# Rule-Based Multi-Query Optimization *

Mingsheng Hong
Dept. of Computer Science
Cornell University
mshong@cs.cornell.edu

Mirek Riedewald
Dept. of Computer Science
Cornell University
mirek@cs.cornell.edu

Christoph Koch
Dept. of Computer Science
Cornell University
koch@cs.cornell.edu

Johannes Gehrke
Dept. of Computer Science
Cornell University
johannes@cs.cornell.edu

Alan Demers
Dept. of Computer Science
Cornell University
ademers@cs.cornell.edu

## ABSTRACT

Data stream management systems usually have to process many long-running queries that are active at the same time. Multiple queries can be evaluated more efficiently together than independently, because it is often possible to share state and computation. Motivated by this observation, various Multi-Query Optimization (MQO) techniques have been proposed. However, these approaches suffer from two limitations. First, they focus on very specialized workloads. Second, integrating MQO techniques for CQL-style stream engines and those for event pattern detection engines is even harder, as the processing models of these two types of stream engines are radically different.

In this paper, we propose a rule-based MQO framework. This framework incorporates a set of new abstractions, extending their counterparts, physical operators, transformation rules, and streams, in a traditional RDBMS or stream processing system. Within this framework, we can integrate new and existing MQO techniques through the use of transformation rules. This allows us to build an expressive and scalable stream system. Just as relational optimizers are crucial for the success of RDBMSes, a powerful multi-query optimizer is needed for data stream processing. This work lays the foundation for such a multi-query optimizer, creating opportunities for future research. We experimentally demonstrate the efficacy of our approach.

## 1. INTRODUCTION

Query optimizers have been instrumental for the success of relational database technology. The cost difference between a good and a bad query plan can be several orders of magnitude. For data stream systems the stakes are even higher. Instead of one-shot queries in a relational DBMS, a stream system is processing *many continuous queries simultaneously*. These queries are active for long periods of time and they process massive streams in real time. A poor query implementation choice can negatively affect system performance for the lifetime of the query.

The key to achieving good stream processing performance is to optimize multiple queries together, rather than individually. In a stream query workload, it is often the case that multiple concurrently active queries can share state and computation. Query evaluation techniques that exploit this property are referred to as Multi-Query Optimization (MQO) techniques. The importance of MQO for stream processing is widely accepted and various stream MQO techniques have been proposed [10, 16, 12, 22, 14, 15, 7].

Unfortunately, existing MQO techniques apply only to very specific queries or workload properties. For example, predicate indexing [10, 16] is tailored for a set of selection operators that all read the same input stream. In addition, work on MQO techniques so far has happened in parallel for CQL-style stream engines [2, 5], referred to as *Relational Engines (RE)*, and event pattern detection engines [8, 21], referred to as *Event Engines (EE)*. The former use an operator model similar to relational databases, while the latter implement queries with automata. This has led to an unsatisfactory state of MQO, characterized by a confusing variety of individual techniques that apply to specific workloads or implementation models only. This prevents effective MQO for complex queries and leads to a situation where similar approaches might be re-invented by the different communities for REs and EEs.

To address these problems, we propose a Rule-based MQO framework, called *RUMOR*. It is inspired by the classical Query Graph Model (QGM) of RDBMSes [17], where query optimization techniques for *single* queries can be naturally modeled as transformation rules on query plans. RUMOR provides a modular and extensible framework, enabling new optimization techniques to be developed and incorporated incrementally into the system.

To support rule-based MQO, we have to extend the key abstractions that are used in a traditional RDBMS or stream system: physical operators, transformation rules, and streams. We introduce a small number of carefully designed abstractions that together create a powerful MQO framework. In fact, RUMOR incorporates all previously proposed MQO techniques for stream processing. In particular, it successfully incorporates MQO techniques from both relational stream engines and automata-based event processing engines. Hence an additional benefit of RUMOR is that it enables the unification of these diverse camps of stream processing systems. Experimental results for our prototype implementation indicate that we can efficiently process a large number of CQL-style relational

*This material is based upon work supported by the AFOSR under Award No. FA9550-06-1-0111, by the U.S. Department of Homeland Security under Grant Award Number 5-36423.5750, and by the National Science Foundation under Grant No. 0621438. Any opinions, findings, and conclusions or recommendations expressed in this publications are those of the authors and do not necessarily reflect the views of the sponsors.


*EDBT 2009*, March 24–26, 2009, Saint Petersburg, Russia.

stream queries, event processing queries, as well as *hybrid queries* involving query features from both types of query workloads.

RUMOR lays the foundation for multi-query optimizers (MQOptimizers) for data stream processing. It opens up opportunities for exciting future research on finding new rewrite rules and on extending the approach to cost-based MQOptimizers, incorporating ideas from the classical dynamic programming approach to cost-based *single* query optimization in RDBMSes [18].

**Contributions and roadmap.** Our contributions can be summarized as follows.

- We propose RUMOR, a rule-based MQO framework, which naturally extends the rule-based query optimization and query-plan-based processing model used by current RDBMSes and stream systems.
- We show how new and existing MQO techniques for relational stream engines and for event engines can be integrated into RUMOR. This is done by defining a small number of carefully designed abstractions.
- We demonstrate the efficacy of our approach by presenting experimental results using a prototype implementation of RUMOR.

RUMOR integrates MQO techniques for REs and EEs. For ease of exposition, in Section 2 and 3, we interleave the description of RUMOR and integration of MQO techniques for REs into RUMOR. We then describe the integration of MQO techniques for EEs in Section 4. The experimental results are presented in Section 5. Finally, we survey related work in Section 6, and conclude in Section 7.

# 2. RUMOR: PART I

RUMOR incorporates three abstractions, respectively extending physical operators, transformation rules, and streams. For ease of exposition, in this section we introduce only the first two abstractions (Sections 2.2 and 2.3), and show how they can be used to express a set of interesting MQO techniques (Section 2.4). We describe the last abstraction in Section 3. Due to space constraints, we choose to present RUMOR in an intuitive way, accompanied by examples.

## 2.1 Background

We briefly review the related concepts in a relational query processing engine. A *logical query* is specified by a user through a query language such as CQL, which has well-defined semantics. A *query optimizer* reads a logical query as input, and produces a *physical query*, also known as a *query plan*, as the result of optimization. The optimization process involves the application of *transformation rules*, also known as rewrite rules, on the query plans. A transformation rule maps one query plan to another semantically equivalent plan (e.g. pushing selection below join). The query plan produced by the optimizer is executed by the *query engine* to produce results conforming to the logical query semantics. We say the query plan *implements* its corresponding logical query.

For efficiency we want the query engine to process multiple queries together. We therefore extend the notion of a query plan to be one that implements *all* the currently active logical queries. A query plan is composed of *physical operators*, the basic scheduling and execution units in the engine. A physical operator *consumes* one or multiple input streams, and it *produces* one output stream. A physical operator is called the *consumer operator* of its input streams, and the *producer operator* of its output stream.

This paper focuses on rewrite rules for query plans.

## 2.2 Physical Multi-Operator

MQO techniques identify opportunities for sharing between operators, and they modify parts of the query plan to exploit these opportunities. For example, consider a query plan with multiple selection operators reading the same input stream. The *predicate indexing* MQO technique shares work among them by indexing the selection predicates of the operators. For each incoming stream tuple this index is probed. It returns all satisfied predicates at a much lower cost than the naive strategy of evaluating each selection predicate individually one-by-one [10, 16].

To model a set of operators with shared computation, we propose an abstraction called *physical multi-operator* (or *m-op*). We say that an m-op *implements* a set of operators. An m-op is defined as follows. For every stream $S$, $S$ is an input (resp. output) stream of the m-op, if and only if it is an input (resp. output) stream of at least one of the operators the m-op implements. The semantics of the m-op are defined as follows. Let $t$ be an input tuple arriving in stream $S$. Then the m-op *conceptually* executes all its operators that have input stream $S$, and it writes the output produced for $t$ by these operators to the corresponding output streams. The state of the m-op *conceptually* is a vector; each entry in the vector is equivalent to the state of one of the implemented operators if this operator was executed in isolation.

Notice that the definition of m-op semantics is based on the one-by-one execution of the implemented operators without sharing state. This defines the correct semantics, but of course *our goal is to find more efficient m-op implementations that still guarantee the same input-output behavior as defined by the above semantics*. Intuitively, the m-op consumes the set of input streams of the physical operators it implements, and it produces a corresponding set of output streams. The notion of consumer and producer operators for physical operators extends naturally to m-ops.

The m-op abstraction generalizes the traditional physical operator abstraction. It therefore takes the place of a physical operator in RUMOR: A query plan is composed of m-ops, and an m-op is the new scheduling and execution unit in the query engine. We illustrate the use of m-ops in the following example.

