Multi-Query Optimization for Complex Event Processing in SAP ESP

Shuhao Zhang[1,2], Hoang Tam Vo[3*], Daniel Dahlmeier[1], Bingsheng He[2]
[1]SAP Innovation Center Singapore, [2]National University of Singapore, [3]IBM Research, Australia

***Abstract*—SAP Event Stream Processor (ESP) platform aims at delivering real-time stream processing and analytics in many time-critical areas such as Capital Markets, Internet of Things (IoT) and Data Center Intelligence. SAP ESP allows users to realize complex event processing (CEP) in the form of pattern queries. In this paper, we present MOTTO – a multi-query optimizer in SAP ESP in order to improve the performance of many concurrent pattern queries. This is motivated by the observations that many real-world applications usually have concurrent pattern queries working on the same data streams, leading to tremendous sharing opportunities among queries. In MOTTO, we leverage three major sharing techniques, namely merge, decomposition and operator transformation sharing, to reduce redundant computation among pattern queries. In addition, MOTTO supports nested pattern queries as well as pattern queries with different window sizes. The experiments demonstrate the efficiency of the MOTTO with real-world application scenarios and sensitivity studies.**

I. Introduction

Complex event processing (CEP) has been successfully applied in many areas such as Capital Markets [1], Internet of Things (IoT) [2] and Data Center Intelligence [3]. Those domains are usually "big data" applications with high velocity. SAP ESP aims at delivering real-time stream processing and analytics in time-critical applications. In SAP ESP, users can implement their complex event processing tasks in Continuous Computation Language (CCL).

Figure 1 illustrates an application scenario of stock market analysis in SAP ESP. Financial analysts may define their interested events generated from market data such as $< buy\_order, stockId >$ event (i.e., a significant buy order of stock with id of $stockId$ posted in stock market), and $< sell\_order, stockId >$ event (i.e., a significant sell order of stock with id of $stockId$). The analyst may also define events generated from other tools such as a report of uptrend of stock with id of $stockId$ as $< uptrend, stockId >$ and a report of relative strength index (RSI) of stock with id of $stockId$ is currently below 30 as $< RSI_{low}, stockId >$. The analyst can then express CEP pattern queries based on those events in CCL. For instance, Q1 monitors the event that, within 10 minutes, $< sell\_order, MSFT >$ event happens followed by $< buy\_order, AAPL >$ and $< buy\_order, IBM >$ and $< RSI_{low}, IBM >$ is received. Those queries (illustrated in the middle of Figure 1) continuously monitor input data stream (illustrated on the left side of Figure 1) and report output (illustrated on the right side of Figure 1), once the predefined event patterns are detected. We formally define the pattern query in Section II.

Many pattern queries can be registered to the system on the same data streams. Similarities among pattern queries create opportunities for sharing optimization. For instance, in Figure 1, Q1, Q2, and Q3 are all interested in the event of $< buy\_order, IBM >$, and Q1 and Q2 share a common interest in the event of $< RSI_{low}, IBM >$. Evaluating each pattern query individually results in redundant computing efforts. Thus, a multi-query optimizer is needed to determine the sharing opportunities and to realize the sharing for efficiency.

We have faced the following challenges in building a multi-query optimizer for SAP ESP.

1) There can be a large number of pattern queries registered in SAP ESP. To identify an optimal execution plan involves the challenge of solving a complex optimization problem.
2) Pattern queries have different configurations and structures, although they may have sharing opportunities. First, they may have different window constraints. This poses challenges on correctly sharing among queries even those with an identical pattern of interest. Second, pattern queries may be expressed with nested patterns. Specifically, a pattern query may involve a complex event that is the monitoring results of another pattern query. This further increases the space of identifying sharing.

In this paper, we present MOTTO – a multi-query optimizer for pattern query processing in SAP ESP. With the flavor of multi-query optimizations in relational databases, MOTTO is specially designed for complex event processing in SAP ESP. To achieve more substantial sharing opportunities, MOTTO has three sharing techniques that are applied together to eliminate redundant computation among pattern queries. The first technique, called merge sharing technique (MST), serves as the basic sharing technique allows the results of one query to be shared by another to reduce computational cost. The second technique, called decomposition sharing technique (DST), allows sharing computation after proper decomposition of pattern queries. More precisely, we can efficiently decompose the original pattern query into multiple sub-queries, so that sharing opportunities based on these sub-queries can be enabled in a fine-grained manner. The third technique, called operator transformation technique (OTT), provides a set of rules for transforming one pattern operator to another, so that we can discover sharing opportunities for

*This work was done while this author was working at SAP.

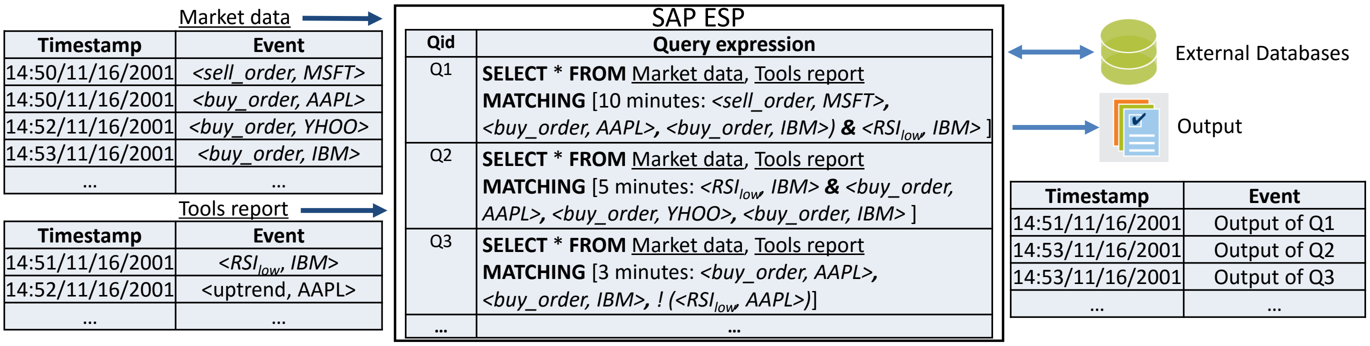


Fig. 1: Stock market monitoring application scenario.

pattern queries even with different types of operators. We further extend these sharing techniques (i.e., merge sharing, decomposition and transformation) to support nested pattern queries and queries with different window sizes. To find the optimal plan, we map the multi-query optimization problem into the Directed Steiner Minimum Tree (DSMT) problem [4] and adopt existing DSMT solvers [4, 5, 6, 7] for the efficient solution of this problem.

We conduct experimental studies with both real application scenarios and sensitivity studies. The experimental results confirm the efficiency of MOTTO in improving the efficiency of complex event processing in SAP ESP.

The rest of the paper is organized as follows. Section II presents preliminary and background. Section III gives an overview of MOTTO. The sharing techniques are discussed in detail in Section IV. Section V presents the optimization problem and our solution. We describe the cost model for pattern query and sharing techniques in Section VI. We present the experimental results in Section VII, followed by related work in Section VIII. We conclude the paper in Section IX.

II. Preliminary

In this section, we briefly introduce the event processing language named Continuous Computation Language (CCL) in SAP ESP. CCL is based on SQL and adapted for stream processing. CCL supports pattern matching on real-time data streams. The basic structure of a pattern query is specified as follows.

SELECT ***OperandList***
FROM ***Streams***
MATCHING : [***windowConstraint*** : ***PatternList***]

We introduce the following terminologies related to this query structure. Data **streams** consist of *event instances*. We use lower-case letters (e.g., e) to denote event instances, which can be either *primitive events* or *composite events*. Primitive events are predefined single-occurrence events according to user interests, which cannot be further divided. Each primitive event has a timestamp (ts) that denotes the occurrence of the event (e.g., timestamp of a trade event). Composite events are a collection of primitive events detected by pattern queries, such as a composite event (denoted as $\{e_1,e_2\}$) consisting of an event e_1 followed by an event e_2, which can be further used as input to pattern queries. The time-stamp of a composite event can be a time range depends on the corresponding pattern query. We force a *complete-history* temporal model [8] when a composite event is further used as input by adding additional time filter whenever necessary. An *event type* denotes the unique feature associated with event instances, which is used in pattern query to specify the desired event pattern. We use uppercase letters to denote event type (e.g., E_i denotes the type of event instance e_i). The event types specified in the pattern query is called *operand* and forming the **OperandList**. **PatternList** connects operands together by *pattern operators* to form a pattern of event types to be monitored. **WindowConstraint** requires that the events composing the monitoring event pattern of the query occur within the specified time interval.

