Complex Patten Processing in Spatio-temporal Databases*

Yang Zheng, Annies Ductan, Devin Thomas and Mohamed Y. Eltabakh

Computer Science Department, Worcester Polytechnic Institute (WPI), Worcester, MA, U.S.A.

Keywords: Spatio-temporal Query Processing, Pattern-matching Queries, GIS, Query Optimization.

Abstract: The increasing complexity of spatio-temporal applications has caused the underlying queries to be more sophisticated and usually carry complex semantics. As a result, the traditional spatio-temporal query types, e.g., range, kNN, and aggregation queries, have become just building blocks in more complex query plans. In this paper, we present the STEPQ system, which is an extensible spatio-temporal query engine for complex pattern processing over spatio-temporal data. STEPQ enables full-fledged and optimized integration between spatio-temporal queries and complex event processing (CEP). This integration enables expressing complex queries that execute the desired application semantics without the need for indifferent middle-aware or application-level support. The system is implemented using TerraLib module on top of PostgreSQL DBMSs. The experimental evaluation demonstrates the feasibility and practicality of the STEPQ system, and the efficiency of the proposed optimizations.

SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

Spatio-temporal data processing provides eminent support for many application domains such as traffic monitoring and transportation systems, geographic information systems (GIS), location-based services (LBS), and environmental science. As these applications get more complex, the underlying queries typically go beyond the traditional spatio-temporal query types, e.g., *range*, *k nearest-neighbor (kNN)*, and *aggregation* queries, e.g., (Benetis et al., 2002; Cai et al., 2004; Shahabi et al., 2003), to more complex and semantics-rich pattern queries, i.e., queries looking for complex patterns in the spatio-temporal data. The following examples demonstrate such complex queries:

Example 1− Capturing Suspicious Activities: *Assume that we want to monitor and report suspicious activities from child-abuse criminals. A suspicious activity can be defined as a child-abuse criminal who appears in a school area, say A1, moves to a suspicious area, say A2, within one hour, and then appears again in area A1 in the say day. And his/her presence in each area should be at least x time interval (to avoid noise events).*

In Example 1, the query involves several spatio-temporal range queries, time-based correlation among the queries' results, and persistency semantics, e.g., the criminal should stay in the school area for more than x continuous minutes. Such complex query, which captures the application semantics, cannot be supported only by the traditional spatio-temporal database systems.

Example 2− Generating Healthcare Alerts: *In healthcare systems, detecting potential threats for patients as early as possible can be used for preventive medicine. For example, when monitoring patients with a weak immune system, if a patient P stays in contact with another patient having a transferable disease, then an alert should be sent to P to get away from the current location. The definition of "being in contact" can be defined as "being within a distance d for a time interval more than m minutes from the other person".*

In this example, the query involves multiple ranges queries, synchronized execution between them, and also persistency semantics, e.g., it is not practical to assume a health threat and send an alert to a patient depending solely on a single instance or snapshot of the data.

Spatio-temporal pattern queries involve several challenges including: **(1)** They capture and impose diverse semantics that cannot be captured by current spatio-temporal queries, **(2)** Unlike the tradi-

*This project is partially supported by the NSF-CRI 1305258 grant.

Zheng Y., Ductan A., Thomas D. and Y. Eltabakh M..
Complex Patten Processing in Spatio-temporal Databases.
DOI: 10.5220/0004992401570169
In *Proceedings of 3rd International Conference on Data Management Technologies and Applications* (DATA-2014), pages 157-169
ISBN: 978-989-758-035-2
Copyright © 2014 SCITEPRESS (Science and Technology Publications, Lda.)

tional queries, e.g., *range* or *kNN* queries, that can be evaluated on each instance of the database in isolation, pattern queries require correlation among spatio-temporal events (both in time and space) over multiple instances of the database, **(3)** They require a full-fledged system with not only efficient event-processing techniques but also efficient spatio-temporal processing, and **(4)** Unlike current event-processing techniques that assume no control over the input stream of events, spatio-temporal pattern queries generate these streams of events using underlying traditional queries, e.g., *range*, *kNN*, or *aggregation*, and hence crucial optimization opportunities emerge to control which events to generate.

These challenges combined make the state-of-art in complex event processing (CEP), e.g., (Wu et al., 2006; Aguilera et al., 1999; Gehani et al., 1992), not applicable since CEP techniques cannot process traditional *range* or *kNN* queries efficiently, let alone the more complex spatio-temporal pattern queries. They also make the state-of-art in spatio-temporal databases (STDBs), e.g., (Behr and Guting, 2005; Gedik and Liu., 2004; Wolfson et al., 1999), fall short since they lack the expressiveness power and processing capabilities of handling complex pattern queries. Moreover, as we will discuss in detail in Section 2, loosely stacking the CEP systems on top of STDBs has serious limitations w.r.t communication, integration, and optimization.

In this paper, we propose the STEPQ system, an *extensible spatio-temporal engine for complex pattern queries*. STEPQ enables *efficient* and *scalable* evaluation of complex spatio-temporal pattern queries. The key unique and novel cornerstones of this project are: **(1)** Coherent integrated system, where spatio-temporal and pattern-matching processing are combined and fully integrated in one consistent system, **(2)** Cross-cutting optimizations, where the pattern-matching queries can optimize the execution of the underlying spatio-temporal queries by controlling when to run/suspend a query and what events to generate, and **(3)** Extensible architecture, where we provide not only concrete and well-defined query operators, but also abstract interfaces that can accommodate for a wide range of applications and semantics through pluggable modules.

2 CHALLENGES & SOLUTIONS

2.1 Challenges in Spatio-temporal Pattern Queries

A spatio-temporal pattern query can be viewed as a two-layered query. The first layer executes one or more of traditional spatio-temporal queries, e.g., *range* and *kNN*, on top of the raw input stream coming from moving objects (we refer to these queries as *base* queries). The second layer executes one or more complex pattern-matching queries on top of the results generated from the base queries.

One possible solution to support spatio-temporal pattern queries is the *application-level* solution depicted in Figure 1(a). As highlighted in (Xiao and Eltabakh, 2013), this solution has several drawbacks including: **(1)** mobile devices usually have limited power and processing capabilities, and hence the required processing may not be even feasible on these small devices, and and **(2)** lack of many possible optimizations that could have been performed by the execution engine.

