Querying Streaming Point Clusters as Regions

Chengyang Zhang and Yan Huang
University of North Texas
Denton, TX 76203, U.S.A.
[chengyang, huangyan]@unt.edu

ABSTRACT

This paper focuses on one important type of geo-streaming data - point geo-streams. Many interesting applications require selected discrete points with similar observations to be clustered according to spatial proximity and further elevated into higher-level spatial regions. Querying streaming point clusters as regions directly in a geo-stream database has many benefits, but is also very challenging. We propose two query optimization strategies, namely *semantics-based optimization* and *incremental optimization* for answering queries involving both point geo-streams and static data set. The experimental results on a streaming meteorological data set demonstrate the effectiveness and the efficiency of the query processing techniques. Compared with the baseline methods, our optimization methods can reduce the total execution time by more than an order of magnitude.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Query processing; H.2.8 [**Database Applications**]: Spatial databases and GIS

General Terms

Algorithms

Keywords

Geo-stream, spatial clustering, query optimization

1. INTRODUCTION

With the advent of sensory and communication technologies, the volume of real-time streaming data produced by sensor networks is staggeringly large and growing rapidly. Traditional sensor networks such as those managed by the National Weather Service (NWS) [23] are collecting meteorological observations across the country at progressively finer spatial and temporal granularities. The increasingly popular location enabled devices are continuously producing large volumes of streaming data – complementary to those produced by traditional automobile sensors, e.g. loop detectors, and amenable to traffic analysis. The development of MEMS (Micro-Electro-Mechanical Systems) and nano technology promises miniature and dust-like intelligent sensors that can be scattered around to self-organize into a network to measure just about everything that you can imagine.

While stream databases such as [4, 1, 5, 2] and sensor databases such as [16, 7, 3] have successfully adopted some and reinvented the other salient features of traditional databases (e.g. query language and query processing strategy) to fit the new scenarios, insufficient attention has been paid to a large class of streaming data - geo-streams that are produced by geo-sensor networks monitoring spatially and temporally continuous phenomena such as flooding and traffic jam.

In this paper, we focus on one important type of geo-streaming data, namely, **point geo-streams**. Many interesting applications, e.g. weather and traffic, often require selected discrete points with similar observations (water level, vehicle speed, wind speed, etc.) to be **clustered** according to spatial proximity and further elevated into higher-level spatial objects, e.g. **regions**[1].

Traditionally, cluster analysis and polygonization (i.e., converting clusters to polygons) are considered as mining processes and are separated from a database system. However, there are several important benefits to query streaming point clusters as regions directly in a geo-stream database. First of all, many spatial databases have built-in functions to support a variety of operations on regions. By querying clusters as regions, we avoid the effort to reinvent many geometric functions. Secondly, regions can be more efficiently computed and stored than point clusters (e.g., the representation of a polygon only requires all of its boundary vertices). Last but not least, regions can better characterize certain phenomena (such as flooding zones) than points in many GIS applications.

Efficiently querying point clusters as regions is very challenging in streaming applications. Clustering algorithm is usually beyond linear time complexity. Elevating point clusters to polygons is non-trivial except for simple polygons or convex hulls. Continuously performing the clustering and polygonization for point clusters is unacceptable in most real time applications.

In our previous papers [14, 29], We have proposed a data-type-based approach [14] to uniformly represent geo-streams

[1]Please note that we use region and polygon interchangeably in this paper for writing convenience. However, strictly speaking a region is defined as a set of polygons.

with and without extended spatial extents. We also proposed to generalize GROUP BY to CLUSTER BY in SQL to allow static points to be aggregated into static regions and then participate in spatial queries. This paper builds on top of our existing work, and focuses on *query optimization strategies for querying streaming point clusters as regions*. Specifically, the paper brings together the following contributions:

1. We propose a novel *semantics-based query optimization* algorithm for processing queries involving streaming regions abstracted from point clusters;

2. We further develop an *incremental algorithm* to optimize queries on streaming windows;

3. We perform experimental evaluation on a real streaming meteorological data set to demonstrate the effectiveness and the efficiency of the query optimization techniques. Compared with the baseline methods, our optimization strategies can reduce the total execution time by more than an order of magnitude.

The rest of the paper is organized as follows: Section 2 elaborates the motivation and formulates the problem definition. Section 3 proposes the semantics-based query optimization algorithm as well as the incremental query optimization strategy. Experiment results are shown in section 4. Section 6 summarizes the paper and discusses future work. Finally we review the related work in Section 5.

2. MOTIVATION AND PROBLEM FORMULATION

In this section, we first show a motivating example, and then formulate the problem definition.