For the ease of presentation on sharing opportunities, we focus on the techniques for pattern queries on the same stream (i.e., no sharing otherwise). Also, we assume they have the same window sizes, and extend the support for different window sizes and nested pattern query in Section IV-D. In the following, we denote query in the form of *"Pattern operator (OperandList)"* for compactness.

SAP ESP supports the following pattern operators:

- **Conjunction** (CONJ or &) requires the occurrence of all operands, regardless of their arrival order. For example, CONJ(E_1&E_2) produces a composite event of type $\{E_1\&E_2\}$ if both events of type E_1 and E_2 happen within window constraint regardless of their occurrence sequence.
- **Disjunction** (DISJ or |) requires, at least, one occurrence of operands in order to generate output. For example, DISJ ($E_1|E_2$) produces an event of type (a) $\{E_1\&E_2\}$ if both events of type E_1 and E_2 occurred, *or* (b) $\{E_1$ $(E_2)\}$ if the only event of type E_1 (E_2) occurred. To simplify notation, we use $\{E_1|E_2\}$ to denote both cases.
- **Sequence** (SEQ or ,) requires the ordered occurrence of all operands linked by it. For instance, SEQ(E_1, E_2) generates a composite event of type $\{E_1, E_2\}$ if an event of type E_1 and E_2 occurred sequentially, not necessary continuously (i.e., other events may happen in between).
- **Negation** (NEG or !) is an unary operator and must be used with SEQ or CONJ. For instance, SEQ(E_1, E_3,

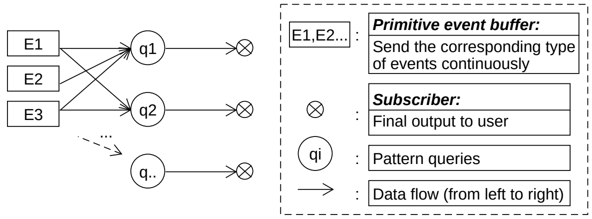


Fig. 2: Jumbo Query Plan.

NEG(E_2)) requires an event of type E_1, and E_3 occur sequentially, but *no* event of E_2 happen (regardless of the arrival order of E_2) within the specified window constraint. In SAP ESP, pattern matching on NEG is evaluated to be successful only after the expiration of the specified time interval. When NEG is used with SEQ, changing the ordering or grouping of it does not change semantic. We assume NEG always be the last component of the query, which is because events succeeding the NEG will never be evaluated by the pattern match engine owing to the expiration of the time interval. For example, SEQ(E_1,NEG(E_2),E_3) is equivalent to SEQ(E_1,E_3,NEG(E_2)).

We discuss two key properties (commutativity and associativity) for each operator, which are important for the sharing techniques. 1) Commutativity: CONJ and DISJ are commutative, but SEQ is not [9]. 2) Associativity: CONJ and DISJ are associative since they do not require order of the operands [9]. However, the associativity for SEQ does not naturally hold but depends on the temporal model [8]. Here, we need to use an additional time constraint to preserve associative property for SEQ operator. For instance, we preserve semantic equivalence between SEQ((E_1, E_2), E_3) (left-associative plan) and SEQ(E_1, (E_2, E_3)) (right-associative plan), by adding additional time filter operation E_2.ts $<$ E_3.ts and E_1.ts $<$ E_2.ts after the pattern queries, respectively. This effectively forces complete-history temporary model and is being used when the composite event is used as input event as mentioned before.

III. MOTTO Workflow Overview

In this section, we present the workflow overview for MOTTO, a multi-query optimizer in SAP ESP. The multi-query optimization is motivated by many applications consisting of queries with a lot of sharing opportunities.

We use **jumbo query plan** (JQP) to denote the query execution plans involving a set of pattern queries. We illustrate the notation of JQP in Figure 2. This JQP represents the execution plan that all queries are directly connected to the data source, and no sharing optimization are applied.

There are two main modules in MOTTO:

(1) Query Rewriter. This module applies sharing techniques to a given workload and produces different execution plans. In this paper, we present three sharing techniques: the basic merge sharing, and two fine-grained sharing techniques namely decomposition sharing and operator transformation.

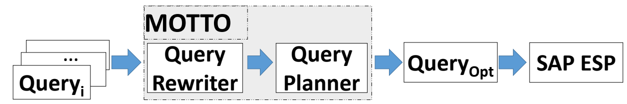


Fig. 3: Multi-query optimization workflow of MOTTO.

(2) Query Planner. This module examines all possible execution plans produced by query rewriter and selects the most efficient one. It employs a search strategy based on branch and bound. A cost model is used to estimate the cost of each execution plan with or without applying sharing techniques.

Given a workload consisting of multiple pattern queries, MOTTO identifies the sharing opportunities among pattern queries and finds the most efficient JQP for those queries. The optimal JQP is submitted to SAP ESP for execution. The workflow of MOTTO is shown in Figure 3, where the input is a workload containing multiple pattern queries.

IV. Query Rewriter

In this section, we introduce the detailed design for sharing techniques in MOTTO. We start with a basic sharing technique named *merge sharing*, and then two fine-grained techniques named *decomposition sharing* and *operator transformation*. For each technique, we present its definition, followed by how to apply the techniques on queries and discuss the correctness regarding semantic equivalence. We also give some simple examples to illustrate our techniques. At the end of this section, we extend the sharing techniques to nested pattern queries and queries with different window constraints.

We first define sharing dependency between two queries as follows.

DEFINITION 1. *A* ***source query*** *is a query whose generated composite events are shared by its* ***beneficiary query****. Each source query can have multiple beneficiaries and vice versa.*

A. Merge Sharing Technique

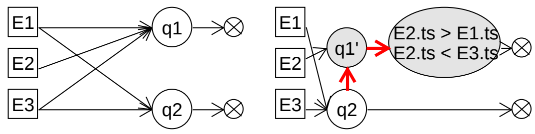
The main idea of *Merge Sharing Technique (*__MST__*)* is that we can conceptually merge the results of one query into another. It applies to queries with the same pattern operator and same window constraint. This basic sharing strategy is inspired by the previous study [10].

Given two pattern queries, q = OP(L), q' = OP(L'), where OP stands for SEQ/CONJ/DISJ, we consider the following two cases for merge sharing. In both cases, q is the source query, and q' is the beneficiary query.

(a) Substring case: L is a substring of L'. Then, L and L' can share their common pre/in/suffix, which is straightforward.

(b) Non-substring case: L is subsequence, but not the substring of L'.

The detailed procedure of the merge operation depends on the type of pattern query: (1) if OP is SEQ, then merge operation stands for linking operands of the type of generated composite event from q and the rest type of events of q' by a CONJ operator. An extra filter operation may be required to enforce the correct sequence; (2) if OP is CONJ/DISJ, then



(a) Original JQP of q1 and q2 (b) Optimized JQP of q1 and q2

Fig. 4: Example of applying merge sharing technique.

merge operation stands for linking operands of the type of generated composite event from q and the rest type of events of q' by the same operator. Essentially, the original q' is replaced by the *merge* operation, which takes results from another query as its input.

Example 1. *Figure 4 illustrates one example of applying MST upon* q_1 = *SEQ(*E_1, E_2, E_3*) and* q_2 = *SEQ(*E_1, E_3*). Note that,* q'_1 = *CONJ(*$\{E_1,E_3\}$&E_2*), which is essentially the merge operation. Although an extra filter* $E_1.ts < E_2.ts < E_3.ts$ *is needed to enforce the correct time sequence of event of type* E_2*, the pattern detection process of events of type* E_1 *and* E_3 *is shared, which reduces the total computing cost.*

Application of MST. Given two pattern queries with operand list L and L', the searching for sharing opportunities based on MST essentially involves checking whether L (L') is substring or subsequence of L' (L). This can be simply done by traversing through one of the lists, which requires $O(n)$ where n is the length of the shorter one of L and L'.