Another, more feasible, solution is the *middleware-level* solution depicted in Figure 1(b). In this solution, the existing CEP systems act as a middleware layer deployed on top of the STDBs. In this case, applications need to decompose a given pattern query into one or more spatio-temporal base queries that can be executed by the STDBs and separate pattern queries that can be executed by the CEP system. The results from the base queries should act as input streams to the CEP system. Although this approach is more feasible, it does not address the following challenges in spatio-temporal pattern queries:

(1) Coupling Challenges: There are several linking problems that emerge between the STDB and CEP layers. First, STDBs deploy incremental evaluation techniques for purposes of efficiency and scalability, whereas CEP systems do not handle incremental updates of the input events. Second, base queries may produce empty results, e.g., empty range query, which will be mistakenly interpreted by CEP systems as no input events. Interestingly, empty results still need to be processed as *special* events since they may invalidate or reset patterns looking for continuity, e.g., the query in Example 1 requires continuous range-existence for at least x minutes in area $A1$. Third, base queries can themselves be moving objects, and hence CEP systems need to get as input not only the query answer, but also the query points.

(2) Optimization Challenges: Since spatio-temporal pattern queries generate both base queries

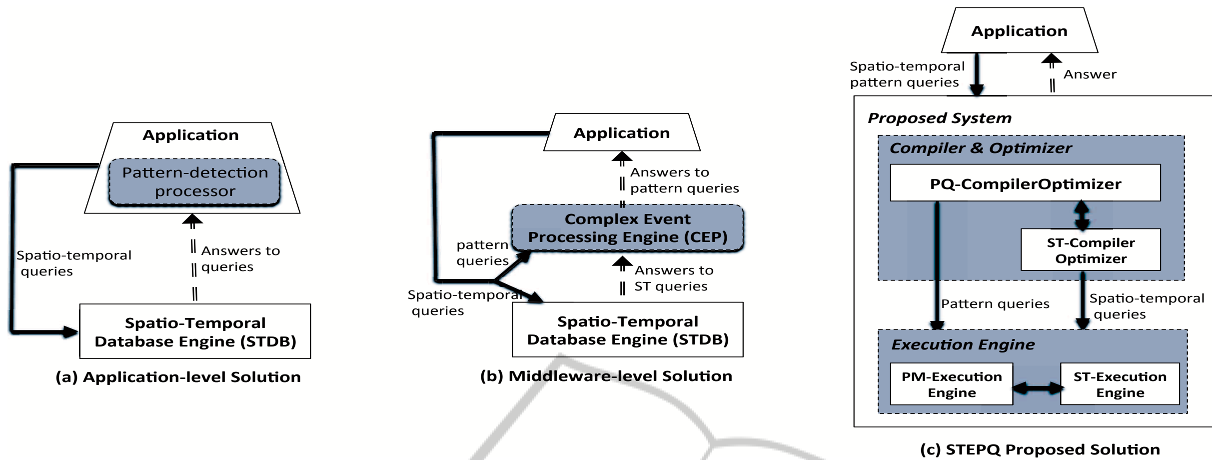


Figure 1: Possible Architectures for Spatio-Temporal Pattern Queries (Xiao and Eltabakh, 2013).

and pattern-matching queries, then several optimization opportunities arise. However, since the STDB and CEP layers are loosely coupled and queries are isolated from each other, then these cross-cutting optimizations cannot be performed. The following example illustrates possible optimizations that cannot be performed within the *application-* or *middleware-level* solutions.

Example 3– Testing Effect of Radar Signs: *Assume a radar sign is added in a certain area, A2, and we want to test the affect of the sign on the speeding cars. A query can be: For consecutive areas A1, A2, and A3, report speeding cars (over the speed limit for at least x mins) in A1 and A3 but not in A2.*

The query in Example 3 involves three *range* queries over areas *A1*, *A2*, and *A3*. A naive plan is to run the queries in isolation from each other. However, in a more optimized plan, the queries over *A2* and *A3* should run only if there is a match in the previous areas. The two latter queries can be further optimized by considering only the cars that matched the pattern in the previous areas. These types of optimizations are not feasible in the *middleware-level* approach.

(3) Synchronization and Transformation Challenges: A spatio-temporal pattern query may require not only executing multiple base queries to generate events, but also synchronizing their execution. The query in Example 4, demonstrates the need for such synchronization.

Example 4– Common kNN Query: *Assume two moving cars and they want to find restaurants that are in the kNN of them and getting closer to both cars over and interval T. That is, finding common nearby restaurants in the direction of the moving cars.*

In this example, the query requires synchronizing the execution of two moving *kNN* queries, i.e., they should produce results almost simultaneously, and then jointly processing their results, e.g., intersecting the output from both queries. Such synchronization and transformation over the event streams are not feasible in the middleware-level approach and not even supported by current STDBs.

2.2 Overview on The STEPQ System

To address the above challenges, we proposed the STEPQ system depicted in Figure 1(c). In (Xiao and Eltabakh, 2013), we highlighted the architecture of the system. In this paper, we present the details of the system along with its experimental evaluation.

STEPQ consists of two standard layers; compilation/optimization and execution layers. In the compilation/optimization layer, we introduce the *pattern-query compiler & optimizer* (*PQ-CompilerOptimizer*) component which is the central component of the system responsible for compiling and optimizing the entire query. Given a spatio-temporal pattern query, *PQ-CompilerOptimizer* decomposes it into one or more traditional queries (the *base* queries) and pattern-matching queries. The individual base queries, e.g., *range* or *kNN* queries, are compiled and optimized using an extended *spatio-temporal compiler & optimizer* (*ST-CompilerOptimizer*) that works under the control of the *PQ-CompilerOptimizer*. In contrast, pattern-matching queries are fully compiled by *PQ-CompilerOptimizer*. The base queries will be executed by the extended spatio-temporal execution engine (*ST-ExecutionEngine*), while the pattern-matching queries will be executed by the *pattern-matching execution engine* (*PM-ExecutionEngine*). The continuously generated results from the base

queries will drive the progress of the pattern-matching queries.

The *ST-CompilerOptimizer* and *ST-ExecutionEngine* components will inherit and leverage the state-of-art technologies from spatio-temporal databases and will be extended with new features as needed. For example, new operators will be introduced such as: **(1)** Materialization operators that materialize the incremental updates produced from the base queries to complete answer sets before feeding them to the *PM-ExecutionEngine*, **(2)** Synchronization operators that ensure synchronized execution over multiple spatio-temporal queries, and **(3)**Transformation operators that apply any required transformation over the answer sets produced by the base queries.

To evaluate the pattern-matching queries, the *PM-ExecutionEngine* uses a variant of *Non-deterministic Finite Automata (NFA)* as well as a set of operators that manipulate the automata results. NFAs have been adopted by several systems that represent a pattern query as a sequence of automata states (Wu et al., 2006; Adaikkalavan and Chakravarthy, 2003; Demers et al., 2006). While leveraging this work, we provide substantial and core extensions crucial to spatio-temporal pattern queries such as extended event model, abstract automata interfaces for extensibility, and new event operators.