Example 1 (Motivating Scenario: Flood Damage Analysis): During and after a flood, the city hydrologic information center needs to continuously monitor and analyze the flooding and its damage. Surface water data in the format of *(location,time,waterLevel)* can be obtained by a variety of sensors, such as a network of submersible pressure transducers. A location is considered flooded if it has a surface water level above a user-given threshold d. A flooding zone is formed by elevation of flooded point clusters into regions. A spatial database is used to store the county and house information in two separate tables. We list three challenging queries.

- **Q1**: Continuously list the counties traversed by a flood in the past 2 days.

- **Q2**: Continuously return the areas(sizes) of the flooding regions.

- **Q3**: Continuously list the phone numbers of all the houses within 3 miles of the flooding water front.

Assuming we have the following table schemas, the above queries cannot be easily answered without complicated clustering and polygonization process. This is because there is an inherent *mismatch* between discrete point-based observations (e.g. waterLevel) and continuous phenomenon (e.g. flood).

waterLevel(location: *point*, waterLevel: *streaming real*)
county(name: *string*, extent: *region*)
house(owner: *string*, phone: *string*, extent: *point*)

However, if the geo-stream database supports the elevation of point clusters as flooding regions, i.e., we have the following schema, the above queries can be much more intuitively expressed and answered.

flood(floodname: *string*, extent: *streaming region*)

We may use the syntax proposed in our previous paper [14, 29] to express query Q1, Q2 and Q3.

```
Q1 SELECT  c.name
    FROM  flood f, county c
    WHERE intersect(f.extent[past 2 days], c.extent)

Q2. SELECT name, extent[now], area(extent[now])
    FROM   flood

Q3. SELECT  h.phone
    FROM    flood f, house h
    WHERE   distance(f.extent[now], h.exent) < 3
```

The above example can be generalized to the following query that involves streaming regions (e.g. flood) and static data set (e.g., county and house).

```
Q: SELECT     Streaming/non-streaming attributes,
           aggregate function
  FROM       Streaming regions, static dataset
  WHERE      Streaming spatial predicates
```

The rest of the paper will focus on how to **optimize** this type of query. More formally, the problem is defined as the following:

Given: A set of point streaming observations S (location: *point*, reading: *streaming real*).

Find: Answers to query Q that involves streaming regions S_r formed by clusters of points in S above a user specified threshold d and static regions.

Objective: Optimize the query execution in terms of CPU and IO costs.

3. QUERY PROCESSING STRATEGIES

To process real time point cluster streams and answer continuous queries over them, we need efficient data structures and query processing mechanisms. In this section, we will use the motivating query Q1 in Section 2 to illustrate query processing strategies. First, we will go through the basic data structure and a naive query processing method. Then our semantics-based and incremental optimization strategies are proposed. At the end of the section, we will discuss how our approaches can be applied in a general context.

3.1 Basic Data Structure

Consider a geo-stream S representing point observations in our context. Without loss of generality, we assume that the schema of S is of the format: S(reading: *streaming value*, extent: *point*), where *extent* is a static point data type with two component x and y in certain reference coordinate framework (e.g. longitude and latitude, or relative locations).

In most queries, *WHERE* statement is used to apply boolean conditions in order to select a subset of the original streams emitted by discrete sensor points. We assume that memory is enough to hold all the real time streaming data of recent sliding window. At each time slice, a different set of points may be selected to fulfill the query, therefore it is desirable to have a memory-based indexing structure to efficiently access each point.

We maintain a doubly linked list for all the points. Each point is inserted into the doubly linked list based on the ascending order of x coordinate, and then on the ascending order of y coordinate when the points have equal values for x coordinate. However, locating an arbitrary point in the list still requires a linear scan from the beginning or end of the doubly linked list. Therefore, in addition, we maintain a sparse index over x coordinates. Thus, the whole space is divided into equally sized stripes according to the x coordinates. There is a single entry in the sparse index for each stripe pointing to the first point in the linked list that appears in that stripe. Each point in the list is also associated with a sequence of timestamps, e.g. $(t_0, t_2, ..., t_w)$, which contains all the timestamps when data emitted by the point satisfies the selection criteria. New timestamps are appended to the end of the sequence, and expired timestamps are removed from the beginning of the sequence. Thus the timestamp sequence is incrementally updated and has a maximum size of query window size w. In the application where sensors are static (such as the example queries), sorting and indexing only need to be performed once, while the timestamp sequence is updated as data streams come in.

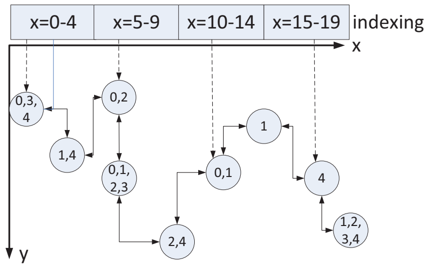


Figure 1: Data Sorting and Indexing

Example 2(Data Sorting and Indexing) Figure 1 shows an illustrative example of the above data sorting and indexing mechanism. Each entry in the sparse index points to the first point appearing in that stripe. The sequence of numbers associated with each point contains all the timestamps in the query window when reading from that point meets the selection criteria.

