Continuous Query Processing in Data Streams Using Duality of Data and Queries

Hyo-Sang Lim, Jae-Gil Lee, Min-Jae Lee, Kyu-Young Whang
Department of Computer Science and
Advanced Information Technology Research Center (AITrc)
Korea Advanced Institute of Science and Technology (KAIST)
Daejeon, Korea
{hslim,jglee,mjlee,kywhang}@mozart.kaist.ac.kr

Il-Yeol Song
College of Information Science and Technology
Drexel University
Philadelphia, Pennsylvania 19104, USA
song@drexel.edu

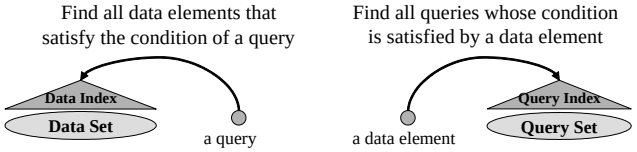
ABSTRACT

Recent data stream systems such as TelegraphCQ have employed the well-known property of duality between data and queries. In these systems, query processing methods are classified into two dual categories – *data-initiative* and *query-initiative* – depending on whether query processing is initiated by selecting a data element or a query. Although the duality property has been widely recognized, previous data stream systems do not fully take advantages of this property since they use the two dual methods independently: data-initiative methods only for continuous queries and query-initiative methods only for ad-hoc queries. We contend that continuous query processing can be better optimized by adopting an approach that integrates the two dual methods. Our primary contribution is based on the observation that spatial join is a powerful tool for achieving this objective. In this paper, we first present a new viewpoint of transforming the continuous query processing problem to a multi-dimensional spatial join problem. We then present a continuous query processing algorithm based on spatial join, which we name *Spatial Join CQ*. This algorithm processes continuous queries by finding the pairs of overlapping regions from a set of data elements and a set of queries, both defined as regions in the multi-dimensional space. The algorithm achieves the advantages of the two dual methods simultaneously. Experimental results show that the proposed algorithm outperforms earlier algorithms by up to 36 times for simple selection continuous queries and by up to 7 times for sliding window join queries.

1. INTRODUCTION

A data stream is a sequence of data elements that arrive orderly[1, 9, 16]. Examples of data streams include sensor data or network packet data. The primary characteristics of data streams are that they arrive continuously, rapidly, unboundedly, and in real-time[9, 16]. Due to these characteristics, it is not possible to control the order in which elements arrive nor to store the entire data elements[16]. Hence, queries over data streams are not one-time queries, which are executed only once against stored data. Instead, queries over data streams are *continuous queries* that are registered in advance and run repeatedly over a period of time[16, 20].

There has been a lot of research effort on efficiently managing stream data and processing continuous queries; many data stream systems such as NiagaraCQ[6], TelegraphCQ[5], Aurora[24] and STREAM[1] have been developed. Especially, PSoup[4], the query processor of TelegraphCQ, has employed the well-known property of *duality between data and queries* to process ah-hoc queries and continuous queries. Ad-hoc queries are processed by finding a set of data elements that satisfy the condition of a query (Figure 1(a)). In contrast, continuous queries are processed by finding a set of continuous queries whose condition is satisfied by a data element (Figure 1(b)). We call these two dual methods *query-initiative* and *data-initiative*, respectively.



(a) Query-initiative method. (b) Data-initiative method.

Figure 1: Two dual methods for query processing.

Although the duality property has been widely recognized, previous data stream systems do not fully take advantages of this property since they use the two dual methods independently: data-initiative methods only for continuous queries and query-initiative methods only for ad-hoc queries. Thus, when processing continuous queries, the performance gain that could be obtained by query-initiative methods has been overlooked.

We contend that continuous query processing can be better optimized by adopting an approach that integrates these two dual methods. As shown in Figure 2, our method finds all pairs of a query and a data element where the data element satisfies the condition of the query. That is, our method al-

SIGMOD 2006, June 27–29, 2006, Chicago, Illinois, USA.

lows us to process *multiple* continuous queries against *multiple* data elements. Compared with data-initiative methods, our method has more opportunities for optimization on the data set as well. In particular, it can share query executions against individual data elements, significantly improving the overall performance of query processing against multiple data elements. Thus, the advantages of our method become more prominent when a large number of data elements have to be handled due to a high input rate.

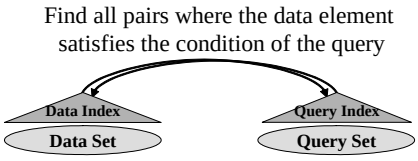


Figure 2: Our method that integrates the two dual methods.

Our major contribution is based on the observation that spatial join is a powerful tool for combining the two dual methods. Spatial join is a spatial database operation that finds all pairs of objects satisfying a given spatial relationship(i.e., overlap as in this paper)[3]. We find out and exploit the similarities between our method and spatial join.

We first present a new viewpoint of transforming the continuous query processing problem to a multi-dimensional spatial join problem. For the transformation, we present a formalization of duality between data and queries, which we call the *duality model*. The duality model represents both a data element and a query as regions in the multi-dimensional space and defines query processing as a process of finding the pairs of overlapping regions from a set of data elements and a set of queries.

We then propose the *continuous query processing algorithm based on spatial join*, which we call *Spatial Join CQ*. The algorithm achieves the effects of both of the two dual methods by using spatial join, which is a symmetric operation. To show progressive development of the algorithm, we first develop the algorithm for simple selection continuous queries, and then, extend it to accommodate sliding window join continuous queries. We demonstrate excellence of the algorithm by extensive experiments. As stated earlier, our algorithm becomes more effective – especially for batch processing, where we can take advantage of optimization on the data set as well.

The rest of this paper is organized as follows. Section 2 introduces the forms of data streams and continuous queries that are dealt with in the paper. Section 3 reviews prior work on continuous query processing methods. Section 4 formalizes the duality model. Section 5 presents the continuous query processing algorithm based on spatial join. Section 6 extends the algorithm so as to accommodate sliding window join continuous queries. Section 7 summarizes the results of performance evaluation. Section 8 concludes the paper.

2. PRELIMINARIES

In this section, we discuss the forms of data streams, continuous queries, and the continuous query processing methods that we handle in the paper.

The Form of Data Streams

Most of existing data stream systems such as TelegraphCQ[5], Aurora[24], and STREAM[1] handle the relational tuples having a fixed schema for each data stream source. We handle the same form of input data streams.

For simplicity of explanation, we assume that the domain of all the attributes is a real number between 0 and 1. Here, 0 and 1 mean the minimum and maximum values of the domain, respectively. We make this assumption without loss of generality because any value can be transformed to a real number using a hash function. For example, in the case of a string value, the order-preserving hash functions proposed by Fox et al.[8] can be exploited.

A data element in a data stream has a timestamp, meaning the time when it arrived, in addition to attributes. It can be used to identify the order of data elements and to expire old data elements.

The Form of Continuous Queries

Most of existing data stream systems[1, 6, 24] handle Select-Project-Join (SPJ) queries against relational tuples.[1] We handle the same form of continuous queries, which is shown in Equation (1).

$$\Pi_{\text{projection\_attributes}}\ (\sigma_{\text{selection\_predicate}}(\text{D})) \qquad (1)$$

Here, D stands for the set of data stream sources. The *selection_predicate* represents the condition of a query. The condition is represented as a disjunction of conjunctions, i.e., in the Disjunctive Normal Form(DNF). We note that any query condition can be translated to a DNF. For a single data stream source, this condition consists of a boolean expression of simple predicates involving the operators $>$, $\geq$, $=$, $<$, and $\leq$. For multiple data stream sources, this condition contains a join condition including an equi-join or a theta-join. The *projection_attributes* represents the list of the attributes returned as the query result.