The proposed STEPQ system addresses the three key challenges highlighted in Section 2.1 as follows. The *Coupling* challenges are addressed by the *PQ-CompilerOptimizer* component, where the spatio-temporal and patten queries are linked and aligned together to achieve correct execution. The *Optimization* challenges are addressed through a feedback mechanism built between the pattern queries and the the corresponding spatio-temporal queries. This feedback mechanism enables the pattern queries to send hints and control signals to the spatio-temporal queries to optimize their execution. The *Synchronization* challenges are addressed through the newly introduced operators, i.e., the Materialization, Synchronization, and Transformation operators, that operate on the results of the spatio-temporal queries before passing them to the *PM-ExecutionEngine*.

3 STEPQ MODEL & LANGUAGE

3.1 STEPQ Event Model

STEPQ manages several events at different granularities, e.g., raw events from moving objects, and spatio-temporal events generated from the running queries. Therefore, the event model in STEPQ consists of three event types, which are *raw*, *event set*, and *event list*, defined as follows:

Raw Event: *A raw event R_S is a primitive instantaneous event that has no time duration and follows the schema S that defines the attributes of R.*

An example of raw events is a stream identifying an object's movement, e.g., (ObjectId, Location, Speed, and Direction). These events are the input to the *ST-ExecutionEngine* to execute the spatio-temporal queries.

Event Set: *An event set $E_S = \{R_S^1, R_S^2, ..., R_S^k\}$ consists of a set of primitive events following the schema S and with no order imposed between them. E_S is an instantaneous event that has no time duration.*

An example of an event set is the answer produced from one of the base queries, e.g., the query monitoring the speeding cars in region $A1$ (Example 3), or the query reporting the k nearest neighbor restaurants to a moving car (Example 4). These events are the input to *PM-ExecutionEngine* to execute the pattern queries.

Event List: *An event List $L_S = [R_S^1, R_S^2, ..., R_S^k]$ consists of a list of ordered primitive events following the schema S. L_S spans the time duration between the first and last primitive events R_S^1 and R_S^k.*

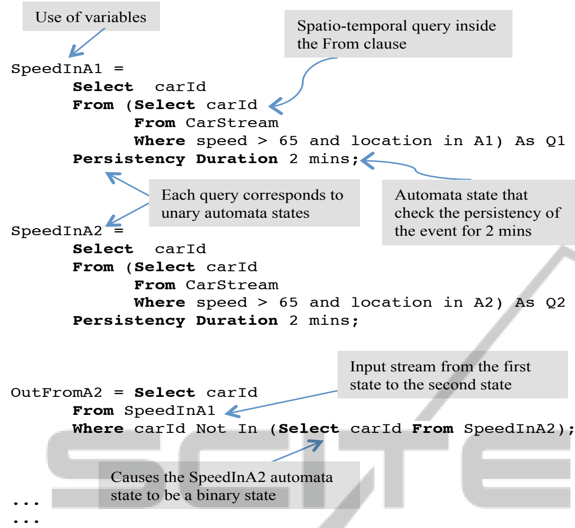
An example of an event list is the speeding event of a car $L_S = [R_S^1, R_S^2]$, where R_S^1 is the primitive event corresponding to the first detection that the car is over the speed limit, and R_S^2 is the primitive event corresponding to reporting the car as speeding.

3.2 Language Building Blocks

In this section, we present several features of the STEPQ query language through examples. Since the event model in the system is based on the relational model, i.e., each event or event set has a well-defined schema, the query language is designed to be very similar to SQL with several extensions, which include: **(1)** The language allows the computations to be performed as a series of steps, where the output from each step can be assigned to a named variable, and then used later by other steps, **(2)** New commands and clauses that enable synchronized query processing, and persistency predicates, And **(3)** Handling streams of events instead of static relational tables. The following examples demonstrate the main building blocks of the language.

The Query in Example 3: Assume the query is processes an input spatio-temporal stream called *CarStream*. The stream has the following schema

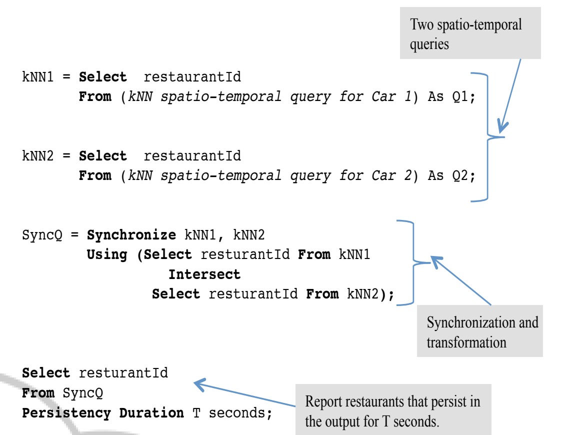
(carId, location, speed). Then, The query can be expressed as:



In the language, the From clause takes either a spatio-temporal query or an output from a previous automata state. The language will not allow multiple streams in the From clause. For example, in the 1st and 2nd statements, the From clause takes range queries over areas A1, and A2, respectively. The *Persistency* clause indicates that a successful event should be persistent for 2 mins. Given the first two statements, their compilation output will be a spatio-temporal range query for each statement as well as a unary automata state accepting the output from each query and checking for the persistency (See Figure 2).

The 3rd statement in the script will then link the two automata states together. The statement indicate that each event in `SpeedInA1` should be reported if not in `SpeedInA2`. Thus, the second automata state will bind to the first state as its left input as indicated in Figure 2. Notice that the first two statements in the script, referred to as *"State Statements"*, do not impose any sequential order between the two queries, but the third statement, referred to as *"Binding Statement"*, creates this order. To complete the query in Example 3, the script will include two more statements, one statement similar to the first two statements to create `SpeedInA3` stream, and the final one to produce the output where the `OutFromA2` stream should exist in the *SpeedInA3*.

The Query in Example 4: In this query, we need to synchronize the execution between two kNN queries. The language enables this synchronizations using new commands as follows:



Notice that Step 1 and 2 create two kNN spatio-temporal queries for the two cars involved in the query. Then, in Step 3, the `Synchronize` statement takes two or more base queries and synchorinze their execution, i.e., they produce results almost simultaneously (More details are presented in Section 6). The Synchronize command treats the output from each query as a relational table, and hence on top of the synchronization, any transformations on these relations is possible (The transformation phase is also presented in Section 6). In our example, the transformation involves intersecting the two relations to find the common restaurants to the two cars. The 4th statement in the script is to ensure the persistency of any common restaurant for a duration T seconds before reporting it.