Please note that the basic data structure is independent of our query optimization strategies that are introduced later. In the scenarios where sensors are moving, we may need to use some moving object indexing [20, 19]. The details of these indexing methods are beyond the scope of this paper.

3.2 Naive Query Processing

A naive query processing approach applies clustering algorithm and polygonization algorithm to aggregate/elevate point observations as regions at each time slice. The resultant streaming regions can then participate in queries directly. However, both clustering and polygonization operations are computationally expensive, which limits the feasibility of the approach in processing geo-streams. We propose a straightforward approach with basic optimizations and then compare it with our semantics-based and incremental query optimization methods later.

Various clustering methods can be performed on point data. For example, DBSCAN [9] identifies a number of clusters from a set of points using the estimated density distribution of points. Density-based algorithms fit our applications better because they identify arbitrary shaped clusters. The original DBSCAN algorithm has two parameters: maximum radius of the neighborhood Eps and minimum number of points $MinPts$ in an Eps-neighborhood of any point. The algorithm starts with an arbitrary point p, and retrieves all points whose distance to p is no more than Eps. If the number of such points is larger than $MinPts$, p is considered a core point of a cluster. The *density-reachable* points of p are points within Eps of P, and are either border points or core points depending on whether they have enough points around them. Otherwise p may be a noise point or border point of some other cluster. This process iterates until all the points are visited. Paper [9] provides more details of the algorithm.

The spatial clusters generated in the clustering step need to be elevated into regions, which then participate in queries involving other geo-streams and/or static spatial datasets. Paper [15] provides one such method. However, this method is only for visualization purposes and does not compute the sequence of the vertices that constitute the resultant polygons. Therefore we go one step further to obtain the ordered edge sequences for the resultant regions. Briefly, we first construct delaunay triangulations(DT) over all the border points in each cluster respectively. During the DT construction, we also record the length of each edge. Using the average length of all the edges for a particular cluster, we may heuristically remove all the edges that are longer than the average length (with a tuning factor α) because these edges are unlikely to be the boundary of the resultant regions. Then we remove inner edges of the polygon by deleting all edges that are shared by more than one triangulations. Finally we start from one arbitrary edge $e0$ and continue to another edge that share the same vertex with $e0$. This process iterates until a cycle is formed. We then start with another unvisited edge until all the edges have been traced.

Let S_r (name: *string*, extent: *streaming region*) represent the streaming regions; S_r may participate in queries in different ways. For example, Query Q1 involves joining S_r(i.e. flood) with a static spatial relation R(i.e. county) on their *extent* attributes. We use query Q1 to illustrate the algorithms and discuss other query types later in this section. We may use the following query processing strategies:

- **Linear Scan Join** Since every time slice of S_r for current window is in memory, we perform a linear scan on R. For each tuple r in each retrieved block of R, we check every snapshot of S_r to see whether any region intersects the extent of r. The results from all snapshots can be used to determine r's eligibility as the returned query answer. Then we continue with the next tuple in the block. When a block is consumed, a new block of R is read into the memory. In this strategy, R only needs to be linearly scanned once for the current window, while S_r needs to be checked n_R times, where n_R denotes the number of tuples in R.

- **Indexed Scan Join** An alternative approach is to compute the bounding box for each individual region in S_r. Then indexes built on static relation R can be used to limit the number of potential tuples of R to be retrieved. For each region s in S_r, all the tuples in R that intersect s's bounding box are retrieved and tested. The results from all the regions in S_r can collectively determine the final query answers for current window.

3.3 Semantics-based Query Optimization

The above selection-clustering-polygonization approach answers queries by explicitly aggregating and elevating the points into streaming regions, which may then participate in queries directly. However, it is often not necessary to find the actual regions(polygons) to answer the queries. Depending on the semantics of the query, answers(or approximation of answers) may be obtained without clustering and polygonization process. In this subsection, we will propose the semantics-based query optimization method for the join query illustrated in query Q1. The end of the section discusses how the approach can be applied to other query types.

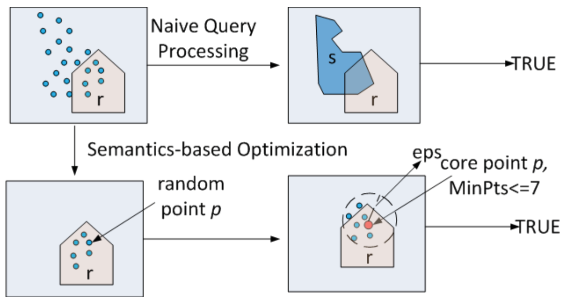


Figure 2: Semantics-based Optimization v.s. Naive Query Processing

Figure 2 compares the semantics-based optimization with the naive approach in processing one time slice of query Q1. In naive query processing, points are aggregated and elevated into regions to participate in join queries. Whether any resultant region s in S_r intersects any tuple (polygon of county) r in R can be determined by calling standard programming libraries or functions built into the spatial database.

