High-Performance Row Pattern Recognition Using Joins (Technical Report)

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

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

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

ABSTRACT

The SQL standard introduced MATCH_RECOGNIZE in 2016 for row pattern recognition. Since then, MATCH_RECOGNIZE has been supported by several leading relation systems, they implemented this function using Non-Deterministic Finite Automaton (NFA). While NFA is suitable for pattern recognition in streaming scenarios, the current uses of NFA by the relational systems for historical data analysis scenarios overlook important optimization opportunities. We propose a new approach to use Join to speed up row pattern recognition in historical analysis scenarios for relational systems. Implemented as a logical plan rewrite rule, the new approach first filters the input relation to MATCH_RECOGNIZE using Joins constructed based on a subset of symbols taken from the PATTERN expression, then run the NFA-based MATCH_RECOGNIZE on the filtered rows, reducing the net cost. The rule also includes a specialized cardinality model for the Joins and a cost model for the NFA-based MATCH_RECOGNIZE operator for choosing an appropriate symbol set. The rewrite rule is applicable when the query pattern's definition is self-contained and either the input table has no duplicates or there is a window condition. Applying the rewrite rule to a query benchmark with 1,800 queries spanning over 6 patterns and 3 pattern definitions, we observed median speedups of 5.4× on Trino (v373 with ORC files on Hive), 57.5× on SQL Server (2019) using column store and 41.6× on row store.


```
SELECT * FROM Crimes MATCH_RECOGNIZE (
    ORDER BY datetime
    MEASURES R.id AS RID, B.id AS BID,M.id AS MID,count(Z.id) AS GAP
    ONE ROW PER MATCH
    AFTER MATCH SKIP TO NEXT ROW
    PATTERN (R Z* B Z* M)
    DEFINE R AS R.primary_type = 'ROBBERY',
        B AS B.primary_type = 'BATTERY'
            AND B.lon BETWEEN R.lon - 0.05 AND R.lon + 0.05
            AND B.lat BETWEEN R.lat - 0.02 AND R.lat + 0.02,
        M AS M.primary_type = 'MOTOR VEHICLE THEFT'
            AND M.lon BETWEEN R.lon - 0.05 AND R.lon + 0.05
            AND M.lat BETWEEN R.lat - 0.02 AND R.lat + 0.02
            AND M.datetime - R.datetime <= INTERVAL '30' MINUTE)
```

Figure 1: A MATCH_RECOGNIZE query on Chicago Crimes data set looking for potentially related sequences of crimes.

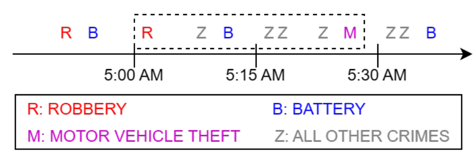


Figure 2: A match (dashed box) for the pattern (R Z* B Z* M) found in the sequence of crimes reports ordered by datetime.

Example 1. Figure 1 is a query on the Chicago Crimes data set [12]. Each row is a crime report and this query detects sequences of possibly related crimes ordered by datetime. Specifically, the pattern refers to three ordered instances of "ROBBERY", "BATTERY", and "MOTOR VEHICLE THEFT" occurred within 30 minutes in the same latitude-longitude "box" centered at the location of "ROBBERY". The pattern is expressed using a regular expression style notation in the PATTERN clause, which composes symbols (e.g., R, B) in sequential order. Each symbol is defined as a set of Boolean conditions through the DEFINE clause specifying when a row can be matched to the symbol (Z is undefined thus matches any row). Each of R, B, and M matches exactly one row, and Z* is a Kleene Star matching 0 or more rows, indicating there may be other crimes between the crimes of interest. A window of 30 minutes is also defined inside the DEFINE clause. The AFTER MATCH SKIP clause determines the starting row to resume pattern matching after a non-empty match has been found. Figure 2 illustrates a match of this pattern. For further discussion about syntax and semantics please refer to Appendix A.1.

1 INTRODUCTION

In relational systems, a row pattern recognition task is to detect a sequence of ordered rows from an input table that match a user-specified pattern. For example, a financial service provider needs to identify sequences of suspicious transactions that match known patterns of criminal activities; an e-commerce site analyzes the steps taken by customers from landing through a social media referrer to a successful purchase [18].

In response to the increasing importance of row pattern recognition, MATCH_RECOGNIZE was added to the official SQL standard [26] to perform these tasks using a declarative interface and avoid exporting data to external programs. Oracle, Apache Flink, Azure Streaming Analytics, Snowflake and Trino have already announced support for MATCH_RECOGNIZE [27, 28, 38, 47, 53].

There are two scenarios of row pattern recognition: streaming and historical analysis. For the streaming scenario, the input table

* Equal contributors.

The published paper is in Proceedings of the VLDB Endowment, Vol. 16, No. 5 .
https://www.vldb.org/pvldb/vol16/p1181-zhu.pdf

is an event stream, and the queries emit results in real-time when specified patterns are detected. Streaming systems that support MATCH_RECOGNIZE use executors based on the Non-deterministic Finite Automaton (NFA), which compiles a query into a directed state-transition graph and identifies ordered sequences of events that match any path from start to end of the graph (e.g., R $\longrightarrow$ B $\longrightarrow$ M) while consuming the events sequentially. Notable examples of streaming systems supporting MATCH_RECOGNIZE include Apache Flink [8] and Azure Streaming Analytics [40].

Relational systems for historical analysis also use NFA to implement MATCH_RECOGNIZE, as in Trino [56]. NFA works well in streaming but the current NFA-based MATCH_RECOGNIZE implementations ignore several optimization opportunities in historical analysis setting, namely: (1) flexible order of execution, (2) availability of indexes and (3) operator-level parallelism. The following example is a case in which these opportunities can be taken.

Example 2. To execute the query in Figure 1, NFA consumes the ordered input rows one-by-one and attempts to match the symbol R. If successful then it moves to Z*, followed by B, Z* and lastly M. Suppose 5M rows match R, 100 rows match B, and 50 row sequences match pattern (R Z* B), NFA will over its life cycle incur 5M incomplete matching states started by R and checked for Z*. With at most 50 of them matching (R Z* B), most of the computation is wasted. Because all rows are available, there exists another way, in which, first, we find rows matching B's primary_type, then check for R and M within a window-sized datetime interval around each of those rows, incurring only 100 incomplete matching states. Furthermore, an index on primary_type will allow us to get rows matching B's condition directly without a table scan, and process those rows grouped by intervals in parallel.

Previous works in streaming have found flaws in NFA's fixed order of execution and proposed alternatives like tree-based executor [39] and lazy evaluator [33], but as standalone streaming systems, they command significant modifications to the executor and optimizer in order to integrate with existing relational systems. For historical analysis, Korber et al. [34] proposed an approach to make use of indexes via a strategy called "prefiltering": use indexes to filter the input records, then apply the NFA-based MATCH_RECOGNIZE on the filtered result. This requires the input table to be physically sorted with a clustered index, and secondary indexes for relevant columns in the query. It limits applicability as the input table may be stored as files in a data lake with no indexes. It also demands adding a new physical operator for prefiltering. We have not seen other work concerning operator-level parallel execution of MATCH_RECOGNIZE.

Given the importance of simple system integration and efficiency, we adopted the prefiltering strategy but chose to use Join to perform the filtering. Join's order of evaluation can be optimized; it also makes use of parallel execution algorithms and available indexes. Above all, Join is supported by nearly all relational systems, so our approach can be implemented as a logical plan rewrite rule without modifying other parts of the host system. The following is an example of how we use Join:

Example 3. Following Example 2, we first obtain two sets of rows, each set matches R's or M's primary_type condition, then join them on the following conditions: R.datetime <= M.datetime, latitude-longitude box, and window constraint. Based on the pattern expression, the R.datetime and the M.datetime in each joined tuple form a temporal range of a potential match. We can then run the NFA-based MATCH_RECOGNIZE on the few rows that fall into those ranges to get the same result as running on all the original input table.

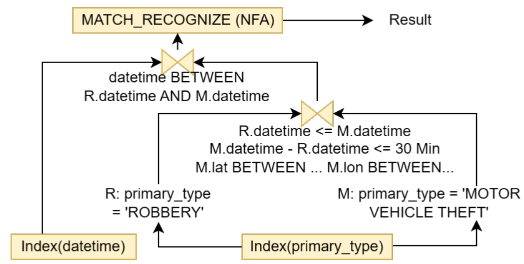


Figure 3: A plan for the query in Figure 1 showing the logical steps to find matches for the pattern (R Z* B Z* M)**.**

Figure 3 illustrates the Joins in Example 3 with indexes. It is 7× faster than NFA in SQL Server (2019). When indexes are not available, in order to avoid expensive cross-product Joins, we make use of window constraint to virtually bucketize the input table, and rewrite the Joins into equality Joins so that each row is only joined with its own and neighboring buckets.

Another challenge we have encountered is query optimization. Though the previous example uses the symbol set {R, M} to create the Joins for generating ranges, it is also possible to use any subset, e.g., {R, B, M}. The cost of a rewritten plan can vary significantly depending on the Joins and their filtering power, so it is important to choose a symbol set that maximizes the net cost reduction. For this, we use a cardinality model tailored to our rewrite by incorporating the semantic of the Joins for a more differentiating cost estimate. We also designed a new cost model for NFA-based MATCH_RECOGNIZE based on the number of state transition function evaluations, estimated using a simulator. We drew a relationship between the MATCH_RECOGNIZE cost model and the join cost model through a bootstrapping calibration.

Compared with NFA-based MATCH_RECOGNIZE implementations on a benchmark of 1,800 query instances spanning over 6 patterns and 3 pattern definitions based on existing datasets [34], our approach hits median speed-ups of 5.4× on Trino (v373 with ORC files on Hive), 57.5× on SQL Server (2019) using column store and 41.6× on row store with indexes.

In summary, we improved MATCH_RECOGNIZE performance in historical analysis scenarios by introducing a logical plan rewrite rule that uses Join-based prefiltering with a specialized cardinality model for the Joins and a new cost model for MATCH_RECOGNIZE. Section 2 references related work; Section 3 presents the rewrite rule; Section 3.4 discusses applicability – the rule requires self-contained pattern definition and either a window condition or distinct rows in the input table. Section 4 covers the cardinality model and cost model; lastly, Section 5 illustrates and discusses the experimental results.

2 RELATED WORK

Complex Event Processing (CEP) is a type of analytics performed on data streams to detect sequences of events matching a user-specified pattern. SASE [58] and its subsequent work [7, 15] proposed a new query language and a execution engine based on Non-deterministic Finite Automaton (NFA) (see Appendix A.2 for a quick overview). Cayuga [13], SPASS [45] and NEEL [36] proposed to optimize CEP queries using sub-pattern/expression sharing among concurrent queries. Kolchinsky and Schuster [32] also explored optimizing concurrent CEP queries but also considered pattern reordering in the optimization space. An alternative to NFA is explored by ZStream [39]: a tree-based query engine with buffers that "assembles" patterns from events through a tree of operators – this idea inspired our approach to use Join operators for partial matches. Kolchinsky and Schuster [31] proposed to unify query optimization for both NFA-based and tree-based execution engines by quantifying the relationship between partial matches in NFA and intermediate results in operator tree. This work helped us gain insight into why our Join-based approach is beneficial (see Appendix F). An enhancement to NFA is AFA [11] (part of Trill [10]): it introduced the concept of "registers" to let users manage matching states easily and write queries using a programming language API. We used AFA to implement the user-defined aggregate (UDA) version of `MATCH_RECOGNIZE` for our experiments. The survey paper from Giatrakos et al. [21] goes into detail to summarize the landscape of CEP works. Overall, the focus of CEP systems is for continuous queries providing online results. This is different from relational systems that are optimized for batch, off-line historical analysis queries.

Sequence Processing in Databases has been studied for decades. The SEQ Project [50–52] proposed a database designed after a sequence data model. SRQL [44] took a different approach to enhance SQL support for sequences by introducing new operators. SQL-TS [48] introduced an earlier version of regular expression syntax for pattern search. It was implemented using a single-pass algorithm inspired by the KMT algorithm [30] for text-matching. DejaVu [16] studied the problem of pattern correlation queries which correlates online streaming data with offline archive data, and proposed query processing algorithms and optimizations. Most recently, Korber et al. [34] studied the problem of improving performance of `MATCH_RECOGNIZE` queries in offline analysis scenario using indexes for prefiltering. Their approach assumes that the data store is ordered by timestamp with a primary index on the timestamp and secondary indexes on other attributes. Additional execution logic that interacts with indexes is required to carry out the prefiltering. In contrasts, our approach does not assume availability of indexes or require changes to the executor.

Join Query Processing is a mature research area and our approach is based on many existing works, namely, join algorithms, statistics estimation, cost-based access path selection, and query optimization. For join algorithms, the survey paper from Graefe [22] provides a practical summary. Worst-case optimal join algorithms [6, 29, 41, 42, 57] provide better runtime guarantee than binary joins in the presence of growing intermediate results. A hybrid of worst-case optimal join and binary joins are employed in Umbra [19]. Most host systems, including Trino and SQL Server, have not yet implemented these algorithms. In our work we utilize the host system's existing index join when index is available, and use hash join when not. For statistics we employ the statistical profile model presented by Mannino et al. [37] with our own cardinality estimators (Section 4.1). For many databases their cost models are influenced by the System R Project from Selinger et al. [49]. Lastly, the Starburst [25] and Volcano/Cascade [23, 24] query optimization frameworks have heavily influenced many relational databases. Thus we assume the host system's optimization framework supports adding logical query plan rewrite rules as in Starburst and Volcano/Cascade.

Band Join is a special type of range join with condition in the form of $A + x < B < A + y$. Many [14, 35, 46, 54] have worked on improving its performance using specialized physical operators. There are two approaches: sorting-based [14] and partitioning-based [35, 54]. Recently [46] proposed to use kd-tree for general range join. A few systems have implemented such specialized band-join operator, e.g., Databricks, Oracle and Vertica, but most have not. Our bucketized prefilter (Section 3.2) was inspired by [54], but we implemented our approach as a logical plan rewrite rule rather than adding a specialized physical operator in the host system.

3 THE REWRITE RULE

In this section, we present our logical query plan rewrite rule that uses Joins to filter the input table to `MATCH_RECOGNIZE` and a temporal bucketization technique to speed up the Joins.

3.1 Basic Prefilter

As described in Section 1, the rewrite rule creates a prefilter that filters the input table for the original `MATCH_RECOGNIZE` query so the final output is unchanged. As sketch of our prefilter construction steps: we first choose a symbol sub-sequence (of length at most 3), e.g., (`R`, `M`), from the query pattern; then construct a join to find all the timestamp ranges $R = \{(t_s, t_e), \ldots\}$ such that R contains the "envolopes" of all timestamps t of rows matching the sub-sequence; lastly, use $R \bowtie_{t_s \leq t \leq t_e} T$ to filter the input table.

Let us dive into the details starting with the Boolean conditions, using the query in Figure 1 as our running example.

Definition 3.1. *An **independent condition** is a Boolean condition that can be evaluated on a single row.*

Definition 3.2. *A **dependent condition** is a Boolean condition that must be evaluated on multiple rows.*

For example, `B.primary_type = 'BATTERY'` is an independent condition on a row matching `B`; `M.lat BETWEEN R.lat - 0.02 AND R.lat + 0.02` is a dependent condition, which must be evaluated on a row matching `M` and a row matching `R`.

Definition 3.3. *A **self-contained dependent condition** is a dependent condition that can be evaluated on rows from the same match*[1] *only.*

All dependent conditions in a `MATCH_RECOGNIZE` query must be self-contained for our rewrite rule to apply. For example, the condition `PREV(R.primary_type) = 'ASSAULT'` would have to

[1] A match is a row sequence that matches the input pattern. Figure 2 depicts an example.

be evaluated on the row immediately preceding the row matching R and outside of the same matching sequence starting at R, thus it is not self-contained. On the other hand, the condition `PREV(B.primary_type) != B.primary_type` is self-contained because the row immediately preceding the row matching B is either a G or R – inside the same match.

