STEPQ: Extensible Spatio-Temporal Engine for Complex Pattern Queries

by

Mohamed Eltabakh
STEPQ: Extensible Spatio-Temporal Engine for Complex Pattern Queries

Mohamed Eltabakh
Worcester Polytechnic Institute
100 Institute Rd., Worcester, MA
meltabakh@cs.wpi.edu

ABSTRACT
With the increasing complexity and wide diversity of spatio-temporal applications, the query processing requirements over spatio-temporal data far exceed the traditional query types beyond range, kNN, and aggregation queries along with their variants. Most spatio-temporal applications in domains from traffic monitoring, transportation and emergency services, surveillance to and healthcare systems require support for evaluating powerful spatio-temporal pattern queries (STPQs) that form higher-order correlations and compositions of sequences of events to infer real-world semantics of importance to the targeted application. STPQs can be supported by neither traditional spatio-temporal databases (STDBs) nor by modern complex-event-processing systems (CEP). While the former lack the expressiveness and processing capabilities for handling such complex sequence pattern queries, the later mostly focus on the Time dimension, and hence lack the power of the special-purpose processing technologies established in STDBs over the past decades. In this paper, we propose to develop an efficient, scalable spatio-temporal engine for complex pattern queries (STEPQ). STEPQ has several innovative features that will advance the research not only in spatio-temporal databases, but also in complex event processing. First, the project is unique in addressing, in a fundamental way, complex pattern queries over spatio-temporal data currently overlooked by state-of-the-art systems. Second, the proposed approach of the extensible architecture for pattern-matching queries is applicable not only to spatio-temporal pattern queries, but also to complex event processing techniques in general. Third, the project addresses several novel optimization strategies that arise from the integration of spatio-temporal and pattern-matching techniques into one integrated query processing technology. With the STEPQ system currently under development, we sketch several example scenarios demonstrating the applicability of this technology.

1. INTRODUCTION
Several emerging and life-critical applications inherently depend on spatio-temporal data processing including traffic monitoring and transportation systems [29], surveillance systems [23], geographic information systems (GIS) [27], location-based services (LBS) [6], healthcare systems [25], and environmental sciences [30]. The recent advances and widespread popularity of mobile devices, wireless cellular phones, and Global Positioning Systems (GPS) have enabled these applications to continuously monitor and track all objects of interest. With such continuous streams of spatio-temporal data being produced and the increasing complexity and wide diversity of spatio-temporal applications, the query and data exploration requirements of these applications now reach far beyond the traditional spatio-temporal query types, e.g., range, k nearest-neighbor (kNN), and aggregation queries, e.g., [10, 11, 13, 14], to more expressive and semantics-rich spatio-temporal pattern queries (or STPQ). Examples of these powerful new query types include: “Q1: Report child-abuse criminals who stay in school area A1 for more than x minutes and then move to another suspicious area A2 within one hour”, “Q2: Report the kNN cars that are continuously getting closer to my moving car over interval T”, and “Q3: Send an alert to patient P, e.g., patient with a weak immune system, if (s)he stays in contact (within distance D for at least interval T) with another patient having a transferable disease”. See Table 1 for more STEPQ examples.

Evidently, spatio-temporal pattern queries are prevalent in many applications as they capture real-world semantics that otherwise would have been lost or delegated to the application layer for ad-hoc and inefficient processing. It is not meaningful to assume that a suspicious criminal activity in Q1 or the alert condition for a patient in Q3 depend solely on a single data instance (or even snapshot) of the data stream - rather separate snapshots of instances in the high-speed stream must be trapped at the right moments of time and synchronized to determine the correct match of such a complex STPQ query. In spite of this complexity, the real-time processing of spatio-temporal pattern queries is imperative for a wide range of mission-critical applications.

In this paper, we propose the STEPQ system—Spatio-Temporal Engine for complex Pattern Queries—that addresses the unique challenges of handling STPQ queries, including: (1) They embed powerful semantics not captured by current spatio-temporal query types, (2) Unlike traditional query types, such as range or kNN queries, that can be evaluated on each instance of the database in isolation (e.g., a stream query window), STPQs require correlation among spatio-temporal events (both in time and in space) over multiple instances of the database, (3) They require a full-fledged query engine equipped not only with efficient event-processing techniques but also with effective spatio-temporal processing capabilities, and (4) Unlike state-of-the-art event-processing techniques that assert no control over the input stream of...
Table 1: Examples of spatio-temporal pattern queries (STPQs).