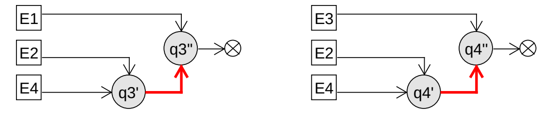
Semantic equivalence. After applying MST, there is a possibility of changing the grouping of operands in the *substring case*, and a possibility of changing in operand order in the *non-substring case*. We briefly show that MST has semantic equivalence in both cases. For CONJ/DISJ, the semantic equivalence in both cases is guaranteed by their associative and commutative property. In other words, merging the results with the rest in any order should produce the same results. For SEQ, the correctness in *substring case* is given by its associative property. In *non-substring* case, after we merge the results using a CONJ operator, an extra time filter to enforce the correct sequence as shown in Example 1.

B. Decomposition Sharing Technique

One of the restrictions of MST is that a query can only share its results entirely with another or none. We propose *decomposition sharing technique (DST)* in MOTTO to enable more sharing among queries.

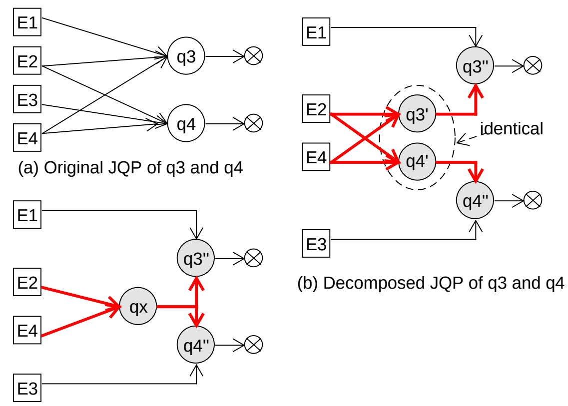
Pattern query decomposition. A pattern query can be decomposed into multiple *sub-queries* without changing its semantic meaning [9, 11]. We call a query plan is left (resp. right) decomposed plan if we sequentially break its original query plan in such a way that we always break the prefix (resp. suffix) two operands into one sub-query, which then connect with the rest. By mixing left and right decomposition, we can get a decomposed plan with decomposition at arbitrary places.

We denote decomposed query plan with the form of "sub-query" $\rightarrow$ "the rest part of original query", where $\rightarrow$ means



(a) Right decomposed query plan of q3 (b) Left decomposed query plan of q4

Fig. 5: Decomposed query plan of q3 and q4



(c) Optimized JQP of q3 and q4

Fig. 6: Example of applying decomposition sharing technique.

connecting the output of left operation (upstream) to the input of right operation (downstream).

Essentially, a sub-query is a source query of its original query. Applying DST reduces global execution cost because the generated sub-queries from multiple queries may be simply combined (if they are identical) or may be optimized by applying sharing techniques such as MST. Figure 5 (a) and (b) illustrate one of decomposed query plans of q_3 = SEQ(E_1, E_2, E_4) and q_4 = SEQ(E_2, E_4, E_3), respectively. Each of the decomposed query plans can be denoted as SEQ(E_2,E_4) $\rightarrow$ SEQ(E_1,$\{E_2$,$E_4\}$), and SEQ(E_2,E_4) $\rightarrow$ SEQ($\{E_2$,$E_4\}$,E_3), where $\{E_2$,$E_4\}$ stands for the composite event type generated from SEQ(E_2,E_4). Note that they employ common sub-queries: q'_3 = q'_4 = q_x = SEQ(E_2,E_4), we can therefore simply combine them to serve both queries.

Given two pattern queries (q, q'), we apply DST in the following two steps.

Step 1: rewrite the query plan into different decomposed plans.

Step 2: for each pair of sub-queries from the decomposed plans, they are combined into one if they are identical. Otherwise, apply MST between them.

Example 2. *As shown in Figures 6, (a) represents the original JQP of* q_3 *and* q_4*; (b) represents the JQP of combined decomposed plan of* q_3 *and* q_4*; (c) represents the JQP after combining their common sub-queries:* q'_3 = q'_4 = q_x = *SEQ(*E_2,E_4*).*

Application of DST. A naive solution to identify sharing opportunities between two pattern queries with n operands based on DST is that we shall first generate all possible sub-queries, i.e., $\frac{n*(n-1)}{2}$ of one query, and then check each pair of sub-queries whether applying MST on them brings benefits. Identifying sharing between two pattern queries hence requires $O(n^4)$. Therefore, identifying among m queries based on this approach requires $O(\binom{m}{2} * n^4)$, which is computationally expensive considering m can also be large in practice.

In the following, we introduce a simple yet effective approach to realize the sharing by DST. Our approach is based on a concept named **interesting sub-query**, which is a common sub-query between a pair of two queries that may be used to generate optimized query plan by sharing the result of it. There can be multiple interesting sub-queries between two queries. We introduce an approach to identify the *interesting sub-queries* quickly.

The idea is that we directly search based on the operand list of each two queries. As a result, this problem can be essentially transformed into the problem of finding *all common substrings*.

A simple solution for finding all common substrings between string L and L' works as follows. First, build a *suffix tree* for the L, all the nodes in the developed suffix tree are marked as *left*. Then, all the suffixes of L' are inserted in the suffix tree. During the insertion, all old nodes that the suffixes pass through (or new node created) are marked as *right*. Finally, the paths of every node that are marked with both *left* and *right* are all common substrings of L and L'. The construction of suffix tree can be done in linear time [12]. Hence, the run-time complexity of this method is in linear time and is proportional to the number of matches.

After that, all identified common substrings with the length greater than one can be used to build common sub-queries that can be naturally combined to serve both queries. All common strings of length one are sequentially combined into "long" string. To preserve correct sequence, common strings of length one that appears in reverse order must be separate into different "long" string, when SEQ is used. Then, these "long" strings are used to build an MST applicable sub-queries that can be shared by both queries by merging its results.

Example 3. *We use q_6 = SEQ(E_1,E_2,E_3,E_5,E_6,E_7,E_8), q_7 = SEQ(E_1,E_3,E_6,E_5,E_7,E_8) as an example to illustrate the searching for sharing opportunities. First, we identify all common substrings based on their operand list. Five common substrings can be identified as follows, S_1:"E_1"; S_2:"E_3"; S_3:"E_5"; S_4:"E_6"; S_5:"E_7, E_8". Next, we merge all common substrings of length one into one string. Since S_3 and S_4 appear in reverse order in two SEQ queries, they are separated. As a result, three common strings are finally generated, MS_1: "E_1, E_3, E_5"; MS_2: "E_1, E_3, E_6"; S_5: "E_7, E_8". Then, MS_1 is used to build q_s = SEQ(E_1, E_3, E_5), MS_2 is used to build q_s' = SEQ(E_1, E_3, E_6), S_5 is used to build q_s'' = SEQ(E_7, E_8). q_s, q_s' and q_s'' are then marked as interesting sub-queries of q_6 and q_7.*

It is noteworthy that there might be sharing opportunities between the generated sub-queries as well. After we identify all the interesting sub-queries, we need to search *recursively* among them. This process may create further sub-queries. The searching only stops when no more interesting sub-queries can be identified.

There are two issues worth noting. First, we maintain all interesting sub-queries between every pair of two queries. The selecting of those sub-queries are left to the query planner to decide as we discuss in Section V. Second, given the commutative and associative properties of CONJ/DISJ, we pre-sort non-ordered operators on their operand list according to a predefined order (e.g., lexicographical order), so that the same method can be used for them.

Here, we further highlights an example where MST and DST work in combination to enable sharing between two queries, which otherwise have to be executed independently.

Example 4. *Only applying MST or only applying DST without MST between q_8 = SEQ(E_1,E_2,E_3,E_5) and q_9 = SEQ(E_1,E_3,E_4) does not generate any alternative plans. However, we can decompose them into SEQ(E_1,E_2,E_3) $\rightarrow$ SEQ($\{E_1,E_2,E_3\}$,E_5) and SEQ(E_1,E_3) $\rightarrow$ SEQ($\{E_1,E_3\}$,E_4), respectively. After that, MST can be applied between the sub-queries SEQ(E_1,E_2,E_3) and SEQ(E_1,E_3) to reduce the total computing cost. Note, this example also reaffirms that sub-expression sharing is inadequate compared to our techniques. As according to sub-expression sharing, we can only merge the common prefix E_1 between two queries.*

Semantic equivalence. Essentially, decomposition changes the operands grouping of the original query plan. Since associative property holds for SEQ/CONJ/DISJ pattern query, changing the operands grouping does not alter the semantic of the query. Therefore, the decomposed plan is semantic equivalent to the original.