On the other hand, in the semantics-based optimization approach, we take the following steps:

1. We first select a random point p from all the points inside any polygon r in R. Then we search p's Eps neighborhood (a circle with center p and radius Eps) to find all the points close to p. If more than $MinPts$(number of points that are enough to from a cluster) neighboring points are found, we consider p a core point, and the potential streaming region s in S_r is considered intersecting r at this time snapshot.

2. If p is not a core point, we continue with another point q in the p's neighborhood to see whether q is a core point and p is a border point.

3. If p is neither a core point nor a border point, we mark p visited and start with another unvisited point inside r. The process stops as soon as we find one core/border point inside r.

4. If we cannot find any core or border points inside r (either there is no point inside r, or the points inside r are not dense enough to form a region), we consider s not intersecting r at this time slice.

The details of algorithm to determine intersecting condition in our semantics-based query optimization is formally presented in Algorithm 1.

Algorithm 1 Intersecting Condition Test Algorithm for polygon r at time slice t_0 ($ICT(r, t_0)$)

```
1: Compute $r$'s bounding box $BB(r)$
2: From original geo-stream $S$'s snapshot at time $t_0$, find
   all the points $P[i], i = 1$ to $n$ that are inside $BB(r)$
3: for $P[i] = P[1]$ to $P[n]$ do
4:    Compute $P[i]$'s $Eps$ neighborhood $N_{P[i]}$, with center
      $P[i]$ and radius $Eps$
5:    Compute $numP1$ as number of points inside $N_{P[i]}$
6:    if $numP1 \geq MinPts$, where $MinPts$ is the minimum
      number of points to form a region then
7:       $ICT(r, t_0) = TRUE$
8:       Exit
9:    else
10:      for $N_{P[i]}[j] = N_{P[i]}[1]$ to $N_{P[i]}[numP1]$ do
11:         Compute $numP2$ as number of points inside
            $N_{P[i]}[j]$
12:         if $numP2 \geq MinPts$ then
13:            $ICT(r, t_0) = TRUE$
14:            Exit
15:         end if
16:      end for
17:   end if
18: end for
19: $ICT(r, t_0) = FALSE$
```

Given Algorithm 1, queries that join S_r and R over window w(w = 2 days in Query Q1) can be answered using Algorithm 2. It processes tuples from R sequentially. Initially, for each tuple r, every snapshot over window w is evaluated using Algorithm 1 and the result is appended to queue Q. When the current window slides, Q's content is updated, i.e. the value for the earliest time slice is removed, and the value for latest time is appended. Then we evaluate the content of Q to produce answers according to the semantics of the query. For example, when Q contains only TRUE values, r is considered the answer for query Q1.

Algorithm 2 Semantic based Query Processing over current window w

```
1: for each retrieved tuple $r$ from static relation $R$ do
2:    Let $t_c$ = NOW
3:    if $t_c - w \leq 0$ then
4:       $Q.append(ICT(r, t_c))$
5:    else
6:       $Q.append(ICT(r, t_c))$
7:       $Q.remove(ICT(r, t_c - w - 1))$
8:       if $Q(j)$ is TRUE for all $j$ from 0 to $w$ then
9:          Output $r$ as query answer for query Q1
10:      end if
11:   end if
12: end for
```

3.4 Incremental Optimization

The approaches we have discussed so far observe a snapshot-based pattern to answer all the queries. However, the temporal continuity of streaming data implies that it may not be necessary to explicitly evaluate query conditions at each time slice. For example, to answer query Q1, if we find that condition at one time snapshot t_k is FALSE, we can decide that query condition over any window w that covers t_k is FALSE. We call t_k the *spoiler* (time). We may safely skip some time snapshots until the current window has passed the *spoiler*, as shown in Figure 3. Note that in Figure 3, we continuously evaluate query conditions starting from current time t_c until we find time t_k, at which the condition is FALSE.

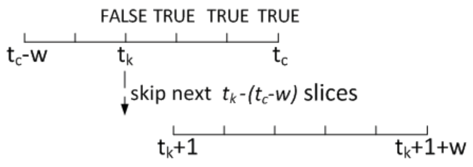


Figure 3: Illustration of Incremental Optimization for Query Q1

The process of incremental optimization for query Q1 is formally given in Algorithm 3.