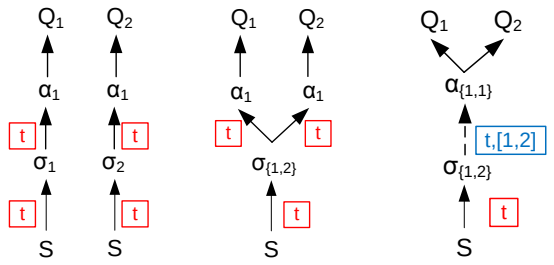
EXAMPLE 1. *Figure 1(a) shows two queries $Q_1$ and $Q_2$, where $\sigma_1$ and $\sigma_2$ are selection operators, and $\alpha_1$ denotes a sliding window aggregation operator, occurring in both queries. Note that we use the query name to denote its output stream name.*

*Let $\sigma_{\{1,2\}}$ denote the m-op implementing $\sigma_1$ and $\sigma_2$ with predicate indexing. It produces two output streams, respectively corresponding to the output streams of $\sigma_1$ and $\sigma_2$ in Figure 1(a). Figure 1(b) shows the query plan using $\sigma_{\{1,2\}}$.*

*Suppose tuple $t$ in stream $S$ satisfies both $\sigma_1$ and $\sigma_2$. In Figure 1(a), an output tuple is produced by both $\sigma_1$ and $\sigma_2$. In Figure 1(b), an output tuple is produced by $\sigma_{\{1,2\}}$ on each of its two output streams.*

## 2.3 Transformation Rules on m-ops

We now extend the traditional transformation rules, which operate on query plans composed of physical operators, to *multi-query transformation rules*, or *m-rules* for short. M-rules operate on query plans composed of m-ops. Similar to a traditional transformation rule, an m-rule consists of a pair of *condition* and *action* functions [17]. The condition function is a Boolean side-effect-free function on the query plan to identify opportunities for sharing. Once a sharing opportunity is identified among a set of operators in a query plan, the action function modifies the query plan by replacing that set of operators with a single m-op. We say the m-rule *maps* a set of m-ops to a single m-op, or it *merges* these operators.




(a) Queries and Input (b) Plan with Selection m-op (c) Plan with Channel

**Figure 1: Query Plans in RUMOR (Red Rectangles Represent Stream Tuples; the Blue Rectangle is a Channel Tuple)**

More precisely, the condition of an m-rule is a function from the powerset of the set of all possible m-ops to $\{\text{true}, \text{false}\}$. For a given set of m-ops, the rule can only be applied if the condition function evaluates to true.

The action of an m-rule is another function. This function maps a set of m-ops (for which the condition function evaluated to true) to a single m-op, referred to as the *target m-op*, which implements the input m-ops more efficiently with an MQO technique. In the query plan, we simply replace all edges that previously connected other operators with the to-be merged operators by edges to the corresponding input and output streams of the target m-op.

## 2.4 Expressing MQO Techniques with m-ops and m-rules

Most of the existing specialized MQO techniques share work among operators reading the *same stream(s)*. These can be implemented in RUMOR through m-ops and m-rules. For example, we can model predicate indexing for equality predicates on a single attribute as an m-rule as follows. The condition of the m-rule evaluates to true only for a set of selection operators that all read the same stream and whose selection predicate is an equality predicate on the same attribute $A$. The action rule then replaces them with a target m-op that uses a hash index on attribute $A$ for a more efficient evaluation of the selection predicates of these operators.

It is not hard to see that all these previously proposed MQO approaches for multiple selection [10, 16], aggregation [22], and join [12] operators can be expressed similarly through corresponding m-ops and m-rules. The first three rows in Table 1 summarize these rules. Notice that in data stream processing systems, join and aggregation operators usually contain window specifications to prevent unbounded memory consumption. Also, aggregation operators may contain optional group-by specifications. For each operator type $\tau$, we name the corresponding m-rule $s_\tau$, indicating that it is an m-rule for instances of operator $\tau$ that all process the same input stream(s). The remaining rows in Table 1 will be discussed later in this paper. The current set of m-rules is not intended to be complete—the extensible nature of rule-based query optimization allows for adding new rules.

# 3. RUMOR: PART II

In Section 2, we have shown how to use the two abstractions m-op and m-rule to express a set of existing MQO techniques, including predicate indexing [10, 16], multiple aggregate processing with different group-by specifications [22], and shared join evaluation [12]. All of these techniques attempt to share work among "similar" operators reading identical stream(s).

A complementary MQO approach is to support sharing also in the case where "similar" streams are processed by identical operators. Consider the example in Figure 1(a). The same aggregation operator $\alpha_1$ occurs in both queries, and it aggregates some subset of the tuples from $S$. However, if $\sigma_1$ and $\sigma_2$ have different selection predicates, then m-op $\sigma_{\{1,2\}}$ will have two different output streams, as shown in Figure 1(b). This in turn implies that the two instances of $\alpha_1$ read different input streams and therefore cannot be combined using m-rule $s_\tau$. On the other hand, the selection predicates might be similar so that many tuples that pass $\sigma_1$ might also pass $\sigma_2$. In that case it would be beneficial to avoid duplication of stream tuples as well as duplication of work for the aggregation. This example might appear like a rare corner case, but streams with common tuples occur frequently in practice as a result of multiple operators processing the same input stream. Recent work on MQO techniques like precision sharing join [14] and shared fragment aggregation [15] have shown that exploiting these sharing opportunities can result in significant performance improvement.

In this section, we propose an abstraction to model such MQO techniques in RUMOR. We refer to it as a *channel*. Channels generalize and replace streams in RUMOR (Section 3.1). We then describe how to decide which streams should be replaced with channels in the query plans in RUMOR (Section 3.2). Finally, we add a new set of m-rules and m-ops leveraging channels, which express both existing and new MQO techniques (Section 3.3).

## 3.1 Extending Streams to Channels

Logically, a channel is equivalent to the union of a set of streams, The streams that are combined to form a channel are required to have compatible schemas. This can always be achieved by "padding" the schemas of individual streams with the attributes from other streams after appropriate attribute renaming.

Unlike a union of streams, a channel keeps track of which original stream a tuple belongs to. We say the channel *encodes* these streams. More formally, a channel encodes a set of data streams with union-compatible schemas as follows. The channel is defined as the union of its streams, but each stream tuple has an additional attribute called *membership component*. The membership component specifies the set of streams to which this tuple belongs. For efficiency, the membership component is implemented by a bit vector.

Through the use of a channel we can share work in two ways. First, when identical tuples from different streams are encoded as a single channel tuple, their *space* is shared. Second, when multiple streams are encoded into the same channel, the *computation* of their consumer operators may be shared.

Clearly, channels generalize streams. In RUMOR, they take the place of streams as the input and output of m-ops. For each m-op, the input (resp. output) channels together partition the set of input (resp. output) streams of this m-op. When an m-op $o$ processes an input channel tuple $t$, a *decoding* and an *encoding* step are involved as follows. $o$ first determines to which set of input streams $t$ belongs, so that it conceptually only evaluates those physical operators implemented by $o$ that take this tuple as input. This is the decoding step. Similarly, when $o$ is about to produce a set of output tuples, it needs to encode it into a set of channel tuples with the appropriate stream membership component, and then write them to the appropriate output channels. This is the encoding step.

Note that the decoding and encoding steps can often be implemented very efficiently, or might actually not be necessary at all. For example, consider an m-op $\pi_{\{1,\cdots,n\}}$ implementing $n$ projections with the same projection specification, but with different input

| m-rule name | Set of input operators to which the m-rule is applicable | Target m-op |
|---|---|---|
| $s_\sigma$ | A set of selection operators which read the same stream | Predicate indexing [10, 16] |
| $s_\alpha$ | A set of aggregation operators which read the same stream, with the same aggregate function but potentially different group-by specifications | Shared aggregate evaluation [22] |
| $s_\bowtie$ | A set of join operators which read the same two streams, with the same join predicate but potentially different window lengths | Shared join evaluation [12] |
| $c_\alpha$ | A set of aggregation operators reading sharable streams, with the same definition | Shared fragment aggregation [15] |
| $c_\bowtie$ | A set of join operators which read sharable streams, with the same definition | Precision sharing join [14] |
| $s_;$ (or $s_\mu$) | A set of ; (or $\mu$) operators reading the same two streams, with the same definition | Common Subexpression Elimination (Section 4.3) |
| $c_;$ (or $c_\mu$) | A set of ; (or $\mu$) operators which a) have the same definition b) read sharable input streams for the first input stream parameter, where these input streams are produced by the same m-op c) read the same input stream for the second input stream parameter | Channel Based MQO (Section 4.4) |

**Table 1: Representative m-rules to Express Existing and New MQO Techniques**

streams $S_1$ through $S_n$. Suppose these $n$ input streams are encoded by channel $C$, and the $n$ output streams are encoded by channel $D$. In this case, for each input channel tuple $t$ from $C$, $\pi_{\{1,\cdots,n\}}$ needs to perform projection only once and to produce only one output channel tuple in $D$, keeping the membership component of $t$ intact in the output $D$ tuple.

To continue Example 1, we can use a channel to encode the two output streams of $\sigma_{\{1,2\}}$ in Figure 1(b), resulting in the query plan shown in Figure 1(c). Here the dashed arrow represents the channel, and $\alpha_{\{1,1\}}$ represents the aggregation m-op, implemented by the shared fragment aggregation technique described in [15]. Suppose an input tuple $t$ from stream $S$ satisfies both predicates in $\sigma_{\{1,2\}}$. $\sigma_{\{1,2\}}$ then produces a single output channel tuple, represented by the blue rectangle in Figure 1(c). That channel tuple has the same content as the input tuple $t$, but is associated with a membership component denoted as [1,2], indicating that it belongs to both output streams of $\sigma_{\{1,2\}}$.

Note that ideas similar to channels were used for specific MQO algorithms for joins and aggregates in relational engines [14, 15]. Our contribution is to propose the addition of the channel concept to an MQO framework as a general abstraction for sharing work. As we will show in Section 4, the combination of m-ops, m-rules, and channels also leads to powerful new MQO techniques for event processing queries.

## 3.2 Mapping Streams to Channels

Channels are a powerful mechanism that allows us to aggressively share work among operators that read even different streams. Given a set of streams, how do we decide which ones to map to the same channel? The following tradeoffs have to be taken into account. First, if two streams $S_i$ and $S_j$ are encoded into the same channel, then stream tuples with the same content can share storage by being represented as the same channel tuple. Second, if the consumer operators of $S_i$ and $S_j$ have the same definition, the evaluation on channel tuples will be more efficient than evaluating tuples from stream $S_i$ and $S_j$ separately. Third, mapping multiple streams to the same channel creates overhead. Time-wise, with multiple streams being mapped to the same channel, the consumer m-op of this channel now has to process the membership component of each input tuple. Space-wise, each channel tuple has to carry the membership component.

Based on these tradeoffs, it is clear that streams should only be mapped to the same channel if there is a large enough fraction of channel tuples that belong to multiple streams and if the streams are consumed by identical operators. We now propose a simple lightweight heuristic for deciding which streams to map to the same channel. This heuristic was used in our experimental evaluation and works very well in practice. More sophisticated cost models can be developed, but are left for future work.

Our proposed algorithm for deciding which streams to merge into a single channel is based on the concept of *sharable* streams. Two streams $S_1$ and $S_2$ are sharable, denoted $S_1 \sim S_2$, if the following holds:

**Base case 1.** If $S_1 = S_2$, then $S_1 \sim S_2$.

**Base case 2.** If $S_1$ and $S_2$ are produced by two stream sources that are labeled to be sharable, then $S_1 \sim S_2$.

**Output of unary ops.** For any unary operators $o_1, o_2$, if $S_1 := o_1(T_1), S_2 := o_2(T_2), o_1 = o_2$, and $T_1 \sim T_2$, then $S_1 \sim S_2$.

**Output of binary ops.** For any binary operators $o_1, o_2$, if $S_1 := o_1(T_1, U_1), S_2 := o_2(T_2, U_2), o_1 = o_2, T_1 \sim T_2$, and $U_1 \sim U_2$, then $S_1 \sim S_2$.

**Special case for selection.** For a selection operator that reads $T$ and produces $S$, $S \sim T$.

**Symmetry** $\forall S_1, S_2 : S_1 \sim S_2 \Rightarrow S_2 \sim S_1$

**Transitivity** $\forall S_1, S_2, S_3 : (S_1 \sim S_2 \wedge S_2 \sim S_3) \Rightarrow S_1 \sim S_3$

Intuitively, streams are sharable if they are the result of the same query plans, modulo any selection operators anywhere in the plan, applied to the same input streams. Clearly, $\sim$ is an equivalence relation and it generalizes the stream identity relation $=$. This makes $\sim$ very efficient to compute and store.

Even if streams are sharable, we map them to the same channel only if they originate from the same m-op. If they are produced by different m-ops, the runtime system would have to synchronize these operators to ensure identical tuples are available at the same time for the channel encoding step. This is conceptually not hard, but an analysis of the tradeoffs is beyond the scope of this paper.

Furthermore, if sharable streams are consumed by m-ops that cannot share any work, there is no benefit in encoding them with the same channel. Typically two m-ops reading different streams can effectively share work only if they have exactly the same definition. For example, two selection operators with the same predicate, two projection operators with the same projection specification, or two aggregation operators with the same aggregate function and group-by specification can share work when reading two different input streams that are sharable. We only consider this type of work sharing with channels in this paper.

| Traditional abstraction | RUMOR abstraction |
|---|---|
| physical operator | m-op (Section 2.2) |
| transformation rule | m-rule (Section 2.3) |
| stream | channel (Section 3) |

**Table 2: Correspondence between new and existing abstractions for building a stream system**

To conclude, given a set of streams $S_1$ through $S_n$, we map them to the same channel, only if (a) the $S_i$'s belong to the same equivalence class defined by $\sim$, (b) the $S_i$'s are produced by the same m-op, and (c) the consumers of the $S_i$'s have the same definition. These criteria, referred to as *channel-based MQO sharing criteria*, are currently used in RUMOR. When these criteria are met, we map the streams to a single channel and then combine the (identical) consumers of the $S_i$'s into the same m-op, achieving effective work sharing among them.

*The above sharing criteria may appear restrictive, but are met surprisingly often in practice. E.g., they apply when queries contain parametrized components that differ in some selection predicates but otherwise follow the same query template. For example, precision sharing join [14] and shared fragment aggregation [15] are both implicitly based on the above criteria, additionally limited to join operators and aggregation operators, respectively.*

## 3.3 Expressing MQO Techniques with Channels

To benefit from channels, we add the following new m-rules. For each operator type $\tau$ (e.g. selection, join, aggregation), we add an m-rule which identifies operators of type $\tau$ whose input streams satisfy the channel-based MQO sharing criteria defined at the end of Section 3.2. It then maps these operators to a single m-op. We refer to this m-rule as $c_\tau$, indicating that this is an m-rule for operators $\tau$ processing tuples from the same channel. For example, the fourth and fifth m-rule in Table 1 respectively express shared fragment aggregation [15] and precision sharing join [14].

Note the interesting duality between the two m-rules $s_\tau$ (Section 2.4) and $c_\tau$ of an operator type $\tau$. $s_\tau$ is applicable to a set of *sharable* operators (i.e., operators of type $\tau$) reading the *same* stream(s), whereas $s_\tau$ is applicable to a set of operators of the *same* definition, reading *sharable* stream(s). Assuming $\tau$ is unary, we present an illustration for the difference between $s_\tau$ and $c_\tau$. In Figure 2, the enclosing rectangle denotes the set of unary operators of type $\tau$, reading sharable streams. Each row labeled $S_i$ corresponds to a subset of operators of type $\tau$, reading the same stream $S_i$. Each application of $s_\tau$ will pick a row of operators here, and map them to an m-op. Repeated applications of this m-rule therefore form a partition of this set of operators. The setting of Figure 3 is similar to that of Figure 2. Each column corresponds to a set of operators of type $\tau$ with the same definition, reading a set of sharable input streams $S_1, S_2, \cdots$. One application of $c_\tau$ selects a column of operators, and maps them to an m-op. Repeated applications of $c_\tau$ therefore also form a partition of this set of operators.

As a result, for any operator in the shaded region $X$ (i.e., any operator with definition $o_1$, reading stream $S_1$), both $s_\tau$ and $c_\tau$ are applicable to it. Therefore, as in many other rule-based applications, different orderings of m-rule applications may result in different optimized query plans.

To summarize, Table 2 shows the newly proposed abstractions in RUMOR, and their correspondences with existing abstractions.

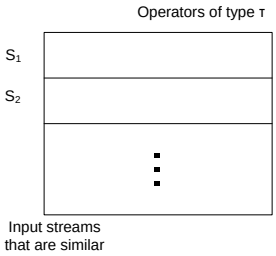



**Figure 2: $\mathcal{R}_1$ Applied to a Set of Operators of Type $\tau$**

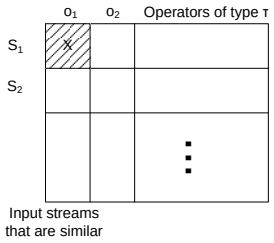



**Figure 3: $\mathcal{R}_2$ Applied to a Set of Operators of Type $\tau$**

# 4. INTEGRATING MQO TECHNIQUES FOR EVENT ENGINES

In Sections 2 and 3, we have presented RUMOR, as well as how the MQO techniques for REs can be integrated into RUMOR. In this section, we describe how the MQO techniques for EEs can be integrated as well. This is a more challenging task, as EEs are often based on automata, instead of query plans composed of relational operators. On the other hand, if we are able to integrate the MQO techniques for both REs and EEs into RUMOR, we will be able to build an expressive and scalable stream system unifying REs and EEs, of which there are obvious and significant benefits (Section 4.1). The challenge is to integrate EE MQO techniques without cluttering RUMOR with numerous new abstractions or complex special rules for event queries. Instead, the integration should result in a clean simple optimization framework where only new operators and new rules need to be added to extend its functionality.

Our solution consists of two parts. First, we translate automata into query plans (Section 4.2), second, we express the MQO techniques designed for automata in RUMOR (Section 4.3). To illustrate the benefit of integrating RE and EE MQO techniques into RUMOR, we show new channel-based MQO techniques for EEs that combine the channel concept with traditional EE MQO approaches (Section 4.4).

## 4.1 A Motivating Scenario for Unifying REs and EEs

The separation of stream processing systems into REs and EEs has led to parallel developments of MQO techniques that are tailored to these systems. Due to the lack of a common MQO framework, similar ideas have to be re-invented and optimization opportunities are missed because some techniques exist in only one world. Also, queries that require functionality from both REs and EEs are not effectively optimized by either system.

Consider the following scenario. In performance monitoring of computer systems [20, 9], each stream corresponds to readings of a particular performance counter, such as the amount of current CPU consumption of a particular thread or process. Users can register continuous queries in a stream system; e.g. to compute the average CPU load in a time-based sliding window, or to raise alerts on specified conditions and optionally to perform certain actions, such as terminating resource hogging processes. The following simple example illustrates performance monitoring workloads.

**Input streams.** We assume the following input stream schema: `CPU(pid, load; ts)`, indicating the CPU load of each process in the system. `pid` denotes process ID; `load` denotes CPU load; `ts` denotes the required timestamp attribute for each stream.[1]

[1]In practice there are more performance counters than just CPU,

**Queries.** Query workloads for system monitoring often have the following two characteristics. First, there may exist a large number of concurrent queries in the system, since different queries may be registered to monitor the behavior of different processes. Furthermore, for a particular process, different monitoring conditions may be posed in different queries. To obtain high throughput, it is crucial to apply MQO techniques to these queries. Second, some performance monitoring queries demand functionality from both CQL-style queries supported by a relational engine and pattern matching queries supported by an event engine. We refer to such queries as *hybrid queries*.

Consider the following hybrid query, which detects processes that are ramping up in CPU consumption. This query combines the functionality of sliding window aggregates (which have received a lot of attention in work on REs) for smoothing the incoming performance counter readings, and the functionality of event pattern detection (supported by an event engine) for finding a monotonically increasing sequence in CPU load consumption.

QUERY 1. *For a particular process p, smooth the CPU load value by replacing the current CPU load for p with an average load of p over the last 5 seconds. Call the smoothed stream* `SMOOTHED`*. Next, find in* `SMOOTHED` *an event pattern composed of a sequence of monotonically increasing CPU loads on p, where this sequence pattern satisfies a customizable starting condition $\theta_s$, e.g., $\theta_s = CPU.load < 20$, and a fixed stopping condition, say $CPU.load > 90$.*

To efficiently process such a query, one has to combine optimization techniques from REs (for the sliding window aggregate) and EEs (for the monotonic sequence). Having a common framework like RUMOR greatly simplifies this process, especially when it comes to the even more challenging problem of processing a large number of such hybrid queries:

QUERY 2. *We have a set of queries $\{Q_1, \cdots, Q_n\}$, where each $Q_i$ differs from Query 1 only in the starting condition $\theta_s$.*

Note that for this comparably simple example workload, it is possible to *manually* construct query plans that achieve good computation sharing. However, the focus of this paper is to automate MQO with RUMOR so that more complex workloads can be optimized as well. In the remainder of Section 4, we will revisit this query workload and describe how automated MQO is achieved in RUMOR. This is however predicated on the understanding of how the MQO techniques for EEs are integrated into RUMOR, which we discuss next.

## 4.2 Translating Automata to Query Plans

Event Engines are often based on automata [4, 11, 7, 21]. In order to integrate the MQO techniques for EEs into RUMOR, our first step is to model the automata used in EEs as query plans in RUMOR. Since the Cayuga system [7] is the EE representative with the by far strongest emphasis on MQO, we chose it as the example to show how to express automata as query plans in RUMOR. It is possible to integrate other event engines, such as SASE [21], into the RUMOR framework in a similar manner.

At a high level, the translation of automata to query plans is based on the idea that automaton states can be mapped to operators while automaton edges correspond to streams where tuples flow from one operator to the next. However, the challenges are in the details: Automaton edges have predicates and there is nondeterminism. Also, our main goal is to integrate EE functionality into RUMOR without significantly increasing RUMOR's complexity. Ideally, one should only have to add new specific m-ops and m-rules to support EE style MQO inside RUMOR. The following discussion shows how this can be achieved for Cayuga automata. Our general approach can be extended to other automaton-based systems, including the recently proposed NFA$^b$ [1] automata.

such as for memory and disk. The streams' schemas also involve more attributes. We simplify the scenario here for ease of presentation.

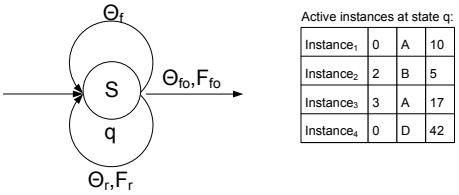



**Figure 4: Single State $q$ of Cayuga Automaton**

The basic building block of Cayuga automata are states like the one shown in Figure 4. A state has an associated input stream $S$ and it maintains a set of active automaton instances. These instances correspond to partially processed queries that have advanced to this state. Each instance has the same fixed schema; its values record relevant data from previously matched events. A state has three outgoing edge types: a single filter edge (top loop edge), one or more forward edges (horizontal outgoing edge), and a single rebind edge (bottom loop edge). States can only be connected through forward edges, resulting in automata that are directed acyclic graphs. Conceptually, whenever a new event arrives on stream $S$, the Cayuga engine checks for each instance at state $q$ if any of the edge predicates is satisfied. These predicates can reference attributes of both the incoming event a well as the instance. Instances for which no edge predicate is satisfied are deleted. All others non-deterministically traverse all edges whose predicates are satisfied. Non-determinism is implemented by duplicating the instances and letting each copy traverse the corresponding edge.

When an instance traverses the filter edge, it remains at state $q$ unchanged. When it traverses the rebind edge, the automaton executes formula $F_r$ on the concatenation of the instance and the incoming event. It then stores the modified instance at state $q$. For forward edges, the instance is also modified based on formula $F_{fo}$ and the incoming event, but it is sent to the corresponding next state. $F_r$ and $F_{fo}$ are schema map functions. A schema map function can rename and project attributes, as well as introducing new attributes via simple arithmetic computation or user-defined functions. It is similar to a SQL projection operator (which implements the SQL SELECT clause). Details can be found in [7, 8].

A complete Cayuga automaton with states $q_1$, $q_2$, and $q_3$ is shown in Figure 5(a). States $q_1$ and $q_3$ are start and final state, respectively. The start state has only forward edges, while the final state has no outgoing edge. For state $q_2$ the rebind edge is omitted, which is equivalent to having a rebind edge with $\theta_r = \text{false}$.

Ideally we would like to map all edge predicates $\theta$ to selection operators and all schema map functions $F$ to the corresponding combination of projection, renaming, and arithmetic manipulations. Unfortunately, the semantics of filter and rebind edges makes it necessary to introduce two special m-ops into RUMOR. Given an automaton state with a filter edge but no rebind edge ($\theta_r = \text{false}$), that state will be translated into an m-op denoted as `;`. Its semantics is the same as its counterpart in the Cayuga algebra [7]. Intuitively, `;` is a sequence operator concatenating two input events. Similarly,

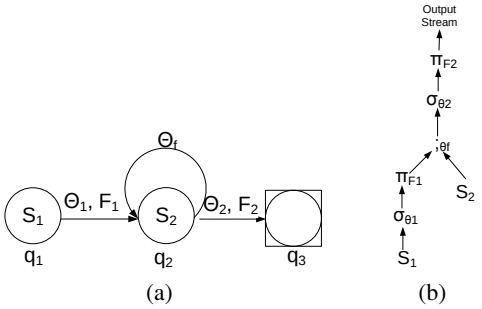



**Figure 5: (a) An Example Cayuga Automaton, and (b) the Equivalent Query Plan**

an automaton state with a filter and a rebind edge is translated into an m-op denoted as $\mu$, whose semantics is also the same as its counterpart in the Cayuga algebra. $\mu$ is an iterative version of ;, capable of concatenating an unbounded number of input events into an event sequence pattern. The formal definitions of ; and $\mu$ can be found in [7]. Forward edges, however, can be mapped to a selection operator followed by a schema mapping operator as expected.

We have developed a formal mapping from Cayuga automata to RUMOR query plans. For simplicity, we illustrate this process through an example. For the Cayuga automaton in Figure 5(a), we start with the input stream $S_1$, read by $q_1$. Predicate $\theta_1$ on the forward edge of $q_1$ is translated to $\sigma_{\theta_1}$ in the query plan. Similarly, the schema map function $F_1$ on the same edge is translated to $\pi_{F_1}$ in the query plan, reading the output stream of $\sigma_{\theta_1}$.[2] Next, we translate state $q_2$ into a binary operator $;_{\theta_f}$, reading the output stream of $\pi_{F_1}$ as well as $S_2$. Finally, the forward edge from $q_2$ to $q_3$ is translated in a similar way as the forward edge from $q_1$ to $q_2$. We use $\sigma_{\theta_2}$ and $\pi_{F_2}$ respectively to implement the predicate $\theta_2$ and the schema map function $F_2$ on that forward edge. The output stream of $\pi_{F_2}$ is equivalent to the output stream of the automaton. This finishes the translation. The resulting query plan is shown in Figure 5(b).

The translation of a Cayuga automaton involving states with rebind edges is similar. For example, if state $q_2$ in Figure 5(a) also had a rebind edge with predicate $\theta_r$, then the operator $;_{\theta_f}$ in Figure 5(b) would be replaced with $\mu_{\theta_f,\theta_r}$.

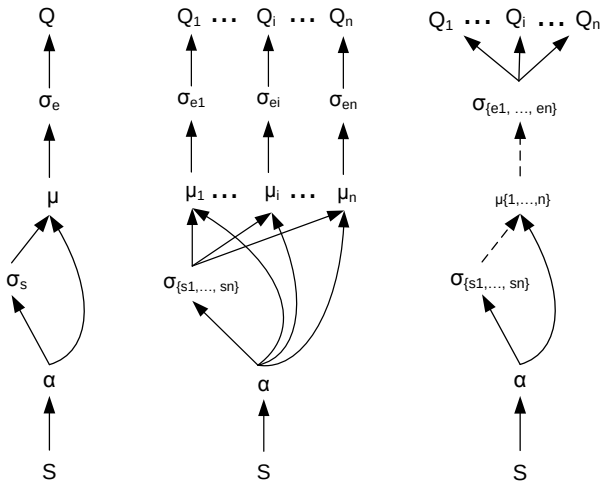
EXAMPLE 2. *The RUMOR query plan for Query 1 in Section 4.1 is shown in Figure 6(a). For clarity, we omit projection operators and the parameters of some operators in the query plan.*

*The input stream is denoted as $S$. $\alpha$ denotes the sliding window aggregate operator for smoothing the CPU load readings of each process. $\sigma_s$ and $\sigma_e$ are respectively the starting and stopping conditions. $\mu$ builds up the event sequence pattern consisting of monotonically increasing values in the CPU loads of a particular process. Finally, as in Example 1, we use the query name $Q$ to denote its output stream name.*

## 4.3 Expressing Automata Based MQO Techniques in RUMOR

In Section 4.2, we achieved the unification of an RE and an EE on a single automaton level, by translating an automaton into a query plan. This was done by adding only two operators to RUMOR.

[2]Here $\pi$ denotes the more expressive SQL projection operator (SELECT clause), as opposed to the projection operator in relational algebra.




(a) Query Plan for Query 1

(b) Query Plan for $n$ Instances of Query 2 Without Channel

(c) Query Plan for $n$ Instances of Query 2 With Channel

**Figure 6: RUMOR Query Plans for the Motivating Queries in Section 4.1 (Omitting Projection Operators for Clarity)**

To efficiently process a large number of event pattern queries, an EE often employs a set of MQO techniques specially designed for automata. To make RUMOR practical for event processing applications, we have to express these techniques as m-rules. We again use Cayuga as a representative EE which adopts MQO, and express *all* its MQO techniques by m-rules and m-ops in RUMOR. New MQO techniques for EEs could be integrated similarly.

Given that we have introduced two new operators ; and $\mu$ into RUMOR in Section 4.2, next we add a new m-rule for each of these two operators. The m-rule $s_;$ for ; (resp. $s_\mu$ for $\mu$) maps a set of ; operators (resp. $\mu$ operators) to an m-op, if they read the same pairs of streams, and have the same definition. These two m-rules are shown in the second-to-last row in Table 1.

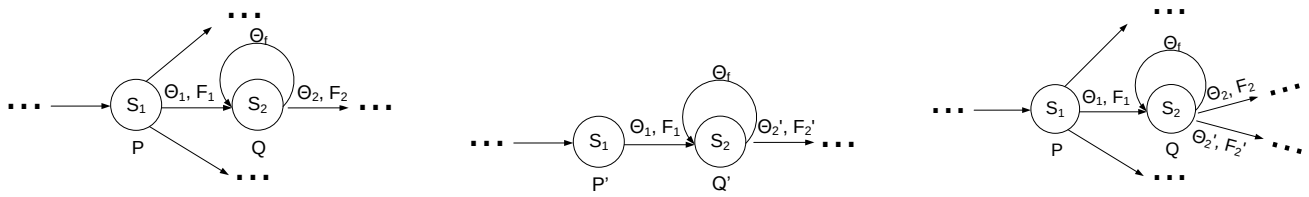
There are two major categories of MQO techniques in Cayuga, *state merging* and *indexing*. We show how these techniques can be expressed by m-rules.

**State merging.** The first type of state merging in Cayuga is prefix state merging. Intuitively, given an existing automaton $\mathcal{F}$, and a new input automaton $A$, $A$ can be merged into $\mathcal{F}$ by identifying the longest prefixes of $\mathcal{F}$ and $A$ that are identical, and share the two prefixes in the merged automaton.

As a concrete example, the existing automaton and the input automaton to merge are shown respectively in Figure 7(a) and 7(b). In this example, suppose inductively that state $P$ and $P'$ have been merged, and state $Q$ and $Q'$ read the same stream (in this case it is $S_2$), then we can merge states $Q$ and $Q'$. The resulting automaton is shown in Figure 7(c).

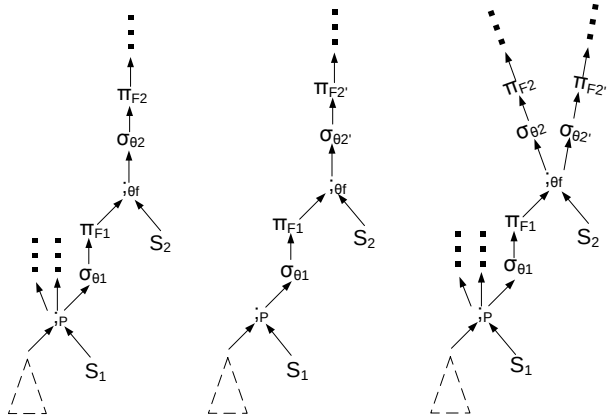
This prefix state merging technique is expressed by the m-rules $s_;$ and $s_\mu$ together. We illustrate this with the above example. The query plans corresponding to the existing automaton and the input automaton are shown respectively in Figure 8(a) and 8(b). Note that the operators $;_{\theta_f}$ in Figure 8(a) and in Figure 8(b) respectively implement states $Q$ and $Q'$ in the corresponding automata.

Suppose inductively that the common sub-expressions below operator $;_{\theta_f}$ in Figure 8(a) and in Figure 8(b) have been merged. The m-rule $s_;$ corresponding to ; is now applicable to $;_{\theta_f}$ in both figures,




(a) Current NFA Forest before Merge (b) The Query Automaton to Merge (c) Current NFA Forest after Merge

**Figure 7: Cayuga Automata State Merging Process**




(a) Current Query Plan before Merge (b) The Input Query Plan to Merge (c) Current Query Plan after Merge

**Figure 8: RUMOR Query Plans for Cayuga Automata**

since by assumption they read the same pair of streams, and have the same definition. Hence, they are mapped by the m-rule $s_;$ to the same m-op, which we still denote as $;_{\theta_f}$, since it has the same definition as the two input operators that are merged. The resulting query plan is shown in Figure 8(c). The prefix state merging performed on multiple $\mu$ operators can be done in a similar way.

Note that we have translated the prefix state merging MQO technique on automata into the well-known MQO technique on query plans: Common Subexpression Elimination (CSE). This is a good example for how a common optimization framework can help avoid duplicate work (in this case the development of "new" specialized automaton state merging techniques).

In addition, in RUMOR, we have more opportunities for inlining, illustrated as follows. An input Cayuga query that is not left-associative, such as $S_1;(S_2;S_3)$, has to be implemented by two Cayuga automata $A$ and $B$, where $A$ implements $S_2;S_3$, producing an intermediate stream $S'$, and $B$ implements $S_1;S'$. This is referred to as resubscription in [7], and in this case $A$ cannot be inlined into $B$. However, this query can be implemented by a *single* query plan, which effectively inlines the query plan corresponding to $A$ to that corresponding to $B$, providing additional MQO opportunities.

**Automaton indexing.** There are three types of indices in Cayuga. Below we describe how to express the Forward-Rebind (FR) Index technique in RUMOR. The other two indices, Active Node Index and Active Instance Index [7, 8], are handled similarly. We omit their description due to space constraints.