For continuous queries, the concept of the *sliding window*[1, 9, 16] has been presented to restrict the range of data elements to be processed. This concept becomes necessary due to unboundedness of data streams. The sliding window is categorized into the time-based sliding window and the count-based sliding window. The former consists of the data elements that arrived within the given w time unit. The latter consists of the w data elements that most recently arrived.

Strategies for Continuous Query Processing

There are two strategies for continuous query processing: immediate processing and batch processing. The former processes continuous queries immediately after a data element arrives. The latter keeps data elements for some duration and processes continuous queries against the data elements kept. We support both strategies in this paper.

[1]In data streams, aggregation functions are usually processed by finding approximate results because of unboundedness of the input data[1, 9]. It can be done in a post-processing step after the SPJ queries are processed, and we will leave this issue as a further study.

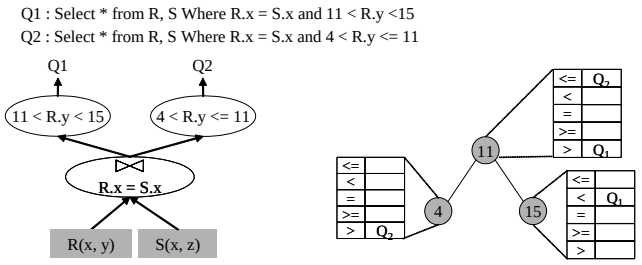
Immediate processing is hard to use in an environment where data elements are arriving at a very fast rate. In contrast, batch processing is adequate for such an environment since it can improve the performance by sharing query executions against individual data elements. We note that sliding window join can take advantages of batch processing since it is processed against sets of data elements contained in sliding windows. Batch processing, however, is not suitable for real-time monitoring because of the delay in query processing.

3. RELATED WORK

To process continuous queries, existing data stream systems such as NiagaraCQ[6], TelegraphCQ[5], Aurora[24], and STREAM[1] adopt the data-initiative approach. That is, these systems find continuous queries whose conditions are satisfied by the data element that newly arrives. Hence, in these systems, optimization is allowed only for the query set, but not for the data set. Accordingly, the differences between TelegraphCQ and our method become remarkable when we process continuous queries over a large number of data elements.

Golab et al.[9] classifies optimization techniques employed in existing systems into query plan sharing[5, 6] and query predicate indexing[6].

Query plan sharing identifies the common operations among continuous queries registered and executes these common operations only once. Hence, repeated execution can be avoided. This method is mainly used to share join, which is a very costly operation. Figure 3(a) shows an example of query plan sharing between the queries Q1 and Q2. Since the queries Q1 and Q2 have a common join operation R.x = S.y, they can share it to speed up continuous query processing.



(a) Query plan sharing between the queries Q1 and Q2.

(b) Query predicate indexing on the attribute R.y.

Figure 3: Query optimization techniques for the data-initiative method.

Query predicate indexing[6] allows us to efficiently find continuous queries whose conditions are satisfied by a given data element. This can be done by indexing the ranges of an attribute value that makes a query predicate true. NiagaraCQ[6] and TelegraphCQ[5] use a predicate index that is similar to the IBS tree[10] for this purpose. The IBS tree is a balanced binary search tree, which is created for each attribute. Its node stores a constant value appearing in the predicates as well as the identifiers of the predicates having that constant value. These identifers are categorized by the operators (i.e., $>$, $\geq$, $=$, $<$, and $\leq$) used in the predicates. Figure 3(b) shows an example of predicate indexing on the attribute R.y of the queries Q1 and Q2. To find the predicates that become true, we traverse the IBS tree using the attribute value of a given data element. Suppose a data element whose value for the attribute R.y is 14. In the node 11, the query Q1 is selected because 14 is grater than 11. In the node 15, the query Q1 is selected because 14 is less than 15. Since Q1 is selected in every node that stores it, Q1's condition on R.y becomes true for R.y = 14.

This method has a weakness of having to retrieve the result for each IBS tree and merge those results since an IBS tree is created for each attribute. Besides, it cannot index the predicates, such as join, that involve multiple attributes.

On the other hand, a method similar to *Spatial Join CQ* has been independently developed in another area. SINA [15], proposed in the area of spatial databases, has some similarities to our method, i.e., it uses spatial join to process continuous queries over moving objects. Our distinct contribution is proposing transformation of continuous query processing, which is not originally a spatial database operation, into spatial join. Furthermore, SINA deals primarily with simple range predicates, but does not deal with complex queries typically found in recent data stream systems, e.g., conjunction(or disjunction) of predicates and join involving multiple data stream sources. Spatial Join CQ handles these complex queries easily.

4. THE DUALITY MODEL

In this section, we present a formalization of duality between data and queries, which we call the *duality model*. Section 4.1 presents the definition of a data element and a query in the duality model; Section 4.2 query processing in this model.

4.1 The Definition of a Data Element and a Query

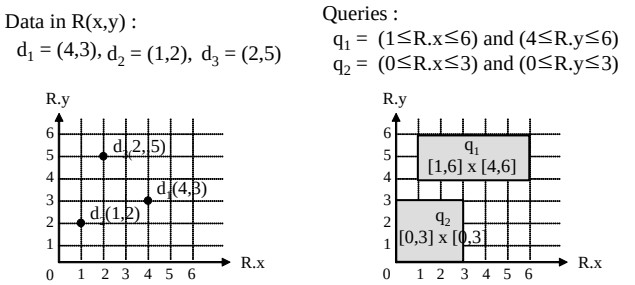
We define a data element and a query symmetrically. Definitions 1 and 2 define a data element and a query as a point and a region in the multi-dimensional space, respectively. Hereafter, since a point is a special form of a region, we call a point as a region unless there is confusion.

Definition 1. (**A Data Element in the Duality Model**) Suppose that a data element $d = (v_1, v_2, ..., v_n)$ has n attributes, and the domain of $attr_i$ is Dom_i $(0 \leq i \leq n)$. A data element d is a point $(v_1, v_2, ..., v_n)$ in the domain space $Dom_1 \times Dom_2 \times ... \times Dom_n$. That is, a data element d is a point in the n-dimensional space where an attribute becomes an axis with the coordinate value of the axis being the value of the attribute. $\mathbb{D}$ denotes a set of data elements. □

Definition 2. (**A Query in the Duality Model**) Suppose that the condition of a query q is $\varphi_q(attr_1, attr_2, ..., attr_n)$, and the domain of $attr_i$ is Dom_i $(0 \leq i \leq n)$. A query q is the region $\{(v_1, v_2, ..., v_n) \mid \varphi_q(v_1, v_2, ..., v_n),\ v_1 \in Dom_i, ..., v_n \in Dom_n\}$ in the domain space $Dom_1 \times Dom_2 \times$

... × Dom_n.[2] That is, a query is a region where the points located in the region satisfy the condition of the query. $\mathbb{Q}$ denotes a set of queries. □

Example 1. Figure 4 shows an example of data elements and queries in the duality model. The data elements d_1, d_2, and d_3, which are tuples of the relation R(x,y), are represented as the points in the two-dimensional space consisting of x and y axes as shown in Figure 4(a). For instance, d_1 is the point whose value of x is 4 and value of y is 3. The queries q_1 and q_2 over the relation R(x,y) are represented as the regions in the two-dimensional space as shown in Figure 4(b). For instance, q_1 is the region whose interval of x is [1,6] and that of y is [4,6]. □



(a) Data elements in the multi-dimensional space

(b) Queries in the multi-dimensional space

Figure 4: An example of data elements and queries in the duality model.