<table>
<thead>
<tr>
<th>Query</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>Report child-abuse criminals who stay in a school area A1 for more than x minutes and then move to a suspicious area A2 within one hour (E.g., suspicious criminal activity).</td>
</tr>
<tr>
<td>Q2</td>
<td>Report cars that stay in my kNN over interval T and continuously are getting closer to my moving car.</td>
</tr>
<tr>
<td>Q3</td>
<td>Send alert to patient P, if she stays in contact (within distance D for at least interval T) with a patient having a transferable disease (E.g., health threat).</td>
</tr>
<tr>
<td>Q4</td>
<td>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 (E.g., testing effect of radar signs over A2 on drivers’ behavior).</td>
</tr>
<tr>
<td>Q5</td>
<td>Report restaurants located in kNN of two moving cars and getting closer to both cars over interval T (E.g., find common nearby restaurants in direction of moving cars).</td>
</tr>
</tbody>
</table>

This table provides examples of spatio-temporal pattern queries (STPQs).
to generate events, but also synchronizing their execution. For example, Query Q5 in Table 1 requires synchronizing the execution of two moving kNN queries, i.e., they should produce results almost simultaneously and then intersecting their results. Such synchronization and transformation over the event streams are not feasible in the middleware-level approach and not even supported by current STDBs.

3. OVERVIEW OF PROPOSED SYSTEM

Given the above limitations, it is clear that engineering existing systems to handle STPQs is not the right approach. The proposed STEPQ system, on the other hand, is a coherent system that addresses all of the above limitations with extensible interfaces that enables expressing complex and diverse query semantics. The architecture of STEPQ is summarized as follows (See Figure 1(c)). The system 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 processed 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. Regarding the PM-ExecutionEngine, our plan is that pattern-matching queries will be evaluated using a variant of Non-deterministic Finite Automata (NFA) as well as a set of operators that manipulate the automata results. The choice of NFAs is based on their efficiency and flexibility in answering pattern-matching queries as in SASE [31], Cayuga [16, 15], and SnoopyDB [2, 1]. The PM-ExecutionEngine will leverage the work from these systems while including substantial and core extensions crucial to STPQs such as extended event model, abstract automata interfaces for extensibility, and new event operators.

The main characteristics of STEPQ system are:

- Leveraging & extending state-of-art in STDBs: This is achieved by the ST-CompilerOptimizer and ST-ExecutionEngine components that retain all the innovations in STDBs such as continuous and incremental evaluation, spatial-aware operators and access methods, and scalable execution. In addition to that, base queries will be subject to several new optimizations triggered by the PQ-CompilerOptimizer.

- Coherent integration between spatio-temporal and pattern-matching techniques: This is achieved by having a single system with interacting components orchestrated by the PQ-CompilerOptimizer. The PQ-CompilerOptimizer along with the newly introduced operators, e.g., materialization, synchronization, and transformation operators, will ensure seamless interaction and communication between the base and pattern-matching queries.

- Cross-cutting optimizations: This is achieved by the PQ-CompilerOptimizer component that enables PM-ExecutionEngine to provide feedback information to ST-ExecutionEngine to control the execution of the base queries depending on the progress of the pattern queries. Cross-cutting optimizations are essential for system-wide scalability and efficiency, otherwise base queries can be continuously and unnecessarily running to produce unused results.

- Event Model and Extensible Language for Pattern Queries: This is achieved by the ST-ExecutionEngine and PM-ExecutionEngine components. The core extensions introduced over existing systems include: (1) Extending the event model with a new concept called event sets that provides logical grouping of events produced from the base queries. An event set is the unit of processing in STEPQ containing a set of simultaneous events (either primitive or compos-
4. RESEARCH ISSUES AND CHALLENGES

In this section, we highlight in more details some of the research issues addressed by STEPQ.

