window joins, defined over relations $R(A, B, T)$, and $S(B, C, T)$, where T stores the timestamp of each tuple. We assume that R and S are append-only and that the new tuples arrive in increasing timestamp order.

—**Window join (select join with window predicate)**:

$$\sigma_{A \in rangeA} R \bowtie_{R.B=S.B \wedge |R.T-S.T| \leq w} \sigma_{C \in rangeC} S$$

Window joins can be seen as generalizing select joins with special band-join conditions. The window predicate is essentially a band-join condition with range $[-w, w]$. This predicate ensures that each incoming tuple joins only with tuples arrived within a window of

query-specified length w. Such window joins are common in applications that deal with streaming data.

3.1 Band Joins

We first consider the problem of processing a group of continuous band joins 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.

3.1.1 *Previous Approaches.* We first note that existing techniques based on sharing identical join operations [Chen et al. 2000] 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 [Chandrasekaran and Franklin 2003], 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 approach of [Chandrasekaran and Franklin 2003], 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 $\{rangeB_i\}$.[3] 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 strategy 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 2 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.

[3] Possibilities include *priority search tree* [de Berg et al. 2000] and *external interval tree* [Arge and Vitter 2003].

3.1.2 *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 $S(B)$.

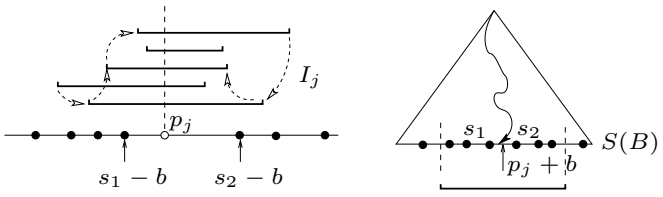


Fig. 4. The SSI algorithm for band join processing. Arrows indicate the order in which the intervals are visited.

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

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

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

—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);

—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 2. *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:* $O(\tau \log m + k)$*;*

3.1.3 *The Hotspot Approach.* 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 2 (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 with the other processing strategies. The scattered queries, on the other hand, are processed separately using one of 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.

3.2.1 *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 $\{rangeA_i\}$ (cf. footnote 3). 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 by the first step 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 by their first steps.

3.2.2 *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.

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.

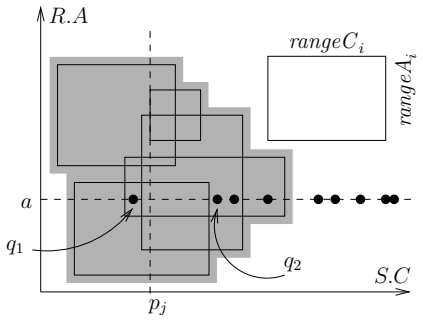


Fig. 5. The SSI algorithm for processing equality joins with local selections.

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 (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:

—SJ-SSI never considers a join result point unless it is covered by some query rectangle or is closest to some stabbing point;

—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 3. *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: $O(\tau(\log m + g(n)) + k)$.*

3.2.3 *The Hotspot Approach.* 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 2 (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'. On the other hand, for the scattered queries, we can still use the other two algorithms.

3.3 Window Joins

We now consider how to process a group of continuous window joins Q_i of the form

$$\sigma_{A \in rangeA_i} R \bowtie_{R.B=S.B \wedge |R.T-S.T| \leq w_i} \sigma_{C \in rangeC_i} S.$$

Recall that T denotes the timestamp attributes, and that the new tuples arrive in increasing timestamp order. When a new R-tuple r arrives, we need to identify the subset of queries whose results are affected by r.

To help describe solutions to this problem, we give a geometric representation of the window joins. Each query Q_i can be regarded as a query box $rangeA_i \times rangeB_i \times [0, w_i]$ in the three-dimensional product space $R.A \times S.C \times W$, where the W dimension captures lengths of join windows.

3.3.1 *Baseline Solutions.* Analogous to SJ-JoinFirst and SJ-SelectFirst presented in Section 3.2.1 for select joins, we have two basic strategies:

—*WJ-JoinFirst* (*window-join processing with join first*) proceeds as follows. We 1) join $r = (a, b, t)$ with S, and 2) for each join result tuple, find the queries whose local selections are satisfied. In more detail, the first step, identifying joining S-tuples, amounts to evaluating a selection $S.B = b \wedge S.T \geq t - w_{\max}$ over S, where $w_{\max} = \max_i w_i$ is the maximum window length among all queries. This selection can be supported by a B-tree index on $S(B, T)$. The second step, processing each join result tuple with $S.C = c$ and $S.T = t'$, is handled by a data structure $\mathcal{D}$. $\mathcal{D}$ indexes the set of all window joins (three-dimensional boxes) such that, given a point in the space $R.A \times S.C \times W$, we can quickly find the subset of boxes containing this point.[4] Specifically, we probe $\mathcal{D}$ with the point $(a, c, t - t')$ for the set of boxes containing it. Every window join affected by the incoming tuple r can be found by probing $\mathcal{D}$ with some join result tuple.

—*WJ-SelectFirst* (*window-join processing with selection first*) proceeds as follows. We 1) identify those queries whose local selections on R are satisfied by the incoming tuple $r = (a, b, t)$, and 2) for each such query, check whether it is actually affected, i.e., whether some S-tuple satisfies both the window-join condition and the local selection condition on S. In more detail, the first step probes an index on query ranges $\{rangeA_i\}$, as in SJ-SelectFirst (cf. footnote 3). The second step, for each query Q_i identified by the first step, we look for S-tuples with $S.B = b \wedge S.C \in rangeC_i \wedge S.T \geq t - w_i$. This lookup can be supported efficiently by a two-level data structure, where the first level is a B-tree indexing $S.B$, and the second level is a data structure supporting two-dimensional range queries, e.g., a range tree or R-tree indexing $(S.C, S.T)$.

[4] Theoretically, a two-level segment tree [Vaishnavi 1982] produces polylogarithmic query and update time; in practice, some variant of R-trees will serve the same purpose (albeit without the theoretical worst-case performance guarantees).

Like SJ-JoinFirst and SJ-SelectFirst, the performance of WJ-JoinFirst and WJ-SelectFirst can be adversely affected by potentially large intermediate results generated by the first step of these algorithms. In the case of WJ-JoinFirst, every join result tuple leads to a probe into $\mathcal{D}$, even though many probes may return similar subsets of queries, or no queries at all. In the case of WJ-SelectFirst, an incoming R tuple can satisfy many queries' local selection conditions on $R.A$, even though it may actually affect few or none of such queries.

3.3.2 *The SSI Approach.* We now present an alternative, *WJ-SSI* (*window-join processing with SSI*), which uses the ideas of SSI and *skylines* to effectively reduce the number of probes required to identify all affected queries as compared with WJ-JoinFirst. Like SJ-SSI, partitioning of queries into stabbing groups allows efficient group processing of queries in each stabbing group. In this case, however, two probes per stabbing group are no longer sufficient. Because of the additional timestamp dimension, we instead need to probe with the points on two skylines around each stabbing point.

Before describing the details of WJ-SSI, we review the notion of two-dimensional *skylines* below. Let P be a set of points in a plane and h be a vertical line in the plane. We denote by P_h^+ the subset of points in P lying to the right of h, and by P_h^- the subset of points in P lying to the left of h. The skyline of P *to the right of* h is the set

$$\Big\{(x,y) \in P_h^+ \mid \nexists (x',y') \in P_h^+ \text{ such that } x' < x \text{ and } y' < y\Big\}.$$

Similarly, the skyline of P *to the left of* h is the set

$$\Big\{(x,y) \in P_h^- \mid \nexists (x',y') \in P_h^- \text{ such that } x' > x \text{ and } y' < y\Big\}.$$

WJ-SSI employs the following data structures in addition to $\mathcal{D}$ (defined earlier in WJ-JoinFirst):

(1) We maintain an SSI for the set of query intervals $\{rangeC_i\}$. Let $p_1, \cdots, p_\tau$ be the stabbing points of this SSI in sorted order.

(2) For each value (say b) of $S.B$ in S, and for each stabbing point p_i, we maintain two skylines $\mathcal{L}_b^i$ and $\mathcal{R}_b^i$ defined as follows. In the plane $S.C \times T$, let

$$S_b = \{(s.C, s.T) \mid s \in B \wedge s.B = b \wedge s.T \geq t - w_{\max}\},$$

where t denotes the current time and $w_{\max}$ is the maximum window length among all queries. Intuitively, S_b corresponds to the S-tuples that will join with an incoming R-tuple with $R.B = b$. Let each point p_i induce a vertical line $h_i : S.C = p_i$ in the plane $S.C \times T$. Let S_b^0 denote the subset of S_b lying to the left of h_1, let S_b^i $(1 \leq i < \tau)$ denote the subset of S_b between h_i and h_{i+1}, and let S_b^τ denote the subset of S_b lying to the right of h_τ. Then, for $i = 1, \ldots, \tau$:
—$\mathcal{L}_b^i$ is the skyline of S_b^{i-1} to the left of h_i;
—$\mathcal{R}_b^i$ is the skyline of S_b^i to the right of h_i.

(3) We index the (pointers to) skylines in a B-tree index, with a composite key formed by their corresponding $S.B$ and p_i values.