4 PLAN GENERATION

The PQ-CompilerOptimizer component is the heart of the STEPQ system. It is responsible for compiling—and ultimately optimizing—the entire spatio-temporal pattern query. The input to PQ-CompilerOptimizer is a query expressed in STEPQ's high-level language (Section 3.2). The output is a complete query plan divided into two layers; the lower layer is one or more spatio-temporal queries (executed by the ST-Execution Engine), and the upper layer is the pattern matching query (executed by the PM-Execution Engine).

For example, given the query in Example 3, the system will generate the query plan illustrated in Figure 2. The raw events will feed three different range queries monitoring areas $A1$, $A2$, or $A3$, respectively, and reporting any car with speed higher than the speed limit. Notice that these queries check the speed on each individual instance (without memorization or pattern construction). The patterns are then formed

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

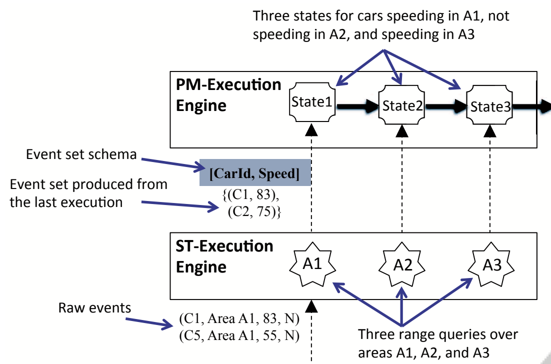


Figure 2: Execution Plan for Example 3 Query.

and checked in the upper layer of the query. For example, State 1 in the automata will check that a car stays above the speed limit for all event sets within an interval T. If that is the case, then the car will be considered as speeding in area $A1$ and the automata moves to the next state. The basic functionalities of the PQ-CompilerOptimizer include:

(1) Query Decomposition: Given a spatio-temporal pattern query, *PQ-CompilerOptimizer* decomposes the query into one or more base queries and a pattern-matching query. For example, given the code for Example 3 query highlighted in Section 3.2, the system identifies three range queries over regions A1, A2, and A3. These individual base queries are passed to *ST-CompilerOptimizer* for compilation and optimization according to the state-of-art in STDBs.

The next step is to identify the automata states in the query. This is defined through some clauses in the query such as the *Persistency* clause. For example, referring to the query's code, each of the three range queries mandate a persistency of 2 mins. Thus, the system creates an automata state corresponding to each base query. Each state has several properties and interface functions that define the behavior of the state. For the *Persistency* clause, the system can automatically set these properties and interface functions—More details are presented in Section 5 on these interface functions.

Each created automata state is either a unary or binary state. A unary state has a single input stream coming from a base query, e.g., State 1 in our example. In contrast, a binary state takes an additional input from another automata state. Referring to the query's code, the 3rd statement specifies that the output from State 1 is filtered by by the output from State 2. Thus, the system will make State 2 as a binary state and feeds State 1's output to it as depicted in Figure 2. The same generation process applies to State 3. At this stage, the base and automata states are generated but without a connection between between them. In the current version of STEPQ, the creation of the automata states is not based on a cost model. Instead, the system uses a set or rules and templates to map from the high-level language to the exception plan.

(2) Coupling Base and Pattern Queries: The two layers of the query have to communicate with each other by passing events and hints. We developed a new mechanism through a *tagging scheme* controlled by the *PQ-CompilerOptimizer*, in which a tag will be assigned to each base query. A query's output will be labeled with that tag. The automata states generated by *PQ-CompilerOptimizer* will be also tagged such that each stream of input events is coupled with its corresponding automata state(s), i.e., each automata state will accept only events corresponding to tag. It is possible that a single base query may feed multiple states (in the same automaton or different automata), hence the same tag can be assigned to multiple states. Referring to the example query in Figure 2, each of the three base queries will be assigned tags, *t1*, t2, and t3, respectively. The three automata States will be also assigned the same tags.

(3) Adding Materialization Operators: The output produced from the *ST-ExecutionEngine* can incremental, i.e., producing only the delta from the previous output. Most spatio-temporal database systems support the incremental update of results for efficiency purposes (Gedik and Liu., 2004; Hu et al., 2005). There are two options for handling incremental updates; either the *PM-ExecutionEngine* becomes aware of the incremental updates and capable of processing the positive- and negative-tuples generated from the base queries, or the answer set becomes fully materialized before it is sent out from the *ST-ExecutionEngine*. We favored the latter approach because this issue is more tight to the base queries than the pattern-matching queries. Therefore, we introduced a new *materialization* operator, which will be added by the *PQ-CompilerOptimizer* on top of the base query plan whenever needed. Therefore, we extended the *ST-CompilerOptimizer*

(4) Building Feedback Channels: The created query plan is now ready for execution. The three base queries are event-driven queries, i.e., once a new event arrives to their input streams, they update their outputs and produce a new result. Although this execution will produce correct results for the entire query, it is an optimized execution. The reason is that starting the 2nd and 3rd base queries without having any output from State 1 would be wasting of resources. An important optimization that STEPQ offers is the feedback mechanism by which the automata states can control the execution of the base queries. Therefore, the 2nd and 3rd base queries will be controlled by sig-

nals from the 1st and 2nd automata states, respectively. In this case, they start producing results only when the preceding state produces a result.

5 QUERY EXECUTION

After the query plan has been formed, the execution starts by consuming raw spatio-temporal events by the base queries and producing event sets to the pattern query. The *PM-ExecutionEngine* is responsible for the execution and maintenance of the automata and their manipulation operations. The key tasks during the execution are:

(1) Handling of Event Sets: Each answer set produced from a base query is treated as an *event set E* conforming to a specific known schema S_E. The set E carries a tag that identifies the target automata states. When *PM-ExecutionEngine* receives an event set, it will forward it—depending on its tag—to the state(s) of interest. Currently, the forwarding mechanism is performed in a straightforward manner, in which we scan all states to find the ones having a matching tag. However, for scalability issues, we plan in the future to support indexing the automata states based on their tags. And hence, only the target states will be retrieved for execution. Once a state receives an input event, it updates its local information and may change the query's state.

(2) Extensible Execution of Automata States: The construction of an automaton and the behavior of its states is typically defined at compilation time based on the query operators. When start executing, it is possible that several states be running simultaneously and consuming or producing results. Although our long term plan is to provide a rich set of event operators, we strongly believe that a degree of extensibility is essential for broader applicability of the system. Therefore, we abstract the definition of an automata state using a set of properties that define its structure and behavior, e.g., internal functionality, input stream(s), output stream, the transition to next state, etc.

We categorize the properties of each state into *Structural Properties* and *Behavioral Properties*. The structural properties define the inputs and outputs of a given state along with their schemas including: (i) Right input stream R with schema S_R (always exist), (ii) Left input stream L with schema S_L (only for binary states), and (iii) Output stream O with schema S_O. If the state is binary, then its left input stream is basically the output stream of the previous states. The behavioral properties define the functionality and actions of the state and can be abstracted using the five interface functions presented in Figure 3.

