E-Cube: Multi-Dimensional Event Sequence Analysis Using Hierarchical Pattern Query Sharing

Mo Liu, Elke A. Rundensteiner, Kara Greenfield
Worcester Polytechnic Institute, Worcester, MA 01609, USA
(liumo|rundenst|kgreenfield)@cs.wpi.edu
Chetan Gupta, Song Wang, Ismail Ari‡, Abhay Mehta
USA Hewlett-Packard Labs, USA
‡Ozyegin University, Turkey
(chetan.gupta|songw|abhay.mehta)@hp.com ‡Ismail.Ari@ozyegin.edu.tr

ABSTRACT

Many modern applications, including online financial feeds, tag-based mass transit systems and RFID-based supply chain management systems transmit real-time data streams. There is a need for event stream processing technology to analyze this vast amount of sequential data to enable online operational decision making. Existing techniques such as traditional online analytical processing (OLAP) systems are not designed for real-time pattern-based operations, while state-of-the-art Complex Event Processing (CEP) systems designed for sequence detection do not support OLAP operations. We propose a novel *E-Cube* model which combines CEP and OLAP techniques for efficient multi-dimensional event pattern analysis at different abstraction levels. Our analysis of the interrelationships in both concept abstraction and pattern refinement among queries facilitates the composition of these queries into an integrated *E-Cube* hierarchy. Based on this *E-Cube* hierarchy, strategies of drill-down (refinement from abstract to more specific patterns) and of roll-up (generalization from specific to more abstract patterns) are developed for the efficient workload evaluation. Our proposed execution strategies reuse intermediate results along both the concept and the pattern refinement relationships between queries. Based on this foundation, we design a cost-driven adaptive optimizer called *Chase*, that exploits the above reuse strategies for optimal *E-Cube* hierarchy execution. Our experimental studies comparing alternate strategies on a real world financial data stream under different workload conditions demonstrate the superiority of the *Chase* method. In particular, our *Chase* execution in many cases performs ten fold faster than the state-of-the-art strategy for real stock market query workloads.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

SIGMOD'11, June 12–16, 2011, Athens, Greece.

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Complex Event Processing, OLAP, Streaming, Optimization, Algorithm

1. INTRODUCTION

1.1 Motivation

There are numerous emerging applications, such as online financial transactions, IT operations management, and sensor networks that generate real-time streaming data. This streaming data has many dimensions (time, location, objects) and each dimension can be hierarchical in nature. One important common problem over such data is to be able to analyze multiple pattern queries that exist at various abstraction levels in real-time.

One example is data from transportation systems. In many metropolitan areas such as London, Moscow and Beijing, mass transit agencies issue their passengers near-field contactless (NFC) or contact-based smart cards for fast payment and convenient access to metros, buses, light-rails, and places such as museums. In addition to people's movements, these agencies are also beginning to continuously track the position and status of their vehicles. The collected data continuously flows to a central location in the form of structured event streams for storage. Unfortunately, their analysis lags. Officials are demanding tools that can help them analyze the current status of these complex systems in real-time and over different abstractions levels. Such knowledge would enable them to make strategic decisions about issues such as resource scheduling, route planning, variable pricing, etc. However today, they can only obtain aggregate (weekly, or even monthly) statistics through offline analysis, thus missing critical opportunities that could be gained via real-time analysis.

Another example is an evacuation system where RFID technology is used to track mass movement of people and goods during natural disasters. Terabytes of RFID data could be generated by such a tracking system. Facing a huge volume of RFID data, emergency personnel need to perform pattern detection on various dimensions at different granularities in real-time. In particular, one may need to monitor people movement and traffic patterns of needed resources (say, water and blankets) at different levels of abstraction to ensure fast and optimized relief efforts. Figure 1 lists several sample "pattern queries" for such a scenario. For example, during hurricane Ike federal government personnel may monitor

movement of people from cities in Texas to Oklahoma represented by the pattern SEQ(TX, OK) for global resource placement as in q_1; while local authorities in Dallas may focus on people movement starting from the Dallas bus station, traveling through the Tulsa bus station, and ending in the Tulsa hospital within a 48 hours time window as in q_5 to determine the need for additional means of transportation. The rest of the queries in Figure 1, including the concepts of negation, predicates and query hierarchy refinements, will be elaborated upon later in Sections 2 and 3.

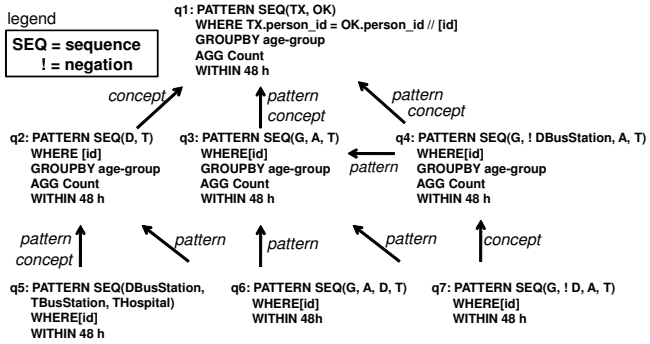


Figure 1: Sample pattern queries organized hierarchically (see Section 2.).

Common across the above scenarios is a need to process and query large volumes of streaming sequence data in real-time at various abstraction levels. This is exactly the problem we tackle in this paper. Detecting complex patterns in high-rate event streams requires substantial CPU resources. The authors in [9] observe that with increasing stream arrival rates and large operator states, the computing resources typically become strained before the memory does. Temporary data flushing [19] and highly efficient compressed data representations make a memory-limited scenario less likely. Therefore, our E-Cube solution targets the efficient processing of workloads of complex pattern detection queries at multiple levels of abstraction over extremely high-speed event streams by effectively leveraging their CPU resource utilization.

E-Cube leverages two existing technologies, OLAP and CEP. Traditional OLAP aims to provide answers to analytical queries that are multi-dimensional in nature via aggregation [4, 15, 11]. Complex Event Processing (CEP) systems demonstrate sophisticated capabilities for pattern matching [3, 5, 29] in real-time by processing huge volumes of complex stream data. However, these technologies by themselves are not always sufficient. Current CEP systems don't support queries over different concept abstraction levels. In addition, they don't support the efficient computation for multiple such queries at different concept and pattern hierarchies concurrently. In short, state-of-the-art CEP systems do not support OLAP operations, and thus are not suitable for multi-dimensional event analysis at different abstraction levels. The state-of-art OLAP solutions [24, 10, 14] either don't support real-time streams at all, or they do not tackle CEP sequence queries. Hence, in the context of event streams where the order and sequence of events are important, OLAP is insufficient in supporting efficient event sequence analysis. Section 8 further discusses deficiencies of the state of art.

1.2 The E-Cube Approach and Contributions

Combining OLAP and CEP technologies requires both theoretical and practical contributions. On the theoretical front, we develop the solid foundation of a combined concept and pattern hierarchy. On the practical front, we present a methodology to efficiently process queries on streaming data over this hierarchy. Specifically, we make the following technical contributions:

- We are the first to analyze the interrelationships between pattern queries in terms of both concept abstractions and pattern refinements given a workload of event pattern queries. The analysis then facilitates the design of the novel E-Cube model – which integrates complex pattern queries specified using sequence, negation and concept abstractions into an E-Cube hierarchy. OLAP-like operations allow users to navigate through the E-Cube hierarchy to extract results at different levels of abstraction and refinement.

- We design several re-use based evaluation strategies for online operational decision making, the first of its kind in the context of real-time multi-dimensional sequence analysis. Namely, the *specific-to-general (resp. general-to-specific)* reuse method evaluates patterns at the finest (resp. coarsest) level of abstraction first, and then moves up (resp. down) to evaluate coarser (resp. finer) level pattern queries. In this context, we address challenges related to partial sharing of subpatterns and extraction of non-matches via event negation.

- A cost model to estimate the relative costs of different re-use classes is designed and verified within our E-Cube system. We design a cost-driven query optimizer, called Chase, capable to selectively exploit specific-to-general and general-to-specific evaluation strategies to determine an overall evaluation ordering for maximal re-use. Chase is shown to deliver optimal results.

- We implement all strategies using the Chaos stream system as a testbed [12]. Our experimental study conducted on real world data sets compares the performance of the proposed methods and the traditional non-shared strategy, and determines their respective scope of applicability. The superiority and robustness of the E-Cube optimizer, Chase, is demonstrated.

The rest of the paper is organized as follows: Section 2 provides background on event pattern query processing. Section 3 introduces the design details of our E-Cube model and operations. Section 4 describes our optimal algorithm called Chase for E-Cube evaluation. Section 5 introduce our reuse-based pattern evaluation strategies. Section 6 presents plan adaption. Section 7 shows the evaluation results. Section 8 discusses related work while Section 9 concludes the paper.

2. CEP BASICS

An **event instance** is an occurrence of interest denoted by lowercase letters (e.g.,'e'). An event instance can be either *primitive* (smallest, atomic occurrence of interest) or *composite* (a list of constituent primitive event instances). We use $e_i.ts$ and $e_i.te$ to denote the start and the end time-stamps of the event instance e_i, respectively. For a primitive event instance, $e_i.ts = e_i.te$. For compactness the time of occurrence of a primitive event instance is denoted by its subscript i. Composite event instances occur over an interval. For example, one result of query q_6 in Figure 1 is a composite event instance $e = < g_1, a_2, d_3, t_5 >$ with four primitive event instances of event types Galveston, Austin, Dallas, and Tulsa readings, respectively where $e.ts = 1$ and $e.te = 5$.