The size of the SSI in (1) above is linear in the number of queries. The size of the B-tree in (3) above is linear in the size of S. The total size of all skylines in (2) is also linear in

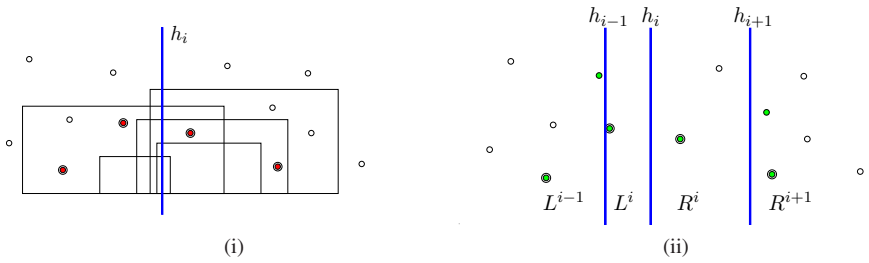


Fig. 6. (i) A rectangle in the i-th stabbing group is stabbed by one of the points if and only if it is stabbed by a point on the skylines to the left and right of h_i (double-circle points). (ii) Points on the skylines to the left and right of h_i (double-circle points) belong to the union of the set of skylines between consecutive vertical lines (colored points).

the size of S, because:

$$\sum_b \sum_{i=1}^{\tau} (|\mathcal{L}_b^i| + |\mathcal{R}_b^i|) \leq \sum_b \sum_{i=0}^{\tau} 2|S_b^i| = \sum_b 2|S_b| \leq 2|S|.$$

Using the data structures above, we can process each incoming R tuple $r = (a, b, t)$ as follows. First, we search the B-tree in (3) for the skylines associated with $S.B = b$. Let $\bar{S}_b = \bigcup_{i=1}^{\tau} (\mathcal{R}_b^i \cup \mathcal{L}_b^i)$ denote the set of skyline points found in this step. Next, for each point $(c, t') \in \bar{S}_b$ (in the plane $S.C \times T$), we probe $\mathcal{D}$ for queries whose corresponding boxes (in the space $R.A \times S.C \times W$) contain the point $(a, c, t - t')$.

We claim that the queries returned by the probes above constitute precisely those queries affected by the incoming tuple r. To see why, consider any query Q with box $rangeA \times rangeC \times [0, w]$. Suppose that $rangeC$ belongs to the stabbing group p_i in the SSI. It is not difficult to see that, in the plane $S.C \times W$, the following conditions are equivalent (see Figure 6(i)):

(a) The rectangle $rangeC \times [0, w]$ is stabbed by a point in $\{(c, t - t') \mid (c, t') \in S_b\}$.

(b) The same rectangle is stabbed by a point in $\{(c, t - t') \mid (c, t') \in \mathcal{L} \cup \mathcal{R}\}$, where $\mathcal{L}$ is the skyline of S_b to the left of h_i and $\mathcal{R}$ is the skyline of S_b to the right of h_i in the plane $S.C \times T$.

What remains to be shown is that the above conditions are equivalent to the following:

(c) The same rectangle is stabbed by a point in $\{(c, t - t') \mid (c, t') \in \bar{S}_b\}$.

To this end, consider any point $(c, t') \in \mathcal{L}$. Observe that (c, t') also belongs to $\mathcal{L}_b^j$, where j is the smallest index such that $c < p_j$. Similarly, for any point $(c, t') \in \mathcal{R}$, (c, t') belongs to $\mathcal{R}_b^j$, where j is the largest index such that $p_j < c$ (see Figure 6(ii)). Hence, $\mathcal{L} \cup \mathcal{R} \subseteq \bar{S}_b$, and therefore (b) implies (c). At the same time, $\bar{S}_b \subseteq S_b$, so (c) implies (a). Since (a) and (b) are equivalent, (c) must also be equivalent. Adding the check for $a \in rangeA$ in the $R.A$ dimension ensures that our algorithm correctly identifies all affected queries.

Finally, we discuss how to maintain the data structures used by WJ-SSI when data and queries change. We focus on maintaining the skylines in (2), because the SSI in (1) can be maintained using the technique in Appendix A, and the B-tree in (3) and $\mathcal{D}$ can be maintained using standard techniques. Recall that we assume R and S to be append-only and that the new tuples arrive in increasing timestamp order.

—For an insertion $s = (b, c, t)$ into S, we can identify the appropriate index i such that $(c, t) \in S_b^i$, using the B-tree in (3), which effectively sorts the stabbing points. Should the point (c, t) appear on the skyline $\mathcal{R}_b^i$ (or $\mathcal{L}_b^i$), it would replace a consecutive sequence of points on $\mathcal{R}_b^i$ (or $\mathcal{L}_b^i$). To enable logarithmic updates to skylines, we maintain each skyline in a search tree that supports logarithmic insert, tree split, and tree merge operations.[5]

—Besides data insertions, changes to queries may cause changes to stabbing points and therefore skylines. With our technique in Appendix A.2, the number of stabbing points affected by an insertion or deletion of a continuous query is $O(1)$. Therefore, the number of skylines that need recomputation is $O(1)$. Nevertheless, recomputing $O(1)$ skylines for every possible $S.B$ value would still be expensive. Instead, we take a lazy approach. When a stabbing point has changed, we mark the affected skylines as stale (for all $S.B$ values). Only when a particular $S.B$ value b is queried (by a incoming R tuple with $R.B = b$), we recompute the stale skylines associated with b. As is well known, a skyline can be computed in a single pass of the (sorted) data; in our context, there may be multiple stale skylines and we can recompute them together in a single pass over S_b (sorted by $S.C$). Hence, the query performance of our approach cannot be any worse than WJ-JoinFirst, which always requires a pass over S_b.

—One final technicality remains: We have defined S_b such that it contains only S-tuples whose timestamps are within $w_{\max}$ (the maximum window length) of the current time. However, it is expensive to update all skylines whenever the current time advances, and whenever $w_{\max}$ changes due to changes in queries. Again, we take a lazy approach instead. Only before we query a particular $S.B$ value b, we enforce that the skylines associated with b are computed over the correct time window. An advance of time or decrease in $w_{\max}$ can be processed by removing those skyline points outside of window, which involves only one logarithmic-time tree split operation per skyline. An increase in $w_{\max}$ requires extending the skylines, which can be done in a single pass over the tuples in S_b with timestamps earlier than the beginning of the old maximum window; therefore, the query performance remains no worse than WJ-JoinFirst, which makes a pass over all S_b tuples.

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

THEOREM 4. *Let n denote the number of continuous window joins, m denote the size of S, and k denote the output size. Furthermore, let $g_2(n)$ (and $g_3(n)$) denote the complexity of answering a stabbing query on an index of n two-dimensional boxes (three-dimensional boxes, respectively). The worst-case running times to identify all queries affected by an incoming R-tuple with $R.B = b$ are as follows, where $m' \leq m$ is the number of S-tuples with $S.B = b$, and $m_b = |S_b| \leq m'$ is the number of S-tuples that join with the incoming R-tuple for at least one of the window joins.*

—*WJ-JoinFirst: $O(\log m + m_b g_3(n) + k)$;*

—*WJ-SelectFirst: $O(\log n + \log m + n' g_2(m') + k)$, where $n' \leq n$ is the number of queries whose local selections on R are satisfied by the incoming tuple;*

[5] A variant of the B-tree [Agarwal et al. 1999] can support tree split/merge operations in an I/O-efficient manner.

—*WJ-SSI:* $O(\log m + m'_b + \bar{m}_b g_3(n) + k)$*, where* $\bar{m}_b = |\bar{S}_b| \leq m_b$ *is the number of unique skyline points associated with* b*, and* $m'_b = O(m_b)$ *is either the total size of all skylines associated with* b *(when lazy recomputation is not triggered) or the number of* S*-tuples for which skylines need to be recomputed.*

This theorem highlights the improvement of WJ-SSI over WJ-JoinFirst, as m_b (the number of joining S-tuples) is often much larger than $\bar{m}_b$ (the number of unique skyline points among these tuples). Note that although τ (the stabbing number) is not directly included in the complexity of WJ-SSI above, it affects the complexity of WJ-SSI through $\bar{m}_b$: A larger τ means more skylines, which usually imply a larger $\bar{m}_b$.

3.3.3 *The Hotspot Approach.* Applying WJ-SSI to the set $\mathcal{I}_H$ of Theorem 1 (i.e., the collection of hotspots), we get an efficient algorithm for processing the subset of hotspot queries. For the scattered queries, we can use either WJ-JoinFirst or WJ-SelectFirst.

4. DATA-SENSITIVE OPTIMIZATION OF CONTINUOUS JOINS

As we shall see in Section 5, there is no silver bullet for processing a large number of continuous joins; each algorithm may fare better or worse for certain inputs. While our hotspot-based algorithms in Section 3 are able to monitor the clustering patterns and only apply SSI-based processing to large clusters, it does not recognize the fact that different alternatives may work better for particular incoming tuples. For example, suppose we have a large number of continuous select-joins of the form

$$\sigma_{A \in rangeA} R \bowtie_{R.B=S.B} \sigma_{C \in rangeC} S.$$

If an incoming R-tuple can only satisfy a few queries' range selection conditions on R, it is possible that SJ-SelectFirst is the most efficient way to process this tuple. The hotspot-based algorithm still needs to examine the hotspots, which may not be as efficient. In Section 5, we will see experiments where the characteristics of incoming data have dramatic influence on the relative performance of different processing strategies, and that such characteristics may change from one incoming tuple to the next.