The *Initialize()* function initializes a state when it is first created. If the state is binary, it may receive its first input event set from the previous state as a parameter during the creating time. The *RightStatusUpdate()*, and *LeftStatusUpdate()* functions update the state's information whenever a new event set arrives either from the right, or left input streams, respectively. The *ForwardCondition()* specifies, depending on the current state information, whether or not the state is ready to produce an output event to the next state. This function is executed automatically after each call to the status-update functions. Finally, the *ForwardAction()* function is executed only if *ForwardCondition()* returns True to produce an output event from this state and pass it to the next state.

The interface functions are generic functions created by either of the following two means: (**(1)** Automated Creation− Given a high-level query (Refer to the examples in Section 3.2), the system automatically generates the corresponding automata states along with their interface functions to implement the query's semantics. (**(2)** Manual Creation− In the cases where the system-defined operators fall short in expressing certain complex patterns, the query developers may supply these interface functions as black-box pluggable modules to define the query's behavior.

Example 5: Consider State 1 in Figure 2. This state should keep track of the cars reported continuously from the base query over an interval T. Thus, the state maintains a local data structure, e.g., a hash table, consisting of two columns: the carId and the timestamp at which this entry has been created. Different automata states may maintain different data structures to efficiently perform their intended functionalities, e.g., the hash table is a query-specific structure to perform fast lookup for its entries. The basic functionality of each of the interface functions presented in Figure 3 is as follows:

`Initialization()`: This function builds the hash table with no entires.

`RightStatusUpdate()`: This function takes the input event sets produced from the corresponding range query, and for each event, say for car id = x, the function checks if x is not in the hash table, then a new entry is created for x with the current timestamp. If x is in the hash table, then nothing will be done. Moreover, to check for continuity, each carId that is in the hash table and not in the current event set, will be removed from the table (because it is not continuously over the speed for interval T).

`LeftStatusUpdate()`: This function will be empty since the state is a unary state.

Interface Function	Triggering Time	Description
Void Initialize(Schema:S_L metaData)	Once when the state is first created.	Initializes the state information. If it is a binary state, then it may receive metadata from its previous state following schema S_L.
Void RightStatusUpdate(Schema:S_R E)	When a new event set E arrives from the right input stream (following schema S_R).	Updates the state information based on the newly arrived event set.
Void LeftStatusUpdate(Schema:S_R E)	When a new event set E arrives from the left input stream (following schema S_L).	Updates the state information based on the newly arrived event set (Left streams exist only for binary states).
Boolean ForwardCondition()	After each execution of RightStatusUpdate() or LeftStatusUpdate() functions.	Checks the state information and returns True if a new output event should be produced from the state. Otherwise, returns False.
Schema:S_O ForwardAction()	Called when ForwardCondition() function returns True	Creates a new output event set following schema S_O and pass it to the next state.

Figure 3: Extensible Interface functions of the automata states.

`ForwardCondition()`: Given the current time, this function checks each entry in the hash table, and if an entry has been in the table for interval T or more, the function returns True. Otherwise, it returns False.

`ForwardAction()`: This function removes each entry from the hash table with interval larger than or equal to T, and forms an output event set to be sent for the second state.

6 QUERY SYNCHRONIZATION

In some cases the base queries need to be coordinating their execution and not running in isolation of each other, i.e., queries may require synchronization and joint processing over the generated events. For example, the query in Example 4 requires synchronization of two concurrent *kNN* queries as illustrated in Figure 4. Notice that this query cannot be represented by, for example, two consecutive states because automata states can enforce sequential execution but not synchronized execution. In order to support this type of query processing we extended *PQ-CompilerOptimizer* with new features, which include: identifying the base queries to be jointly processed, determining the best way of synchronizing their executions, and jointly processing their outputs.

• **Determining Synchronization Mechanism:** Synchronized queries need to produce results almost simultaneously. Therefore, synchronization mechanisms among spatio-temporal queries are needed. For example, synchronization can be *time-based*, i.e., queries execute every Δ time units, or *event-based*, i.e., queries execute whenever a triggering event initiates the execution of any of the synchronized queries. In the current version of the system, we support time-based synchronization, and in the future we plan to also support event-based synchronization. For example, the query in Figure 4, the two base queries are getting the common kNN restaurants for two moving cars. Thus, the execution of these two queries will be synchronized, i.e., when one query executes, the other one will be also triggered. As will be explained in sequel, synchronizing the execution timing alone does not guarantee the outputs will be produced at the same time.

• **Synchronization Operator:** Even though the execution of the base queries will be synchronized, e.g., *time-based* or *event-based* synchronization, there are still no guarantees that the base queries will produce results at the same time. For example, one query can be more complex and takes more time than the other queries. Therefore, we introduce a new synchronization operator that blocks its output until it receives all inputs coming from the underlying synchronized base queries. The synchronization operator is a *blocking N-ary* operator that is ready to produce output once all its inputs are received. For its output side, given an index i, where $1 \leq i \leq N$, the operator returns its ith event set. For example, the query in Figure 4 uses a binary synchronization operator that pipelines either of its two inputs to the next operators on request.

• **Building Transformation Plan:** The results from the synchronized base queries need to be transformed and jointly processed to produce a single stream of event sets. The *PQ-CompilerOptimizer* is responsible for building the needed transformation plan on top of the synchronized queries at compile time and passing that plan to *ST-ExecutionEngine* for execution. Since each event set produced from a base query can be viewed as a relational table, i.e., it has a schema and several events following this schema, then a powerful and practical mechanism for the joint processing and transformation is the use of relational operators, e.g., select, project, join, grouping, and set

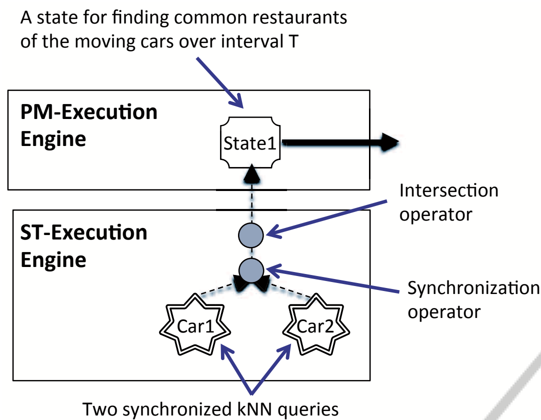


Figure 4: Synchronized Execution Plan for Example 4 Query.

operators. Hence, in a given a spatio-temporal pattern query, users can express the required transformation using standard SQL queries over the synchronized event sets. The transformation operators are typically deployed on top of the synchronization operator to ensure that the input relations are simultaneously present. For example, in Figure 4, the transformation plan contains only an intersection operator between the 1st and 2nd arguments of the downstream synchronization operator. However, in general, it may contain joins, grouping and aggregation, duplicate elimination, etc.

