Continuous Constraint Query Evaluation For Spatiotemporal Streams *

Marios Hadjieleftheriou[1], Nikos Mamoulis[2], and Yufei Tao[3]

[1] AT&T Labs Inc., 180 Park Avenue, Florham Park, NJ 07932,
marioh@research.att.com
[2] Department of Computer Science, University of Hong Kong, Pokfulam Road, Hong Kong,
nikos@cs.hku.hk
[3] Department of Computer Science and Engineering, Chinese University of Hong Kong, Sha Tin, New Territories, Hong Kong, taoyf@cse.cuhk.edu.hk

Abstract. In this paper we study the evaluation of *continuous constraint queries* (CCQs) for spatiotemporal streams. A CCQ triggers an alert whenever a configuration of constraints between streaming events in space and time are satisfied. Consider, for instance, a server that receives updates from GPS-enabled agents that report their positions and other measurements (e.g., environmental readings). An example of CCQ is: "Alert whenever at least 5 readings closer than 5km to each other and within a time difference of 5 minutes report high pressures and low temperatures". We model CCQs as Constraint Satisfaction Problems (CSPs) and develop solutions for their *continuous evaluation*. Our techniques (1) consider the fast arrival rate of incoming events, and (2) minimize the memory requirements, without using predefined window constraints, but by utilizing the structure of the queries. In order to show the merits of the proposed techniques, we implement a system prototype and evaluate it with real data.

1 Introduction

The recent advances in telecommunications have made it possible to collect unbounded streams of spatiotemporal information from various sources (GPS devices, sensors, etc.). Consider, for instance, a server which receives updates from GPS-enabled agents that continuously report their positions and other measurements (e.g., environmental readings). A real example of such a system is the Global Drifter Center [2] where a large number of buoys have been deployed in oceans all around the world. The buoys report various measurements at regular time-intervals in a streaming fashion while drifting in the water according to sea currents.

The huge size and fast arrival rate of such information renders its storage and off-line analysis infeasible. In addition, users are often interested in answering queries in an on-line, dynamic manner, as streaming data arrive. In this paper, we study the processing of *continuous constraint queries* (CCQs), which trigger an alert whenever a configuration of *constraints* between streaming events in space and time are satisfied. For instance, consider the following query: "Alert whenever at least 5 readings closer

* Supported by grant HKU 7160/05E from Hong Kong RGC.

than 5 km to each other and within a time difference of 5 minutes report high pressures and low temperatures". CCQs facilitate the automatic and continuous monitoring of interesting combinations of spatiotemporal events.

There is an abundance of interesting and useful queries that can be formulated as a combination of diverse types of constraints. The constraints may capture both spatial (e.g., proximity, intersection, and containment) and temporal (e.g., during, before, and after) relationships between a large number of events, as well as to other event characteristics (e.g., measurements like velocity and temperature).

CCQs are similar to traditional triggers in database systems, in that they should be evaluated every time a new event arrives. A CCQ states that if the new event forms a specific spatiotemporal configuration (e.g., a number of abnormal thermal indications in space and time) with past events, an alert should be triggered. The important difference to traditional triggers is that the constraints are spatiotemporal. First, the methods for evaluating them are related to spatial and spatiotemporal query processing. Second, typically, there is no need to keep the whole history of events in memory, since the spatiotemporal constraints define the essential intervals in time and space, for which information needs to be maintained. For instance, for a CCQ asking for a number of abnormal thermal indications within 10 minutes in time, we need not keep events in memory older than 10 minutes, since they could not form query results with current or future data.

In this paper we present a system with the following characteristics: (1) a stream of events arrives on a central server at fast rates; (2) events are associated with spatiotemporal properties, and other, alphanumeric measurements; (3) users register CCQs, and (4) the system trigger alerts whenever a newly arriving event together with past events form a result of a CCQ.

Our main focus is on critical applications where it is essential not to dismiss any query alerts and, in addition, not to produce any false alarms. Hence, we propose techniques that produce exact query results, raising alerts if and only if a combination of events satisfies all constraints for a given query. This is accomplished by guaranteeing that all useful events (the ones that can contribute to a query answer) are stored in main memory during processing. Since storing the complete event stream is not feasible for most practical applications, we introduce algorithms for determining *event expiration times* — computed by inspecting all query constraints related to a specific event — and establishing a time-instant after which the event will not possibly satisfy any constraints and can be deleted. We introduce a simple expiration time computation algorithm, as well as a tighter technique that guarantees that every event is kept in main memory for a very short amount of time. With this approach we minimize the peak amount of main memory that is consumed by the system. Finally, to show the merits of our architecture, we implement and evaluate a prototype for the proposed system.

2 Problem Formulation and Definitions

This section introduces a formal problem statement and some necessary definitions to simplify our analysis. As already discussed, a stream of events arrives on a central server where each event carries a timestamp (or a time interval), spatial (i.e., geometric) prop-

erties, and other properties of a simple type (e.g., alphanumeric). Users register queries that pose various constraints between properties of a possibly large number of events. The goal is to find in real-time tuples of event instances that have already appeared on the stream and satisfy all query constraints, in which case an alert is triggered. First, we formally define an event instance. Then, we present a formal way of expressing generic constraint queries between events. Finally, we discuss algorithms for evaluating such queries continuously, as new events arrive on the stream.

A spatiotemporal stream is a never ending sequence of events $\mathcal{S} = \langle e_1\ e_2\ \cdots\ e_n\ \cdots \rangle$ where:

Definition 1. *An event e is a tuple $\{t, r, p\}$, $e.t$ are the temporal properties of the event (e.g., a time-interval or the arrival time), $e.r$ the geometric properties (e.g., $e.r \in \mathbb{R}^d$ for a d-dimensional point), and $e.p$ a set of application dependent properties (e.g., temperature, velocity, other measurements).*

Table 1. Notation used throughout the paper

Symbol	Meaning
e	Streaming event
t	Time-instant or time-interval
r	Geometric properties
p	Generic properties
$\odot$	Spatial predicate
$V, \mathcal{V}$	Variable, set of variables
D	Variable domain
$C, \mathcal{C}$	Constraint, set of constraints
$\mathcal{G} = (\mathcal{V}, \mathcal{C})$	Constraint graph
$\{e_i, e_j\} \propto C$	Tuple $\{e_i, e_j\}$ satisfies constraint C