In this section, we propose a flexible, data-sensitive processing framework that makes cost-based decisions at runtime to process each incoming tuple using the most efficient plan for it. In this framework, each incoming tuple is first sent to a special routing operator that directs the incoming tuple to the most efficient plan among a set of pre-compiled alternatives. This operator has access to various statistics and makes use of the content of the incoming tuple in making routing decisions. Essentially, it optimizes each incoming tuple as a separate query over the database and the set of continuous queries. Since the number of pre-compiled alternatives is limited, the overhead of per-tuple optimization is small. For systems with a large number of continuous queries, this overhead is minimal compared with the cost-saving potential.

In the remainder of this section, we illustrate how to implement our framework using cost estimation and statistics collection techniques. We will focus on select-joins. We omit some details for band joins and window joins, as they are handled in a similar fashion; we only discuss issues that are unique or non-straightforward for these types of queries.

4.1 Optimizing Select-Joins

Consider a collection of continuous select-joins with the form given at the beginning of this section. First, we derive the cost models used in our cost-based processing framework.

Without loss of generality, we assume that the incoming tuple r is for table R, which joins with table S. Let Q denote the set of select-join queries. Let m' denote the number of S-tuples that join with the incoming tuple, and let n' denote the number of queries whose local selections on R are satisfied by the incoming tuple.

We consider four alternative processing strategies below. Note that the cost formulas below do not consider the cost of final output, since it is the same across all strategies. Also note that we estimate the lookup cost for an R-tree indexing n rectangles as a constant times $\sqrt{n}$ (not counting the cost of producing results). The worst-case lookup cost in an actual R-tree implementation may be linear in n, but is overly pessimistic for the purpose of cost estimation. On the other hand, the best-case lookup cost, $O(\log(n))$, is overly optimistic. We choose $\sqrt{n}$ as it reflects the theoretical lower bound for lookups in a two-dimensional R-tree.

—SJ-JoinFirst: As described in Section 3.2.1, we first join tuple r with S to find the joining S-tuples; this step requires a lookup on the B-tree on $S(B)$, which takes $O(\log |S|)$ time. Then, for each of the m' joining S-tuples, we probe the R-tree on all queries to identify affected queries; each probe takes $O(\sqrt{|Q|})$ time. Thus, the total query cost is

$$C_J \;=\; \alpha_J \log |S| + \delta_J m' \sqrt{|Q|},$$

where α_J and δ_J are constant parameters of the cost model.

—SJ-SelectFirst: This strategy is an improved version of the basic SJ-SelectFirst algorithm described in Section 3.2.1. First, we probe the index on all queries' $R.A$ selection ranges to identify those queries whose local selections on R are satisfied; this probe takes $O(\log |Q|)$ time. Next, instead of probing the B-tree on $S(B, C)$ repeatedly from the root, we share the cost across probes as follows. Note that all S-tuples that join with r conceptually form a contiguous substructure of the B-tree sorted and indexed by $S(C)$. We can identify this substructure in $O(\log |S|)$ time by probing the index on $S(B, C)$ using just the value of $r.B$; this step takes $O(\log |S|)$ time. Then, for each query identified earlier (whose local selection on R is satisfied by r), we look up S-tuples satisfying the corresponding local selection on S just within the substructure; each lookup only takes $O(\log |m'|)$ time. Thus, the total cost is

$$C_S \;=\; \alpha_S \log |S| + \beta_S \log |Q| + \gamma_S n' \log m',$$

where α_S, β_S, and γ_S are constant parameters of the cost model.

—*SJ-Vanilla*: This simple strategy requires only the B-tree on $S(B, C)$ and no indexes on queries. First, as in SJ-SelectFirst, we identify the substructure of the B-tree containing all joining S-tuples, which takes $O(\log |S|)$ time. Then, for each query, we probe this substructure to find results of the select-join; each probe takes $O(\log m')$ time. The total query cost is

$$C_V \;=\; \alpha_V \log |S| + \gamma_V |Q| \log m',$$

where α_V and γ_V are constant parameters of the cost model.

—*SJ-Hotspot*: This particular hotspot-based strategy (cf. Section 3.2.3) applies SJ-SSI (Section 3.2.2) to the hotspot queries, and SJ-SelectFirst (the improved version described above) to the scattered queries. Let $\{Q_i^H\}$ denote the set of hotspots, i.e., large stabbing groups of queries. Let Q^S denote the set of all scattered queries. The total query cost can be broken down into two components. The first component, for processing scattered

queries, is analogous to the cost of SJ-SelectFirst, but restricted to just Q^S. The second component, for processing hotspot queries, consists of the cost of processing each hotspot Q_i^H, which takes $O(\sqrt{|Q_i^H|})$ time for the R-tree lookup. The total query cost is

$$C_H \;=\; \alpha_H \log|S| + \beta_H \log|Q^C| + \gamma_H n'' \log m' + \delta_H \sum_i \sqrt{|Q_i^H|},$$

where n'' is the number of scattered queries that survive the first step of SJ-SelectFirst (i.e., those whose local selections on R are satisfied), and α_H, β_H, γ_H, and δ_H are constant parameters of the cost model.

The constant parameters in the cost formulas above are obtained by profiling the execution of different processing strategies, measuring the different component costs, and then fitting the parameters using standard statistical techniques. More details are described in Section 5.

The cost formulas above also use a number of quantities. The number of tuples in S ($|S|$) and the total number of queries ($|Q|$) are known at runtime; the sizes of hotspots ($|Q_i^H|$) and the number of scattered queries ($|Q^S|$) are also readily available because we maintain SSI for the hotspot queries. The other quantities, m', n', and n'', may vary depending on the incoming tuple r. They can be estimated at runtime as follows:

—m', the number of S-tuples that join with r, is the number of S-tuples satisfying the condition $S.B = r.B$. Given r, a standard histogram for $S(B)$ readily provides the estimate.

—n', the number of queries with their local selections on R satisfied by r, is the number of $R.A$ query ranges that are stabbed by $r.A$. We can use a simple bucket-based histogram for estimation. Conceptually, this histogram approximates a step function of a, which returns the number of query ranges in $\{rangeA_i\}$ stabbed by a. To update this histogram when a new query with range $rangeA$ is added, we simply raise the estimate by 1 for all buckets completely covered by $rangeA$, and raise the estimate for each partially covered bucket by an amount equal to the fraction of the bucket covered. Deletions of queries can be handled analogously.

—n'', the number of scattered queries with their local selections on R satisfied by r, can be estimated in a similar way as n', by maintaining a histogram for scattered queries alone.

We note that these techniques are widely used in database systems today, so our approach is practical and straightforward to support.

At runtime, to choose a processing strategy for an incoming tuple r, we simply evaluate the cost formulas of various strategies, by plugging in the pre-trained parameters and the exact or estimated quantities. The strategy with the lowest estimated total cost is chosen. Because of the limited number of strategies and the simplicity of cost estimation, the overhead of this runtime optimization is minimal.

Our cost-based data-sensitive optimization framework, combined with the query-sensitive, hotspot-based processing strategy, can capture a number of intuitive optimizations. First, if there are few joining S-tuples, then SF-JoinFirst would be the strategy of choice. Second, if there are few queries with their local selections on R satisfied by the incoming R-tuple, then SJ-SelectFirst wins. Finally, if most queries fall into a few hotspots, then SJ-Hotspot is likely more preferable. Instead of heuristically applying these intuitions, however, our approach identifies these optimization opportunities in a systematic way.

4.2 Optimizing Band Joins and Window Joins

For band joins, we only need $|Q|$, $|S|$, and τ (cf. Theorem 2) to make an informed choice among BJ-QOuter, BJ-DOuter, BJ-MJ, and BJ-SSI for processing an incoming R-tuple; all three quantities are readily available. To consider BJ-Hotspot in optimizing band join processing, we additionally need the sizes of hotspots and the number of scattered queries; again, such information is available from the SSI we maintain for the hotspot queries.

For window joins, to optimize processing of an incoming R-tuple with $R.B = b$, we need to know or estimate the following quantities in order to choose among WJ-JoinFirst, WJ-SelectFirst, and WJ-SSI: $|Q|$, $|S|$, n', m', m_b, $\bar{m}_b$, and m'_b (cf. Theorem 4). $|Q|$ and $|S|$ are known. Estimation of n', the number of queries whose local selections on R are satisfied, is exactly the same as the estimation of n' in Section 4.1. Note that all three algorithms use indexes whose top-level key is $S.B$—the B-tree with composite key $S(B, T)$ (WJ-JoinFirst), the two-level index whose first level is a B-tree indexing $S.B$ (WJ-SelectFirst), and the B-tree indexing skylines by $S.B$ (WJ-SSI). The three indexes can share the top level, and we can incrementally maintain m', m_b, and $\bar{m}_b$ at this level for each possible $S.B$ value. Since all three algorithms involve looking up $S.B = b$ anyway, we can perform this lookup first without any penalty, obtain these quantities, and then decide which algorithm to use for the remaining processing steps. Because of lazy update of skylines (Section 3.3.2), values of m_b and $\bar{m}_b$ may be inaccurate. To obtain reasonable estimates, we scale m_b by the ratio of $w_{\max}$ to the length of the window over which m_b is maintained, and we scale $\bar{m}_b$ by the ratio of 2τ (available from the SSI) to the number of skylines currently kept for b. Finally, we use $2\bar{m}_b$ as an (upper bound) estimate for m'_b if no recomputation is required (i.e., stabbing points have not changed); otherwise, we use m_b as an (upper bound) estimate. To consider WJ-Hotspot in optimizing window join processing, we need to maintain statistics separately for hotspots and for scattered queries, in the manner described above.