C. Operator Transformation Technique

It is noteworthy that only queries that use the same type of pattern operators can share computation between each other based on MST and DST. For instance, sharing opportunities between SEQ(E_1, E_2, E_3) and CONJ(E_1& E_2& E_3) are overlooked even they look at the same event types. However, we observe that the pattern operator itself can be transformed to each other. Thus, it is possible to create sharing opportunities with operator transformation. Based on this, we develop another technique called *Operator Transformation Technique (OTT)*.

Table I summarizes the formulation and description of the three transformation rules. The transformation is comprehensive for the operators that are considered in this paper. Examples are further presented in Figure 7.

Given two pattern queries q, q', according to Table I, if there is a rule to transform the operator of q into the operator of q', we can enable sharing of OTT in the following two steps.

Step 1: transform q into q^* based on the operator transformation rules so that q^* use the same operator as q'.

Step 2: apply DST between q^* and q'.

TABLE I: Details of the three transformation rules. L stands for the operand list involving E_1,E_2,...,E_n.

Name	Formulation	Description
SEQ to CONJ	SEQ(L)=Filter$_{sc}$(CONJ(L))	Filter$_{sc}$ (op) works as follows: for each composite event generated by *op*, Filter$_{sc}$ output it if and only if E_1.ts < E_2.ts <...< E_n.ts, where ts refer to the timestamp of the event.
CONJ to DISJ	CONJ(L)=Filter$_{cd}$(DISJ(L))	Filter$_{cd}$ (op) works as follows: for each composite event generated by *op*, Filter$_{cd}$ eliminates it unless the result is a composite event consist of all types of events in L.
SEQ to DISJ	SEQ(L)=Filter$_{sc}$(Filter$_{cd}$(DISJ(L)))	It can be naturally derived by composing SEQ to CONJ and CONJ to DISJ.

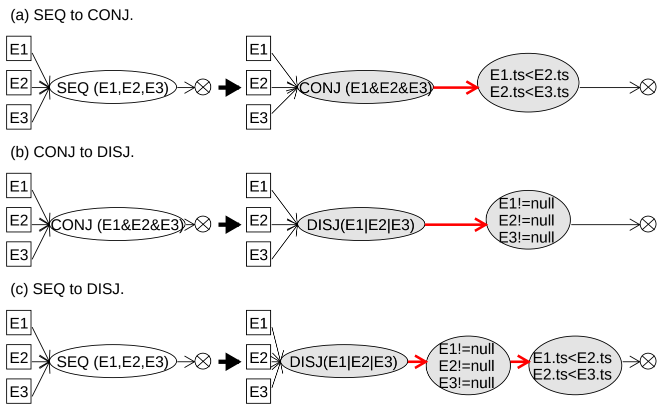


Fig. 7: Examples of three transformation rules.

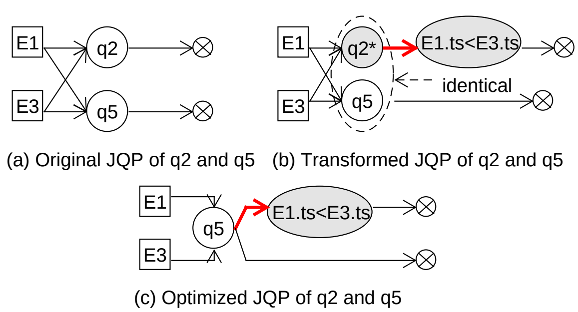


Fig. 8: Example of applying operator transformation technique.

Application of OTT. When OTT is enabled, the same method applied to DST can be further used for pattern queries with different operators based on transformation rules.

Semantic equivalence. The semantic equivalence of operator transformation is naturally given by the definition of pattern operators.

Example 5. *Consider* q_2 *= SEQ(*E_1, E_3*) and* q_5 *= CONJ(*E_1*&* E_3*). Since* q_2 *requires different pattern operators to* q_5*, there is no way to share computation between them. Figure 8 illustrates an example of applying operator transformation technique based on* q_2 *and* q_5*. (a) represents the original jumbo query plan (JQP) where* q_2 *and* q_5 *are executed independently; (b) represents the JQP after applying operator transformation on* q_2*; (c) represents the optimized query plan, where* q_5 *is combined with* q_2^**. Effectively, the optimized plan replaces* q_2 *with a* $Filter_{sc}$ *operation, which results in a more efficient JQP because of the significantly reduced input size.*

Furthermore, the following example highlights how OTT can work in combination with DST.

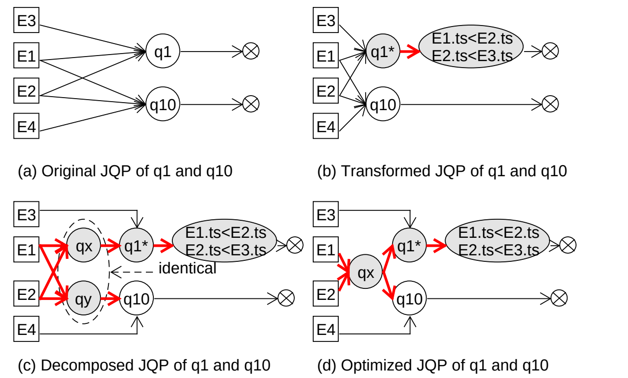


Fig. 9: Example of applying OTT in combination with DST.

Example 6. *Consider query* q_{10} *= CONJ(*E_1*&*E_2*&*E_4*) and* q_1 *= SEQ(*E_1*,* E_2*,* E_3*). According to the transformation rules,* q_1 *can be transformed into* $q_1^* \rightarrow Filter_{sc}$*, where* q_1^* *= CONJ(*E_1 *&*E_2*&*E_3*). There is still no way to share computation between* q_{10} *and* q_1^* *even with MST applied. However,* q_{10} *and* q_1^* *each can be decomposed with sub-query:* q_x *=* q_y *= CONJ(*E_1*&*E_2*), which can be simply combined to serve both* q_{10} *and* q_1*, as shown in Figure 9.*

D. Extension of sharing techniques

The previous subsections assume the same window constraints for all pattern queries. We now discuss how to extend our sharing techniques to handle nested pattern queries and queries with different window sizes.

Handling Nested Pattern Query. To efficiently and correctly identify sharing opportunities within and between nested CEP queries, we divide nested CEP queries into multiple non-nested sub-queries, and then apply sharing techniques between every pair of generated non-nested pattern queries.

DEFINITION 2. *The* ***nested level*** *specifies the nested layer of nested pattern queries. We denote the most inner nested layer as level* 1*, its closest outer layer as level* 2*, and so on until most outer layer as level* n*.*

We divide an nested pattern query into a series of non-nested sub-queries in an iterative process. Given an nested pattern query q, start from level $n-1$, divide it into a sub-query q^{inner} and connect q^{inner} to q by replacing the corresponding part of *PatternList* by event type of output of q^{inner}. Thereafter, q can be denoted as $q^{inner} \rightarrow q*$, where $q*$ becomes non-nested. Repeat the procedure on q^{inner} until there is no more nested pattern query. If more than one nested sub-queries in the same nested level, we divide them together as $(q^{inner_1}$,

q^{inner_2} ,..., , q^{inner_n}) $\rightarrow q^*$, where q^{inner_1} , q^{inner_2} ,..., , q^{inner_n} are all connected to q^*.

After dividing both queries, we can apply the aforementioned three sharing techniques among the generated non-nested sub-queries.