Algorithm 3 Window based Query Processing for Query Q1 over current window w

```
1: for $j = 1$ to $n_R$, $n_R$ is number of tuples in $R$, $j$ is each tuple's id (in memory) do
2:     Let $t_c$ = Current Time
3:     if $(t_c - w) \leq t_k(j)$ then
4:         DO NOT retrieve tuple $r_j$
5:         Exit
6:     end if
7:     Retrieve tuple $r_j$ from $R$
8:     for $t_i = t_c$ to $t_c - w$ do
9:         if $ICT(r_j, t_i) == false$ then
10:            $t_k(j) = t_i$
11:            Exit
12:        end if
13:    end for
14:    Output $r_j$ as the query answer over current window $w$
15: end for
```

Note that in Algorithm 3, we keep *spoiler* t_k associated with each tuple's id in memory. If the current window has not passed the t_k, that tuple is not retrieved from the disk. Thus we can save IO cost as well as computation cost (to test intersecting condition).

Please also note that in Query Q1, we assume the semantics of "intersect" is "intersect for all the time slices". However, we can easily adapt the definition of *spoiler*, as well as Algorithm 1-3 to support the semantics of "intersect for any one of the time slices".

3.5 Optimization for General Query Types

We have illustrated the optimization strategies using query Q1. To see that these approaches are not ad-hoc, we need to identify type of queries that our optimization methods can be applied to.

OBSERVATION 1. *Incremental optimization can be applied to any spatial predicates.*

Spatial predicates evaluate the relationship between spatial objects and return "true/false" boolean results. Spatial predicates usually appear in the "WHERE" statement. OpenGIS specification defines many spatial relationship functions that are predicates, such as *Intersect(s)*, *Contain(s)*, *Disjoint*, *Touch(es)*,*Overlap(s)*, *Cover(s)*,*Within*, *Equal(s)* etc. It is easy to see that the incremental optimization strategies can be applied to any of these predicates. The *spoiler* condition and Algorithm 3 can be easily adapted with minimum modification.

OBSERVATION 2. *Semantics-based optimization can be applied to most spatial predicates and some spatial measurement functions.*

By examining the random sampling approach in processing Query 1, we may notice that semantics-based optimization can be extended to many other spatial predicates, such as *Intersect(s)*, *Contain(s)*, *Disjoint*, *Touch(es)*,*Overlap(s)*, *Cover(s)*, *Equal(s)* etc. For example, if we find a core/border point outside the static data set, we can optimize the queries to evaluate *Contain(s)*, *Touch(es)* or *Equal(s)* conditions similarly.

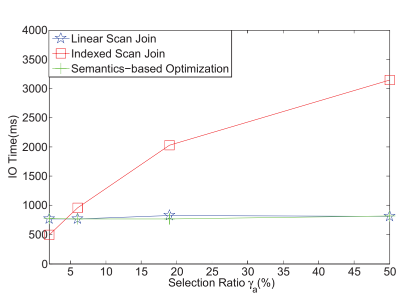
Spatial measurement functions evaluate certain properties of spatial objects (such as distance, area etc.), and return floating values as the output. Spatial measurement functions usually appear in the "SELECT" statement as aggregate functions. The semantics-based optimization can be extended to these functions, but requires careful design of algorithms. For example, to calculate the area of flooding in query Q2, we may use density and average distance information of the points to get the approximate answers. Now the queue Q in Algorithm 2 becomes a sequence of *real* numbers (instead of *booleans*), and an additional step needs to be added to produce the aggregated result over current window w from Q. We plan to address our strategies systematically by examining *EVERY* OpenGIS defined spatial predicates and spatial measurement functions in our future work.

4. EXPERIMENTS

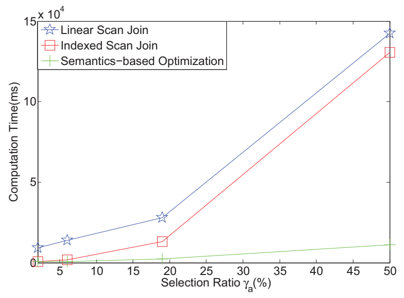
In this section, we will present our experiment results to demonstrate the effectiveness of our proposed query semantics and efficiency of the optimization strategies.

4.1 Experiment Settings

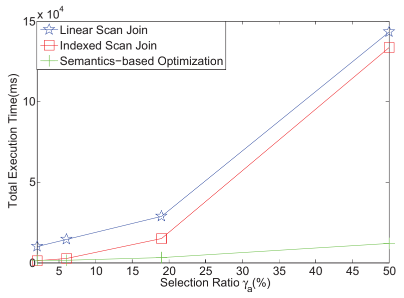
The geo-stream data used in our experiments is a modified version of climate data archives provided by University of Delaware Center for Climatic Research [25]. The archives contain grid-based temperature and precipitation data with 0.5-degree latitude-longitude resolution. Each grid may be considered a spatial point data type. We have chosen a subset of the precipitation data that covers a spatial area across the United States. (Longitude: $-120 \sim -75$ Latitude: $30 \sim 50$) and temporal period of year 1999. The stream is fed into the memory with a schema of (precipitation: *sreal*, extent: *point*). Figure 5 shows the statistics of original streaming precipitation data. Most points' precipitation is within the range of $[0, 20]$. The average precipitation is $54mm$. For the static spatial relation, we use a table *state* that contains all the states in the U.S.A. generated from a shape file provided with ESRI ArcGIS software. It is easy to see that example query Q1 can be directly mapped to this dataset.