An **event type** E of an instance e_i describes the essential features associated with the event instance e_i denoted by $e_i.type$. An

event type can be either primitive or composite [3]. *Primitive event types* are pre-defined in the application domain of interest (such as GBusStation-reading). *Composite event types* are created by composing primitive event types as further explained below.

DEFINITION 1. *A* **SEQ operator** *specifies an order on the time-stamps in which the instances of specific event types must occur to match the pattern and thus form a composite event instance [20].*

$$SEQ(E_1, E_2, ..., E_n) = \{< e_1, e_2, ..., e_n > | e_1.ts < e_2.ts < ... < e_n.ts \wedge (e_1.type = E_1) \wedge (e_2.type = E_2) \wedge ... \wedge (e_n.type = E_n)\}. \quad (1)$$

The symbol "!" before an event type E_i indicates that the instance of type E_i is not allowed to appear within the specified position in the stream [29]. We call such E_i a *negative* event type. Consequently, when an event type E_i is used in a SEQ construct without "!", we call it a *positive* event type.

$$SEQ(E_1, !E_i, E_n) = \{< e_j, e_k > | (e_j.ts \leq e_j.te < e_k.ts \leq e_k.te) \wedge (e_j.type = E_1) \wedge (e_k.type = E_n) \wedge (\neg \exists e_f \ where (e_f.type = E_i) \wedge (e_j.te < e_f.ts \leq e_f.te < e_k.ts))\}. \quad (2)$$

Beyond Equation 2, negative event types can also exist in the beginning or the end in a SEQ operator. For details see [29]. Other unordered (i.e., set-based) event pattern operators such as conjunctions (AND) and disjunctions (OR) can be defined in a similar manner [25]. Expressions with unordered event pattern operators can be rewritten into a normal form composed of AND and SEQ operators [22]. Compositions of SEQ operators can also be used to generate more complex patterns, but for brevity we leave extensions to nested queries as future work here. Instead, we henceforth focus on sequential pattern queries denoted by SEQ and their multi-dimensional analysis in this paper.

In the following, we briefly present the query language adopted from the literature [29] that is also utilized in our running example (see Figure 1). The overall structure of the language is specified as follows:

```
PATTERN <event pattern>
[WHERE <qualification>]
[GROUPBY <groupby specification>]
[AGG <aggregation function>]
[WITHIN <window>]
```

The PATTERN clause uses the SEQ operator to specify sequence and negation constraints, as explained above. The WHERE clause contains predicates on attributes of the events in question, such as transport ids, ages, etc. The GROUPBY clause corresponds to a list of event types from the event pattern P_l (such as Groupby E_i, E_j) and/or event attributes (such as Groupby age-group (young/senior)). The AGG clause specifies an aggregation function such as COUNT on the number of results in a given group. The WITHIN clause ensures that the time difference between the first to the last event instances matched by a pattern query falls within the window. Examples of queries specified using this language can be found in Figure 1.

In the literature, handling queries with different predicates, aggregates and window sizes has been addressed by previous research using sliced time windows and shared data fragments [28, 17, 18]. In this paper, we instead focus on the combination of pattern and concept hierarchies as in Section 3.

State-of-the-art Stack Based Pattern Query Evaluation. First, each pattern query q_i is compiled into a query plan. Beyond commonly used relational-style operators like select, project, join, group-by and aggregation, we support the Window Sequence operator, denoted by WinSeq(E_1 ,..., E_n, window). It extracts all matches of instances within the sliding stream window as specified in query q_i. Queries with negative event types, denoted by WinSeq(E_1 ,..., ! E_i ,..., E_n, window), verify that no event instances of negative components such as E_i exist in the indicated location among the positive instance matches.

An indexing data structure named *SeqState* associates a stack with each event type in the query. Each received event instance is appended to the end of its corresponding stack. Event instances are augmented with pointers ptr_i to adjacent events to facilitate quick locating of related events in other stacks during result construction. An event instance e_m of the last event type E_m of a query q_i arrives, the compute function of q_i is initiated. The result construction is done by a depth first search along instance pointers ptr_i rooted at that last arrived instance e_m. All paths composed of edges "reachable" from this root e_m are enumerated. Each such root-to-leaf path corresponds to one matched event sequence returned for q_i. When negative event types are specified in WinSeq, then during sequence construction any edges "reachable" from the root e_m are skipped if an instance of the negative event type is found in the corresponding stream position. Outdated events based on window constraints are purged. The cost model for stack-based pattern query evaluation described above is presented in [23].

EXAMPLE 1. *Figure 2 shows the event instance stacks for the pattern query q_3 = SEQ(G, A, T)). In each stack, its instances are naturally sorted from top to bottom by their timestamps. When t_{15} of type $Tulsa$ arrives, the most recent instance in the previous stack of type $Austin$ is a_6. The pointer of t_{15} is a_6, as shown in the parenthesis preceding t_{15}. As $Tulsa$ is the last event type in q_3, t_{15} triggers result construction. Two results $<g_1, a_5, t_{15}>$ and $<g_1, a_6, t_{15}>$ are constructed involving t_{15}.*

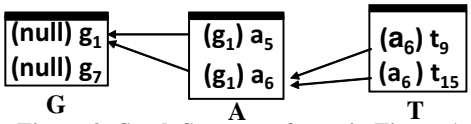


Figure 2: Stack Structure for q_3 in Figure 1

3. E-CUBE MODEL

Based on the CEP query model introduced in Section 2, we now define our E-Cube model. A concept hierarchy is commonly used to summarize information at different levels of abstraction [13]. Here, we focus on event specific features and thus on concept hierarchies over event types. A concept hierarchy applies to primitive event types in the same way as it applies to other concepts in the literature [13]. Event concept hierarchies for primitive event types are predefined by system administrators using domain knowledge.

DEFINITION 2. *An* **event concept hierarchy** *is a tree where nodes correspond to event types. The most specific event types reside at the leafs of the tree, while progressively more general event types reside higher and higher in the tree, with the most general event type residing at the apex of the tree. An event type E_k that is a descendent (resp. ancestor) of an event type E_j in an event concept hierarchy is at a finer (resp. coarser) level of abstraction than E_j, denoted by $E_k <_c E_j$ (resp. $E_k >_c E_j$)*

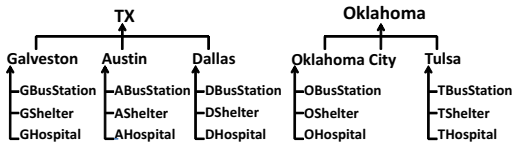


Figure 3: Concept Hierarchy of Primitive Event Types

Figure 3 shows an example event concept hierarchy for primitive event types in our RFID-based tracking scenario. We can use different dimensions to create event types that belong to a *concept hierarchy*. For example, event types in our sample application incorporate semantics of both geographical locations and service station types (hospital, bus, shelter) into one hierarchy. Event instances can be interpreted to be of types at different abstraction levels in such an event concept hierarchy. For example, an instance of type DBusStation can also be interpreted to be of the more coarse types Dallas or TX. The refinement relationships among composite event types are defined by Definitions 3 and 4. A financial concept hierarchy is given later in Figure 11.

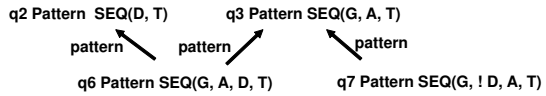


Figure 4: Pattern Hierarchy

DEFINITION 3. **Query Concept Refinement.** *A pattern query q_k = SEQ(E_{1k} ,..., ! E_{hk} ,..., E_{mk}) is* **coarser** *than q_j = SEQ(E_{1j} ,..., ! E_{hj} ,..., E_{mj}), denoted by $q_k >_c q_j$, if (I) for all negative event types E_{hk} and E_{hj}, $E_{hj} >_c E_{hk} \vee E_{hj}.type == E_{hk}.type$ and (II) for all positive event types E_{ik} and E_{ij}, $E_{ik} >_c E_{ij} \vee E_{ik}.type == E_{ij}.type$ and (III) for $1 \leq l \leq m$, $\exists$ (E_{lk}, E_{lj}) such that $E_{lk}.type \neq E_{lj}.type$.*

The non-existence (existence) of a negative (positive) event type at a coarser (finer) concept level enforces more constraints as compared to a negative (positive) event type at a finer (coarser) concept level. In Figure 1, q_1 is at a coarser concept level than q_2, denoted by $q_1 >_c q_2$ because TX $>_c$ D and OK $>_c$ T. q_4 is at a coarser concept level than q_7, denoted by $q_4 >_c q_7$, as the negative type D in q_4 is coarser than DBusStation in q_7 (D $>_c$ DBusStation).