4.2 Definition of Query Processing

In the duality model, query processing is defined as the process of finding the pairs from a set of data elements and a set of queries that overlap in the multi-dimensional space. Lemma 1 shows the correctness.

Lemma 1. Given a set $\mathbb{D}$ of data elements and a set $\mathbb{Q}$ of queries, a data element $d \in \mathbb{D}$ satisfies the condition of a query $q \in \mathbb{Q}$ if and only if d overlaps with q in the multi-dimensional space.

PROOF: By Definitions 1 and 2, the data element d and the query q are defined as the point (v'_1, v'_2, ..., v'_n) and the region $\{(v_1, v_2, ..., v_n) \mid \varphi_q(v_1, v_2, ..., v_n) = true\}$, respectively. If the point and the region overlap, it means that $\varphi_q(v'_1, v'_2, ..., v'_n)$ becomes true according to Definition 2. Thus, the data element d satisfies the condition of the query q. □

Example 2. In Figure 4, the data element d_2 overlaps with the query q_2. We note that the value (1,2) of d_2 satisfies the condition ($0 \leq$ R.x ≤ 3) and ($0 \leq$ R.y ≤ 3) of q_2. Similarly, d_3 overlaps with q_1 satisfying the condition of q_1. In contrast, d_1 does not satisfy the condition of any queries because there is no query that overlaps with d_1. □

[2]Given a query condition $C_1 \vee C_2 \vee \cdots \vee C_n$ where C_i is a conjunction of predicates, each C_i is represented as a region. Thus, a query condition may be represented as a collection of multiple regions. For ease of exposition, we consider only the query condition having one C_i. We can easily extend this simple strategy so as to handle multiple C_i's by ORing the results of individual C_i's.

5. A CONTINUOUS QUERY PROCESSING ALGORITHM BASED ON SPATIAL JOIN

In this section, we propose our continuous query processing algorithm *Spatial Join CQ*. To show progressive development of the algorithm, we first develop the algorithm for simple selection continuous queries, and then, extend it to accommodate sliding window join continuous queries in Section 6.

5.1 Characteristics

Our algorithm integrates two dual methods – data-initiative and query-initiative – into one algorithm. Thus, it allows us to find *multiple* queries that overlap with *multiple* data elements simultaneously. To achieve this, our algorithm first accumulates data elements arriving from data stream sources into a data set, and then, processes continuous queries with the data set in *batch* mode. Our algorithm performs *immediate* continuous query processing as a special case of batch processing where there exists only one data element in the data set. For batch processing, our algorithm first constructs MBR's(Minimum Bounding Rectangle) containing multiple adjacent data elements from the data set, and then, performs spatial join, which is a symmetric operation, between a set of MBR's and a set of queries. Hereafter, we will refer to this MBR as the *data cluster*. (In immediate processing, a newly arriving data element itself becomes a data cluster.) The data cluster allows us to prevent queries that overlap with multiple data elements from being repeatedly accessed because adjacent data elements tend to share a large portion of overlapping queries.

The continuous query processing algorithm consists of three steps as in Figure 5: *index building*, *multi-dimensional spatial join*, and *refinement*.

In the first step, we build indexes over the set of data elements and the set of queries. The data index stores points, which represent input data elements, in a simple one-dimensional index. The query index stores regions, which represent continuous queries, in a multi-dimensional index. Here, if the shape of the region representing a query is complex(i.e. is not a simple hyper-rectangle), the query index stores the MBR for the region instead. In Sections 5.3 and 5.4, we explain the details of the data and query indexes.

In the second step, we find *candidate result pairs* by performing multi-dimensional spatial join with the data and query indexes. We adopt the transform-based spatial join algorithm proposed by Song et al.[19]. Because the query index stores MBR's of queries, the results of the spatial join step could include false positives. In Section 5.5, we explain the details of the spatial join step.

In the third step, we find the final result pairs by testing whether the data element in a candidate result pair indeed makes the the conditions of the query in the pair true.

5.2 The Transform-Based Spatial Join Algorithm

In this section, we briefly introduce the transform-based spatial join algorithm adopted in this paper. The algorithm transforms spatial objects with extents into points without

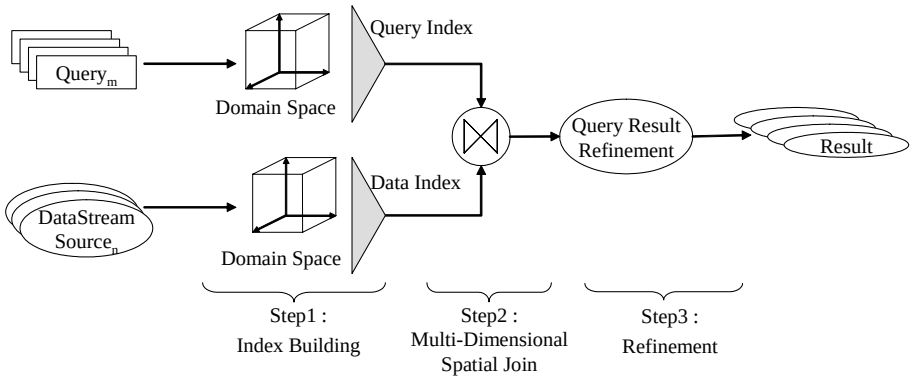


Figure 5: The three steps of the continuous query processing algorithm based on spatial join.

extents using corner transformation[18], and then, performs spatial join. It shows performance comparable to or better than those of other original-space spatial join algorithms proposed in the reference [3, 12].

Corner transformation transforms the MBR of a spatial object in the n-dimensional original space into a point in the $2n$-dimensional transform space. In corner transformation, the coordinates of a point in the $2n$-dimensional space are determined by the minimum and maximum values of the MBR on each of the n axes in the original space[11, 18]. For example, a one-dimensional object whose minimum and maximum values on the x-axis are lx and rx, respectively, is transformed into the point (lx,rx) in the two-dimensional transform space.

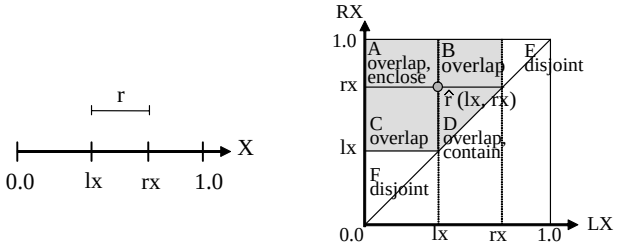
A query that finds spatial objects overlapping with a given region object in the original space is transformed into a query that finds *point* objects *contained* in a certain region in the transform space[18, 19]. Figure 6 shows relationships between a point and regions in the transform space. Here, objects overlapping with the region r in the original space are transformed into points in the regions A, B, C, or D in the transform space[18]. Using these characteristics, the original-space spatial join operation that finds spatial objects overlapping with the region r can be processed by the operation that finds points in the union of the transform-space regions A, B, C, and D, which constitutes the shaded part in Figure 6.

We use the notion of the *spatial join window* to process spatial join and formally define it in Definition 3.

Definition 3. (**Spatial Join Window**) [19] Let transform spaces of indexes R and S to be joined be *TS(R)* and *TS(S)*. The spatial join window *SJW(P)* for a rectangular region P in TS(R) is defined as the minimal region in TS(S) where all the objects overlapping with the object in P can reside. □

The method for finding SJW(P) for the region P, which is the dark shaded region in Figure 7, is as follows.[3] To overlap with objects in P, a given object must overlap with the

[3]For simplicity of explanation, we only explain the SJW in the two-dimensional transform space. The SJW for the $2n$-dimensional transform space is described in detail in Song et al.[19].



(a) Region r in the original space.

(b) Spatial relationships to r in the transform space.

Figure 6: Transform-space regions having various spatial relationships with the query region r in the original space. Here, $\hat{r}$ is the transformed point of r[18].

upper-left point of P, i.e., $\hat{q} = (lx, rx)$, because $\hat{q}$ is the largest object that can exist in the region P. The minimal region that all objects overlapping with $\hat{q}$ can exist in TS(S) is $[0, rx] \times [lx, 1]$ above the diagonal by Lemma 2. That is, SJW(P) is the light shaded region in Figure 7. Since no object can exist under the diagonal, for simplicity, we use SJW(P) as $[0, rx] \times [lx, 1]$ including the striped region from now on.

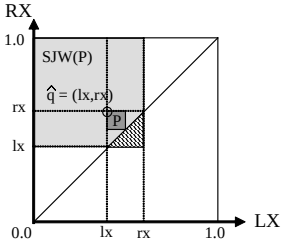


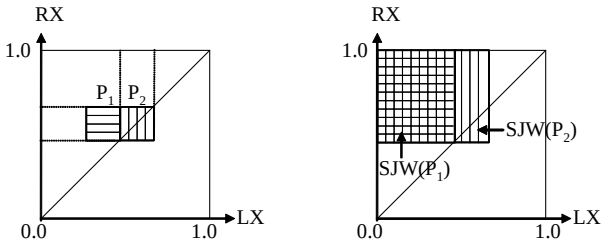
Figure 7: Spatial join window(SJW).

Lemma 2. [19] The minimal region, SJW(P), that all objects overlapping with the original space object $\hat{q} = (lx, rx)$ can exist in TS(S) is $[0, rx] \times [lx, 1]$ above the diagonal.

PROOF: See Song et al.[19]. □

As indicated by Lemma 2, the two SJW's in TS(S) for two adjacent regions in TS(R) significantly overlap. An example

is the SJW's of two adjacent regions P_1 and P_2 in Figure 8. We call this property the *overlapping property* of SJW's. Thus, we can speed up spatial join processing with a fewer number of I/O accesses by consecutively joining two adjacent regions P_i and P_j with their SJW's because objects in SJW(P_i) must have already been read into the buffer when joining P_j with SJW(P_j), and *vice versa*. Here, we assume that the buffer replacement strategy is LRU.



(a) Adjacent regions P_1 and P_2. (b) SJW(P_1) and SJW(P_2).

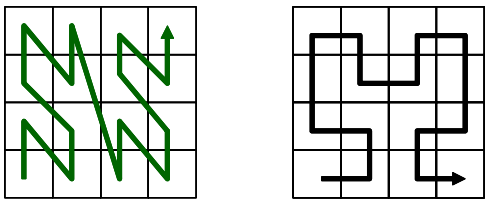
Figure 8: Spatial join windows for adjacent regions.

5.3 Building the Data Index

Our algorithm uses the data index to accumulate data elements for batch processing and to read the data elements in a proximity order. Reading them in a proximity order allows us to take advantage of the *overlapping property* of SJW(P)'s by constructing a data cluster P from adjacent data elements. The data index is destroyed after the batch is processed and is rebuilt for the next batch. The data index is not used in immediate processing.

For the data index, we adopt a one-dimensional index instead of a multi-dimensional index. This simple one-dimensional index provides sufficient capabilities for the data index, which is used only for accessing data elements in a proximity order.

We use a space filling curve[7, 17] to arrange data elements in a proximity order. The space filling curve is a method of linearly ordering regions in the multi-dimensional space such that regions tend to be close to each other if they are proximate in the multi-dimensional space[7, 17]. Representative ones include the Z-ordering[17] and the Hilbert ordering[7] shown in Figures 9(a) and (b).[4] In this paper, we use the Hilbert ordering because it changes the value of only one axis at a time, producing high proximity. We build a binary search tree to index data elements in the proximity order determined by the Hilbert ordering.



(a) Z-ordering. (b) Hilbert ordering.

Figure 9: Representative space filling curves.

[4]These space filling curves can be used in three or higher-dimensional space by recursively applying the curve to each dimension[7, 17].

5.4 Building the Query Index

Our algorithm uses the query index for efficient retrieval of the queries contained in SJW(P) from the query space for a given region P in the data space. Multi-dimensional structures such as the R-Tree, Quad Tree, Buddy Tree, and MLGF[22, 23] can be used as the query index.

Inserting a query into the query index is done as follows. The query is represented as a region by Definition 2, and the shape of the region is approximated as an MBR. The MBR is transformed into a point using corner transformation, and then, is inserted into the multi-dimensional index. As stated in the footnote of Definition 2, disjunctive queries are represented as a collection of multiple regions. In this case, the procedure described above is applied to individual regions except that these regions belong to the same query.

Example 3. Figure 10 shows an example of indexing queries represented as regions. The query q has three conditions: two attributes of data, x and y, have a range [0.2,0.5], and x is greater than y. The query q is represented as a triangular region R_q. Because such a region cannot be indexed directly, the region is approximated to MBR_q. The MBR_q is transformed into a point (0.2, 0.2, 0.5, 0.5) in the four-dimensional space, which is subsequently inserted into the multi-dimensional index. □

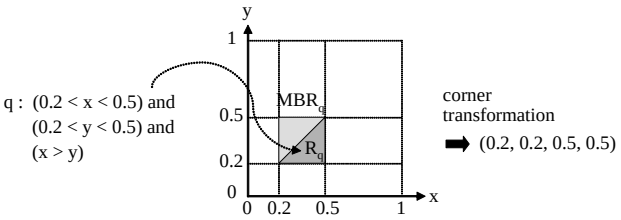


Figure 10: An example of query indexing.

Our method of building the query index is capable of efficiently handling large regions since we transform a region with extents into a point with no extents before indexing. A query becomes a large region in the original space especially when its condition has an open interval; it includes an entire domain of an attribute in the original space when there is no condition on that attribute. These large regions tend to significantly overlap with one another. If these large regions are directly indexed in the multi-dimensional index such as the R-tree, a large portion of the nodes overlap with each other, thus making the index size bigger and the query performance worse[10]. In contrast, we do not suffer from this problem because we index a *point* in our method.

Even though the performance of multi-dimensional indexes decreases as the number of dimensions increases (called *the dimensionality curse*[21]), this problem is not significant in our algorithm. The reason is twofold: the query index deals with only the attributes that appear in a query, but the number of those attributes is known to be typically one or two, and rarely more than five[10]. Besides, we can apply existing methods, such as the Pyramid-technique[2], of minimizing the dimensionality curse for exceptional cases (e.g., when the number of attributes appearing in a query is more than ten).

Algorithm **Spatial Join CQ**

Input: A newly arriving data element or a newly registered query
Output: Data element-query pairs that satisfy conditions
Algorithm:

```
    /* N_batch_data is the number of data elements in a batch */
    /* N_dc is the number of data elements in a data cluster */
    /* for immediate processing, N_batch_data = 1; N_dc = 1 */
01: if a query is newly registered then
02:     Insert the query into the query index (as described in Section 5.4);
03: end if
04: if a data element newly arrives then
05:     Insert the data element into the data index (as described in Section 5.3);
06:     if the number of data elements in the data index is N_batch_data then
07:         repeat by ⌈N_batch_data / N_dc⌉ times
08:             Construct a data cluster DC containing N_dc data elements read in
                in proximity(Hilbert) order using the data index;
                (For immediate processing, DC consists of only one newly arriving data element)
09:             Retrieve the queries contained in SJW(DC) using the query index;
10:             Find data element–query pairs, from DC and SJW(DC),
                that satisfy the overlapping condition (Lemma 1);
11:             Refine the data element–query pairs by checking whether the data element
                makes the conditions of the query true;
12:         end repeat
13:         Delete all data elements from the data index;
14:     end if
15: end if
```

Figure 11: The continuous query processing algorithm Spatial Join CQ.

5.5 Multi-Dimensional Spatial Join

In the multi-dimensional spatial join step, the algorithm constructs a region P by reading data elements in a proximity order and retrieves the queries contained in SJW(P). The former operation exploits the data index; the latter the query index. Then, the algorithm finds the overlapping pairs from data elements in P and queries in SJW(P).

The region P corresponds to the data cluster stated before. The data cluster is an MBR containing a fixed number of data elements, where data elements are retrieved in a proximity order. The algorithm needs to control the size of the data cluster to prevent its size from becoming overly large or small. If the size becomes too large, so does the SJW of the data cluster, thus making a large portion of the query index to be accessed. If the size becomes too small, the performance gain that could be obtained by batch processing is diminished. Hence, the size of the data cluster is an important parameter for tuning the performance of continuous query processing. It is difficult to determine the optimal size of the data cluster by analytical prediction since it varies depending on many parameters such as the numbers or distributions of data elements and queries. Hence, in this paper, we find the optimal size through experiments in Section 7.2, leaving the analytical solution as a future study.

Figure 11 shows our algorithm for processing continuous queries. We call it *Spatial Join CQ*. In the first step(lines 1∼3) and the second step(lines 4∼5), we create the query index and the data index as described in Sections 5.4 and 5.3, respectively. In the third step(lines 6∼14), we perform spatial join to process continuous queries. For batch processing, we perform spatial join when N_{batch_data} data elements have been accumulated; for immediate processing, when one data element arrives. Spatial join is processed $\lceil N_{batch\_data}/N_{dc} \rceil$ times as follows (for immediate processing, $N_{batch\_data} = 1$ and $N_{dc} = 1$). First, N_{dc} data elements are read in in the proximity order by using the data index, and a data cluster is created(lines 6∼8). Second, the queries contained in the SJW of the data cluster are retrieved by using the query index(line 9). Third, the overlapping pairs are found from the data elements and the queries(line 10). After spatial join is done, the algorithm refines the result pairs and obtains the final results(line 11). Finally, the algorithm destroys the data index(line 13). The data index is rebuilt with the next batch of data elements that newly arrive.

Example 4. Figure 12 shows an example of the continuous query processing done in the batch mode. In the data space of Figure 12, we have six data elements consisting of only one attribute x. Each data cluster is created with three data elements. In the query space of Figure 12, Data Cluster$_1$ is joined with the queries q_1, q_2, and q_3, and Data Cluster$_2$ with the queries q_3, q_4, and q_5. □

6. EXTENSION OF SPATIAL JOIN CQ FOR SLIDING WINDOW JOINS

In this section, we extend the algorithm presented in Section 5 so as to accommodate sliding window join continuous queries.

We first explain sliding window join as done by Kang et al.[13]. Figure 13 shows sliding window join processing be-

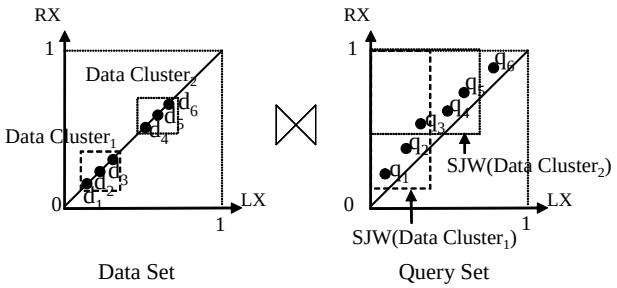
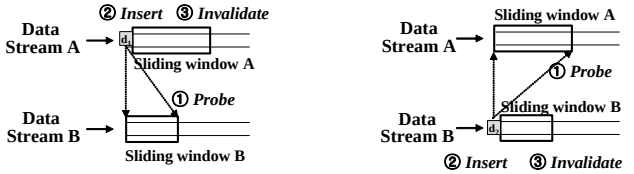


Figure 12: An example of the data element-query spatial join using the data cluster.

tween the data streams A and B. Figure 13(a) shows the join processing steps when the data element d_1 newly arrives from the data stream A: (1) checking whether the join condition is satisfied between the data element d_1 and the data elements contained in the sliding window of the data stream B(*probe*); (2) inserting the data element d_1 into the sliding window of the data stream A(*insert*); and (3) removing the data elements whose timestamps exceeded the sliding window range from the sliding window of the data stream A(*invalidate*). Figure 13(b) shows the join processing steps when the data element d_2 newly arrives from the data stream B. The steps are symmetric with those in Figure 13(a).



(a) A data element d_1 arrives from the data stream A.

(b) A data element d_2 arrives from the data stream B.

Figure 13: Sliding window join.

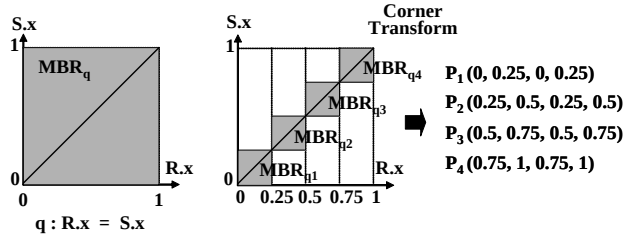
To adapt our algorithm to sliding window join, we need to extend the three methods: 1) indexing data elements, 2) indexing queries, and 3) multi-dimensional spatial join. The first should be extended so as to maintain only data elements within the sliding window range, the second to index the join conditions as well as the sliding window conditions, and the third to check these conditions.

Extension of the data indexing method

The data index uses a doubly linked list as well as a binary search tree to maintain data elements that are within the sliding window range. The data index also stores timestamp values indicating the arrival time of data elements. When a data element arrives from a data stream, it is inserted into the binary search tree, and then, linked at the head of the doubly linked list. If the timestamp of a data element exceeds the range of the sliding window, it is eliminated from the tail of the doubly linked list and from the binary search tree.

Extension of the query indexing method

When the join condition is equi-join, it is represented as a diagonal in a two-dimensional plane. In Figure 14(a), the query q has an euqi-join condition, R.x = S.x, between two data stream sources R and S. It is represented as a diagonal between (0, 0) and (1, 1) in the plane of R.x × R.y. If the join condition is a theta-join instead of an equi-join, the query is represented as a triangular region either above or below the diagonal. If the region is approximated as an MBR, the MBR becomes the entire region in the plane. If such a badly approximated MBR were used to index a query, search efficiency of the query index would become worse. For example, in Figure 14(a), the MBR of the query q is represented as the entire plane of R.x × S.x.



(a) MBR of the join condition.

(b) MBR using the query decomposition method.

Figure 14: An example of indexing join condition.

In order to resolve the problem, we exploit the object decomposition method[14]. The object decomposition method divides a query region into an arbitrary number of pieces and constructs an MBR for each piece. For example, in Figure 14(b), the query q is split into four pieces through the axis R.x, and then, each MBR is stored in the query index. By applying the object decomposition method, we can index join conditions more accurately.

The query index stores the sliding window range as a range condition for an axis in the multi-dimensional space. In the multi-dimensional spatial join step, the axis for the sliding window range in the query index spatial joins with the axis of the timestamp in the data index. By using this method, the query index allows us to store continuous queries having different sizes of sliding windows together and to process these continuous queries as a single spatial join.

This way, the query index can store not only selection conditions but also join conditions. This is a very desirable property in the sense that, when both of selection and join conditions are specified in a query, we can process them simultaneously using one spatial join. In contrast, it has been pointed out as a problem that the IBS tree is inherently incapable of indexing join conditions since it is created on each attribute[10].

Extension of multi-dimensional spatial join

As stated before, sliding window join is processed between a data element newly arriving from a data stream and the set of the data elements contained in the sliding window of the other data stream. Thus, sliding window join can be regarded as a kind of batch processing. We refer to the pair of data elements as *joinable data pair* and formally define it in Definition 4.

Definition 4. (**Joinable Data Pairs**) Consider two data stream sources DS_i and DS_j. The set of joinable data pairs $\mathbb{JD}_{DS_i,DS_j}$ when the data element d_i arrived from DS_i is

defined as follows. Here, SW(DS_j) means a set of data elements contained in the sliding window of DS_j

$$\mathbb{JD}_{DS_i,DS_j} = \{d_i\} \times \mathrm{SW}(\mathrm{DS_j}) \qquad \square$$

Our sliding window join algorithm proceeds as follows. First, the algorithm computes $\mathbb{JD}_{DS_i,DS_j}$ upon arrival of a data element d_i from the data stream source DS_i. While computing $\mathbb{JD}_{DS_i,DS_j}$, data elements in SW(DS_j) are retrieved in the proximity order using the data index. Next, the algorithm constructs data clusters with a fixed number of joinable data pairs. Finally, the algorithm finds query results by doing spatial join between the set of data clusters and the set of queries as stated in Section 5.5. Overall, the algorithm for join queries is analogous to the algorithm in Figure 11 except that a data cluster is constructed with joinable data pairs instead of data elements.

The sliding window join algorithm can be used also for multi-way join queries. For multi-way join queries, we compute $\mathbb{JD}_{DS_1,...,DS_n}$ and perform spatial join between $\mathbb{JD}_{DS_1,...,DS_n}$ and the query index. However, $|\mathbb{JD}_{DS_1,...,DS_n}|$ tends to proliferate because of the Cartesian product among $DS_1, ..., DS_n$. Efficient processing of multi-way join is left as the topic of a future paper.

7. PERFORMANCE EVALUATION

In this section, we present the results of performance evaluation. We describe the experimental data and continuous queries in Section 7.1 and present the results of the experiments in Sections 7.2, 7.3, and 7.4.

7.1 Experimental Data and Queries

We compare the performance of our algorithm with those of existing algorithms: the predicate indexing method using the IBS tree and the query plan sharing method. These methods are commonly used in recent data stream systems such as TelegraphCQ, NiagaraCQ, and STREAM. Table 1 summarizes the algorithms used in the experiments. We first compare the performances of processing simple selection queries, and then, those of processing sliding window join queries.

Table 1: Algorithms compared in the experiments.

Simple selection query processing	
SJ-Batch	Spatial Join CQ (batch processing)
SJ-Immediate	Spatial Join CQ (immediate processing)
IBS	Continuous query processing using the IBS tree
Sliding window join query processing	
SJ-Join	Spatial Join CQ (extended for join)
IBS-JS	Processing first join conditions using the query plan sharing method, and then, processing simple selection conditions using the IBS tree

For the experimental results, we measure the elapsed time for processing a query. The elapsed time is an important measure in a data stream environment since it determines the number of input data elements that can be processed in a unit time. For fair comparison, we include the time for creating the data index in the elapsed time for the Spatial Join CQ algorithm.

Experiments are performed using one data stream source for simple selection queries and two data stream sources for sliding window join queries. We use synthetic continuous queries consisting of randomly generated conditions except when we need to use specific queries. A simple selection query is a conjunction of selection conditions over randomly chosen attributes. A selection condition is expressed as $constant_1$ op_1 *attribute* op_2 $constant_2$. Here, *attribute* means the name of an attribute; $constant_1$ and $constant_2$ are values randomly selected from [0, 1]; and op_1 and op_2 are binary comparison operators randomly selected from $\leq$ and $<$. Besides, a selection condition forms a left open interval, right open interval, or closed interval with a uniform probability. A sliding window join query involves join conditions – $attribute_1$ op $attribute_2$ – as well as selection conditions. Here, $attribute_1$ and $attribute_2$ mean join attributes of data stream sources, and op a binary comparison operator randomly selected from $>$, $\geq$, $=$, $<$, and $\leq$. In addition, we set the sizes of all the sliding windows to be the same. For data elements, we generate and use synthetic data whose attribute values are randomly selected from [0, 1].

We use the Multilevel Grid File(MLGF)[22, 23] for the query index. For fair comparison, our algorithm reads the MLGF into main-memory in advance because the other algorithms compared are main-memory based. We use the Hilbert ordering to arrange data elements in the proximity order in the data index.

All the experiments have been conducted on a PC with a 2.5GHz Pentium4 processor and 1GBytes of main memory running Windows/XP.

We now summarize in Table 2 the notation to be used throughout Section 7.

Table 2: Summary of notation.

Symbols	Definitions
N_{query}	Number of registered continuous queries
N_{attr}	Number of attributes in a data stream[5]
R_{pred}	Number of attributes appearing in the predicates of a query / N_{attr}
N_{batch_data}	Number of data elements in one batch processed together by SJ-Batch
N_{dc}	Number of data elements in a data cluster
$Size_{sw}$	The size of the sliding window (in the number of data elements)

7.2 Optimal Value Of N_{dc}

We first find the optimal value of N_{dc} by experiments. N_{dc} is an important parameter for tuning the performance of SJ-Batch. The optimal size varies depending on N_{batch_data}, N_{query}, N_{attr} as well as distributions of data and queries.

We vary N_{dc} and choose the optimal value of N_{dc} that makes the elapsed time minimized. An interesting result is that the differences in the elapsed time are very small around the optimal value of N_{dc}. For example, the elapsed time

[5] In our experiments, we assume that all attributes in data streams can appear in query predicates. Thus, N_{attr} means the number of attributes stored in the query index.

increases only by 10% as N_{dc} is varied from 200 to 1200 when the optimal value of N_{dc} is 400. This result indicates that we do not need to know the *exact* optimal value of N_{dc}, rendering determination of N_{dc} easy.

Table 3 summarizes the optimal N_{dc}'s obtained by experiments when N_{batch_data} is varied from 2,000 to 10,000 and N_{query} is fixed to 10,000. Using these optimal N_{dc}'s, we measure the elapsed time per a data element. We use these optimal N_{dc}'s also in our subsequent experiments.

Table 3: Summary of the optimal values of N_{dc} and the elapsed times ($N_{query} = 10,000$).

N_{batch_data}		2,000	4,000	6,000	8,000	10,000
N_{attr} = 4	Optimal N_{dc}	100	120	120	160	200
	Elapsed time /data elements (ms)	0.406	0.391	0.391	0.381	0.375
N_{attr} = 8	Optimal N_{dc}	200	240	360	400	400
	Elapsed time /data elements (ms)	0.570	0.526	0.518	0.510	0.506

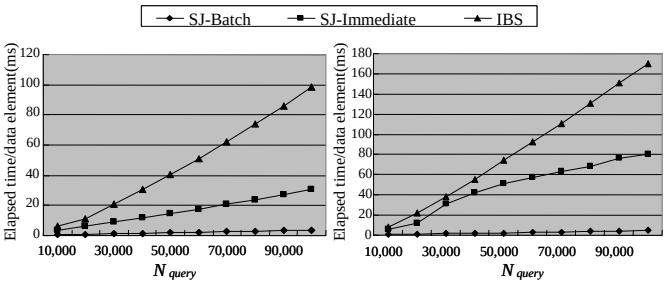
From Table 3, we note that the optimal N_{dc} gradually increases and the elapsed time decreases as N_{batch_data} increases, i.e., when batch processing is performed with more data elements accumulated. The reason why the optimal N_{dc} increases is that the algorithm can construct a data cluster with more data elements located in proximity since the density of points representing data elements in the multi-dimensional space becomes higher as N_{batch_data} increases. The reason why the elapsed time decreases is that the algorithm gets more benefit of batch processing since more data elements can be processed with a single probe into the query index as N_{dc} increases. Thus, for batch processing, it is more profitable to process continuous queries with more data elements accumulated.

7.3 Simple Selection Continuous Queries

Effects of N_{query} and N_{attr}

Figure 15 shows the elapsed time for simple selection continuous queries as N_{query} is varied. For SJ-Batch, we set N_{batch_data} = 10,000 representatively and use N_{dc} = 200 when $N_{attr} = 4$ and $N_{dc} = 400$ when $N_{attr} = 8$. The results indicate that SJ-Batch performs the best, SJ-Immediate the next, and IBS the worst. The reason why SJ-Immediate outperforms IBS is mainly due to different ways of using indexes. IBS searches multiple IBS trees, one for each attribute in the query condition, and merges the results. In contrast, SJ-Immediate accesses only one multi-dimensional query index and does not need to merge the results. The reason why SJ-Batch shows the best performance is that it reduces the number of accesses to the query index by virtue of the data cluster. Moreover, SJ-Batch minimizes the number of I/O's for searching the query index by using the overlapping property of the SJWs.

In Figure 15(a), SJ-Immediate outperforms IBS by 1.7 times when N_{query}=10,000 and by 3.2 times when N_{query}=100,000. Similarly, SJ-Batch outperforms IBS by 15.0 and 28.5 times, respectively. In Figure 15(b), SJ-Immediate outperforms IBS by 1.4 times when N_{query}=10,000 and by 2.1 times when N_{query}=100,000. Similarly, SJ-Batch outperforms IBS by 15.9 and 36.2 times, respectively. These results indicate that the performance advantage of our algorithms over IBS improves as the number of registered queries, N_{query}, increases. We also observe that the number of attributes does not significantly affect the performance advantage of our algorithms.



(a) $N_{attr} = 4$. (b) $N_{attr} = 8$.

Figure 15: Elapsed time for selection query processing as N_{query} is varied (N_{batch_data} = 10,000).

Effects of R_{pred}

Figure 16 shows the elapsed time for simple selection continuous queries as R_{pred} is varied. These results indicate that the elapsed time for IBS increases as R_{pred} does because the depth of the IBS tree is proportional to $\log R_{pred}$ [10]. In contrast, the performances of SJ-Batch and SJ-Immediate improve as R_{pred} increases. The reason is that the depth of our query index (i.e., the MLGF) is not affected by R_{pred} but only by N_{query} since a query is stored as one region in the MLGF regardless of the number of predicates specified in a query. Moreover, in SJ-Batch and SJ-Immediate, the efficiency of searching the query index is enhanced as the number of predicates increases because the size of the region representing a query becomes smaller.

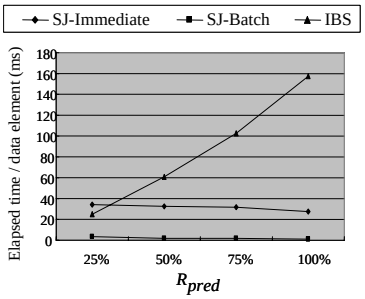


Figure 16: Elapsed time for selection query processing as R_{pred} is varied (N_{attr}=8, N_{query}=50000).

Effects of Distribution of Data Elements

In all the experiments stated above, we assume uniform distribution of input data elements. Now, we show some results for skewed distribution where the values of most input data elements are confined to a specific narrow range. An example of skewed distribution is a data stream from a room temperature sensor: typically, room temperature does not change rapidly. We define the *skewness* R_{skew} of input data elements as the ratio of the number of data elements that fall into a specific range to that of total data elements. For simplicity, we consider a specific range whose size is 10% of the size of the entire domain. Figure 17 shows the performance variation as skewness is varied. The elapsed times for

SJ-Immediate and IBS are shown to be insensitive to data distribution. In contrast, the elapsed time for SJ-Batch decrease as skewness increases, showing improvement by up to 1.7 times compared with that for uniform distribution. The reason for the improvement is that a data cluster can be constructed with a larger number of data elements since the density of data elements in the specific range becomes higher. We find out that, by experiments, the optimal N_{dc} indeed increases from 200 to 400 as skewness does from 20% to 100%. Therefore, the advantages of SJ-Batch get more marked for skewed distributions.

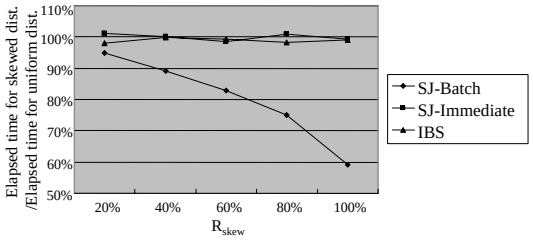


Figure 17: Comparison of elapsed time as R_{skew} is varied (N_{attr}=4, N_{batch_query}=100,000, N_{batch_data}=10,000).

7.4 Sliding Window Join Continuous Queries

Effects of N_{query} and $Size_{sw}$

Figure 18 shows the elapsed time for sliding window join continuous queries against two data stream sources as N_{query} is varied. In SJ-Join, we divide a region representing a join condition into four regions using the object decomposition method as explained in Section 6.2. These results indicate that SJ-Join outperforms IBS-JS by a large margin for a reason similar to that of Figure 15 showing that SJ-Batch outperforms IBS. Furthermore, this margin becomes larger as $Size_{sw}$ increases. The reason is that the elapsed time for IBS-JS increases proportionally to $Size_{sw}$, while SJ-Join reduces the frequency of accessing the query index by virtue of a larger data cluster($Size_{sw}$ corresponds to N_{batch_data} in Table 3). In Figure 18, SJ-Join outperforms IBS-JS by up to 2.6 times when $Size_{sw}$ = 100, 3.9 times when $Size_{sw}$ = 1000, and 6.9 times when $Size_{sw}$ = 10000.

In summary, the experimental results show that our algorithm outperforms earlier optimization methods for both kinds of continuous queries: simple selection and sliding window join. Furthermore, our algorithm has a desirable property that the performance advantage becomes larger as (1) N_{query}, (2) $Size_{sw}$, (3) R_{pred}, or (4) R_{skew} increase.

8. CONCLUSIONS

In this paper, we have developed a framework for efficiently processing continuous queries based on spatial join. The contributions of this paper are summarized as follows.

First, we have presented a new viewpoint of transforming the continuous query processing problem to a multi-dimensional spatial join problem. The main advantage of this transformation is to integrate the two dual methods: data-initiative and query-initiative. Hence, we have shown that spatial join is a powerful tool for processing continuous queries. The transformation takes advantage of the well-known property of duality between data and queries. We have formalized the duality property as the *duality model*.

Second, we have proposed a continuous query processing algorithm based on spatial join, named *Spatial Join CQ*. The algorithm processes continuous queries by performing spatial join, which is a symmetric operation, between a set of data elements and a set of queries. By virtue of this symmetric processing, the algorithm achieves the advantages of both of the two dual methods, rendering continuous query processing more effective.

Third, we have verified excellence of the proposed algorithm by extensive experiments. The results show that our algorithm for simple selection continuous queries outperforms earlier methods by up to 36.2 times in batch processing and by up to 3.2 times in immediate processing. Our algorithm for sliding window join continuous queries outperforms earlier methods by up to 6.9 times. Moreover, our algorithm has a desirable property that the performance advantage becomes more marked as the number of registered continuous queries, the size of sliding windows, the ratio of attributes having predicates, or the skewness of input data elements increase.

Overall, these results indicate that our approach provides a new insight into continuous query processing that can significantly improve the performance. As further studies, we need to improve our algorithm to efficiently process multi-way sliding window join continuous queries and to develop a method of automatically selecting the optimal size of the data cluster.

Acknowledgements

This work was supported by the Ministry of Science and Technology(MOST)/Korea Science and Engineering Foundation(KOSEF) through the Advanced Information Technology Research Center(AITrc).

9. REFERENCES

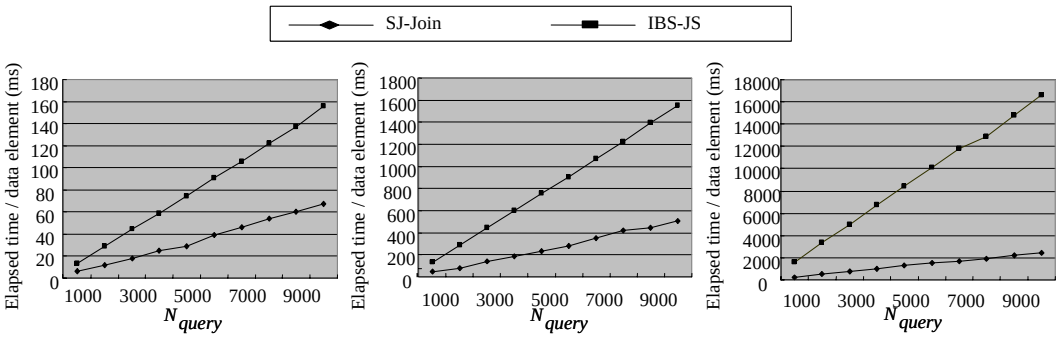
[1] Babcock, B. et al., "Models and Issues in Data Stream Systems," In *Proc. the 21st ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (PODS)*, Madison, Wisconsin, pp. 1-16, June 2002.

[2] Berchtold, S., Bohm, C., and Kriegel, H.-P., "The Pyramid-Technique: Towards Breaking the Curse of Dimensionality," In *Proc. Int'l Conf. on Management of Data*, ACM SIGMOD, Seattle, Washington, pp. 142-153, June 1998.

[3] Brinkhoff, T., Kriegel, H.-P., and Seeger, B., "Efficient Processing of Spatial Join Using R-trees," In *Proc. Int'l Conf. on Management of Data*, ACM SIGMOD, Washington, DC., pp. 237-246, May 1993.

[4] Chandrasekaran, S. and Franklin, M. J., "Streaming Queries over Streaming Data," In *Proc. the 28th Int'l Conf. on Very Large Data Bases*, Hong Kong, China, pp. 203-214, Aug. 2002.

[5] Chandrasekaran, S. et al., "TelegraphCQ: Continuous Dataflow Processing for an Uncertain World," In *Proc. the First Biennial Conf. on Innovative Data Systems Research*, Asiloma, Califonia, pp. 269-280, Jan. 2003.



(a) $Size_{sw}$ = 100. (b) $Size_{sw}$ = 1,000. (b) $Size_{sw}$ = 10,000.

Figure 18: Elapsed time for sliding window join processing as $Size_{sw}$ is varied.

[6] Chen, J. et al., "NiagaraCQ: A Scalable Continuous Query System for Internet Databases," In *Proc. Int'l Conf. on Management of Data*, ACM SIGMOD, Dallas, Texas, pp. 379-390, June 2000.

[7] Faloutsos, C. and Roseman, S., "Fractals for Secondary Key Retrieval," In *Proc. the Eighth ACM SIGACT-SIGMOD Symp. on Principles of Database Systems(PODS)*, Philadelphia, Pennsylvania, pp. 247-252, Mar. 1989.

[8] Fox, E. A. et al., "Order-preserving minimal perfect hash functions and information retrieval," *ACM Trans. on Information Systems*, Vol.9, No.3, pp. 281-308, July 1991.

[9] Golab, L. and Ozsu, M. T., "Issues in Data Stream Management," *ACM SIGMOD Record*, Vol. 32, No. 2, pp. 5-14, June 2003.

[10] Hanson, E. N. et al., "A Predicate Matching Algorithm for Database Rule Systems," In *Proc. Int'l Conf. on Management of Data*, ACM SIGMOD, Atlantic City, New Jersey, pp. 271-280, June 1990.

[11] Hinrichs, K. and Nievergelt, J., "The Grid File: A Data Structure Designed to Support Proximity Queries on Spatial Objects," In *Proc. Int'l Workshop on Graphtheoretic Concepts in Computer Science*, Linz, Austria, pp.100-113, Aug. 1983.

[12] Huang, Y.-W., Jing, N., and Rundensteiner, E. A., "Spatial Joins Using R-trees: Breadth-First Traversal with Global Optimizations," In *Proc. the 23rd Int'l Conf. on Very Large Data Bases*, Athens, Greece, pp.396-405, Aug. 1997.

[13] Kang, J., Naughton, J. F., and Viglas, S. D., "Evaluating Window Joins over Unbounded Streams," In *Proc. the 19th IEEE Int'l Conf. on Data Engineering(ICDE)*, Bangalore, India, pp. 341-352, Mar. 2003.

[14] Kriegel, H.-P. et al., "Spatial Query Processing for High Resolutions," In *Proc. the Eighth Int'l Conf. on Database Systems for Advanced Applications*, Tokyo, Japan, pp. 17-26, Mar. 2003.

[15] Mokbel, M. F., Xiong, X., and Aref, W. G., "SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases," In *Proc. Int'l Conf. on Management of Data*, ACM SIGMOD, Paris, France, pp. 623-634, June 2004.

[16] Motwani, R. et al., "Query Processing, Approximation, and Resource Management in a Data Stream Management System," In *Proc. the First Biennial Conf. on Innovative Data Systems Research*, Asiloma, California, pp. 245-256, Jan. 2003.

[17] Orenstein, J. A. and Merrett, T. H., "A Class of Data Structures for Associative Searching," In *Proc. the Third ACM SIGACT-SIGMOD Symp. on Principles of Database Systems(PODS)*, Waterloo, Canada, pp. 181-190, Apr. 1984.

[18] Seeger, B. and Kriegel, H.-P., "Techniques for Design and Implementation of Efficient Spatial Access Methods," In *Proc. the 14th Int'l Conf. on Very Large Data Bases*, Los Angeles, California, pp.360-371, Aug. 1988.

[19] Song, J.-W., Whang, K.-Y., Lee, Y.-K., and Kim, S.-W, "Spatial Join Processing Using Corner Transformation," *IEEE Trans. on Knowledge and Data Engineering*, Vol. 11, No. 4, July 1999.

[20] Terry, D. et al., "Continuous Queries over Append-Only Databases," In *Proc. Int'l Conf. on Management of Data*, ACM SIGMOD, San Diego, California, pp. 321-330, June 1992.

[21] Weber, R., Schek, H.-J., and Blott, S., "A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces," In *Proc. the 24th Int'l Conf. on Very Large Data Bases*, New York City, New York, pp.194-205, Aug. 1998.

[22] Whang, K.-Y. and Krishnamurthy, R., Multilevel Grid Files, IBM Research Report RC11516, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, Nov. 1985.

[23] Whang, K.-Y. and Krishnamurthy, R., "The Multilevel Grid File – a Dynamic Hierarchical Multidimensional File Structure," In *Proc. Int'l Conf. on Database Systems for Advanced Applications*, Tokyo, Japan, pp. 449-459, Apr. 1991.

[24] Zdonik, S. et al., "The Aurora and Medusa Projects," *IEEE Data Engineering Bulletin*, Vol. 26, No. 1, pp. 3-10, Mar. 2003.