7 RELATED WORK

Dedicated spatio-temporal databases (STDBs), e.g., (Behr and Guting, 2005; Dieker and Guting., 2000; Xiong et al., 2005), have been proposed and flourished over the past three decades to efficiently process spatio-temporal data streams. STDBs address unique characteristics and requirements such as: **(1)** objects are typically moving in space and time, and hence these two dimensions (space and time) require special processing, **(2)** queries themselves can be moving queries, e.g., a moving car or person, and hence special algorithms are needed for efficient evaluation, **(3)** spatial-aware query operators and data structures are needed for efficient processing and fast access, and **(4)** the number of object and queries (both can be stationary or moving) can be very large, and hence scalable processing and incremental evaluation is a must.

Within STDBs, a flurry of research activities and optimizations have sprung up including: continuous and incremental query evaluation (Chen and Patel, 2007; Choi and Chung., 2002; Mokbel et al., 2004b), spatial-aware operators (Elmongui et al., 2005; Mokbel et al., 2004a), spatial indexes (Benetis et al., 2002; Xiong et al., 2006), and shared execution (Dieker and Guting., 2000; Hu et al., 2005). Other existing techniques focus on detecting moving clusters or phenomena over time (Marios Hadjieleftheriou ad George Kollios and Tsotras., 2003; Ali et al., 2007), privacy-aware location services (Cheng et al., 2006; Yiu et al., 2009), and approximate processing of *range* and *kNN* queries (Kanoulas et al., 2006; Mouratidis et al., 2005). However, none of these techniques is applicable to answer the complex pattern queries presented in Examples 1 & 2. The reason is that although these techniques can efficiently build correlations and detect patterns over the raw spatio-temporal streams, they are not designed to detect more complex patterns over the results from spatio-temporal queries. Thus, either of the *application-level* or *middleware-level* approaches need to be integrated with the above techniques, which is not an efficient solution as presented in Section 2.

On the other hand, complex event processing (CEP) has been extensively studied in active databases (Carey et al., 1988; Chakravarthy et al., 1994), publish/subscribe systems (Aguilera et al., 1999; Fabret et al., 2001), and data stream systems (Arasu et al., 2003; Chandrasekaran et al., 2003; Cugola and Margara, 2012). Query languages and operators have been proposed to express pattern queries and to correlate events over time, e.g., (Wu et al., 2006; Lerner and Shasha, 2003). However, these systems use only the *Time* dimension as the driving dimension, and hence they lack all the research technologies established in spatio-temporal DBs over the past decades. As a result, given the raw spatio-temporal streams generated from the moving objects, CEP systems can handle neither the traditional spatio-temporal queries nor the more-complex pattern queries illustrated in Examples 1 & 2.

The spatio-temporal pattern queries addressed in this paper has some similarity to the continuous queries in data stream systems (Nehme and Rundensteiner., 2006; Mukherji et al., 2008) in that both types of queries are continuously executing over streams of data. However, a key difference is that the current technology in continuous queries cannot optimize or control the sources creating the data streams. In contrast, in the proposed query type (spatio-temporal pattern queries), the intermediate sources of the data streams are spatio-temporal queries that can be controlled and optimized. As highlighted in Section 2, loosely integrating the STDBs with CEP or stream processing systems is not an efficient solution as it will miss critical optimizations, synchronization among queries, and unnecessary overheads. In con-

trast, the proposed STEPQ system is a single coherent engine that overcomes these limitations.

8 EXPERIMENTAL EVALUATION

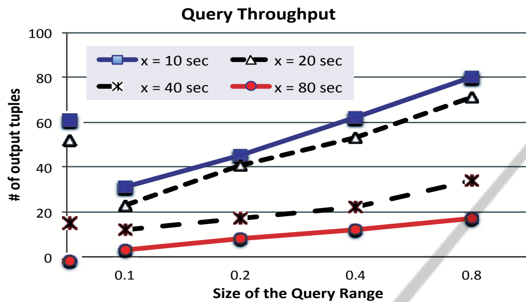


Figure 5: Throughput from Query Q1.

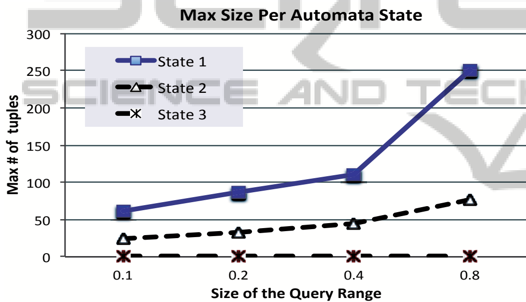


Figure 6: Buffer Size in Automata States in Query Q1.

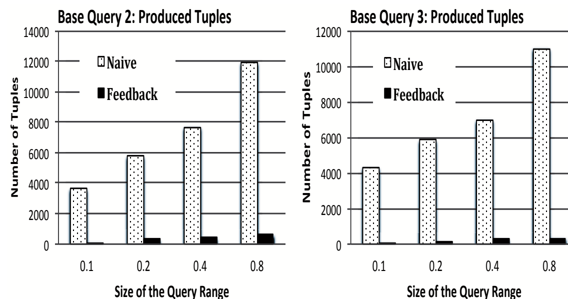


Figure 7: Naive vs. Feedback Base Query Execution in Q1.

STEPQ is implemented using the TerraLib module on top of PostgreSQL. TerraLib is an extension to the database systems to support spatio-temporal data types and queries. We extended the system by adding the *PQ-CompilerOptimizer* and *PM-ExecutionEngine* components (Refer to Figure 1). The experiments are conducted using an AMD Opteron Quadputer compute server with two 16-core AMD CPUs, 128GB memory, and 2 TBs SATA hard drive. We study the performance of the system using two queries; $Q1$ (presented in Example 1), and $Q4$ (presented in Example 4).

Data Sets: We use the *Network-Based Generator of Moving Objects* (Brinkhoff and Str, 2002) to generate a set of 50,000 moving objects. For Query $Q1$, we labeled 10% of the moving objects as *suspicious criminals*, and we labeled 5% of the map area as *school areas*, and each school area has another nearby suspicious area, i.e., another 5% of the map area. The execution of $Q1$ requires three base queries (spatio-temporal range queries) over areas $A1$, $A2$, and $A1$ again, and on top of that we have three automata states, each one checking the persistency condition in its area (Similar to the query plan in Figure 2). In the experiments, specifically in Figure 7, we will clarify the reason behind not merging the first and last base queries together.