DEFINITION 4. **Query Pattern Refinement.** *A pattern query q_k = SEQ(E_{1k} ,..., E_{ik} ,..., E_{mk}) is* **coarser** *than q_j = SEQ(E_{1j} ,..., E_{ij} ,..., E_{nj}), denoted by $q_k >_p q_j$, if (I) $\forall$ $E_{lk} \in q_k$, $\exists$ $E_{hj} \in q_j$ with $E_{lk}.type == E_{hj}.type$ and (II) $\forall$ (E_{lk}, E_{tk}) pairs $\in q_k$ with $l < t$, then must $\exists$ (E_{vj}, E_{wj}) pair $\in q_j$ with $v < w$ such that $E_{lk}.type == E_{vj}.type$ and $E_{tk}.type == E_{wj}.type$ and (III) $\exists$ E_{vj} such that $E_{vj} \in q_j$, $E_{vj} \notin q_k$.*

In other words, we can roll-up a pattern q_k to a coarser (finer) level by deleting (inserting) one or more event types from (into) q_k. For example, in Figure 4, which contains a subset of the SEQ queries from Figure 1, the pattern query q_3 is at a coarser level than q_6, denoted by $q_3 >_p q_6$, because q_6 enforces the existence of more event types and associated sequential event relationships than q_3. Similarly, the pattern query q_3 is at a coarser level than q_7, denoted by $q_3 >_p q_7$, because q_7 includes one extra negative event type D. All event types in Figure 4 are at the same concept level, but at different levels in the pattern hierarchy.

DEFINITION 5. *An* **E-Cube hierarchy** *is a directed acyclic graph H where each node corresponds to a pattern query q_i and each edge corresponds to a pairwise refinement relationship between two pattern queries as defined in Definitions 3 and 4. Each directed edge $<q_i, q_j>$ is labeled with either the label "concept" if $q_i <_c q_j$ by Definition 3, "pattern" if $q_i <_p q_j$ by Definition 4 or both to indicate the refinement relationship among the two queries q_i and q_j.*

Definition 5 says that a pattern query q_i can be rolled up into another pattern query q_j by either changing one or more positive (negative) event types to a coarser (finer) level along the event concept hierarchy of that event type (by Def. 3), changing the pattern to a coarser level (by Def. 4), or both. Figure 1 shows an example E-Cube hierarchy. The E-Cube hierarchy helps us to achieve better performance in multi-query evaluation because it provides a blueprint for shared online pattern filtering and rapid result sharing, as will be explained in Section 5.

DEFINITION 6. **E-Cube** *is an E-Cube hierarchy (see Definition 5) where each pattern query is associated with its query result instances. Each individual pattern query along with its result instances in E-Cube is called an* **E-cuboid**.

Operations on E-Cube. We propose an extension of OLAP operations, namely, pattern-drill-down, pattern-roll-up, concept-roll-up and concept-drill-down for pattern queries in our E-Cube hierarchy. OLAP-like operations on E-Cube allow users to navigate from one E-cuboid to another in E-Cube.

[Pattern-drill-down] The operation pattern-drill-down(q_m, list [$Type_{ij}$, Pos_{kj}]) applied to q_m inserts a list of n event types with the event type $Type_{ij}$ into the position Pos_{kj} of q_m (1 $\leq$ j $\leq$ n).

[Concept-drill-down] The operation concept-drill-down(q_m, list [($Type_{mj}$, $Type_{nj}$), Pos_{kj}]) applied to q_{mj} drills down a list of event types from $Type_{mj}$ to $Type_{nj}$ ($Type_{mj} >_c Type_{nj}$) at the position Pos_{kj} of q_m (1 $\leq$ j $\leq$ n).

[Pattern-roll-up] The operation pattern-roll-up(q_m, list[$Type_{ij}$, Pos_{kj}]) applied to q_m deletes a list of n event types with the event type $Type_{ij}$ from the position Pos_{kj} of q_m (1 $\leq$ j $\leq$ n).

[Concept-roll-up] The operation concept-roll-up(q_m, list[($Type_{mj}$, $Type_{nj}$), Pos_{kj}]) applied to q_m rolls up a list of event types from $Type_{mj}$ to $Type_{nj}$ ($Type_{mj} <_c Type_{nj}$) at the position Pos_{kj} of q_m (1 $\leq$ j $\leq$ n).

EXAMPLE 2. *In Figure 1, we apply a pattern-drill-down operation on q_3 = SEQ(G, A, T) specified by pattern-drill-down(q_3, [(!D, 2)]) and we get q_7 = SEQ(G, !D, A, T). We can apply a concept-drill-down operation on q_1 = SEQ(TX, OK) specified by concept-drill-down(q_1, [(TX, D, 1)]) and we get q_2 = SEQ(D, T). Similarly, we apply a pattern-roll-up operation on q_6 = SEQ(G, A, D, T) specified by pattern-roll-up(q_6, [(G, 1), (A, 2)]) and we get q_2 = SEQ(D, T). Also, we apply a concept-roll-up operation on q_2 = SEQ(D, T) by concept-roll-up(q_2, [(D, TX, 1)]) and we get q_1 = SEQ(TX, OK).*

The results of pattern-drill-down (pattern-roll-up) can be computed by our general-to-specific (specific-to-general) reuse with only pattern changes as introduced in Section 5.1 (Section 5.4). The results of concept-drill-down (concept-roll-up) can be computed by our general-to-specific (specific-to-general) evaluation with only concept changes as introduced in Section 5.2 (Section 5.5).

Hierarchical Event Storage. We design compact *hierarchical instance stacks (HIS)* to hold event instances processed by E-Cube. HIS provides *shared storage* of events across different concept and pattern abstraction levels. Each instance is stored in only one single stack even though it may semantically match multiple event types in an event type concept hierarchy, namely, the finest one in E-Cube hierarchy. HIS is populated with event instances as the stream data

is consumed. The stack based query evaluation in Section 2 could be easily extended to access event instances in hierarchical stacks instead of flat stacks.

4. OPTIMAL E-CUBE EVALUATION

Our objective is to produce query results quickly and improve computational efficiency by sharing results among queries in a unified query plan. Instead of processing each pattern in our E-Cube hierarchy independently using the stack-based strategy explained in Section 2, we now design strategies to compute one pattern from other previously computed patterns within the E-Cube hierarchy.

More precisely, we set out to exploit the concept and pattern relationships between queries identified by the E-Cube model to promote reuse and to reduce redundant computations among queries. In particular, we consider two orthogonal aspects as in the table below, namely, (1) abstraction detection: drill down vs. roll up in E-Cube hierarchy, and (2) refinement type: pattern or concept refinement. More precisely, we consider the following cases: (a-b) general-to-specific with only pattern or concept changes respectively; (c) general-to-specific with simultaneous pattern and concept changes; (d-e) specific-to-general with only pattern or concept changes respectively; (f) specific-to-general with simultaneous pattern and concept changes.

	Direction of Reuse	
Refinement Type	General→Specific	Specific→General
Pattern Only	Section 5.1	Section 5.4
Concept Only	Section 5.2	Section 5.5
Both Refinements	Section 5.3	Section 5.6

Given a workload of pattern queries, our E-Cube system will first translate them into an E-Cube hierarchy H, and then design a strategy to determine an optimal evaluation ordering for all queries in the E-Cube hierarchy such that the total execution cost is minimized. To achieve our goal of finding the best overall execution strategy for the complete workload captured by the E-Cube hierarchy, we consider three choices when evaluating each query q_i in H;

- (I) compute q_j independently by stack-based join, denoted by $C_{compute(qj)}$;
- (II) conditionally compute q_j from one of its ancestors q_i by general-to-specific evaluation, denoted by $C_{compute(qj|qi)}$;
- (III) conditionally compute q_j from one of its descendants q_i by specific-to-general evaluation, denoted by $C_{compute(qj|qi)}$.

C_{qi} represents the computation cost which is either $C_{compute(qi)}$ or $C_{compute(qi|qj)}$ for some q_i in H. We will analyze all pairwise opportunities and detailed physical strategies of how to achieve reuse in each case along with cost models in Sections 5.

4.1 Problem Mapping to Weighted Directed Graph

Given the three alternatives (I), (II) and (III) described above, a valid execution ordering of a query workload expressed by an E-Cube hierarchy H is defined as below.

DEFINITION 7. *An* **execution ordering** *O_i(H) for queries in an E-Cube hierarchy H represents a partial order of n computation strategies for the n queries in H, $O_i(H) = < O_{i1}, ..., O_{ij}, ..., O_{in} >$ such that for $1 \leq j \leq n$, O_{ij} selects one of the three computation strategies (I), (II) or (III) for a query $q_j \in H$. If q_j's computation method is a conditional computation $C_{compute(qj|qi)}$ then q_i must be listed before q_j in O_i. Each query q_j is computed exactly once. Each execution ordering O_i(H) for H has an associated computation cost, denoted by Cost(O_i(H)) as shown in Equation 3.*

$$Cost(O_i(H)) = \sum_{j=1}^{n, q_j \in H} C_{q_j}$$
$$\text{where } C_{q_j} \text{ is equal to the cost to compute } q_j \text{ as selected by } O_{ij}; \quad (3)$$

For an execution ordering O_i(H), each query q_j in H is either computed from scratch or from another query q_i in H. Put differently, each query q_j has one and only one computation source. Thus clearly no computation circles can exist in an O_i(H) ordering. Let us prove this by contradiction. Given two queries q_i and q_j, assume q_i were computed from q_j and q_j were computed from q_i. Then no q_i and q_j results could ever be computed as the two queries would deadlock waiting indefinitely to compute results from each other.