Definition 3.4. *A **sequential condition** is a dependent condition in the form equivalent to $A.t \leq B.t$ where symbol A precedes B absolutely in the* `PATTERN` *expression; t is the primary* `ORDER BY` *key.*

A sequential condition is not stated but rather implied by the pattern expression. For example `R.datetime <= B.datetime` is implied as a row matches R comes before a row matches B in the same pattern match. In case of Alternation (e.g. `(A | B)`) or Permutation (e.g. `PERM(A, B, C)`), there is no sequential condition among the participating symbols.

Definition 3.5. *A **window condition** is a dependent condition in the form equivalent to $B.t - A.t \leq w$ where symbol A precedes B absolutely in the* `PATTERN` *expression; t is the primary* `ORDER BY` *key; w is a non-negative value called window size.*

A window condition can be "propagated backward" through sequential conditions:

Proposition 3.1. *A window condition $C.t - A.t \leq w$ together with sequential conditions $A.t \leq B.t$ and $B.t \leq C.t$ generate new window conditions $C.t - B.t \leq w$ and $B.t - A.t \leq w$.*

For example, there is no stated conditions for Z but using the above proposition, we can assign a new window condition to Z and R: `Z.datetime - R.datetime <= INTERVAL '30' MINUTE`. Note that the propagation of window condition can be applied independently to the query without creating a prefilter.

Definition 3.6. *A **pattern window condition** is a window condition between A and B that are respectively the first and last symbols ordered by their sequential conditions: for the set of all symbols χ, $A.t \leq s.t$ and $B.t \geq s.t$ for any $s \in \chi$.*

For example, `M.datetime - R.datetime <= INTERVAL '30' MINUTE` is a pattern window condition as R and M are the first and the last symbols, respectively.

With these conditions in Def. 3.1-3.6, we can define prefilter by starting with a special case where the input pattern involves no Alternation nor Kleene operator, as well as no duplicates in the input table, and then extending it to the general case by converting the general pattern into special ones.

3.1.1 Special Case. We first consider the case when the input pattern only has Concatenation operator, e.g., `(A B C D E)`.

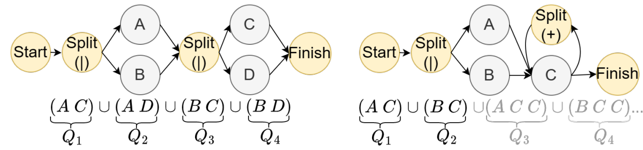
Definition 3.7. *Given an input relation T (without duplicates) and a query pattern $Q = (s_1\ s_2\ ...\ s_n)$ involving only Concatenation operators on an ordered set of pattern symbols $\chi = (s_1, s_2, ..., s_n)$ and self-contained dependent conditions C_χ, for a subsequene $X = (s_{i_1}, s_{i_2}, ..., s_{i_k})$ of χ where $1 \leq i_j \leq n$ for $\forall 1 \leq j \leq k$, the **prefilter** P_X can be constructed using bag-based relational algebra[2]:*

$$P_X = \delta\Big(\pi_{f(t_1,t_k)}\Big(\rho_{t_1/t}\big(\sigma_{C_{s_{i_1}}}(T)\big) \bowtie_{C_{s_{i_1},s_{i_2}}} \sigma_{C_{s_{i_2}}}(T) \bowtie \ldots \sigma_{C_{s_{i_{k-1}}}}(T) \bowtie_{C_{s_{i_1},s_{i_2}\ldots s_{i_k}}} \rho_{t_k/t}\big(\sigma_{C_{s_{i_k}}}(T)\big)\Big) \bowtie_{t_s \leq t \leq t_e} T\Big)$$

where $C_{s_{i_j}}$ is the set of independent conditions associated with s_{i_j}; $C_{s_{i_1},...,s_{i_j}}$ is the set of dependent conditions (including sequential and window conditions) associated with $s_{i_1}, ..., s_{i_j}$; and the function

$$f(t_1, t_k) = \begin{cases} (t_1 \rightarrow t_s, t_k \rightarrow t_e) & \text{if } (i_1 = 1) \wedge (i_k = n) \\ (t_k - w \rightarrow t_s, t_k \rightarrow t_e) & \text{if } (\exists w) \wedge (i_k = n) \\ (t_1 \rightarrow t_s, t_1 + w \rightarrow t_e) & \text{if } (\exists w) \wedge (i_1 = 1) \\ (t_k - w \rightarrow t_s, t_1 + w \rightarrow t_e) & \text{if } (\exists w) \end{cases}$$

where $\exists w$ means there exists a pattern window condition in pattern Q and w is the window size of the pattern window condition; the δ operator [20] removes duplicate rows in the output of the last join with T due to overlapping time ranges.



(a) ***Q*****:** `((A | B) (C | D))` **(b)** ***Q*****:** `((A | B) C+)`

Figure 4: Decomposing general pattern to special patterns

Any row in a match must be part of a prefilter, as stated by the following proposition:

Proposition 3.2. *Given an input relation T without duplicates, a special pattern Q involving only Concatenation operators on an ordered set of pattern symbols χ, and an ordered symbol set $X \subseteq \chi$, if all dependent conditions C_χ are self-contained, $MR(Q,T) = MR(Q, P_X(T))$.*

$MR(Q,T)$ denotes the results returned by `MATCH_RECOGNIZE` with a query pattern Q, an input relation T, and some arbitrary `AFTER MATCH SKIP` clause. The above proposition can be proved by analyzing the four different cases of $f(t_1, t_k)$. The proof is detailed in Appendix B.1.

3.1.2 General Case. We move on to the general case when the pattern may contain Concatenation, Alternation[3], and Kleene operators. In such case, a prefilter can be generated following two steps: *(1)* decompose the general pattern Q into special patterns $\{Q_1, Q_2 ... Q_m\}$ and let χ_j be the the ordered set of pattern symbols in each Q_j; *(2)* generate a prefilter P_{X_j} for each special pattern Q_j following Definition 3.7 where $X_j \subseteq \chi_j$ and union the prefilters to generate a prefilter for Q, i.e., $\cup_{j=1}^{m} P_{X_j}$. We detail each step below.

Step 1 (Decomposition). Given a pattern Q, we first construct a corresponding NFA state transition graph, such as those in Figure 4,

[2] $\rho_{t_1/t}$ and $\rho_{t_k/t}$ means renaming column t as t_1 and t_k, respectively; $\pi_{f(t_1,t_k)}$ [20] denotes an extended projection that takes attributes t_1 and t_k, applies some calculation (e.g., $t_k - w$ and t_k in the second case), and renames them to t_s and t_e respectively; δ is a deduplication operator removing duplicate rows from overlapping time ranges.
[3] Permutation is mapped to Alternations. E.g., `PERM(A, B) === (A B)|(B A)`.

then perform a depth-first graph traversal to generate all possible paths from the start node to finish node. From each path, we construct a special pattern. We note that the number of special patterns derived from Q can be exponential in the query size. Figure 4a is an example of decomposing pattern `((A|B) (C|D))`.

To handle cycles introduced by Kleene operators (e.g., Figure 4b), the algorithm exits any cycle after encountering one `Split(+)` node or a `Split(*)` node for the second time.

Step 2 (Prefilter Union). Let $\{Q_1, Q_2, ..., Q_m\}$ be the set of special patterns and let χ_i be the ordered set of pattern symbols in each Q_j. For each Q_j, construct a prefilter P_{X_j} following Definition 3.7 where $X_j \subseteq \chi_j$. A prefilter for Q can be constructed as $\cup_{j=1}^{m} P_{X_j}$.

Proposition 3.3. *Given a relation T without duplicates, a query pattern Q with conditions self-contained, and a prefilter $\cup_{j=1}^{m} P_{X_j}$ constructed using the steps above, $MR(Q, \cup_{j=1}^{m} P_{X_j}(T)) = MR(Q, T)$.*

This can be proved by showing that although the union of special patterns, $\cup_{j=1}^{m} Q_j$, is not equivalent to the input query pattern Q if Q has any Kleene operator, $\cup_{j=1}^{m} Q_j$ is sufficient for creating a prefilter for Q: since for each Q' (e.g., `(A C C C)` in Figure 4b) created by a path going over a cycle, there exists a Q_j (e.g., `(A C)`) returned by the decomposition procedure such that any prefilter constructed for Q_j following Definition 3.7 is also a valid prefilter for Q'. The complete proof is in Appendix B.3.

We further optimize $\cup_{j=1}^{m} P_{X_j}$ by removing redundant components. Using the pattern in Figure 4b as an example, one possible prefilter for Q is $P_{(\mathsf{A},\mathsf{C})} \cup P_{(\mathsf{C})}$, where $P_{(\mathsf{A},\mathsf{C})}$ is constructed from Q_1 and $P_{(\mathsf{C})}$ from Q_2. Using the fact that $P_{(\mathsf{A},\mathsf{C})} \subseteq P_{(\mathsf{C})}$ we can safely eliminate $P_{(\mathsf{A},\mathsf{C})}$ in the union, which turns out to be just $P_{(\mathsf{C})}$.

Figure 5 shows an example rewrite using prefilter for symbol set (R, M) in SQL: the `ranges` expression finds all pairs of `[R.datetime, M.datetime]` satisfying the conditions stated in the original query; the `prefilter` expression produces a subset of the input table rows that fall into at least one of the ranges produced by `ranges`. The corresponding prefilter $P_{(\mathsf{R},\mathsf{M})}$ can be expressed as:

$$\delta\left(\pi_{(t_1 \to t_s, t_2 \to t_e)}\left(\rho_{t_1/t}\left(\sigma_{C_R}(T)\right) \bowtie_{C_{R,M}} \rho_{t_2/t}\left(\sigma_{C_M}(T)\right)\right) \bowtie_{t_s \le t \le t_e} T\right)$$

where C_R and C_M are the independent conditions defined for symbols R and M respectively (i.e., `(R.primary_type='ROBBERY')` and `(M.primary_type='MOTOR_VEHICLE_THEIFT')`); $C_{\mathsf{R},\mathsf{M}}$ are the dependent conditions between R and M (e.g., `(M.lon BETWEEN R.lon - 0.05 AND R.lon + 0.05)` and `(R.datetime <= M.datetime)`).

This prefilter also handles queries with the optional `PARTITION BY` clause, which specifies the pattern to be found within every partition (see Appendix C).

3.1.3 Symbol Set Search Space. Given a special pattern Q, we still need to choose a symbol set X for the prefilter. We use a simple procedure to produce the choices of symbol sets that satisfy Definition 3.7 for a special pattern Q with symbols $\chi = (s_1, s_2, ..., s_n)$:

(1) Mark (s_1, s_n) as a symbol set, where s_1 and s_n are the first and last symbols.
(2) If there exists a window condition between s_1 and s_n, i.e., pattern window condition, mark all subsets of χ as symbol sets.

For a special pattern `(R B M)`, it has the following symbol sets: (R), (B), (M), (R, B), (R, M), (B, M), (R, B, M). For subset selection, in practice we mark only 1, 2 and 3-symbol subsets to limit the number of choices and to avoid the possible large estimation error when involving more than 3 joins.

```
WITH ranges AS (
    SELECT R.datetime as t_s, M.datetime as t_e
    FROM Crimes AS R, Crimes AS M
    WHERE R.datetime <= M.datetime
        AND R.primary_type = 'ROBBERY'
        AND M.primary_type = 'MOTOR VEHICLE THEFT'
        AND M.lon BETWEEN R.lon - 0.05 AND R.lon + 0.05
        AND M.lat BETWEEN R.lat - 0.02 AND R.lat + 0.02
        AND M.datetime - R.datetime <= INTERVAL '30' MINUTE
), prefilter AS (
    SELECT DISTINCT Crimes.* FROM Crimes, ranges AS r
    WHERE datetime BETWEEN r.t_s AND r.t_e
) SELECT * FROM prefilter MATCH_RECOGNIZE (/* same as before */);
```

Figure 5: A rewrite Figure 1 using symbol set {R, M}.

When there is at least one symbol set, we estimate the costs of all rewrites plus the original plan (no rewrite), and choose the plan with the lowest cost. This is presented in Section 4.

For a general pattern Q, we first generate symbol sets for each decomposed special pattern $\{Q_1, Q_2, ..., Q_m\}$ using the procedure above, and then generate the m-combinations of symbol sets. If any one of the special patterns produces no symbol set, we terminate the rewrite rule. To bound the optimization cost, we limit the number of distinct combinations to 100 and prioritize small symbol sets for each Q_j. For now we focus on the effectiveness of our prefilter strategy, and improving optimizer efficiency is left as a separate work for future research.

3.2 Bucketized Prefilter

Indexes speed up Joins in prefilter generated following Definition 3.7 but indexes are not always available. Relational systems designed for data analysis workload today are mostly using column-oriented storage format [55], instead of row-oriented storage with indexes designed for transaction workload. Thus, it is important to handle scenarios with no indexes.

Our insight is that a row should only join with other rows belonging to the same match, and by combining a sequential condition (Definition 3.4) with a pattern window condition (Definition 3.6), we can create a new equality Join condition to aggressively prune out other rows outside the window-sized neighborhood of that row.

To assign a window-sized neighborhood to each row, we add a new computed column called `bucket`:

Definition 3.8. *When there exists a pattern window condition, a **bucket** is given by the expression $\lfloor t/w \rfloor$ where t is the primary `ORDER BY` key and w the pattern window size.*

Combining the window condition with the sequential conditions, we introduce a new condition that points two rows in the same temporal neighborhood to the same bucket.

Definition 3.9. *A **bucket condition** is an dependent condition derived from a pattern window condition and a sequential condition between symbols A and B where A precedes B, in a form equivalent to*

$$\lfloor A.t/w \rfloor = \lfloor B.t/w \rfloor \textbf{ OR } \lfloor A.t/w \rfloor + 1 = \lfloor B.t/w \rfloor$$

where w is the window size of the pattern window condition.

With the above we update the basic prefilter (Definition 3.7) to use bucket and bucket conditions, and create a new prefilter using the following definition.

Definition 3.10. *Given a relation T with or without duplicates and a query pattern Q, if the query pattern Q is self-contained and has a pattern window condition with size w, the* ***bucketized prefilter*** *B_X for symbol set $X = [s_{i_1}, s_{i_2}, ..., s_{i_k}]$ can be created as*[4]*:*

$$B_X = \delta\Big(\Big(\pi_{g(t_1,t_k)}\Big(\sigma_{C_{s_{i_1},...,s_{i_k}}}\Big(\rho_{t_1/t}\big(\sigma_{C_{s_{i_1}}}(T)\big) \bowtie_{b_{s_{i_1},s_{i_2}}} \cdots \bowtie_{b_{s_{i_1},...,s_{i_k}}} \rho_{t_k/t}\big(\sigma_{C_{s_{i_k}}}(T)\big)\Big)\Big)\Big) \times Seq(bk_s, bk_e)_{bk}\Big) \bowtie_{bk=\lfloor t/w \rfloor} T$$

where $b_{s_{i_1},...,s_{i_j}}$ is the set of bucket conditions among $s_{i_1}, .., s_{i_j} \in X$; bk is the bucket attribute; $Seq(bk_s, bk_e)_{bk}$ is a table-valued function that produces a relation with a single bk attribute with values $bk_s, bk_s + 1, .., bk_e$[5]*; and the tuple-valued function $g(t_1, t_k) =$*

$$\begin{cases} (\lfloor t_1/w \rfloor \rightarrow bk_s, \lfloor t_k/w \rfloor \rightarrow bk_e) & \text{if } (i_1 = 1) \wedge (i_k = n) \\ (\lfloor t_k/w \rfloor - 1 \rightarrow bk_s, \lfloor t_k/w \rfloor \rightarrow bk_e) & \text{else if } i_k = n \\ (\lfloor t_1/w \rfloor \rightarrow bk_s, \lfloor t_1/w \rfloor + 1 \rightarrow bk_e) & \text{else if } i_1 = 1 \\ (\lfloor t_k/w \rfloor - 1 \rightarrow bk_s, \lfloor t_1/w \rfloor + 1 \rightarrow bk_e) & \text{else} \end{cases}$$

Similar to the basic prefilter (Definition 3.7 and Proposition 3.2), any row in a match must also be part of a bucketized prefilter, i.e., $MR(Q, B_X) = MR(Q, T)$. See Appendix B.2 for the proof. Different from the basic prefilter, a bucketized prefilter accepts input table with duplicates. This is because δ is applied on the buckets bk and each input table's row belongs to one bucket, joining the input table with distinct buckets does not introduce new duplicate rows. Thus, the duplicate rows in the input table is preserved as they were never removed.