(a) IO Time



(b) Computation Time



(c) Total Execution Time

Figure 4: Effect of Semantics-based Optimization on Query Q1

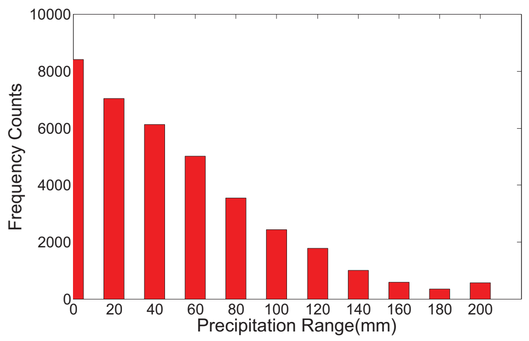


Figure 5: Statistics of Original Geo-stream

We measure both the (in memory) computation time and IO time (which is proportional to the number of IO block transfers) for queries over window size $w = 5$. The results are generated for different selection ratio γ_a to test the scalability of the methods. In measuring the IO time, we use "LIMIT n" statement to simulate the memory buffer used to load the static dataset. Typically the block size is $2KB - 4KB$. Based on the size of the static relation *state*, we set $n = 1$ to best simulate the size of one block, which means each time only one tuple in the *state* relation is loaded into memory.

4.2 Experimental Results

4.2.1 Effect of Semantics-based Optimization

Figure 4 shows the effect of semantics-based optimization on IO time, computation time and total execution time over one sliding window. The window covers 5 time slices (i.e. window size $w = 5$) instead of 2 time slices as in Q1. The semantics is the same as query Q1. As demonstrated by Figure 4 (a), both linear scan join and semantics-based optimization have stable IO time regardless of selection ratio of S, while the IO time for indexed scan join is near linear to selection ratio. When γ_a is high(50%), the IO time for indexed scan join is nearly 6 times of that of other two methods. This can be explained by the fact that the static relation R is always scanned once for the current window in the other two methods, while R may be scanned multiple times in the indexed scan join depending on the number of regions in S_r.

Figure 4 (b), on the other hand, demonstrates the effect of semantics-based query optimization on computation time for query Q1. As can be observed from the figure, the semantics-based optimization can reduce the computation time substantially. When the selection ratio is high, it can save more than an order of magnitude of computation cost. The nice scalability of the optimization is due to the fact that clustering and polygonization have high computation cost in the naive query processing.

From Figure 4 (a) and (b), we may also observe that IO time is a less dominant factor in our experiments. Therefore the total execution time of semantics-based optimization is also significantly smaller than the naive query processing. As shown in Figure 4 (c), depending on the selection ratio, the optimization can save the execution time by 2 to more than 10 times.

4.2.2 Effect of Incremental Optimization

To evaluate the effect of incremental optimization, we perform the queries Q1 that covers 5 time slices for each window. We also let the current window continuously slide until the current time moves 7 time snapshots ahead. The semantics-based optimization has been built into both non-incremental and incremental cases.

As shown by Figure 6 (a), incremental optimization can substantially reduce the IO time. When selection ratio is relatively low, the IO time can be reduced more than 50%.

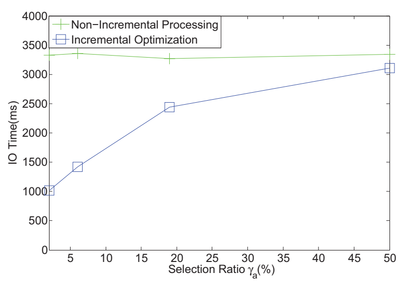
The incremental optimization can also improve computation efficiency on the basis of semantics-based optimization. As shown in Figure 6 (b), the computation time can be reduced by more than 5 times when selection ratio is low. The total execution time is also significantly reduced by up to 5 times, shown in Figure 6 (c).

Combining semantics-based and incremental optimization, our strategy can reduce the total execution time by more than an order of magnitude.

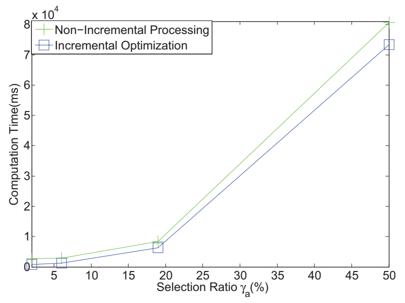
5. RELATED WORK

Traditional database management systems are inadequate in processing complicated queries over continuous data streams. New processing paradigms and methods have been proposed and implemented in several streaming data management systems [6, 4, 1] to achieve common objectives. However, these systems treat each stream as the unit of operation. Aggregating points into high-level phenomena is not considered. Furthermore, there is limited support for spatial data types, such as evolving regions, and queries involving these types.