DEFINITION 8. *The* **optimal execution ordering***, denoted by O-opt(H), is the execution ordering O-opt such that $\forall$ i, Cost(O-opt(H)) $\leq$ Cost(O_i(H)) with Cost() defined in Equation 3.*

PROBLEM 1. *Given an E-Cube hierarchy H, the E-Cube optimization problem is to find an optimal execution ordering O-opt(H) for all queries in H as defined in Definition 8.*

We now illustrate that the E-Cube optimization problem as defined in Problem 1 can be mapped into a well-known graph problem. Given this re-formulation as shown in Definition 9, we can reuse solutions from the literature to efficiently find an optimal solution to our problem.

DEFINITION 9. **Graph Mapping.** *Given an E-Cube hierarchy H, we define a directed weighted graph G = (V, E) where |V| = |queries $\in$ H| + 1; |E| = 2 $\times$ |edges $\in$ H| + |queries $\in$ H|. A mapping from the graph H to G, m: H $\rightarrow$ G, is defined as follows:* **(I)** *$\forall$ $q_i \in H$, there is a one-to-one mapping to one vertex v_i in G. To include the option of self-computation into G, we add one special vertex v_0 as root into V, called virtual ground.* **(II)** *$\forall$ $<q_i, q_j>$ refinement relationships in H, there exist two edges e(v_i, v_j) and e(v_j, v_i) $\in$ E. $\forall$ $v_i \in G$ where $v_i \neq v_0$, we insert a directed edge e(v_0, v_i) into E to model that node v_i is computed from "the ground" v_0 (i.e., from scratch).* **(III)** *Computation costs are assigned as weights on each corresponding directed edge according to our cost model (see Section 5 and Appendix). Each directed edge e(v_0, v_i) $\in$ E is assigned an associated weight w(v_0, v_i) equal to $C_{compute(qi)}$ (choice I). Each directed edge e(v_i, v_j) $\in$ E with $v_i \neq v_0$ and $v_j \neq v_0$ is assigned a weight w(v_i, v_j) to denote $C_{compute(qj|qi)}$ (choices II/III).*

LEMMA 1. *All pattern and concept refinement relationships in H along with their respective computation costs are captured as edges and weights in the graph G, respectively. All possibilities of self-computation for all queries in H, along with their respective computation costs, are captured as edges and weights in the graph G.*

Proof Sketch: All independent and conditional computation relationships are captured by directed edges between vertices. Computation costs are attached to these directed edges. Thus all possible

alternative solutions of computing all queries in H are now represented by G.

EXAMPLE 3. *Figure 5(a) shows the weighted directed graph G for modeling the E-Cube hierarchy H shown in Figure 1. Each vertex with the number i denotes the query q_i from Figure 1. In total, eight nodes are created in the graph G representing q_1-q_7 and the virtual ground v_0. The arrow labeled with 12 from the virtual ground to v_3 represents the fact that the cost to compute q_3 from scratch is 12. The arrow labeled with 5 from v_1 to v_3 represents the fact that the cost to compute q_3 from q_1 is 5.*

4.2 Solution for Optimal Execution Ordering

After constructing the directed graph G, Lemma 2 and Theorem 2 are defined as below to solve Problem 1.

LEMMA 2. *After mapping an E-Cube hierarchy H to a weighted directed graph G by Definition 9, an optimal execution ordering $O_i(H)$ for H is equal to a minimum cost spanning tree MST over G.*

Proof: Consider a directed graph, G(V, E), where V and E are the set of vertices and edges, respectively. Associated with each edge e(v_i, v_j) is a cost weight w(v_i, v_j). The MST problem is to find a rooted directed spanning tree MST of G such that the sum of costs associated with all edges in the MST is the minimum cost among all possible spanning trees. An MST is a graph which connects, without any cycle, all vertices of V in G with |V| - 1 edges, i.e., each vertex, except the root, has one and only one incoming edge. For the optimal execution ordering O-opt(H), except the virtual ground v_0 (root), every query (vertex) has one and only one computation source modeled by an incoming edge in MST. By Definition 7, no computation circles exist in O-opt(H). For each of the |V| - 1 queries (virtual ground not included), one computation source (incoming edge) is selected. |V| - 1 edges are selected such that the sum of computation costs (edge associated costs) is the minimum among all possible execution ordering O_i(H). In summary, finding an optimum execution plan with lowest cost for H is equivalent to finding an MST in G [8, 6].

THEOREM 1. *Solving Problem 1 for an E-Cube hierarchy H is equivalent to solving the MST problem for the corresponding G created by the mapping from H defined by Definition 9.*

Proof sketch: Proof naturally follows from Lemma 2.

Since there are many solutions in the literature for solving the well-known minimum spanning tree MST graph problem, any of these MST algorithms that works on (cyclic) directed graphs could be applied. Our optimizer, called Chase (Cost-based Hybrid Adaptive Sequence Evaluation), applies the Gabow algorithm [8] in detecting the MST over a directed graph. The pseudocode for our Chase strategy is given in Figure 6. Line 02 in Figure 6 applies the Gabow algorithm [8]. The key idea of the Gabow algorithm is to find edges which have the minimum cost to eliminate cycle(s) if any. The algorithm consists of two phases. The first phase uses a depth-first strategy to choose roots for growth steps. The second phase consists of expanding the cycles formed during the first phase, if any, in reverse order of their contraction, discarding one edge from each cycle to form a spanning tree in the original graph. The algorithm recursively finds the tree in the new graph until no circles exist. By braking the cycle into a tree, an MST is guaranteed to be returned eventually. For details see [8].

EXAMPLE 4. *The example in Figure 5 illustrates our use of the Gabow algorithm. The algorithm finds the edge(s) which have the minimum cost to eliminate cycle(s) if any. For each vertex, the incoming edge with the minimum cost is selected (bold arrow) in Figure 5(a). We observe that vertices representing queries q_3, q_4 and q_7 form a circle in the Gabow algorithm. In Figure 5(b), we observe that the edge from vertex 1 to the cycle has the minimum cost among all the ingoing edges to the circle. And vertex 1 points to vertex 3 in the cycle. Thus, the contraction technique finds the minimum cost replacing edge e(4, 3) by edge e(1, 3). Hence the cycle is eliminated.*

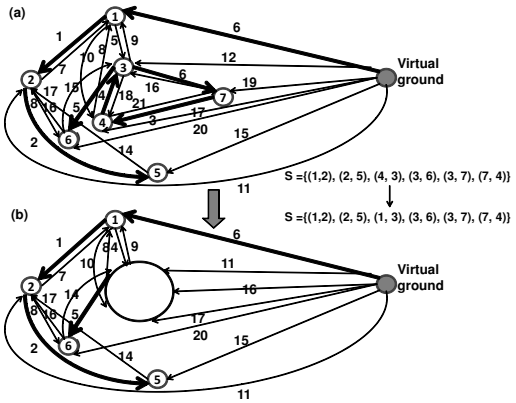


Figure 5: Use of Gabow Algorithm in our Optimal Solution

```
Chase Evaluation (
Q={q1 , ..., qi , ..., qn}-Queries;
Wij- The weight for the edge from vi to vj;
Rqi-Results of qi)
01 G graph = DirectedGraphConstruction(Q)
   // construct weighted directed graph for Q (Section 4.1)
02 MinimumSpannigTree(G, w) (Section 4.2);
   i = 0;
   // compute optimum execution ordering
   // and store in optArray
03 while(i <= optArray.size)
04 {if (compute qi independently)
05 compute qi by stack-based join
06 if(compute qi from its child qj)
07 compute qi by specific-to-general
   (Sections 5.4 5.5 5.6)
08 if(compute qi from its parent qk)
09 compute qi by general-to-specific
   (Sections 5.1 5.2 5.3)
10 cache Rqi; i++; }
```

Figure 6: Chase Executor

THEOREM 2. *The execution ordering decided by our Chase executor (Figure 6) is guaranteed to find the optimal solution for the E-Cube optimization defined in Problem 1.*

Proof sketch: Since the MST algorithm [8] is guaranteed to find the optimal MST solution, so is Chase.

THEOREM 3. *The time complexity of the Chase algorithm is O(E + VlogV) [8].*

Proof sketch: As we map our optimization problem into the MST problem, the complexity of our Chase strategy is the same as that of the MST algorithm we deploy [8].

Chase automatically yet efficiently optimizes the execution of a set of queries in E-Cube. Doing this operation manually would not only be time consuming but also difficult for humans to detect the optimal solution for larger E-Cube hierarchies. On the other hand, the Chase strategy clearly scales even for larger number of queries in the E-Cube hierarchy. Therefore, Chase contributes to both performance and scalability of our E-Cube system.

Table 1: Terminology Used in Cost Estimation

Term	Definition
$C_{compute(qi\|qj)}$	The evaluation cost for query q_i basing on evaluation results for q_j
$C_{compute(qi)}$	The cost of computing results for a query q_i independently
$\|S_i\|$	Number of tuples of type E_i that are in time window TW_P. This can be estimated as $Rate_E * TW_P * P_E$
TW_P	Time window specified in a pattern query P
$Rate_E$	Rate of primitive events for the event type E
P_E	Selectivity of all single-class predicates for event class E. This is the product of selectivity of each single-class predicate of E.
$Pt_{Ei,Ej}$	Selectivity of the implicit time predicate of subsequence (E_i, E_j). The default value is set to 1/2.
$P_{Ei,Ej}$	Selectivity of multi-class predicates between event class E_i and E_j. If E1 and E2 do not have predicates, it is set to 1.
$\|R_E\|$	Number of results for the composite event E
C_{type}	The unit cost to check type of one event instance
$q_i.length$	The number of event types in a query q_i
$NumE$	Number of total events received so far
$NumRE$	Number of relevant events received of the types in query set Q
C_{access}	The cost of accessing one event
C_{app}	The unit cost of appending one event to a stack and setting up pointers for the event
C_{ct}	The unit cost to compare timestamp of one event instance with another one

5. REUSE-BASED PATTERN EVALUATION STRATEGIES

We now address the six alternative scenarios of reuse indicated in Section 4 by designing customized execution strategies for query processing that maximally reuse the previously computed results. Challenges related to partial sharing of subpatterns, extraction of non-matches via event negation, and redundancy elimination are tackled. Cost models for each of the strategies are developed.

5.1 General-to-Specific with Pattern Changes

Considering only pattern changes, the computation of the lower level query can be optimized by reusing results from the upper level query. The two sharing cases are stated as below. Given queries q_i and q_j ($q_i >_p q_j$) in a pattern hierarchy and the results of q_i, then the results for q_j can be constructed as bellow. In **case I: Differ by positive types,** we join the results of q_i with the events of positive types listed in q_j but not in q_i. In **case II: Differ by negative types** we filter the results from q_i that don't satisfy the sequence constraints formed by negative event types listed in q_j but not in q_i. Figure 7 depicts the pseudocode for general-to-specific evaluation guided by the pattern hierarchy.

For **case I** above, the costs for the compute operation depend on two key factors, namely (1) if pointers exist between joining events and (2) if the re-used result is ordered or not on the joining event type. Assume two pattern queries q_i = SEQ(E_i, E_j, E_k) and q_j = SEQ(E_i, E_j, E_k, E_m, E_n) differ by two positive event types E_m and E_n. Also, let us assume pointers exist between events of type E_m and E_n. To compute q_j, we first construct results for SEQ(E_m, E_n) by an efficient stack-based join. These results will by default be sorted by E_n's timestamp. We then join these results with q_i results using the most appropriate join method. Table 1 shows the factors used in the cost estimation in Equation 4.

General-to-specific evaluation with only pattern changes (
q_i and q_j are queries in a pattern hierarchy
with $q_i >_p q_j$; R_{qi} - the results of q_i)
01 $R_{q_j} = R_{q_i}$
02 **for** every negative $E_k \in q_j$ but $E_k \notin q_i$
03 R_{q_j} = checkNegativeE(R_{q_j}, E_k, q_j)
04 **for** every positive $E_i \in q_j$ but $E_i \notin q_i$
05 **if**(joining events in R_{q_j} and E_i are sorted and pointers exist)
06 R_{q_j} = stack-based-join(R_{q_j}, E_i);
07 **else if**(events are sorted with no pointers)
08 R_{q_j} = merge-join(R_{q_j}, E_i);
09 **else** R_{q_j} = sorted-merge-join(R_{q_j}, E_i);
checkNegativeE(R_{q_j}, E_k, q_j)
01 **for** each result $r_i \in R_{q_j}$
02 **if**(E_k events exist in the specified interval) remove r_i

Figure 7: General-to-Specific Evaluation in Pattern Hierarchy

$$\begin{aligned} C_{compute(qj|qi).gp} =& |S_m| * |S_n| * Pt_{E_m,E_n} * P_{E_m,E_n} \\ &+ |R_{SEQ(E_m,E_n)}| log |R_{SEQ(E_m,E_n)}| \\ &+ |R_{qi}| * |R_{SEQ(E_m,E_n)}| * Pt_{E_k,E_m} \\ &* P_{E_k,E_m} + |R_{SEQ(E_m,E_n)}| + |R_{qi}| \end{aligned} \quad (4)$$

For **case II**, assume two pattern queries q_i = SEQ(E_m, E_n) and q_j = SEQ(E_m, ! E_k, E_n) differ by one negative event type E_k. For every q_i result, it can be returned for q_j if no E_k events are found between the particular interval in q_j. The cost formula is shown in Equation 5.

$$\begin{aligned} C_{compute(qj|qi).gp} =& |S_m| * |S_n| * Pt_{E_m,E_n} * P_{E_m,E_n} * \\ & (1 - Pt_{E_m,E_k} * Pt_{E_k,E_n}) \end{aligned} \quad (5)$$

Besides this computation sharing, we can also achieve online pattern filtering and thus potentially save the computation costs of q_i completely ($C_{compute(qi)}$). The idea is that, if a pattern q_i is at a coarser level than a pattern q_j, and a matching attempt with q_i fails, then there is no need to carry out the evaluation for q_j. That is, q_j being stricter is guaranteed to fail as well.

EXAMPLE 5. *Given pattern queries q_3, q_6 and q_7 in Figure 1, q_3 and q_6 differ by one event type D and q_3 and q_7 differ by one event type $!D$. We check the results for q_3 first. If no new matches are found, then we know that the results for q_6 and q_7 would also be negative. Thus, we can skip their evaluation. If new matches for q_3 are found, as no pointers exist between results of q_3 and events of type D. Yet the joining attributes for T and D, namely, $D.ts$ and $T.ts$ are sorted on timestamps. We thus can apply the fairly efficient merge join to compute q_6.*

5.2 General-to-Specific with Concept Changes

Considering only concept changes, composite results constructed involving events of the highest event concept level are a super set of pattern query results below it in a E-Cube hierarchy. The lower level query can be computed by reusing and further filtering the upper query results.

Given two pattern queries q_i and q_j with only concept changes ($q_i >_c q_j$) on **positive** event types, our cost model is formulated in Equation 6. For each result of q_i, we interpret the event types for the constructed composite event instances to determine which of them indeed match a given lower level type. The strategy becomes less efficient as the number of results to be re-interpreted increases.

$$C_{compute(qj|qi).gc} = |R_{qi}| * C_{type} * q_i.length \quad (6)$$

EXAMPLE 6. *In Figure 1, from q_1 to q_2 only the concept hierarchy level is changed. q_1 is computed before q_2 and the results are cached. As all results of q_2 satisfy q_1, q_2 can be computed simply by re-interpreting the q_1 results. If one result with component events of types TX and OK is also a composite event with types D and T, that particular result will be returned for q_2. Otherwise, the result will be filtered out.*

Given two pattern queries q_i = SEQ(E_m, ! E_{k1}, E_n) and q_j = SEQ(E_m, ! E_k, E_n) with only concept changes ($q_i >_c q_j$) on **negative** event types where E_k is a super concept of E_{k1} in the event concept hierarchy. To facilitate query sharing, we rewrite q_j into the expression shown in Equation 7. For every q_i result, it can be returned for q_j if no E_{k2}, E_{k3} ... and E_{kn} events are found between the position in specified query.

$$SEQ(E_m, !E_k, E_n) = SEQ(E_m, !E_{k1} \wedge ...! \wedge E_{kn}, E_n) \quad (7)$$

EXAMPLE 7. *In Figure 1, when computing q_7 from q_4 , each q_4 result is qualified for q_7 if no DHospital and DShelter events exist between G and A events.*

5.3 General-to-Specific with Concept & Pattern Refinement

Given q_i and q_j in an E-Cube hierarchy with simultaneous concept and pattern changes ($q_i >_{cp} q_j$), the cost to compute the child q_j from the parent q_i corresponds to Equation 8. The main idea is to consider this as a two step process that composes the strategies for concept and then pattern-based reuse (or, vice versa) effectively with minimal cost.

$$C_{compute(qj|qi)} = \min_p(C_{compute(p|qi)} + C_{compute(qj|p)})$$
where p has either only concept or only pattern changes from q_i and q_j, respectively. (8)

5.4 Specific-to-General with Pattern Changes

Given queries q_i and q_j ($q_i >_p q_j$) in a pattern hierarchy and the results of q_j, then q_i can be computed by reusing q_j results and unioning them with the delta results not captured by q_j. Our compute operation includes two key factors, namely, *result reuse* and *delta result computation*. Figure 8 depicts the pseudocode for the specific-to-general evaluation.

In general, assume q_i = SEQ(E_i, E_j, E_k) is refined by an extra event E_m into q_j = SEQ(E_i, E_m, E_j, E_k). q_j results are reused for q_i and SEQ(E_i, ! E_m, E_j, E_k) results are the delta results. The cost model is given in Equation 9. This specific-to-general computation for a pattern hierarchy would need to check the non-existence of a possibly long intermediate pattern for delta result computation when two queries differing by more than one event type. These overhead costs in some cases may not warrant the benefits of such partial reuse. When two queries differ by **negative** event types, the specific-to-general method is similar to above except that during delta result computation we need to compute some additional sequence results filtered in the specific query due to the existence of events of negative types.