Example 7. *Table II illustrates the process of dividing q_{11} = SEQ(E_1, DISJ($E_4|E_5$), CONJ(E_2&E_3)) and q_{12} = SEQ(E_1, CONJ(E_2&E_3)). Note that, $E_{\gamma'}$ = $\{E_4 \mid E_5\}$ and $E_{\gamma''}$ = $\{E_2$ & $E_3\}$ denote the composite event type generated from corresponding inner nested sub-queries (i.e., $q_{11}^{inner_1}$ and $q_{11}^{inner_2}$).*

Table III illustrates the searching process of identifying sharing opportunities among the generated non-nested sub-queries of q_{11} and q_{12}, which requires six iterations. At 3^{rd} iteration, we identify CONJ(E_2&E_3) as a common sub-query, which is then marked as "interesting sub-query." At 6^{th} iteration, we identify SEQ(E_1, $E_{\gamma''}$) as MST applicable sub-query, which is also marked as "interesting sub-query". Both "interesting sub-queries" are kept, and let the query planner decide which one should be eventually selected to optimize the jumbo query plan globally.

TABLE II: Divide q_{11} and q_{12}.

Target	Final divided query plan	Detail explanations on terms
q_{11}	($q_{11}^{inner_1}$, $q_{11}^{inner_2}$) $\rightarrow q_{11}^*$	$q_{11}^{inner_1}$ = DISJ(E_4| E_5), $q_{11}^{inner_2}$ = CONJ(E_2&E_3), q_{11}^* = SEQ(E_1,$E_{\gamma'}$, $E_{\gamma''}$)
q_{12}	$q_{12}^i nner \rightarrow q_{12}^*$	q_{12}^{inner} = CONJ(E_2&E_3), q_{12}^* = SEQ(E_1, $E_{\gamma''}$)

TABLE III: Identifying sharing opportunities between q_{11} and q_{12}.

Iteration	sub-queries of q_{11}	sub-queries of q_{12}	output
1	$q_{11}^{inner_1}$	q_{12}^{inner}	*null*
2		q_{12}^*	*null*
3	$q_{11}^{inner_2}$	q_{12}^{inner}	CONJ(E_2&E_3)
4		q_{12}^*	*null*
5	q_{11}^*	q_{12}^{inner}	*null*
6		q_{12}^*	SEQ(E_1, $E_{\gamma''}$)

Handling Different Window Constraints. Pattern queries subscribed by multiple users may have different window constraints. We now discuss how to apply the sharing techniques discussed before to pattern queries with different window constraints.

Consider applying sharing techniques as before on two queries with different window constraints, the resulting source query (denoted as q_s), and the beneficiary query (denoted as q_b) may have different window constraints. We denote their shared composite event as E_{compo}, and denote E_{first} and E_{last} as the first and the last primitive events in E_{compo}, respectively. For example, consider q_2 (as q_s) = SEQ(E_1,E_3) and q_1 (as q_b) = SEQ(E_1,E_2,E_3). E_{first} = E_1, E_{last} = E_3, and E_{compo} is $\{E_1,E_3\}$. E_{compo} should be detected by q_2, and be shared to q_1. Intuitively, if E_{last} arrives exceeding the larger window constraint of them, no composite event should be produced.

The basic idea is that we first assume q_s and q_b are shareable, and then filter out the results which violate window constraint. We analyze two cases of varying window constraints between q_s and q_b, and we have following observations for each case.

Case 1: q_s has a larger window than q_b. In this case, we have three scenarios to consider: 1) If E_{last} arrives within q_b's window, E_{compo} is detected by q_s, and is shared by q_b (denoted as "shared"); 2) If E_{last} arrives exceeding q_b's window but within q_s's window, E_{compo} is being detected by q_b, but should not be shared by q_b (denoted as "filtered"); 3) If E_{last} arrives exceeding q_s's window, E_{compo} is not generated and can be safely discarded since it violates both queries' window constraints (denoted as "discard").

Case 2: q_s has a smaller window than q_b. In this case, we have two scenarios to consider: 1) If E_{last} arrives within q_s's window, E_{compo} is detected by q_s, and is shared by q_b (denoted as "shared"). 2) If E_{last} arrives exceeding q_s's window but within q_b's window, E_{compo} cannot be detected by q_s (denoted as "fail to share").

Based on the above analysis, we propose a *window mark-point* strategy to handle each case described as follows.

For the first case, we align the beneficiary query (q_b)'s upper window bound to the source query (q_s) and make the alignment point as the mark point. The results from q_s can be used to answer q_b by filtering out those composite events spanning across the mark point, which violate q_b's window constraint. For the second case, in order to solve the aforementioned "fail-to-share" issue, we propose to first extend the window of q_s into the same of q_b, resulting in q_s'. Thereafter, the extended source query (q_s') can be used to answer q_b as they have same window constraint. To answer the source query (q_s), we mark the alignment point of q_s and q_s' similar to the first case. Then, we can let the generated composite events from q_s' answer q_s, but filter out the events spanning across the mark point, since they violate window constraint of q_s.

Figure 10 illustrates the details of how to realize sharing opportunities between pattern queries with different window sizes: (a) where q_s has a larger window than q_b; (b) where q_s has a smaller window than q_b, with the "fail to share" issue; (c) how to extend q_s's window to solve the "fail to share" issue.

Finally, we analyze the overhead of the two cases in handling the window constraints. In the first case, the overhead of adding the additional filter operation is negligible. In the second case, the overhead mainly originates from the additional cost of extended source query compare to the original source query due to the larger window size. If the overhead of extending the window offsets its benefits, we choose not to apply sharing techniques on this pair of queries.

V. Query Planer

After applying the aforementioned sharing techniques in the previous section among multiple queries in a workload,

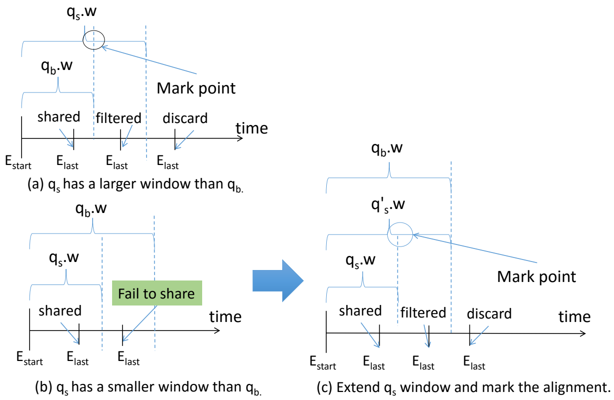


Fig. 10: Window mark-point strategy.

MOTTO generates many query plans. The query planner is used to identify the most efficient query plan. In this section, we introduce the implementation details. We first formulate the problem and subsequently describe our solution to the problem.

A. Problem Formulation

Applying sharing technique (i.e., MST/DST/OTT) between two queries causes a **JQP transformation** since it essentially causes a change in JQP. If a workload contains many queries, a significant number of alternative JQPs can be generated. This process can be *conceptually* modeled as a *JQP search tree* defined as below.

DEFINITION 3. ***JQP Search Tree.*** *Given a batch of queries, the default JQP stands for every pattern queries executed independently. Let the root of a tree be the default JQP. Starting from the root, for every JQP transformation, we add a new node to the tree and add an edge from the old node to the new node. The old node and the new node correspond to original JQP and new JQP, respectively. The edge represents the sharing techniques applied to make the transformation happen. The resulting tree is called a* ***JQP search tree***.

DEFINITION 4. *The* ***optimal jumbo query plan****, denoted by JQP_{opt} is the JQP such that $\forall JQPs$ in the JQP search tree, $Cost(JQP_{opt}) \leq Cost(JQP)$.*

Given a set of input queries, our optimization problem is to find an optimal JQP, which is the JQP_{opt} in the JQP search tree. From each node of the JQP search tree, we ideally need to consider all three sharing techniques for the JQP transformation. However, in practice, we only need to consider feasible JQP transformation.

Figure 11 illustrates the search process for conceptual JQP search tree for an example workload contains the following queries: q_1= SEQ(E_1, E_2, E_3), q_2= SEQ(E_1, E_3), q_3= SEQ(E_1, E_2, E_4), q_4= SEQ(E_2, E_4, E_3), and q_5= CONJ(E_1& E_3). Although each of the three techniques can be applied together, the decision of one sharing plan between two queries may be in conflict with the decision of sharing plan considering another query. For instance, the decision to share sub-query q_x= SEQ(E_1,E_2) between q_1 and q_3 and the decision to share computing of q_2 entirely to q_1 by MST are in conflict. Because the former requires q_1 been decomposed into SEQ(E_1,E_2) $\rightarrow$ SEQ({E_1,E_2},E_3), while the later requires q_1 been replaced by a merge operation CONJ({E_1,E_3}&E_2).

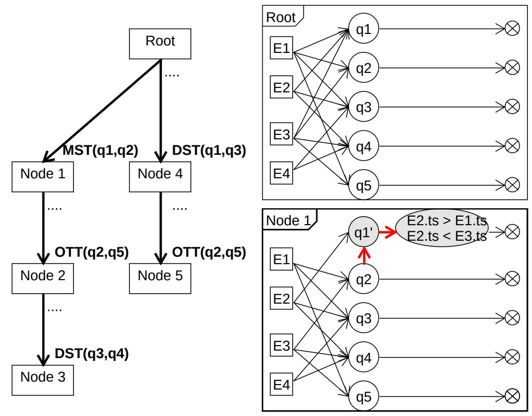


Fig. 11: Conceptual search tree of the example workload.

B. Generating Query Plan

We solve the problem by formulating the search process of JQP search tree to the problem of solving the Directed Steiner Minimum Tree (DSMT) problem [4].

The Directed Steiner Minimum Tree (DSMT) Problem is defined as follows. Given a directed weighted graph G = (V, E) with a cost c_i on each edge e_i, a specified root r $\in V$, and a subset of vertices $X \subseteq V$ (called *Terminal nodes*). The goal is to find a tree with minimum cost rooted at r and spanning all the vertices in X (in other words, r should have a path to every vertex in X). The cost of the tree is defined as the sum of the costs of the edges in the tree. Note that the tree may include vertices not in X as well (these are known as *Steiner nodes*).

Our optimization problem can be mapped to DSMT as follows. Queries in the submitted workload are treated as Terminal nodes since they are required by users and must be selected. "Interesting sub-queries" identified by application of sharing techniques are treated as Steiner nodes since they may or may not need to be selected (executed). We add one special query q_0 with no execution cost as root into G, called virtual ground. q_0 is treated as a Terminal node. For every query (including sub-query) we insert an edge from q_0 representing computing from scratch. If a query can be computed based on results of another, either query (in the case of MST &/ OTT) or sub-query (in the case of DST &/ OTT), then add an edge from the source query to the beneficiary query. The weight on edge represents the cost of executing the query based on the source (either q_0 or other queries).

The DSMT problem is well studied in the literature and it is NP-complete [4]. To consider the tradeoff between optimization overhead and quality of the solution, we apply branch-and-bound algorithm [4] to get an exact solution when optimization process takes less than the configured time budget

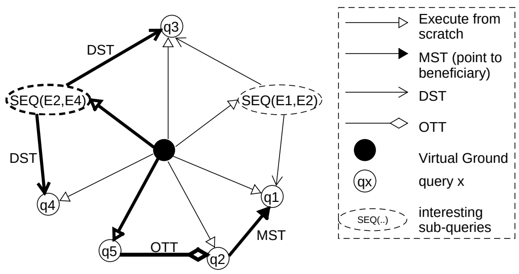


Fig. 12: Optimal plan of the example workload represented by Steiner Minimum Tree.

(e.g., 5 minutes) and switch to a simulated-annealing-based approximate solution [6] for larger problem sizes.

Example 8. *Figure 12 illustrates the optimal plan of the example workload represented by Steiner Minimum Tree (highlighted by the bold lines).*

VI. Cost model

As discussed in Section V, in order to solve the DSMT problem, we need to compare the cost of different jumbo query plan. Hence, in this section, we briefly show how to estimate the cost of query plan with one of the sharing techniques applied. By substituting the cost model for each individual query (the original query or sub-pattern query) of SAP ESP, we can then estimate the cost of query plan with or without sharing techniques applied. We denote the cost to calculate q_i based on q_x as Cost(q_i | q_x), which is the weight of the edge from q_x to q_i.

MST. Consider q_1 = SEQ(E_1, E_2, E_3) and q_2 = SEQ(E_1, E_3). q_1 can be computed based on q_2 through a merge operation (q_1') as illustrated previously in Figure 4. Therefore,

$$Cost(q_1|q_2)=Cost(q_1') + Cost(filter),$$

where q_1' = CONJ($\{E_1,E_3\}\&E_2$), and its cost can be obtained by substituting the cost model for each individual query of SAP ESP. Note that $\{E_1,E_3\}$ denotes the type of event generated from q_2.

DST. Consider q_3 = SEQ(E_1, E_2, E_4) and q_4 = SEQ(E_2, E_4, E_3). Right decomposed query plan of q_3 can be represented as SEQ(E_2,E_4) $\rightarrow$ SEQ (E_1,$\{E_2,E_4\}$), and q_4 can be left composed into SEQ(E_2,E_4) $\rightarrow$ SEQ($\{E_2,E_4\}$,E_3). Let q_x = SEQ(E_2,E_4), which is shared by both q_3 and q_4.

$$Cost(q_3|q_x)=Cost(q_3'), Cost(q_4|q_x) = Cost(q_4'),$$

where q_3' = SEQ(E_1,$\{E_2,E_4\}$), q_4' = SEQ($\{E_2,E_4\}$,E_3), and their cost can be calculated similarly as q_1'.

OTT. With OTT, queries with different pattern operator that otherwise have to be executed independently can now share their computation. Take q_2 = SEQ(E_1, E_3) and q_5 = CONJ(E_1& E_3) to illustrate. q_2 can be transformed to q_2' = CONJ (E_1&E_3) $\rightarrow$ filter$_{sc}$. Thus,

$$Cost(q_2)=Cost(q_2') + Cost(filter_{sc}).$$

In this case, q_2' can be answered by q_5 entirely (i.e., no compute cost) since they are identical,

$$Cost(q_2|q_5)=Cost(q_2') + Cost(filter_{sc})=Cost(filter_{sc}).$$

VII. Experiments

In this section, we experimentally evaluate MOTTO in SAP ESP. Overall, there are two groups of experiments. First, we show the performance comparison of different sharing techniques in two real application scenarios (Section VII-B). Second, we perform sensitivity studies to gain a better understanding of the effectiveness of MOTTO in varies aspects (Section VII-C).

A. Experimental Setup

All experiments are conducted on a virtual machine (VM) in SAP Monsoon cloud infrastructure with the configuration as follows. The VM is running on Intel Xeon-E7-4830 CPU processors (2.2 GHz) with Ubuntu 14.10. We fix one core to publish streaming data and all other cores for running MOTTO on top of SAP ESP. We vary the number of CPU cores used for MOTTO and SAP ESP for sensitivity studies and fix each core with 2GB memory. By default, the experiments run on a VM with 4 CPU cores and 8GB of memory.

To evaluate MOTTO, we have also implemented a number of sharing techniques:

- *NA*: the baseline executes pattern queries independently (without any sharing).
- *MST* (merge sharing technique): this approach requires one query to be shared entirely with other queries. This technique has been used in the previous study [10].
- *LCSE* (longest common sub-expression sharing): this approach identifies the longest common sub-expression among pattern queries to be shared. It has more sharing opportunities than MST and has been used in many previous studies (e.g., [13, 14, 15]).

Application Scenario. We consider two different application scenarios, namely stock market monitoring and data center monitoring [3]. According to the application scenarios, pattern queries in stock market monitoring have relatively longer operand lists compared to data center monitoring workload.

Since Figure 1 has already demonstrated the scenario of stock market monitoring, we here illustrate two sample queries for data center monitoring. q_a = SEQ(E_s,E_d,E_e,NEG(E_a)) is used to identify if any network transmitting problem. E_s and E_e stand for “Start transmit” and “End transmit” events, which are generated on the packet-sending-site to indicate the successful starting and ending of packet transmission. E_d denotes “Delivery successful notification” event, which is generated by the network routers that indicates the successful receive and re-route the corresponding packet. E_a denotes “Acknowledgment” event, which is generated from the receiving site. The sending site closes the transmitting with no “Acknowledgment” may indicate some problems happened. q_b = SEQ(E_s,E_e,E_a) is used with a post-aggregation operation to continuously calculate the average round-trip network communication time. These two pattern queries can be optimized by realizing sharing computation of E_s and E_e.

Data sets: For stock market monitoring study, we use real stock trade event data set [16]. The trade event data set includes 2 million trade events sorted according to the timestamp. Each event contains a stock symbol, timestamp and price information. Event types are defined according to the stock symbol attribute (13 different types). For example, trade events of stock symbols "AAPL", "MSFT", "INTC", and "FB" are denoted as event types E_1, E_2, E_3, and E_4.

In data center monitoring, there are two catalogs of events: 1) network transmission related event data, 2) virtual machine logging event data. For instance, we can define event E_1 as "one package received from IP: *xx.xx.149.22* is less than *5 byte*", E_2 as "one package received from the same IP and is larger than *5 kb*", and E_3 represents the rest events. We generate a data set containing 4 million events of 36 different event types based on a small sample set of real operation data provided by SAP HANA Data Center Intelligence [3].

Workload generations: In order to extensively study the performance of MOTTO, we generate multiple workloads based on the application scenarios. The workload does not contain duplicate queries, and the total number of queries generated is 100 for each case.

We use the same methodology for the workload generations of these two application scenarios. Specifically, we divide the workload into two groups with seven types of sharing opportunities shown in Table IV. The first group (basic workload group) contains non-nested pattern query of the same operator, and the same window constraint and can be used to evaluate the power of realizing the sharing opportunities from different approaches. The second group (complex workload group) contains pattern queries with different operators, and different window constraints as well as nested pattern queries. The sharing opportunities in the second group are overlooked in the comparison approaches (except MOTTO).

Queries in the first group are generated from query templates (Types $1 \sim 4$ in Table IV) with the following four types of sharing opportunities: 1) L is a prefix of L', 2) L is a suffix of L', 3) L is a subsequence but not a substring of L' and 4) L and L' have substrings but do not have the first three types of sharing opportunities. Queries of the first group are configured with the same window constraint of 10 seconds.

The second group contains the rest three types of sharing opportunities (Types $5 \sim 7$ in Table IV): 5) queries with different window constraints, 6) queries with same pattern list but different pattern operators, and 7) nested pattern queries with common sub-query in the most inner layer. The nested level is set to 2 by default, and their common sub-patterns are uniformly generated from samples according to Table IV.

We define a basic workload ratio r as the ratio of queries of the first group in the workload. Our goal is to understand the comparison of MOTTO to other techniques on different values of r in the workload, where r is varied from 0% to 100%. We generate query workloads with different sharing opportunity types randomly in a uniform distribution.

TABLE IV: Seven types of sharing opportunities, $E_x, E_y, E_{x'}$ and $E_{y'}$ belongs to one of the primitive event types, and x, y, x' and y' are distinct. Example queries q and q' are shown in the last column with the patterns L and L', respectively. Both queries are assumed to have the same window size (e.g., 10 seconds), unless specified in Type 5.

Group	Type	Description	Examples: q and q'
1) basic workload group	1	L is a prefix of L'	SEQ (E_1,E_2,E_3) and SEQ (E_1,E_2,E_3,E_x).
	2	L is a suffix of L'	CONJ (E_1 & E_2) and CONJ(E_x & E_1 &E_2).
	3	L is a subsequence but not a substring of L'	SEQ(E_1,E_2,E_3) and SEQ(E_1,E_2,E_x,E_3).
	4	L and L' have substrings but do not have the first three types of sharing opportunities	SEQ(E_x,E_1,E_2,E_y) and SEQ($E_{x'}$,E_1,E_2,E_y).
2) complex workload group	5	queries with different window constraints	SEQ (E_1,E_2,E_3) and SEQ (E_1,E_2,E_3,E_x).
	6	queries with same pattern list but different pattern operators	SEQ (E_1,E_2,E_3) and CONJ(E_1,E_2,E_3).
	7	nested pattern queries with common sub-query in the most inner layer	SEQ(E_x, SEQ(E_1,E_2),E_y) and CONJ($E_{x'}$ & SEQ(E_1,E_2) & $E_{y'}$).

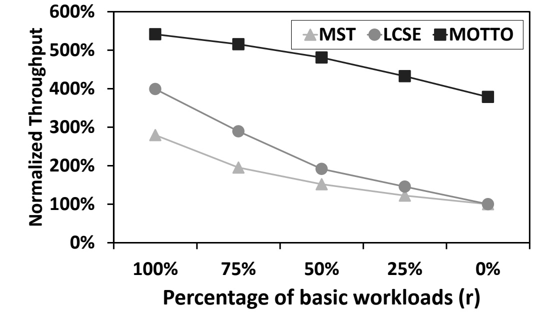
B. Overall Comparison

Figures 13a and 13b show the normalized throughput of the two applications of MOTTO in comparison with other techniques. All the measurements are normalized to the baseline (NA). The experiments are conducted on the VM with the default setting. We have made the following observations. First, as the basic workload ratio (r) decreases, the workload contains an increasing number of queries from the second group. As expected, the throughput of MST and LCSE decreases. MST and LCSE have limitations in realizing all sharing capabilities, especially from the second group. In contrast, MOTTO gives much more efficient execution solution. Second, as r decreases, the throughput of MOTTO decreases, because the sharing opportunities in queries from the second group tend to be more strict, which we study in detail in the sensitivity study. Third, the improvement is more significant in the stock market monitoring than in data center monitoring. As the stock market monitoring workload has longer operand lists, the sharing opportunities tend to be more. MOTTO takes advantage of those sharing opportunities and effectively reduces global execution cost.

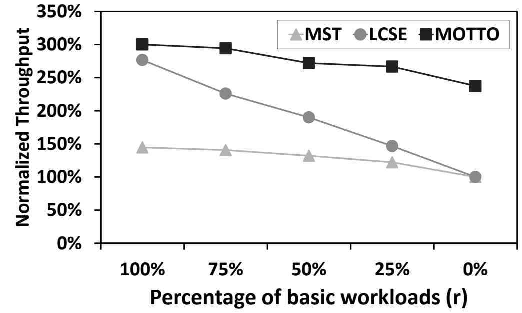
C. Sensitivity Studies

In this section, we perform sensitivity studies on stock market monitoring and have observed similar results on data center monitoring. Particularly, we use the stock market monitoring workload of r=100% (except that we use r=0% for nested pattern query studies).

Varying the number of queries. We study the impact of varying the number of queries to evaluate the overhead of the exact and approximate optimization approaches as well as their benefits in throughput. Figure 14a shows results on the throughput and the overhead. We have two major observations here. First, the improvement with both approaches scales with an increasing number of queries. As expected, the exact

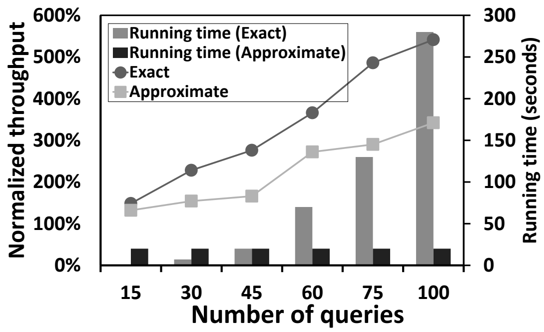


(a) Stock Market Monitoring

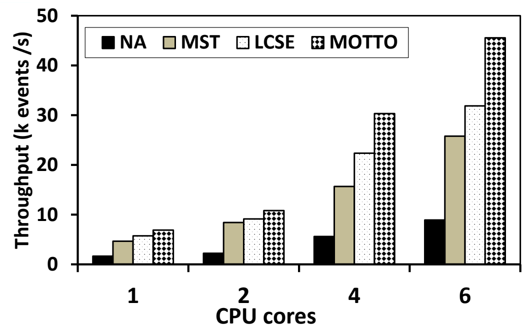


(b) Data Center Monitoring

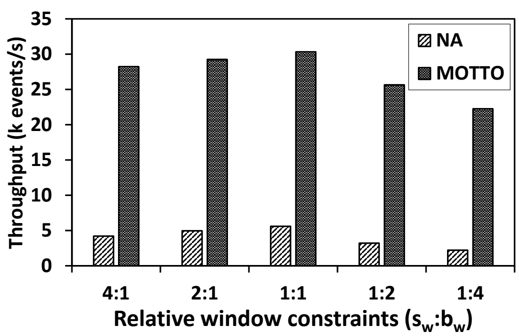
Fig. 13: Normalized throughput of the two applications.