We can follow the same procedure in Section 3.1.2 to construct a bucketized prefilter for a general pattern Q, i.e., $\cup_{j=1}^{m} B_{X_j}$.

Figure 6 gives an example in SQL. The `input_bucketized` expression assigns the computed column `bk` using Definition 3.8. The `ranges` expression is a union of two equality Joins, one for each part of the bucket condition's OR. By generating equality Joins, we make it possible for the host system to execute this plan using efficient algorithms such as hash Join [22] rather than a nested loop Join. The `buckets` expression produces the set of buckets from the ranges produced earlier. Seq is commonly available in many relational systems such as Trino (`sequence`) and PostgreSQL (`generate_series`) and easy to add if needed[6]. The `prefilter` expression produces the set or rows for `MATCH_RECOGNIZE` by joining the `bk` column of the input with `buckets`. The corresponding prefilter $B_{(\mathrm{R},\mathrm{M})}$ can be expressed as below:

$$\delta\Big(\pi_{(\lfloor t_1/w \rfloor \rightarrow bk_s, \lfloor t_2/w \rfloor \rightarrow bk_e)}\Big(\rho_{t_1/t}\big(\sigma_{C_R}(T)\big) \bowtie_{C_{R,M}} \rho_{t_2/t}\big(\sigma_{C_M}(T)\big)\Big) \times Seq(bk_s, bk_e)_{bk}\Big) \bowtie_{bk=\lfloor t/w \rfloor} T$$

Bucketized prefilter uses "lower-resolution" ranges on buckets so it produces more rows, but it is a small price for significantly faster

```
WITH input_bucketized AS (
    SELECT *, cast(datetime / '30' MINIUTE AS bigint) AS bk
    FROM Crimes
), ranges AS (
    SELECT R.bk as bk_s, M.bk as bk_e
    FROM input_partitioned AS R, input_partitioned AS M
    WHERE R.bk = M.bk /* rest same as before */
    UNION
    SELECT R.bk as bk_s, M.bk as bk_e
    FROM input_bucketized AS R, input_bucketized AS M
    WHERE R.bk + 1 = M.bk /* rest same as before */
), buckets AS (
    SELECT DISTINCT bk FROM ranges
    CROSS JOIN Seq(bk_s, bk_e) AS t(bk)
), prefilter AS (
    SELECT i.* FROM input_partitioned AS i, buckets AS b
    WHERE i.bk = b.bk
) SELECT * FROM prefilter MATCH_RECOGNIZE (/* same as before */);
```

Figure 6: A rewrite of the query in Figure 1 using symbol set `{R, M}` using bucketized prefilter.

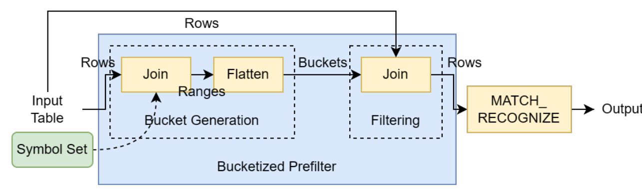


Figure 7: A candidate plan generated by the rewrite rule using bucketized prefilter given a symbol set.

Join algorithms. Indeed, the previous rewrite (Figure 5) clocked 19s in SQL Server with columnar storage format, while this rewrite finished in 2s under the same setting.

3.3 Rule Implementation

Given a `MATCH_RECOGNIZE` query, the rewrite follows these steps:

(1) Extract symbol sets, terminate if no symbol set was found.
(2) Extract independent, dependent, sequential, window, and bucket conditions, terminate if Definitions 3.7 and 3.10 are infeasible.
(3) For each symbol set, generate a candidate plan (Figure 7).
(4) Use cost model to select a plan (Section 4).

As shown in Figure 7, a candidate plan using bucketized prefilter has two major components: the prefilter and the `MATCH_RECOGNIZE` node. The prefilter has two sub-components: (a) bucket generation, which includes the Joins to identify ranges that contain all matches followed by flattening the ranges to obtain the buckets, and (b) filtering, which joins the input table with the buckets to obtain the prefiltered rows. For flattening ranges in bucket generation, we create a lateral Join followed by a distinct aggregation node as exemplified by Figure 6.

When the input table follows a row-oriented layout and has a clustered index on the `ORDER BY` key of the query, we use basic prefilter, simply by removing the fattening step and replacing buckets with ranges in Figure 7.

[4] δ is a duplicate-elimination operator [20].
[5] Because $Seq(bk_s, bk_e)$ takes a tuple (bk_s, bk_e) from the other side of the cross-product Join, the "×" in Definition 3.10 is implemented as a lateral Join.
[6] Note that any sequence is at most length 3 due to $g(t_1, t_k)$ in Definition 3.10, so the cost of generating it is minimal. It is also possible to use `UNION` instead.

3.4 Rule Applicability

A basic prefilter applies when the input table rows are made distinct and the dependent conditions in the query pattern are self-contained (Definition 3.7). To make the input rows distinct, one simple way is to add an auto-increment ID column to the input table. Appendx C describes a more efficient solution to handle duplicates utilizing windows functions.

A bucketized prefilter accepts duplicate rows but requires a pattern window condition in addition to self-contained dependent conditions (Definition 3.10). Existing research work [34] assumed pattern window, and both Oracle's [43] and Apache Flink's [8] `MATCH_RECOGNIZE` syntax has a `WITHIN` clause for specifying a pattern window.

Intuitively, the prefilter-based rewrite is useful when there exists selective symbols or selective combinations of symbols, and less useful when there are too many alternations in the query pattern. We rely on the optimizer to decide whether to trigger the rewrite based on estimated cost, and the optimizer is integrated with our cost model to obtain a more accurate estimation as detailed in Section 4 below.

4 SYMBOL SELECTION

In this section, we present our cost-based approach to select symbols for creating prefilter. We first detail the cost models for each stage, i.e., prefilter and `MATCH_RECOGNIZE` (Section 3.3), followed by how we consolidate them into one unified cost model. Even though a bucketized prefilter does not rely on indexes and tends to have less pruning power than a corresponding basic prefilter, we model a basic prefilter's cost "pessimistically" under the assumption of it being a bucketized prefilter to be more confident that a rewrite, when triggered, brings performance improvement. We focus on the CPU cost, since under the bucketized prefilter scenario, the I/O cost is dominated by scanning the input table in Filtering step and is the same for all symbol sets.

4.1 Cost Model for the Prefilter

We focus on the cost model of B_X for a special pattern Q. For a general pattern $\mathcal{Q}$, we can calculate the cost of $\cup_{j=1}^{m} B_{X_j}$ as the summation of each B_{X_j}'s cost. As illustrated in Figure 7, a bucketized prefilter consists of two steps: (1) Bucket Generation; and (2) Filtering. The prefilter's CPU cost is a sum of the two steps' costs.

$$C_{\text{prefilter}} = C_{\text{bucket generation}} + C_{\text{filtering}} \tag{1}$$

In the cost model, each Join operator's CPU cost is the sum of the input and output cardinalities multiplied by respective record sizes, so accurate cardinality estimation is crucial.

We found that the approach of recursively applying existing operator-level cardinality estimators [37] in each step cannot correctly differentiate the pruning power of candidate symbol sets. Figure 8a illustrates `SQLServer`'s estimated cardinality of prefilter output on a synthetic dataset (y-axis), compared with the true cardinality (x-axis). The detailed setting can be found in Appendix E.1. We found that `SQLServer` tends to overestimate the prefilter cardinalities by a large margin. We have also tried the cardinality estimate in `Trino` and `PostgreSQL`, but all provide unsatisfactory estimate due to the challenges in estimating SQL constructs like `UNNEST` and `DISTINCT`: `Trino` does not provide an estimate since it is lacking estimators for such SQL construct; `PostgreSQL` provides the same estimate for all candidate symbol sets due to its constant estimator for `DISTINCT`, i.e., 200. Given this, we employ a specialized cardinality estimator that incorporates the unique semantic of prefilter. As shown in Figure 8a, in general our estimated cardinality (blue dots) increases as the increase of the true cardinality. Quantitatively, our estimator's median Q-Error [17], i.e., $\max(\frac{est}{true}, \frac{true}{est})$, is 1.93 while `SQLServer`'s is 8.46.

4.1.1 Bucket Generation. Bucket Generation takes a table T with buckets computed following Definition 3.8 and a symbol set $X = [s_{i_1}, s_{i_2}, .., s_{i_k}]$ as the inputs, and outputs a set of buckets where pattern matches might occur. Our goal here is to estimate the number of buckets in the output. First, assuming even distribution of rows over the buckets, we estimate the number of buckets with rows satisfying each symbol s_{i_j}'s independent conditions $C_{s_{i_j}}$:

$$|B_{\sigma_{C_{s_{i_j}}}}| = (1 - (1 - \frac{1}{\beta})^{|\sigma_{c_{s_{i_j}}}(T)|}) \cdot \beta \tag{2}$$

where β refers to the total number of buckets and is estimated as $\beta = \frac{t.max - t.min}{w}$; $t.max$ and $t.min$ are maximum and minimum of t. By assuming independence of the independent conditions, we then estimate the number of buckets each satisfies all independent conditions $C_{s_{i_1}}...C_{s_{i_k}}$ – the intersection:

$$|B_{\sigma_{C_{s_{i_1}}...C_{s_{i_k}}}}| = \beta \cdot \Pi_{j=1}^{k} \frac{|B_{\sigma_{C_{s_{i_j}}}}|}{\beta} \tag{3}$$

Lastly, we estimate the number of buckets that satisfy both independent and dependent conditions of all symbols in X – the output of Bucket Generation:

$$|B_{\sigma_{C_X}}| = (1 - (1 - \delta)^{(\frac{|T|}{\beta})^k}) \cdot |B_{\sigma_{C_{s_{i_1}}...C_{s_{i_k}}}}| \tag{4}$$

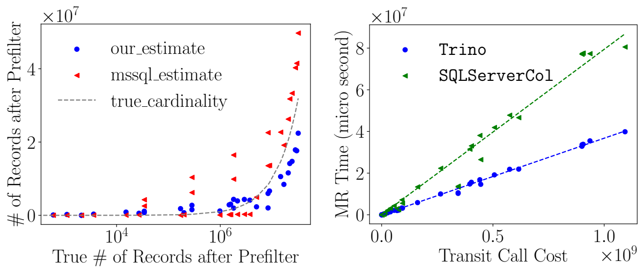
where $\delta = \frac{|\sigma_{C_X}(\bowtie T)|}{\Pi_{j=1}^{k}|\sigma_{C_{s_{i_j}}}(T)|}$ represents the selectivity of all dependent conditions, and $|\sigma_{C_X}(\bowtie T)|$ the estimate of Join cardinality produced by the host system. $(1-\delta)^{(\frac{|T|}{\beta})^k}$ is the probability that no row combination in a bucket satisfies all dependent conditions.

What is unique about this cardinality estimator is that it considers the Joins and Flatten as a single step without analyzing the many relational operators involved. This enables a more accurate cardinality estimate that is inline with the semantic of Bucket Generation. With the cardinality we calculate the CPU cost estimate:

$$\begin{aligned} C_{\text{bucket generation}} = & \sum_{j=1}^{k} |r_{s_{i_j}}| \cdot |\sigma_{C_{s_{i_j}}}(T)| \\ & + C_{\text{join}}(|\sigma_{C_{s_{i_1}}}(T)|, .., |\sigma_{C_{s_{i_k}}}(T)|, |\sigma_{C_X}(\bowtie T)|) \\ & + |r_{\text{bucket}}| \cdot |B_{\sigma_{C_X}}| \end{aligned} \tag{5}$$

where C_{join} is the minimum join CPU cost estimate given the input and output cardinalities; $|r_{s_{i_j}}|$ is the byte size of a projected row matching with symbol s_{i_j}; $|r_{\text{bucket}}|$ is the byte size of a bucket row.

4.1.2 Filtering. Filtering takes input a set of buckets output by Bucket Generation and joins them with the original input table to obtain the prefiltered rows as the output. Because each bucket from Bucket Generation is unique and corresponds to a temporal range in the input table, we can estimate the number of prefiltered rows



(a) Estimated Cardinality for the Prefilter **(b)** $CPU_{\texttt{MATCH\_RECOGNIZE}}$ **vs.** $C_{\texttt{MATCH\_RECOGNIZE}}$

Figure 8: Cost Modeling

as $|B_{\sigma_{C_X}}| \cdot \frac{|T|}{\beta}$ assuming uniform distribution of these buckets. The CPU cost estimate is calculated as:

$$C_{\text{filtering}} = |r_{\text{bucket}}| \cdot |B_{\sigma_{C_X}}| + |r| \cdot |T| + |r| \cdot |B_{\sigma_{C_X}}| \cdot \frac{|T|}{\beta} \tag{6}$$

where $|r|$ is the byte size of a projected row with attributes used in `MATCH_RECOGNIZE`.

4.1.3 Required Stat. To make system integration simpler, these estimators only require a few common statistics. For each column: (1) the maximum and minimum (for β); (2) distinct count or histogram (for $|\sigma_{C_{s_{ij}}}(T)|$ and $|\sigma_{C_X}(\bowtie T)|$), (3) null fraction and (4) byte size for cost estimate; and for each table the total number of rows.

4.2 Cost Model for `MATCH_RECOGNIZE`

The second piece of the puzzle to complete the cost model of the rewrite is a cost model for `MATCH_RECOGNIZE`. `Trino` has a physical operator for `MATCH_RECOGNIZE` but it has no cost model. In the index-accelerated `MATCH_RECOGNIZE` approach [34], cost is modeled as a linear function of input cardinality with a fixed parameter. This model counters our observation that different query patterns have vastly different execution times. For instance, pattern `(R Z* B Z* M)` in Figure 1 takes 60s, while `(R)` only takes 5s in `Trino`. We propose to model the CPU cost of `MATCH_RECOGNIZE` using not only input cardinality but also the query pattern (i.e., NFA structure).

4.2.1 Cost Model Analysis. `MATCH_RECOGNIZE` is implemented using NFA. Algorithm 1 illustrates a simplified matching process with NFA. `partial_matches` in line 1 maintains a list of partial matches available so far and is updated after processing each new row (line 9 and 12). Each partial match `pm` consists of two main components: (1) current state in NFA denoted as `pm.state`; (2) past matching information, denoted as `pm.info`. Depending on the detailed implementation, different matching information is stored. For instance, `Trino` keeps a full sequence of symbol names that this partial match *pm* has ever matched; while AFA [11] keeps only some necessary matching information with a predefined schema. Initially, `partial_matches` contains one dummy partial match with `Start` state and empty past matching information. Next, let us see how the matching procedure works (line 2-12). Rows from the input relation are consumed in ascending order of `t`. When a new row comes in (line 2), for each partial match `pm` in `partial_matches` and each out-edge of its state `pm.state`, we evaluate the transit

Algorithm 1: NFA Matching

```
   input  : NFA, relation R sorted by datetime
   output : All matches
1  matches ← [] ; partial_matches ← [(Start,∅)] ;
2  for r ∈ T do
3      next_partial_matches ← [(Start,∅)] ;
4      for pm ∈ partial_matches do
5          for e ∈ pm.state.out_edges do
6              if e.transit_func(r, pm.info) then
7                  state ← e.out_node ;
8                  info ← Update(pm.info) ;
9                  next_partial_matches.Add((state,info)) ;
10                 if state.out_edges == ∅ then
11                     matches.Add(info) ;
12     partial_matches ← next_partial_matches
13 return matches ;
```

function (line 6). If the transit function returns true, a new partial match with updated `state` and `info` (line 7-8) is created and inserted into `partial_matches` for the next iteration's consumption (line 9). Line 10-11 updates the matching results when the partial match `pm` has reached any finish state with no out-going edges.

The transit function (line 6) is evaluated for every iteration looking up for partial matches to extend or terminate. It is reasonable to model the CPU cost of `MATCH_RECOGNIZE` as the total cost of transit function evaluations. We use the row size, which is calculated as the total byte size of participating columns, as a proxy for estimating the unit CPU cost of transit function: $C_{\texttt{MATCH\_RECOGNIZE}} = \gamma \cdot |r|$, where γ is the total number of transit function evaluations and $|r|$ is the row size. We experimentally validate this cost function on a synthetic dataset as shown in Figure 8b, where x-axis denotes $C_{\texttt{MATCH\_RECOGNIZE}}$ and y-axis is the CPU time for `MATCH_RECOGNIZE` with single thread. The detailed setting can be found in Appendix E.1. In both systems, i.e., `Trino` + `Hive` and `SQLServer` + `Col` as detailed in Section 5.1, $CPU_{\texttt{MATCH\_RECOGNIZE}}$ is generally linear to $C_{\texttt{MATCH\_RECOGNIZE}}$ with high R^2 score. However, the linear coefficient differs from system to system.

4.2.2 Estimating the Number of Transit Function Evaluations (γ) in NFA. Since we have expressed the cost of `MATCH_RECOGNIZE` as the cost of transit function evaluations, our task becomes estimating the number of transit calls in NFA, i.e., γ. Note that γ depends on both the data and query pattern. See examples in Appendix E.2.

As described in Section 4.1.3, query optimizer relies on statistics on base table to estimate intermediate statistics for each operator and their cost. So we employ existing base profiles for estimating γ. Since γ also depends on the query pattern, our approach simulates the NFA matching process with base statistical profiles as follow:

(1) Given base profiles, estimate the transit probabilities between NFA states, denoted as p.
(2) Given transit probabilities p, simulate the matching process for ψ iterations and count the number of transit functions evaluated, denoted as γ_ψ.
(3) Given γ_ψ and ψ, estimate γ.

The main challenge lies in step (1). In particular, window condition in Definition 3.5 is highly correlated with sequential conditions in Definition 3.4. Naively considering window conditions in estimating state transition probabilities is error-prone, as sequential conditions have already been encoded in NFA. To tackle this, we propose a *window-based simulation* in step (2) to take care of window conditions, so we can safely ignore window conditions when calculating transit probabilities in step (1). See details below.

Step (1). we apply Join selectivity estimation to estimate the state transition probability between two states. Each state (or node) in NFA corresponds to a symbol $s_i \in \mathcal{S}$ or some utility state like s_{start} and s_{split}. We traverse the NFA state transition graph in a breath-first manner: for each edge $e = (u, v)$, we estimate p_e as the selectivity of $u \bowtie v$, and update v's statistical profile as the output relation's profile. Mathematically, we have

$$p_e = \begin{cases} 1 & \text{if } v = s_{split} \\ \texttt{Selectivity}(\sigma_{C_v}) & \text{elif } u = s_{start} \\ \texttt{Selectivity}(u \bowtie_{C_{uv}} (\sigma_{C_v}(v))) & \text{else} \end{cases}$$

where the `Selectivity` is calculated using existing estimators for selection (σ) and Join ($\bowtie$).

Step (2). with transit probability p from Step (1), we can now introduce our window-based simulation. Note that the simulation process does not use actual data, rather, it evaluates the transit functions randomly according to the probabilities.

To fulfill the window constraint, we fix a starting row and simulate the matching process within its time window. In other words, the simulation is performed within one sliding window. Specifically, we first translate the *time* window constraint, i.e., $B.t - A.t \leq w$ in Definition 3.5, into *row* window constraint, i.e., $B.rid - A.rid \leq \psi$ where rid refers to row id in ascending order of t and we call ψ *row window size.* Row window size ψ is estimated as $\frac{|T|\cdot w}{\texttt{t.max}-\texttt{t.min}}$ by assuming records are evenly spaced in domain t. Having obtained p_e for each edge in NFA and the row window size ψ, we can then conduct NFA simulation starting from row 1 to row ψ. The simulation is similar to Algorithm 1 but with a few modifications: (1) `partial_matches` (and `next_partial_matches`) maintains the number of partial matches at each state without distinguishing their past matching information at each state; (2) instead of iterating over all records in T (line 2), our simulation is conducted for ψ iterations; (3) `next_partial_matches` is initialized as empty in line 3 since the starting row is fixed in our window-based simulation; (4) without checking the transit function on each row (line 6), we always add $(\texttt{state}, p_e)$ to `next_partial_matches` (line 9); (5) we count the number of times we hit line 6 in variable γ_ψ. An example can be found in Appendix E.3.

Step (3). γ_ψ calculates the number of transit function evaluated when matching from a fixed starting row. In total, there are $|T|$ sliding windows starting at each record. Thus, we extrapolate γ=γ_ψ×$|T|$, where $|T|$ is the total number of records in T.

4.3 Rewrite Cost Model and Calibration

We use a linear model for the total CPU cost of the rewrite combining the cost models of the prefilter and `MATCH_RECOGNIZE`:

$$C_{\text{rewrite}} = C_{\text{prefilter}} + \omega \cdot C_{\texttt{MATCH\_RECOGNIZE}} \tag{7}$$

where ω is a scale calibration parameter depending on the host system and platform. ω is needed because the two cost models are developed using different estimators.

We use a bootstrapping calibration process to estimate ω for a new environment. First, for prefilter stage, using a synthetic table and a query, we measure the execution CPU time of a number of rewrites by varying the symbol sets and estimate the costs based on Section 4.1. We then fit a linear model, $CPU_{\text{prefilter}} = \omega_{\text{prefilter}} \cdot C_{\text{prefilter}}$, on the CPU time and the estimated costs. Then, for `MATCH_RECOGNIZE` stage, we run `MATCH_RECOGNIZE` on the materialized results from the prefilter stage, measure the CPU time and estimated costs based on Section 4.2. Similarly, fit another linear model $CPU_{\texttt{MATCH\_RECOGNIZE}} = \omega_{\texttt{MATCH\_RECOGNIZE}} \cdot C_{\texttt{MATCH\_RECOGNIZE}}$. Finally, we use $\omega = \frac{\omega_{\texttt{MATCH\_RECOGNIZE}}}{\omega_{\text{prefilter}}}$ to bring both cost models into the same scale. ω is approximately 5 in both `Trino` and `SQLServer`.

5 EXPERIMENTS

In this section we present an empirical evaluation of our approach. The end-to-end assessment of our rewrite rule is presented in Section 5.2. An evaluation of our cost model is presented in Section 5.3. Comparisons with existing systems are presented in Section 5.4.

5.1 Setup

In this section we present the details of our experimental setup.

Dataset We used the `Crimes` datasets from the existing work on index-accelerated `MATCH_RECOGNIZE` [34]. `Crimes` records crimes in Chicago from January 2001 to June 2020. There are 6.5M records, each representing a crime report with 22 attributes, including [Primary_Type], [District], [Beat], [Longitude], and [Latitude].

Queries For the `Crimes` dataset, we tested 6 query patterns listed in Table 1. Among those, `(A Z* B Z* C)` is from [34]. For each pattern, we tested 3 pattern variable definitions (i.e., `DEFINE` clause):

1 `WithinDistrict`: every crimes report (except for `Z`) has a user-specified [Primary_Type] and all of them within the same user-specified [District]. Window size is 30 minutes as in [34].
2 `PartByBeat`: similar to `WithinDistrict` but instead of the conditions on [District] it uses `PARTITION BY` [Beat], which specifies the pattern to be found within every [Beat].
3 `DyGeoBox`: similar to `WithinDistrict` but the [District] conditions are replaced by latitude-longitude proximity conditions with respect to the first report (e.g., Figure 1). It is based on a definition from [34] that uses a constant geo-boundary.

As such, we tested 18 query templates for the `Crimes` dataset.

Methods We evaluate the following methods:

- `BaseNFA`: the NFA-based `MATCH_RECOGNIZE` with pattern window conditions propagated to every symbol (Definition 3.1).
- `JoinNFA`: the rewrite using our Join-based prefilter selected by the cost model. For basic prefilter, we use additional optimization see Appendix C.

Table 1: Query patterns for the `Crimes` dataset.

Query Patterns	Note
`(A Z* B Z* C)`	Three crime reports optionally separated by undefined reports `Z`
`(A B C D E F G)`	7 consecutive and defined crime reports
`(A B+ C D+ E F+)`	At least 6 consecutive and defined crime reports with possible repeats for `B`, `D` and `F`
`(A (B+ C)+ D)`	Consecutive defined crime reports with a repeating sub-sequence
`((A|B) (C|D))`	Two consecutive crime reports with two alternative definitions for each report
`(A Z* (B+|C+) Z* D)`	Defined first and last; two alternatives for a repeating sub-sequence

- `IndexNFA`: the index-accelerated `MATCH_RECOGNIZE` [34]. It is only applicable when there exist both primary clustered index and secondary indexes. In a nutshell, it (1) uses indexes to first identify feasible ranges, then (2) executes the NFA-based `MATCH_RECOGNIZE` on the feasible ranges.

The rewrite rule and cost model was implemented as SQL rewrite in Python, using statistics obtained from Trino (`SHOW STATS`) and SQL Server (`DBCC SHOW_STATISTICS`).

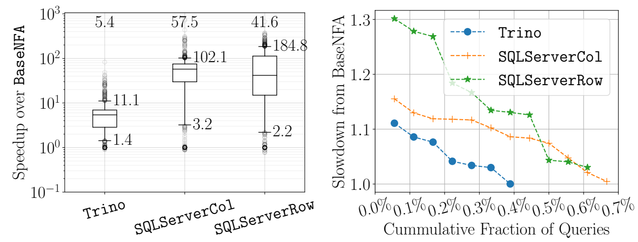
Host Systems We tested the following host systems:

- `Trino` (v373): a distributed SQL query engine with a NFA-based `MATCH_RECOGNIZE` implementation, i.e., `BaseNFA`. `Trino` connects to separate storage via connectors. In this evaluation we mainly use the Hive connector [9] to access data stored as ORC files on Hadoop Distributed File System (HDFS) because this is the most common setup for `Trino`.
- `SQLServer` (2019): a commercial database from Microsoft. Because `SQLServer` currently does not support `MATCH_RECOGNIZE`, we implemented a NFA-based `MATCH_RECOGNIZE` (i.e., `BaseNFA`), as a user-defined aggregate (UDA), using the augmented finite automaton (AFA) [11]. We experimented with two physical layouts supported by `SQLServer`: (1) `SQLServerCol`, column store created using the "clustered columnstore index" [4], and (2) `SQLServerRow`, row store with clustered indexes on timestamp and secondary indexes on query columns.
- `Flink` (v1.14.4): a stream-processing engine with a SQL API supporting `MATCH_RECOGNIZE` [8], implemented using `BaseNFA`.

Platform We conduct experiments on a Windows 11 PC with Intel® Core™ i7-9800X CPU @3.80GHz and 64GB memory at 2666MHz. All host systems are run with all available cores (8).

5.2 End-to-End Performance Improvement

We compared `JoinNFA` against the baselines hosted in `Trino` and `SQLServer`. We evaluated 18 query templates for the `Crimes` dataset. For each of the pattern definitions, we generated 100 query instances by uniformly sampling [District] and [Primary_Type], while skipping [Primary_Type] with less than 10k reports to avoid empty matches. For `DyGeoBox`, the longitudinal difference between a specified report and the first report in the same match must be less than 0.025 based on the original definition [34]. Same for latitude.



(a) Speedups of `JoinNFA` **(b) Regression analysis (Sec 5.3.2)**

Figure 9: Performance by host systems

5.2.1 Performance across host systems. We first look at the overall speedups of `JoinNFA` over `BaseNFA` in the three host systems we tested, each covering 1,800 query instances across 6 patterns and 3 pattern variable definitions. Figure 9a is a box-plot[7] of our results. Out of the three, `Trino` saw the lowest median speedup, 5.4× with 95% of query instances seeing speedup better than 1.4×. `SQLServerCol` hit the highest median speedup of 57.5× and the highest 5^{th} percentile of 3.2×. Both of them use column-oriented storage which allows the use of bucketized prefilter (Section 3.2). The hash join between buckets and the input table in the bucketized prefilter has relatively stable performance as it depends primarily on the size of the input table.

`SQLServerRow` experienced the highest 95^{th} percentile speedup of 184.8× and a second highest 5^{th} percentile of 2.2×. `SQLServerRow` uses the basic prefilter (Section 3.1) so the prefilter's output rows are produced by nested loop join between the time ranges and the input table using the index on timestamp, and the resulting performance is primarily influenced by the number of time ranges produced. Hence, we see a wider distribution of speedups in `SQLServerRow`.

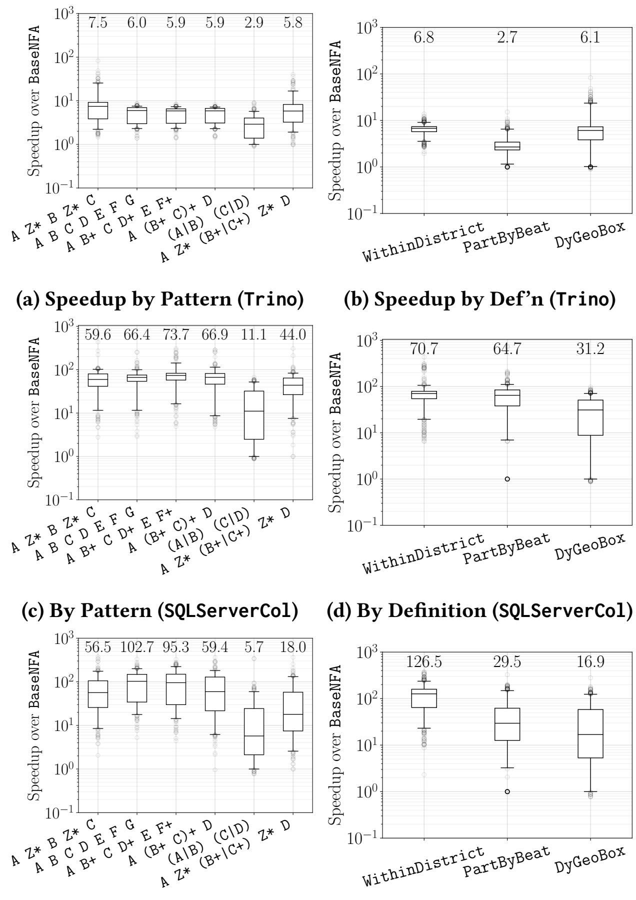
Comparing host systems, the prefiltering strategy yields more performance gain when integrated with host systems that are faster at executing joins, such as `SQLServer`. In addition, column stores' performance is generally more stable.

5.2.2 Performance across different patterns. We report the speedups of `JoinNFA` across different patterns and pattern variable definitions, and results are illustrated in Figure 10.

Grouping the results by different patterns, as shown in Figure 10a, 10c and 10e, it is clear that the median speedups are mostly in the same order of magnitude across patterns. The notable exceptions are the pattern `((A|B) (C|D))`, which has the lowest speedup in all host systems, and the pattern `(A Z* (B+|C+) Z* D)` has the second lowest speedup in all. What they have in common is that both contain alternation, which introduces unions in the prefilter (Section 3.1.2). Having unions in the prefilter leads to larger join cost and larger input to the `MATCH_RECOGNIZE` step, thus lowering the eventual speedup.

Grouping the results by different pattern definitions, as in Figure 10b, 10d and 10f, we get a different perspective. Across all host systems, `WithinDistrict` has the highest median speedup, up to 126.5×. This is because the independent conditions on [District]

[7] Each box shows the 25^{th} percentile (lower border), median (middle line), and 75^{th} percentile (upper border), with the median annotated at the top; lower and upper whiskers extend to the 5^{th} and 95^{th} percentiles respectively; the remaining are dots.

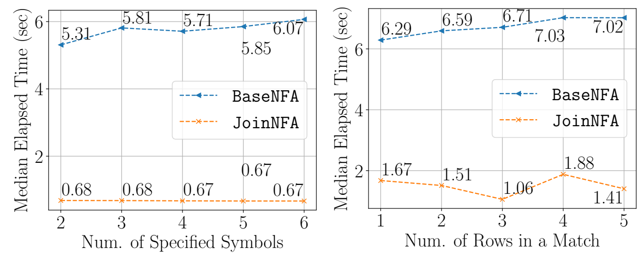


(a) Speedup by Pattern (Trino) **(b) Speedup by Def'n (Trino)**

(c) By Pattern (SQLServerCol) **(d) By Definition (SQLServerCol)**

(e) By Pattern (SQLServerRow) **(f) By Definition (SQLServerRow)**

Figure 10: JoinNFA speedups grouped by 6 patterns and 3 pattern definitions



(a) (A Z* B), (A Z* B Z* C), ... **(b)** (A|B), ((A|B) (C|D)), ...

Figure 11: Median query time versus pattern length for patterns defined using WithinDistrict (Trino)

combined with those on [Primary_Type] greatly reduced the selectivity of input rows to the prefilter, making the join much cheaper to execute. PartByBeat does not have the conditions on [District], making the input to the prefilter larger. We note that in Trino, since the native implementation of MATCH_RECOGNIZE already executes partitions defined by PARTITION BY in parallel, the speedup is more limited as the benefit of parallel join execution is less significant. DyGeoBox has the lowest speedup in SQLServer. This is because not only it does not have very selective independent conditions, the dependent conditions among participating rows in a match are purely inequality conditions. For Trino and SQLServerCol, they can still rely on the equality conditions on buckets in their bucketized prefilters to have efficient joins. For SQLServerRow however, because there is no equality join condition, the performance of the joins in the prefilter can suffer, thus DyGeoBox's median speedup is a magnitude lower than the other two definitions.

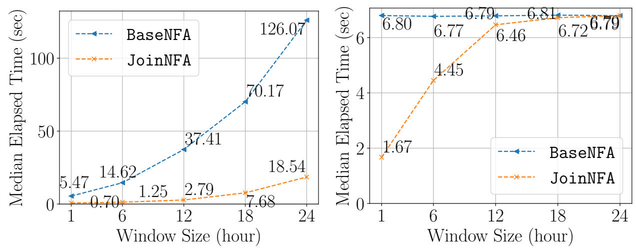
5.2.3 Impact of pattern length and window size. Now we investigate JoinNFA for different pattern lengths and window sizes.

To construct variable pattern lengths, we took the pattern (A Z* B Z* C) from [34], and varied its length by changing the number of defined symbols (i.e., symbols other than Z) from 2 to 6, as going beyond 6 all queries have no match. We also took the alternation pattern ((A|B) (C|D)), and varied its length by changing the number of rows in a match, e.g., (A|B) matches exactly one row, ((A|B) (C|D)) matches exactly two rows, and so on. We tested the patterns using WithinDistrict, 30-minute window, and for every length, we generated 100 query instances using the random process discussed earlier. We report the median query times in Figure 11. In summary, JoinNFA maintains its higher performance over BaseNFA as pattern length increases.

The results show an increasing trend for the query time of BaseNFA with respect to increasing pattern length. This is because for the two patterns we tested, the longer the pattern the more possible partial matching states that need to be processed by an NFA-based MATCH_RECOGNIZE operator, thus taking more time.

For the (A Z* B Z* C) family of patterns, the median time of JoinNFA is approximately constant for all lengths. This is because the prefilter uses at most 3 symbols, so the effect of increasing pattern length has no impact on its execution time. For the ((A|B) (C|D)) family of patterns, the median time of JoinNFA decreases until the length reaches 3, and then increases. When the length increases from 1 to 3, the prefilter becomes more selective because there are more symbols in the pattern providing more constraints – conditions on two rows is more strict than on one. However, at length 3 and above, selecting an optimal symbol set combination becomes very difficult: the possible combinations to construct the union prefilter is 7^8 (7 possible symbol sets per path in the NFA graph and 8 possible paths in total). The optimizer stops at 100 distinct combinations (using ~25ms), thus it is unlikely to find the optimal so the performance regresses slightly, although still maintaining at least 3.7× speedup.

Let us turn to the effect of window size. For both patterns in their original form, we varied the window size from 1 to 24 hours to "stress test" our approach. For (A Z* B Z* C), the median query time increases from 5.4s to 126s for BaseNFA as window size increases from 1 to 24 hour, due to more matches and partial matching states allowed by larger windows. In comparison, the increase in the median query time for JoinNFA is much less – from 0.7s to 18.5s, due to the prefilter aggressively pruning the input to the NFA-based operator. A different picture is shown for ((A|B) (C|D)): BaseNFA's median query time barely moves but JoinNFA's increases and converges to the BaseNFA's. Window size does not



(a) (A Z* B Z* C) **(b)** ((A|B) (C|D))

Figure 12: Median query time versus window size for patterns defined using WithinDistrict (Trino)

Table 2: Median percentage reduction in speedup from the true optimal plan for (A Z* B Z* C)**.**

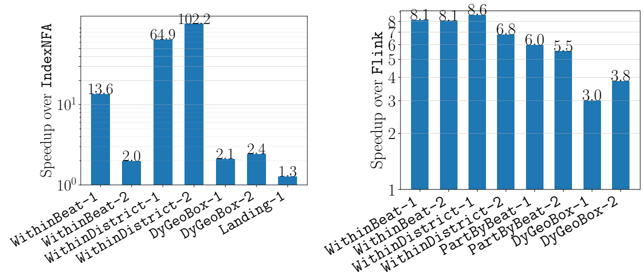
Pattern Definition	Trino	SQLServerCol	SQLServerRow
WithinDistrict	-21%	-13%	-0.0%
PartByBeat	-27%	-0.0%	-5.3%
DyGeoBox	-0.0%	-0.0%	-0.0%

affect the NFA-based operator's execution because ((A|B) (C|D)) matches two consecutive rows so the number of partial matches during execution is bounded at two – a partial match of either an A or a B. For JoinNFA, the median query time increases because prefilter cannot enforce the consecutive condition (e.g., A or B is followed *directly* by C or D). The median query time of JoinNFA gradually converges to that of BaseNFA because the estimated cost of rewrite becomes higher than no rewrite, and JoinNFA's optimizer chooses to avoid rewrite and revert to BaseNFA.

5.3 Cost Model Evaluation

So far we have discussed the end-to-end speedup given by JoinNFA integrated with our cost model. In this section, we dive deeper to the effectiveness of our cost model when it comes to choosing symbol sets for rewrite, and to avoiding regression.

5.3.1 Speedup Analysis. First, to understand the effectiveness of our cost model when it comes to choosing a good symbol set, for every query instance, we compared the results of JoinNFA which selects a symbol set for rewrite (or no rewrite) using our cost model, with that of the true optimal plan selected based on the actual execution times of all possible rewrites including no rewrite. We used only pattern (A Z* B Z* C) because finding the true optimal plan takes excessive amount of time for some other patterns. For each query instance, we calculated the percentage reduction in speedup from the optimal. Table 2 lists the medians of percentage reductions. The median reduction in speedup is at most -21% in Trino; at most -13% in SQLServerCol, and at most -5.3% in SQLServerRow. This tells us that the optimizer was not always picking the best prefilter. Nevertheless, the optimizer still displays respectable utility, e.g., at -27% median reduction for WithinDistrict in Trino, the median speedup for (A Z* B Z* C) is 2.3×, down from 3.8× achieved by the optimal prefilter. Moreover, for DyGeoBox, all of the query instances observed 0% reduction – the optimal prefilter were picked.



(a) JoinNFA vs. IndexNFA, both using SQLServerRow **(b) JoinNFA (Trino) vs. Flink, both using PostgreSQL storage**

Figure 13: JoinNFA speedups over IndexNFA and Flink

5.3.2 Regression Analysis. A regression happens when JoinNFA is slower than BaseNFA– the chosen rewritten plan ends up being slower than no rewrite. Figure 9b shows the slowdowns of queries experiencing regression. Specifically, 7 and 12 out of 1,800 queries respectively experienced regression for Trino and SQLServerCol, and their largest slowdowns were 1.11× (from 7.3s to 8.2s) for Trino and 1.15× (from 8.9s to 10.3s) for SQLServerCol. For SQLServerRow, 11 out of 1,800 queries experienced regressions but only 3 of them saw greater than 1.2× slowdowns, and the largest slowdown were 1.3× (from 10.5s to 13.7s). In summary, the regressions are insignificant comparing to the performance improvement.

5.4 Comparison with Existing Systems

To put our proposal in the context of existing work in speeding up MATCH_RECOGNIZE, we compare with IndexNFA [34] and Flink. For Crimes dataset we used pattern (A Z* B Z* C) from [34], and their pattern definition WithinBeat (same as WithinDistrict except specifying the [Beat] rather than [District] for each report). This is in IndexNFA's favor because beat is a smaller patrol unit than district so an index on [Beat] has lower selectivity.

5.4.1 Comparison with IndexNFA*.* We compared JoinNFA with IndexNFA [34] on SQLServerRow, since IndexNFA requires access to data with a clustered index on timestamp and secondary indexes for other query attributes. Specifically, we followed the original work's procedure [34] to run IndexNFA on databases: we first ran IndexNFA on its original Java-based engine [5] to obtain ranges, then joined the ranges to the input table on SQLServerRow, and executed the NFA-based MATCH_RECOGNIZE on the resulting rows. To be more favorable to IndexNFA, we only counted the time for index selection and feasible range generation, and the time for executing MATCH_RECOGNIZE on SQLServer. We omitted the time for reading indexes and the time for importing ranges to SQLServer.

Figure 13a shows the speedups: JoinNFA outperformed IndexNFA up to 102× with a median of 2.4×. Because JoinNFA can incorporate dependent conditions on query attributes such as longitude and latitude while IndexNFA cannot, its prefilter has more pruning power. Take DyGeoBox 1 as an example, the number of rows after the prefilter in JoinNFA is 10× less than IndexNFA when both selected symbol set (A, B, C). We used the original parameter sets from [5] for WithinBeat 2. For WithinBeat 1, we kept the same

[beat] and switched the [Primary_Type] conditions of A and B in WithinBeat 1, and we did the same to create two parameter sets for each of WithinDistrict, which uses the parent district of the beat, and DyGeoBox. IndexAccel does not support PARTITION BY in MATCH_RECOGNIZE hence it cannot run PartByBeat.

5.4.2 Comparison with Flink*.* Because Flink is a streaming system, it is not easy to integrate JoinNFA with it. For a fair comparison, we connected both Flink and Trino to a PostgreSQL database with the benchmark tables stored in column store via the Citus extension [3]. Figure 13b shows the speedups: JoinNFA on Trino outperformed BaseNFA on Flink with a median speed up of 6.4×.

6 CONCLUSION

In this work we explored using a Join-based prefilter to accelerate MATCH_RECOGNIZE in relational systems under historical analysis setting. To realize this approach with minimal system integration effort, we put forward 1) a logical plan rewrite rule to implement the prefilter using symbols and conditions extracted from the original query, and 2) a cost model to choose a subset of symbols for prefilter construction. In experiments we observed 5.4× to 57.5× median query time speedups over the NFA-based MATCH_RECOGNIZE implementations on Trino (v373) and SQL Sever (2019), using a benchmark of 1,800 query instances. It performed better than the index-based prefilter [34] on their benchmarks when indexes were available. In the future, we will investigate further speedup potential through operator-level parallelism.

REFERENCES

[1] 2022. https://github.com/ekzhu/trino-interval-merge-group-agg.
[2] 2022. https://gist.github.com/ekzhu/78f027a0bfc123c6f5aca99409846256.
[3] 2022. Citus. https://github.com/citusdata/citus.
[4] 2022. Columnstore indexes: Overview. https://docs.microsoft.com/en-us/sql/relational-databases/indexes/columnstore-indexes-overview.
[5] 2022. Index Accelerated Pattern Matching on Persistent Event Streams. https://github.com/sigmod2021-index-pattern/index-pattern.
[6] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Trans. Database Syst.* 42, 4 (2017), 20:1–20:44. https://doi.org/10.1145/3129246
[7] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. 2008. Efficient pattern matching over event streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, Jason Tsong-Li Wang (Ed.). ACM, 147–160. https://doi.org/10.1145/1376616.1376634
[8] Apache Flink. 2022. Pattern Recognition. https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/dev/table/sql/queries/match_recognize/.
[9] Brian Olsen. 2020. A gentle introduction to the Hive connector. https://trino.io/blog/2020/10/20/intro-to-hive-connector.html.
[10] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, John C. Platt, James F. Terwilliger, and John Wernsing. 2014. Trill: A High-Performance Incremental Query Processor for Diverse Analytics. *Proc. VLDB Endow.* 8, 4 (2014), 401–412. https://doi.org/10.14778/2735496.2735503
[11] Badrish Chandramouli, Jonathan Goldstein, and David Maier. 2010. High-Performance Dynamic Pattern Matching over Disordered Streams. *Proc. VLDB Endow.* 3, 1 (2010), 220–231. https://doi.org/10.14778/1920841.1920873
[12] Chicago. 2022. Chicago Crimes. https://www.kaggle.com/chicago/chicago-crime.
[13] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. 2007. 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*. www.cidrdb.org, 412–422. http://cidrdb.org/cidr2007/papers/cidr07p47.pdf
[14] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. 1991. An Evaluation of Non-Equijoin Algorithms. In *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*, Guy M. Lohman, Amílcar Sernadas, and Rafael Camps (Eds.). Morgan Kaufmann, 443–452. http://www.vldb.org/conf/1991/P443.PDF
[15] Yanlei Diao, Neil Immerman, and Daniel Gyllstrom. 2007. Sase+: An agile language for kleene closure over event streams. *UMass Technical Report* (2007).
[16] Nihal Dindar, Peter M. Fischer, Merve Soner, and Nesime Tatbul. 2011. Efficiently correlating complex events over live and archived data streams. In *Proceedings of the Fifth ACM International Conference on Distributed Event-Based Systems, DEBS 2011, New York, NY, USA, July 11-15, 2011*, David M. Eyers, Opher Etzion, Avigdor Gal, Stanley B. Zdonik, and Paul Vincent (Eds.). ACM, 243–254. https://doi.org/10.1145/2002259.2002293
[17] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek R. Narasayya, and Surajit Chaudhuri. 2019. Selectivity Estimation for Range Predicates using Lightweight Models. *Proc. VLDB Endow.* 12, 9 (2019), 1044–1057. https://doi.org/10.14778/3329772.3329780
[18] Felipe Hoffa. 2021. Funnel analytics with SQL: MATCH_RECOGNIZE() on Snowflake. https://towardsdatascience.com/funnel-analytics-with-sql-match-recognize-on-snowflake-8bd576d9b7b1.
[19] Michael J. Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting Worst-Case Optimal Joins in Relational Database Systems. *Proc. VLDB Endow.* 13, 11 (2020), 1891–1904. http://www.vldb.org/pvldb/vol13/p1891-freitag.pdf
[20] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2009. *Database systems - the complete book (2. ed.).* Pearson Education.
[21] Nikos Giatrakos, Elias Alevizos, Alexander Artikis, Antonios Deligiannakis, and Minos N. Garofalakis. 2020. Complex event recognition in the Big Data era: a survey. *VLDB J.* 29, 1 (2020), 313–352. https://doi.org/10.1007/s00778-019-00557-w
[22] Goetz Graefe. 1993. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.* 25, 2 (1993), 73–170. https://doi.org/10.1145/152610.152611
[23] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29. http://sites.computer.org/debull/95SEP-CD.pdf
[24] Goetz Graefe and William J. McKenna. 1993. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria*. IEEE Computer Society, 209–218. https://doi.org/10.1109/ICDE.1993.344061
[25] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman, and Hamid Pirahesh. 1989. Extensible Query Processing in Starburst. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, USA, May 31 - June 2, 1989*, James Clifford, Bruce G. Lindsay, and David Maier (Eds.). ACM Press, 377–388. https://doi.org/10.1145/67544.66962
[26] ISO/IEC JTC 1/SC 32 Data management and interchange. 2016. 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.
[27] Kasia Findeisen. 2021. Row pattern recognition with MATCH_RECOGNIZE. https://trino.io/blog/2021/05/19/row_pattern_matching.html.
[28] Keith Laker. 2017. MATCH_RECOGNIZE and predicates - everything you need to know. https://blogs.oracle.com/datawarehousing/post/match_recognize-and-predicates-everything-you-need-to-know.
[29] Mahmoud Abo Khamis, Hung Q. Ngo, Christopher Ré, and Atri Rudra. 2016. Joins via Geometric Resolutions: Worst Case and Beyond. *ACM Trans. Database Syst.* 41, 4 (2016), 22:1–22:45. https://doi.org/10.1145/2967101
[30] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. 1977. Fast Pattern Matching in Strings. *SIAM J. Comput.* 6, 2 (1977), 323–350. https://doi.org/10.1137/0206024
[31] Ilya Kolchinsky and Assaf Schuster. 2018. Join Query Optimization Techniques for Complex Event Processing Applications. *Proc. VLDB Endow.* 11, 11 (2018), 1332–1345. https://doi.org/10.14778/3236187.3236189
[32] Ilya Kolchinsky and Assaf Schuster. 2019. Real-Time Multi-Pattern Detection over Event Streams. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 589–606. https://doi.org/10.1145/3299869.3319869
[33] Ilya Kolchinsky, Izchak Sharfman, and Assaf Schuster. 2015. Lazy evaluation methods for detecting complex events. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, Oslo, Norway, June 29 - July 3, 2015*, Frank Eliassen and Roman Vitenberg (Eds.). ACM, 34–45. https://doi.org/10.1145/2675743.2771832
[34] Michael Körber, Nikolaus Glombiewski, and Bernhard Seeger. 2021. Index-Accelerated Pattern Matching in Event Stores. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1023–1036. https://doi.org/10.1145/3448016.3457245
[35] Rundong Li, Wolfgang Gatterbauer, and Mirek Riedewald. 2020. Near-Optimal Distributed Band-Joins through Recursive Partitioning. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.).

ACM, 2375–2390. https://doi.org/10.1145/3318464.3389750
[36] Mo Liu, Elke A. Rundensteiner, Daniel J. Dougherty, Chetan Gupta, Song Wang, Ismail Ari, and Abhay Mehta. 2011. High-performance nested CEP query processing over event streams. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan (Eds.). IEEE Computer Society, 123–134. https://doi.org/10.1109/ICDE.2011.5767839
[37] Michael V. Mannino, Paicheng Chu, and Thomas Sager. 1988. Statistical Profile Estimation in Database Systems. *ACM Comput. Surv.* 20, 3 (1988), 191–221. https://doi.org/10.1145/62061.62063
[38] Marta Paes. 2019. MATCH_RECOGNIZE: where Flink SQL and Complex Event Processing meet. https://www.ververica.com/blog/match_recognize-where-flink-sql-and-complex-event-processing-meet.
[39] Yuan Mei and Samuel Madden. 2009. ZStream: a cost-based query processor for adaptively detecting composite events. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul (Eds.). ACM, 193–206. https://doi.org/10.1145/1559845.1559867
[40] Microsoft Azure. 2021. MATCH_RECOGNIZE (Stream Analytics). https://docs.microsoft.com/en-us/stream-analytics-query/match-recognize-stream-analytics.
[41] Hung Q. Ngo, Dung T. Nguyen, Christopher Ré, and Atri Rudra. 2014. Beyond worst-case analysis for joins with minesweeper. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*, Richard Hull and Martin Grohe (Eds.). ACM, 234–245. https://doi.org/10.1145/2594538.2594547
[42] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case Optimal Join Algorithms. *J. ACM* 65, 3 (2018), 16:1–16:40. https://doi.org/10.1145/3180143
[43] Oracle. 2022. Pattern Recognition With MATCH_RECOGNIZE. https://docs.oracle.com/en/middleware/fusion-middleware/osa/19.1/cqlreference/pattern-recognition-match_recognize.html.
[44] Raghu Ramakrishnan, Donko Donjerkovic, Arvind Ranganathan, Kevin S. Beyer, and Muralidhar Krishnaprasad. 1998. SRQL: Sorted Relational Query Language. In *10th International Conference on Scientific and Statistical Database Management, Proceedings, Capri, Italy, July 1-3, 1998*, Maurizio Rafanelli and Matthias Jarke (Eds.). IEEE Computer Society, 84–95. https://doi.org/10.1109/SSDM.1998.688114
[45] Medhabi Ray, Chuan Lei, and Elke A. Rundensteiner. 2016. Scalable Pattern Sharing on Event Streams. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 495–510. https://doi.org/10.1145/2882903.2882947
[46] Maximilian Reif and Thomas Neumann. 2022. A Scalable and Generic Approach to Range Joins. *Proc. VLDB Endow.* 15, 11 (2022), 3018–3030. https://www.vldb.org/pvldb/vol15/p3018-reif.pdf
[47] Rodrigo Alves. 2019. Azure Stream Analytics now supports MATCH_RECOGNIZE. https://azure.microsoft.com/en-us/blog/azure-stream-analytics-now-supports-match-recognize/.
[48] Reza Sadri, Carlo Zaniolo, Amir M. Zarkesh, and Jafar Adibi. 2004. Expressing and optimizing sequence queries in database systems. *ACM Trans. Database Syst.* 29, 2 (2004), 282–318. https://doi.org/10.1145/1005566.1005568
[49] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*, Philip A. Bernstein (Ed.). ACM, 23–34. https://doi.org/10.1145/582095.582099
[50] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. 1994. Sequence Query Processing. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA, May 24-27, 1994*, Richard T. Snodgrass and Marianne Winslett (Eds.). ACM Press, 430–441. https://doi.org/10.1145/191839.191926
[51] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. 1995. SEQ: A Model for Sequence Databases. In *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, Philip S. Yu and Arbee L. P. Chen (Eds.). IEEE Computer Society, 232–239. https://doi.org/10.1109/ICDE.1995.380388
[52] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. 1996. The Design and Implementation of a Sequence Database System. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda (Eds.). Morgan Kaufmann, 99–110. http://www.vldb.org/conf/1996/P099.PDF
[53] Snowflake. 2021. Identifying Sequences of Rows That Match a Pattern. https://docs.snowflake.com/en/user-guide/match-recognize-introduction.html.
[54] Valery Soloviev. 1993. A Truncating Hash Algorithm for Processing Band-Join Queries. In *Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria*. IEEE Computer Society, 419–427. https://doi.org/10.1109/ICDE.1993.344039
[55] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2019. C-store: a column-oriented DBMS. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, Michael L. Brodie (Ed.). ACM / Morgan & Claypool, 491–518. https://doi.org/10.1145/3226595.3226638
[56] Trino. 2022. MATCH_RECOGNIZE. https://trino.io/docs/current/sql/match-recognize.html.
[57] Todd L. Veldhuizen. 2014. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*, Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy (Eds.). OpenProceedings.org, 96–106. https://doi.org/10.5441/002/icdt.2014.13
[58] Eugene Wu, Yanlei Diao, and Shariq Rizvi. 2006. High-performance complex event processing over streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis (Eds.). ACM, 407–418. https://doi.org/10.1145/1142473.1142520

A AN OVERVIEW OF MATCH_RECOGNIZE

In this section, we provide an overview of MATCH_RECOGNIZE. First we focus on its syntax, and then we discuss the NFA implementation. For the full specification of MATCH_RECOGNIZE please refer to the SQL Standard [26].

A.1 Syntax and Semantics

Let us start with an example of MATCH_RECOGNIZE query. Figure 14 is a query on the Chicago Crimes data set [12], (see Figure 15 for a synthetic reproduction), and each row of the table represents a crime. The query is to detect sequences of possibly related crimes in the ordered domain specified by ORDER BY. Specifically, the pattern refers to a sequence of "ROBBERY", "BATTERY", and "MOTOR VEHICLE THEFT" that occurred within 30 minutes starting from and ending in the same latitude-longitude "box" centered at the location of the first crime.

The pattern is expressed using a regular expression style notation in the PATTERN clause. Each symbol is defined as a set of Boolean conditions through the DEFINE clause specifying when a row can be matched to the symbol (G is undefined thus matches any row). Each of R, B, and M matches exactly one row, and Z* is a Kleene Star matching 0 or more rows, indicating there can be other crimes between the crimes of interest. The PATTERN clause implicitly states the sequential constraints among the symbols. For example, because R comes before B in the expression, R.datetime <= B.datetime is true for any match.

Unlike in Complex Event Processing (CEP) languages, there is no WITHIN clause for defining windows in the SQL standard for MATCH_RECOGNIZE[8]. So in this example a time window is defined inside the DEFINE clause. While different systems may have a slightly different syntax for MATCH_RECOGNIZE (e.g., WITHIN clause in Apache Flink [8]), we focus on the specifications in the SQL Standard [26].

The MEASURES clause specifies the schema of the output table, and the schema lists columns computed based on every matching sequence. Given a matching sequence, R.id maps to the id column of the row matching symbol R in the matching sequence, and B.id maps to the id column of the row matching symbol B and so on. It is also possible to define a computed column in MEASURE as an aggregation over the rows in a match sequence. For example, count(Z.id) counts the number of rows matching symbol G in the matching sequence, and count(*) counts the total number of rows in the matching sequence. Figure 15 shows the output table of the query in Figure 14.

The ONE ROW PER MATCH clause specifies that for each matched sequence of rows, output exactly one row with the schema specified in MEASURES. Alternatively, user can state ALL ROWS PER MATCH, which will output all the rows in the matched sequence. The difference between these two output choices is significant but not essential to the understanding of our technical contributions. Lastly, AFTER MATCH SKIP TO NEXT ROW specifies the after-match skipping policy: after finding a complete match, the matching process "moves back" to the first row of the match, skip to the next row, and then restart search from there, the implication is that for every row there can be at most one matching sequence starting from there.

[8]Some streaming systems actually support WITHIN, e.g., Apache Flink.

```
SELECT * FROM Crimes MATCH_RECOGNIZE (
    ORDER BY datetime
    MEASURES R.id AS RID, B.id AS BID,M.id AS MID,count(Z.id) AS GAP
    ONE ROW PER MATCH
    AFTER MATCH SKIP TO NEXT ROW
    PATTERN (R Z* B Z* M)
    DEFINE R AS R.primary_type = 'ROBBERY',
        B AS B.primary_type = 'BATTERY'
            AND B.lon BETWEEN R.lon - 0.05 AND R.lon + 0.05
            AND B.lat BETWEEN R.lat - 0.02 AND R.lat + 0.02,
        M AS M.primary_type = 'MOTOR VEHICLE THEFT'
            AND M.lon BETWEEN R.lon - 0.05 AND R.lon + 0.05
            AND M.lat BETWEEN R.lat - 0.02 AND R.lat + 0.02
            AND M.datetime - R.datetime <= INTERVAL '30' MINUTE)
```

Figure 14: A MATCH_RECOGNIZE query on Chicago Crimes data set looking for potentially related sequences of crimes.

id	datetime	primary_type	lat	lon
1	1/2/2018 5:30	ASSULT	41.69	-87.66
2	1/2/2018 5:35	ROBBERY	41.10	-87.50
3	1/2/2018 5:40	BURGLARY	41.34	-87.57
4	1/2/2018 5:45	ROBBERY	41.13	-87.55
5	1/2/2018 5:50	ASSULT	41.25	-87.61
6	1/2/2018 5:55	BATTERY	41.12	-87.51
7	1/2/2018 6:00	NARCOTICS	41.17	-87.59
8	1/2/2018 6:05	MOTOR VEHICLE THEFT	41.11	-87.53
9	1/2/2018 6:10	OTHER OFFENCE	41.18	-87.56

(a) Input table

RID	BID	MID	GAP
2	6	8	4
4	6	8	2

(b) Output table

Figure 15: Example data for the query in Figure 14.

Alternatively, user can state AFTER MATCH SKIP PAST LAST ROW to restart search from the row after the last row of the previous match – potentially result in less matches. In actual implementations, the matching process may not need to "move back" due to the use of NFA as described in the next section.

We have skipped the optional PARTITION BY clause, which we introduce in Appendix C. Interested readers can find more details about the syntax in the SQL Standard [26] or Trino's documentation [56], which implements MATCH_RECOGNIZE according to the SQL Standard.

A.2 NFA and MATCH_RECOGNIZE Implementation

A common implementation of MATCH_RECOGNIZE is using Nondeterministic Finite Automaton (NFA). A NFA consumes an input table row-by-row while maintaining a set of partial matches as matching state. The matching logic of NFA is dictated by its state transition graph compiled from the user-specified query pattern. Given the graph and the partial matches, NFA knows which symbol in the PATTERN (and its associated conditions in DEFINE) to match, and if succeeded, what is the next symbol to match. Figure 16 shows the compiled state transition graph from the example query in Figure 14. The following example illustrates how does NFA work.

We use the NFA state transition graph in Figure 16 to execute pattern matching on the data in Figure 15. As specified by the query, the rows are consumed in the ascending order of datetime. Initially there is no partial match, so we only check for the conditions

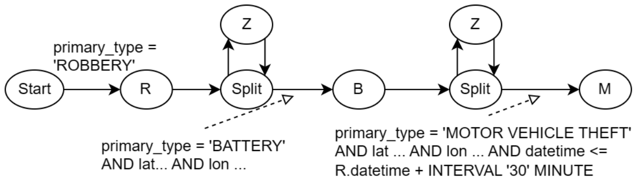


Figure 16: The NFA state transition graph compiled from query in Figure 14.

required for the first transition to state R, and we found row 2, adding it to our partial matches. Now we check for both the conditions for the transition to B and to G. For each, if successful, the current partial match is extended accordingly to generate a new partial match. If neither is successful, this partial match is failed and discarded. We also check for the condition of the transition to R to see of we can start a new partial match from the beginning of the pattern because the after-match skipping policy states that a match can start from any row. Keep going on like this until row 8 and we have partial matches `R G G G B G` starting from row 2, `R G B G` starting from row 4, `R G G G G G` from row 2, and `R G G G` from row 4. The transitions to M and G are checked for row 8 and the former is successful, so we promote the first two partial matches to complete matches. We can discard the other two partial matches because at most one match can start from any row.

B PROOF OF CORRECTNESS

In this section we prove the correctness of the Join-based prefilters. To recap, a prefilter filters the input table to a NFA-based `MATCH_RECOGNIZE`, so the input size to the latter is reduced. To maintain the correctness of the final results, the prefilter must preserve matched rows.

B.1 Proof for Basic Prefilter (Special Case)

For a basic prefilter P_X (Definition 3.7) constructed using symbol set $X = [s_{i_1}, s_{i_2}, ..., s_{i_k}\}$, Proposition 3.2 states that given a special pattern Q involving only concatenation operators, the results returned by `MATCH_RECOGNIZE` on the original input relation T is the same as that on the prefiltered relation P_X, i.e., $MR(Q,T) = MR(Q,P_X)$. In essence, any row in a match must be part of P_X. This can be proved in four cases of the tuple-valued function $f(t_1, t_k)$ in Definition 3.7.

Let t be the primary `ORDER BY` attribute, w be the window size of the pattern window condition if exists, and $\chi = [s_1, s_2, ..., s_n]$ be the set of all symbols in the query pattern Q. Let $R = [r_1, r_2, ..., r_n]$, $R \subseteq T$ be a sequence of rows ordered by t matching the query pattern, i.e., each r_i matches s_i. Each $s_{i_j} \in X$ thus matches one row in R, i.e., r_{i_j}. old, since pattern Q contains concatenation operators only and thus no symbol in symbol set X matches an absent row.

Case 1: $i_1 = 1 \wedge i_k = n$

In this case r_1 matches s_{i_1} and r_n matches s_{i_k}, i.e., $r_1 = r_{i_1}$ and $r_n = r_{i_k}$. Following the relational algebra expression in Definition 3.7 to analyze (r_1, r_n):

(1) $r_{i_1} \in \sigma_{C_{s_{i_1}}}(T)$ because $C_{s_{i_1}}$ is the set of independent conditions (Definition 3.1) that can be evaluated on a row matching s_{i_1} alone. Since r_{i_1} matches s_{i_1}, it must also satisfies $C_{s_{i_1}}$. Similarly, we have $r_{i_k} \in \sigma_{C_{s_{i_k}}}(T)$
(2) $(r_{i_1}..r_{i_j}..r_{i_k}) \in \sigma_{C_{s_{i_1}}}(T) \bowtie_{C_{s_{i_1},s_{i_2}}} \sigma_{C_{s_{i_2}}}(T) \bowtie ... \bowtie_{C_{s_{i_1}..s_{i_k}}} \sigma_{C_{s_{i_k}}}(T)$ because $R = [r_1, r_2, ..., r_n]$ is a match of the pattern, it must satisfies all the dependent conditions $C_{s_{i_1},s_{i_2}} ... C_{s_{i_1}..s_{i_k}}$.
(3) $(r_1.t, r_n.t)$ must be in the temporal range produced by $f(t_1, t_k)$, which is evaluated as $(r_{i_1}.t, r_{i_k}.t)$ because $r_1 = r_{i_1}$ and $r_n = r_{i_k}$.

Case 2: $\exists w \wedge i_k = n$

In this case s_{i_k} matches the last row r_n, i.e., $r_n = r_{i_k}$. We first follow the same step (1) and (2) as in Case 1 and then:

(3) $(r_1.t, r_n.t)$ must be within the temporal range produced by $f(t_1, t_k)$ which is evaluated as $(r_{i_k}.t - w, r_{i_k}.t)$. Because $r_1.t \geq r_n.t - w$ by applying the pattern window (Definition 3.6) and $r_n = r_{i_k}$, $(r_1.t, r_n.t) \subset (r_{i_k}.t - w, r_{i_k}.t)$.

Case 3: $\exists w \wedge i_1 = 1$

In this case, s_{i_1} matches the first row r_1, i.e., $r_1 = r_{i_1}$. We first follow the same step (1) and (2) as in Case 1 and then:

(3) $(r_1.t, r_n.t)$ must be within the temporal range produced by $f(t_1, t_k)$ which is evaluated as $(r_{i_1}.t, r_{i_1}.t + w)$. Because $r_n.t \leq r_1.t + w$ by applying the pattern window (Definition 3.6) and $r_1 = r_{i_1}$, $(r_1.t, r_n.t) \subset (r_{i_1}.t, r_{i_1}.t + w)$.

Case 4: $\exists w$

In this case, s_{i_1} matches some row r_{i_1}, s_{i_k} matches some row r_{i_k}, and $r_1.t < r_{i_1}.t \leq r_{i_k}.t < r_n.t$. We first follow the same step (1) and (2) as in Case 1 and then:

(3) $(r_1.t, r_n.t)$ must be within the temporal range produced by $f(t_1, t_k)$ which is evaluated as $(r_{i_k}.t - w, r_{i_1}.t + w)$. Because $r_{i_k}.t - w \leq r_1.t$ and $r_n \leq r_{i_1} + w$ by applying the pattern window (Definition 3.6), $(r_1.t, r_n.t) \subset (r_{i_k}.t - w, r_{i_1}.t + w)$.

Since in all four cases of $f(t_1, t_k)$, $(r_1.t, r_n.t)$ is within the temporal range produced by $f(t_1, t_k)$ and $r_1.t \leq r_i.t \leq r_n.t$ for any $r_i \in R$, we can conclude that r_i must be in the join result of $\rho_{\text{start,end}}(\pi_{f(t_1,t_k)}) \bowtie_{\text{start} \leq t \leq \text{end}} T$. That is, $r_i \in P_X, \forall r_i \in R$. Therefore, for any set of rows $R \subseteq T$ matching the query pattern, $R \subseteq P_X$, and Proposition 3.2 holds. □.

B.2 Proof for Bucketized Prefilter (Special Case)

The bucketized prefilter enjoys the same property as the basic prefilter, stated by the following proposition.

Proposition B.1. *Given an input relation T, a special pattern Q involving only Concatenation operators on an ordered set of pattern symbols χ, and an ordered symbol set $X \subseteq \chi$, if Q contains a pattern window condition ($\exists w$) and all dependent conditions C_χ are self-contained, $MR(Q,T) = MR(Q,B_X)$.*

To prove the above proposition, we can utilize the fact that a bucketized prefilter is simply using the bucket ranges produced by a basic prefilter with a pattern window condition. So we only have to show that for any $r_i \in P_X$, if $\exists w$, $r_i \in B_X$, that is $P_X \subseteq B_X$. This can be done by analyzing the four cases of $g(t_1, t_k)$.

Let t be the primary `ORDER BY` attribute, β be the derived `bucket` column with $\beta = \lfloor t/w \rfloor$, w be the window size of the pattern window condition, and χ be the set of all symbols in the query pattern Q. $\left(\sigma_{C_{s_{i_1}}}(T) \bowtie_{C_{s_{i_1}, s_{i_1}}} \sigma_{C_{s_{i_1}}}(T) \bowtie \ldots \bowtie_{C_{s_{i_1}, \ldots, s_{i_k}}} \sigma_{C_{s_{i_k}}}(T)\right)$ in Definition 3.7 and $\left(\sigma_{C_{s_1, \ldots, s_k}}\left(\sigma_{C_{s_{i_1}}}(T) \bowtie_{b_{s_{i_1}, s_{i_1}}} \ldots \bowtie_{b_{s_{i_1} \ldots s_{i_k}}} \sigma_{C_{s_{i_k}}}(T)\right)\right)$ in Definition 3.10 have the same output relation, since the conditions involved are essentially the same. Let $R_S = [r_{i_1}, .., r_{i_j}, .., r_{i_k}]$ denote one row in this output relation, where r_{i_j} refers to the corresponding record from $\sigma_{C_{s_{i_j}}} T$, i.e., r_{i_j} matches s_{i_j}. Now we know that the input relations of the last Join in P_X (Definition 3.7) and B_X (Definition 3.10) are the same, we can simply show the Join condition in B_X ($\bowtie_{\text{start} \leq \beta \leq \text{end}}$) is loose than that in P_X ($\bowtie_{\text{start} \leq t \leq \text{end}}$) to prove $P_S \subset B_S$.

Case 1: $\exists w \wedge i_1 = 1 \wedge i_k = n$

For any $r \in T$ satisfying $r_{i_1}.t \leq r.t \leq r_{i_k}.t$ (the last Join condition in P_X), it naturally satisfies $\lfloor r_{i_1}.t/w \rfloor \leq \lfloor r.t/w \rfloor \leq \lfloor r_{i_k}.t/w \rfloor$ and thus $r_{i_1}.\beta \leq r.\beta \leq r_{i_k}.\beta$ (the last Join condition in B_X). That is, if $r \in P_X, r \in B_X$

Case 2: $\exists w \wedge i_k = n$

For any $r \in T$ satisfying $r_{i_k}.t - w \leq r.t \leq r_{i_k}.t$ (the last Join condition in P_X), it naturally satisfies $\lfloor r_{i_k}.t/w - 1 \rfloor \leq \lfloor r.t/w \rfloor \leq \lfloor r_{i_k}.t/w \rfloor$ and thus $r_{i_k}.\beta - 1 \leq r.\beta \leq r_{i_k}.\beta$ (the last Join condition in B_X). That is, if $r \in P_X, r \in B_X$

Case 3: $\exists w \wedge i_1 = 1$

For any $r \in T$ satisfying $r_{i_1}.t \leq r.t \leq r_{i_1}.t + w$ (the last Join condition in P_S), it naturally satisfies $\lfloor r_{i_1}.t/w \rfloor \leq \lfloor r.t/w \rfloor \leq \lfloor r_{i_1}.t/w + 1 \rfloor$ and thus $r_{i_1}.\beta \leq r.\beta \leq r_{i_1}.\beta + 1$ (the last Join condition in B_X). That is, if $r \in P_X, r \in B_X$

Case 4: $\exists w$

For any $r \in T$ satisfying $r_{i_k}.t - w \leq r.t \leq r_{i_1}.t + w$ (the last Join condition in P_X), it naturally satisfies $\lfloor r_{i_k}.t/w - 1 \rfloor \leq \lfloor r.t/w \rfloor \leq \lfloor r_{i_1}.t/w + 1 \rfloor$ and thus $r_{i_k}.\beta - 1 \leq r.\beta \leq r_{i_1}.\beta + 1$ (the last Join condition in B_S). That is, if $r \in P_X, r \in B_X$

Since in all four cases of $g(t_1, t_k)$, $r \in B_X, \forall r \in P_X$, we can conclude that $P_X \subseteq B_X$ if there exists a pattern window condition, and Proposition B.1 holds. □

B.3 Proof for Basic Prefilter (General Case)

In this section we prove the general case for the basic prefilter: Proposition 3.3, which follows the two steps described in Section 3.1.2: 1) Decomposition and 2) Prefilter Union.

Algorithm 2 lists the pseudo code for decomposing a general pattern into special patterns. Given a pattern Q, we first construct a corresponding NFA state transition graph using NFAGraph and then perform a depth-first graph traversal to generate all possible paths from the start node to finish node. A special pattern is constructed from each path using SpecialPattern, which removes functional nodes such as Start.

Algorithm 2: Decompose(Q)

Input: A query pattern Q
Output: A set of special patterns $\{Q_1, Q_2, ..., Q_m\}$

```
1  G ← NFAGraph(Q);
2  res = {} ;                        // res stores all matching paths
3  ss.push({G.Start}) ;                      // Initialize stack
4  while !ss.empty() do
5      p ← ss.pop(); u ← p.tail() ;       // u is the last node
6      for v ∈ u.out do
7          if v is Finish then
8              res.add(SpecialPattern(p)); continue
9          if v is Split(+) then continue ;
10         if v is Split(*) and v ∈ p then continue ;
11         ss.push(p + {v})
12 return res
```

Example 4. We walk through Algorithm 2 using an example in Figure 4b. Given a query Q=((A | B) C+), we first construct its NFA in Figure 4b (line 1). We next start a DFS graph traversal on G, initializing the stack ss with a dummy partial path $\{Start\}$ (line 3). In the 1^{st} iteration of DFS (line 5), $p = \{Start\}$ and $u = Start$. We then expand the partial matching path p by including u's out-neighbor $v = Split(|)$ (line 11). Moving on to the 2^{nd} iteration, $p = \{Start, Split(|)\}$, $u = Split(|)$, and u has two out neighbors: $\{A, B\}$. We expand p with A and B respectively, and add them into stack ss (line 11). Now we have two partial matching paths in ss: {Start, Split(|), A} and {Start, Split(|), B}. Similarly, in the 3^{rd} iteration, $p = \{Start, Split(|), B\}$, $u = B$, and p is expanded with C. By the end of the 3^{rd} iteration, We have $ss = [\{Start, Split(|), A\}, \{Start, Split(|), B, C\}]$. In the 4^{th} iteration, $p = \{Start, Split(|), B, C\}$, $u = C$, and u has two out neighbors, i.e., $Finish$ and $Split(+)$. When $v = Finish$, we obtain one matching path $\{Start, Split(|), B, C, Finish\}$, remove functional nodes, and add (BC) to res (line 8); when $v = Split(+)$, we simply continue (line 9). By the end of the 4^{th} iteration, We have $ss = [\{Start, Split(|), A\}]$ and $res = [(BC)]$. Algorithm 2 finally returns $res = [(BC), (AC)]$ as depicted in Figure 4b. Without line 9-10 in Algorithm 2, more matching paths would get generated, e.g., (A C C) and (B C C) as shown in gray color in Figure 4b. We note that any prefilter for pattern `(A C)` can serve as the prefilter for `(A C C)`. Similarly for `(B C C)` and `(B C)`. Thus, it is sufficient to only consider `(A C) ∪ (B C)` during the prefiltering.

The next step after decomposition is prefilter union. Let Algorithm 2 return special patterns $\{Q_1, Q_2, ..., Q_m\}$ and let χ_j be the ordered set of pattern symbols in each Q_j. For each Q_j, construct a prefilter P_{X_j} following Definition 3.7 where $X_j \subseteq \chi_j$. A prefilter for pattern Q can be constructed as $\cup_{j=1}^{m} P_{X_j}$.

Now we can prove the following statement from Proposition 3.3: *Given an input relation T and a query pattern Q, we have $MR(Q, T) = MR(Q, \cup_{j=1}^{m} P_{X_j})$.*

Proof. We use the fact that an NFA is equivalent to a regular expression query Q, that is, any sequence of rows that match Q corresponds to a path from `Start` to `Finish` in an NFA graph compiled from Q and vice versa. Therefore, a pattern Q is equivalent

to a union of special patterns $\{Q_j\}$, each corresponds to a path in the NFA graph of Q, i.e., $Q = \cup_{j=1}^{m'} Q_j$ for some $m' \geq 1$. This can be shown by using the fact that a Kleene closure corresponds to a union of infinite number of Concatenations. For example, (A C+) can be written as (AC) ∪ (ACC) ∪ (ACCC) ∪ ....
Thus, running on Q is equivalent to the union of the results of running on all of $Q_1, Q_2, ...$[9], i.e.,

$$MR(\cup_{j=1}^{m'} Q_i, T) = \cup_{j=1}^{m'} MR(Q_i, T) \tag{8}$$

Using this property, we have:

$$MR(Q,T) = MR(\cup_{j=1}^{m'} Q_j, T) = \cup_{j=1}^{m'} MR(Q_j, T) = \cup_{j=1}^{m'} MR(Q_j, P_{X_j}) \tag{9}$$

The first equation holds since $Q = \cup_{j=1}^{m'} Q_j$; the second equation is derived based on Equation 8; and the last equation is based on Proposition 3.2, where X_j is a symbol subset of pattern Q_j.
Algorithm 2 without the escaping logic for cycles (line 9-10) generates the complete set $\cup_{j=1}^{m'} Q_j$; while Algorithm 2 generates a subset of special patterns $\cup_{j=1}^{m} Q_j$, where $m \leq m'$. By analyzing the escaping logic, we know $\forall Q_l$, there exists Q_j such that Q_j is the prefix of Q_l where $1 \leq l \leq m'$ and $1 \leq j \leq m$, and hence any symbol subset X_j of pattern Q_j is also a valid symbol subset of pattern Q_l. For instance, $m = 2$ in Figure 4b; for $Q_3 = (A\ C\ C)$, there exists $Q_1 = (A\ C)$ which is a prefix of Q_3; let $X_1 = \{A\}$, we can see X_1 is also a valid symbol subset of Q_3. For $\forall 1 \leq j \leq m$, let $\delta(j) = \{l | Q_j \text{ is prefix of } Q_l, 1 \leq l \leq m'\}$. Based on the above observation, $\delta(j)$ are disjoint and $\cup_{j=1}^{m} \delta(j) = [1, 2, ...m']$.
Using the above, we can derive the equality in Proposition 3.3:

$$\begin{aligned}
MR(Q,T) &= \cup_{l=1}^{m'} MR(Q_l, P_{X_l}) \\
&= \cup_{j=1}^{m} \cup_{l \in \delta(j)} MR(Q_l, P_{X_j}) \\
&= \cup_{j=1}^{m} MR(\cup_{l \in \delta(j)} Q_l, P_{X_j}) \\
&= \cup_{j=1}^{m} MR(\cup_{l \in \delta(j)} Q_i, \cup_{j=1}^{m} P_{X_j}) \\
&= MR(\cup_{j=1}^{m} \cup_{l \in \delta(j)} Q_i, \cup_{j=1}^{m} P_{X_j}) \\
&= MR(Q, \cup_{j=1}^{m} P_{X_j})
\end{aligned}$$

where P_Q =. The first equation is from Equation 9; the second equation is based on the definition and properties of $\delta(j)$; the third, fifth, and the last equation are all derived using Equation 8; and the fourth equation holds since $P_{X_j} \subseteq \cup_{j=1}^{m} P_{X_j}$ and thus $\cup_{j=1}^{m} P_{X_j}$ is a valid prefilter for $\forall j$. Hence, we have $MR(Q,T) = MR(Q, \cup_{j=1}^{m} P_{X_j})$. □

C THE REWRITE RULE WITH PARTITION BY

MATCH_RECOGNIZE supports the optional PARTITION BY clause to perform row pattern matching within independent partitions of the input table [26]. For example, Figure 17 shows a query with PARTITION BY district on the Chicago Crimes dataset [12]. Unlike the previous example in Figure 14, this query, logically, first split the input table into partitions of rows by the column district, so within each partition all rows have the same district value; then it essentially run the same query as before on each partition independently. Thus, the R.id, B.id and M.id in each match result will have the same district.

```
SELECT * FROM Crimes MATCH_RECONGIZE (
    PARTITION BY district
    ORDER BY datetime
    MEASURES R.id AS RID, B.id AS BID,M.id AS MID,count(Z.id) AS GAP
    ONE ROW PER MATCH
    AFTER MATCH SKIP TO NEXT ROW
    PATTERN (R Z* B Z* M)
    DEFINE R AS R.primary_type = 'ROBBERY',
        B AS B.primary_type = 'BATTERY'
            AND B.lon BETWEEN A.lon - 0.05 AND A.lon + 0.05
            AND B.lat BETWEEN A.lat - 0.02 AND A.lat + 0.02,
        M AS M.primary_type = 'MOTOR VEHICLE THEFT'
            AND M.lon BETWEEN A.lon - 0.05 AND A.lon + 0.05
            AND M.lat BETWEEN A.lat - 0.02 AND A.lat + 0.02
            AND M.datetime - R.datetime <= INTERVAL '30' MINUTE)
```

Figure 17: A MATCH_RECOGNIZE query on Chicago Crimes data set looking for potentially related sequences of crimes with PARTITION BY.