FR Index is a per-state index on some of the predicates of forward/rebind edges of its associated state. For example, in Figure 7(c), the predicates $\theta_2$ and $\theta_2'$ associated with the forward edges going out of state $Q$ can be managed by an FR Index. For an incoming event $e$ from stream $S_2$ which does not satisfy the filter edge predicate $\theta_f$ associated with state $Q$, $e$ will be used to probe this FR index to obtain the set of satisfied predicates associated with the forward edges (i.e., a subset of $\{\theta_2, \theta_2'\}$).

An FR Index on state $q$ can be expressed by the m-rule $s_\sigma$ in RUMOR as follows. Let the operator corresponding to $q$ in the translated RUMOR query plan be $o$ ($o$ is of type $;$ or $\mu$). For those selection operators that are consumers of the output stream of $o$, we apply the m-rule $s_\sigma$ to map them to the same m-op. That m-op effectively implements the FR Index for the translated query plan.

For example, recall that for the automaton shown in Figure 7(c), its corresponding query plan in RUMOR is shown in Figure 8(c). For the FR Index on the forward edges of state $Q$ in Figure 7(c), which we described above, it can be expressed by applying $s_\sigma$ to $\sigma_{\theta_2}$ and $\sigma_{\theta_2'}$ above the $;_{\theta_f}$ operator in Figure 8(c).

To conclude, we have shown that with RUMOR, all the MQO techniques employed by Cayuga can be expressed elegantly as m-rules on RUMOR query plans. Hence, asymptotically, the evaluation efficiency of a set of event pattern queries in RUMOR is at least as good as that in the Cayuga engine, as is confirmed by our experiments.

EXAMPLE 3. *The RUMOR query plan for $n$ query instances of Query 2 in Section 4.1, denoted as $Q_1$ through $Q_n$, is shown in Figure 6(b). The aggregation operator $\alpha$ is shared by all $n$ queries. It produces a single stream called* SMOOTHED *in Query 1, and multiplexes it to all its consumer operators. The $n$ starting conditions are implemented by the m-op $\sigma_{\{s_1,\cdots,s_n\}}$, which produces $n$ output streams corresponding to that of $\sigma_{s_1}$ through $\sigma_{s_n}$. $\mu_i$ builds the event sequence pattern for query $Q_i$. It reads the two streams produced by $\sigma_{s_i}$ and $\alpha$ respectively. Its output stream is consumed by $\sigma_{e_i}$, the stopping condition for $Q_i$.*

Note that in Example 3, even though the $\mu_i$ operators have the same definition, they cannot share work, since their left input streams are different. The same observation holds for the $\sigma_{e_i}$ operators. This is a limitation for RUMOR without channels, and is also the case for Cayuga automata. We will show in the next subsection how to use channels to overcome this limitation.

## 4.4 Query Plans with Channels

In the previous subsection, we have discussed how to express all Cayuga MQO techniques as m-rules and m-ops in RUMOR. In this subsection, we demonstrate one of the major benefits of integrating MQO techniques from RE and EE into a single framework. More precisely, we show that, somewhat surprisingly, there are event pattern queries that can be evaluated more efficiently in the form of

RUMOR query plans than in the Cayuga engine. This is due to new MQO opportunities with channels, illustrated through the following example.

EXAMPLE 4. *Let us revisit Query 2 from Section 4.1, and consider how to process $n$ instances of this query more efficiently than the query plan shown in Figure 6(b). The sliding window aggregation part of these queries for smoothing the input stream is already shared. For the pattern matching part, a good evaluation strategy is to first evaluate the starting conditions in the $n$ queries. If any subset of them is satisfied, we remember this information and continue to match the monotonic sequence patterns of these queries, implemented by the $\mu$ operators. When the stopping condition is satisfied, we then use the information we remembered for which $\theta_{s_i}$'s are satisfied to produce result tuples for the right set of queries.*

*The RUMOR query plan implementing this evaluation strategy is shown in Figure 6(c). As in Figure 1(c), we use dashed arrows to represent channels. However, note that this evaluation strategy is outside of the Cayuga automata model, and therefore cannot be used by the Cayuga engine.*

In order to produce the desired query plan with channels shown in Figure 6(c), we add one m-rule for ; and $\mu$ each. The m-rule for ;, denoted as $c_;$, maps a set of ; operators to a single m-op, if these operators satisfy (a) they have the same definition, (b) they read sharable input streams for the first input stream parameter, where these input streams are produced by the same m-op, and (c) they read the same input stream for the second input stream parameter. In this case, we encode the first input streams of these operators with a channel. The new m-rule for $\mu$ works in a similar way. These two m-rules are shown in the last row in Table 1. The stream shareability computation and the channel-based MQO sharing criteria defined in Section 3 are extended accordingly for ; and $\mu$.

We now show how to use the m-rules to optimize $n$ instances of Query 2, denoted as $Q_1$ through $Q_n$. Starting from the query plan in Figure 6(b), we first apply the m-rule $s_\sigma$ to the set of starting conditions in $Q_1, \ldots, Q_n$, and encode their output streams with a channel $C$. Next, we apply the m-rule $c_\mu$ to the set of $\mu$ operators in the $n$ queries, and again encode their output streams with a channel $D$. Finally, we apply the m-rule $c_\sigma$ to the set of stopping conditions in $Q_1, \ldots, Q_n$, resulting in a selection m-op that reads channel $D$ as input, and produces $n$ output streams for the $n$ queries.

# 5. PERFORMANCE EVALUATION

We have implemented in Java a prototype stream engine based on RUMOR, which is capable of processing RE queries, EE queries, as well as hybrid queries. In this section we report the performance of our engine in evaluating the optimized query plans. The experiments are conducted on a machine with Intel Pentium D 2.80 GHz processor and 2 GB main memory, running Sun Java Hotspot Server VM 1.6.02 on Windows Vista.

To leverage the JVM just-in-time code optimization, for each experiment, we first process the input stream for a few iterations, before we start to measure throughput. To reduce experimental variance, we perform each experiment for ten times, and report the average throughputs we measured.

## 5.1 Setup

We first use a synthetic benchmark to measure the performance of our system for processing event pattern queries and hybrid queries. We do not measure the performance for RE queries, because RUMOR is query-plan based like REs. RUMOR therefore can use the same query plans as these systems, which have been well studied [2, 5].

| Variable | Default Value |
|---|---|
| Number of queries | 1000 |
| Number of attributes in stream schemas | 10 |
| Constant domain size | 1000 |
| Window length domain size | 1000 |
| Zipfian parameter | 1.5 |

**Table 3: Parameters (default values)**

The stream schema we choose consists of 10 integer attributes, denoted as $a[0], \cdots, a[9]$, and 1 (integer) timestamp attribute. We generate two streams conforming to this schema, denoted as $S$ and $T$, as follows. The generated stream tuples have consecutive timestamps, starting from 0. For each tuple, we set its 10 integer attributes to integer values from 0 to 999 chosen uniformly at random. We interleave the generation of tuples for $S$ and $T$. That is, tuples with timestamps 0, 2, $\cdots$ belong to $S$, and tuples with timestamps 1, 3, $\cdots$ belong to $T$. For each experiment, we generate a total of at least 100000 tuples, and feed the tuples from $S$ and $T$ in their timestamp ordering.

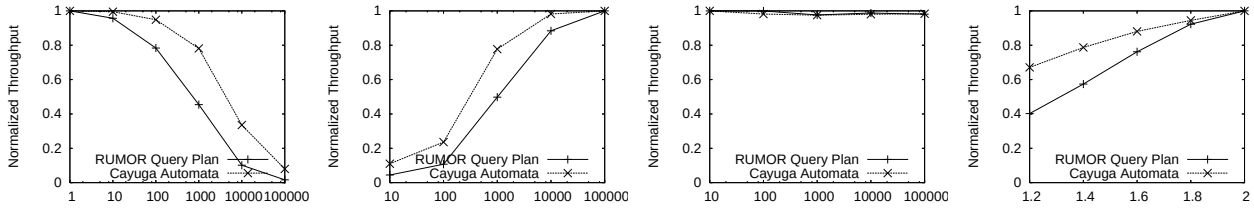
We use the following common parameters to generate query loads. For each randomly generated query, we choose a window length for it from 1 to 1000, where 1000 is the default domain size for generating window lengths. Each window length is chosen with a Zipfian distribution, favoring larger windows (i.e., a window of length 1000 is most likely to be chosen). The default Zipfian parameter value is 1.5. The Zipfian distribution is to model commonality among queries that is often observed in real, large-scale workloads. The parameters are summarized in Table 3.

## 5.2 Event Pattern Queries

In Section 4, we have chosen Cayuga as a representative event engine, and shown how to express its automata queries and MQO techniques in RUMOR. In this subsection, we compare the performance of our system based on RUMOR with Cayuga. Due to the significant differences in the architecture and implementation platform of both systems, a comparison of their absolute performance is not meaningful. Instead, we follow the experimental approach used in SASE [21], and report *normalized throughput* obtained as follows: as the query processing load changes from light to heavy in each experiment, we use the throughput for the lightest workload to normalize other measurements. This approach allows us to observe and compare the performance trends of both systems when we vary the values of experimental variables, indicating system scalability.

**Workload 1.** In the first query workload, we generate a set of queries of template $\sigma_{\theta_1}(S)$ $;_{\theta_2 \wedge \theta_3}$ $T$, where ; is the Cayuga sequence operator. $\theta_1$ is of form $a[0] = c$, where $c$ is chosen at random between 0 and 999 with the same Zipfian distribution as for window lengths. Similarly, $\theta_3$ is of the same form and generated in the same way, but it is evaluated on each $T$ tuple, whereas $\theta_1$ is evaluated on each $S$ tuple. $\theta_2$ is a "duration predicate" in Cayuga terminology — it expresses the window length of this query. Note that this query workload benefits from the AN index and FR index in Cayuga, which we described in Section 4.3. In particular, the $\theta_1$'s of the set of queries we generate can be indexed by an FR index, while the $\theta_3$'s can be indexed by an AN index.

We first vary the number of queries. Figure 9(a) shows that by expressing AN indexes and FR indexes with m-rules in RUMOR, our system scales very well. Note that even if the predicates $\theta_1$ and $\theta_3$ on each query are quite selective, this is not a trivial query workload — with 100K queries in the system, the input stream of




(a) Number of Queries (b) Constant Domain Size (c) Window Length Domain Size (d) Zipf Parameter

**Figure 9: Event Pattern Query Workload Exercising AN Index and FR Index in Cayuga**

100K events generates a total of 1.7 million output events.

Next, we vary the domain size from which query constants are drawn. Intuitively, the larger this domain size, the more selective $\theta_1$ and $\theta_3$ are, and therefore the lower load each query generates. The result in Figure 9(b) matches our expectation.

We then vary the size of the domain from which sliding window lengths are drawn. For a sliding window join query, the larger the window length, the more expensive the query becomes, since it needs to hold more state and may produce more output tuples. However, the Cayuga sequence operator has the special semantics that when a tuple in the operator state is matched by an incoming tuple from its second input stream, that tuple in the state is deleted. For this query workload, most tuples in the operator states are matched by incoming tuples, before they fall out of the query windows and expire. Therefore, as we increase the size of window length domain, creating queries with larger windows, the load of the queries does not significantly increase. This is confirmed by the result shown in Figure 9(c).

Finally, we vary the Zipfian parameter value used to generate window lengths and predicate constants, and report the result in Figure 9(d). As we increase the Zipfian value, there is increased commonality among the generated queries. When two queries share a common subexpression, both systems will avoid repeated evaluation of the common subexpression. As a result, the throughputs of both systems increase. However, the impact of Zipfian value is not very significant for this query workload (the throughputs of both systems increase by a factor of around 2 when the Zipfian value increases from 1.2 to 2). This is because with the use of an AN index and an FR index, the added value of Common Subexpression Elimination in terms of increasing throughput is little. This is consistent with the conclusion drawn in previous work [7].

**Workload 2.** We next use a query workload which benefits from the AI index technique in Cayuga, an MQO technique which can again be expressed in RUMOR.

The query template we use is $S ;_{\theta_1 \wedge \theta_2} T$, where $\theta_1$ is of form $S.a[0] = T.a[0]$, and $\theta_2$ expresses the window length of this query, as in the previous workload. The AI index in Cayuga indexes for each query the input tuples from $S$ that remain in the state of `;`. This is so that the evaluation of $\theta_1$ on the input $T$ tuples can be sped up.

The result of varying the number of queries is shown in Figure 10(a). This query workload is more expensive than the previous one, since each query here does not specify predicates that compare $S$ and $T$ tuples with constants. Intuitively, a query in this workload, which looks for a pair of $S, T$ tuples with the same value on $a[0]$, is a parametrized version of a query in the previous workload, which looks for a pair of $S, T$ tuples with $a[0]$ values specified by the predicate constants. Hence, when we process each query in this workload, each input $S$ tuple is inserted into the `;` operator state, and each input $T$ tuple probes the operator state. Still, our system is able to maintain high throughput in the presence of 10000 queries.

We also tried a variant of the query template used in this workload: $S \; \mu_{\theta_1 \wedge \theta_2, \theta_3} \; T$. Here $\mu$ is the Cayuga iteration operator. $\theta_3$ is the "rebind predicate" which is defined by $T.a[1] > last.a[1]$, where $last.a[1]$ denotes the $a[1]$ value of the last input event that contributes the event pattern being built by this query. Intuitively, each such query looks for an event sequence pattern starting with an $S$ tuple, followed by a sequence of $T$ tuples with increasing $a[1]$ values. The result of varying the number of queries of this template is shown in Figure 10(b). The throughput trends of both systems are similar to those in 10(a). However, the absolute values are lower, since $\mu$ is a more expensive operator to evaluate than `;`.

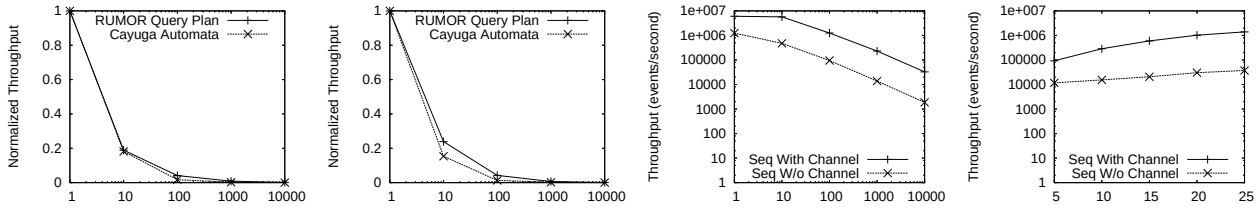
Similarly to the experiments for the previous workload, we also varied other parameters for this workload, and obtained similar results. We omit them due to space constraints.

**Conclusions of Workload 1 and 2.** As we observe in the above results, both systems display similar trends of throughputs when we vary the experimental parameters in the above settings. This is expected since the Cayuga MQO techniques are translated into RUMOR, and are used by the RUMOR query plans.

**Workload 3.** The above experiments do not involve channels. Next, we use channels to further share work among queries generated in a variant of Workload 2, with the techniques described in Section 4.4. In this workload variant, the query template we use is $S_i ;_{\theta_1 \wedge \theta_2} T$, where the predicate parameter of `;` is defined in the same way as above workload. The first input stream $S_i$'s for the generated queries are different streams but *sharable* as defined in Section 3.2, and they are encoded with the same channel, denoted as $C$. The second input stream $T$ is the same for all the queries. By default, there are 10 different $S_i$ streams, referred to as $S_1$ through $S_{10}$. For a channel, we define the number of streams encoded by it its *channel capacity*. Thus the default channel capacity of $C$ is 10.

As we have compared the performance of our system with Cayuga in the above workloads, to quantify the benefits of using channels, here we compare the performance of our system with channel against that of our system without using channel. Since both competitors are based on the same software infrastructure, we report absolute throughput in the results below.

We modify the way we generate stream tuples as follows. When we use channels in the query plan, we interleave the generation of tuples from $C$, encoding $S_1$ through $S_{10}$, and the generation of tuples from the channel encoding $T$ alone. Each $C$ tuple belongs to all $S_i$'s. In the case when we do not use channels in the query plan, we use a round robin policy to generate stream tuples: we first generate 10 tuples respectively from $S_1$ through $S_{10}$, and then generate a tuple from $T$. The set of 11 consecutive tuples is referred to as a *round* of tuples. We then repeatedly generate rounds of tuples. To ensure fairness in the comparison, We make the first 10 tuples in every round have the same content. This way, the generated stream




(a) Number of Sequence Queries (b) Number of $\mu$ Queries (c) Number of Queries (d) Channel Capacity

**Figure 10: Event Pattern Query Workload Exercising AI Index in Cayuga**

used for the query plan with channels, and the one used for the query plan without channels, have exactly the same content.

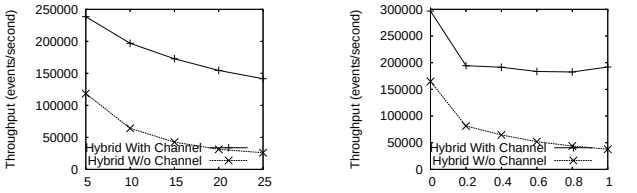
Figure 10(c) reports the result of varying the number of queries. The throughput of our system using channel is one order of magnitude higher than that of our system not using channel. Note however that this workload generation is optimistic, in that it assumes each $C$ tuple belongs to all $S_i$'s. In a realistic workload, it is usually the case that a channel tuple belongs to a subset of streams the channel encodes, which we measure in Section 5.3. Nevertheless, Figure 10(c) shows that channel is a very good mechanism for sharing work. Figure 10(d) reports the result of varying channel capacity. Clearly, when our system uses channels, the more streams channel $C$ encodes, the higher throughput it achieves. We also performed experiments on channels with query template $S_i\mu_{\theta_1 \wedge \theta_2, \theta_3}T$, and obtained similar results.