We can naturally express queries with arbitrary constraints between a large number of events as *Constraint Satisfaction Problems* (CSP) [32, 3, 13, 15]. Constraint satisfaction is a paradigm that can capture a wide variety of applications from AI, engineering, databases, and other disciplines. A CSP is defined by a set of variables, each of which has a finite set of potential values, and a set of constraints between these variables. CSPs that contain constraints between at most two variables are called *binary*. A binary CSP is usually represented by a *Constraint Graph* (CG) where nodes correspond to variables and edges correspond to constraints. The basic goal in a CSP is to find one or all assignments of values to variables so that all constraints are satisfied. In our setting, the potential values of a given variable are event instances that have appeared on the stream, and the binary constraints are the spatial, temporal, and other possible constraints between events. More formally:

Definition 2. *A binary Constraint Satisfaction Problem is:*

1. *A set of variables $\mathcal{V} = \{V_1, \ldots, V_m\}$.*

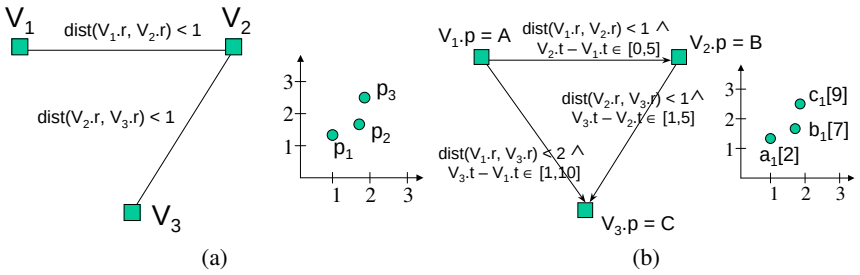

Fig. 1. Examples of Constraint Graphs

2. *For each V_i, a finite domain $D_i = \{e^i_1, \ldots, e^i_{k_i}\}$ of k_i possible values.*
3. *A set of binary constraints. A constraint $C_{i,j}$ is a relation of permitted values for the pair of variables $\{V_i, V_j\}$.*

We say that an assignment of values $\{V_i = e^i_l, V_j = e^j_k\}$ satisfies constraint $C_{i,j}$ if $\{e^i_l, e^j_k\}$ is in the relation $C_{i,j}$. A solution *to a CSP is an assignment $\{V_1 = e^1_{s_1}, \ldots, V_m = e^m_{s_m}, \}$ such that for all $1 \leq i, j \leq m$, constraint $C_{i,j}$ is satisfied.*

Figure 1a illustrates a CG corresponding to a simple CSP with the domain of each variable being a collection of events (whose spatial properties are 2D point locations) and constraints between variables being 'distance within 1 space unit'. Note that the constraints in this graph are not expressed explicitly by allowed pairs of values, but implicitly with the use of spatial predicates. A solution to this CSP is the triplet $\{V_1 = p_1, V_2 = p_2, V_3 = p_3\}$ of points shown next to the figure.

In our setting, the domain of each variable (e.g., the set of events that correspond to variable V_i) can be implicitly defined by *unary* constraints, which correspond to selections in database languages. Thus, a unary constraint restricts a variable's domain only to those events that qualify predicates on certain event properties, e.g., "events with measured temperature greater than 30°C". Selection predicates can range from simple relational operators (e.g., $\leq, <, \geq, >, =$) to more advanced selection conditions (e.g., "select events with property A") or *metric* spatiotemporal constraints (i.e., relating $e.r$ and $e.t$ to fixed coordinates of space and time). In the rest, we use notation $e^i_l \propto C_i$ to denote that event e^i_l satisfies unary constraint C_i of variable V_i.

As in our spatial CSP example, we can implicitly define binary constraints by spatial, temporal, and/or any other predicates between event properties $e.p$. Thus, in our setting, we can specialize the definition of constraints as follows:

Definition 3. *A binary constraint $C_{i,j}$ between variables $\{V_i, V_j\}$ is a tuple $\{t, \odot, p\}$, where t is a temporal predicate (e.g., a time-interval), $\odot$ is any binary spatial predicate (for example intersection or proximity), and p any non spatiotemporal constraint between properties $e.p$.*

A binary constraint $C_{i,j}$ is satisfied only when the assignment $\{V_i = e^i_l, V_j = e^j_k\}$ satisfies all $C_{i,j}.t$, $C_{i,j}.\odot$, and $C_{i,j}.p$. Notice that the temporal predicate semantics are application specific. In the simplest case, $e.t$ can be a time-instant and $C_{i,j}.t$ can be a time-interval containment (e.g., $e^i_l.t, e^j_k.t \in \mathbb{R}$ and $C_{i,j}.t = \{e^j_k.t - e^i_l.t \in [t_{lb}, t_{ub}]\}$ but

more general semantics can also be considered. Without loss of generality, we restrict our analysis to time-interval temporal constraints only. Also, for convenience, in the rest we use notation $\{e_l^i, e_k^j\} \propto C_{i,j}$ to denote that assignment $\{e_l^i, e_k^j\}$ satisfies binary constraint $C_{i,j}$. In addition, we allow negative temporal values as well, which can express chronological precedence between events — negative values refer to the past, while positive refer to the future. For example, time-interval $[-3, 4]$ on edge (V_i, V_j) means that $\{e_l^i, e_k^j\}$ satisfies the temporal constraint if e_k^j arrives 0–4 time-instants *after* or 0–3 time-instants *before* e_l^i. This generalization is useful since it helps express inferred constraints between events: Constraint $[t_1, t_2]$ on edge (V_i, V_j) implies an *inverse* constraint $[-t_2, -t_1]$ between (V_j, V_i).

Constraint inference facilitates the derivation of complete constraint graphs where all possible constraints $C_{i,j}$ are specified (i.e., cliques). In general, user queries may not provide such complete information (e.g., the graph of Figure 1a). Nevertheless, any partial query graph can be converted into a complete graph using spatial and temporal inference rules like inversion, composition and intersection [7, 21, 18]. For instance, *dist*$(V_1.r, V_2.r) < 1$ and *dist*$(V_2.r, V_3.r) < 1$ imply that *dist*$(V_1.r, V_3.r) < 2$, assuming that the geometric properties $e.r$ are points. Such CG transformations are useful for several reasons: (1) for identifying if a query is unsatisfiable by discovering negative cycles; such queries can be ignored, (2) in order to tighten existing constraints and make them more selective, and (3) to simplify the proposed algorithms and in some cases improve query evaluation performance (as will become clear in later sections). The interested reader can refer to [7, 21, 18] for more details on temporal and spatial inference, which are beyond the scope of this paper.

Figure 1b shows a complete CG with spatiotemporal constraints (inverse edges are omitted for clarity). Nodes are annotated with unary constraints, which are selections on the non-spatial properties for simplicity. In practice $V_1.p$ = A could be $V_1.temperature > 30°$C. We denote events, for which $e.p$ = A by a_1, a_2, etc. Thus the event a_1 on the right of Figure 1b has $a_1.p$ = A, the event b_1 has $b_1.p$ = B, etc. The *continuous constraint query* (CCQ) that corresponds to the CG triggers an alert if any event a_i is close to an event b_j which arrives later and at most 5 time-instants after a_i (i.e., $C_{1,2}.t = [0, 5]$), and b_j is close to an event c_k, which arrives at least 1 and at most 5 time-instants after b_j (i.e., $C_{2,3}.t = [1, 5]$, and, equivalently, the inverse constraint $C_{3,2}.t = [-5, -1]$ means that event b_j arrived at least 1 and at most 5 time-instants before c_k). These two constraints also infer that c_k must arrive at least 1 and at most 10 time-instants after a_i, illustrated by the inferred temporal constraint between variables V_1 and V_3.

An extended SQL can be used to express CCQs. For instance, our example can be expressed in pseudo-SQL with spatial and temporal predicates, as follows:

```
CREATE TRIGGER collision
FOR E as V1, E as V2, E as V3
WHEN V1.p = A AND V2.p = B AND V3.p = C
AND DISTANCE(V1.r, V2.r)<1 AND V2.t-V1.t IN [0, 5]
AND DISTANCE(V2.r, V3.r)<1 AND V3.t-V2.t IN [1, 5]
```

The continuous constraint query evaluation problem can be formulated as follows:
Problem Statement: Given a number of constraint queries, continuously evaluate their

satisfiability as new events appear on the stream, and trigger alerts whenever a combination of events constitutes a solution.

Previous methods on processing complex spatiotemporal queries with multiple selections and joins employ spatiotemporal indexes in combination with constraint satisfaction algorithms [23, 20]. Our problem is different in that data arrive continuously (instead of being stored to a database in advance) and that the queries should be continuously evaluated. In other words, the domains of the constrained variables are not static, but dynamically change over time. Therefore, CCQs fall into the class of queries described in [6] and special methods are required for them. In addition, our problem is different than handling traditional triggers in DBMSs [12], since CCQs need not be evaluated over the whole (past) time horizon; a query refers to the interval back to the oldest event in it. As a result, the whole events history needs not be maintained, but we can apply rules that minimize the required space, to be discussed shortly. In the next section, we describe our system prototype for registering and evaluating CCQs over continuous data streams.

3 System Architecture

The problem formulation presented in the previous section raises a number of interesting problems. We need to engineer appropriate data structures and algorithms for evaluating CSPs in real-time. In addition, we need to evaluate queries incrementally every time a new event appears on the stream by restricting the domain of each variable in the given CGs, to speed up execution. Finally, we must continuously identify and delete from the system events that cannot possibly belong to future solutions, in order to minimize main memory requirements.

For simplicity and clarity we make the following assumptions: First, we consider only transitory events — the information associated with each event is valid only for a specific time-instant or time-interval and the properties $e.p$ (i.e., the associated measurements) of an event remain static throughout its interval. In addition, we restrict our analysis only to streams of events with measurements that appear in chronological order. In other words, the events arrive in the same order as their temporal properties define. If this is not the case, then a large enough buffer could be used in order to rank events according to their arrival time-instants first. The size of the buffer can be determined according to the maximum expected delay of an arrival, which in most cases is a system characteristic. Finally, we assume that all registered query CGs have been transformed into complete constraint graphs. A variation of the Floyd-Warshall algorithm as it appears in [21] can be used for temporal constraints. The same algorithm can also be applied for spatial constraints, using the inference rules of [18].

3.1 CCQ evaluation

We first discuss the basic algorithm used to continuously evaluate constraint queries, in order to identify the operations that must be supported by our system. When a new event e arrives, it can be part of a variable domain in a CCQ, and therefore part of a potential solution. Thus, the first thing to do is to check the unary constraints of all variables of

all registered queries, in order to identify the CG, for which e could participate in a solution. Let V_i be such a variable in a query corresponding to CG $\mathcal{G}$ (i.e., $e \propto C_i$ in $\mathcal{G}$). V_i is connected with binary constraints to all other variables in $\mathcal{G}$ and these constraints refer either to the future, or to the past, or in both directions. For example, V_2 in the CCQ of Figure 1b is connected to a past variable V_1 (since $C_{2,1}.t = [-5, 0]$) and a future variable V_3 (since $C_{2,3}.t = [1, 5]$).

Let $\mathcal{V}_i^-$ and $\mathcal{V}_i^+$, be the sets of past and future variables to V_i. Note that a variable can be in both $\mathcal{V}_i^-$ and $\mathcal{V}_i^+$. If the subgraph $\mathcal{G}'$ containing $V_i = e$ and $\mathcal{V}_i^-$ is satisfiable, e is interesting for two reasons. First, e triggers an alert if $V_i \cup \mathcal{V}_i^- = \mathcal{V}$ (i.e., $\mathcal{G}' = \mathcal{G}$). For example, variable V_3 in the graph of Figure 1b has past variables only (i.e., $V_3 \cup \mathcal{V}_3^- = \mathcal{V}$). Thus, an event e with $e.p =$ C may trigger an alert if there exist past events consistent to e and the subgraph $\mathcal{G}'$, containing V_1, V_2 and their constraints. Second, if $\mathcal{V}_i^+ \neq \varnothing$, it is possible for e to participate in an alert in the future when variables in $\mathcal{V}_i^+$ get values from events that are consistent with $\mathcal{G}'$. For example, variable V_2 in the graph of Figure 1b has a future variable (V_3). Thus, an event e with $e.p =$ B, which makes the subgraph containing V_1 and V_2 satisfiable, can participate in an alert with some future event that instantiates V_3.

Thus, if $\mathcal{G}'$ is satisfiable and $\mathcal{V}_i^+ \neq \varnothing$, we say that e is *useful* and we keep it in the system for potential future inclusion in an alert. The question is now *how long* is it essential to keep e. We assign e an *expiration* time $e.X$, after which e becomes obsolete and should be deleted. Intuitively, the expiration time cannot be longer than the longest interval length of temporal constraints that use e.

Algorithm 1 summarizes the procedure for handling a new event e that arrives in the system. Initially, the expiration time is set to its time-instant $e.t$; if the event is not found useful by the algorithm, it will be immediately deleted. All relevant CG and variables to e are then retrieved. A variable V_i in a CG $\mathcal{G}$ is relevant if $e \propto C_i$, as already discussed. For each such variable, we find the set $\mathcal{V}_i^-$ of past only variables and compute their domains according to the past events stored in the system. The domain of a variable $V_j \in \mathcal{V}_i^-$ is determined such that (1) its values are consistent with the assignment $V_i = e$ (for this, constraint $C_{i,j}$ is used), and (2) they are consistent with C_j, i.e., the unary constraint of V_j. If a domain of a variable is empty, we know that the assignment $V_i = e$ for query $\mathcal{G}$ is inconsistent. Otherwise, we solve the CSP for graph $\mathcal{G}$ only if all variables in $\mathcal{G}.V$ are consistent with e (and thus if V_i is chronologically last). If a solution is found (line 12), the algorithm generates an alert. Finally, the expiration time of e is updated accordingly, if V_i has any future variables ($\mathcal{V}_i^+ \neq \varnothing$) and if the past only variables are consistent with V_i. Otherwise, the event can be deleted, since these variables can never be satisfied. Algorithm 1 sets the expiration time as the maximum required by outgoing edges to future variables (i.e., the maximum upper bound of the temporal constraints linking the current variable with future variables). If there are many queries or variables that may use e in the future, the expiration time is the maximum timestamp determined by all of them. Later, we will discuss alternative policies that result in tighter expiration times and reduce the memory requirements. In addition, we will discuss the details for storing and indexing events, computing variable domains, solving constraint graphs, and managing event expirations.

Algorithm 1 HandleNewEvent(e: event, $\mathcal{Q}$: queries)

1: $e.X := e.t$;
2: Get all $(V_i, \mathcal{G})$ pairs such that $\mathcal{G} \in \mathcal{Q}, V_i \in \mathcal{G}, e \propto C_i$;
3: **for** each $(V_i, \mathcal{G})$ **do**
4: $V_i.sat :=$ true;
5: **for** each $V_j \in \mathcal{V}_i^-$ **do**
6: use $V_i = e$, $C_{i,j}$, and C_j to compute D_j;
7: **if** $D_j = \varnothing$ **then**
8: $V_i.sat :=$ false;
9: **break**; ▷ break V_j for-loop
10: **if** $V_i.sat$ and $\mathcal{V}_i^- \cup V_i = \mathcal{G}.\mathcal{V}$ **then**
11: Solve $\mathcal{G}$
12: **if** $\mathcal{G}$ is satisfiable **then alert** solution
13: **if** $\mathcal{V}_i^+ \neq \varnothing$ and $\forall V_j \in \mathcal{V}_i^- / \mathcal{V}_i^+ : D_j \neq \varnothing$ **then**
14: $e.X := \max\{e.X, e.t + \max_{\forall V_j \in \mathcal{V}_i^+}(C_{i,j}.t_{ub})\}$;
15: **if** $e.X > e.t$ **then** Store(e); ▷ event is useful

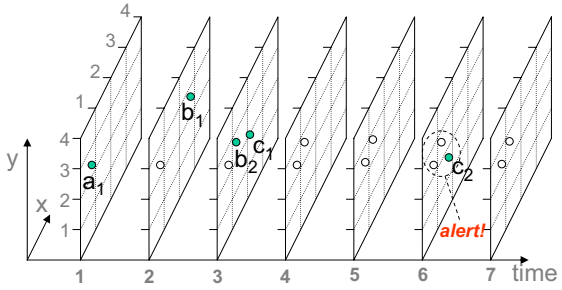

Fig. 2. A continuous query evaluation example

3.2 An example

In this section we present a simple example that clarifies the functionality of Algorithm 1. Let us assume that only the CCQ shown in Figure 1b has been registered with the system. Suppose that the arrivals on the stream follow the sequence shown in Figure 2. Event a_1 (for which $a_1.p = \text{A}$) arrives at time-instant 1 and $a_1 \propto C_1$. The event should be stored as a future candidate (i.e., it qualifies the unary constraint of query variable V_1). An *expiration time* of 11 (due to $C_{1,3}.t = [0, 10]$) is assigned to a_1, after which a_1 will be deleted from main memory in order to save space.

Next, event b_1 appears at time-instant 2. Variable V_2 is related to b_1, which has V_1 as past variable. D_1 becomes $\varnothing$ at line 6 of the algorithm, because a_1 (the only stored event that satisfies C_1) is further than 1 spatial units from b_1 ($C_{2,1}.\odot$ is violated). $V_1.sat$ becomes false, implying that b_1 can never participate in a query alert; the temporal predicate $C_{2,1}.t = [-5, 0]$ refers only to the past and no events with property A close to b_1 have arrived already. Thus, b_1 will directly be deleted from the system (i.e., $b_1.X$ remains $b_1.t = 2$). At the next time-instant, events b_2 and c_1 arrive simultaneously. This time $\text{dist}(a_1, b_2) < 1$ and $\text{dist}(b_2, c_1) < 1$, however, c_1 violates the temporal constraint $C_{3,2}.t = [-5, -1]$ and it can be deleted. Nevertheless, b_2 can potentially belong to a

future solution, hence, it needs to be retained and its expiration time becomes 8, given the current time 3 and the temporal constraint $C_{2,3}.t = [1, 5]$.

Finally, at time-instant 6 event c_2 arrives, and since $\text{dist}(c_2, b_2) < 1$ and a_1, b_2 have not expired yet, an alert is triggered with solution tuple $\{a_1, b_2, c_2\}$. Then c_2 is deleted (V_3 has no future variables). On the other hand, events a_1 and b_2 need to be retained until their expiration times, since more events containing property C might appear in the stream, triggering additional alerts. After that time both events can be deleted.

3.3 A Detailed Analysis of the Proposed Framework

We now describe in detail the components of our system prototype that manages incoming events and evaluates CCQs based on Algorithm 1. Our system prototype consists of five basic components, shown schematically in Figure 3.

Queries are stored in memory-based constraint graph representations. Given a new event e on the stream, the Query Index retrieves all queries that contain at least one variable with an associated unary constraint that is satisfied by e. An important component is the Spatiotemporal Index, which is used for storing useful past event instances. Given a new event e and the spatiotemporal constraints associated with some related variable, the index is probed and all past events e' that qualify for these constraints given e, are retrieved. This will help populate all CG variable domains fast, with only a few candidates, instantly pruning a large number of unrelated events. An Expiration Time Array indexes events according to the time they should be deleted from the system and enables efficient deletion from the Spatiotemporal Index. Finally, a CSP Solver solves CGs given appropriate variable domains.

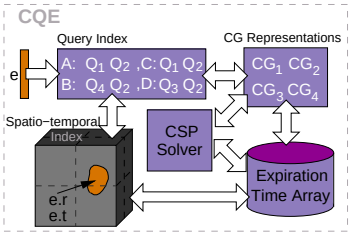

Fig. 3. A system prototype.

Constraint Graph Representation Given query Q with a complete directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{C}), |\mathcal{V}| = m$ we represent the graph using a 2-dimensional matrix M such that $M[i, j] = C_{i,j}, 1 \leq i, j \leq m$.

Query Index Queries contain a number of unary variable predicates, each one corresponding to a number of properties. Given a large number of queries, we must locate efficiently the ones that contain unary constraints, satisfied by newly arriving events; these are the only queries that have to be considered for further processing (see line 2 of Algorithm 1). Many predicate indices have appeared in the literature [19, 14, 33,

12, 29]. Although these techniques can also be applied straightforwardly, here we opt for a simpler approach with small memory requirements. In order to efficiently identify all variables V_i related to an incoming event e, we maintain a hash-table indexed by all known properties appearing in the unary predicates of registered queries. The data entries of the hash-table are pointers to the queries that have at least one variable with a unary constraint that is related to a specific property. Using the hash-table, we can locate fast all queries Q that contain a variable related to e. Then, by retrieving the unary constraint associated with that variable we can evaluate if e satisfies the constraint.

Spatiotemporal Index The basic functionality of the Spatiotemporal Index is to store all useful past events that might contribute to a CG solution (i.e., an alert) in the future (see line 15 of Algorithm 1). The index acts as a filtering step that facilitates efficient evaluation of a query by substantially limiting variable domains before evaluation (lines 5–6 of Algorithm 1). A naive algorithm would consider all events indiscretely as possible domain values. Instead, the Spatiotemporal index, given a new event e, returns only those stored events that satisfy the spatiotemporal predicates of the binary constraints associated with the past variables that are related to e. Any data or space partitioning structures can be used for that purpose, like the R-tree [10] and the Multi-Layer Grid File [27], both extended with a temporal dimension. For illustration purposes in the following discussion we assume that a 3D R-tree is used to index useful past events.

When an event e arrives, the domains of past variables that are related to $V_i = e$ are computed (lines 5–6 of Algorithm 1), as follows. For each $V_j \in \mathcal{V}_i^-$, the spatiotemporal index is probed using the spatial $e.r$ and temporal $e.t$ properties of e in combination with predicates $C_{j,i}.\odot$ and $C_{j,i}.t$. For instance, when event b_2 arrives at time instant $b_2.t = 2$ in the example of Figure 2, for constraint $C_{2,1}$ a spatiotemporal range query is applied on the index. The query has temporal extent $[-3, 2]$ (since we are looking for events at least 0 and at most 5 instants before $b_2.t$) and spatial extent a circle with radius 1 around the location of b_2. Since no events are contained in this range, $D_1 = \varnothing$ and b_2 will be found not useful and deleted from the system. In general, the results yielded by the spatiotemporal search are filtered using C_j, the unary constraint of the involved variable, before the domain D_j is finalized. This guarantees that the domains of all past variables become consistent with the assignment $V_i = e$, before solving the CSP on graph $\mathcal{G}$.

Expiration Time Array While new events arrive on the stream, older events that have been stored in the index become obsolete. A structure is required that can index events in increasing expiration time, for easy deletion of them from the system, when they expire. In Section 4.1 we will propose a technique that assigns tight expiration times to events. This technique calls for efficient operations for updating the expiration time of an event to a new value. We term this structure the Expiration Time Array.

In order to satisfy these requirements we use the following architecture (shown in Figure 4). We store all useful events in main memory and associate them with unique identifiers. In addition, we build a hash-table on the unique ids. The spatiotemporal index stores direct pointers to the events in main memory. The Expiration Time Array is another hash-table with key being the expiration time t of the events, and data entries

being arrays that contain pointers to the main memory location of the events that have expiration time equal to t.

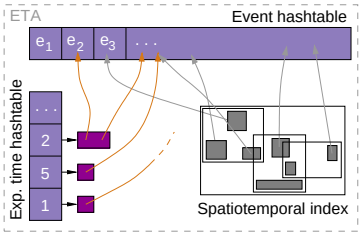

Fig. 4. The Expiration Time Array.

Assume that a new event arrives on the stream and it is inserted in the index. After the completion of the insertion operation the event is inserted in the Expiration Time Array according to the computed expiration time t. The appropriate array is located (or a new one is created if needed) and a pointer to the entry is inserted. The cost of this operation is bounded by the index insertion.

The Expiration Time Array contains possibly one array of pointers per future time-instant. Notice that the hash-table does not need to store empty arrays for time-instants that have no expiring events. So, we expect the structure to be fairly small in size. On the other hand, all arrays combined contain as many pointers as the total number of stored events.

Removing expiring events is straightforward. Every time-instant the hash-table is probed for locating the corresponding expiration array. If such an array exists, all events that it points to are accessed and deleted from the index. The deletions can happen in bulk and in a bottom-up fashion for efficiency [17]. Finally, the array is removed from the hash-table. This operation is very efficient since the main cost of this process is the update cost of the spatiotemporal index; updating the hash-table has fixed cost.

CSP Solver The CSP Solver takes as input the constraint graph $\mathcal{G}$ (line 11 of Algorithm 1) and finds a solution or shows that the graph is insoluble. The general class of CSPs is NP-complete. However, a number of algorithms have been proposed in the literature that try to efficiently evaluate CSPs [4, 8, 13, 3]. The size of all possible value that can be solutions is the Cartesian product of the variable domains. The most popular search method uses backtracking; variables are instantiated sequentially and as soon as all variables relevant to a constraint have assumed a value, the satisfiability of the constraint is tested. If a partial instantiation violates any constraint, backtracking is performed to the most recently instantiated variable that still has alternatives available. The algorithm can prune a whole subtree of the Cartesian product every time a constraint is violated.

In this work we use a variant of the backtracking algorithm, called Forward Checking (FC) [13]. This algorithm has been shown to be very efficient for a wide variety of CSP settings. In addition, it is very simple to implement and, most importantly, the state that needs to be kept during evaluation does not consume substantial amount of space. The basic idea behind FC is that every time a variable is instantiated, the new

value is checked for consistency with all available values of the domains of outstanding variables and inconsistent values are removed from them. The characteristic data structure used by FC is one array per variable domain, with length equal to the size of the domain. Each element of the array is the id of the variable that made the corresponding domain value inconsistent. For few variables and small variable domains this collection of arrays will be very small in size (each element can be one byte or less). Since we make sure that before the CSP Solver is called the variable domains have been restricted as much as possible, FC is expected to be very robust for spatial and spatiotemporal CSPs, as demonstrated in [23]. An alternative way is to evaluate the CSP as a multiway spatiotemporal join of the variable domains using their binary constraints as join predicates. Nevertheless, secondary memory techniques for multiway joins [20] are not expected to perform better than CSP algorithms for main-memory problems of small domains.

4 Alternative Query Evaluation Techniques

In this Section, we propose some variants to our basic algorithm which trade memory requirements for computational performance.

4.1 Computing and Updating Tight Expiration Times

Algorithm 1 may compute very loose expiration times for newly arriving events, which affect negatively the memory requirements of the system. Consider again the example query of Figure 1b and the stream of Figure 2. When a_1 arrives, Algorithm 1 sets its expiration time to 11, i.e., $a_1.t = 1$ plus the maximum t_{ub} of an outgoing edge ($C_{1.3}.t$ in this example). Nevertheless observe that unless an event with property B arrives before time 6, a_1 should be deleted, because $C_{1,2}.t$ may never be satisfied. Thus a *tight* expiration time for a_1 is 6. When an event of type B arrives at or before time-instant 6, which satisfies $C_{1,2}$ with a_1, then the expiration time of a_1 is *renewed*. Indeed, event b_2 satisfies $C_{1,2}$ with a_1, thus a_1 remains in the system until time-instant 8 (the expiration time of b_2). This simple example shows that we could minimize the memory requirements of CCQ evaluation at the expense of computing and maintaining the expiration times of active events.

Let e be a *new-coming* event and V_i a variable in a query $\mathcal{G}$, such that $e \propto C_i$. Assume that e is useful with respect to V_i, i.e., the graph $\mathcal{G}'$ containing $\mathcal{V}_i^-$ is soluble. For setting a tight expiration time for e, with respect to V_i, we separate the following two cases:

1. $\mathcal{V}_i^- \cup V_i = \mathcal{G}.\mathcal{V}$. In this case, e triggers an alert, however, newly arriving events for variables $V_j \in \mathcal{V}_i^+$ may keep triggering alerts for as long as the temporal constraint $C_{i,j}.t$ is active (given that the events satisfy the spatial constraints as well). This is true, since for all other j, the constraints are already satisfied. Intuitively, this can keep happening for as long as the longest lived temporal constraint $C_{i,j}.t$. Thus $e.X$ should be updated to $\max\{e.X, e.t + \max_{\forall V_j \in \mathcal{V}_i^+}(C_{i,j}.t_{ub})\}$, exactly like in the original algorithm.

2. $\mathcal{V}_i^- \cup V_i \neq \mathcal{G}.\mathcal{V}$. In this case, there are future variables to V_i not in $\mathcal{V}_i^-$. If there are many such variables, we should set the expiration time for e as the *minimum* $C_{i,j}.t_{ub}$ of all future variables V_j.

Algorithm 2 HandleEventTight(e: event, $\mathcal{Q}$: queries)

```
..... lines 1–11 of Algorithm 1 .....
if G is satisfiable then
    alert solution;
    e.X := max{e.X, e.t + max_{∀V_j ∈ V_i^+} (C_{i,j}.t_ub)};
else e.X := max{e.X, e.t + min_{∀V_j ∈ V_i^+} (C_{i,j}.t_ub)};        ▷ case 2
..... the rest of Algorithm 1 .....
```

Since e is independently important for all $(V_i, \mathcal{G})$ pairs, its expiration time is eventually set as the maximum of all expiration times due to each V_i. Algorithm 2 summarizes the changes to Algorithm 1 for computing tight expiration times.

Defining tight expiration times requires their correct maintenance as new events arrive. In our example, recall that the expiration time 6 for a_1 was updated to 8, after the arrival of b_2. Let e be a new event, handled by Algorithm 2. When solving graph $\mathcal{G}'$, for all variables $V_j \in \mathcal{V}_i^-$ and for each consistent assignment $V_j \leftarrow e'$, the expiration time of e' is updated to $\max\{e'.X, e.X\}$. In other words, e' should remain in the database at least as long as e is useful, if there are events future to e that may generate alerts with e and e'. Embedding expiration time updates in Algorithm 2 is straightforward.

4.2 Explicit Maintenance of Variable Domains

Algorithm 1 and its extension (Algorithm 2) employs the spatiotemporal index for each incoming event to define variable domains on the fly and then solve the CSP on the domains restricted by binary and unary constraints. Observe that, in this way for a variable V_j in a CG, the constraint C_j may be validated on the same event e multiple times (i.e., the first time e arrives and every time it satisfies the binary constraints in a past variable). An alternative method is to apply C_j only once per event e and then store e explicitly in the domain of V_j, until e expires. In other words, the domains of past variables V_j are not computed by probing the spatiotemporal index and then filtering by C_j, but by filtering the existing domains of V_j (due to C_j) using the spatiotemporal constraints (and not the index). This method may result in more expensive CSP evaluation, since the spatiotemporal constraints are validated without the use of an index for each CSP, however, we do not need to bother about a construction and maintenance of a spatiotemporal index. In addition, the required space increases, since one event may be stored in multiple domains (or a pointer to an event, for decreasing the cost of duplication). In Section 5 we evaluate the performance of this approach.

5 System Prototype Evaluation

This section presents a system prototype evaluation that will illustrate the applicability of the proposed techniques using real datasets. The performance of a CCQ evaluation

engine depends mainly on two major operations, populating the variable domains and solving the CSPs. Therefore, we compare multiple variants of the CCQ prototype to quantify the effects of different evaluation strategies.

5.1 Testbed and Methodology

We use off-the-shelf tools to implement our system. More specifically, an R-tree index from [11] and a CSP solver based on the FC algorithm [13]. The system can be downloaded from [1]. All experiments are run on an Intel(R) Xeon(TM) CPU 3.2GHz.

We use real datasets from the Tropical Atmosphere Ocean Project [25], where a large number of buoys have been deployed around the pacific ocean to collect oceanic and atmospheric data several times a day. An archive of the measurements of the past 25 years is available from [25]. For our purposes we used a total of 900,000 measurements, interpreted as a stream of data arriving at a central server for processing. These measurements include the location of the buoy at the time of the measurement, sea surface temperature, pressure, dynamic height, salinity, relative humidity, wind speed, and wind direction.

We generate synthetic queries by variying the number and type of variables, the temporal constraints, and the spatial predicates. We generate a large number of query workloads based on query verbosity. Verbosity is defined as the total number of query results (alerts) produced by the query over the total number of streaming events, and can be adjusted by appropriately tuning unary and binary constraints, making event selections tighter or looser. The dataset and the queries can be downloaded from [1].

To test various aspects of the system prototype we used two performance measures: (1) the maximum sustainable processing rate that can be achieved; and (2) the maximum memory utilization. The first measure is computed as the number of streaming events that can be processed per second, and the second as the peak number of events that need to be stored in main memory to achieve the corresponding processing rate. We measure the system's performance according to the following measures: (1) query verbocity; (2) scalability; (3) temporal constraint length, number of variables and number of constraints per query. We compare three CCQ evaluation methods: (1) Algorithm 1 (Loose); (2) Algorithm 2 (Tight); (3) The alternative proposed in Section 4.2 (NoIndex).

Query Verbosity. We measured the performance of our system as a function of query verbosity (i.e., number of alerts produced over the total number of events processed). We run all variants using five registered queries of known verbosity.

Results are shown in Figure 5. The trend of the graph shows that for all variants, performance deteriorates as verbosity increases since a growing number of events qualify for the variable domains of the query as more alerts are being produced, meaning that CSP evaluation becomes more expensive. Notice that the Loose variant (the only one that does not utilize tight expiration times) has somewhat worse performance than the rest of the techniques, attributed to the larger variable domain sizes. The NoIndex approach offers the highest processing rates since it does not have to maintain and query an index.

Figure 5b plots the memory utilization for each variant. The numbers on the graph correspond to the peak number of events that need to be stored as a percentage of the

total number of events. This measure illustrates the pruning ability of our system with respect to a brute force approach that would retain all measurements. Clearly, all techniques save substantial amount of main memory, especially for higher query verbosity. The NoIndex approach has larger memory requirements due to event replication (see Section 4.2).

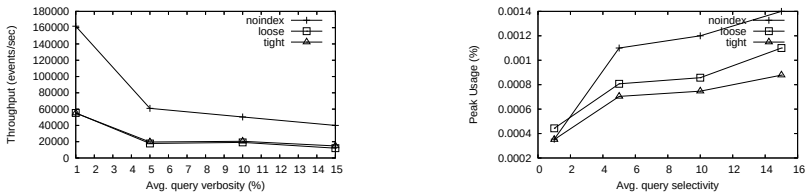

Fig. 5. Query Verbosity

Scalability. Next, we evaluate system performance as a function of the number of registered queries. We keep the verbosity fixed to 1% and vary from 5 up to 100 registered queries. The NoIndex approach can sustain up to 15,000 events per second even for 100 registered queries. The other two approaches suffer as the number of queries increases, but have viable processing rates for most application scenarios even for up to 25 queries (we should stress the fact here that the algorithms produce *exact* query results). In terms of memory utilization, the NoIndex approach introduces substantial event duplication, making it a less favorable approach for tight memory constraints. Nevertheless, the memory requirements of all algorithms are still extremely small.

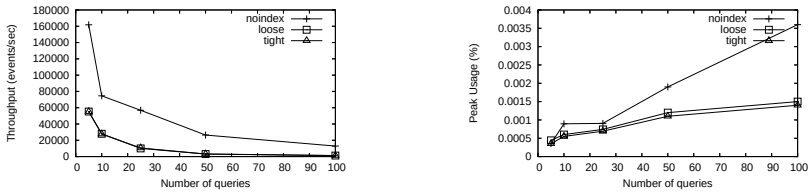

Fig. 6. Number of Queries

Temporal Constraint Length. Finally, we tested the system using queries with varying temporal constraint lengths. We use five registered queries and fix the verbosity to 1%. The larger the temporal extents the bigger the event expiration times, so it is expected that the memory requirements will increase as the temporal extents increase. On the other hand, throughput should remain unaffected since the verbosity is fixed. The results in Figure 7 attest to that observation. We conducted similar experiments for varying number of variables per query and number of constraints. The results were the same. Throughput is only affected by query verbosity and the total number of queries registered in the system.

To conclude, the NoIndex approach exhibits very good scalability in terms of registered queries and query verbosity, but higher memory requirements then the other approaches. The Loose and Tight algorithms can sustain a smaller number of queries at streaming rates due to the index lookups. All approaches prune a very large percentage of events, compared to the naive alternative.

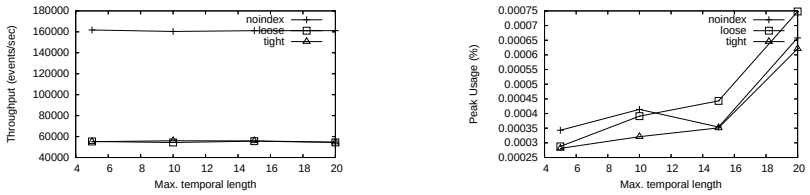

Fig. 7. Temporal Constraint Length

6 Related Work

There is a lot of research on data streams in general but little work in the realm of spatiotemporal streams. Moreover, most work has concentrated on the special case of moving object applications. Finally, current work addresses only traditional spatial queries like range searches and nearest-neighbors. In contrast, here we deal with any type of spatiotemporal stream as well as complex queries with numerous problem variables and general spatiotemporal predicates between them (i.e., not only selections but also joins between events).

SINA [22] is a recently proposed framework for incremental evaluation of continuous range queries on data streams. It uses the shared execution paradigm to incrementally evaluate a large number of concurrent queries. SINA indexes the queries along with the data in order to be able to compute answers incrementally. Previous work with the same characteristics include [26, 5, 9]. In [26] the authors use incremental query evaluation, reversing the role of queries and data, and exploiting the relative locations of objects and queries to provide answers efficiently. They also propose an index method that exploits the maximum permissible velocity of the objects to delay expensive updating operations of the index. Finally, the authors of [5] and [9] assume that clients can process and store information, so that they can share query processing with the server in a distributed fashion. All these works are suited only for moving object applications and continuous range queries with absolute spatial coordinates that do not involve constraints in-between the objects. Similar work, concerning nearest neighbor queries, includes [16, 28, 30].

Related to our work is research concerning pattern mining with constraints in streaming databases. In [31] the authors address the issue of extracting frequent temporal patterns from the stream. They use a regression based algorithm to scan online transaction flows and generate candidate frequent patterns in real time. Similarly, a mining perspective is also adopted in [24], where the authors introduce techniques for answering queries with a wide range of constraints related to the length of the patterns, the items

they contain, their duration, etc. However, these works do not consider transactions with spatial characteristics and concentrate on mining patterns that exceed a user specified threshold instead of identifying tuples that satisfy spatiotemporal or other constraints in real-time.

In [6] a system is proposed that can answer continuous queries over streaming data. The system registers a number of queries and a number of streams and applies new queries to old data, and old queries to new data on the streams. Nevertheless, this system does not consider spatial or temporal constraints between streaming data; it only considers predicates that resemble what we term unary variable constraints. Finally, it does not introduce the concept of expiration times to evict older data from main memory, but rather indexes all incoming data according to user specified window sizes.

7 Conclusions

We have presented a system prototype for evaluating *Continuous Constraint Queries* on spatiotemporal streams. The proposed system represents queries as Constraint Graphs that are incrementally evaluated as Constraint Satisfaction Problems every time a new event arrives on the stream. We introduce special algorithms for computing event expiration times in order to limit the number of events that need be maintained for providing exact answers. Finally, we present a concise experimental evaluation of a system prototype implementation. As future work we plan to extend the system for dynamic event properties (that change over time) and also investigate robust approximation policies for limiting main memory consumption even further.

References

1. CCQ system prototype. http://www.cs.ucr.edu/ ∼marioh/ccq.
2. AOML. Global Drifter Center. http://www.aoml.noaa.gov/ phod/dac/gdc.html.
3. C. Bessière and J.C. Régin. Refining the basic constraint propagation algorithm. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 309–315, 2001.
4. J. R. Bitner and E. Reingold. Backtracking programming techniques. *Communications of the ACM (CACM)*, 18(11):651–656, 1975.
5. Y. Cai, K. A. Hua, and G. Cao. Processing range-monitoring queries on heterogeneous mobile objects. In *Proc. of the International Conference on Mobile Data Management (MDM)*, pages 27–38, 2004.
6. S. Chandrasekaran and M. J. Franklin. Streaming queries over streaming data. In *Proc. of Very Large Data Bases (VLDB)*, 2002.
7. R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Journal of Artificial Intelligence*, 49(1-3):61–95, 1991.
8. J. Gaschnig. Experimental case studies of backtrack vs. waltz-type vs. new algorithms for satisficing assignment problems. In *Proc. of the Canadian Artificial Intelligence Conference*, pages 268–277, 1978.
9. B. Gedik and L. Liu. MobiEyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *Proc. of Extending Database Technology (EDBT)*, pages 67–87, 2004.
10. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of ACM Management of Data (SIGMOD)*, pages 47–57, 1984.

11. M. Hadjieleftheriou, E. Hoel, and V. J. Tsotras. Sail: A library for efficient application integration of spatial indices. In *Proc. of Scientific and Statistical Database Management (SSDBM)*, 2004.
12. E. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. Park, and A. Vernon. Scalable trigger processing. In *Proc. of International Conference on Data Engineering (ICDE)*, pages 266–275, 1999.
13. M. Haralick and J. Elliot. Increasing tree-search efficiency for constraint satisfaction problems. *Journal of Artificial Intelligence*, 14(3):263–313, 1980.
14. M. Keidl, A. Kreutz, A. Kemper, and D. Kossmann. A publish & subscribe architecture for distributed metadata management. In *Proc. of International Conference on Data Engineering (ICDE)*, pages 309–320, 2002.
15. V. Kumar. Algorithms for constraints satisfaction problems: A survey. *The AI Magazine*, 13(1):32–44, 1992.
16. I. Lazaridis, K. Porkaew, and S. Mehrotra. Dynamic queries over mobile objects. In *Proc. of Extending Database Technology (EDBT)*, 2002.
17. M.-L. Lee, W. Hsu, C. S. Jensen, and K. L. Teo. Supporting frequent updates in R-Trees: A bottom-up approach. In *Proc. of Very Large Data Bases (VLDB)*, 2003.
18. C. Papadimitriou M. Grigni, D. Papadias. Topological inference. In *Proc. of the International Joint Conference of Artificial Intelligence (IJCAI)*, 1995.
19. S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. of ACM Management of Data (SIGMOD)*, 2002.
20. N. Mamoulis and D. Papadias. Multiway spatial joins. *ACM Transactions on Database Systems (TODS)*, 26(4):424–475, 2001.
21. N. Mamoulis and M.L. Yiu. Non-contiguous sequence pattern queries. In *Proc. of Extending Database Technology (EDBT)*, pages 783–800, 2004.
22. M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable incremental processing of continuous queries in spatiotemporal databases. In *Proc. of ACM Management of Data (SIGMOD)*, 2004.
23. D. Papadias, N. Mamoulis, and V. Delis. Algorithms for querying by spatial structure. In *Proc. of Very Large Data Bases (VLDB)*, pages 546–557, 1998.
24. J. Pei, J. Han, and W. Wang. Mining sequential patterns with constraints in large databases. In *Proc. of Conference on Information and Knowledge Management (CIKM)*, 2002.
25. PMEL. Tropical Atmosphere Ocean Project. http://www.pmel.noaa.gov/tao.
26. S. Prabhakar, Y. Xia, D. Kalashnikov, W. G. Aref, and S. E. Hambrusch. Query indexing and velocity constraint indexing: Scalable techniques for continuous queries on moving objects. *IEEE Transactions on Computers*, 51(10):1–17, 2002.
27. H. Six and P. Widmayer. Spatial searching in geometric databases. In *Proc. of International Conference on Data Engineering (ICDE)*, pages 496–503, 1988.
28. Z. Song and N. Roussopoulos. K-nearest neighbor search for moving query point. In *Proc. of Symposium on Advances in Spatial and Temporal Databases (SSTD)*, pages 79–96, 2001.
29. M. Stonebraker, T. K. Sellis, and E. N. Hanson. An analysis of rule indexing implementations in data base systems. In *Expert Database Conference*, pages 465–476, 1986.
30. Y. Tao, D. Papadias, and Q. Shen. Continuous nearest neighbor search. In *Proc. of Very Large Data Bases (VLDB)*, pages 287–298, 2002.
31. W.-G. Teng, M.-S. Chen, and P. S. Yu. A regression-based temporal pattern mining scheme for data streams. In *Proc. of Very Large Data Bases (VLDB)*, 2003.
32. E.P.K. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London and San Diego, 1993.
33. T. W. Yan and H. Garcia-Molina. The sift information dissemination system. In *ACM Transactions on Database Systems (TODS)*, pages 529–565, 1999.