Query $Q1$ has two key parameters, which are the range size around the *school* and *suspicious* areas, and the time interval x that represents the persistency over a period of time. In Figure 5, we varied these two parameters. The range size varies over the range between 0.1 and 0.8 miles around the above mentioned areas, and the persistency interval varies over the values of 10, 20, 40, and 80 seconds. In the experiment, we measured the query throughput over a simulated 20-mins run. The experiment in performed 5 times and the average numbers are reported in the figure. The results show that as the size of the query range increases, the number of matched tuples also increase but at a linear rate, i.e., the range size is increasing exponentially, but the number of produced tuples is increasing linearly. This is partially due to the persistency condition that imposes additional constraint over the reported data. But more importantly due to the transitions over different states as will be discussed in Figure 6. On the other side, the query throughput is inversely proportional to the persistency interval x, i.e., as x increases, less number of the tuples satisfy the query semantics as depicted in Figure 5.

In Figure 6, we study the maximum number of tuples buffered in each of the query states at any given point in time during the query. The query ($Q1$) has three states monitoring the transitions of the *criminals* over the *school* and *suspicious* areas. In the experiment, we fix the persistency interval to 40 seconds, and vary the size of the query range as illustrated in the figure. The results illustrate that for State 1, the number of buffered tuples is increasing exponentially as the range size also increases exponentially. However, this trend does not continue to States 2 and 3 because only small subset of the tuples satisfying State 1 move to State 2, and then a smaller number satisfies State 2 conditions and move to State 3.

In Figure 7, we study the effect of the cross-cutting optimizations between the automata states and

the base queries generating the events. In a naive approach (labeled "Naive" in the figure), each query would report the tuples within each range, and then that will be filtered by the automata state based on its buffered data. This approach corresponds to the baseline techniques, e.g., a CEP algorithm loosely stacked over a STDB system. In this case, no optimizations can be performed across the queries. In contrast, STEPQ can provide feedback from the automata states to the base queries to control their execution. Figure 7 shows that the number of intermediate tuples in the states' buffers is very large and proportionally increasing as the query size increases. Most of these tuples are not actually needed because whatever base queries 2 and 3 report are not necessarily satisfied State 1 in the automata. In contrast, when STEPQ establishes a feedback from State 1 output to feed the 2nd base query, and the same between State 2 output and the 3rd base query (labeled "Feedback" in Figure 7), the number of tuples reported from these base queries is reduced dramatically. This is because, each base query is only looking for and reporting the tuples that are already satisfied the preceding automata state.

In Figure 8, we measured the effect of intermediate tuples and the extra time needed for processing them. In the experiment, we allow the system to first run for 10 mins to reach a steady state, and then we measure the execution time with each new event. We collect this measurement for 100 consecutive executions, and present the average of these measurements in Figure 8. The results show that under the naive approach the system send significant time processing intermediate tuples not contributing to the final answer. With the feedback mechanism in STEPQ, the execution time is reduced by an 8x factor.

The performance of $Q4$ is studied in Figures 9 and 10. This query is distinct from $Q1$ in that it requires a synchronization and transformation operators over the two involved base queries (Refer to Figure 4). In Figure 9, we study the overheads involved in these two operations (for each execution) during the query lifetime, i.e., we measure the time taken by either of the synchronization or transformation over each execution, and then we report the average value. As illustrated in the figure, the synchronization is more expensive than the transformation operator as it has to wait to receive the input from both queries, and then pass them to the next operator. The results also confirm that although both base queries are triggered at the same time, they are not necessarily producing the outputs at the same time. The transformation operator in our example is a simple *intersect* operator that involves low overhead as depicted in Figure 9.

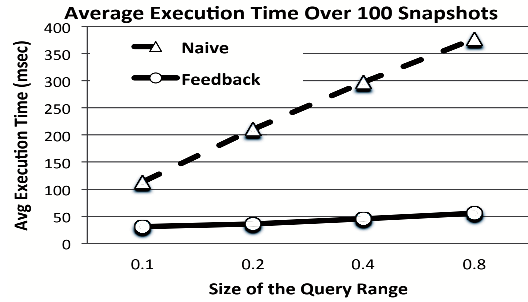


Figure 8: Avg Snapshot Execution Time in Query Q1.

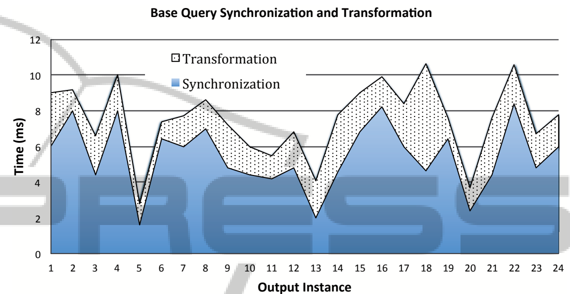


Figure 9: Synchronization & Transformation Overheads (in each execution) in Query Q4.

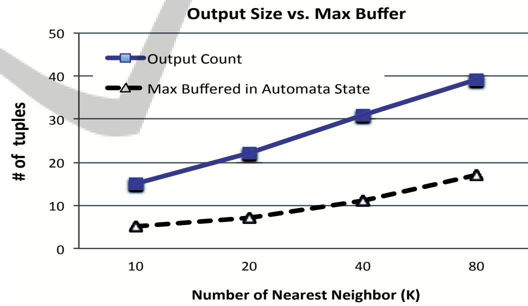


Figure 10: Throughput and Buffer Size in Query Q4.

The experiment in Figure 10 study the effect of varying the parameter K (the number of nearest neighbors reported from each query) on the number of tuples produced in the output or buffered in the automata state for the persistency test. As expected, as K increases the chances of finding intersection between the produced output from the base queries are increases. The *Output Count* (blue line) indicates the total number of reported tuples of the simulated run for 15 minutes. In contrast, the *Max Buffered* (dotted black line) indicate the maximum number of tuples buffered in the automata state before either reporting or discarding each one. This results show that the memory requirements of the automata state in this query is very small.

9 CONCLUSION

We presented the STEPQ system for efficiently managing and executing complex spatio-temporal pattern queries. The unique features of STEPQ include: coherent integration between spatio-temporal queries and CEP queries, centralized module for optimizing both query types and providing cross-cutting optimization between them, and the extensibility feature that enables expressing complex patterns beyond the built-in operations. The system is still under development and we are working on adding additional extension and optimizations, which include: extending its language, shared execution among pattern queries, and optimizing synchronized queries.

REFERENCES

Adaikkalavan, R. and Chakravarthy, S. (2003). SnoopIB: Interval-based event specification and detection for active databases. In *Proceedings of ADBIS*, pages 190–204.