4.1 Integrated Spatio-Temporal and Pattern Queries

PQ-CompilerOptimizer is the heart of the proposed system as it is responsible for compiling (and ultimately optimizing) the entire STPQ. Given a STPQ, PQ-CompilerOptimizer decomposes the query and generates the needed base and pattern-matching queries as illustrated in Figure 2. To couple the base queries with the automata states in the pattern queries, PQ-CompilerOptimizer uses a tagging scheme in which a tag will be assigned to each base query, the query’s output, and the corresponding automata states receiving this output. 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.

It is also possible that different STPQs may involve the same base query. In this case, PQ-CompilerOptimizer should detect that a single base query can be re-used to feed multiple automata states. Sharing partial execution among multiple base queries is also feasible but it requires communication between ST-CompilerOptimizer and PQ-CompilerOptimizer where the former decides on the shared components while the latter decides on the tagging scheme to appropriately couple the outputs from each query to the automata states—Recall that ST-CompilerOptimizer is not aware of the pattern-matching subqueries.

Another important issue is that most STDBs support incremental evaluation of results for efficiency purposes, e.g., [18, 21, 22]. Therefore, there are two options for handling incremental updates in STEPQ system; 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 to decouple the pattern-matching subqueries from the way the base queries are evaluated. Thus, we introduced a new materialization operator that will added by the PQ-CompilerOptimizer on top of the base-query plans whenever needed to fully materialize the output before passing it to the PM-ExecutionEngine.

4.2 Synchronized-Query Processing

A single STPQ may require not only executing multiple base queries, but also synchronizing their execution and jointly processing their results. For example, Query Q5 in Table 1 requires synchronization of two concurrent kNN queries and then intersecting their outputs as illustrated in Figure 2(b). Notice that Q5 cannot be represented by, for example, two consecutive states because automata states can enforce sequential execution but not synchronized execution. In contrast, Query Q4 can be represented by a single automaton having three states plus three base range queries but without the need for synchronization as illustrated in Figure 2(a).

STEPQ supports two types of synchronization mechanisms, i.e., time-based where queries execute every Δ time units, or event-based where queries execute whenever a triggering event initiates the execution of any of the synchronized queries. The synchronization type is either provided as a used-defined parameter or left for PQ-CompilerOptimizer as an optimization decision. In either cases, synchronizing the execution still does not guarantee that the base queries will produce results at the same time, e.g., 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. This operator is a blocking N-ary operator that is ready to produce output only when all its inputs are received. For its output side, given an index i, where 1 ≤ i ≤ N, the operator returns its ith input set. For example, Query Q5 in Figure 2(b) uses a binary synchronization operator that pipelines either of its two inputs to the next operators on request. In addition to the synchronization operator, base queries may go out-of-sync for some reason, and hence new mechanisms will be investigated to monitor the base queries, make sure their execution is correctly aligned, and re-synchronize queries when needed.

Typically, the outputs from the synchronized queries need to be transformed and jointly processed to produce a single stream of event sets. STEPQ allows any relational plan, e.g., select, project, join, grouping, and set operators, to be
deployed on top of the the synchronized queries. For example, Query Q5 in Figure 2(b) uses an intersection operator to intersect the results from the two underlying events and produce a single stream of event sets. Relational operators will be extended to operate on (and read their inputs from) the synchronization operator.

### 4.3 Extensibility for Complex Pattern Queries

As indicated in Section 1, spatio-temporal applications are very diverse, and hence providing a set of event operators such as those in SASE [31], Cayuga [16, 15], and SnooIPB [2, 1], may fall short in expressing complex semantics. Therefore, in addition to the system-provided event and automata operators, we design STEPQ with abstract and extensible interfaces through which users can plug in user-defined functions that define how a query pattern evolves. Extensibility has proven to be a desired property in database systems, e.g., [3, 8], indexing frameworks, e.g., [5, 17], and query optimization [19, 7].

STEPQ abstracts the definition of an automata state using a set of properties categorized 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. All the input and output events follow specific schemas as depicted in Figure 3. In the cases where the provided operators fall short in expressing certain complex patterns, the system will enable users to supply these interface functions as black-box pluggable modules. $PM-ExecutionEngine$ and the entire system should operate seamlessly regardless of how these interfaces are provided.