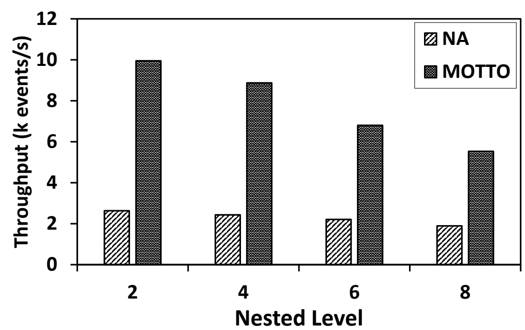
(a) Varying the number of queries



(b) Varying the number of CPU cores



(c) Varying the window constraints



(d) Varying the nested level of pattern queries

Fig. 14: Sensitivity Studies.

algorithm gives a better performance improvement. Second, the approximate solution coverages in a constant running time of 20 seconds with significant improvements, while the overhead of exact solution increases fast with increasing number of queries. As a balance in the tradeoff, we choose the setting that MOTTO has to converge within 5 minutes with an exact solution. Otherwise, it switches to approximate solution with a constant time budget.

Varying the number of CPU cores. Figure 14b shows the throughput of varying the number of CPU cores from 1 to 6. MOTTO demonstrates excellent scalability. Although our sharing techniques compose multiple queries into one, the parallelism does not decrease because of the sufficient number of sub-queries in the global query plan. It is our future work to study how MOTTO can scale to multi-socket and multi-core machines [17].

Varying the window constraints. In this experiment, we study the effectiveness of MOTTO of realizing sharing among queries of different window constraints. We denote the window constraint of all the source queries as s_w and the window constraint of the corresponding beneficiary query as b_w. Figure 14c shows the results when the relative window constraints between s_w and b_w is varied from 4:1 to 1:4. MOTTO improves the throughput on all settings. We have the following observations. First, the performance gain for the case where $s_w = b_w$ is the highest since sharing is enabled without extra overhead on handling different window constraints. Second, when $s_w > b_w$, the performance gain decreases slightly due to the additional filter operation. Third, the performance gain for the case where $s_w < b_w$ is the lowest because we need to extend the window of the source query, which essentially increases the chance of detecting the composite event of source query and hence adds computation cost.

Varying the nested level of pattern queries. We now study the sharing benefits among nested pattern queries. We vary the nested levels of all the nested queries in the workload from 2 (default) to 8, where the common sub-query is in the most inner layer. MOTTO is still able to significantly reduce the execution cost of multiple nested pattern queries, even though the deeper nested level decreases the sharing opportunities (as shown in Figure 14d).

VIII. Related work

Data stream processing systems have attracted a great amount of research effort [9, 11, 14, 17, 18, 19]. In the literature, a considerable amount of research has been devoted to the query optimization on complex event processing (CEP) [9, 11, 14, 18, 19]. Recently, several CEP systems have gained popularity such as IBM System-S [20], APAMA [21], StreamInsight (TimeStream) [22], TIBCO Event Processing [23] and HP CHAOS [24]. In the reminder of this section, we briefly describe those systems.

IBM System-S [20]: Pattern queries are treated as stateful operators in a stream application in IBM System-S. Schneider et al. [25] proposed a compiler and runtime system for IBM System-S that are capable of automatically extracting data parallelism from streaming applications for stateful operators.

StreamInsight [22]: TimeStream [22] extends programming model of StreamInsight for CEP to large-scale distributed execution by providing automatic supports for parallel execution, fault tolerance, and dynamic reconfiguration.

HP CHAOS [24]: Mo et al. [19] proposed several rewriting rules for efficient evaluation of nested pattern query on CHAOS. However, such single query optimization strategy overlooks the sharing opportunities among multiple queries. Mo et al. [10] introduced on-line analytical processing (OLAP) for multidimensional event pattern analysis at different levels of abstraction into CHAOS, where they have considered sharing results from one level query to another. MOTTO implements a similar technique called MST, and our experiment has shown the insufficiency of MST in realizing sharing opportunities in complex query workloads. A more recent work based on CHAOS [26] proposed an optimizer for CEP, which identifies opportunities for effectively shared processing by leveraging time-based event correlations among queries. In contrast, MOTTO, a multi-query optimizer in SAP ESP, is based on decomposition and operator transformation on pattern query processing. Moreover, MOTTO is successfully extended to handle both nested pattern query and queries with different window constraints, which are not explicit mentioned in any existing works.

IX. Conclusion

This paper presents MOTTO, a multi-query optimizer for complex event processing in SAP ESP. MOTTO realizes more sharing opportunities by introducing pattern query decomposition and transformation. Those sharing techniques are also extented to support multiple nested pattern queries and pattern queries with different window constraints. Experiments demonstrate the efficiency of MOTTO with both real-world application scenarios and sensitivity studies.

Acknowledgement

This work is partially funded by the Economic Development Board and the National Research Foundation of Singapore. Bingsheng's work is partially funded by a MoE AcRF Tier 1 grant (T1 251RES1610) in Singapore.

References

[1] "Complex Event Processing: Beyond Capital Markets," http://aitegroup.com/report/complex-event-processing-beyond-capital-markets.
[2] W. Fengjuan and et al., "The Research on Complex Event Processing Method of Internet of Things," in *ICMTMA*, 2013.
[3] "Overview - SAP HANA Data Center Intelligence," https://archive.sap.com/documents/docs/DOC-60065.
[4] A. Lucena and J. E. Beasley, "A Branch and Cut Algorithm for the Steiner Problem in Graphs," *Networks*, 1998.
[5] M. Charikar and et al., "Approximation Algorithms for Directed Steiner Problems," in *SODA*, 1998.
[6] A. Zelikovsky, "A Series of Approximation Algorithms for the Acyclic Directed Steiner Tree Problem," *Algorithmica*, 1997.
[7] J. W. Van Laarhoven, "Exact and heuristic algorithms for the Euclidean Steiner tree problem," 2010.
[8] W. White and et al., "What is "Next" in Event Processing?" in *PODS*, 2007.
[9] N. P. Schultz-Møller and et al., "Distributed Complex Event Processing with Query Rewriting," in *DEBS*, 2009.
[10] M. Liu and et al., "E-Cube: Multi-dimensional Event Sequence Analysis Using Hierarchical Pattern Query Sharing," in *SIGMOD*, 2011.
[11] L. Brenna and et al., "Distributed Event Stream Processing with Non-deterministic Finite Automata," in *DEBS*, 2009.
[12] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica*, 1995.
[13] M. Hong and et al., "Rule-based Multi-query Optimization," in *EDBT*, 2009.
[14] M. Akdere and et al., "Plan-based Complex Event Detection Across Distributed Sources," *Proc. VLDB Endow.*, 2008.
[15] A. Bremler-Barr and et al., "CompactDFA: Scalable Pattern Matching Using Longest Prefix Match Solutions," *IEEE/ACM Trans. Netw.*, 2014.
[16] "netfonds," http://www.netfonds.no.
[17] S. Zhang and et al., "Revisiting the Design of Data Stream Processing Systems on Multi-Core Processors," in *ICDE*, 2017.
[18] Y. Mei and S. Madden, "ZStream: A Cost-based Query Processor for Adaptively Detecting Composite Events," in *SIGMOD*, 2009.
[19] M. Liu and et al., "High-Performance Nested CEP Query Processing over Event Streams," in *ICDE*, 2011.
[20] H. Andrade and et al., "Processing High Data Rate Streams in System S," *J. Parallel Distrib. Comput.*, 2011.
[21] "Apama Complex Event Processing Engine," http://www.softwareag.com/corporate/products/az/apama/.
[22] Z. Qian and et al., "TimeStream: Reliable Stream Computation in the Cloud," in *EuroSys*, 2013.
[23] "TIBCO Event Processing," http://www.tibco.com/products/event-processing.
[24] C. Gupta and et al., "CHAOS: A Data Stream Analysis Architecture for Enterprise Applications," in *CEC*, 2009.
[25] S. Schneider and et al., "Auto-parallelizing Stateful Distributed Streaming Applications," in *PACT*, 2012.
[26] M. Ray and et al., "Scalable Pattern Sharing on Event Streams," in *SIGMOD*, 2016.