Aguilera, M., Strom, R., Sturman, D., Astley, M., and Chandra, T. (1999). Matching events in a content-based subscription system. In *Proceedings of Principles of Distributed Computing*.

Ali, M. H., Mokbel, M. F., and Aref, W. G. (2007). Phenomenon-aware Stream Query Processing. In *Proceedings of the International Conference on Mobile Data Management, MDM*.

Arasu, A., Babu, S., and Widom, J. (2003). CQL: A language for continuous queries over streams and relations. In *DBPL*, pages 1–19.

Behr, T. and Guting, R. H. (2005). Fuzzy Spatial Objects: An Algebra Implementation in SECONDO. In *In Proceedings of the International Conference on Data Engineering, ICDE*, page 1137 1139.

Benetis, R., Jensen, C. S., Karciauskas, G., and Saltenis., S. (2002). Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. In *Proceedings of the International Database Engineering and Applications Symposium, IDEAS*, pages 44–53.

Brinkhoff, T. and Str, O. (2002). A framework for generating network-based moving objects. *Geoinformatica*, 6:2002.

Cai, Y., Hua, K. A., and Cao., G. (2004). Processing Range-Monitoring Queries on Heterogeneous Mobile Objects. In *Proceedings of the International Conference on Mobile Data Management, MDM*.

Carey, M., Livny, M., and Jauhari, R. (1988). The HiPAC project: Combining active databases and timing constraints. *SIGMOD Record*, 17(1).

Chakravarthy, S., Krishnaprasad, V., Anwar, E., and Kim, S. (1994). Composite Events for Active Databases: Semantics, Contexts and Detection. In *VLDB*, pages 606–617.

Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M., Hellerstein, J., Hong, W., and et al. (2003). TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*.

Chen, Y. and Patel, J. M. (2007). Efficient Evaluation of All-Nearest-Neighbor Queries. In *In Proceedings of the International Conference on Data Engineering, ICDE*, page 10561065.

Cheng, R., Zhang, Y., Bertino, E., and Prabhakar., S. (2006). Preserving User Location Privacy in Mobile Data Management Infrastructures. In *In Proceedings of Privacy Enhancing Technology Workshop*.

Choi, Y.-J. and Chung., C.-W. (2002). Selectivity Estimation for Spatio-temporal Queries to Moving Objects. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, page 440451.

Cugola, G. and Margara, A. (2012). Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62.

Demers, A., Gehrke, J., Hong, M., Riedewald, M., and et al. (2006). Towards expressive publish/subscribe systems. In *EDBT*, pages 627–644.

Dieker, S. and Guting., R. H. (2000). Plug and Play with Query Algebras: SECONDO- A Generic DBMS Development Environment. In *Proceedings of the International Database Engineering and Applications Symposium, IDEAS*, page 380392.

Elmongui, H. G., Mokbel, M. F., and Aref., W. G. (2005). Spatio-temporal Histograms. In *Proceedings of the International Symposium on Advances in Spatial and Temporal Databases, SSTD*, page 1936.

Fabret, F., Jacobsen, H., Llirbat, J., Ross, K., and Shasha, D. (2001). Filtering algorithms and implementation for very fast publish/subscribe systems. In *SIGMOD*, pages 115–126.

Gedik, B. and Liu., L. (2004). MobiEyes: Distributed Processing of Continuously Moving Queries on Moving Objects in a Mobile System. In *Proceedings of the International Conference on Extending Database Technology, EDBT*.

Gehani, N., Jagadish, H., and Shmueli, O. (1992). Composite Event Specification in Active Databases: Model and Implementation. In *VLDB*, pages 327–338.

Hu, H., Xu, J., and Lee., D. L. (2005). A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects. . In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, page 479490.

Kanoulas, E., Du, Y., Xia, T., and Zhang., D. (2006). Finding Fastest Paths on A Road Network with Speed Patterns. . In *Proceedings of the International Conference on Data Engineering, ICDE*.

Lerner, A. and Shasha, D. (2003). AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments. In *VLDB*, pages 345–356.

Marios Hadjieleftheriou ad George Kollios, D. G. and Tsotras., V. J. (2003). On-Line Discovery of Dense Areas in Spatio-temporal Databases. . In *Proceedings of the International Symposium on Advances in Spatial and Temporal Databases, SSTD*, page 306324.

Mokbel, M. F., Xiong, X., and Aref., W. G. (2004a). SINA: Scalable Incremental Processing of Continuous

Queries in Spatio-temporal Databases. . In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, page 443454.

Mokbel, M. F., Xiong, X., Hammad, M. A., and Aref, W. G. (2004b). Continuous query processing of spatio-temporal data streams in place. In *STDBM*, pages 57–64.

Mouratidis, K., Papadias, D., and Papadimitriou., S. (2005). Medoid Queries in Large Spatial Databases. . In *Proceedings of the International Symposium on Advances in Spatial and Temporal Databases, SSTD*, page 5572.

Mukherji, A., Rundensteiner, E. A., Brown, D. C., and Raghavan, V. (2008). SNIF TOOL: sniffing for patterns in continuous streams. In *CIKM*, pages 369–378.

Nehme, R. and Rundensteiner., E. (2006). SCUBA: Scalable Cluster-Based Algorithm for Evaluating Continuous Spatio-Temporal Queries on Moving Objects. In *Proceedings of the International Conference on Extending Database Technology, EDBT*, page 10011019.

Shahabi, C., Kolahdouzan, M. R., and Sharifzadeh., M. (2003). A Road Network Embedding Technique for K-Nearest Neighbor Search in Moving Object Databases. . *GeoInformatica*, 7(3):255273.

Wolfson, O., Sistla, A. P., Xu, B., Zhou, J., and Chamberlain., S. (1999). DOMINO: Databases for Moving Objects tracking (Demo). In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, page 547549.

Wu, E., Diao, Y., and Rizvi, S. (2006). High-performance complex event processing over streams. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 407–418.

Xiao, D. and Eltabakh, M. (2013). STEPQ: Spatio-temporal Engine for Complex Pattern Queries. In *International Conference on Advances in Spatial and Temporal Databases (SSTD)*, pages 386–390.

Xiong, X., Mokbel, M. F., and Aref., W. G. (2005). SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-temporal Databases. . In *Proceedings of the IEEE International Conference on Data Engineering, ICDE*, page 643654.

Xiong, X., Mokbel, M. F., and Aref., W. G. (2006). LU-Grid: Update-tolerant Grid-based Index- ing for Moving Objects. . In *Proceedings of the International Conference on Mobile Data Management, MDM*, pages 13–21.

Yiu, M. L., Ghinita, G., Jensen, C. S., and Kalnis, P. (2009). Outsourcing Search Services on Private Spatial Data. In *ICDE*, pages 1140–1143.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS