T-ReX: Optimizing Pattern Search on Time Series (Extended Version†)

Silu Huang*
Microsoft Research
Redmond, WA, U.S.A.
silu.huang@microsoft.com

Erkang Zhu*
Microsoft Research
Redmond, WA, U.S.A.
ekzhu@microsoft.com

Surajit Chaudhuri
Microsoft Research
Redmond, WA, U.S.A.
surajitc@microsoft.com

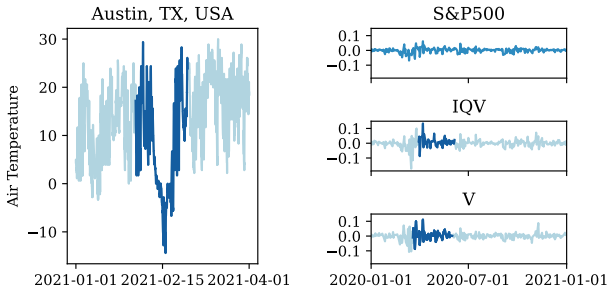
Leonhard Spiegelberg**
Brown University
Providence, RI, U.S.A.
leonhard@brown.edu

ABSTRACT

Pattern search is an important class of queries for time series data. Time series patterns often match variable-length segments with a large search space, thereby posing a significant performance challenge. The existing pattern search systems, for example, SQL query engines supporting `MATCH_RECOGNIZE`, are ineffective in pruning the large search space of variable-length segments. In many cases, the issue is due to the use of a restrictive query language modeled on time series points and a computational model that limits search space pruning. We built T-ReX to address this problem using two main building blocks: first, a `MATCH_RECOGNIZE` language extension that exposes the notion of segment variable and adds new operators, lending itself to better optimization; second, an executor capable of pruning the search space of matches and minimizing total query time using an optimizer. We conducted experiments using 5 real-world datasets and 11 query templates, including those from existing works. T-ReX outperformed an optimized NFA-based pattern search executor by 6× in median query time and an optimized tree-based executor by 19×.

1 INTRODUCTION

Variable-length time series patterns are ubiquitous in scientific research and business decision making [24, 40, 45], but searching for these patterns over large historical data can be computationally challenging. For example, a climate scientist interested in finding all past occurrences of cold waves that reversed a seasonal trend may look for a short segment of steeply decreasing temperature within a longer segment of multi-week meandering warm-up [3] from historical temperature time series. Figure 1a shows an occurrence of cold wave in Austin, Texas in February 2021. The steep drop segment can be matched using a linear regression's goodness-of-fit measure (R^2), and the warm-up segment can be matched using Mann-Kendall test for monotone trends [51]. Unfortunately, due to the variable-length nature of these segments, simply testing all possible segments has a quadratic complexity with respect to the series length. Another example is illustrated by Figure 1b, which shows a variable-length period of high correlation between each stock's daily returns and those of the S&P 500 Index. A stock analyst may search for stocks highly correlated to the index when creating an investment portfolio, but the duration of high correlation is not known ahead of time. Simply testing all segments for potential matches leads to high latency and frustration for the user.



(a) Cold Wave Pattern **(b) Stock Correlation Pattern**

Figure 1: Examples of time series patterns.

To perform pattern search over historical time series using SQL, the standard solution is `MATCH_RECOGNIZE`, which provides a declarative and composable query interface as part of the 2016 SQL standardized syntax [9]. `MATCH_RECOGNIZE` is supported in several query engines including Oracle [13], Apache Flink SQL [14], Azure Stream Analytics [15], Snowflake [12], and Trino [11]. It has been used to perform web session analysis [19] and financial audit [18].

Example 1. Figure 2 lists a query for finding the cold wave pattern in a dataset of historical temperature observations over many weather stations. The `PARTITION BY` and `ORDER BY` clauses instruct the system to construct one time series per weather station, each ordered by the timestamp attribute `tstamp`. The `PATTERN` clause specifies a regular expression that consists of *variables*, each matching a record of temperature observation. The `DEFINE` clauses provide Boolean matching conditions that define those variables. If a variables is not defined in the `DEFINE` clause, e.g., `A` and `B`, then this variable can match any record. The regular expression declares that the query pattern contains a sub-pattern of a steep decreasing segment `D+`, with "`+`" indicating a repeated occurrence of at least one `D` record using Kleene Closure. `A*` and `B*` represent two other segments with zero or more records using Kleene Star; together with `D+` they form a complex segment `U` with an upward trend over a larger time window. The steep decreasing trend is defined by, first, the difference in temperature, and second, the linear regression's goodness-of-fit measure, R^2, as an aggregate over timestamp and temperature.

† Conference proceeding accepted at SIGMOD 2023.
∗ Equal Contributors.
∗∗ Work done at Microsoft.

```
-- Record Schema in table Weather: [tstamp, station, temp]
SELECT * FROM Weather MATCH_RECOGNIZE(
    PARTITION BY station ORDER BY tstamp
    PATTERN (A* D+ B* Z)
    SUBSET U = (A, D, B)
    DEFINE  D  AS tstamp - first(D.tstamp) <= INTERVAL '5' DAY,
            Z  AS last(U.tstamp) - first(U.tstamp) BETWEEN
                    INTERVAL '25' DAY AND INTERVAL '30' DAY
                AND mann_kendall_test(U.temp) >= 3.0
                AND linear_regression_r2(D.tstamp, D.temp) >= 0.95
                AND last(D.temp) - first(D.temp) < -20)
```

Figure 2: A `MATCH_RECOGNIZE` query for finding cold wave patterns in historical temperature data.

The upward trend on `U` is defined by a Mann-Kendall monotone trend test aggregate. The extra variable `Z` is needed to evaluate the conditions on `D+` and `U` under the "final semantics"[1].

Poor Performance in Current Systems. Though pattern search queries with complex matching conditions on variable-length segments are common [24, 40, 45], their performance in existing systems is poor for queries such as the one shown in Example 1. They either use a Non-deterministic Finite Automaton (NFA) [28, 30] or a tree-based executor [34, 35, 41]. For an NFA-based executor, the cold wave pattern query took 85 minutes[2] over 36 time series with an average length of 1,854 on a Trino server with 16 parallel threads; this query timed out after 2 hours on Apache Flink SQL. For a tree-based executor, ZStream [41] took 16 minutes.

The poor performance has two causes. First, these executors are ineffective in pruning the large search space of possible matches in the pattern. Given the temporal constraints, the search space contains as many as 1.4M possible matches[3] for the query in Example 1. However, both the NFA-based and tree-based executor do not prune this search space at all, since non-temporal conditions are all defined on `Z` and `Z` can only be evaluated after instantiating `{A, D, B}`. As a result, all 1.4M partial matches are generated before being evaluated against the conditions on `Z`. Second, redundant computation is performed when evaluating these 1.4M partial matches against the conditions on `Z`, since these partial matches correspond to overlapping segments and consequently aggregates such as `linear_regression_r2()` are evaluated on overlapping segments.

Our Proposal. We built T-ReX, a time series pattern search engine with a novel executor and optimizer to address this performance issue. For the cold wave pattern, T-ReX found all results in 6 seconds using a single thread on the same machine – an 850× speedup over Trino and 160× speedup over ZStream.

[1]If a condition, say `linear_regression_r2(D.tstamp, D.temp) >= 0.95`, was defined on the variable `D` instead, then a sequence of records that satisfies the condition would not have been matched unless all prefixes of the sequence matched the condition, according to the "running semantics" of `MATCH_RECOGNIZE`. Thus, conditions such as those involving `D+` must be evaluated after the corresponding sequence has been matched to avoid discarding matches prematurely. For details see the SQL standard [9], Trino's [11] and Oracle's [13] documentation.

[2]We removed the Mann-Kendall test condition for Trino which currently does not support this aggregate; we also propagate the upper bound (i.e., 30 days) of the window condition to `A` and `B` to avoid wasting computation beyond the current window.

[3]Assuming 1 point per day, count possible matches for the overall window (25 to 30 days) and possible matches for the steep-decrease window (1 to 5 days) given each overall window size: $\sum_{w=25}^{30}(1854 - w)\left(\sum_{v=1}^{5}(w - v + 1)\right) = 1,397,185$.

By reasoning about patterns in terms of *segments*, i.e., contiguous sequences of points, T-ReX is able to utilize a new set of performance optimizations. First, T-ReX exploits selective and inexpensive sub-patterns to prune the search space of other sub-patterns. For example, when interpreting the cold wave pattern in terms of segments, we can decompose it into three major sub-patterns: the steep-drop segment (`DIFF`), linear-decrease segment (`DOWN`), and the overall warm-up segment (`UP`). If `DIFF` is more selective and less expensive to evaluate than `DOWN`, we first evaluate `DIFF` in the full search space (i.e., all segments satisfying the temporal conditions), and then use the results as anchors to prune the search space of `DOWN`, since these segments should satisfy both `DIFF` and `DOWN` sub-patterns. Such selectivity and cost information has to be reasoned about at segment level, as the Boolean expressions in `DIFF` and `DOWN` are defined on a segment not a point. Second, reasoning about the pattern in terms of segments enables computation-sharing within a query. Instead of evaluating the Boolean expression in `DOWN` for each candidate segment, its computation can be shared and amortized across all candidate segments.

T-ReX supports the `MATCH_RECOGNIZE` syntax but also extends it by introducing *segment variable* and two new operators, And (`&`) and Not (`~`), in the regular expression syntax. With segment variables and the new operators, pattern regular expressions can be specified in terms of segments. Though T-ReX shares the same expressiveness as standard `MATCH_RECOGNIZE`, the language extension enables significant performance improvement. Furthermore, to fulfill our insight on search space pruning, we designed (1) a tree-based executor with *search space awareness*, (2) novel physical operators capable of *search space pruning*, and (3) a new optimizer that *incorporates the search space* in its cardinality and cost estimation. Empirically, on a benchmark of 5 real-world datasets and 11 query templates including those from existing works [20, 28], T-ReX outperforms an optimized NFA-based [28] by 6× and an optimized tree-based executor [41] by 19×, in median query time, with computation-sharing enabled for the baseline executors.

To outline the content of this paper: we introduce the language extension and discuss its expressiveness and complexity in Section 2; we discuss query processing in Section 3, physical operators in Section 4, and optimizer in Section 5; we present experimental evaluation in Section 6; finally in Section 7 we put T-ReX in the context of existing works.

2 LANGUAGE EXTENSION

In this section, we describe our extension to `MATCH_RECOGNIZE`.

2.1 Point and Segment Variables

A *point* is a timestamped record in a time series. We use the term "point" and "record" interchangeably. A *point variable* `X` [9] is defined using the clause `DEFINE P AS Boolean_Condition(P)`. A point p is said to *match* point variable `P` if `Boolean_Condition(p)` is True. In our extension, point variables maintain the same syntax as variables in existing `MATCH_RECOGNIZE` [9].

A *segment* is a contiguous sequence of points, denoted as $[i, j]$ where i and j are the inclusive start and end integer index positions of the time series. All points including the start and end are part of

```
-- Record Schema: [tstamp, station, temp]
PARTITION BY station
ORDER BY tstamp
PATTERN ((W (DOWN & DIFF & WAVE_WINDOW) W) & UP & OVERALL_WINDOW)
DEFINE SEGMENT W AS true, -- matches any segment
SEGMENT WAVE_WINDOW AS window(WAVE_WINDOW.tstamp, 1, 5, DAY),
SEGMENT DIFF AS last(DIFF.temp) - first(DIFF.temp) < -20,
SEGMENT DOWN AS linear_regression_r2(DOWN.tstamp, DOWN.temp) >= 0.95,
SEGMENT OVERALL_WINDOW AS window(tstamp, 25, 30, DAY),
SEGMENT UP as mann_kendall_test(temp) >= 3.0
```

Figure 3: A `MATCH_RECOGNIZE` query with T-ReX extension.

the segment. A single point is a segment where the start and end are the same, i.e., $[i, i]$.

A *segment variable* is defined using clause `DEFINE SEGMENT S AS Boolean_Condition(S)` with keyword `SEGMENT` or `SEG`. A segment s is said to *match* segment variable `S` if `Boolean_Condition(s)` is True.

For instance, Figure 3 shows a T-ReX query for the cold wave pattern discussed in Section 1. In this query, `W`, `DOWN`, `DIFF` and `UP` are all segment variables. Segment variable `W` is used to represent a time window before or after the cold wave segment. T-ReX provides built-in aggregates to help define segment variables, for example, `linear_regression_r2(x, y)` and `mann_kandall_test(x)`. It also support user-defined aggregates. Besides, the `window(x, lo, hi, unit)` function is used to define window condition[4].

2.2 Operators

A variable is an atomic unit of a pattern. An *operator* is a function that takes one or two patterns as inputs and returns a new pattern. Complex patterns can be composed from variables using operators.

We first describe how existing `MATCH_RECOGNIZE` operators work with segment and point variables. Then we introduce two new operators: And and Not.

Definition 2.1 (Concatenation). *Given two patterns* `(A)` *and* `(B)`*, a Concatenation operator on* `(A)` *and* `(B)` *is denoted as* `(A B)`.

- *If* `(A)` *and* `(B)` *contain only point variables, then a segment* $[i, j]$ *matches pattern* `(A B)` *if and only if there exists some* k *such that* $[i, k]$ *matches* `(A)` *and* $[k + 1, j]$ *matches* `(B)`*;*
- *Else, i.e., either* `(A)` *or* `(B)` *contains segment variable, then a segment* $[i, j]$ *matches pattern* `(A B)` *if and only if there exists some* k *such that* $[i, k]$ *matches* `(A)` *and* $[k, j]$ *matches* `(B)`.

When `A` and `B` contain only point variables, Concatenation falls back to what is currently in `MATCH_RECOGNIZE`. When there is a segment variable in either pattern `(A)` or `(B)`, Concatenation's semantics changes slightly: the ending point of the first segment is the starting point of the second – they share the boundary point k.

Definition 2.2 (Alternation). *An Alternation operator (`|`) unions two patterns* `(A)` *and* `(B)` *so that the pattern* `(A | B)` *is matched to a segment if at least one of* `A` *and* `B` *are matched to this segment.*

[4]There are two types of windows: time-based windows and point-based windows. For time-based window, it can be specified as `window(x, lo, hi, unit)` on column `x` with lower bound size `lo` and upper bound size `hi`. For point-based window, it can be specified as `window(lo, hi)`. There are also fixed-sized windows: `window(x, size, unit)` for time-based and `window(size)` for point-based.

Definition 2.3 (Kleene). *A Kleene, or quantifier operator (`*`, `?`, `+`, `{n}`, `{m, n}`) repeats a pattern* `(A)` *a given number of times so that the resulting pattern is matched to a segment if and only if the segment is matched by pattern* `(A)` *concatenated to itself appearing the same number of times. Specifically,* `(A*)` *means zero or more,* `(A?)` *means zero or one,* `(A+)` *means at least one,* `(A{n})` *means exactly* `n` *times, and* `(A{m,n})` *means between* `m` *and* `n` *times inclusive.*

Definition 2.2 and 2.3 are what in the original `MATCH_RECOGNIZE` where there is only point variables and they work in the same way when there exists segment variables. Next, we describe two new operators not yet supported by `MATCH_RECOGNIZE`.

Definition 2.4 (And). *An And operator (`&`) joins two patterns* `(A)` *and* `(B)`*. The pattern* `(A & B)` *matches a segment if and only if both* `(A)` *and* `(B)` *match this segment.*

For example, the sub-pattern `(DOWN & DIFF & WAVE_WINDOW)` (Figure 3) specifies that a matching segment, e.g., $[10, 14]$, must meet all the conditions defined for `DOWN`, `DIFF` and `WAVE_WINDOW`. The sub-pattern `(W (DOWN & DIFF & WAVE_WINDOW) W)` concatenates three sub-patterns to match segments such as $[5, 20]$: the segment $[10, 14]$ is "padded" by segments $[5, 10]$ and $[14, 20]$. Lastly, the sub-pattern is combined with `UP & OVERALL_WINDOW` through an And operator to specify the Mann-Kendall test and window conditions for the overall pattern.

Definition 2.5 (Not). *A Not operator (`~`) negates a pattern* `A` *so that the pattern* `(~A)` *matches a segment if and only if the segment does not match* `(A)`.

The Not operator is useful when expressing none existence of some segment. For example, in the temperature dataset, if we were to find segments of warming-up without any cold wave occurrence, we can use the pattern `((~(W (DOWN & DIFF & WAVE_WINDOW) W)) & UP & OVERALL_WINDOW)`.

2.3 Comparison with `MATCH_RECOGNIZE`

Next, we will compare T-ReX's extension with `MATCH_RECOGNIZE` by answering the following two questions: (1) does the language extension add expressiveness to `MATCH_RECOGNIZE`? (2) does the language extension lead to lower complexity of query evaluation?

2.3.1 Language Expressiveness. While T-ReX's language extension provides a syntactic sugar for expressing patterns involving conditions on segments, T-ReX shares the same expressiveness as the standard `MATCH_RECOGNIZE`. Below we provide a sketch of how to prove the expressiveness equivalence between `MATCH_RECOGNIZE` and T-ReX. See the complete proof in Appendix A.

Proposition 2.1. *The introduction of segment variable in* T-ReX*'s language does not add expressiveness to* `MATCH_RECOGNIZE`.

Proof sketch: given any segment variable `S`, we can replace `S` with `(p* z) DEFINE p As true, z as S_Boolean_Condition(p, z)`, where `p` and `z` are point variables, and `S_Boolean_Condition` is the Boolean Condition in S's definition.

Proposition 2.2. *The introduction of new operators (And and Not) in* T-ReX*'s language does not add expressiveness to* `MATCH_RECOGNIZE`.

Proof sketch: based on Proposition 2.1, we can assume a given pattern contains only point variables and no segment variables. Next, we show an invariant that any pattern can be expressed using only operators in standard MATCH_RECOGNIZE. This is proved via an induction: assume sub-pattern $\mathcal{A}$ and $\mathcal{B}$ satisfy this invariant, we prove $(\mathcal{A}$ & $\mathcal{B})$ and $(\sim\mathcal{A})$ would also satisfy this invariant by first decomposing sub-pattern $\mathcal{A}$ (and $\mathcal{B}$) into alternations of special patterns (i.e., patterns involving only concatenation operators, see Lemma A.1 in Appendix A for details) and then performing a rewrite on $(\mathcal{A}$ & $\mathcal{B})$ and $(\sim\mathcal{A})$.

Theorem 2.3. T-ReX *shares the same expressiveness as standard* MATCH_RECOGNIZE.

Proof. This follows directly from Proposition 2.1 and 2.2. □

2.3.2 Runtime complexity. T-ReX's extension to MATCH_RECOGNIZE provides a direct way to reason about patterns in terms of segments. As described in Section 1, this offers a few performance advantages. For one, placing the matching conditions at their corresponding variables, e.g., DOWN and DIFF, as opposed to placing the conditions at the last variable, e.g., Z as in the original query (Figure 2), enables selectivity reasoning for each variable and makes earlier pruning of partial matches possible. Furthermore, the explicit Boolean expression defined on each segment variable enables the reasoning of computation sharing across candidate segments (see Section 4.2 for details). Last, using operators (e.g., And) to compose finer grain variables into complex patterns (e.g., DOWN & DIFF & WAVE_WINDOW) surfaces more opportunities for the optimizer to select better physical operators and evaluation order. We will discuss the executor and query optimization in detail in the next three sections.

Such performance improvement is both data- and query-dependent. Following the runtime analysis in [52], if the pattern is without Kleene operator, both the standard MATCH_RECOGNIZE executor and T-ReX have linear complexity in n, where n is the series length; else, both have exponential complexity in n in the worst case for exponential matches in the output. However, as we will show in Section 3 and 4, T-ReX achieves significant performance gain over prior executors when the query involves a large search space and there exist low selective and inexpensive sub-patterns.

While we hope that the syntax extension makes it easier to express time series patterns, it is also possible to translate a standard MATCH_RECOGNIZE query into one using the extension, which we discuss in Appendix B.

3 QUERY PROCESSING OVERVIEW

In this section, we presents an overview of T-ReX's query engine.

Time Series Data Model. In T-ReX, a time series is constructed from the input data according to a query's PARTITION BY and ORDER BY clauses, and stored in memory. A segment can be directly accessed via its start and end index positions.

Tree-based Executor. T-ReX uses a tree-based executor. Figure 4 illustrates a logical execution plan for the cold wave pattern query from Section 2. The plan is generated by parsing the query pattern and variable definitions. As in existing tree-based executors [32, 41], T-ReX's executor follows a top-down iterator model to obtain intermediate results from sub-trees. Every leaf operator is a "Segment Generator". It produces segments matching its embedded Boolean conditions. The internal operators implement the query operators: "Concat" and "And" as shown in this example, as well as "Or" (Alternation), "Kleene", and "Not" not shown here. They assemble matching segments from sub-trees to produce new segments matching their own sub-patterns. For example, the sub-pattern (DOWN & DIFF) is implemented by the lowest sub-tree rooted at an And operator.

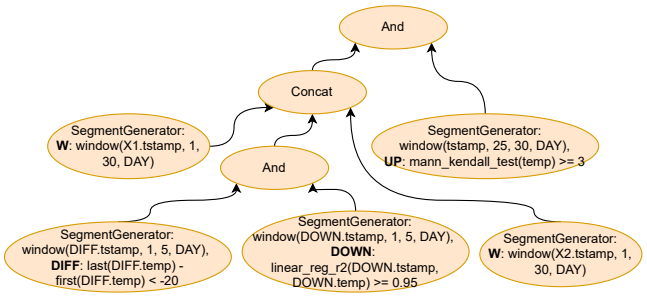


Figure 4: The logical execution plan of the cold wave pattern.

Main Insights and Contributions. The core challenge for T-ReX is to prune search space and minimize the total query processing cost. We have three main insights and make corresponding contributions to the space of pattern search executors:

(1) Sub-trees with cheaper or more selective conditions can be used to prune the search space of those with more expensive or less selective ones, e.g., use DIFF to prune DOWN. We introduce *probe operators* (Section 4.3.1 and 4.4.2) to support data-dependent pruning during execution.
(2) Since segments can overlap, it may be desirable to share the computation of aggregation functions on overlapping segments. We introduce new aggregation primitives `index()` and `lookup()` (Section 4.2.1) to support computation sharing.
(3) The optimizer must be aware of search space when estimating cost and cardinality but the actual search space of each operator is only available during execution. To overcome this, our optimizer (Section 5) uses range sizes of search space as proxies for cost estimation.

Apart from these, ideas from relational databases are also applied, namely: (1) windows as the simplest and cheapest-to-evaluate conditions should be pushed-down to the leaf nodes to prune the search space of other variables; (2) having multiple physical operator implementations and operator reordering allows the optimizer to adapt the execution plan toward query and data.

Life of a Query. The first step of query processing is logical query plan rewrite. The following plan rewrite rules are used: *(1)* Window embedding: variables with window conditions through the And operator have the windows embedded in them directly; point variables are assigned a window of size 1. *(2)* Window push-down: windows assigned to the parent patterns are pushed down to child sub-patterns, with appropriate relaxing of the length conditions for Concatenation and Kleene operators. The logical plan in Figure 4 has been rewritten by these rules: (1) WAVE_WINDOW, i.e., `window(tstamp, 1, 5, DAY)`, is embedded directly in variable DIFF and DOWN, and OVERALL_WINDOW, i.e., `window(tstamp, 25, 30, DAY)`, is embedded in variable UP; (2) OVERALL_WINDOW is pushed down to W, DIFF, and DOWN by taking OVERALL_WINDOW's upper bound, i.e., 30 days.

The second step is physical query plan optimization: logical operators with more than two sub-trees are split into binary operators; the choices of physical operators and the order of the splits are determined. For the optimizer to work, T-ReX must collect sampled statistics about selectivity of variables with respect to the input time series. The statistics are fed into a cardinality model to estimate the input and output cardinalities of all physical operators in a physical plan, and then a cost model, which is bootstrapped off-line, is used to calculate the estimated cost of the plan. T-ReX uses the dynamic programming approach similar to many relational databases to obtain the minimal-cost plan.

Finally, the plan is executed following the iterator model [32].

4 PHYSICAL OPERATORS

In this section, we describe T-ReX's physical operators, how they perform search space pruning and support language features.

4.1 Physical Operator Interface

In T-ReX each operator implements an `eval()` method producing an iterator of segments, given the input series, search space (`sp`) and references (`refs`), as shown in the Python snippet below.

```
class RightProbeConcatOperator(RightProbeOperator, Concat):
  def eval(self, series: Series, sp: SearchSpace, refs: Reference):
    # Implementation.

class Segment:
  segment: Tuple[int, int] # a tuple of start and end positions.
  payload: List[Tuple[int, int]] # segments matched to sub-patterns.
```

The `series` argument is the time series object. `sp` is the search space, it specifies the possible ranges of start and end index positions in the time series that the returning segments should follow. For example, a search space could be $(S = [5, 10], E = [10, 10])$, meaning the start position is in the range from 5 to 10 (inclusive), and the end position is fixed at 10. At the root operator, the search space is $(S = [0, n-1], E = [0, n-1])$, n being the number of points in the time series. The inclusion of search space in `eval()` method enables a native support of search space pruning in our execution framework and lays a foundation for implementing our main insight (1) in Section 3. Physical operators can achieve performance gain by reducing the search space of its child operator (Section 4.3.1 and 4.4.2).

The `refs` argument is for implementing variable references. It is a reference object that carries the referenced segments that are needed for variable with conditions on segments matched to other variables. For example, in the query shown in Figure 5, `CORRELATE`'s condition `corr(CORRELATE.x, UP.x)` references the segment matched by `UP`. During execution, the sub-tree containing the referenced sub-patterns are executed first to obtain the referenced segments; then those segments are used to evaluate the sub-tree that requires them. A segment matches a referenced sub-pattern is stored in the `payload` field of its upper-level pattern's segment, and this `payload` data is passed all the way up until it is no longer required by an `eval()` through the `refs` argument (or by the query itself[5]). Figure 5 illustrates a possible physical query plan tree, and shows how the payload field is used to store referenced segments. As we will discuss in Section 4.4.2, the `refs` argument removes the need for special handling the Not operator, such as post-processing in existing CEP systems. This reduces intermediate results, leading to better performance as shown in our experiments (Section 6.3).

[5] `MATCH_RECOGNIZE`'s `MEASURES` clause can be used to customize a match's output [9, 11, 13].

4.2 Segment Generators

In T-ReX, Segment Generators are the leaf operators of a physical query plan tree. Each Segment Generator is embedded with a variable and a window, and its `eval()` method produces segments from the time series matching the Boolean conditions of the embedded variable and window. They are designed following our main insight (2) discussed earlier in Section 3.

Because segments often overlap, for aggregates in Boolean conditions, T-ReX uses computation sharing: at query time, it first builds an "index-like" data structure over the whole series, such as the complete result materialized using dynamic programming for `mann_kendall_test`. Such computation can be amortized because after building the index, T-ReX can simply lookup the index to check if a segment is a match, instead of invoking a full aggregation. Still, the upfront cost may out-weigh the benefits of faster lookup, especially when the number of lookup is small. Thus, T-ReX provides two different implementations of Segment Generator.

4.2.1 Segment Generator with Indexing. The SegGenIndexing operator in T-ReX implements the indexing approach. Variables with conditions involving aggregates that have implemented an `index()` method are eligible to be part of an SegGenIndexing operator. For each built-in aggregate, whenever possible, we analyze and implement an `index()` method. For user-defined aggregates, we provide an interface for advanced users to implement their own `index()` methods. In SegGenIndexing's `eval()` method, this physical operator first builds an index by calling the `index()` method on the given series , and then for each segment produced by its embedded window's `iterate()`, it calls the index's `lookup()` method to check if this segment is a match and should be emitted.

Example 2. For the aggregate `linear_reg_r2(x,y)`, its `index()` method computes accumulative sums on 5 expressions: x, y, x^2, y^2 and xy, over the input series. Let $P_x(i)$ denote the accumulative sum of x from index 0 to i, similarly $P_y(i)$ for y and so on. The `lookup()` method takes input a segment (i, j) and returns the value of R^2 over this segment, using the formula: $R^2 = \frac{(\overline{xy}-\overline{x}\,\overline{y})^2}{(\overline{x^2}-\overline{x}^2)(\overline{y^2}-\overline{y}^2)}$, where $\overline{x}$ can be computed using the accumulated sum as $\frac{P_x(j)-P_x(i-1)}{j-i+1}$, similarly for $\overline{y}$, $\overline{x^2}$, $\overline{y^2}$ and $\overline{xy}$.

4.2.2 Segment Generator with Filter. The SegGenFilter operator in T-ReX implements a simpler approach: for each segment produced by its embedded window's `iterate()` method, it evaluates the embedded variable's condition to determine whether to emit the segment. By pushing embedded window down to the very bottom of processing, T-ReX avoids unnecessary evaluations of the embedded variable's conditions. For a window-only leaf operator with no embedded variable, this becomes SegGenWindow, which only generates the windowed segments.

4.3 Binary Operators

Binary physical operators (Concatenation, And, and Or) find pairs of "joinable" segments with one from the left child and the other from the right child, and create a new segment.

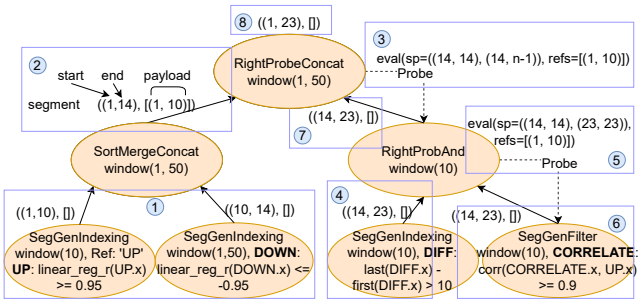
For a Concatenation, the left segment's end position must equal the right segment's start position to form a pair[6], and the new segments has the left's start and right's end. Therefore, the search space is expanded when passed to its children. For example, if the Concatenation's search space is $(S = [s_i, s_j], E = [e_i, e_j])$, then its left child's search space is expanded to $(S = [s_i, s_j], E = [s_i, e_j])$ – same S but larger E, and the right's becomes $(S = [s_i, e_j], E = [e_i, e_j])$ – same E but larger S, because the intersecting point of two joined segments can be anywhere between s_i and e_j. For a And, it requires both the left and right segments to have identical positions. For an Or, all segments from both children are accepted and emitted. Unlike Concatenation, search space of an And or an Or is passed to the children unchanged.

4.3.1 Probe Operators. The search space is either unchanged (And, Or) or expanded (Concatenation) by a binary operator. So how can we prune search space? To this point, we use our first insight: the sub-tree with cheaper or more selective conditions can be used to prune the search space of the one with more expensive or less selective ones (Section 3). This leads to the design of Left and Right Probe operators.

A Left/Right Probe operator performs pruning of one of its child's search space. It first calls `eval()` on one child, for example, the left child, using the child's search space as discussed earlier. For each segment from the left child, it calls the `eval()` method on the right child, but using a smaller search space conditioned on the segment of the left child – a "probe". For example, if the left segment is $[l_i, l_j]$, for RightProbeConcat, because the right segment starts from the end position of the left segment (l_j), the search space for the right child in this probe is reduced to $(S = [l_j, l_j], E = [e_i, e_j])$ – the size of S is reduced from $e_j - s_i$ to 1. For RightProbeAnd, given the same left segment, because the right segment must be the same as the left, the right search space for this probe is reduced to $(S = [l_i, l_i], E = [l_j, l_j])$ with a size of 1 for both S and E. An Or operator does not have a Probe operator as it simply unions the segments from both children.

For example, if we choose RightProbeAnd for both of the And operators in the execution plan in Figure 4, segments matching `DIFF` is used to prune `DOWN`, and segments matching `(W (DIFF & DOWN & WAVE_WINDOW) W)` is used to prune `UP`. Assuming 1% of `WAVE_WINDOW` windows match `DIFF`, and 1% out of those matches `DOWN`, then the total number of segments to be evaluated for `mann_kendall_test` in `UP`'s condition is about 140, instead of 1.4M using NFA (Figure 2).

Due to search space pruning, a Left/Right Probe operator can effectively reduce the total processing cost when one of the child is selective – producing few segments, and/or cheap – costing little to evaluate. However, there is potentially redundant computation at the leaf operators. When the probed child's sub-tree has an operator that expands search space (i.e., Concatenation or Kleene), redundant computation can happen at the leaf operators that receive




```
1 PATTERN (((UP & W1) DOWN
2           (DIFF & CORRELATE & W1)) & W2)
3 DEFINE SEG UP AS linear_reg_r2(UP.x) >= 0.95,
4 SEG DOWN AS linear_reg_r2(DOWN.x) <= -0.95,
5 SEG DIFF AS last(DIFF.x)-first(DIFF.x) > 10,
6 SEG CORRELATE AS corr(CORRELATE.x, UP.x) >= 0.9,
7 SEG W1 AS window(10), SEG W2 AS window(0, 50)
```

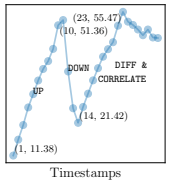


Figure 5: A time series, a query and its physical plan for finding correlated sub-patterns, `(UP)` and `(CORRELATE)`, with segment variable references defined in `CORRELATE`. Ordered blue boxes mark the flow of segments through the plan tree.

overlapping search space across probes. Thus, when neither child is selective or cheap, a Left/Right Probe operator may be costly.

4.3.2 Sort-Merge Operators. This type of operator uses the Sort-Merge join algorithm in relational systems so both children must be evaluated independently. Unlike Left and Right Probe operators, Sort-Merge operators do not have the issue of search space overlap across probes because each child is evaluated exactly once. However, it requires the children to be executed independently of one-another, that is, there can be no condition from one sub-tree referencing a operator in the other sub-tree because the second sub-tree must be executed to pass the referenced segments to the first.

While hash join algorithm has the same time complexity, in practice the Sort-Merge algorithm often early-terminates due to exhaustion of one sub-tree's segments, and gets a significant run time reduction. So T-ReX does not use hash join algorithm.

4.3.3 A Data Flow Example. Figure 5 illustrates a physical query plan with a SortMergeConcat, a RightProbeConcat, and a RightProbeAnd. We can visualize the working of these operators by following the data flow through the plan tree:

① Two segments, `[1,10]` and `[10,14]` matched variables `UP` and `DOWN` are independently generated by the two SegGenIndexing operators. Because they have no sub-tree, segments from SegGen always have empty payload. ② SortMergeConcat joins the two segments to produce a new segment `[1,14]` with a payload carrying the referenced segment matched by `UP` with reference key `'UP'`. ③ After receiving the left segment, RightProbeConcat calls the `eval()` method on the right child, setting the search space argument to `([14,14],[14,n-1])` where `n` is the series length. because the right segment must start from the end position of the left segment. The `refs` argument is set with the referenced segment obtained from the payload of the left segment, as the right sub-tree requires it to evaluate the condition `corr(CORRELATE.x,`

[6]For simplicity we only discuss segment variables. The specific join rules for points and segments is orthogonal to the implementation.

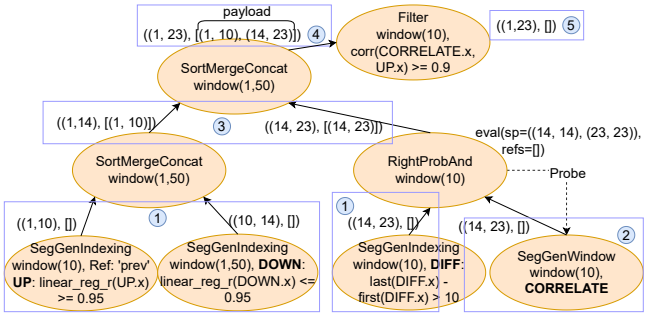


Figure 6: An equivalent physical query plan to Figure 5 with the condition `corr(CORRELATE.x, UP.x) >= 0.9` placed in a FILTER after the SORTMERGECONCAT, with a SEGGENWINDOW as the leaf operator to generate unfiltered segments.

`UP.x) >= 0.9`. ④ RIGHTPROBEAND evaluates its left sub-tree using the search space `([14,14],[14,n-1])` and receives a segment `[14,23]`. ⑤ This segment is then used to form the search space, together with the referenced segment `[1,10]`, to form the arguments to call `eval()` on the right sub-tree of the And operator. ⑥ The SEGGENFILTER confirms the segment to be a match, thus ⑦ the And operator emits this segment. ⑧ The right segment is joined with the left segment to form the final output segment `[1,23]`, which has empty payload because there is no more referenced segment consumer in the upper levels of the tree.

4.4 Unary Operators

A unary physical operator has exactly one child. It calls its child's `eval()` method to obtain segments, processing them, and emits transformed/filtered segments.

4.4.1 Filter Operators. A Filter physical operator (FILTER) has an embedded Boolean condition. When called on its `eval()` method, the FILTER calls its own child's `eval()` and uses the embedded condition to filter out unmatched segments, while emitting matched ones. FILTER is used to complement a Sort-Merge operator that must evaluate both children independently. Similarly, FILTER is also useful when there exists *cyclic* dependencies among multiple sub-patterns that reference each other. Specifically, we can place the FILTER operator as a parent to the sub-trees that contain such cyclic dependencies. This way, all dependent conditions are lifted and evaluated at the FILTER operator, and each SEGGENFILTER operator is replaced by a SEGGENWINDOW.

As an example, the query plan in Figure 5 cannot use a SORTMERGECONCAT for the top-level Concatenation operator because in the right sub-tree `CORRELATE` in the SEGGENFILTER references the segment matched by `UP` in the left sub-tree, so the two children cannot be evaluated independently. FILTER operator can solve this issue. Figure 6 shows an equivalent query plan but with a SORTMERGECONCAT: the original SEGGENFILTER in the right is replaced by a SEGGENWINDOW that has no dependency, and a FILTER with the condition `corr(CORRELATE.x, UP.x) >= 0.9` is placed right above the SORTMERGECONCAT. In this FILTER, the condition is evaluated using the payloads of the segments joined by the SORTMERGECONCAT. An analogy can be made with relational databases: a FILTER is like evaluating a predicate involving two relations on their JOIN result because this predicate cannot be "pushed-down". In the end, FILTER operator adds more flexibility to the plan space, but the optimizer still needs to choose which plan to use.

4.4.2 Not Operators. A Not operator negates the result of its child: when called on its `eval()`, the Not operator emits segments that are not matched by its child given the same search space. T-REX uses two physical implementations of the Not operator: MATERIALIZENOT and PROBENOT. MATERIALIZENOT calls the child's `eval()` once and uses a cache to materialize all segments matched by its child in the given search space, and then emits the complement. PROBENOT, on the other hand, iterates over all windowed segments in the search space, and calls the child's `eval()` on every segment to check if the segment is a match: skip the matched ones and emits the unmatched ones. The two implementations impact performance differently: if the number of probes is small because the Not's search space is small, then PROBENOT may be preferred; when the opposite is true, then MATERIALIZENOT may be better. It is the job of the optimizer to decide which one to use.

Because referenced segments are provided by `refs` in `eval()`, T-REX's Not operator supports arbitrary sub-patterns with unlimited references to outside of the operator. It is more general than existing CEP systems' Not operators that must be on a single event (i.e., point variable) [20], and must use post-processing in case of multiple references [41].

It is also useful to note that a condition in the sub-tree of a Not operator cannot be placed into a FILTER above the Not because that would produce incorrect result.

4.4.3 Kleene Operators. MATERIALIZEKLEENE is the only implementation of the Kleene operator in T-REX. It calls its child's `eval()` once and materialize all segments in a hash table using their start positions as keys. From there, the operator assembles "linked" segments into Kleene results using a breadth-first search algorithm with search space pruning using the embedded window.

4.5 Special Operators

Besides the operators discussed so far, T-REX uses a several special operators that potentially improve execution performance, and we will discuss them briefly.

4.5.1 Sub-pattern Materialization. The SUBPATTERN operator materializes a repeated sub-pattern in a query pattern to avoid redundant evaluation.

4.5.2 Wild Window Concatenation. A wild window is a window with no length constraint. A query pattern such as `UP, WINDOW, DOWN` where `WINDOW AS window()` specifies a "gap" of arbitrary length between the two segments matching `UP` and `DOWN` respectively. When implemented as two physical Concatenation operators, the first operator is between a sub-pattern and a SEGGENWINDOW with a wild window, so every segment produced by the sub-pattern will be "amplified" by concatenating with all possible next segments of lengths from 1 to the maximum allowed given the time series length. This potentially creates a huge intermediate result set and increases the input size of the second Concatenation operator. In order avoid this issue, T-REX would fuse such chain of two physical Concatenations into a single Wild Window Concatenation physical

operator (WConcat), which uses a Nested-Loop algorithm to join segments such that the left's ending position is less or equal to the right's starting position. Because the segments are paired directly, the intermediate results are avoided.

5 OPTIMIZATION

T-ReX's optimizer is responsible for the selection of physical operators and ordering binary physical operators like Concatenations and And. Using a cost model, the optimizer searches over the complete plan space[7], including bushy plans, using dynamic programming to find the minimal-cost plan. This section dives into the details of the optimizer. Following our third insight discussed in Section 3, the highlights of the optimizer are: (1) using range size as proxy for the search space for a light-weight cost model; (2) incorporating embedded windows for better cardinality estimates; and (3) using parametrized functions of running time as cost estimates.

5.1 Cost Model

Search space affects plan cost. Consider a physical plan with a root search space ($S = [s_i, s_j], E = [e_i, e_j]$) – a matched segment's start position must be within $[s_i, s_j]$ and its end position in $[e_i, e_j]$. The larger the search space, the higher number of possible segments.

One challenge in incorporating search space in the cost model is that it is not feasible to use search space directly. As discussed in Section 4, search space may be data-dependent. For example, in a RightProbeConcat, the right sub-tree's search space is conditioned on the segment from the left sub-tree. This leads to a heavy-weight cost models needing many complex statistics.

In order to keep the cost model light-weight, T-ReX uses search space's range sizes (i.e., the lengths of the start position's range and end position's range, respectively) as a proxy for the real search space. Let ℓ_s and ℓ_e denote the start and end position's range size respectively, i.e., $\ell_s = |S| = |s_j - s_i|$ and $\ell_e = |E| = |e_j - e_i|$, and let $\text{COST}_{op}(\ell_s, \ell_e)$ denote the plan's cost rooted at op and with search space range sizes (ℓ_s, ℓ_e).

The cost of a plan tree with root op has three components:

(1) The cost of the sub-tree(s), or the cost of its children's `eval()`: denoted by COST_{left} and COST_{right} for binary operators, and COST_{sub} for unary operators.
(2) The cost of evaluating the embedded Boolean condition in op: when the condition contains an aggregate, the unit cost of each aggregation is f_δ, and the unit cost of each `lookup()` call using index is $f_{\delta'}$; when the condition has no aggregate, the cost is 0.
(3) The cost of the operator op itself, denoted by f_{op}.

Cost of Sub-trees. For non-probe operators, the cost of sub-trees is obtained by using the range sizes of the sub-trees' search space. For probe operators, their cost models become interesting because the probed sub-tree's `eval()` is called multiple times. We discuss three operators, while all are listed in Table 1.

For RightProbeConcat with search space ($S = [s_i, s_j], E = [e_i, e_j]$), it calls the left's `eval()` with ($S = [s_i, s_j], E = [s_i, e_j]$), and the cost is $\text{COST}_{left}(\ell_s, \ell_{se})$, where ℓ_{se} denotes the concatenation point's search space range size $|e_j - s_i|$. Since the search space of the root operator is ($S = [0, n-1], E = [0, n-1]$) (Section 4.1), the search space of any sub-tree falls into one of 4 cases based on how search space is propagated among operators: (1) $s_i = s_j < e_i = e_j$; (2) $s_i = s_j = e_i < e_j$; (3) $s_i < s_j = e_i = e_j$; (4) $s_i = e_i < s_j = e_j$. For (1), $\ell_s = \ell_e = 1$, the expected value of $\ell_{se} = n/3$ (uniform assumption [4]). For (2), $\ell_{se} = \ell_e$; for (3), $\ell_{se} = \ell_s$; for (4), $\ell_{se} = \ell_s = \ell_e$. Please refer to Appendix C.1 for an expanded discussion. For each left segment $[l_i, l_j]$, RightProbeConcat calls the right `eval()` with search space ($S = [l_j, l_j], E = [e_i, e_j]$) which has range sizes of $(1, \ell_e)$ So the total cost is $D(\mathbb{C}_l, \ell_{se}) \times \text{COST}_{right}(1, \ell_e)$, where $\mathbb{C}_l$ is the cardinality of left sub-tree, and $D(\mathbb{C}_l, \ell_{se})$ is the number of unique end positions in left segments due to caching—it calculates the expected number of distinct items from $\mathbb{C}_l$ number of draws with replacement from a collection of ℓ_{se} items, assuming independent uniform draws [5].

RightProbeAnd calls its left's `eval()` with the same search space as itself, which has range sizes (ℓ_s, ℓ_e). So the left's total cost is $\text{COST}_{left}(\ell_s, \ell_e)$. For each left segment $[l_i, l_j]$, the right's `eval()` is called with the search space ($S = [l_i, l_i], E = [l_j, l_j]$) (range size equals $(1, 1)$). Because sub-trees of an And have identical embedded windows due to push-down, the total cost of the right is rectified with a factor of $1/\text{Sel}_w$, where w is the window embedded in op and Sel_w is the selectivity of w given the search space.

ProbeNot calls the sub-tree's `eval()` once per candidate segment with search space range size $(\ell'_s, \ell'_e) = (1, 1)$ and there is in total $\ell_s \ell_e \times \text{Sel}_w$ candidate segments. Since the Not operator is looking for the negation of matching condition, the iterator from each call to the sub-tree's `eval()` closes after one segment gets emitted. Thus the cost per call is $\frac{\text{COST}_{sub}(1,1)}{\max(\mathbb{C}'_{in}, 1)}$, where $\mathbb{C}'_{in}$ is the input cardinality per call.

Cost of Operator. T-ReX approximates f_{op} using a linear function with one parameter θ on one variable. The choice of this variable is based on the operator's implementation. For a Right Probe operator, the variable is $(\mathbb{C}_l + \mathbb{C}_{out})$ because only the left sub-tree is enumerated over. Similarly for a Left Probe the variable is $(\mathbb{C}_r + \mathbb{C}_{out})$. For other operators, $(\mathbb{C}_l + \mathbb{C}_r + \mathbb{C}_{out})$ if it is binary; otherwise, $(\mathbb{C}_{in} + \mathbb{C}_{out})$. T-ReX bootstraps their parameters in an offline profiling procedure, using synthetic data generated by uniform distributions. Higher order functions potentially achieve better accuracy, however, because they are more expensive to get low-variance estimate in bootstrapping, we do not use them.

Cost of Evaluating Aggregates. Only three physical operators, SegGenFilter, SegGenIndexing, and Filter, may have aggregate evaluation. SegGenFilter and Filter evaluate once for each segment, taking $\mathbb{C}_{in} \times f_\delta$ cost in total for each aggregation. SegGenIndexing incurs an additional overhead of query-time index building, f_{ind}, but smaller unit time $f_{\delta'}$ using the index's `lookup()` method, taking $f_{ind} + \mathbb{C}_{in} \times f_{\delta'}$.

We have observed that the query-time indexing cost typically depends on start-end range size of search space, thus f_{ind} takes ℓ_{se} as a parameter; while aggregation cost per segment typically depends on segment length, hence f_δ and $f_{\delta'}$ takes average segment length ℓ_{in} as a parameter. Cost functions (f_{ind}, f_δ, $f_{\delta'}$) depend on the specific aggregate function P. For instance, `mann_kendall_test` and `linear_reg_r2` have quadratic and linear relationship between indexing cost and start-end range size, respectively. Therefore, T-ReX uses different cost functions (e.g., constant, linear, or quadratic)

[7] Conditions on multiple variables can constrain the order of sub-tree executions so a *validator* is built into our execution plan to enforce such constraints.

Logical Op	Input Card $\mathbb{C}_{in}(\ell_s, \ell_e)$	Output Card $\mathbb{C}_{out}(\ell_s, \ell_e)$	Physical Op	Plan Cost (COST_{op})
Segment Generator	$\mathbb{C}_{in}$=$\ell_s\ell_e \times \text{Sel}_w$	$\mathbb{C}_{in} \times \text{Sel}_{P\|w}$	SegGenWindow	$f_{op}(\mathbb{C}_{in}, \mathbb{C}_{out})$
			SegGenFilter	$f_{op}(\mathbb{C}_{in}, \mathbb{C}_{out}) + \mathbb{C}_{in} \times f_\delta(P, \ell_{in})$
			SegGenIndexing	$f_{op}(\mathbb{C}_{in}, \mathbb{C}_{out}) + f_{ind}(P, \ell_{se}) + \mathbb{C}_{in} \times f_{\delta'}(P, \ell_{in})$
Filter	$\mathbb{C}_{in}$= $sub.\mathbb{C}_{out}(\ell_s, \ell_e)$	$\mathbb{C}_{in} \times \text{Sel}_{P\|w}$	Filter	$f_{op}(\mathbb{C}_{in}, \mathbb{C}_{out}) + \mathbb{C}_{in} \times f_\delta(P, \ell_{in}) + \text{COST}_{sub}(\ell_s, \ell_e)$
Not	$\mathbb{C}_{in}$= $sub.\mathbb{C}_{out}(\ell_s, \ell_e)$	$\ell_s\ell_e \times \text{Sel}_w$ - $\mathbb{C}_{in}$	MaterializeNot	$f_{op}(\mathbb{C}_{in}, \mathbb{C}_{out}) + \text{COST}_{sub}(\ell_s, \ell_e)$
	$\mathbb{C}'_{in}$= $sub.\mathbb{C}_{out}(1, 1)$		ProbeNot	$f_{op}(\mathbb{C}'_{in}, \mathbb{C}_{out}) + \ell_s\ell_e \times \text{Sel}_w \times \frac{\text{COST}_{sub}(1,1)}{max(\mathbb{C}'_{in}, 1)}$
Kleene	$\mathbb{C}_{in}$= $sub.\mathbb{C}_{out}(\ell_{se}, \ell_{se})$	$\mathbb{C}_{in} \times \frac{\ell_s\ell_e}{\ell_{se}^2} \times \text{Sel}_{w\|w_s}$ + $\mathbb{C}_{in}^2 \times \frac{\ell_s\ell_e}{\ell_{se}^3} \times \text{Sel}_{w\|w_s, w_s}$	MaterializeKleene	$f_{op}(\mathbb{C}_{in}, \mathbb{C}_{out}) + \text{COST}_{sub}(\ell_{se}, \ell_{se})$
Concat	$\mathbb{C}_l$=$left.\mathbb{C}_{out}(\ell_s, \ell_{se})$ $\mathbb{C}_r$=$right.\mathbb{C}_{out}(\ell_{se}, \ell_e)$	$\frac{\mathbb{C}_l \times \mathbb{C}_r}{\ell_{se}} \times \text{Sel}_{w\|w_l, w_r}$	SortMergeConcat	$f_{op}(\mathbb{C}_l, \mathbb{C}_r, \mathbb{C}_{out}) + \text{COST}_{left}(\ell_s, \ell_{se}) + \text{COST}_{right}(\ell_{se}, \ell_e)$
	$\mathbb{C}'_l$=$left.\mathbb{C}_{out}(\ell_s, 1)$ $\mathbb{C}_r$=$right.\mathbb{C}_{out}(\ell_{se}, \ell_e)$		LeftProbeConcat	$f_{op}(\mathbb{C}'_l, \mathbb{C}_r, \mathbb{C}_{out}) + \text{COST}_{right}(\ell_{se}, \ell_e) + D(\mathbb{C}_r, \ell_{se})\text{COST}_{left}(\ell_s, 1)$
	$\mathbb{C}_l$=$left.\mathbb{C}_{out}(\ell_s, \ell_{se})$ $\mathbb{C}'_r$=$right.\mathbb{C}_{out}(1, \ell_e)$		RightProbeConcat	$f_{op}(\mathbb{C}_l, \mathbb{C}'_r, \mathbb{C}_{out}) + \text{COST}_{left}(\ell_s, \ell_{se}) + D(\mathbb{C}_l, \ell_{se})\text{COST}_{right}(1, \ell_e)$
And	$\mathbb{C}_l$=$left.\mathbb{C}_{out}(\ell_s, \ell_e)$ $\mathbb{C}_r$=$right.\mathbb{C}_{out}(\ell_s, \ell_e)$	$\frac{\mathbb{C}_l \times \mathbb{C}_r}{\ell_s\ell_e \times \text{Sel}_w}$	SortMergeAnd	$f_{op}(\mathbb{C}_l, \mathbb{C}_r, \mathbb{C}_{out}) + \text{COST}_{left}(\ell_s, \ell_e) + \text{COST}_{right}(\ell_s, \ell_e)$
	$\mathbb{C}'_l$=$left.\mathbb{C}_{out}(1, 1)$ $\mathbb{C}_r$=$right.\mathbb{C}_{out}(\ell_s, \ell_e)$		LeftProbeAnd	$f_{op}(\mathbb{C}'_l, \mathbb{C}_r, \mathbb{C}_{out}) + \text{COST}_{right}(\ell_s, \ell_e) + \mathbb{C}_r \times \frac{\text{COST}_{left}(1,1)}{\text{Sel}_w}$
	$\mathbb{C}_l$=$left.\mathbb{C}_{out}(\ell_s, \ell_e)$ $\mathbb{C}'_r$=$right.\mathbb{C}_{out}(1, 1)$		RightProbeAnd	$f_{op}(\mathbb{C}_l, \mathbb{C}'_r, \mathbb{C}_{out}) + \text{COST}_{left}(\ell_s, \ell_e) + \mathbb{C}_l \times \frac{\text{COST}_{right}(1,1)}{\text{Sel}_w}$
Or	$\mathbb{C}_l$=$left.\mathbb{C}_{out}(\ell_s, \ell_e)$ $\mathbb{C}_r$=$right.\mathbb{C}_{out}(\ell_s, \ell_e)$	$\mathbb{C}_l \times \text{Sel}_{w\|w_l} + \mathbb{C}_r \times \text{Sel}_{w\|w_r}$	SortMergeOr	$f_{op}(\mathbb{C}_l, \mathbb{C}_r, \mathbb{C}_{out}) + \text{COST}_{left}(\ell_s, \ell_e) + \text{COST}_{right}(\ell_s, \ell_e)$

Table 1: T-ReX **Operators, Cardinality Estimators, and Cost Models**

for different aggregates. Similar to operators, T-ReX bootstraps parameters of aggregate cost functions offline, and average segment length ℓ_{in} is sampled at query time.

We discuss offline parameter bootstrapping in Appendix D.

5.2 Cardinality Estimation

T-ReX's cost model requires estimates of input and output cardinalities. Unlike existing works, T-ReX's cardinality estimator uses search space and embedded window to generate estimations.

The cardinality estimation of a physical plan is performed bottom-up starting from the leaf nodes, which are always Segment Generators. The input cardinality is $\ell_s\ell_e$ without window and $\ell_s\ell_e \times \text{Sel}_w$ with window due to window's selectivity. For a non-leaf operator node, its input cardinality is the output cardinality of its child(s). Table 1 lists the input cardinality and the range sizes for each child.

Given a plan and a search space, its output should be the same regardless of the selection of physical operators. Observing this, our cardinality estimator returns the same result for physical plans of the same logical plan. Table 1 lists the output cardinality for every logical operator. We discuss the details for each operator below.

Segment Generator and Filter. Both operators take a Boolean condition and only output segments satisfying it, so the output cardinality is $\mathbb{C}_{in} \times \text{Sel}_{P|w}$, where $\text{Sel}_{P|w}$ denotes the condition P's selectivity within the windowed search space. T-ReX samples data-dependent statistics like selectivity ($\text{Sel}_{P|w}$) at query time. Please refer to Appendix D.3 for details. In our experiments, the sampling cost is negligible compared to the overall query processing cost.

Not. As a Not's output is the complement of the set of the sub-tree's segments in the search space, its output cardinality is $\ell_s\ell_e \times \text{Sel}_w - \mathbb{C}_{in}$. Sel_w denotes the selectivity of the embedded window condition w, which is assigned to the Not operator and pushed down to its child sub-tree during logical plan rewrite (Section 3). We provide an expanded discussion in Appendix C.2. When there is Concatenation within Not operator, we rectify $\mathbb{C}_{in}$ as $D(\mathbb{C}_{in}, \ell_s\ell_e \times \text{Sel}_w)$ to accommodate for the duplicates caused by the Concatenation.

Concatenation. Concatenation can be modeled as a Cartesian product followed by a filter for `left.end=right.start`. We estimate this filter selectivity as $\frac{1}{\ell_{se}}$ assuming uniform distribution of start and end positions over the start-end range. Combined with the window's selectivity conditioned on the sub-trees' windows, the estimated output cardinality is $\frac{\mathbb{C}_l \times \mathbb{C}_r}{\ell_{se}} \times \text{Sel}_{w|w_l, w_r}$.

Kleene. Kleene is a self-concatenation and its cardinality depends on the number of occurrences specified by the query. For simplicity, we use the estimator for `A*` for all cases. The estimator is approximated as a sum the of single occurrence, `A`, and two occurrences, `Con(A, A)`. For the single occurrence, we check whether the segment from sub-tree falls in the search space of Kleene. Assuming uniform distribution of start and end positions, the selectivity is estimated as $\frac{\ell_s}{\ell_{se}}\frac{\ell_e}{\ell_{se}}$ and the output cardinality is $\mathbb{C}_{in} \times \frac{\ell_s\ell_e}{\ell_{se}^2} \times \text{Sel}_{w|w_s}$, where w_s denotes its sub-tree's window. For two occurrences, the estimation is similar to Concatenation: the cardinality is $\frac{\mathbb{C}_{in}^2}{\ell_{se}}$ after accounting for the concatenation condition (i.e., left's end is right's start), combined with the same search space selectivity as above, all together it is $\mathbb{C}_{in}^2 \times \frac{\ell_s\ell_e}{\ell_{se}^3} \times \text{Sel}_{w|w_s, w_s}$. The general case is approximated using one and two occurrences because we empirically observed that the number of three or more occurrences drops significantly due to the concatenation condition, the window and search space constraint in Kleene. Such approximation also incurs lower estimation cost.

And. The estimator is similar to concatenation operator with an equality filter condition – the left and right segments must have identical starts and ends. Again assuming uniform distribution,

Dataset	SP500	COVID-19	Weather	Taxi	NASDAQ
# of Series	503	3342	36	1	1
Series Length	252	64	1854	10320	351795

Table 2: Dataset Statistics

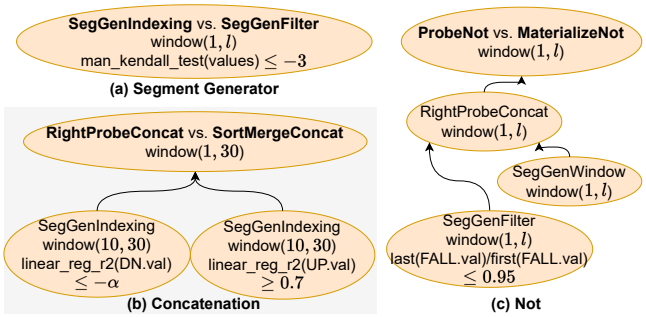


Figure 7: Physical Plans Used in Section 6.1

the filter selectivity is $\frac{1}{\ell_s \ell_e \times \text{Sel}_w}$. Due to window push-down, the windows embedded in the And node and its sub-trees are the same. Hence, $\text{Sel}_{w|w_l,w_r} = 1$ and the output cardinality is $\frac{\mathbb{C}_l \times \mathbb{C}_r}{\ell_s \ell_e \times \text{Sel}_w}$.

Or. Because this operator emits segments coming from both sub-trees, the cardinality estimate is simply $\mathbb{C}_l \times \text{Sel}_{w|w_l} + \mathbb{C}_r \times \text{Sel}_{w|w_r}$.

Note that all window selectivity used in the cardinality estimator, such as Sel_w, $\text{Sel}_{w|w_s}$, and $\text{Sel}_{w|w_l,w_r}$, are calculated with closed form formulas based on the each window's length constraints.

6 EXPERIMENTS

We start with a micro-benchmark that demonstrates the relative performance of physical operators under different scenarios while highlighting the need for the new physical operators in T-ReX (Section 6.1). Next, we move on to end-to-end performance evaluations to measure our cost-based optimizer's performance (Section 6.2) and how does T-ReX stand up to existing works in (Section 6.3).

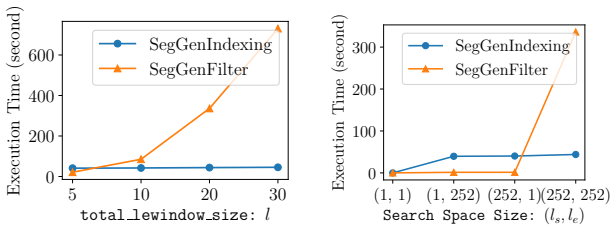
Datasets. Table 2 lists the datasets used in our experiment, along with the number of series and the length per series. `SP500` contains the daily opening prices of stock tickers in the S&P500 in the past year, following the literature [28, 41]. `NASDAQ` is used by OpenCEP [20]. `COVID-19` contains weekly confirmed cases per county in the U.S. (aggregated from the daily cases [1]) since 2020-01 and there are in total 3342 counties. `Weather` [6] contains 5 years of hourly temperature in 36 cities. `Taxi` [16] contains taxi trips counted every half-hour in New York City from 2014-07-01 to 2015-01-31.

Platform We conduct experiments on a Windows 11 PC with Intel® Core™ i7-9800X CPU @3.80GHz and 64GB memory at 2666MHz. All executors are single-thread.

6.1 Benefits of Multiple Physical Operators

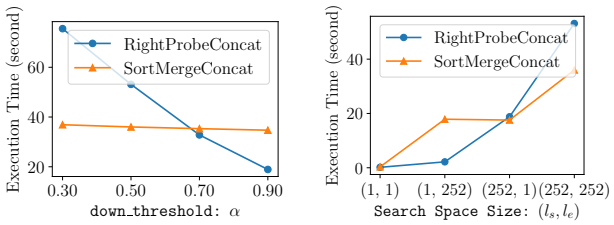
T-ReX designs and implements multiple physical algorithms per logical operator. In this section, we show that different physical operators are preferred under different scenarios.

6.1.1 Queries. This micro-benchmark uses `SP500` dataset. The physical plans used are depicted in Figure 7 with varying parameters including window length ℓ, R2 threshold α, and search space range size (ℓ_s, ℓ_e). The meaning of the corresponding queries are: (a) identify sub-series with downward trend; (b) identify sub-series with V-shape; (c) identify sub-series that does not exist 5% drop from the starting price. The queries are used to test Segment Generators, Concatenation, and Not operators, respectively. We omit And operators due to its similarity to Concatenation.



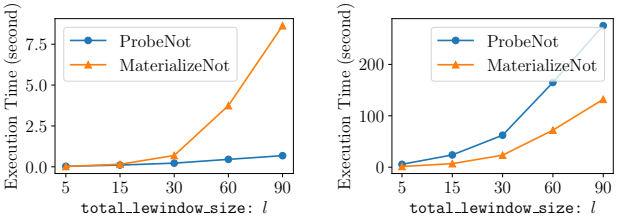
(a) Execution time vs. window size with search space being $(252, 252)$. (b) Execution time vs. search space size, max window size 20.

Figure 8: SegGenIndexing vs. SegGenFilter.



(a) Execution time vs. thresholds and search space size $(252, 252)$ (b) Execution time vs. search space size with a threshold of 0.5.

Figure 9: RightProbeConcat vs. SortMergeConcat.



(a) Execution time vs. window size, search space size $(1, 252)$. (b) Execution time vs. window size, search space size $(252, 252)$.

Figure 10: ProbeNot vs. MaterializeNot.

6.1.2 Results and Discussion. The results are shown in Figure 8, 9, and 10. The highlights are: (1) there exists no single physical operator that is always better than the other; (2) SegGenIndexing is useful in eliminating duplicated computation among segments especially when there are many segments; (3) Probe-based operators have an edge when the probe count is small under smaller search space or having selective sub-trees.

Segment Generator. SegGenIndexing and SegGenFilter outperform each other when with different values of l and (ℓ_s, ℓ_e). In Figure 8a, SegGenIndexing's execution time stays relatively stable, while SegGenFilter's increases with l. With the help of computation sharing, the cost per aggregate function call in SegGenIndexing is much smaller than in SegGenFilter. As a trade-off, SegGenIndexing incurs a one-time index-building overhead. Thus, only when the number of candidate segments is small, such as when the window size or search space is small (Figure 8b), SegGenFilter has the opportunity to outperform SegGenIndexing.

Concatenation. As shown in Figure 9a, SortMergeConcat's time stays stable since the time spent in its sub-trees dominates

and it remains the same regardless of α. On the other hand, RightProbeConcat's time decreases with increasing α, due to more selective `linear_reg_r2(DN.val)` $\leq -\alpha$ in the left sub-tree, hence less number of left segments and right probes. Similar observations can be made from Figure 9b: when the input search space makes the left search space small (i.e., $[1, 1]$ and $[1, 252]$) thereby more selective, RightProbeConcat performs better.

Not. Figure 10a shows that when the input search space size is small, i.e., $(l_s, l_e) = (1, 252)$, ProbeNot is faster than MaterializeNot. Figure 10b shows when the search pace size is large, i.e., $(l_s, l_e) = (252, 252)$, ProbeNot is more expensive. This is because smaller input search space leads to smaller number of probes in ProbeNot.

6.2 Optimizer Evaluation

To evaluate the effectiveness of T-ReX's optimizer, we compare optimizer-generated plans with plans generated using rules.

6.2.1 Baselines. There are two dimensions of variations for generating baseline plans: the join ordering of binary operators and the choice of physical operators (see Table 1). There are two join orders: Left-Deep and Right-Deep. The selection rules are: (1) for Left-Deep a binary operator is either a Sort-Merge or a Right-Probe, and for Right-Deep a binary operator is either Sort-Merge or Left-Probe; (2) if a plan contains a Not operator, we can choose either MaterializeNot or ProbeNot; (3) always choose SegGenIndexing over SegGenFilter whenever the variable involving aggregates that have implemented the `index()` method because we observe that baselines with SegGenIndexing is always faster. Thus, there are in total $2 \times 2 = 4$ baselines for plans without a Not operator, and $2 \times 2 \times 2 = 8$ baselines for plans with a Not operator as shown in Table 4. For instance, `pr_left` refers to the baseline using Left-Deep with Probe operators for Binary operator and MaterializeNot.

6.2.2 Queries. We use 11 parameterized queries, and all of them have variable-length patterns. First 6 queries have conditions with aggregate functions on segments, and the next 5 queries including `limit_sell` and those from prior works [20, 28] have no condition with aggregate. `v_shape`, `head_shldr`, `outlier`, `limit_sell`, `AFA_Q1` and `AFA_Q2` use the `SP500` dataset; `rebound` uses the `COVID-19` dataset; `cld_wave` uses the `Weather` dataset; `rptd_pttrn` uses the `Taxi` dataset; and `OpenCEP_Q1` and `OpenCEP_Q2` use the `NASDAQ` dataset. We use at least 9 parameter sets for every query except for `OpenCEP_Q1` and `OpenCEP_Q2` which use 5. For each query instance, we count the total time to find all matches from all time series in the dataset it uses.

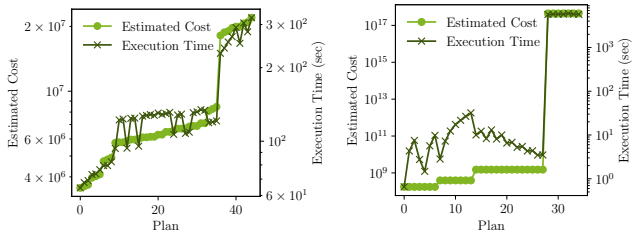
Table 3 lists the queries' regular expression patterns; all variables are segment variables except for the point variabels `A1`, `A2` and `A3` in OpenCEP queries. In these patterns, `DOWN` and `UP` are defined using linear regression R^2; `UP_MK` is defined using Mann-Kendall test for up trend; `RISE` and `FALL` are defined by comparing the first and last points' values; `W`, `W1`, `WINDOW`, etc. are segment variables with window conditions. In `head_shdlr`, `NCK_2_HD`, `SHDLR_2_HD`, `HD_2_NCK` and `HD_2_SHDLR` are defined using the ratio of the first and last points' values. In `OpenCEP_Q1` and `OpenCEP_Q2`, `A1`, `A2` and `A3` are point variables defined by equality conditions on stock tickers; `INC1` is defined as checking if `A2`'s value is greater than `A1`, similarly for `INC2`. For a query, each parameter set specifies the parameters such as the thresholds on the difference in `RISE` and ratios in `HD_2_SHDLR`. The complete query definitions and parameter sets are listed in Appendix E.

Query	Pattern
`v_shape`	`((DOWN & W) (UP & W)) & WINDOW`

`head_shldr`	`((UP1 & W)`
	`((DN1 & W) (UP2 & W & NCK_2_HD)) & SHLDR_2_HD`
	`((DN2 & W & HD_2_NCK) (UP3 & W)) & HD_2_SHLDR`
	`(DN3 & W)) & WINDOW`

`outlier`	`(UP1 OUTLIER UP2) & WINDOW`
`rebound`	`(UP1 ((DOWN & FALL) UP2) & RISE) & WINDOW`
`cld_wave`	`(W1 (DOWN & FALL & W2) W1) & UP_MK & WINDOW`
`rptd_pttrn`	`((W1 (UP & RISE & W2) W3 (DOWN & FALL & W2)` `W1) & WINDOW){n}`
`limit_sell`	`RISE & WINDOW & ~(FALL W)`
`OpenCEP_Q1`	`(A1 W (A2 & INC1) W (A3 & INC2)) & WINDOW`
`OpenCEP_Q2`	`(((A1 W A2) & FALL)+) & WINDOW`
`AFA_Q1`	`(LARGE_FALL & W ((FALL & W)+ (RISE & W)+){k}` `& EQ_FALL_AND_RISE ) & WINDOW`
`AFA_Q2`	`(LARGE_FALL & W ((FALL & W)+ (RISE & W)+)+)` `& RECOVER & WINDOW`

Table 3: Queries used in experiments.



(a) `rebound` **(NDCG = 1.00)** **(b)** `OpenCEP_Q1` **(NDCG = 0.74)**

Figure 11: Estimated costs and execution times for queries with the best (left) and worst (right) NDCG scores.

6.2.3 Results and Discussion. The results are shown in Table 4. The numbers show the median slow-down ratio of each method over the fastest plan for each query. So the smaller the ratio, the better the performance, and 1.0 indicates the fastest plan. For instance, `pr_left` is 12.9× (median) slower than the fastest query plan across the parameter space of `cld_wave` query. Box-plots of all queries can be found in Appendix F. A few takeaways from Table 4: *(1)* no single baseline performs consistently well across all baselines on all queries; *(2)* T-ReX optimizer outperforms all baselines for all queries in terms of the median slow-down over the fastest plan; *(3)* T-ReX optimizer successfully selects the best plan for at least 50% instantiated queries for all query templates.

To test the optimizer's accuracy in ranking plans based on costs, we calculated the Normalized Discounted Cumulative Gain (NDCG) scores [33] using the physical plans generated in the previous experiment. The score, between 0 and 1, measures agreement between two lists of plans respectively ordered by estimated cost and execution time. T-ReX's optimizer produces accurate ranking of plans – its NDCG score is greater than 0.9 for 8 out of 11 queries using only 5 series samples and 1 ms of profiling time. On `SP500`, increasing the series sampled from 5 to 500 brought at most 0.13 improvement on two queries while other queries saw less than 0.01 difference.

	v_shape	head_shldr	outlier	rebound	cld_wave	rptd_pttrn	OpenCEP_Q1	OpenCEP_Q2	AFA_Q1	AFA_Q2
pr_left	1.0	2.0	4.4	3.6	12.9	3.4	4.0	1.1	7.0	7.9
pr_right	1.0	4.4	4.5	1.5	1.3	4.4	1333.2	t.o.	210.7	1841.8
sm_left	1.5	5.0	1.8	1.7	12.2	8.3	1.6	6104.4	6.1	9.6
sm_right	1.5	3.8	1.7	1.7	11.9	9.1	1.5	6059.4	6.2	9.6
optimizer	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

	pr_left	pr_right	sm_left	sm_right	pr_left_pnot	pr_right_pnot	sm_left_pnot	sm_right_pnot	optimizer
limit_sell	1.1	6.0	2.4	2.4	1.1	65.9	1.3	1.3	1.0

Table 4: Median slow-downs of methods over the fastest method for each query. Each cell refers to a method and query. Cell value of 1.0 indicates the method selected the best plan for the query; 't.o.' indicates time-out of query instances at 2 hours.

Figure 11 shows the estimated costs using 5 series sampled versus execution times of the two queries with the best and worst NDCG scores respectively. We can observe positive association in both plots. Figure 23 in Appendix F shows the complete results.

6.3 Comparing with Other Executors

In this section, we compare T-Rex with other executors on the same queries used in Section 6.2. The goal is to investigate the effectiveness of the three insights discussed in Section 3: search space pruning, computation sharing for aggregates, and optimization.

6.3.1 Baselines. The following baselines are used.

- T-ReX Batch. We disabled probe-based operators (LeftProbeConcat, RightProbeAnd, ProbeNot, etc.) to set T-ReX's tree-based executor in batch mode – every operator works on the whole series' search space.
- AFA [28], an improved version of NFA using augmented registers, eliminating the need to track matched records for every partial matching state. AFA represents the state-of-the-art of NFA-based pattern search executors used by existing systems such as Trino and Apache Flink. It is part of the Trill [27] framework for streaming analytic and we implemented a Python version based on the original. For each query, we created the state transition graph manually so that (1) we put cheaper Boolean conditions ahead of more expensive ones whenever possible; and (2) we pushed window conditions as early in the graph as possible. We note that our hand-tuned transition graph favors this AFA-based baseline, compared to automatically generated transition graph [25, 52].
- Nested-AFA, an AFA-based executor that evaluates nested pattern (e.g., `limit_sell`) "top-down": parent pattern (e.g., `RISE & WINDOW`) is evaluated first while treating inner patterns as match-all placeholders; then, evaluate inner patterns (e.g., `~(FALL W)`) under the search space conditions inferred from the parent pattern matches. For patterns without nested sub-patterns, Nested-AFA reverts back to AFA. We implemented the algorithm proposed by [43] and optimized it for batch execution.
- OpenCEP [20], a Python library for pattern search on streaming data, developed based on existing research works [34–37, 41]. We used its default tree-based executor.
- ZStream [41], the original tree-based executor. We use the implementation provided by the OpenCEP library [20].

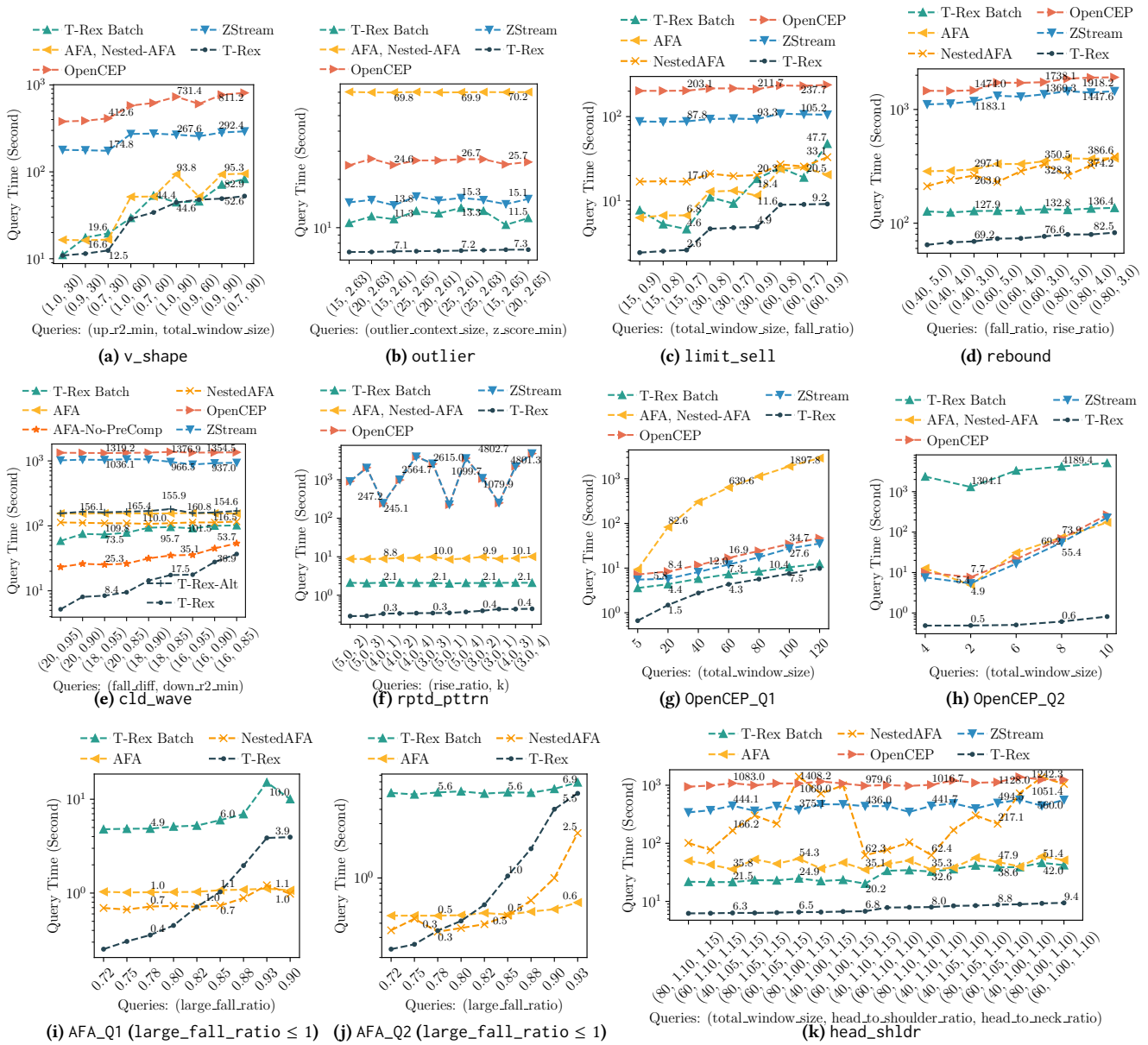
All executors including T-ReX are implemented using Python.

6.3.2 Result and Discussion. Figure 12 shows the query time (including optimizer time for T-ReX) of all queries and parameters. Figure 22a in Appendix F shows the median speedups or slowdowns (negative) of T-ReX over the baselines. Computation sharing was used for aggregate functions for all baselines – indexing-building was the first step of every baseline's query execution, in order to measure the effect of search space pruning and optimizer. From the result, T-ReX out-performed all baselines except for some parameters on `AFA_Q1` and `AFA_Q2` in comparison with AFA, within less than 5 seconds.

Comparing T-ReX with T-ReX Batch, the observed speedups are explained by the availability of probe operators. The median of median speedups of all queries is 3.9×, with the highest median speedup reaching 6269× for `OpenCEP_Q2` (Figure 12h). The highest speedup was due to T-ReX Batch using a SortMergeAnd operator that passed a whole-series search space on a long series (351K points) to both sub-trees, while in fact one sub-tree was much more selective than the other and a probe operator would have been able to take advantage of the difference. The result shows that probe operators can be extremely effective for search space pruning, and because of them T-ReX is able to perform well in a wider range of scenarios than batch-based executors.

AFA performed relatively well due to our manual optimization mentioned previously, however, such optimization is not performed in existing systems such as Trino and Apache Flink, which translate user's query directly into state transition graph without modification. Nevertheless, T-ReX achieved 6× over AFA in median across median speedups, the highest among them is 147× for `OpenCEP_Q1`. This is a feat of T-ReX's optimizer which reorders the evaluation of Boolen conditions to minimize net processing cost. For example, for `cld_wave` (result in Figure 12e; see queries and execution plans in Section 1), AFA was manually optimized so the cheap condition on temperature difference was evaluated before linear regression condition, and both of these conditions were evaluated at the end of each `DOWN+`. Still, AFA evaluated `A+` (least selective) first before `D+`. T-ReX on the other hand evaluated `DIFF` (most selective) first to reduce evaluations of subsequent conditions. For `AFA_Q1` and `AFA_Q2`, `large_fall_ratio` controls the ratio of the last and first values in `LARGE_FALL`. When the ratio approached 1.0, T-ReX underperformed AFA by a few seconds because the MaterializeKleene operator's cost increased with larger input size due to decreasing selectivity of `LARGE_FALL`. AFA's query time leveled because `LARGE_FALL` remained the most selective segment and happened to be evaluated first in these queries. A more efficient physical operator for Kleene patterns is part of our future work.

For queries with nested sub-patterns, Nested-AFA performed similarly to AFA with some slow-downs (e.g., `head_shldr`) and some speed-ups (e.g., `cld_wave`). Since Nested-AFA still uses a



(a) `v_shape` (b) `outlier` (c) `limit_sell` (d) `rebound`

(e) `cld_wave` (f) `rptd_pttrn` (g) `OpenCEP_Q1` (h) `OpenCEP_Q2`

(i) `AFA_Q1` (`large_fall_ratio` ≤ 1) (j) `AFA_Q2` (`large_fall_ratio` ≤ 1) (k) `head_shldr`

Figure 12: Query time versus parameter set.

fixed execution order, it does not guarantee optimality. Because T-ReX's optimizer chooses execution order based on data, it generally performs better, with a median of median speedups of 3.9× over Nested-AFA for queries with nested sub-patterns. Such queries have separate lines for AFA and Nested-AFA in Figure 12. For queries without nested sub-patterns, Nested-AFA reverts to AFA as shown by the same line. Besides, the algorithm proposed by [43], which we use, cannot evaluate nested segments inside Kleene closure, so for those queries it reverts to AFA as well.

T-ReX out-performed OpenCEP and ZStream on all queries, with a median of median speedups of 42× and 19× respectively; the highest median speedup is 4177× over OpenCEP and 4104× over ZStream for `rptd_pttrn`. T-ReX's window-aware physical operators played an important role. Specifically, both OpenCEP's and ZStream's execution times for `OpenCEP_Q2` (Figure 12h) increased significantly with window size and reached 267 and 227 seconds respectively when the maximum window size is 10 minutes – a small window considering the time series spans one trading day. On the contrary, benefiting from our window-aware MaterializeKleene operator, T-ReX's execution time stayed within 1 second. The OpenCEP library's query interface does not support `AFA_Q1` and `AFA_Q2` which involve nested Kleene closures, so OpenCEP and ZStream have no result for these two queries.

To measure the effect of computation sharing on aggregate functions, we compare every executor (T-ReX and baselines) with itself

on whether computation sharing is enabled or disabled. Figure 22b in Appendix F shows the median speedups or slowdowns (negative) of enabling over disabling computation sharing. Queries with expensive aggregate functions mostly saw performance gain. Specifically, T-ReX, T-ReX Batch, AFA and Nested-AFA saw 10× gain for `v_shape` and at least 3× gain for `rebound` in median query time; OpenCEP and ZStream also saw their greatest gains from executing `rebound` and `v_shape`. Interestingly, AFA suffered a 4.9× slowdown for `cld_wave` for pre-computing Mann-Kendall test (Figure 12e), while T-ReX's optimizer did not choose computation sharing. The cost of computing Mann-Kendall test ($O(n^2)$) one time for the whole series is more expensive than the total cost of individual evaluations on segments as the condition is located last in the optimized AFA's state transition graph. This observation demonstrated again the need for a search-space-aware optimizer to determine whether computation sharing for aggregates is needed.

We also looked at how query specification itself impacts performance. For instance, we use an alternative query for `cld_wave`: `((W1 (DOWN_AND_FALL & W2) W1) & UP_MK & WINDOW)`, in which `DOWN_AND_FALL` combines the conditions for `DOWN` and `FALL` in the sub-pattern `(DOWN & FALL)`. In Figure 12e the line T-Rex-Alt corresponds to the query time of this alternative specification, which is at least 4× slower than the original query. Based on all results in Figure 12, we recommend finer-grained specification utilizing the And (`&`) operator for more optimization opportunities and replacing point variables with segment-variables when dependent conditions exist across point variables.

7 RELATED WORK

Complex Event Processing (CEP). Existing works in CEP study pattern search for streaming scenario [21, 29, 34–37, 41, 44, 49, 52]. Pattern search languages such as SASE [21, 49] and Cayuga [29], share the same core features with slightly different syntax and grammar. Similar to SQL 2016's `MATCH_RECOGNIZE` [9], pattern variables in these languages are defined at event level with Boolean constraints on events or records. The execution model of most existing works in CEP [21, 28, 49, 52] is based on non-deterministic finite automaton (NFA).

NFA-based Execution Model. Plain NFA is insufficient in expressing query patterns with dependent conditions across pattern variables. Recognizing this limitation, execution models like NFA^b [21][8] and AFA [28] have been proposed to allow added runtime information to be associated with NFA states — such automata are called augmented NFA (AFA) in [28]. Hence, when talking about NFA-based executor in CEP and row pattern recognition literature, we are essentially referring to AFA instead of the Plain NFA. Furthermore, iterative nested AFA [39] has been proposed for queries involving sub-queries. In essence, the outer query is evaluated using AFA first, followed by its inner sub-queries. Observing that the evaluation of inner sub-queries can repeat for multiple outer results, nested AFA with sharing is proposed by materializing intermediate results [43]. T-ReX is shown to outperform AFA [28] and nested AFA with sharing-enabled [43] in our experiments.

Translation from Query to NFA-based Execution Graph. How to construct an efficient transition graph under AFA is left to the users in [28]. Other works have studied how to construct efficient transition graphs automatically for the core language constructs (`Concatenation, Alternation, Kleene closure, Not`) [52] in CEP and also for special operators like Permutation [25]. In our experiment, we focused on the core language constructs [52] and hand-tuned an efficient transition graph for each query, which is more favorable to the NFA-based baseline.

Nested AFA is particularly useful when the compiler is unable to compile a nested pattern into one single AFA, although this is not an issue for standard `MATCH_RECOGNIZE` since it does not contain And and Not operator and the translation from a pattern to one single AFA is straight-forward. Given a nested pattern, the nested AFA is constructed iteratively with one sub-AFA per sub-query [39, 43]. In our experiments, we not only hand-tuned a single holistic AFA for each query pattern, but also automatically constructed a nested AFA with sharing based on [43].

Tree-based Execution Model. In NFA-based executor, the matching conditions are evaluated in the same order as they appear in the query. This misses optimization opportunities like reordering for selective conditions. To enable reordering, ZStream [41] leverages a tree-based execution model, while [37] proposes a tree-based NFA. In T-ReX, we adapts tree-based execution model, but with different physical operators allowing search space refinement and reference passing. Together with the optimizer, this design helps T-ReX achieve better performance over a wider range of queries.

Techniques Applicable to Any Execution Model. Cadonna et al. [26] proposes a two-phase matching strategy, consisting of (1) a pre-processing phase to eliminate irrelevant events and (2) a pattern-matching phase. Any executor, including T-ReX, can be plugged into phase (2) and potentially benefit. Ray et al. speeds up concurrent queries in streaming scenario using common sub-patterns across queries [42], and proposes to materialize common sub-query within one query for reducing redundant computation [43]. In T-ReX, we have also proposed computation sharing for aggregation functions, however, it is only applicable in historical setting and is orthogonal to the sub-pattern sharing in [42, 43]. That is, we can potentially improve T-ReX using shared sub-patterns within one query too. In all, such techniques are orthogonal to our proposed T-ReX executor and can be applied to T-ReX for further improvement.

Shape Search. Several existing works [22, 31, 40, 45] focus specifically on shape-related patterns. Among them, Qetch [40] and ShapeSearch [45] have segment-level abstraction for patterns. Still, Qetch and ShapeSearch only support a limited set of shape-related patterns, let alone user-defined matching conditions as in T-ReX. Another difference is that they only support a subset of T-ReX's operators. Specifically, Qetch only supports Kleene and Not operator; ShapeSearch supports Concatenation, And, and Or operator. These systems have different focuses: Qetch proposes a novel matching algorithm and distance function for matching user sketches to time series; ShapeSearch proposes a pre-defined scoring function and a novel approximation algorithm for returning top-1 match. They are different from T-ReX which supports a `MATCH_RECOGNIZE` interface and returns all matches exactly.

[8] Symbol b in NFA^b is short for 'buffer'.

Time series Database. Lately, many efforts have been made in developing time series databases (TSDBs). Popular TSDB includes InfluxDB[7], Kdb+[10], TimescaleDB[17], and Azure Data Explorer[2]. TSDBs focus mainly on data ingestion and storage, while also provide query interface for analytics. Both SEQ and TSDBs focus mostly on the traditional data analytics operators like Select, Project, Aggregate, and Join. How to cross-optimize `MATCH_RECOGNIZE`, with or without T-ReX's extension, and these operators is still an open and interesting research problem.

Aggregates and Window Functions. Many papers have studied efficient evaluation of aggregates and window functions in databases [23, 38, 47, 48, 50]. Vogelsgesang et al. [47] showed merge sort tree to be a promising underlying index structure for implementing a wide range of aggregates and window functions with state sharing across evaluations. In T-ReX, every aggregate implements the interface functions `index()` and `lookup()`, and the implementation can use techniques in these works.

8 CONCLUSION

We presented T-ReX, a search engine for time series patterns. T-ReX addresses the performance issue in existing historical time series pattern search systems for handling variable-length query patterns. T-ReX leverages two major insights: (1) reasoning about query patterns in terms of *segments* enables a set of optimization opportunities, including variable evaluation reordering and computation sharing; (2) a *search-space-aware* executor and optimizer are needed for effective search space pruning and efficient query processing. Specifically, T-ReX extends the `MATCH_RECOGNIZE` syntax with segment variables and new operators. Its query processing framework uses an optimized tree-based executor to automatically exploit the relative difference in the cost and selectivity of segment variables to prune search space, employs novel physical operators capable of search space pruning, and enables computation sharing for aggregates. In our experiments on new and existing benchmarks, T-ReX demonstrated clear advantage in a wider range of patterns than existing executors, due to its reasoning at segment level, search-space-aware executor, new physical operators, and optimizer. For future work, we will investigate the possibilities of operator-level parallel execution and integration with SQL systems.

REFERENCES

[1] Covid-19 data repository by the center for systems science and engineering (csse) at johns hopkins university. https://github.com/CSSEGISandData/COVID-19/tree/master/archived_data/archived_time_series, 2021.

[2] Azure Data Explorer. https://azure.microsoft.com/en-us/services/data-explorer/, 2022.

[3] Cold Wave. https://en.wikipedia.org/wiki/Cold_wave, 2022.

[4] Expected Distance Between Random Points on a Line Segment. https://math.stackexchange.com/questions/195245/average-distance-between-random-points-on-a-line-segment, 2022.

[5] Expected Number of Distinct Items from a Multi-Set. https://math.stackexchange.com/a/72351, 2022.

[6] Historical hourly weather data 2012-2017. https://www.kaggle.com/selfishgene/historical-hourly-weather-data, 2022.

[7] Influxdb. https://www.influxdata.com/, 2022.

[8] Internals of PostgreSQL – Rewriter. http://www.interdb.jp/pg/pgsql03.html, 2022.

[9] ISO/IEC TR 19075-5:2016 Information technology – Database languages – SQL Technical Reports – Part 5: Row Pattern Recognition in SQL. https://www.iso.org/standard/65143.html, 2022.

[10] Kdb+. https://code.kx.com/q/learn/, 2022.

[11] MATCH_RECOGNIZE. https://trino.io/docs/current/sql/match-recognize.html, 2022.

[12] MATCH_RECOGNIZE - Snowflake Documentation. https://docs.snowflake.com/en/sql-reference/constructs/match_recognize.html, 2022.

[13] MATCH_RECOGNIZE for Pattern Recognition. https://docs.oracle.com/en/middleware/fusion-middleware/osa/19.1/cqlreference/pattern-recognition-match_recognize.html, 2022.

[14] MATCH_RECOGNIZE for Pattern Recognition. https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/dev/table/sql/queries/match_recognize/, 2022.

[15] MATCH_RECOGNIZE (Stream Analytics). https://docs.microsoft.com/en-us/stream-analytics-query/match-recognize-stream-analytics, 2022.

[16] The numenta anomaly benchmark (nab). https://github.com/numenta/NAB/tree/master/data/realKnownCause, 2022.

[17] TimescaleDB. https://www.timescale.com/, 2022.

[18] Snowflake MATCH_RECOGNIZE for finding transactions that do NOT match a pattern. https://livenewcapital.com/snowflake-match_recognize-for-finding-transactions-that-do-not-match-a-pattern/, 2023.

[19] Snowflake MATCH_RECOGNIZE for performing A/B analysis on streaming data. https://livenewcapital.com/match_recognize-for-performing-a-b-analysis/, 2023.

[20] Kolchinsky, Ilya and Schuster, Assaf. OpenCEP. https://research.redhat.com/blog/research_project/complex-event-processing-2/ https://github.com/ilya-kolchinsky/OpenCEP, 2021.

[21] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In J. T. Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 147–160. ACM, 2008.

[22] R. Agrawal, G. Psaila, E. L. Wimmers, and M. Zaït. Querying shapes of histories. In U. Dayal, P. M. D. Gray, and S. Nishio, editors, *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pages 502–514. Morgan Kaufmann, 1995.

[23] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. B. Schiefer, editors, *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, pages 336–347. Morgan Kaufmann, 2004.

[24] P. Buono, A. Aris, C. Plaisant, A. Khella, and B. Shneiderman. Interactive pattern search in time series. In R. F. Erbacher, J. C. Roberts, M. T. Gröhn, and K. Börner, editors, *Visualization and Data Analysis 2005, San Jose, CA, USA, January 17, 2005*, volume 5669 of *SPIE Proceedings*, pages 175–186. SPIE, 2005.

[25] B. Cadonna, J. Gamper, and M. H. Böhlen. Sequenced event set pattern matching. In A. Ailamaki, S. Amer-Yahia, J. M. Patel, T. Risch, P. Senellart, and J. Stoyanovich, editors, *EDBT 2011, 14th International Conference on Extending Database Technology, Uppsala, Sweden, March 21-24, 2011, Proceedings*, pages 33–44. ACM, 2011.

[26] B. Cadonna, J. Gamper, and M. H. Böhlen. Efficient event pattern matching with match windows. In Q. Yang, D. Agarwal, and J. Pei, editors, *The 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12, Beijing, China, August 12-16, 2012*, pages 471–479. ACM, 2012.

[27] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *Proc. VLDB Endow.*, 8(4):401–412, 2014.

[28] B. Chandramouli, J. Goldstein, and D. Maier. High-performance dynamic pattern matching over disordered streams. *Proc. VLDB Endow.*, 3(1):220–231, 2010.

[29] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White. Cayuga: A general purpose event monitoring system. In *Third Biennial Conference on Innovative Data Systems Research, CIDR 2007, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, pages 412–422. www.cidrdb.org, 2007.

[30] Y. Diao, N. Immerman, and D. Gyllstrom. Sase+: An agile language for kleene closure over event streams. *UMass Technical Report*, 2007.

[31] T. Fu, K. F. Chung, R. W. P. Luk, and C. Ng. Stock time series pattern matching: Template-based vs. rule-based approaches. *Eng. Appl. Artif. Intell.*, 20(3):347–364, 2007.

[32] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[33] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of IR techniques. *ACM Trans. Inf. Syst.*, 20(4):422–446, 2002.

[34] I. Kolchinsky and A. Schuster. Efficient adaptive detection of complex event patterns. *Proc. VLDB Endow.*, 11(11):1346–1359, 2018.

[35] I. Kolchinsky and A. Schuster. Join query optimization techniques for complex event processing applications. *Proc. VLDB Endow.*, 11(11):1332–1345, 2018.

[36] I. Kolchinsky and A. Schuster. Real-time multi-pattern detection over event streams. In P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 589–606. ACM, 2019.

[37] I. Kolchinsky, I. Sharfman, and A. Schuster. Lazy evaluation methods for detecting complex events. In F. Eliassen and R. Vitenberg, editors, *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, Oslo, Norway, June 29 - July 3, 2015*, pages 34–45. ACM, 2015.

[38] V. Leis, K. Kundhikanjana, A. Kemper, and T. Neumann. Efficient processing of window functions in analytical SQL queries. *Proc. VLDB Endow.*, 8(10):1058–1069, 2015.

[39] M. Liu, M. Ray, E. A. Rundensteiner, D. J. Dougherty, C. Gupta, S. Wang, I. Ari, and A. Mehta. Processing nested complex sequence pattern queries over event streams. In D. Zeinalipour-Yazti and W. Lee, editors, *Proceedings of the 7th Workshop on Data Management for Sensor Networks, in conjunction with VLDB, DMSN 2010, Singapore, September 13, 2010*, ACM International Conference Proceeding Series, pages 14–19. ACM, 2010.

[40] M. Mannino and A. Abouzied. Expressive time series querying with hand-drawn scale-free sketches. In R. L. Mandryk, M. Hancock, M. Perry, and A. L. Cox, editors, *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, CHI 2018, Montreal, QC, Canada, April 21-26, 2018*, page 388. ACM, 2018.

[41] Y. Mei and S. Madden. Zstream: a cost-based query processor for adaptively detecting composite events. In U. Çetintemel, S. B. Zdonik, D. Kossmann, and N. Tatbul, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 193–206. ACM, 2009.

[42] M. Ray, C. Lei, and E. A. Rundensteiner. Scalable pattern sharing on event streams. In F. Özcan, G. Koutrika, and S. Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 495–510. ACM, 2016.

[43] M. Ray, E. A. Rundensteiner, M. Liu, C. Gupta, S. Wang, and I. Ari. High-performance complex event processing using continuous sliding views. In G. Guerrini and N. W. Paton, editors, *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, pages 525–536. ACM, 2013.

[44] R. Sadri, C. Zaniolo, A. M. Zarkesh, and J. Adibi. Expressing and optimizing sequence queries in database systems. *ACM Trans. Database Syst.*, 29(2):282–318, 2004.

[45] T. Siddiqui, P. Luh, Z. Wang, K. Karahalios, and A. G. Parameswaran. Shapesearch: A flexible and efficient system for shape-based exploration of trendlines. In D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 51–65. ACM, 2020.

[46] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedure, caching and views in data base systems. *ACM SIGMOD Record*, 19(2):281–290, 1990.

[47] A. Vogelsgesang, T. Neumann, V. Leis, and A. Kemper. Efficient evaluation of arbitrarily-framed holistic SQL aggregates and window functions. In Z. Ives, A. Bonifati, and A. E. Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 1243–1256. ACM, 2022.

[48] R. Wesley and F. Xu. Incremental computation of common windowed holistic aggregates. *Proc. VLDB Endow.*, 9(12):1221–1232, 2016.

[49] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In S. Chaudhuri, V. Hristidis, and N. Polyzotis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 407–418. ACM, 2006.

[50] J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. In D. Georgakopoulos and A. Buchmann, editors, *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, pages 51–60. IEEE Computer Society, 2001.

[51] S. Yue, P. Pilon, and G. Cavadias. Power of the mann–kendall and spearman's rho tests for detecting monotonic trends in hydrological series. *Journal of Hydrology*,

259(1):254–271, 2002.

[52] H. Zhang, Y. Diao, and N. Immerman. On complexity and optimization of expensive queries in complex event processing. In C. E. Dyreson, F. Li, and M. T. Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 217–228. ACM, 2014.

A PROOF ON LANGUAGE EXPRESSIVENESS

A.1 Proof of Proposition 2.1

Proof. We first show a segment variable is expressible using standard MATCH_RECOGNIZE. Consider a segment variable S defined via clause 'DEFINE SEGMENT S AS S_Boolean_Condition(S)', we can express S using point variables and operators in standard MATCH_RECOGNIZE as follows: '(p* z) DEFINE p As true, z as S_Boolean_Condition(p, z)'. Note that p in S_Boolean_Condition(p, z) is bound to all events that have matched p so far in a partial match. As a concrete example, letting S_Boolean_Condition(S) be AVG(S.temp)>70, equivalently we can use '(p* z) DEFINE p As true, z as $\frac{\texttt{SUM(p.temp)+z.temp}}{\texttt{COUNT(p.temp)+1}}$>70'. This way, the whole event sequence matching pattern (p* z) collectively matches the segment variable S, and vice versa. Hence, S and (p* z) are equivalent.

Given a pattern with segment variables, we can replace each segment variable S with (p* z), as illustrated above. Hence proved. □

A.2 Proof of Proposition 2.2

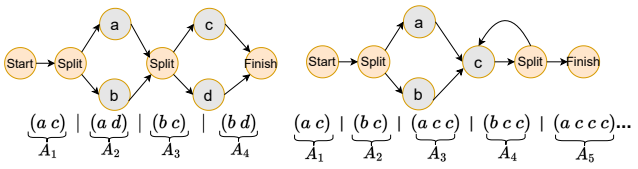
Proof. We prove this by showing that any pattern involving And or Not can be reduced to a pattern with only existing operators in standard MATCH_RECOGNIZE via induction. Without loss of generality, we assume a given pattern contains only point variables and no segment variables based on Proposition 2.1.

Let us first consider And operator: $(\mathcal{A}$ & $\mathcal{B})$. Assuming sub-pattern $\mathcal{A}$ and $\mathcal{B}$ are expressed using standard MATCH_RECOGNIZE, we will prove that $(\mathcal{A}$ & $\mathcal{B})$ can also be rewritten to a pattern expressible using standard MATCH_RECOGNIZE. To prove this, we first show some property of the sub-pattern as described in Lemma A.1.

Lemma A.1. *Given a pattern $\mathcal{A}$ in standard* MATCH_RECOGNIZE, *$\mathcal{A}$ can be reduced to a form of $(A_1$ | A_2 |...| $A_{l_A})$, where A_i is a special pattern containing only Concatenation operator $\forall 1 \le i \le l_A$.*

Proof. The rewriting procedure contains two steps: (1) construct an NFA based on pattern $\mathcal{A}$; (2) enumerate all possible paths from Start node to Finish node in this constructed transition graph via a DFS recursion: each path A_i generated refers to a special pattern with Concatenation operator only, i.e., $A_i = (a_i^1\ a_i^2 \cdots a_i^{l_i})$ where each a_i^x denotes a point variable $\forall 1 \le x \le l_i$. Hence, we have $\mathcal{A}$=$(A_1$ | A_2 |...| A_i |..). Next, we prove the number of enumerated paths is finite for any given pattern $\mathcal{A}$.

If pattern $\mathcal{A}$ contains no Kleene operator (e.g., Figure 13a), the transition graph is a directed acyclic graph (DAG) and the number of enumerated paths is finite; otherwise (e.g., Figure 13b), the transition graph contains cycles and the number of paths can be potentially infinite – however, we can safely terminate the enumeration process when the path length exceeds the series length n, since each point variable a_i^x in path A_i must match a distinct event and any path A_i with length greater than n will have no match. Hence, we can rewrite $\mathcal{A}$=$(A_1$ | A_2 |...| $A_{l_A})$ where $l_A \in \mathbb{N}^+$. □



(a) $\mathcal{A}$: ((a | b) (c | d)) **(b)** $\mathcal{A}$: ((a | b) c+)

Figure 13: Rewriting general pattern to special patterns

Based on Lemma A.1, $(\mathcal{A}$ & $\mathcal{B})$=$((A_1$ | A_2 |...| $A_{l_A})$ & $(B_1$ | B_2 |...| $B_{l_B}))$=$((A_1$ & $B_1)$ | $(A_1$ & $B_2)$ |...| $(A_{l_A}$ & $B_{l_B}))$, where A_i and B_j are both special patterns containing only Concatenation operator $\forall 1 \le i \le l_A, 1 \le j \le l_B$. Next, we rewrite $(A_i$ & $B_j)$=$((a_i^1\ a_i^2 \cdots a_i^{l_i})$ & $(b_j^1\ b_j^2 \cdots b_j^{l_j}))$. Since each a_i^x and b_j^y denote a point variable $\forall 1 \le x \le l_i, 1 \le y \le l_j$: if $l_i \ne l_j$, $(A_i$ & $B_j)$=$\emptyset$; else, $(A_i$ & $B_j)$=$(c_{ij}^1\ c_{ij}^2 \cdots c_{ij}^{l_{ij}})$, where $l_{ij} = l_i = l_j$ and c_{ij}^z denotes a point variable defined with Boolean conditions from both a_i^z and b_j^z $\forall 1 \le z \le l_{ij}$. Hence, $(\mathcal{A}$ & $\mathcal{B})$ is reduced to a pattern with only Concatenation and Alternation operator.

Similarly, given a pattern $(!\mathcal{A})$, assume sub-pattern $\mathcal{A}$ is expressed using standard MATCH_RECOGNIZE and we will prove $(!\mathcal{A})$ can also be rewritten to a pattern using standard MATCH_RECOGNIZE. Based on Lemma A.1, $(!\mathcal{A})$=$(!(A_1$ | A_2 |...| $A_{l_A}))$=$(!A_1$ & $!A_2$ &...& $!A_{l_A})$. For each A_i, $!A_i$=$!(a_i^1\ a_i^2 \cdots a_i^{l_i})$=$((c_i^1)$ | $(c_i^1\ c_i^2)$... |$(c_i^1\ c_i^2 \cdots c_i^{l_i})$ | ... | $(c_i^1\ c_i^2 \cdots c_i^n))$, where n is the series length, point variable c_i^z is defined as true $\forall z \ne l_i$, $c_i^{l_i}$ is defined as (a_i^1_Boolean_Condition(c_i^1) && a_i^2_Boolean_Condition(c_i^2)...&& $a_i^{l_i}$_Boolean_Condition($c_i^{l_i}$)) == false, and a_i^x_Boolean_Condition() denotes the Boolean condition associated with point variable a_i^x $\forall 1 \le x \le l_i$. The last equation is derived by enumerating all complementary patterns to $(a_i^1\ a_i^2 \cdots a_i^{l_i})$. Hence, each $!A_i$ is expressible in standard MATCH_RECOGNIZE. Combined with the above claim on $(\mathcal{A}$ & $\mathcal{B})$, $(!\mathcal{A})$ is rewritten to a pattern using standard MATCH_RECOGNIZE. □

B TRANSLATING MATCH_RECOGNIZE QUERY TO T-REX QUERY

In Section 2, we introduce an extension to MATCH_RECOGNIZE to enable the direct reasoning about query in terms of segment. While we hope users can write queries using the extended language directly, it is also possible to implement our extension as an intermediate representation (IR) and translate queries written with standard MATCH_RECOGNIZE into our IR. Firstly, T-ReX's language is an extension of MATCH_RECOGNIZE, and any pattern expressed in standard MATCH_RECOGNIZE can be executed directly in T-ReX. However, as pointed out in Section 2.3.2, alternative expression of the same pattern using T-ReX's language extension, i.e., segment variables and newly added operators, offers a few performance optimization opportunities. In the following, we present our initial trial of building a *rewriter* that rewrites a given MATCH_RECOGNIZE query into a T-ReX query via a *rule system* — this is similar to the rewriter [8] in SQL optimizer, which transforms a query tree using rules stored in the rule system [46]. Below we list a few rewrite rules and new rules can be extended easily. We omit the correctness proof of each rewrite rule, since they are rather straight-forward and not the focus of this paper. With these rules, the rewriter can then generate various query patterns in our IR. Ideally, a cost-based optimizer

shall pick the best rewritten pattern, similar to SQL's cost-based optimizer.

Rewrite Rule 1 (Convert point variable to segment variable). If x is a point variable with condition that is always true, then rewrite (x*) into (X) DEFINE SEGMENT X as true and replace all references to x in variable definitions with X.
If x is a point variable and its definition only involves time-related condition in the form of DEFINE x as (col - first(x.col) <= time_delta), then (1) rewrite (x*) into (X) with DEFINE SEGMENT X as window(col, 0, time_delta, unit) and replace all references to x with X; (2) rewrite (x+) into (X & W) with DEFINE SEGMENT X as window(col, 0, time_delta, unit), SEGMENT W as window(1, ∞) and replace all references to x with X.

Rewrite Rule 2 (Convert subset variable to segment variable). Consider a given pattern $\mathcal{P}$ and a subset variable U, let $\mathcal{A}$ be the minimal sub-pattern of $\mathcal{P}$ that contains all point variables from U. If sub-pattern $\mathcal{A}$ contains only point variables from U and only Concatenation and Kleene Closure operator, then rewrite sub-pattern $\mathcal{A}$ into ($\mathcal{A}$ & UU) DEFINE SEGMENT UU as true and replace all references to U in variable definitions with UU.

Rewrite Rule 3 (Reassign Boolean condition). Consider a (point or segment) variable z, normalize its Boolean condition into conjunctive normal form (CNF), i.e., (c_1 AND c_2 AND $\cdots$ AND c_l) where each clause c_i is a disjunction of literals $\forall 1 \leq i \leq l$. If clause c_i depends only on some other segment variable X, then remove clause c_i from z's Boolean condition and add c_i to X's Boolean condition via conjunction operator.

Rewrite Rule 4 (Decompose segment variables into finer granularity). If X is a segment variable and its definition contains a Boolean condition that can be decomposed into CNF with more than two clauses, i.e., (c_1 AND c_2 AND $\cdots$ AND c_l), then rewrite (X) into (X_1 & X_2 & $\cdots$ & X_l) DEFINE X_i as c_i $\forall 1 \leq i \leq l$.

Rewrite Rule 5 (Remove irrelevant variables). Consider a (point or segment) variable Z, if Z is defined as true, not referenced in other variables' DEFINE clause or in MEASURE clause, and there exists some sub-pattern $\mathcal{A}$: (1) connecting with Z via And operator, i.e., ($\mathcal{A}$ & Z), then rewrite ($\mathcal{A}$ & Z) into $\mathcal{A}$; (2) such that the given pattern is in the form of ($\mathcal{A}$ Z), then rewrite ($\mathcal{A}$ Z) into $\mathcal{A}$.

Next, we show how the rewriter applies these rules to generate patterns in our IR using the cold wave pattern example (Figure 2).

```
PATTERN (((A* D+ B*) & UU) Z)
DEFINE D AS tstamp - first(D.tstamp) <= INTERVAL '5' DAY,
       Z AS last(UU.tstamp) - first(UU.tstamp) BETWEEN
                INTERVAL '25' DAY AND INTERVAL '30' DAY
            AND mann_kendall_test(UU.temp) >= 3.0
            AND linear_regression_r2(D.tstamp, D.temp) >= 0.95
            AND last(D.temp) - first(D.temp) < -20,
       SEGMENT UU AS true
```

Figure 14: Pattern after rewrite step-1.

Example 3. [Rewrite Step-1]: given pattern (A* D+ B* Z) in Figure 2, both rewrite rule 1 and 2 can get fired. The rewriter will apply rule 1 and rule 2 separately, and obtain two alternative patterns. In the following, we only illustrate one branch where rule 2 is first applied. Based on rule 2, (A* D+ B* Z) is rewritten into (((A* D+ B*) & UU) Z) as shown in Figure 14.

```
PATTERN (((AA (DD & WW) BB) & UU) Z)
DEFINE SEGMENT DD AS window(DD.tstamp, 0, 5, DAY),
       SEGMENT WW AS window(1, ∞),
       Z AS last(UU.tstamp) - first(UU.tstamp) BETWEEN
                INTERVAL '25' DAY AND INTERVAL '30' DAY
            AND mann_kendall_test(UU.temp) >= 3.0
            AND linear_regression_r2(DD.tstamp, DD.temp) >= 0.95
            AND last(DD.temp) - first(DD.temp) < -20,
```

Figure 15: Pattern after rewrite step-2.

[Rewrite Step-2]: after step-1, only rule 1 can be triggered. Applying rewrite rule 1, (((A* D+ B*) & UU) Z) is rewritten into (((AA (DD & WW) BB) & UU) Z) as depicted in Figure 15, where segment variable AA, BB and UU's definition defaults to true.

```
PATTERN ((AA (DD & WW) BB) & UU) Z)
DEFINE SEGMENT DD AS linear_regression_r2(DD.tstamp, DD.temp) >= 0.95
            AND last(DD.temp) - first(DD.temp) < -20
            AND window(DD.tstamp, 0, 5, DAY),
SEGMENT WW AS window(1, ∞),
SEGMENT UU AS mann_kendall_test(UU.temp) >= 3.0,
            AND window(UU.tstamp, 25, 30, DAY),
Z AS true
```

Figure 16: Pattern after rewrite step-3.

[Rewrite Step-3]: after step-2, rule 3 gets fired. The Boolean condition in variable Z is already in CNF form. Applying rule 3, clauses are removed from Z's Boolean condition and added to DD and UU's definition respectively as illustrated in Figure 16.

```
PATTERN ((AA (DD1 & DD2 & DD3 & WW) BB) & UU1 & UU2) Z)
DEFINE SEGMENT DD1 AS linear_regression_r2(DD.tstamp, DD.temp) >= 0.95,
SEGMENT DD2 AS last(DD.temp) - first(DD.temp) < -20,
SEGMENT DD3 AS window(DD.tstamp, 0, 5, DAY),
SEGMENT WW AS window(1, ∞)
SEGMENT UU1 AS mann_kendall_test(UU.temp) >= 3.0,
SEGMENT UU2 AS window(UU.tstamp, 25, 30, DAY),
Z AS true
```

Figure 17: Pattern after rewrite step-4.

[Rewrite Step-4]: next, rewrite rule 4 gets fired. The Boolean condition associated with segment variable DD are decomposed into three clauses and thus we can rewrite DD into (DD1 & DD2 & DD3) as shown in Figure 17. Similarly, UU is decomposed into (UU1 & UU2).
[Rewrite Step-5]: lastly, rewrite rule 5 gets fired, removing irrelevant variable Z.

Comparing the automatically translated pattern in Figure 18 with our hand-written pattern in Figure 3, segment variable AA and BB in Figure 18 correspond to W in Figure 3; DD1 and DD2 correspond to DOWN and DIFF respectively; DD3 and WW together correspond to

```
PATTERN (AA (DD1 & DD2 & DD3 & WW) BB) & UU1 & UU2)
DEFINE  SEGMENT DD1  AS linear_regression_r2(DD.tstamp, DD.temp) >= 0.95,
 SEGMENT DD2  AS last(DD.temp) - first(DD.temp) < -20,
 SEGMENT DD3  AS window(DD.tstamp, 0, 5, DAY),
 SEGMENT WW  AS window(1, ∞)
 SEGMENT UU1  AS mann_kendall_test(UU.temp) >= 3.0,
 SEGMENT UU2  AS window(UU.tstamp, 25, 30, DAY),
```

Figure 18: Pattern after rewrite step-5.

WAVE_WINDOW[9]; UU1 and UU2 correspond to UP and OVERALL_WINDOW respectively. We note Figure 18 only depicts one possible rewritten pattern in our IR, obtained by applying the rewrite rules in a particular order, and there exists other alternative rewritten patterns in our IR. A cost-based optimizer can be employed to pick the best rewritten pattern and is left as future work. From our lessons working with different patterns, our intuition is that patterns expressed using segment variable when dependent conditions exist across points, using finer granular variables, and without Kleene Closure operator in general have better performance.

C REASONING OF COST MODEL

C.1 Estimating ℓ_{se} in RIGHTPROBECONCAT

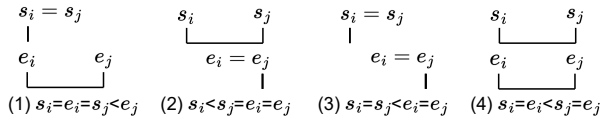


Figure 19: 4 Cases of any Physical Operator's Search Space.

Based on all the physical operators in Table 1, if the search space of the root operator is $(S = [0, n-1], E = [0, n-1])$, which is always the case under MATCH_RECOGNIZE, then the search space $(S = [s_i, s_j], E = [e_i, e_j])$ of any physical operator in this tree falls into one of the following 4 cases as depicted in Figure 19:

(1) $s_i = e_i = s_j < e_j$
(2) $s_i < s_j = e_i = e_j$
(3) $s_i = s_j < e_i = e_j$
(4) $s_i = e_i < s_j = e_j$

Recall that $\ell_{se} = e_j - s_i$, $\ell_s = s_j - s_i$, and $\ell_e = e_j - e_i$. Now because we do not have access to the actual segments during query optimization, we can only estimate ℓ_{se}. We are given the search space sizes of the start and end positions, ℓ_s and ℓ_e, respectively, of the concatenated segment. So, we estimate ℓ_{se} as follows:

- If $\ell_s = 1$ and $\ell_e = 1$ (Case (3)), then we know there is only one choice for the start position (i.e., s_i) and end position (i.e., e_i) of the concatenated segment. We don't actually know what the choice is as we don't have access to the data, so we use uniform assumption: consider drawing two integers x and y from the integer range $[1, n]$ and $x <= y$, the expected value of $|x - y|$ is $n/3$. Therefore, we use $n/3$, where n is the number of points in the series, or series length.
- If $\ell_s > 1$ or $\ell_e > 1$, then it is one of Cases (1), (2), and (4):
  - Case (1): $\ell_{se} = e_j - e_i = \ell_e$
  - Case (2): $\ell_{se} = s_j - s_i = \ell_s$
  - Case (4): $\ell_{se} = s_j - s_i = \ell_s = e_j - e_i = \ell_e$

  Thus we can use $\ell_{se} = max(l_s, l_e)$.

[9]When we hand write the pattern in Figure 3, DD3 and WW are combined to form WAVE_WINDOW since we know each point corresponds to 1 DAY in this dataset.

C.2 Cardinality Estimator for Not Operator

Not operator outputs the complement of the sub-tree's segments in the search space. The search space's cardinality is estimated as $\ell_s\ell_e \times Sel_w$, where Sel_w denotes the selectivity of the embedded window condition w. The embedded window condition is assigned during plan rewrite as described in Section 3. Using a query pattern ((~A) & W) as an example, W is a window condition. The original logical query plan is as shown in the left-hand side of Figure 20(a); through rewrite, W gets "fused" with Not and pushed down to A, and the plan becomes the right-hand side of Figure 20(a). The output segment has to satisfy this window condition, hence the output cardinality is estimated as $\ell_s\ell_e \times Sel_w - \mathbb{C}_{in}$. If we want output segments that do not satisfy both A and the window condition, we can write the query pattern as (~(A & W)), and the plan after rewrite is as shown in the right-hand side of Figure 20(b). There would be no embedded window condition on the Not with $Sel_w = 1$ and the output cardinality estimate equals $\ell_s\ell_e - \mathbb{C}_{in}$.

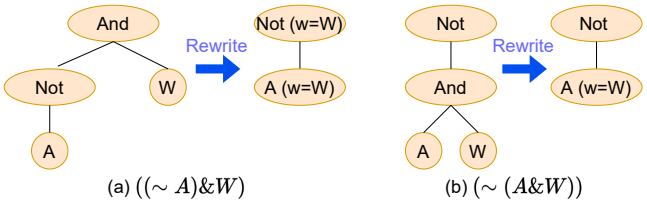


Figure 20: Not Operator with/without Embedded Window.

D STATISTICS COLLECTION

A few building blocks in the cost model require instantiating: the operator cost function (f_{op}), the aggregate's indexing cost function (f_{ind}), and the aggregate's evaluation cost functions with and without index ($f_{\delta'}$ and f_{δ}, respectively). T-REX bootstraps these parameters in an offline profiling procedure, while other data-dependent statistics like $\text{Sel}_{P|w}$ are sampled at query time.

D.1 Offline Profiling for the Cost of Operator

For each physical operator op in Table 1, T-REX estimates its w in f_{op} (Equation 1) during an offline profiling procedure, which could be run just before code release or during the software's installation for better platform-specific tuning. Specifically, T-REX first generates synthetic segments for each operator with varying $\mathbb{C}_{in}$ for unary operators; $\mathbb{C}_l$ and $\mathbb{C}_r$ for binary operators. Next, T-REX evaluates each operator on its synthetic segments, counts the output cardinality $\mathbb{C}_{out}$, and measures the run-time t_{op} (in nanosecond) spent inside this operator's *eval()* method. With these data points, i.e., $(\mathbb{C}_{in}, \mathbb{C}_{out}, t_{op})$ for unary operator and $(\mathbb{C}_l, \mathbb{C}_r, \mathbb{C}_{out}, t_{op})$ for

binary operator, T-ReX then estimates w for each f_{op} using linear regression. Table 5 depicts each w in f_{op} for different physical operators obtained from our offline profiling.

$$f_{op} = \begin{cases} w \cdot (\mathbb{C}_{in} + \mathbb{C}_{out}), & \text{if op is Unary} \\ w \cdot (\mathbb{C}_{r} + \mathbb{C}_{out}), & \text{if op is LeftProbe} \\ w \cdot (\mathbb{C}_{l} + \mathbb{C}_{out}), & \text{if op is RightProbe} \\ w \cdot (\mathbb{C}_{l} + \mathbb{C}_{r} + \mathbb{C}_{out}), & \text{otherwise} \end{cases} \quad (1)$$

Physical Operator	w	Physical Operator	w
SegGenWindow	193	MaterializeNot	440
SegGenFilter	502	ProbeNot	2168
SegGenIndexing	501	MaterializeKleene	1577
SortMergeConcat	671	SortMergeAnd	588
RightProbeConcat	1583	LeftProbeAnd	2077
LeftProbeConcat	1583	RightProbeAnd	2077
SortMergeOr	747		

Table 5: w in f_{op} Obtained In Our Offline Profiling.

D.2 Offline Profiling for the Cost of Evaluating Aggregates

T-ReX implements three cost functions: *(a)* constant cost function; *(b)* linear cost function; and *(c)* quadratic cost function. For instance, given a specific aggregate function F, f_{ind} can be expressed as Equation 2.

$$f_{ind}(F, \ell_{se}) = \begin{cases} w, & \text{if } F \text{ conforms constant cost model} \\ w \cdot \ell_{se}, & \text{if } F \text{ conforms linear cost model} \\ w \cdot \ell_{se}^2, & \text{if } F \text{ conforms quadratic cost model} \end{cases} \quad (2)$$

Similarly for $f_{\delta}(F, \ell_{in})$ and $f_{\delta'}(F, \ell_{in})$. During offline profiling, we generate synthetic segments with varying ℓ_{se} and ℓ_{in}, measure the indexing time (t_{ind}) and matching time (t_{δ} and $t_{\delta'}$), and then profile w according to this primitive's cost function. Table 6 depicts some example w in f_{ind}, $f_{\delta'}$, and f_{δ} of our built-in aggregate functions. For user defined primitives, we rely on users to provide annotations for a proper cost function, otherwise, we will use linear cost function by default. We perform profiling when users register their defined primitives.

Aggregate	$w(f_{ind})$	$w(f_{\delta'})$	$w(f_{\delta})$
`linear_regression_r2(x, y)`	380 (L)	0 (C)	903 (L)
`mann_kandall_test(x)`	761 (Q)	0 (C)	99 (Q)
`ZScoreOutlier(`ℓ`)`	-	-	34 (L)

Table 6: w in f_{ind}, f_{δ}, and $f_{\delta'}$ Obtained In Our Offline Profiling. (C), (L), and (Q) denote constant, linear, and quadratic cost model, respectively.

D.3 Online Profiling for Estimating $\text{Sel}_{P|w}$

Unlike w in cost functions, primitives' selectivity is almost always data-dependent. T-ReX estimates the selectivity ($\text{Sel}_{P|w}$) based on samples at query time.

Online latency can hurt user experience, thus T-ReX estimates variables' selectivity on a sampled set of series, denoted as $\mathbb{X}_S$. If the aggregate involved in a variable's condition P does not implements an `index()` method, T-ReX estimates $\text{Sel}_{P|w}$ by *(1).* randomly sampling segments from the windowed search space, denoted as O; *(2).* counting the number of segments that passes the variable's `match` method, denoted as $|\widetilde{O}|$; *(3).* estimating $\text{Sel}_{P|w}$ as $\frac{|\widetilde{O}|}{|O|}$. On the other hand, if the aggregate involved in a variable's condition P implements `index()`, T-ReX first performs indexing and then conduct step (1-3) above, but use the index's `match()` method instead, which likely reduces the overall cost. Based on experiments, the online sampling cost is negligible compared to the overall query processing cost. Another statistics from sampling is the estimated average segment length ℓ_{in} as $avg_{o \in O}|o|$.

E QUERIES USED IN EXPERIMENTS

```
-- Query name: V-shape (v_shape)
-- Schema: [tstamp: int, ticker: string, price: float]
PARTITION BY ticker
ORDER BY tstamp
PATTERN (((DN & W) (UP & W)) & WINDOW)
DEFINE
SEGMENT W AS window(15, null), -- window at least 15 points.
SEGMENT DN AS
    linear_reg_r2_signed(DN.tstamp, DN.temp) <= down_r2_max,
SEGMENT UP AS linear_reg_r2_signed(UP.tstamp, UP.temp) >= up_r2_min,
SEGMENT WINDOW AS window(1, total_window_size)
-- Parameters:
-- down_r2_max = [-0.7]
-- up_r2_min = [0.7, 0.9, 1.0]
-- total_window_size = [30, 60, 90]
```

```
-- Query name:  Header and Shoulder (head_shldr)
-- Schema: [tstamp: int, ticker: string, price: float]
PARTITION BY ticker
ORDER BY tstamp
PATTERN (((UP1 & W)
    ((DN1 & W) (UP2 & W & NECK_TO_HEAD)) & SHLDER_TO_HEAD
    ((DN2 & W & HEAD_TO_NECK) (UP3 & W)) & HEAD_TO_SHLDER
    (DN3 & W)) & WINDOW)
DEFINE
SEGMENT W AS window(3, 10),
SEGMENT DN1 AS linear_reg_r2_signed(DN1.tstamp, DN1.temp) <= -t,
SEGMENT DN2 AS linear_reg_r2_signed(DN2.tstamp, DN2.temp) <= -t,
SEGMENT DN3 AS linear_reg_r2_signed(DN3.tstamp, DN3.temp) <= -t,
SEGMENT UP1 AS linear_reg_r2_signed(UP1.tstamp, UP1.temp) >= t,
SEGMENT UP2 AS linear_reg_r2_signed(UP2.tstamp, UP2.temp) >= t,
SEGMENT UP3 AS linear_reg_r2_signed(UP3.tstamp, UP3.temp) >= t,
SEGMENT NECK_TO_HEAD AS
    last(NECK_TO_HEAD.price) / first(NECK_TO_HEAD.price) > r1,
SEGMENT HEAD_TO_NECK AS
    first(HEAD_TO_NECK.price) / last(HEAD_TO_NECK.price) > r1,
SEGMENT SHLDER_TO_HEAD AS
    last(SHLDER_TO_HEAD.price) / first(SHLDER_TO_HEAD.price) > r2,
SEGMENT HEAD_TO_SHLDER AS
    first(HEAD_TO_SHLDER.price) / last(HEAD_TO_SHLDER.price) > r2,
SEGMENT WINDOW AS window(1, total_window_size)
-- Parameters:
-- t = [0.7]
-- total_window_size = [40, 60, 80]
-- r1 (head_to_neck_ratio) = [1.1, 1.15]
-- r2 (head_to_shoulder_ratio) = [1.0, 1.05, 1.11]
```

```
-- Query name:  Outlier (outlier)
-- Schema: [tstamp: int, ticker: string, price: float]
PARTITION BY ticker
ORDER BY tstamp
PATTERN ((UP1 OUTLIER UP2) & WINDOW)
DEFINE
OUTLIER AS ZScoreOutlier(outlier_context_size) > z_score_min,
```

```
SEGMENT UP1 AS
    linear_reg_r2_signed(UP1.tstamp, UP1.temp) >= up_r2_min,
SEGMENT UP2 AS
    linear_reg_r2_signed(UP2.tstamp, UP2.temp) >= up_r2_min,
SEGMENT WINDOW AS window(1, total_window_size)
-- Parameters:
-- up_r2_min = [0.7]
-- total_window_size = [30]
-- outlier_context_size = [15, 20, 25]
-- z_score_min = [2.61, 2.63, 2.65]
```

```
-- Query name:  Rebound (rebound)
-- Schema [tstamp: int, county: string, confirmed: int]
PARTITION BY county
ORDER BY tstamp
PATTERN (UP1 ((DOWN & FALL) UP2) & RISE) & WINDOW)
DEFINE
SEGMENT FALL AS
    last(FALL.confirmed) / first(FALL.confirmed) < fall_ratio,
SEGMENT RISE AS
    last(RISE.confirmed) / first(RISE.confirmed) > rise_ratio,
SEGMENT UP1 AS linear_reg_r2_signed(UP1.tstamp, UP1.confirmed) >= t,
SEGMENT UP2 AS linear_reg_r2_signed(UP2.tstamp, UP2.confirmed) >= t,
SEGMENT DOWN AS
    linear_reg_r2_signed(DOWN.tstamp, DOWN.confirmed) <= -t,
SEGMENT WINDOW AS window(0, 60)
-- Parameters:
-- t = [0.7]
-- fall_ratio = [0.4, 0.6, 0.8]
-- rise_ratio = [3, 4, 5]
```

```
-- Query name:  Cold Wave (cld_wave)
-- Schema [tstamp: int, city: string, temp: float]
PARTITION BY city
ORDER BY tstamp
PATTERN ((W1 (DOWN & FALL & W2) W1) & UP_MK & WINDOW)
DEFINE SEGMENT W1 AS true,
SEGMENT W2 AS window(1, 5),
SEGMENT FALL AS last(FALL.temp) - first(FALL.temp) < -fall_diff,
SEGMENT DOWN AS
    linear_reg_r2_signed(DOWN.tstamp, DOWN.temp) <= -down_r2_min,
SEGMENT WINDOW AS window(25, 30),
SEGMENT UP_MK as mann_kendall_test(temp) >= 3.0
-- Parameters:
-- fall_diff = [16, 18, 20]
-- down_r2_min = [0.85, 0.9, 0.95]
```

```
-- Query name:  Repeated Pattern (rptd_pttrn)
-- Schema [tstamp: int, rides: int]
ORDER BY tstamp
PATTERN (((W1 (UP & RISE & W2) W3 (DOWN & FALL & W2) W1)
    & WINDOW){k}
DEFINE SEGMENT W1 AS true,
SEGMENT W2 AS window(20),
SEGMENT W3 AS window(4),
SEGMENT WINDOW AS window(48),
SEGMENT UP AS linear_reg_r2_signed(UP.tstamp, UP.rides) >= t,
SEGMENT DOWN AS linear_reg_r2_signed(DOWN.tstamp, DOWN.rides) <= -t,
SEGMENT FALL AS
    last(FALL.confirmed) / first(FALL.confirmed) < 1/rise_ratio,
SEGMENT RISE AS
    last(RISE.confirmed) / first(RISE.confirmed) > rise_ratio
-- Parameters:
-- t = [0.7]
-- rise_ratio = [3, 4, 5]
-- k = [1, 2, 3, 4]
```

```
-- Query name:  Limit Sell (limit_sell)
-- Schema: [tstamp: int, ticker: string, price: float]
PARTITION BY ticker
ORDER BY tstamp
PATTERN (RISE & WINDOW & ~(FALL W))
DEFINE SEGMENT W AS true,
SEGMENT RISE AS last(RISE.price) / first(RISE.price) > rise_ratio,
SEGMENT WINDOW AS window(1, total_window_size),
SEGMENT FALL AS last(FALL.price) / first(FALL.price) < fall_ratio
-- Parameters:
-- rise_ratio = [2.0]
-- fall_ratio = [0.7, 0.8, 0.9]
-- total_window_size = [15, 30, 60]
```

```
-- Query name:  OpenCEP Q1 (OpenCEP_Q1)
-- Schema [tstamp: int, ticker: string, open: float,
--         peak: float, close: float]
ORDER BY tstamp
PATTERN ((A1 W (A2 & INC1) W (A3 & INC2)) & WINDOW)
DEFINE SEGMENT W AS true,
A1 AS A1.ticker = a,
A2 AS A2.ticker = a,
A3 AS A3.ticker = a,
INC1 AS INC1.peak > A1.peak,
INC2 AS INC2.peak > A2.peak,
SEGMENT WINDOW AS window(0, total_window_size, MINUTE)
-- Parameters:
-- a = ["GOOG"]
-- total_window_size = [5, 20, 40, 60, 80, 100, 120]
```

```
-- Query name:  OpenCEP Q2 (OpenCEP_Q2)
-- Schema [tstamp: int, ticker: string, open: float, peak: float]
ORDER BY tstamp
PATTERN ((((A1 W A2) & FALL)+) & WINDOW)
DEFINE SEGMENT W AS true,
A1 AS A1.ticker = a,
A2 AS A2.ticker = a,
SEGMENT FALL AS last(FALL.peak) < first(FALL.peak),
SEGMENT WINDOW AS window(0, total_window_size, MINUTE),
-- Parameters:
-- a = ["GOOG"]
-- total_window_size = [5, 20, 40, 60, 80, 100, 120]
```

```
-- Query name:  AFA Q1 (AFA_Q1)
-- Schema: [tstamp: int, ticker: string, price: float]
ORDER BY tstamp
PATTERN (((LARGE_FALL & W) (((FALL & W)+ (RISE & W)+){k})
    & EQ_FALL_AND_RISE) & WINDOW)
DEFINE
SEGMENT W AS window(2),
SEGMENT LARGE_FALL AS
    last(LARGE_FALL.price) / first(LARGE_FALL.price)
    < large_fall_ratio,
SEGMENT FALL AS last(FALL.price) < first(FALL.price),
SEGMENT RISE AS last(RISE.price) > first(RISE.price),
SEGMENT EQ_FALL_AND_RISE AS EqualUpDownTicks(EQ.price),
SEGMENT WINDOW AS window(0, 30)
-- Parameters:
-- K = [5]
-- large_fall_ratio = [0.925, 0.9, 0.875, 0.85, 0.825,
--                     0.8, 0.775, 0.75, 0.725]
```

```
-- Query name:  AFA Q2 (AFA_Q2)
-- Schema: [tstamp: int, ticker: string, price: float]
ORDER BY tstamp
PATTERN (((LARGE_FALL & W) ((FALL & W)+ (RISE & W)+)+)
    & RECOVER & WINDOW)
DEFINE
SEGMENT W AS window(2),
SEGMENT LARGE_FALL AS
    last(LARGE_FALL.price) / first(LARGE_FALL.price)
    < large_fall_ratio,
SEGMENT FALL AS last(FALL.price) < first(FALL.price),
SEGMENT RISE AS last(FALL.price) > first(FALL.price),
SEGMENT RECOVER AS last(FALL.price) >= first(FALL.price),
SEGMENT WINDOW AS window(0, 30)
-- Parameters:
```

	NDCG Score			Median Stats Collection (ms)		
Queries	5	50	500	5	50	500
v_shape	0.92	0.92	0.93	0.07	0.55	4.96
head_shldr	1.00	0.99	0.99	0.16	1.56	13.46
outlier	0.95	0.95	0.95	0.08	0.67	6.20
limit_sell	0.95	0.95	0.94	0.01	0.08	0.68
rebound	1.00	1.00	1.00	0.07	0.73	6.76
cld_wave	0.96	0.93	0.93	1.02	7.20	6.98
rptd_pttrn	0.97	0.97	0.97	0.01	0.01	0.01
OpenCEP_Q1	0.74	0.74	0.74	0.00	0.00	0.00
OpenCEP_Q2	0.99	0.99	0.99	0.02	0.02	0.02
AFA_Q1	0.84	0.97	0.98	0.01	0.13	1.09
AFA_Q2	0.84	0.97	0.98	0.01	0.06	0.57

Table 7: NDCG scores and median statistics collection times grouped by series sampled (5, 50, and 500). For datasets with less than 500 series, the sample size "tops-out" at the number of available series.

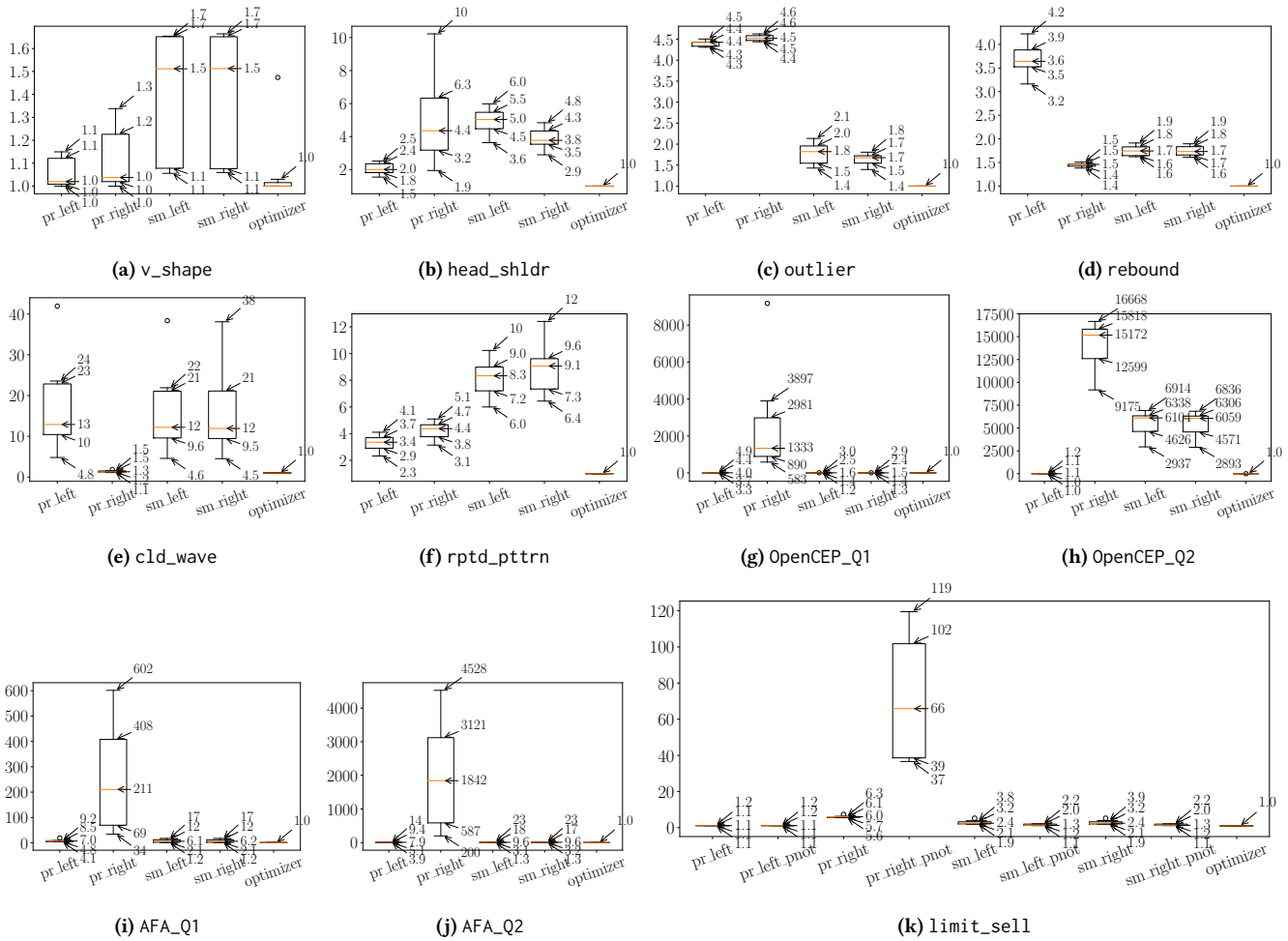
```
-- large_fall_ratio = [0.925, 0.9, 0.875, 0.85, 0.825,
--                     0.8, 0.775, 0.75, 0.725]
```

F DETAILED EXPERIMENTAL RESULTS

Figure 21 depicts the detailed comparison between T-ReX optimizer and some baselines using rules. It is easy to see T-ReX optimizer returns consistently better plan than rule-based baselines across different pattern templates and query instances. No single baseline performs well across all queries, e.g., `pr_left` has low latency in `OpenCEP_Q2`, `limit_sell`, and `v_shape`, but not in `outlier`, `rebound`, and `cld_wave`.

Figure 22a presents a summarized view of Figure 12 for each pattern template, showing the median speedup of T-ReX over each baseline with pre-computation enabled for aggregate functions. In summary, T-ReX achieves a median speedup of 3.9×, 4×, and 147× over T-ReX Batch, which approximates the behavior of ZStream [41], AFA [28], and OpenCEP [20].

Figure 22b depicts the median speedups of enabling pre-computation over disabling pre-computation for aggregate functions. In general, enabling pre-computation is beneficial for most queries. Two exceptions occur for `cld_wave` and `AFA_Q1`, AFA performs worse when using aggregate function pre-computation. This is because pre-computating Mann-Kendall test on the whole series incurs unnecessary computation, compared to independent computation on necessary segments. We note that T-ReX optimizer successfully chooses to not use pre-computation for these two queries, i.e., `cld_wave` and `AFA_Q1`, even when allowing pre-computation. This showcases the effectiveness of T-ReX optimizer and the need for such an optimizer to determine whether aggregate pre-computation shall be used or not.

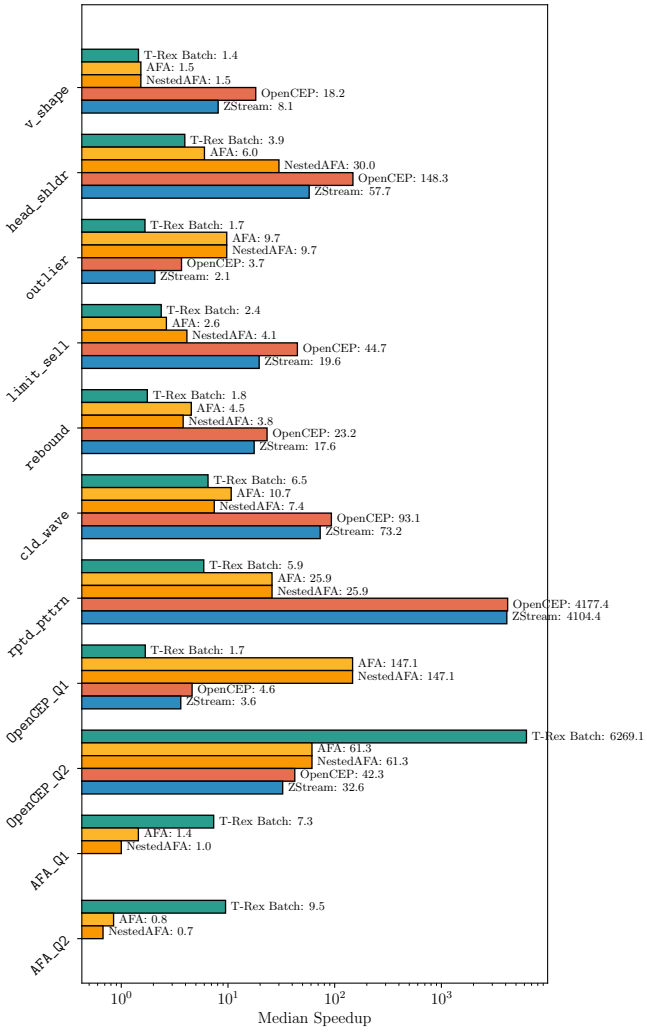


(a) v_shape **(b)** head_shldr **(c)** outlier **(d)** rebound

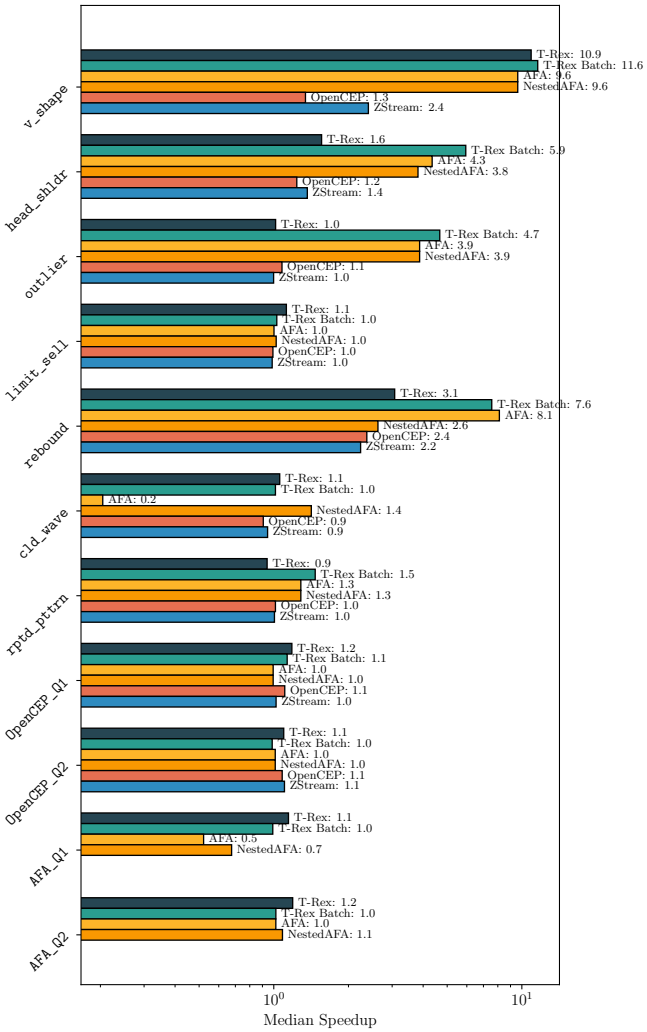
(e) cld_wave **(f)** rptd_pttrn **(g)** OpenCEP_Q1 **(h)** OpenCEP_Q2

(i) AFA_Q1 **(j)** AFA_Q2 **(k)** limit_sell

Figure 21: Slow-Down Ratios Over the Fastest Method.

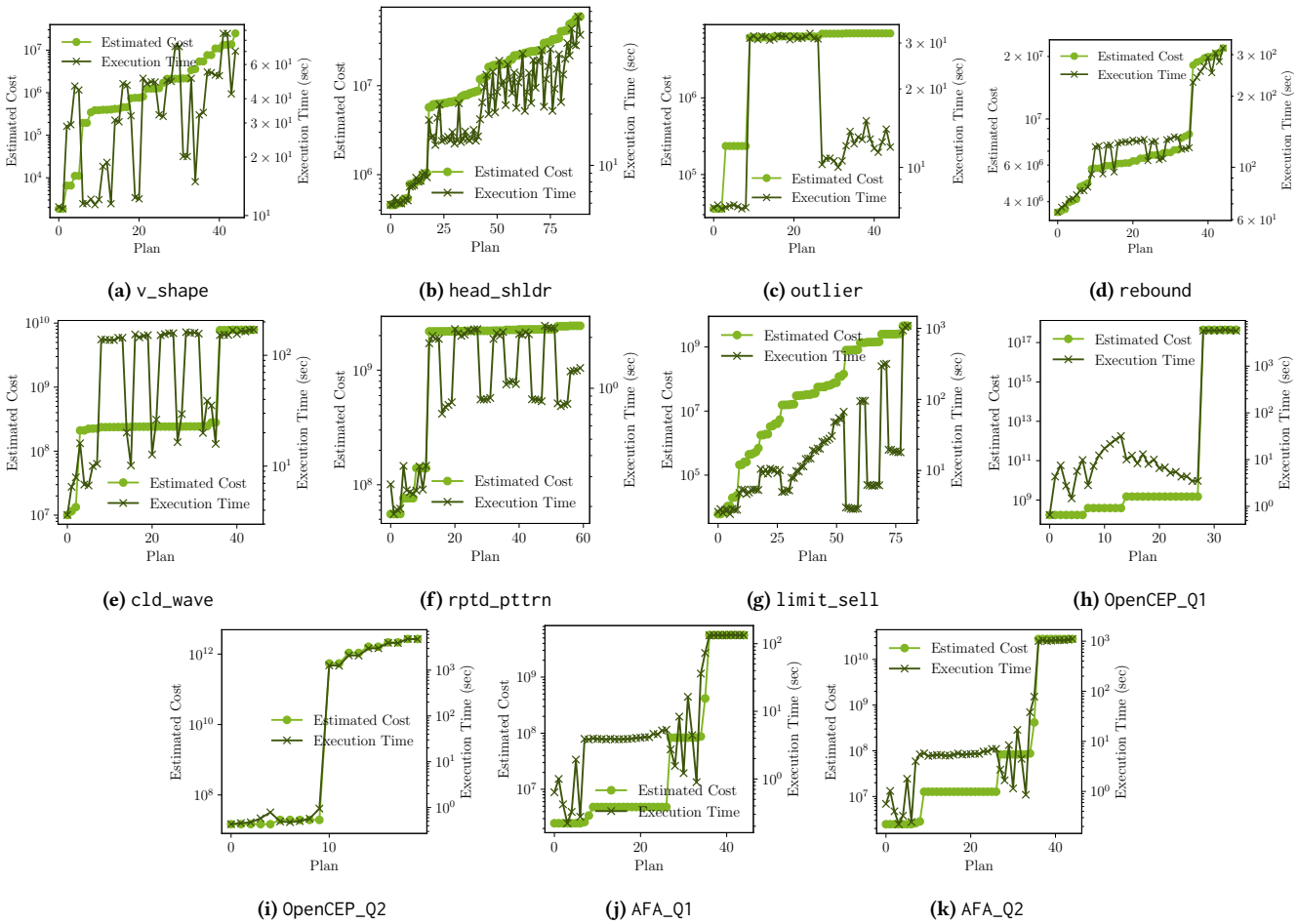


(a) Median speedups of T-ReX over baselines with aggregate function pre-computation. OpenCEP and ZStream do not support nested Kleene Closure in AFA_Q1 and AFA_Q2.



(b) Median speedups of T-ReX and baselines over each of themselves resulted from using pre-computation for aggregate functions.

Figure 22: Median Speedups



(a) v_shape **(b)** head_shldr **(c)** outlier **(d)** rebound

(e) cld_wave **(f)** rptd_pttrn **(g)** limit_sell **(h)** OpenCEP_Q1

(i) OpenCEP_Q2 **(j)** AFA_Q1 **(k)** AFA_Q2

Figure 23: T-REX **Estimated Cost Versus Actual Execution Time (Series Sampled for Estimation: 5).**