Specific-to-general evaluation with only pattern changes (
q_i and q_j are queries in a pattern hierarchy with $q_i >_p q_j$; R_{qi} - the results of q_i)
01 R_{q_i} = ReuseSubpatternResult(q_i, q_j, R_{q_j})
02 R_{q_i} = $R_{q_i} \cup ComputeDeltaResults(q_i, q_j)$
ReuseSubpatternResult(q_i, q_j, R_{q_j})
01 **for** each result $r_k \in R_{q_j}$
02 **for** each component $e_i \in r_k$
if(e_i.type $\notin q_j \wedge e_i$.type $\in q_i$)
remove e_i from r_k;
ComputeDeltaResults(q_i, q_j)
01 **for** each positive event type E_i or SEQ(E_i ,..., E_k)$\in q_j$ but $\notin q_i$
02 construct results for q_i with events failed in q_j due to non-existence of E_i or SEQ(E_i, E_j, ..., E_k) events
03 **for** each negative event type $E_i \in q_j$ but $\notin q_i$
04 construct results for q_i with events failed in q_j due to existence of E_i events

Figure 8: Specific-to-General Evaluation in Pattern Hierarchy

$$\begin{aligned} C_{compute(qi|qj).sp} =& |R_{qj}| * C_{type} * q_j.length + |S_k| * |S_j| \\ & * Pt_{E_j,E_k} * P_{E_j,E_k} + |S_k| * |S_j| \\ & * Pt_{E_j,E_k} * P_{E_j,E_k} * |S_i| * P_{E_i,E_j} \\ & * P_{E_i,E_j} * (1 - P_{E_i,E_j} * P_{E_m,E_j} * \\ & P_{E_i,E_j} * P_{E_m,E_j}) \end{aligned} \quad (9)$$

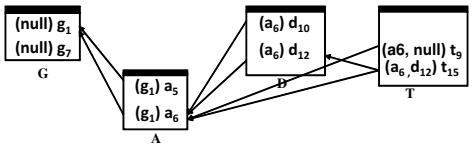


Figure 9: Stack Structure for q_3 and q_6 in Figure 1

EXAMPLE 8. *Figure 9 shows the hierarchical instance stacks for pattern queries q_3 and q_6 in Figure 1. Result reuse and delta result computation for q_3 are explained below.*
ReuseSubpatternResult. *q_3 is computed from the results of q_6 by subtracting subsequences composed of positive event types G, A and T. For example, in Figure 9, the result $< g_1, a_5, d_{10}, t_{15} >$ for q_6 is first generated using the stack-based join method. Then $< g_1, a_5, t_{15} >$ is prepared for q_3 by removing the event d_{10} of the event type D, because D is not listed in q_3. Lastly, we check whether this result is duplicated before returning it for q_3.*
ComputeDeltaResults. *Some sequences may not have been constructed for q_6 due to the non-existence of events of type D. However, such sequence results must now be constructed for q_3. In this case, each instance of type T has one pointer to an A event for q_3 and another pointer to a D event for q_6. Hence, for a T event that doesn't point to any D event, we can infer that a sequence involving this T event would not have been constructed for q_6. This T event thus should trigger its sequence construction for q_3 by a stack-based join. If one T event points to both an A and a D event, then the A and D events may still not satisfy the time constraints. If the timestamp of the A event is greater than the timestamp of the D event, sequence construction is triggered by such T event for q_3. In Figure 9, we observe that t_9 doesn't point to any D event. Hence sequence results $< g_1, a_5, t_9 >$ and $< g_1, a_6, t_9 >$ are*

constructed for t_9 by a stack-based join. The conditional cost to compute q_3 includes the costs of result reuse and the cost to compute $SEQ(G, A, !D, T)$ results.

5.5 Specific-to-General with Concept Changes

The result set of a higher concept abstraction level is a super set of all the results of pattern queries below it. Thus upper level query can be computed in part by reusing the lower level query results. The lower level pattern query is computed first. Then all these results are also returned for the upper level pattern. In addition, the events of the higher event type concept level not captured by the lower queries must also be constructed. Such specific-to-general computation requires no extra interpretation costs as compared to the general-to-specific evaluation. Given two pattern queries q_i and q_j with only concept changes ($q_i >_c q_j$), our cost model is formulated by Equation 10.

$$C_{compute(qi|qj).sc} = C_{compute(qi)} - C_{compute(qj)} \quad (10)$$

EXAMPLE 9. *Figure 10 shows the hierarchical instance stacks for q_1 to q_2 in Figure 1. From q_1 to q_2 only concept relationships are refined. Results for q_2 { dh_{10}, ts_{33}}, {dh_{16}, ts_{33}} are computed first. And these results are also returned for q_1. Next, we need to compute the delta results belonging to q_1 that were not captured by q_2. In Figure 10, the pointers between D and T are already traversed during the evaluation of q_2. The other pointers between D and OK, TX and OK, TX and T need now to be traversed. Results $\{ah_{12}, oh_{15}\}$, $\{ah_{10}, oh_{15}\}$, $\{ah_{12}, oh_{38}\}$, $\{as_{18}, os_{38}\}$, $\{dh_{10}, os_{38}\}$, $\{dh_{18}, os_{38}\}$, $\{ah_{12}, ts_{33}\}$, $\{as_{18}, ts_{33}\}$ are constructed for q_1.*

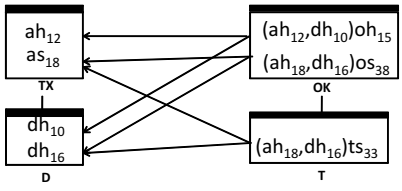


Figure 10: Stack Structure for q_1 and q_2 in Figure 1

5.6 Specific-to-General with Concept & Pattern Refinement

Given q_i and q_j in an E-Cube hierarchy with simultaneous concept and pattern changes ($q_i >_{cp} q_j$), we first find one intermediate query p with either only concept or pattern changes from q_j so that query p minimizes Equation 11. As above, we then compute results in two stages from q_j to p and from p to q_i by using specific-to-general evaluation with first only pattern and then only concept changes or vice versa effectively with minimal cost.

$$C_{compute(qi|qj)} = \min_p(C_{compute(p|qj)} + C_{compute(qi|p)}) \quad \text{where } p \text{ has either only concept or only pattern changes from } q_i \text{ and } q_j, \text{ respectively.} \quad (11)$$

6. PLAN ADAPTATION

High variability in input stream rates and selectivities may render an initially optimal execution ordering not optimal or possibly even ineffective after some time. A query could be added to or removed from the system as well. To recompute the query execution order on the fly, we maintain a running estimate of the statistics. When the statistics vary by more than some error threshold θ, we re-run the Chase optimizer in a separate system thread to generate a new ordering recommendation. If the performance improvement predicted by the cost model is greater than a given performance threshold γ, we then install the new updated plan.

To change the execution ordering on the fly, we would need to simply switch from utilizing one result buffer to another buffer space for conditional computation. The process for changing the query execution ordering on-line thus uses the following steps:
1. Discard intermediate results based on the execution ordering after finishing the result computation for the current input event e_i;
2. Rebuild intermediate results based on the newly determined execution ordering as if it were the first round before starting to process the next instance e_{i+1} from input stream. No results are output during this preparation stage.

The advantage of our adaptation method is its simplicity. More sophisticated adaptive strategies that may incrementally reuse some of the intermediate results to minimize the recalculation effect [30] could be designed. However, the complexity of such a method may offset its potential gains. We thus leave this analysis as future work.

7. PERFORMANCE EVALUATION

The primary objective of our experimental study is to compare four different strategies, namely, *state-of-the-art*, *a pure top-down*, *a pure bottom-up* and *our optimized Chase* strategies for E-Cube evaluation, and to determine their respective scope of applicability. As explained in Section 2, the state-of-the-art method processes queries independently using stack-based query evaluation [29]. The *top-down* (*bottom-up*) method proceeds by evaluating general (specific) patterns first and then iteratively processing patterns lower (higher) in the E-Cube hierarchy (Section 5). Finally, the Chase method applies the Chase optimizer to construct and then utilize the optimal cost-based reuse strategy (Section 4).

7.1 Experimental Setup

We implement our proposed E-Cube framework [21] inside the Chaos stream management system [12] using Java. We ran the experiments on Intel Pentium IV CPU 2.8GHz with 1GB RAM. We evaluated our techniques using real stock trades data from [1]. The data contained stock ticker, timestamp and price information. We used sliding window of size 1 second in the experiments. A concept hierarchy for stock companies is built as in Figure 11. The performance metric *result latency* is the accumulative time difference between the sequence output time and the arrival time of the latest event instance composed into the sequence result. We compared the result latency of various strategies using different pattern query sets. Specifically, the financial sector is very sensitive to query result latency and uses extensive CPU resources to achieve this goal. We start with controlled query sets where we control one single parameter (pattern or concept) and later also conduct larger typical workloads mixed the two types of workloads to demonstrate a more realistic concurrent CEP query processing scenario. The results are extremely encouraging showing benefits of using our adaptive Chase strategy over all other methods.

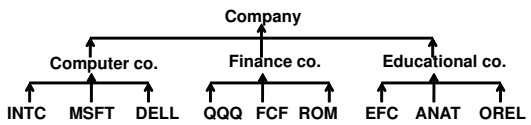
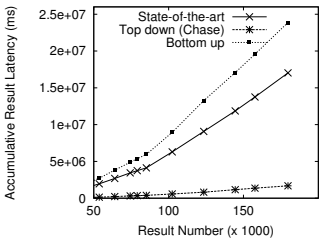


Figure 11: Company Concept Hierarchy

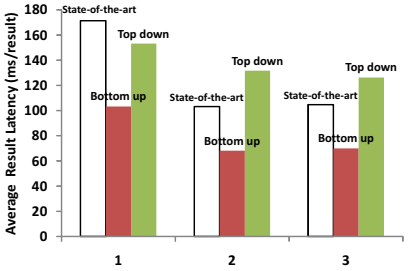
We first tested the cost models (Equations 4-11) to verify that

Length	State-of-the-art	Top down (Chase)	Bottom up	Speedup
3	0.835	0.11	1.29	7.59
4	96.415	9.71	136.39	9.93
5	5252.5	188.16	11593	27.92

(a) Workload with only Pattern Changes: Average Result Latency (ms/result)

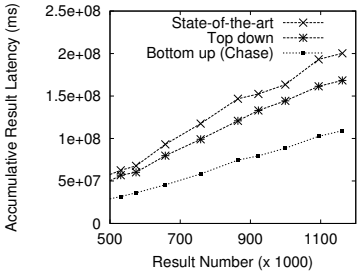


(b) Workload with only Pattern Changes

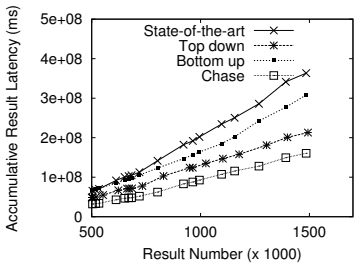


(c) Workload with only Concept Changes

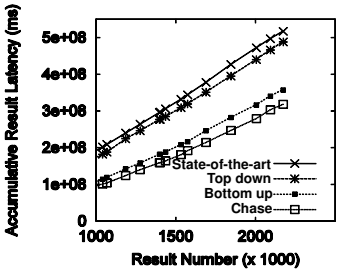
Figure 12: Controlled Workloads.



(a) Workload with only Concept Changes



(b) Workload 7



(c) Workload 8

Figure 13: (a) Controlled Workload; (b)(c) Complex Query Workloads with Both Refinement Relationships.

they accurately reflect the system performance. We ran these experiments on all the pattern query workloads given below. We found that the estimates produced by our cost model for the four methods correctly reflected the actual system behavior of the four alternative methods (state-of-the-art, top-down, bottom-up and Chase). We now omit these verification details for brevity and skip directly to the experimental results for strategy comparisons.

7.2 Scenarios with Pattern Hierarchy Queries

In this first experiment, we compare the four methods (state-of-the-art, top-down, bottom-up and Chase) evaluating queries forming a pure pattern hierarchy (i.e., no concept changes). The root query size is increased from 3 to 5 in the workloads 1, 2 and 3. Figure 12(a) shows the average result latency (ms) of the four methods and speedup of the top-down (chase) method over the state-of-the-art method. Figure 12(b) shows the accumulative result latency for workload 2 (while workloads 1 and 3 with similar trends are omitted for space reasons). We observe that the top-down method generates results faster than the state-of-the-art and the bottom-up methods. It outperforms the others because it avoids result recomputation by applying conditional computation. We also notice that the average latency difference between the state-of-the-art method and the top down increases as the result sharing length increases from 3 to 5 due to reuse and computational savings. The speed-up factor of the method chosen by Chase over the state-of-the-art method starts at x8 at length 3, increasing to x10 and x28 for lengths 4 and 5, respectively. The bottom up method generates results slower than the other methods, because it introduces an extra delta result computation cost (see Section 5.4 for explanation).

```
Workload 1 (shared length 3):
  q1 = SEQ(INTC, ! MSFT, FCF)
  q2 = SEQ(INTC, ! MSFT, FCF, ROM)
  q3 = SEQ(INTC, ! MSFT, FCF, EFC)
  q4 = SEQ(INTC, ! MSFT, FCF, ANAT)
  q5 = SEQ(INTC, ! MSFT, FCF, OREL)

Workload 2 (shared length 4):
  q6 = SEQ(DELL, INTC, ! MSFT, FCF)
  q7 = SEQ(DELL, INTC, ! MSFT, FCF, ROM)
  q8 = SEQ(DELL, INTC, ! MSFT, FCF, OREL)
  q9 = SEQ(DELL, INTC, ! MSFT, FCF, ANAT)
  q10= SEQ(DELL, INTC, ! MSFT, FCF, QQQ)

Workload 3(shared length 5):
  q11 = SEQ(QQQ, DELL, INTC, ! MSFT, FCF)
  q12 = SEQ(QQQ, DELL, INTC, ! MSFT, FCF, ROM)
  q13 = SEQ(QQQ, DELL, INTC, ! MSFT, FCF, ANAT)
  q14 = SEQ(QQQ, DELL, INTC, ! MSFT, FCF, OREL)
  q15 = SEQ(QQQ, DELL, INTC, ! MSFT, FCF, EFC)
```

7.3 Scenarios with Concept Hierarchy Queries

Next, we compare methods for evaluating query workloads with only concept changes. We ran experiments on workloads 4, 5 and 6 below. Figure 12(c) shows the average result latency of the three methods for each workload and Figure 13(a) shows the accumulative result latency for workload 4. We observe that the bottom up method now produces results faster than the other methods. This is because results from q_{17}, q_{18} and q_{19} are reused for q_{16}. The top down method is better than the state-of-the-art method in workload 4 because a large percentage of q_{16} results match the child query q_{17} (only one concept change). The top down method does even worse than the state-of-the-art method in workloads 5 and 6. This is because in the top down method, we need to check the types of component events for each result of q_{16}. When only a small percentage of q_{16} results match children queries q_{18} and q_{19}, direct result computation (state-of-the-art method) is better than result interpretation (top down method) in the concept hierarchy.

```
Workload 4:
  q16 = SEQ(Computer, Finance, Education)
  q17 = SEQ(Computer, Finance, EFC)

Workload 5:
  q16 = SEQ(Computer, Finance, Education)
  q18 = SEQ(Computer, QQQ, EFC)
```

```
Workload 6:
  q16 = SEQ(Computer, Finance, Education)
  q19 = SEQ(INTC, QQQ, EFC)
```

7.4 Scenarios with Representative Mixed Workloads

We compare the four methods with workloads involving both concept and pattern changes. This Chase optimizer took 16 ms to find the optimal execution ordering. We designed workloads 7 and 8 to be representative and interesting mixes of changes. DELL stock belongs to Computer and QQQ, FCF, ROM stocks belong to Finance. EFC, ANAT and OREL stocks belong to Education. Figures 13(b) and 13(c) show the accumulative result latency of the four methods, respectively. As expected, Chase produces results faster than the others. On closer analysis in Chase for workload 7, q_{20} is executed first and its results are reused for q_{27} using the bottom up method and for q_{21}, q_{22}, q_{23} and q_{24} by the general-to-specific evaluation. Results of q_{24} are reused for q_{25} using the general-to-specific evaluation. Results of q_{27} are reused for q_{26} and q_{16} by the specific-to-general evaluation and for q_{28} by the general-to-specific evaluation. Workload 8 is similar to workload 7. In other words, Chase carefully selects the optimal combination of execution and reuse strategies.

```
Workload 7:
  q20 = SEQ(DELL, QQQ, ANAT)
  q21 = SEQ(DELL, QQQ, ANAT, ROM)
  q22 = SEQ(FCF, DELL, QQQ, ANAT)
  q23 = SEQ(DELL, QQQ, ANAT, OREL)
  q24 = SEQ(DELL, QQQ, ANAT, INTC)
  q25 = SEQ(DELL, QQQ, ANAT, INTC, EFC)
  q16 = SEQ(Computer, Finance, Education)
  q26 = SEQ(Computer, Finance, ANAT)
  q27 = SEQ(DELL, Finance, ANAT)
  q28 = SEQ(QQQ, DELL, Finance, ANAT)


Workload 8:
  q16 = SEQ(Computer, Finance, Education)
  q29 = SEQ(Computer, Finance, OREL)
  q30 = SEQ(INTC, QQQ, Education)
  q31 = SEQ(INTC, QQQ, EFC)
  q32 = SEQ(MSFT, INTC, QQQ, EFC)
  q33 = SEQ(INTC, QQQ, EFC, DELL)
  q34 = SEQ(Computer, ROM, Education)
  q35 = SEQ(Computer, ROM, ANAT)
  q36 = SEQ(INTC, ROM, ANAT)
```

Accumulative CPU processing time is the total pattern query execution time without any I/O waiting time. In a complementary set of experiments we measure the CPU-only execution time as shown in Figures 14-15. These experiments were conducted using the same workloads 1-8. This finding shows that the strategies are mostly CPU-bound and not I/O bound. Other findings include (1) The top down method runs on average 10 fold faster than the state-of-the-art and the bottom up methods for queries with only pattern changes as depicted in Figures 14(a), 14(b). (2) The bottom up method runs on average 2 times faster than the state-of-the-art and the top down methods for queries with only concept changes as in Figure 14(c), 15(a). (3) For a mixed workload, the Chase method constantly outperforms the other methods as shown in Figures 15(b), 15(c).

8. RELATED WORK

Traditional OLAP focuses on static pre-computed and indexed data sets and aims to quickly provide answers to analytical queries that are multi-dimensional in nature [4, 15, 11]. OLAP techniques allow users to navigate the data at different abstraction levels. However, the state-of-the-art OLAP technology tends to be set-based instead of sequence based [11]. Further, aggregation (count, sum, max, ave) is conducted over scalar values, namely, the set of values within a single column such as salary, and not over ordered sequences. Hence, in the context of event patterns where the order of events is important, OLAP is insufficient in supporting efficient multi-dimensional event sequence analysis.

The state-of-art OLAP solutions [24, 10, 14] either don't support real-time streams at all, or they do not tackle CEP sequence queries. The work that is most closely related to ours is Sequence OLAP [24] which proposed to support OLAP operations for sequences. However, sequence OLAP does not support the notion of concept refinement for pattern queries as done in our work. Second, sequence OLAP preprocesses all data off-line, and then inserts the data into inverted indices. Thereafter, the results are joined using the inverted indices. In short, Sequence OLAP neither supports incremental maintenance of its precomputed index, nor streaming, nor negation in sequence - while these are all contributions of our work. Such (static) techniques used in Sequence OLAP are inappropriate in a stream setting.

A second related work is Flow Cube [10] which constructs a data warehouse of RFID-tagged commodity flow. The commodity flowgraph captures the major movement trends and significant deviations of the items over time. It can be viewed at multiple levels by changing the level of abstraction of path stages. However, it neither support streaming data nor concept hierarchies. Furthermore, it does not consider any optimization algorithms for hierarchical pattern query evaluation such as sequence reuse nor the cost-driven Chase method which is our core contribution. This line of work also does not consider event negation, which is covered in our system. Lastly, Stream Cube [14] has recently been proposed to facilitate online multi-dimensional analysis of stream data. However, it provides neither result reuse strategies nor any cost analysis for pattern queries including neither sequence nor negation.

Complex Event Processing (CEP) systems demonstrate sophisticated capabilities for pattern matching [3, 5, 29]. Yet, they do not support OLAP-like operations for multi-dimensional event sequence analysis at different abstraction levels. We borrow a variety of techniques from CEP, including stack-based joins [29] and cost models for stack-based joins [25]. However, work in CEP has not studied hierarchical pattern refinement relations, such as concept hierarchies as proposed in our work. CEP systems such as Cayuga [5, 16], SASE [29] and ZStream [25] focus on event sequence detection over streams. However, these systems do not address the issue of supporting queries at different concept and pattern hierarchies nor do they design efficient computation strategies for processing multiple such queries. Recently, work in CEP has considered pushing negation into sequence processing [25]. We exploit this as part of our proposed solution for determining if additional delta results must be generated in the specific-to-general reuse.

Multiple-query optimization (MQO) in databases [27, 26, 7], typically focussed on static relational databases. MQO identifies common subexpressions among queries such as common joins or filters. Multiple-query optimization (MQO) for stack-based pattern evaluation for CEP queries has not yet been studied, in particular, sharing for CEP queries with negation and concept refinements was an open problem prior to our work.

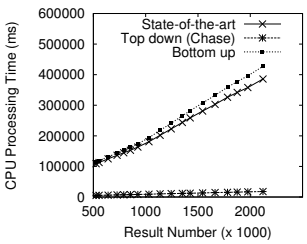
Lastly, [2] proposes sharing among XML queries, in particular, prefix sharing and suffix clustering. However, they neither consider concept nor pattern hierarchies.

9. CONCLUSION

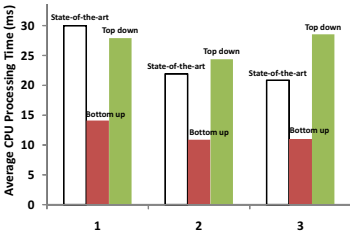
E-Cube combines OLAP and CEP functionalities, i.e., technologies that allow users to efficiently query large amounts of event stream data in multiple dimensions and at multiple abstraction lev-

Length	State-of-the-art	Top down (Chase)	Bottom up
3	0.28	0.04	0.31
4	7.77	1.2	7.87
5	384.8	17.75	426.57

(a) Workloads 1-3 with only Pattern Changes: Average CPU Processing Time (ms/result)

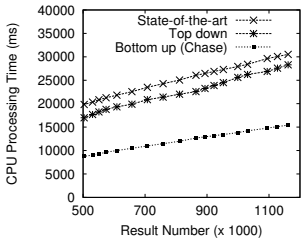


(b) Workload 2 with only Pattern Changes

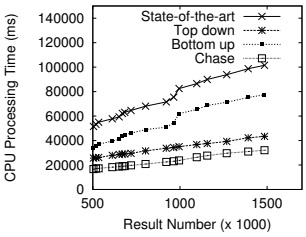


(c) Workloads 4-6 with only Concept Changes

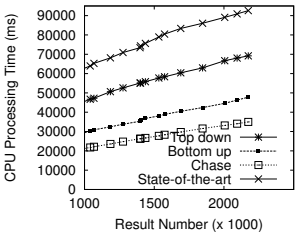
Figure 14: Controlled Workloads.



(a) Workload 4 with only Concept Changes



(b) Workload 7



(c) Workload 8

Figure 15: (a) Controlled Workload; (b)(c) Complex Query Workloads with Both Refinement Relationships.

els. To the best of our knowledge, no prior work combines CEP and OLAP techniques for multi-dimensional pattern analysis over event streams as described in this paper. Our E-Cube solution improves computational efficiency for multi-dimensional event pattern detection by sharing results among queries in a unified query plan. Based on this foundation, we design a cost-driven adaptive optimizer called Chase which delivers optimal results. In the Chase method, our E-Cube optimization problem is mapped into a well-known graph problem. Our Chase method in many cases performs ten fold faster than the state-of-art strategy. Interesting future work includes supporting additional query features like recursion and closure as well as deployment on the cloud.

Acknowledgements. This work is supported by HP Labs Innovation Research Program and National Science Foundation under grants NSF 1018443 and NSF IIS 0917017, Turkish National Science Foundation TUBITAK under career award 109E194

10. REFERENCES

[1] I. inetats. stock trade traces. http://www.inetats.com/.

[2] K. S. Candan, W.-P. Hsiung, S. Chen, J. Tatemura, and D. Agrawal. AFilter: Adaptable XML filtering with prefix-caching and suffix-clustering. In *VLDB*, pages 559–570, 2006.

[3] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *VLDB*, pages 606–617, 1994.

[4] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1):65–74, 1997.

[5] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White. Cayuga: A general purpose event monitoring system. In *CIDR*, pages 412–422, 2007.

[6] J. Edmonds. Optimum branchings. In *J. Research of the National Bureau of Standards*, pages 233–240., 1967.

[7] S. Finkelstein. Common expression analysis in database applications. In *SIGMOD*, 1982.

[8] H. N. Gabow, Z. Galil, T. H. Spencer, and R. E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986.

[9] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu. Adaptive load shedding for windowed stream joins. In *CIKM*, pages 171–178, 2005.

[10] H. Gonzalez, J. Han, and X. Li. Flowcube: Constructuing RFID FlowCubes for multi-dimensional analysis of commodity flows. In *VLDB*, pages 834–845, 2006.

[11] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *VLDB*, pages 358–369, 1995.

[12] C. Gupta, S. Wang, I. Ari, M. C. Hao, U. Dayal, A. Mehta, M. Marwah, and R. K. Sharma. Chaos: A data stream analysis architecture for enterprise applications. In *CEC*, pages 33–40, 2009.

[13] J. Han, Y. Cai, and N. Cercone. Knowledge discovery in databases: An attribute-oriented approach. In *VLDB*, pages 547–559, 1992.

[14] J. Han, Y. Chen, G. Dong, J. Pei, B. W. Wah, J. Wang, and Y. D. Cai. Stream Cube: An architecture for multi-dimensional analysis of data streams. *Distributed and Parallel Databases*, 18(2):173–197, 2005.

[15] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD*, pages 205–216, 1996.

[16] M. Hong, M. Riedewald, C. Koch, J. Gehrke, and A. J. Demers. Rule-based multi-query optimization. In *EDBT*, pages 120–131, 2009.

[17] S. Krishnamurthy, C. Wu, and M. J. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, pages 623–634, 2006.

[18] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD*, pages 311–322, 2005.

[19] B. Liu, Y. Zhu, and E. A. Rundensteiner. Run-time operator state spilling for memory intensive long-running queries. In *SIGMOD*, pages 347–358, 2006.

[20] M. Liu, M. Li, D. Golovnya, E. A. Rundensteiner, and K. T. Claypool. Sequence pattern query processing over out-of-order event streams. In *ICDE*, pages 784–795, 2009.

[21] M. Liu, E. Rundensteiner, K. Greenfield, C. Gupta, S. Wang, I. Ari, and A. Mehta. E-cube: Multi-dimensional event sequence processing using concept and pattern hierarchies. In *ICDE*, pages 1097–1100, 2010.

[22] M. Liu, E. Rundensteiner, K. Greenfield, C. Gupta, S. Wang, I. Ari, and A. Mehta. High-performance nested cep query processing over event streams. In *ICDE*, 2011.

[23] M. Liu, E. Rundensteiner, K. Greenfield, C. Gupta, S. Wang, I. Ari, and A. Mehta. E-cube: Multi-dimensional event sequence processing using concept and pattern hierarchies. Technical Report WPI-CS-TR-09-08.

[24] E. Lo, B. Kao, W.-S. Ho, S. D. Lee, C. K. Chui, and D. W. Cheung. OLAP on sequence data. In *SIGMOD Conference*, pages 649–660, 2008.

[25] Y. Mei and S. Madden. Zstream: a cost-based query processor for adaptively detecting composite events. In *SIGMOD*, pages 193–206, 2009.

[26] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, pages 249–260, 2000.

[27] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[28] S. Wang, E. A. Rundensteiner, S. Ganguly, and S. Bhatnagar. State-slice: New paradigm of multi-query optimization of window-based stream queries. In *VLDB*, pages 619–630, 2006.

[29] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD*, pages 407–418, 2006.

[30] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman. Dynamic plan migration for continuous queries over data streams. In *SIGMOD*, pages 431–442, 2004.