## 5.3 Hybrid Queries with Real Datasets

The experimental results on our synthetic benchmark showed that our system is efficient and scalable. We next test our system with hybrid query workloads and two real datasets.

The two performance counter datasets are both collected with the Performance Monitor component of Windows Vista. For the first dataset, called $D_1$, we chose 104 long running processes on a developer's office machine, and collected the CPU usage of these processes over a 24-hour period of time. For each process, for every second, one stream tuple is recorded for the amount of CPU percentage that process has used in the last second. The second dataset, called $D_2$, is collected in a similar way, recording the CPU usage of 28 long running processes on a home machine over a day.

For the query workload of this experiment, we choose the set of hybrid query instances of Query 2 with the following modifications to make our query workload more challenging. First, instead of monitoring a particular process with a specified pid in the query, each query monitors *all* processes. Thus, if the input stream contains performance readings for $n$ processes, where $n = 104$ for dataset $D_1$, one query in our workload corresponds to $n$ instances of Query 2, each monitoring a particular process out of the $n$ processes. Second, for each query, we increase the window length of the sliding window aggregation operator used to smooth the CPU loads of each process from 5 seconds (as in Query 1) to 60 seconds. We still denote the smoothed stream as `SMOOTHED`. Third, we reduce the selectivity of the stopping condition of each query, by setting the stopping condition to $CPU.load > 10$, instead of $CPU.load > 90$ (as in Query 1). As a result, each query in our workload may produce more output tuples than an instance of Query 2. Finally, for the starting condition of each query, we use a parameter $sel \in [0.0, 1.0]$ to control its selectivity. Intuitively, if $sel = 0.0$ for all queries, no stream tuple from `SMOOTHED` will pass the starting conditions, so no event patterns will be produced. With a higher $sel$ value, each `SMOOTHED` tuple may pass a subset




(a) Number of Hybrid Queries (b) Selectivity of Starting Conditions

**Figure 11: Hybrid Query Workload with Real Dataset $D_1$: Note each query corresponds to 104 instances of Query 2**

of starting conditions of all the queries. We assume these starting conditions are not indexable (since they may have inequality or more complex predicates as in Query 1) , but still use the m-rule $s_\sigma$ to map all of them to an m-op, denoted as $\sigma_{\{s_1,\cdots,s_n\}}$ as in Figure 6(b), which sequentially evaluates them for each input `SMOOTHED` tuple. Without using channels, $\sigma_{\{s_1,\cdots,s_n\}}$ produces one tuple for *each* $\sigma_{s_i}$ that is satisfied by the current `SMOOTHED` tuple being processed. With channel, however, $\sigma_{\{s_1,\cdots,s_n\}}$ produces one (channel) tuple for *all* $\sigma_{s_i}$'s that are satisfied, as in Figure 6(c).

We report the throughputs of the RUMOR query plan without using channels (Figure 6(b)), and the query plan using channels (Figure 6(c)). In Figure 11(a), we fix $sel = 0.5$, and vary $n$, the number of hybrid queries. We are able to achieve very high throughputs in this very challenging hybrid query workload, especially when we use channels in the query plan. This shows that RUMOR is very effective in sharing work among a set of queries. In Figure 11(b), we fix $n = 10$, and vary the selectivity of the starting conditions in the 10 queries. With a higher selectivity, the query plan without using channel experiences a significant degradation in throughput, as more tuples are produced by $\sigma_{s_i}$'s, so $\mu_i$'s have to do more work. As we expected, after the throughput of the query plan using channels experiences a drop when $sel$ increases from 0.0 to 0.2, that throughput remains stable with larger $sel$ values. This is because for each channel tuple $t$ produced by $\sigma_{\{s_1,\cdots,s_n\}}$, the amount of work for processing it in $\mu_{\{1,\cdots,n\}}$ remains the same, regardless of how many stream tuples $t$ encodes. As such, the more streams that can be encoded by channels in the query plan, the more savings we can obtain compared to the query plan without using channels.

We obtain similar results in processing $D_2$, and omit them due to space constraints.

# 6. RELATED WORK

Stream processing has been well studied as a computational paradigm to continuously process and respond to high-speed data streams [6, 3, 2, 5]. The importance of Multi-Query Optimiza-

tion, first studied in the context of relational database query processing [19], is recognized in NiagaraCQ [6]. Subsequent work on stream MQO focuses on individual query operator types, which we summarize below. We have shown in this paper how to model all of these MQO techniques as m-rules and m-ops in RUMOR.

The predicate indexing technique is studied in Le Subscribe [10] and CACQ [16]. In [12], Hammad et al. develop techniques to share work among multiple stream join operators which read the same input streams and have the same join predicate but potentially different window specifications. For multiple aggregate queries with potentially different groupby specifications, Zhang et al. propose to maintain query states in two levels of granularity, such that the aggregation computation performed at the finer-grained level can be shared among queries [22]. For pattern matching queries, state merging and indexing techniques, which we reviewed in Section 4.3, are proposed in Cayuga to efficiently evaluate a set of automata queries [7]. Similar indexing techniques, such as Partitioned Active Instance Stack (PAIS), are used in SASE [21].

All of the above MQO techniques attempt to share work among queries reading the *same* input stream(s). Krishnamurthy et al. propose techniques to share work among queries reading *different* streams, where these streams may share tuples of identical content [14, 15]. We generalize their techniques with the concept of channel (Section 3), and show how channels can be used to share work among multiple event pattern queries (Section 4.4).

Recent work also includes a new automaton model for event processing and techniques for sharing state within a single automaton [1]. Jiang et al. propose a three stage integration model for event and stream processing [13]. Their work takes expressive event processing features other than event pattern queries, such as event consumption modes, into account, which our work has yet to consider. In contrast, our focus is on integrating the MQO techniques for both REs and event engines.

# 7. CONCLUSIONS AND FUTURE WORK

In this paper, we propose RUMOR, a rule-based MQO framework to express and evaluate query plans that share work among multiple stream queries. RUMOR integrates existing and new MQO techniques for both REs and EEs. As a result, we are able to unify REs and EEs, and efficiently process a large number of RE queries, EE queries, and hybrid queries in a single engine.

This work opens up several avenues for future research. First, more work is needed to decide which streams to combine into channels. Second, as is mentioned in Section 3.3, on an input query plan, multiple m-rules can become applicable at the same time. Some conflict resolution strategies will be useful in this case. For example, to reduce or completely eliminate nondeterministic rule applications, rule priorities can be assigned to establish a partial order or a total order in the m-rule set. Furthermore, static analysis techniques can be developed to reason about the confluence of the rule-based query rewrite system. Finally, as in relational query optimization, it is valuable to supplement the rule-based query optimizer with a cost model, such that the optimizer can drive the rule applications based on a cost function, reducing the chance of producing a globally suboptimal query plan.

# 8. REFERENCES


[1] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *Proc. SIGMOD*, pages 147–160, 2008.

[2] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. Technical report, Stanford University, 2003.

[3] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams — a new class of data management applications. In *Proc. VLDB*, 2002.

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

[5] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. CIDR*, 2003.

[6] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proc. SIGMOD*, pages 379–390, 2000.

[7] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards expressive publish/subscribe systems. In *Proc. EDBT*, 2006.

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

[9] M. T. Edmead and P. Hinsberg. *Windows NT Performance Monitoring, Benchmarking and Tuning*. Pearson Education, 1998.

[10] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe. In *Proc. SIGMOD*, pages 115–126, 2001.

[11] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model and implementation. In *Proc. VLDB*, pages 327–338, 1992.

[12] M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. In *Proc. VLDB*, pages 297–308, 2003.

[13] Q. Jiang, R. Adaikkalavan, and S. Chakravarthy. Towards an integrated model for event and stream processing. Technical Report CSE-2004-10, University of Texas at Arlington, 2004. http://www.cse.uta.edu/research/publications/.

[14] S. Krishnamurthy, M. J. Franklin, J. M. Hellerstein, and G. Jacobson. The case for precision sharing. In *Proc. VLDB*, pages 972–986, 2004.

[15] S. Krishnamurthy, C. Wu, and M. Franklin. On-the-fly sharing for streamed aggregation. In *Proc. SIGMOD*, 2006.

[16] S. R. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. SIGMOD*, 2002.

[17] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in starburst. In *Proc. SIGMOD*, pages 39–48, 1992.

[18] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In Philip A. Bernstein, editor, *Proc. SIGMOD*, pages 23–34, 1979.

[19] T. K. Sellis. Multiple-query optimization. *ACM TODS*, 13(1):23–52, 1988.

[20] D. Shasha. *Database Tuning: A Principled Approach*. Prentice Hall, 1992.

[21] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proc. SIGMOD*, 2006.

[22] R. Zhang, N. Koudas, B. C. Ooi, and D. Srivastava. Multiple aggregations over data streams. In *Proc. SIGMOD*, 2005.