### 4.4 Scalability and Optimizations

The design and architecture of STEPQ allows possible optimizations between its components that cannot be performed otherwise. In this section, we highlight few of these optimizations through examples.

**Example 1:** Given Query Q4 illustrated in Figure 2(a), the execution of the second and third range queries over areas A2 and A3 should be initiated only if there are matching cars from the previous states. There are two possible approaches to apply such optimization. The first approach requires little communication between the $PM-ExecutionEngine$ and $ST-ExecutionEngine$ where each automata state reports back its status as being idle or not to its corresponding base query. A state is idle if it is not monitoring any objects, and hence not interested in getting any input events. In this case, the corresponding base query will be suspended until it gets another notification that its corresponding state is no longer idle. For example, the second state in Q4 remains idle until it receives an input from the first state. Once the objects in the second state are deleted, the state becomes idle again until it receives the next input from the first state. The second approach requires more communication between the system components where each state may send back information regarding the events it is currently interested in. This information is incorporated in the base queries as dynamic filters that are continuously changing. Although it is more efficient, the dynamic filtering requires more careful design and precise communication between the system components.

**Example 2:** The number of automata states executed by the $PM-ExecutionEngine$ can be very large, moreover, the number of generated events from the base queries can be also large. Therefore, one obvious optimization is to index the automata states based on their tag labels (The tags that link the base queries to their corresponding automata states). Hence, when a new event set arrives with tag $T$, all automata states having the same tag can be efficiently retrieved without triggering the other states. Another more interesting optimization is derived from the following scenario. Assume two different automata states $S_1$ and $S_2$ which are monitoring police cars, and ambulance cars, respectively, in the same area $A1$. Then, one trivial, but inefficient, solution is to have two separate range queries over $A1$; one for each automata state. A better solution is to merge the two queries into one range query that reports both police and ambulance cars, and then each state filters out the un-desired events. However, this solution also has drawbacks since both $S_1$ and $S_2$ states will be always triggered whenever a new event set comes from the base query even if the event set has only police or ambulance cars. To overcome this inefficiency, we extend the state properties to have tuning properties, in addition to the structural and behavioral properties presented in Section 4.3.

---

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

![Figure 3: Interface functions of the automata states.](image-url)
One of the tuning properties is a filter predicate that indicates the individual events of interest within each event set feeding the state. The PQ-CompilerOptimzer may add these filter predicates as an optimization decision, even if they are not defined in the input query, to enable merging multiple base queries into one without sacrificing scalability or efficiency. Given this tuning property, automata states will be triggered only if there are events of its interest within the event set. For efficiency and scalability, PM-ExecutionEngine needs to index the states based on their filter predicates, in addition to their tags. Empty event sets also require careful consideration and optimization. Any automata state that require persistence, e.g., tracking a pattern over a period of time, needs to receive each event set from its base query even if it is empty. Such forwarding of empty event sets should be optimized to avoid triggering automata states unnecessarily.

5. CONCLUSION AND FUTURE PLAN

In this paper, we proposed the STEPQ system for efficient and scalable processing of spatio-temporal pattern queries. Unlike the traditional range, kNN, and aggregation spatio-temporal queries, STEPQ can form higher-order correlations among events and capture real-world semantics of interest. New architectural design has been proposed to enable full-fledged integration and optimization between spatio-temporal and pattern-matching queries. The design is also based on extensible interfaces to enable expressing queries beyond the provided operators. STEPQ will not only enhance the querying capabilities of countless spatio-temporal applications but also initiate new research directions in the areas of spatio-temporal data and complex event processing.

Currently, STEPQ is in its early stage of development. Our group is investigating the functionalities of PLACE opensource spatio-temporal server [26] from which we will inherit key aspects for the ST-CompilerOptimzer and ST-ExecutionEngine components and then extend them as needed. The second step is the realization of the PM-ExecutionEngine which is fundamentally different from the existing CEP systems because of its extensibility, but will still leverage many of the operators proposed in [31, 16]. The final step is the realization of the PQ-CompilerOptimzer which is the heart of STEPQ as described in the paper.

6. REFERENCES