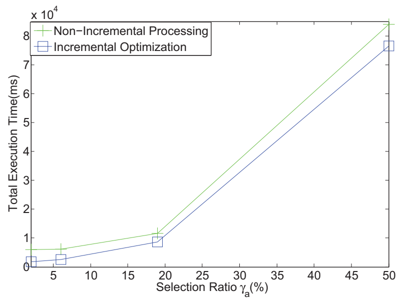
A large body of research has been devoted to moving object databases [26, 22, 21, 13, 19]. Many efficient index-



(a) IO Time: Query Q1



(b) Computation Time: Query Q1



(c) Total Execution Time: Query Q1

Figure 6: Effect of Incremental Optimization

ing methods have been proposed. For example, a time-parameterized R-Tree(TPR Tree) is proposed in [26] to index the positions of moving objects whose moving pattern can be described by a function. The TPR Tree allows for efficient query of the current and future projected locations of the linearly moving objects. Indexing of moving objects with non-linear motion patterns is addressed in [24], where server-level coarse indexes and client-level refined indexes are combined to incorporate predictive queries over non-linear motion patterns of the objects. Query processing and optimization algorithms for continuous query [21, 22, 20, 19], mainly range queries and nearest neighbor queries, have been proposed. In SCUBA [22], moving micro-clusters are utilized for pre-filtering in order to reduce unnecessary spatial joins and perform intelligent load shedding. Our work is different for the following reasons: (1) Our data comes from static geo-sensor networks monitoring spatially and temporally continuous phenomena, not from moving objects; (2) in moving object databases, queries typically originate from a moving object and are mostly moving window queries or nearest neighbor queries. In our context, queries do not come from moving object and join queries are common. As a result, some indexing and query processing strategies designed for queries from moving points are difficult to be applied directly. For example, the SCUBA system's optimization strategy is not applicable in our scenario because there are no individual queries within the micro-cluster; (3) most indexing algorithms assume that positions of moving objects are continuously materialized into disks in some way. We assume the recent geo-streams are memory resident as in *SINA* [20] (in-memory hash tables to incrementally evaluate) and *SOLE* [19] (uses memory-based algorithms to perform online operations).

Work in processing streaming images, such as [10], attempts to treat an image as a complex data type and introduces map/image algebra operations to allow users to manipulate streaming images. Supporting queries with extended spatio-temporal extent in real time were not addressed in the work. Spatial-temporal database systems such as Secondo [12], and Dedale [11], have been built with various levels of support for spatio-temporal data. However, supporting geo-streams has not been addressed.

Our work is also related to spatial clustering and polygonization. Spatial clustering is a well studied area in data mining. However, supporting spatial clustering through database management systems and query language has only been explored recently. Our previous work [29] and [17] recognize the inadequateness of aggregation using GROUP BY in SQL and suggest using CLUSTER BY in static database systems. However, efficient implementation algorithms for continuous queries were not addressed. The elevation of a cluster of points into spatial regions is non-trivial without previous knowledge of cluster boundary information. In [15], average length of all edges was used as the threshold to eliminate edges in a triangulation for polygonization. This approach is for visualization purposes only and does not compute the sequence of the vertices that constitute the resultant polygons. We introduce an approach to elevate discrete spatial data points into higher-level spatial objects and support continuous queries on them (with or without explicit polygonization).

Our work is further related to sensor network databases. The feasibility of abstracting a sensor network as a database has been documented and prototyped in pioneer sensor database systems, notably Cougar [28] and TinyDB [18]. These systems have the same interface as desktop or server databases through standard query languages, e.g. SQL. Many efforts have been dedicated to extending the lifetime of a sensor network. The recent work [8, 27] recognizes the mismatch of discrete sensor readings with high-level phenomena and tries to bridge them. Worboys [27] *et. al.* proposes a framework for detection of global high-level events based on discrete local sensor readings. MauveDB [8] proposes using interpolation to bridge the discrete reading from geo-sensors with the continuous phenomena that a sensor network monitors to deal with the inherent noisy and unreliable underlying sensor networks. However, none of this work allows further abstraction of the phenomena into spatial objects such as flooding zones, nor does it allow real time queries on them.

6. SUMMARY AND FUTURE DIRECTIONS

This paper focuses on one important type of geo-streaming data - point geo-streams. Querying streaming point clusters as regions directly in the geo-stream database has many benefits, but is also very challenging. We proposed two query optimization strategies, namely *semantics-based optimization* and *incremental optimization* for answering queries involving point geo-streams and static data set. The experimental results on a streaming meteorological dataset demonstrate the effectiveness and the efficiency of the query processing techniques.

As an extension to this paper, we plan to work on query processing for different types of queries systematically. We also plan to extend our work to handle more complicated queries involving multiple geo-streams and/or static spatial relations. One opportunity in multiple query optimization is to exploit the similarities of two queries in terms of spatial coverage. If we can build a tree of the queries based on their spatial coverage, the result for an intermediate query node may be computed from the results of its children query nodes. Another possibility is to observe the moving models of the streaming regions and perform pre-computation for future time slices when the query load is not at peak state. Finally, as an integral part of the system, query processing on moving sensors will also be addressed.

7. ACKNOWLEDGEMENTS

This work is supported by the National Science Foundation under Grant No. IIS-1017926 and CNS-0709285. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

8. REFERENCES

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2), 2003.

[2] Y. Ahmad, B. Berg, U. Cetintemel, M. Humphrey, J.-H. Hwang, A. Jhingran, A. Maskey, O. Papaemmanouil, A. Rasin, N. Tatbul, W. Xing, Y. Xing, and S. Zdonik. Distributed operation in the borealis stream processing engine. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 882–884. ACM, 2005.

[3] M. H. Ali, W. G. Aref, and C. Nita-Rotaru. Spass: Scalable and energy-efficient data acquisition in sensor databases. In *Proceedings of the International ACM Workshop on Data Engineering for Wireless and Mobile Access (MobiDE)*, 2005.

[4] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. Stream: the stanford stream data manager. In *SIGMOD*, 2003.

[5] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: continuous dataflow processing. In *SIGMOD '03*, pages 668–668. ACM, 2003.

[6] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: continuous dataflow processing. In *SIGMOD*, 2003.

[7] J. Considine, F. Li, G. Kollios, and J. Byers. Approximate aggregation techniques for sensor databases. In *Proceedings of the 20th International Conference on Data Engineering*, 2004.

[8] A. Deshpande and S. Madden. Mauvedb: supporting model-based user views in database systems. In *SIGMOD*, 2006.

[9] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, 1996.

[10] M. Gertz, Q. Hart, C. Rueda, S. Singhal, and J. Zhang. A data and query model for streaming geospatial image data. In *EDBT*, 2006.

[11] S. Grumbach, P. Rigaux, and L. Segoufin. The dedale system for complex spatial queries. In *SIGMOD*, 1998.

[12] R. H. Güting, V. T. de Almeida, D. Ansorge, T. Behr, Z. Ding, T. Höse, F. Hoffmann, M. Spiekermann, and U. Telle. Secondo: An extensible dbms platform for research prototyping and teaching. In *ICDE*, 2005.

[13] R. H. Guting and M. Schneider. *Moving Objects Databases*. Morgan Kaufmann, 2005.

[14] Y. Huang and C. Zhang. New data types and operations to support geo-streams. In *GIScience*, 2008.

[15] I. Kolingerová and B. alik. Reconstructing domain boundaries within a given set of points, using delaunay triangulation. *Computers and Geosciences*, 32, 2006.

[16] Y. Kotidis. Snapshot queries: Towards data-centric sensor networks. In *ICDE*, pages 131–142, 2005.

[17] C. Li, M. Wang, L. Lim, H. Wang, and K. C.-C. Chang. Supporting ranking and clustering as generalized order-by and group-by. In *SIGMOD*, 2007.

[18] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *SIGMOD*, 2003.

[19] M. F. Mokbel and W. G. Aref. Sole: scalable on-line execution of continuous queries on spatio-temporal data streams. *VLDB Journal*, accepted for publication, 2008.

[20] M. F. Mokbel, X. Xiong, and W. G. Aref. Sina: scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD*, 2004.

[21] K. Mouratidis and D. Papadias. Continuous nearest neighbor queries over sliding windows. *IEEE Trans. on Knowl. and Data Eng.*, 19(6), 2007.

[22] R. V. Nehme and E. A. Rundensteiner. Scuba: Scalable cluster-based algorithm for evaluating continuous spatio-temporal queries on moving objects. In *EDBT*, 2006.

[23] NOAA's National Weather Service. Coopratiive observer program. `http://www.nws.noaa.gov/om/coop`.

[24] Y. Tao, C. Faloutsos, D. Papadias, and B. Liu. Prediction and indexing of moving objects with unknown motion patterns. In *SIGMOD*, 2004.

[25] University of Delaware Center for Climatic Research. Monthly total precipitation data. `http://climate.geog.udel.edu/~climate/html\_pages/archive.html`.

[26] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. *SIGMOD Rec.*, 29(2), 2000.

[27] M. F. Worboys and M. Duckham. Monitoring qualitative spatiotemporal change for geosensor networks. *International Journal of Geographical Information Science*, 20(10), 2006.

[28] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 31(3), 2002.

[29] C. Zhang and Y. Huang. Cluster by: A new sql extension for spatial data aggregation. In *ACMGIS*, 2007.