5. EXPERIMENTS

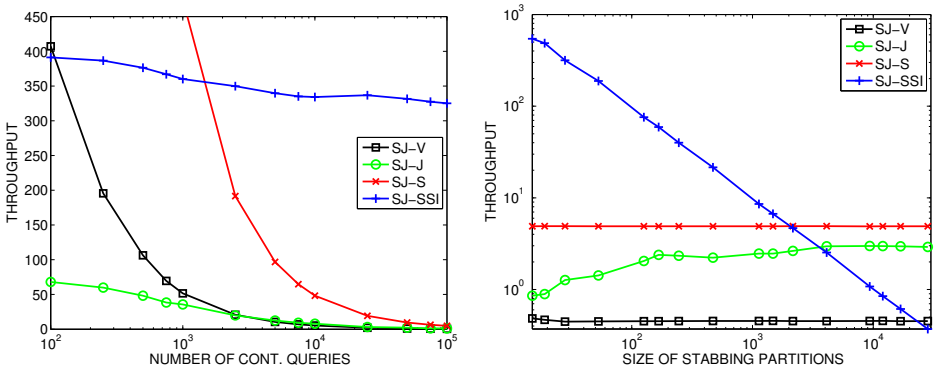
To compare our techniques against traditional processing techniques in terms of their scalability with a large number of continuous queries, we have implemented various algorithms discussed in previous sections in Java. Unless otherwise noted, all experiments were conducted on a Sun Blade 150 with a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

We generate two synthetic tables $R(A, B)$ and $S(B, C)$, where B is the join attribute, and A and 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 sequence of insertions, 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 *event join rate* (i.e., how many S tuples join with an incoming R tuple).

We create two sets of continuous queries, each with 100,000 queries initially. The first set consists of equality joins with local selections and the second set consists of band joins. For the select-joins, the midpoints of $range A_i$ follow a normal distribution, while

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 $R.A$, $S.C$	$\mathrm{Uni}(0, 1{,}000{,}000)$
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 I. Workload parameters.



(i) (ii)

Fig. 7. Throughput of equality joins with local selections (i) over number of continuous queries, and (ii) over number of stabbing groups.

the midpoints of $rangeC_i$ are uniformly distributed. For band joins, the midpoints of $rangeB_i$ 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 I 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 the degree of overlap among continuous queries, event join rate, and *event selection rate* (i.e., how many queries have their $R.A$ selection conditions satisfied by an incoming R tuple).

5.1 Processing Select-Joins

We have implemented the versions of SJ-JoinFirst (abbreviated as SJ-J in figures), SJ-SelectFirst (abbreviated as SJ-S in figures), SJ-Vanilla (abbreviated as SJ-V in figures), and SJ-Hotspot, as discussed in Section 4. We have also implemented SJ-SSI (cf. Section 3.2.2), which applies SSI-based processing to all stabbing groups (regardless of whether they are hotspots). To understand how the degree of clustering in queries affects the performance of SSI-based processing, we first focus on results comparing SJ-SSI with previous approaches; results on SJ-Hotspot will be shown afterwards.

Figure 7(i) compares the throughput of various approaches as the number of continuous

queries increases from 100 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 SJ-Vanilla's performance degrades linearly with the number of continuous queries and therefore is not scalable at all. The average event selection rate is 0.1; that is, an incoming R tuple satisfies the $R.A$ selection conditions for 10% of all continuous queries on average. Consequently, SJ-SelectFirst, which works by iterating through queries whose $R.A$ selection conditions are satisfied, performs well only when the number of queries is small. Similar to SJ-Vanilla, its performance degrades linearly with the number of queries. The performance degradation of SJ-JoinFirst 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-JoinFirst does not drop as drastically as SJ-SelectFirst and SJ-Vanilla, its throughput is less than 5% of SJ-SSI in the case of 100,000 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, SJ-Vanilla and SJ-SelectFirst are completely indifferent to the clusteredness of queries, while SJ-SSI benefits from smaller numbers of stabbing groups. SJ-SelectFirst outperforms SJ-SSI when there are roughly more than 2000 stabbing groups, as the event selection rate is roughly 250 queries in these experiments. In the worse case, when all query ranges are disjoint, SJ-SSI degenerates to SJ-Vanilla. As a side note, it is interesting that SJ-JoinFirst 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(i) shows the throughput of SJ-SelectFirst and SJ-SSI when we decrease the average event selection rate (SJ-JoinFirst and SJ-Vanilla are unaffected by this parameter). We control this rate by fine-tuning the distribution of $rangeA_i$'s. From this figure, we see that SJ-SelectFirst is very sensitive to this rate. Its throughput deteriorates linearly, since this rate directly controls how large n' is in Theorem 3). On the other hand, SJ-SSI is unaffected.

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

In previous experiments, we have only considered SJ-SSI, which applies SSI-based processing to all stabbing groups. Now, we conduct experiments to demonstrate the effectiveness of SJ-Hotspot, which selectively applies SSI-based processing to hotspots. To better control these experiments, we generate workloads differently from the previous experiments. Each workload consists of 500,000 queries, whose degree of clusteredness varies across workloads. We set $\alpha = 0.1\%$ for SJ-Hotspot; i.e., each hotspot contains at least 500

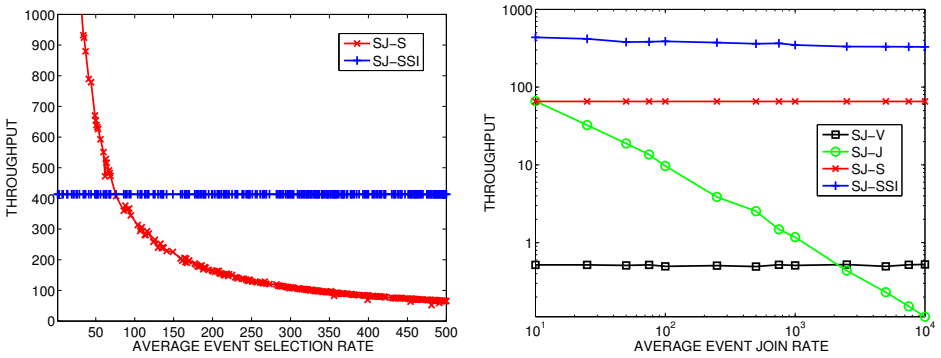


Fig. 8. Throughput of equality joins with local selections (i) over event selection rate (the number of queries whose $R.A$ selection conditions are satisfied by the incoming tuple), and (ii) over event join rate (the number of S tuples that join with the incoming tuple).

queries. We vary the fraction of queries that belong to hotspots by setting the target percentages between 10% and 100%, in increments of 10%. For each target percentage p, we generate $500{,}000 \times p$ queries by drawing the midpoints and lengths of their $rangeC_i$'s both from normal distributions, such that most of these queries belong to α-hotspots. Not all of them do; as can be seen from Figure 9, the actual percentages shown on the horizontal axis are always slightly lower than the corresponding target percentages. We then generate the remaining queries by drawing their interval midpoints uniformly from a range disjoint from the hotspots, and drawing their interval lengths from a normal distribution, such that these intervals form a large number of small groups.

Because of the larger workloads, experiments in this set were conducted on a Linux machine with a 2.13GHz Intel dual-core processor and 2GB of memory. In Figure 9, we compare three approaches: SJ-SelectFirst, SJ-SSI, and SJ-Hotspot (using SJ-SelectFirst on scattered queries). SJ-SelectFirst, unable to exploit the clusteredness among queries, behaves nearly identically across workloads. Between SJ-SelectFirst and SJ-SSI, there is no clear winner: SJ-SSI is better when queries are more clustered, but suffers from large numbers of small stabbing groups when queries are scattered. On the other hand, the performance of SSJ-Hotspot improves linearly with the increasing coverage by hotspots, as it benefits from the ability of SSI-based processing in exploiting clusteredness for efficient group processing. Moreover, by restricting SSI-based processing to hotspots, SJ-Hotspot is able to avoid going through a large number of small stabbing groups, where SJ-SSI suffers.

5.2 Processing Band Joins

In this set of experiments, we study the performance of our SSI-based approach for continuous band joins. We compare BJ-SSI with BJ-DOuter (abbreviated as BJ-D in figures), BJ-QOuter (abbreviated as BJ-Q in figures), and BJ-MJ, discussed in Section 3.1. Again, to focus our attention on the effectiveness of SSI-based processing itself, we show results of BJ-SSI 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

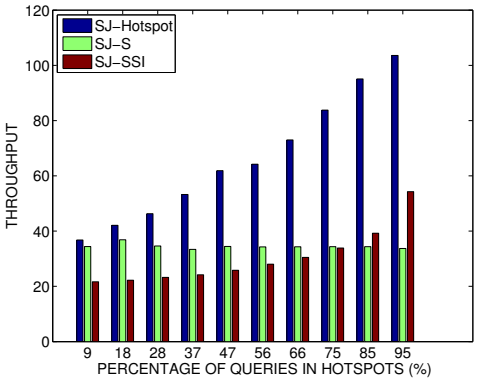


Fig. 9. Performance of SJ-Hotspot with varying degree of clusteredness.

of stabbing groups also increases from about 10 to 60 accordingly. In BJ-DOuter, for each tuple in base table S, an offset is added and used to probe the index of all band join windows. Although BJ-DOuter is not very sensitive to the number of queries, it is inefficient because its throughput decreases linearly with the size of the base table. BJ-QOuter, similar to SJ-Vanilla, 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-QOuter in this figure due to its extremely poor performance on a large number of queries. BJ-MJ and BJ-DOuter 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.

5.3 Dynamic Stabbing Partitions

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 approaches BJ-DOuter, BJ-QOuter, BJ-MJ, and BJ-SSI. Since BJ-QOuter does not maintain any index structure on the queries, its maintenance cost stays constant at 0. For BJ-DOuter, the maintenance involves updating a dynamic priority search tree that indexes all band join windows. For

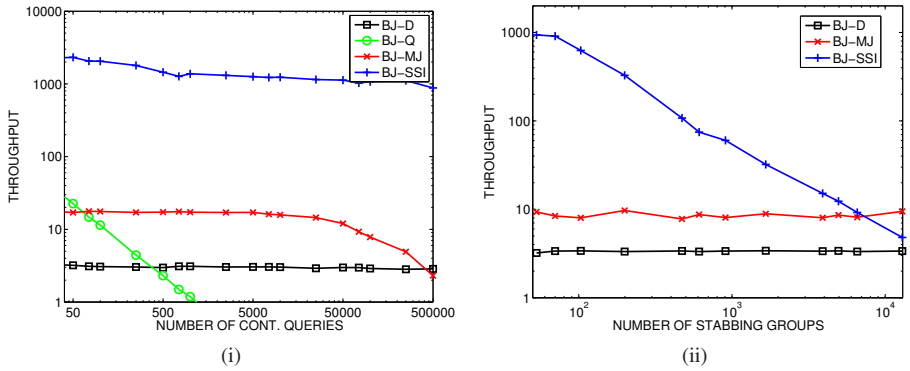


Fig. 10. Throughput of band-joins (i) over the number of continuous queries and (ii) over the number of stabbing groups.

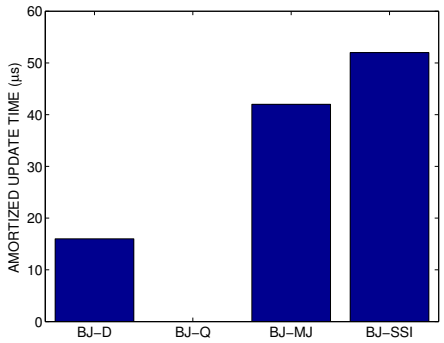


Fig. 11. Maintenance costs for various band-join approaches.

BJ-MJ, the maintenance involves updating a sorted list of band join windows. The dynamic maintenance algorithm for BJ-SSI is described in Appendix A.1. We have chosen $\varepsilon = 3$ for BJ-SSI in this experiment. 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 as shown in the previous experiments. Note that the reconstruction stage occurs fairly infrequently because all subscriptions are from the same distribution and naturally clustered; therefore, with high probability, a new subscription will be inserted into an existent stabbing group without increasing the number 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 BJ-SSI's substantial advantage in processing throughput.

5.4 Data-Sensitive Optimization

The next batch of experiments are designed to evaluate the effectiveness of our data-sensitive processing framework in Section 4, which dynamically routes each incoming tuple to a query processing strategy with the lowest estimated processing cost. The continuous queries we experiment with in this section are equality joins with local selections.

Processing strategy	Parameter	Value, 95% confidence interval
SJ-JoinFirst	α_J	$0.25, [0.24, 0.26]$
	δ_J	$4.34, [4.23, 4.44]$
SJ-SelectFirst	α_S	$0.64, [0.63, 0.65]$
	β_S	$42.49, [35.06, 49.92]$
	γ_S	$1.19, [1.16, 1.22]$
SJ-Vanilla	α_V	$0.36, [0.33, 0.38]$
	γ_V	$0.17, [0.15, 0.20]$
SJ-Hotspot	α_H	$0.19, [0.19, 0.20]$
	β_H	$70.19, [50.49, 89.89]$
	γ_H	$1.17, [1.14, 1.21]$
	δ_H	$0.08, [0.06, 0.10]$

Table II. Fitted parameters in cost models.

We refer to our data-sensitive approach as *SJ-Dynamic*. The quantities m', n', and n'' needed in evaluating cost formulas are estimated at runtime using histograms, as described in Section 4. For our experiments, we simply use standard equi-width histograms with 100 buckets, which turn out to be sufficient in making reasonably good optimization decisions. We first conduct parameter fitting for the cost models as described in Section 4. In particular, to collect training data for parameter fitting, we vary the number of queries from $100{,}000$ to $500{,}000$ (in increments of $100{,}000$), and vary the number of tuples in S from $100{,}000$ to $500{,}000$ (in increments of $100{,}000$); for each combination we generate 1000 incoming R tuples and separately measure each component cost for all processing strategies. After collecting all training data, we use standard linear regression to fit the parameters individually. Table II shows the least squares fit of all parameters and the associated 95% confidence intervals.

We compare SJ-Dynamic with four static strategies SJ-JoinFirst, SJ-SelectFirst, SJ-Vanilla, and SJ-Hotpot, where a single processing strategy is applied to all incoming tuples. In addition, we also compare SJ-Dynamic with an oracle algorithm, *SJ-Optimum*, which always chooses one of SJ-JoinFirst, SJ-SelectFirst, SJ-Vanilla, and SJ-Hotpot with the lowest actual processing cost for each incoming tuple. Note SJ-Optimum is impossible to attain in practice; we obtain the choice of SJ-Optimum only after running all four available strategies.

Figure 12 compares the throughput of different processing strategies. Labels on the horizontal axis indicate the workload size (in thousands); 100 means a workload consisting of $100{,}000$ queries and an S table with $100{,}000$ tuples. From Figure 12, we can see that among the static strategies, SJ-Hotspot performs much better than the others (by more than a factor of 3). However, it is evident that SJ-Hotspot is not always the best strategy for every incoming tuple, because its throughput is still much less than that of SJ-Optimum. For example, when there are $100{,}000$ queries and $100{,}000$ S tuples, throughput of SJ-Hotspot is only less than 50% of SJ-Optimum. SJ-Dynamic is able to achieve higher throughput than SJ-Hotspot in all workloads since it chooses the most promising processing strategy on the fly. In some cases, SJ-dynamic beats SJ-Hotspot by more than 50%.

Figure 13 shows the breakdown of runtime choices made by SJ-Dynamic. Roughly, more than 70% of the incoming tuples are processed using SJ-Hotspot; SJ-SelectFirst and SJ-JoinFirst are chosen for 25% and 5% of the tuples respectively; SJ-Vanilla is never chosen, even for the smallest workload.

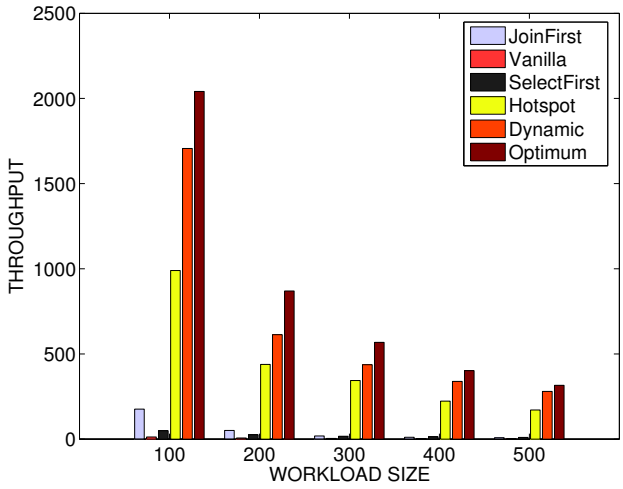


Fig. 12. Throughput comparison between data-sensitive and other processing strategies.

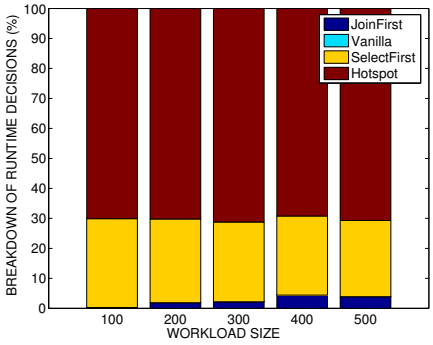


Fig. 13. Breakdown of runtime decisions made by SJ-Dynamic for workloads in Figure 12.

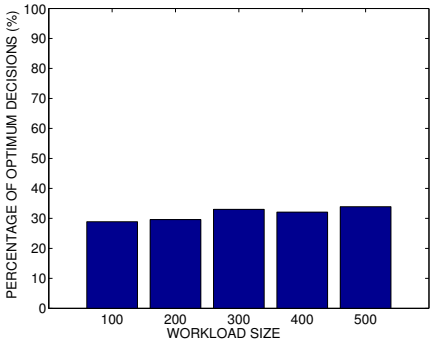


Fig. 14. Percentage of optimum decisions made by SJ-Dynamic for workloads in Figure 12.

As we have seen in Figure 12, SJ-Dynamic still underperforms SJ-Optimum, meaning that that SJ-Dynamic does not always make the optimum decision. Figure 14 shows how often SJ-Dynamic actually makes the optimum decision (using SJ-Optimum as a reference). We can see that SJ-Dynamic only makes optimum decisions 30% of the time, which may seem rather dismal at a first glance. However, recall that as in data query optimization, our goal is not so much making optimum decisions as avoiding bad ones. Finding the optimum processing strategy may be very costly, which negates its marginal advantage over a good strategy that is easier to find. SJ-Dynamic made a conscious tradeoff in employing fairly simple cost models and crude histograms in order to keep the optimization overhead low. The fairly narrow performance gap between SJ-Dynamic and SJ-Optimum shown in Figure 12 implies that tradeoff is acceptable. Although SJ-Dynamic fails to make optimum decisions 70% of the time, it still manages to make reasonably good decisions that maintain good overall efficiency.

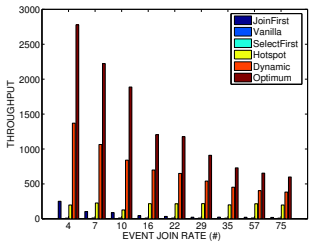


Fig. 15. Throughput comparison between data-sensitive and other processing strategies; varying event join rate.

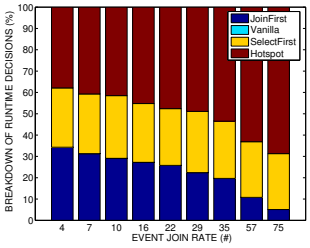


Fig. 16. Breakdown of runtime decisions made by SJ-Dynamic for workloads in Figure 15.

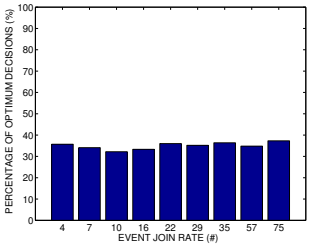


Fig. 17. Percentage of optimum decisions made by SJ-Dynamic for workloads in Figure 15.

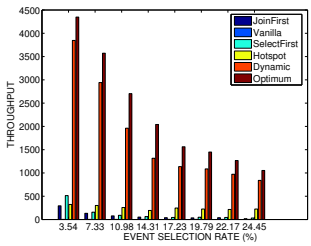


Fig. 18. Throughput comparison between data-sensitive and other processing strategies; varying event selection rate.

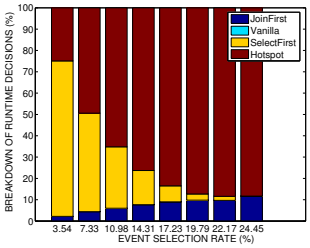


Fig. 19. Breakdown of runtime decisions made by SJ-Dynamic for workloads in Figure 18.

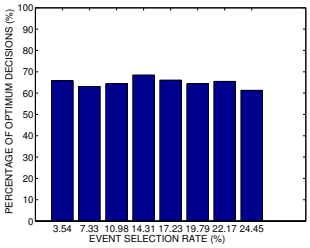


Fig. 20. Percentage of optimum decisions made by SJ-Dynamic for workloads in Figure 18.

The next experiment studies how event join rate affects the overall performance of SJ-Dynamic and its runtime decisions. Given $500{,}000$ queries and $500{,}000$ S tuples, we adjust the distribution of the join attribute value of the incoming tuple to control event join rate (shown in terms of the average number of S tuples that join with an incoming tuple). Figure 15 shows throughput, Figure 16 shows the breakdown of SJ-Dynamic's runtime decisions, and Figure 17 shows how often SJ-Dynamic makes the optimum decision. The performance of SJ-Dynamic relative to other strategies seen in Figure 15 is similar to Figure 12. Since we are only increasing the event join rate in this experiment, throughput decreases for SJ-JoinFirst, but remains stable for SJ-Vanilla, SJ-SelectFirst, and SJ-Hotspot. Hence, from Figure 16, we see that SJ-JoinFirst gradually loses its share to SJ-Hotspot. In Figure 17, only 35% of the runtime decisions made by SJ-Dynamic are optimum, but as explained earlier, this low ratio does not prevent SJ-Dynamic from being effective.

Finally, in the last experiment, we study how event selection rate affects SJ-Dynamic. We again fix the number of queries and the number of S tuples both at $500{,}000$, and we adjust the distribution of $R.A$ values for the incoming tuple to control the event selection rate (shown in terms of the average percentage of queries with their $R.A$ local selections satisfied by the incoming tuple). Figure 18 shows throughput, Figure 19 shows the breakdown of SJ-Dynamic's runtime decisions, and Figure 20 shows how often SJ-Dynamic makes the optimum decision. We see that when the event selection rate is low, SJ-SelectFirst performs well, and is chosen by SJ-Dynamic very often. However, as the event selection

rate increases, SJ-SelectFirst becomes increasingly unfavored by SJ-Dynamic, which intuitively makes sense. We also see that SJ-Dynamic often does not make the optimum decision; nevertheless, SJ-Dynamic still dramatically outperforms any of the static processing strategies.

6. RELATED WORK

As mentioned in Section 1, scalable continuous query processing plays a pivotal role in many applications (e.g., [Widom and Ceri 1996; Hanson et al. 1999; Carney et al. 2002; Special 2003]). For example, publish/subscribe systems [Liu et al. 1999; Chen et al. 2000; Pereira et al. 2001; Dittrich et al. 2005; Demers et al. 2006] by definition need to handle a huge number of subscriptions (continuous queries) efficiently.

Many continuous query and stream processing systems have been proposed (e.g., [Chen et al. 2000; Madden et al. 2002; Chandrasekaran and Franklin 2003; Special 2003]). *NiagaraCQ* [Chen et al. 2000] 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. Our work is able to overcome these limitations. *CACQ* [Madden et al. 2002] is a continuous query engine that leverages *Eddies* [Avnur and Hellerstein 2000] 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* [Chandrasekaran and Franklin 2003] treats data and queries analogously, thereby making it possible to exploit set-oriented processing on a 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. Like PSoup, [Lim et al. 2006] also recognizes the duality of data and queries, and applies spatial join techniques to the problem of processing continuous queries. While the proposed algorithm has additional features such as support for batching and sliding windows, it essentially adopts a strategy similar to SJ-JoinFirst; our approach can offer higher efficiency by exploiting data and query characteristics.

Cayuga [Demers et al. 2006] considers scalable processing of subscriptions expressed in an event algebra, with constructs analogous to joins. Cayuga uses indexes extensively for efficient group processing, but they are essentially predicate indexes. Although Cayuga's automata-based processing is not directly comparable to our work because of different data models and query semantics, on a high level it resembles SJ-SelectFirst. Related to Cayuga, there is a large body work on scalable XML event processing (e.g., *XML Toolkit* [Green et al. 2004], *YFilter* [Diao et al. 2003], and *XSCL* [Hong et al. 2007]), some of which also employed automata-based processing and query indexing.

[Agarwal et al. 2005] 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. [Agarwal et al. 2006] presented data structures and algorithms for query-sensitive processing of continuous select-joins and band joins, as well as histogram construction for intervals. This paper additionally studies

window joins, proposes the data-sensitive processing framework, and presents experiments demonstrating the power of combining data-sensitive, and query-sensitive processing in a unified input-sensitive approach. This also contains more details, including full proofs and the refined algorithm for maintaining stabbing partitions (Appendix A).

Related to data-sensitive processing, there has been extensive work on adaptive query optimization (e.g., [Avnur and Hellerstein 2000; Markl et al. 2004; Bizarro et al. 2005]). With the exception of *CBR* [Bizarro et al. 2005], most previous approaches use a single plan for all almost all tuples at a given time. Some systems, such as Eddies [Avnur and Hellerstein 2000], provides mechanisms for adapting the plan on an individual tuple basis, but their policies typically do not result in plans that changes for every incoming tuple. In CBR [Bizarro et al. 2005], each operator is profiled and learning techniques are used to help choose query plan dynamically for each incoming tuple. In contrast, our data-sensitive processing framework adopts a more conventional optimization approach using more standard cost estimation and statistics collection techniques.

7. CONCLUSION AND FUTURE WORK

In this paper, we have presented a novel, input-sensitive approach to scalable continuous query processing. The input sensitivity of our approach manifests in two ways. First, we have developed query-sensitive data structures and algorithms that exploit the clustering pattern in continuous queries in a principled manner for efficient group processing. Second, we have developed a data-sensitive processing framework that makes runtime, cost-based decisions on how to process each incoming tuple based on its characteristics. Together, query-sensitive and data-sensitive techniques bring dramatic performance improvements over existing methods for scalable continuous query processing.

Our work opens the door to many directions of future work. First, it would be interesting to extend the idea of clustering by stabbing partition to multidimensional spaces, so that we can better 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, this paper assumes that we process one incoming tuple at at time. Batching has been shown to be very effective in publish/subscribe and continuous query systems [Fischer and Kossmann 2005; He et al. 2005; Lim et al. 2006]. We plan to extend our approach to take advantage of batching when possible. How to combine group processing of queries and batch processing of tuples together effectively is an intriguing new challenge.

ACKNOWLEDGMENTS

Removed to ensure anonymity.

REFERENCES

AGARWAL, P. K., ARGE, L., BRODAL, G. S., AND VITTER, J. S. 1999. I/O-efficient dynamic point location in monotone planar subdivisions. In *Proc. of the 1999 ACM-SIAM Symposium on Discrete Algorithms*. 11–20.

AGARWAL, P. K., XIE, J., YANG, J., AND YU, H. 2005. Monitoring continuous band-join queries over dynamic data. In *Proc. of the 16th Intl. Sympos. Algorithms and Computation*. Spring-Verlag, Hanan, China, 349–359.

AGARWAL, P. K., XIE, J., YANG, J., AND YU, H. 2006. Scalable continuous query processing by tracking hotspots. In *Proc. of the 2006 Intl. Conf. on Very Large Data Bases*. Seoul, Korea, 31–42.

ARGE, L. AND VITTER, J. 2003. Optimal external memory interval management. *SIAM J. Comput. 32,* 6, 1488–1508.

AVNUR, R. AND HELLERSTEIN, J. M. 2000. Eddies: Continuously adaptive query processing. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*. ACM, Dallas, Texas, USA, 261–272.

BABU, S., BIZARRO, P., AND DEWITT, D. 2005. Proactive re-optimization. In *Proc. of the 2005 ACM SIGMOD Intl. Conf. on Management of Data*. ACM, Baltimore, Maryland, USA.

BIZARRO, P., BABU, S., DEWITT, D., AND WIDOM, J. 2005. Robust query processing through progressive optimization. In *Proc. of the 2005 Intl. Conf. on Very Large Data Bases*. VLDB, Trondheim, Norway.

CARNEY, D., ÇETINTEMEL, U., CHERNIACK, M., CONVEY, C., LEE, S., SEIDMAN, G., STONEBRAKER, M., TATBUL, N., AND ZDONIK, S. B. 2002. Monitoring streams - a new class of data management applications. In *Proc. of the 2002 Intl. Conf. on Very Large Data Bases*. VLDB, Hongkong, China, 215–226.

CHANDRASEKARAN, S. AND FRANKLIN, M. J. 2003. Psoup: a system for streaming queries over streaming data. *VLDB Journal 12,* 2, 140–156.

CHEN, J., DEWITT, D. J., TIAN, F., AND WANG, Y. 2000. NiagraCQ: A scalable continuous query system for internet databases. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*. ACM, Dallas, Texas, USA, 379–390.

DE BERG, M., VAN KREVELD, M., OVERMARS, M., AND SCHWARZKOPF, O. 2000. *Computational Geometry: Algorithms and Applications*, 2nd ed. Springer, New York, USA.

DEMERS, A. J., GEHRKE, J., HONG, M., RIEDEWALD, M., AND WHITE, W. M. 2006. Towards expressive publish/subscribe systems. In *Proc. of the 2006 Intl. Conf. on Extending Database Technology*. Munich, Germany, 627–644.

DEWITT, D. J., NAUGHTON, J. F., AND SCHNEIDER, D. A. 1991. An evaluation of non-equijoin algorithms. In *Proc. of the 1991 Intl. Conf. on Very Large Data Bases*. VLDB, Barcelona, Catalonia, Spain, 443–452.

DIAO, Y., ALTINEL, M., FRANKLIN, M. J., ZHANG, H., AND FISCHER, P. 2003. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Trans. on Database Systems 28,* 4, 467–516.

DITTRICH, J.-P., FISCHER, P. M., AND KOSSMANN, D. 2005. Agile: adaptive indexing for context-aware information filters. In *Proc. of the 2005 ACM SIGMOD Intl. Conf. on Management of Data*. ACM, Baltimore, Maryland, USA, 215–226.

DU, Q., FABER, V., AND GUNZBURGER, M. 1999. Centroidal Voronoi tessellations: Applications and algorithms. *SIAM Reviews 41*, 637–676.

FISCHER, P. M. AND KOSSMANN, D. 2005. Batched processing for information filters. In *Proc. of the 2005 Intl. Conf. on Data Engineering*. Tokyo, Japan.

GEHRKE, J. AND HELLERSTEIN, J. M. 2004. Guest editorial to the special issue on data stream processing. *VLDB J. 13,* 4, 317.

GREEN, T. J., GUPTA, A., MIKLAU, G., ONIZUKA, M., AND SUCIU, D. 2004. Processing XML streams with deterministic automata and stream indexes. *ACM Trans. on Database Systems 29,* 4, 752–788.

HANSON, E. AND JOHNSON, T. 1991. The interval skip list: A data structure for finding all intervals that overlap a point. In *Proc. of the 1991 Workshop on Algorithms and Data Structures*. Springer, Ottawa, Canada, 153–164.

HANSON, E. N., CARNES, C., HUANG, L., KONYALA, M., NORONHA, L., PARTHASARATHY, S., PARK, J. B., AND VERNON, A. 1999. Scalable trigger processing. In *Proc. of the 1999 Intl. Conf. on Data Engineering*. IEEE Computer Society Press, Sydney, Austrialia, 266–275.

HAR-PELED, S. AND MAZUMDAR, S. 2004. Coresets for k-means and k-median clustering and their applications. In *Proc. of the 2004 Annu. Sympos. Theory of Computing*. ACM, Chicago, Illinois, USA, 291–300.

HE, H., XIE, J., YANG, J., AND YU, H. 2005. Asymmetric batch incremental view maintenance. In *Proc. of the 2005 Intl. Conf. on Data Engineering*. Tokyo, Japan.

HONG, M., DEMERS, A. J., GEHRKE, J., KOCH, C., RIEDEWALD, M., AND WHITE, W. M. 2007. Massively multi-query join processing in publish/subscribe systems. In *Proc. of the 2007 ACM SIGMOD Intl. Conf. on Management of Data*. Beijing, China, 761–772.

IVES, Z., FLORESCU, D., FRIEDMAN, M., LEVY, A., AND WELD, D. 1999. An adaptive query execution system for data integration. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*. ACM, Philadelphia, Pennsylvania, USA.

JAGADISH, H. V., KOUDAS, N., MUTHUKRISHNAN, S., POOSALA, V., SEVCIK, K., AND SUEL, T. 1998. Optimal histograms with quality guarantees. In *Proc. of the 1998 Intl. Conf. on Very Large Data Bases*. VLDB, New York City, USA, 275–286.

KATZ, M., NIELSEN, F., AND SEGAL, M. 2003. Maintenance of a piercing set for intervals with applications. *Algorithmica 36(1)*, 59–73.

KOUDAS, N., MUTHUKRISHNAN, S., AND SRIVASTAVA, D. 2000. Optimal histograms for hierarchical range queries. In *Proc. of the 2000 ACM Sympos. on Principles of Database Systems*. ACM, Dallas, Texas, USA, 196–204.

LIM, H.-S., LEE, J.-G., LEE, M.-J., WHANG, K.-Y., AND SONG, I.-Y. 2006. Continuous query processing in data streams using duality of data and queries. In *Proc. of the 2006 ACM SIGMOD Intl. Conf. on Management of Data*. Chicago, Illinois, USA, 313–324.

LIU, L., PU, C., AND TANG, W. 1999. Continual queries for Internet scale event-driven information delivery. *IEEE Trans. on Knowledge and Data Engineering 11,* 4, 610–628.

MADDEN, S., SHAH, M., HELLERSTEIN, J., AND RAMAN, V. 2002. Continuously adaptive continuous queries over streams. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*. ACM, Madison, Wisconsin, USA, 49–60.

MARKL, V., RAMAN, V., SIMMEN, D., LOHMAN, G., AND PIRAHESH, H. 2004. Robust query processing through progressive optimization. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*. Paris, France.

MCCREIGHT, E. M. 1985. Priority search trees. *SIAM J. Comput. 14*, 257–276.

PEREIRA, J., FABRET, F., JACOBSEN, H. A., LLIRBAT, F., AND SHASHA, D. 2001. Webfilter: A high-throughput XML-based publish and subscribe system. In *Proc. of the 2001 Intl. Conf. on Very Large Data Bases*. VLDB, Rome, Italy, 723–724.

Special 2003. Special issue on data stream processing. *IEEE Data Eng. Bull. 26,* 1.

TARJAN, R. E. 1983. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, USA.

VAISHNAVI, V. K. 1982. Computing point enclosures. *IEEE Trans. Computers 31,* 1, 22–29.

WIDOM, J. AND CERI, S. 1996. *Active Database Systems: Triggers and Rules For Advanced Database Processing.* Morgan Kaufmann, San Fransisco, California, USA.

A. DYNAMIC STABBING PARTITIONS

This section is devoted to an efficient implementation of Lemma 2. Because it is not a prerequisite for understanding other parts 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.1 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. 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 $(1 + \varepsilon)\tau(I)$. □

The above strategy can be improved in several ways. 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 would 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.2 A Refined Algorithm

The amortized cost per insertion and deletion in the simple strategy above is $O(n \log n/(\varepsilon\tau_0))$. Katz et al. [Katz et al. 2003] 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 $\mathcal{I} = \{I_i \mid 1 \le i \le \tau_0\}$ 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 [Tarjan 1983]. 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 $\mathcal{I}$.[6] 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 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 $O(\tau_0 \log n)$. Complete pseudo-code of the reconstruction stage is listed in Figure 21.

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

[6] Again, this can be improved as in the simple 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.

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 $J \in \mathcal{I}$.

```
1:  P, I ← ∅; U, V, T_U ← ∅;
    {The collection of all intervals in the sets of U or V is denoted by J}
2:  while ∃ unprocessed nonempty groups do
3:    K ← next unprocessed nonempty group;
4:    Mark K processed;
5:    if l(K) ≤ r(⋂J) then
6:      if K = I_γ for some γ ∈ A then
7:        V ← V ∪ {γ};
8:      else {K = I_i for some 1 ≤ i ≤ τ_0}
9:        I'_i, I_i ← SPLIT(I_i, r(⋂J));
10:       T'_i, T_i ← SPLIT(T_i, r(⋂J));
11:       U ← U ∪ {I'_i}, T_U ← JOIN(T_U, T'_i);
12:       Re-mark I_i unprocessed if I_i ≠ ∅;
13:   else {l(K) > r(⋂J)}
14:     if K = I_γ for some γ ∈ A then
15:       I_i ← next unprocessed nonempty group in {I_j | 1 ≤ j ≤ τ_0};
16:       I'_i, I_i ← SPLIT(I_i, r(⋂J));
17:       T'_i, T_i ← SPLIT(T_i, r(⋂J));
18:       U ← U ∪ {I'_i}, T_U ← JOIN(T_U, T'_i);
19:     T_J ← T_U;
20:     for every γ ∈ V do
21:       INSERT(T_J, γ);
22:     I ← I ∪ {J}, P ← P ∪ {r(⋂J)};
23:     if K = I_γ for some γ ∈ A then
24:       U, T_U ← ∅, V ← {γ};
25:     else {K = I_i for some 1 ≤ i ≤ τ_0}
26:       U ← {I_i}, T_U ← T_i, V ← ∅;
```

Fig. 21. Pseudo-code for the reconstruction stage.

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 22.

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 $\{I_i \mid 1 \leq i \leq \tau_0\}$ (Figure 23 (a)); otherwise we set $L = L_{u+1}$ (Figure 23 (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

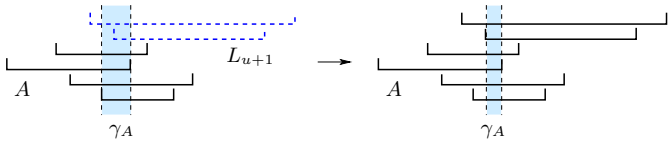


Fig. 22. Case 1: $\gamma_A \cap \gamma_{u+1} \neq \emptyset$. Intervals in A are solid, and intervals in L_{u+1} are dashed. We add L_{u+1} to A.

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 $\bigcup_{i \geq u+1} L_i$ 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}.

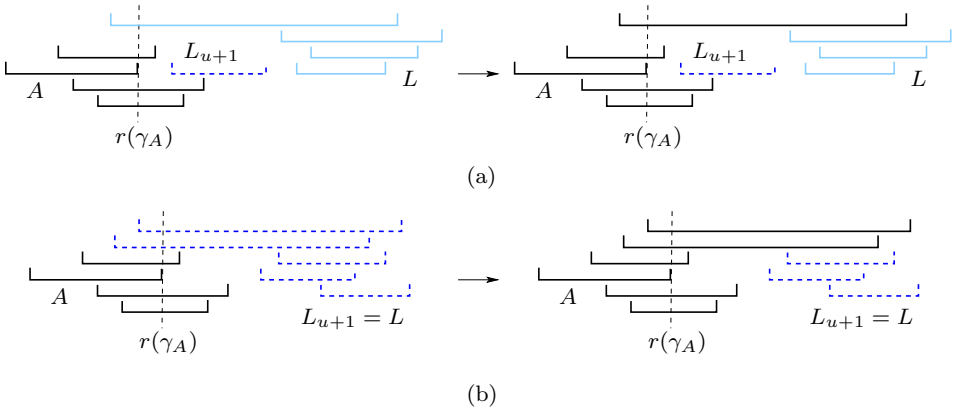


Fig. 23. Case 2: $\gamma_A \cap \gamma_{u+1} = \emptyset$. Intervals in A are solid, and intervals in L_{u+1} are dashed. $r(\gamma_A)$ denotes the right endpoint of γ_A. (a) L is set to be the leftmost unprocessed group in $\{I_i \mid 1 \leq i \leq \tau_0\}$; (b) $L = L_{u+1}$.

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) (but in a batched manner), and thus returns the same optimal stabbing partition as the greedy algorithm does.

THEOREM 5. *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 $O((1+1/\varepsilon)\log n)$. Before the reconstruction stage, each insertion or deletion affects at most one group in the stabbing partition.*

B. 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 high in practice [Jagadish et al. 1998; Koudas et al. 2000]. In contrast, our new algorithm below is simple, runs in nearly linear time, and often provides a high-quality histogram. To be fair, the dynamic-programming approaches usually guarantee optimal solutions (i.e., those that minimize the error), while the histogram returned by our algorithm does not. Nonetheless, since histograms are primarily for estimation purposes, optimal histograms are often unnecessary in practice.

Our Approach Our new approach differs radically from the 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 group $I_i \in \mathcal{I}$, 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 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 24); 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 increas-

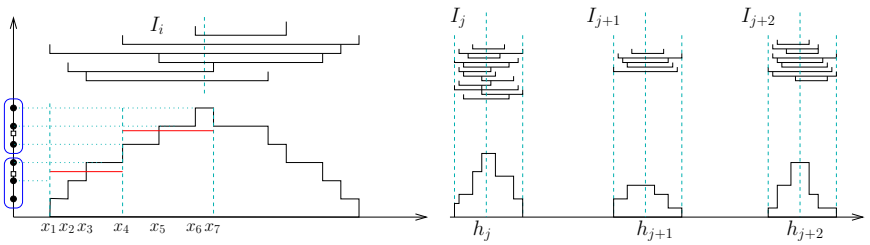


Fig. 24. Reducing to a one-dimensional weighted k-means clustering problem.

ing 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_i^l, suppose its bucket boundaries divide the break points into k groups:

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

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 $0 \le j < k$. 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 24). 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.*

PROOF. The error defined in (1) can be interpreted as the cost of the weighted k-means clustering by choosing the centers to be $c_1, \cdots, c_k$ and assigning $\{y_{z_j+1}, \cdots, y_{z_{j+1}}\}$ to each c_j. On the other hand, given a solution to the clustering problem, one can also easily map it back to a histogram whose mean-squared relative error is the same as the cost of the clustering, using the monotonic property of $f_{I_i}^l$. We omit the straightforward details for brevity. □

Since typically the total number 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 get around this problem completely 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 24. Then we can run the k-means algorithm of [Har-Peled and Mazumdar 2004] 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

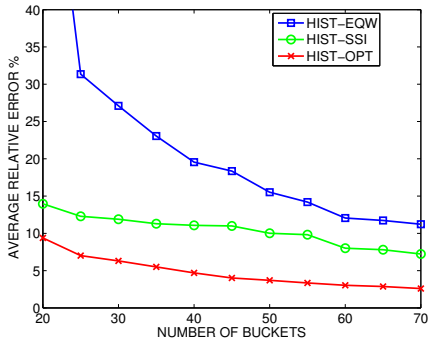


Fig. 25. Quality of various histograms for intervals.

an appropriate number of buckets to each I_i. In practice, one may wish to use the simpler iterative k-means clustering algorithm [Du et al. 1999] 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.

Finally, we note that our histogram construction method works best when the number of stabbing groups is small. If there are a lot of stabbing groups (compared with the number of intervals), our method does not seem to offer better performance than existing techniques. As future work, it would be interesting to investigate how to apply the idea of hotspots to this problem.

Experimental Evaluation We compare *HIST-SSI*, the histogram for intervals constructed by our algorithm based on stabbing partition discussed above, with *HIST-EQW*, the standard equi-width histogram, and *HIST-OPT*, the optimal histogram constructed using dynamic programming. We generate 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 HIST-SSI using the k-mean algorithm and the heuristics of assigning the number of buckets to each stabbing group based on its cardinality, as described earlier in this section. Construction of HIST-SSI completes within one minute. However, construction of HIST-OPT using dynamic programming for 100,000 intervals has proved to be unacceptably slow on our computing platform. Instead, we build HIST-OPT on just a sample of 10,000 intervals and run experiments multiple times until a stable estimation is reached. Even with one-tenth of the original data, HIST-OPT takes roughly 6.5 hours on a computer with 3GHz processor and 2GB memory, in sharp contrast to the ease of constructing HIST-SSI.

Figure 25 compares the quality of HIST-SSI, HIST-EQW, and HIST-OPT, as we increase the size of the histogram from 20 to 70 buckets; we limit the scale to 70 buckets because construction of HIST-OPT becomes too costly. Each data point is obtained by running 5000 uniformly distributed stabbing queries; we compute the relative error between true and estimated result sizes, and then report the average of these errors. As expected, HIST-OPT consistently wins; however, this advantage is greatly offset by its impracticality in terms of construction cost. On the other hand, HIST-SSI outperforms HIST-EQW all the time and dramatically reduces the gap between HIST-EQW and HIST-OPT. Specifically,

given only 20 buckets, HIST-SSI achieves an error rate as small as 14.9%, while that of HIST-EQW is more than 70%. In fact, HIST-EQW would require 50 buckets to reach the same error rate as that of HIST-SSI with 20 buckets.