Our previous definitions for prefilter (Definition 3.7 and 3.10) do not handle the case when the query specifies PARTITION BY. However all it required is a few simple modifications. Essentially, we want the prefilter to incorporate the PARTITION BY key in every step given a partition key h:

(1) an equality dependent condition for each pair of symbol s_{i_j}, s_{i_l} in the given symbol set: $s_{i_j}.h = s_{i_l}.h$.
(2) a partition_key column prefixing the range.
(3) an equality Join condition at the Filtering step partition_key = h.

For the basic prefilter (Definition 3.7), the new definition with partition key is follow:

Definition C.1. *Following Definition 3.7, if a query has a partition key h, the **prefilter** P_X of this symbol set can be constructed as:*

$$P_X = \rho_{partition\_key, t_s, t_e}\Big(\pi_{h_1, f(t_1, t_k)}\Big(\rho_{t_1/t, h_1/h}(\sigma_{C_{s_{i_1}}}(T)) \bowtie_{C_{s_{i_1}, s_{i_2}}} \dots \bowtie_{C_{s_{i_1}, \dots, s_{i_k}}} \rho_{t_k/t}(\sigma_{C_{s_{i_k}}}(T)) \Big)\Big) \bowtie_{t_s \leq t \leq t_e,\ partition\_key = h} T$$

Figure 18 shows an example of a rewrite with the modifications mentioned earlier.

For the bucketized prefilter, the new definition is as follow:

Definition C.2. *Following Definition 3.10, if a query has a partition key h, the **bucketized prefilter** B_X for symbol set $X = [s_{i_1}, s_{i_2}, .., s_{i_k}]$*

[9]With different specifications in AFTER MATCH SKIP clause, the union operator, i.e., right-hand side of Equation 8, differs slightly. E.g., with AFTER MATCH SKIP TO NEXT ROW, the union operator deduplicates all results that share the first matching row.

```sql
WITH ranges AS (
    SELECT R.district as partition_key,
        R.datetime as range_start, M.datetime as range_end
    FROM Crimes AS R, Crimes AS M
    WHERE R.primary_type = 'ROBBERY'
        AND M.primary_type = 'MOTOR VEHICLE THEFT'
        AND M.lon BETWEEN A.lon - 0.05 AND A.lon + 0.05
        AND M.lat BETWEEN A.lat - 0.02 AND A.lat + 0.02
        AND R.datetime <= M.datetime
        AND M.datetime - R.datetime <= INTERVAL '30' MINUTE
        AND R.district = M.district
), prefilter AS (
    SELECT DISTINCT Crimes.* FROM Crimes, ranges AS r
    WHERE district = r.partition_key
        AND datetime BETWEEN r.range_start AND r.range_end
) SELECT * FROM prefilter MATCH_RECOGNIZE (/* same as before */);
```

Figure 18: A rewrite of Figure 17 with using symbol set {R, M}.

```sql
WITH input_partitioned AS (
    SELECT *, cast(datetime / '30' MINUTE AS bigint) AS bucket
    FROM Crimes
), ranges AS (
    SELECT R.district as partition_key,
        R.bucket as range_start, M.bucket as range_end
    FROM input_partitioned AS R, input_partitioned AS M
    WHERE R.bucket = M.bucket /* rest same as before */
    UNION
    SELECT R.bucket as range_start, M.bucket as range_end
    FROM input_partitioned AS R, input_partitioned AS M
    WHERE R.bucket + 1 = M.bucket /* rest same as before */
), buckets AS (
    SELECT DISTINCT partition_key, bucket
    FROM ranges
    CROSS JOIN UNNEST(Seq(range_start, range_end)) AS t(buckets)
), prefilter AS (
    SELECT i.* FROM input_partitioned AS i, buckets AS b
    WHERE i.district = b.partition_key AND i.bucket = b.bucket
) SELECT * FROM prefilter MATCH_RECOGNIZE (/* same as before */);
```

Figure 19: A rewrite of the query in Figure 17 using symbol set {R, M} using bucketized prefilter.

can be constructed as:

$$B_X = \pi_{partition\_key,bk}\Big(\rho_{partition\_key,bk_s,bk_e}\Big(\pi_{h_1,g(t_1,t_k)}\Big(\sigma_{C_{s_{i_1},\ldots,s_{i_k}}}\Big(\rho_{t_1/t,h_1/h}(\sigma_{C_{s_{i_1}}}(T)) \bowtie_{b_{s_{i_1},s_{i_2}}} \cdots \bowtie_{b_{s_{i_1},\ldots,s_{i_k}}} \rho_{t_k/t}(\sigma_{b_{s_{i_k}}}(T))\Big)\Big) \Big) \times Seq(bk_s, bk_e)_{bk}\Big) \bowtie_{bk,partition\_key=h} T$$

Figure 19 shows an exmaple rewrite that incorporates partition key.

D MERGE OVERLAPPING RANGES IN SQL

Definition 3.7 states the basic prefilter rewrite that requires the input table to be without duplicates. The reason for this requirement is to avoid removing the original duplicates in the input table during the last step of applying the de-duplication (δ) operator to remove new duplicates generated as a result of overlapping timestamp ranges.

To remove this requirement, we can add a new SQL function, MergeRange. The MergeRange function takes a relation of (t_s, t_e) and produces a relation of (t_s, t_e) such that all overlapping pairs are merged, for example:

$$\begin{aligned}&\text{MergeRange}(\{(t_{s_i}, t_{e_i}), (t_{s_j}, t_{e_j})\})\\ &\rightarrow \{(min(t_{s_i}, t_{s_j}), max(t_{e_i}, t_{e_j}))\} \text{ if } t_{e_i} \geq t_{s_j} \text{ and } t_{s_i} \leq t_{e_j}\end{aligned}$$

Given disjoint ranges, each row from the input table only joins with at most one range. So no new duplicate row is added, and the duplicate rows in the input table are preserved. A simple algorithm takes two steps: first sort the ranges in ascending order by their start positions; then do a single pass to merge: for each range, if its start is before the previous' end, merge it with the previous range. This algorithm can be added without touching the execution engine as either a user-defined window function with a two-value state [1], or, using built-in window functions, namely `LAG` and `LEAD` [2].

Using MergeRange, Definition 3.7 can be modified to accept input tabel with duplicate rows. Specifically, the expression for P_X can be rewritten as:

$$\begin{aligned}P_X = &\text{MergeRange}\Big(\pi_{f(t_1,t_k)}\Big(\rho_{t_1/t}\left(\sigma_{C_{s_{i_1}}}(T)\right) \bowtie_{C_{s_{i_1},s_{i_2}}} \sigma_{C_{s_{i_2}}}(T) \bowtie\\ &\ldots\sigma_{C_{s_{i_{k-1}}}}(T) \bowtie_{C_{s_{i_1},s_{i_2}\ldots s_{i_k}}} \rho_{t_k/t}\left(\sigma_{C_{s_{i_k}}}(T)\right)\Big)\Big) \bowtie_{t_s \leq t \leq t_e} T\end{aligned}$$

Another effect of merging is that it improves the overall performance by reducing the number of ranges in the join with the input table, leading to improved performance for the basic prefilter. In our experiments (Section 5), we employ this performance optimization for the basic prefilter when evaluating `SQLServerRow`.

The following SQL example showcases how one can use a user-defined window function, `MergeGroup` [1] to implement the MergeRange step of the rewrite, following the example in Figure 5.

```sql
WITH ranges AS (
    SELECT R.datetime as t_s, M.datetime as t_e
    FROM Crimes AS R, Crimes AS M
    WHERE R.datetime <= M.datetime
        AND R.primary_type = 'ROBBERY'
        AND M.primary_type = 'MOTOR VEHICLE THEFT'
        AND M.lon BETWEEN R.lon - 0.05 AND R.lon + 0.05
        AND M.lat BETWEEN R.lat - 0.02 AND R.lat + 0.02
        AND M.datetime - R.datetime <= INTERVAL '30' MINUTE
), merged_ranges AS (
    SELECT min(t_s) as t_s, max(t_e) as t_e FROM (
        SELECT MergeGroup(t_s, t_e) OVER (ORDER BY t_s) AS group_id,
            t_s, t_e FROM ranges
    ) AS r GROUP BY group_id
), prefilter AS (
    SELECT Crimes.* FROM Crimes, merged_ranges AS r
    WHERE datetime BETWEEN r.t_s AND r.t_e
) SELECT * FROM prefilter MATCH_RECOGNIZE (/* same as before */);
```

E COST MODELING FOR SYMBOL SET SELECTION

In this section, we first describe the detailed experimental setting in evaluating our specialized cardinality estimator for the prefilter and cost model for the `MATCH_RECOGNIZE` (Figure 8). Next, we illustrate how γ, i.e., the total number of transit function calls during NFA matching, varies. Lastly, we provide an example illustrating the window-based simulation for estimating γ_ψ in Section 4.2.2.

E.1 Setting for Cost Model Validation

Figure 8b shows the linear relationship between $CPU_{\texttt{MATCH\_RECOGNIZE}}$ and $C_{\texttt{MATCH\_RECOGNIZE}}$. The synthetic dataset used in Figure 8b is generated following [34] with 50M records and two columns (`datetime` and `value`). Column `datetime` is a number sequence from 1 to 50M; while column `value` is a piece-wise linear function of `datetime` with random noise added drawing from $N(0,1)$— the piece-wise linear function consists of two pieces: (1) for `datetime` from 1 to 25M, the linear function goes up from 0 to 1; and (2) from 25M to 50M, the linear function goes down from 1 to 0. The `MATCH_RECOGNIZE` query used in Figure 8b aims to match pattern (A B Z* C) within some time_window, where symbol `A` and `C` require the record has value smaller than some lower_threshold, while symbol `B` requires the record's value to be larger than some upper_threshold. We tested this query template with varying parameters and different candidate symbol sets – for each query, we measured its CPU time $CPU_{\texttt{MATCH\_RECOGNIZE}}$ for `MATCH_RECOGNIZE` and its true number of transit calls γ. We use $|r|$=18 bytes, as each column takes 9 bytes with 8 bytes for value and 1 byte for header. We then plot $CPU_{\texttt{MATCH\_RECOGNIZE}}$ vs. $C_{\texttt{MATCH\_RECOGNIZE}}=\gamma \times |r|$ in Figure 8b.

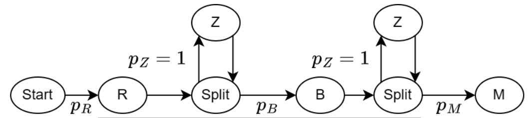
E.2 γ with Varying Query and Data

Example 5. (γ depends on the query) Continue the data example in Figure 15 and consider two different query pattern: (1) R; (2) R Z* B Z* M. For pattern (1), in each iteration (line 3 in Algorithm 1) only one transit call from `Start` state to `R` state is involved and thus the total number of transit calls γ equals to the number of input rows, i.e., 9 in Figure 15. However, for pattern (2), the number of transit calls differs from iteration to iteration. At iteration one and two (row 1 and 2), there is only one dummy partial match at `Start` state and only one transit function from `Start` to `R` is evaluated. At iteration three, there are two partial matches – one at `Start` state involving one transit call and one at `R` state involving two transit calls – and in total three transit functions get evaluated. Similar calculation can be performed for the remaining iterations. The number of transit calls in iteration 1-9 are [1, 1, 3, 3, 5, 5, 9, 9, 9]. In total, $\gamma = 45$. We can see γ in pattern (2) is 6× of that in pattern (1).

Example 6. (γ depends on the data) Continue the query in Figure 1 (R Z* B Z* M) and consider two different data: (1) 9 records in Figure 15; (2) 9 records and none of them has `primary_type='ROBBERY'`. For data (1), the total number of transit calls γ=45 as analyzed in Example 5. On the contrary, for data (2), since no records has `primary_type='ROBBERY'`, `partial_matches` in each iteration only contains one dummy partial match at `Start` state. As a result, only one transit call is required in each iteration and thus γ=9. We can see γ in data (1) is 6× of that in data (2).

E.3 Window-based Simulation

Example 7. (Window-based Simulation) Suppose w=30 minutes corresponds to ψ=7 records. Figure 20 depicts the simulation process



(a) NFA for Query (R Z* B Z* M) in Figure 1

Row\Node	Start	R	Z1	B	Z2	M
0	1					
1		p_R				
2			p_R	$p_R \cdot p_B$		
3			p_R	$p_R \cdot p_B$	$p_R \cdot p_B$	$p_R \cdot p_B \cdot p_M$
4			p_R	$p_R \cdot p_B$	$2p_R \cdot p_B$	$2p_R \cdot p_B \cdot p_M$
5			p_R	$p_R \cdot p_B$	$3p_R \cdot p_B$	$3p_R \cdot p_B \cdot p_M$
6			p_R	$p_R \cdot p_B$	$4p_R \cdot p_B$	$4p_R \cdot p_B \cdot p_M$
7			p_R	$p_R \cdot p_B$	$5p_R \cdot p_B$	$5p_R \cdot p_B \cdot p_M$

(b) Simulation of Matching Process in NFA

Figure 20: Estimating the Number of Transit Calls in NFA

starting from row 1 to row 7. Initially (row 0), one dummy partial match exists at state s_{start}. As row 1 comes in, the dummy partial match at state s_{start} reaches s_R with probability p_R. As a result, we have p_R at s_R and 0 at s_{start}. Next, after processing the second row, we have p_R at s_{G_1} and $p_R \cdot p_B$ at state s_B, since p_R partial matches are at s_R after row 1 and each s_R reaches s_{G_1} and s_B with probability 1 and p_B. Similarly, we can maintain the number of partial matches at each state after processing row 3-7. We note is that the loop between state s_G and s_{plit} blows up the number of partial matches in NFA, since each partial match at s_G will persists in s_G and meanwhile branches to its out state, e.g., s_B and s_M. The numbers are depicted in Figure 20 and we can calculate γ_ψ by summing them up and multiply by 2, since each state triggers two transit calls except s_{start} and s_M.

F OUR INSIGHTS ON JOIN VERSUS NFA

In this section we discuss our insights regarding why Joins as a prefilter followed by `MATCH_RECOGNIZE` tends to out-perform a NFA-based `MATCH_RECOGNIZE` by itself.

F.1 Reduction of Partial Matches

A partial match is an incomplete matching state that may be extended or terminated. Using the query in Figure 1, a sequence of rows matching (`R Z* B`) is a partial match. Kolchinskey and Schuster [31] introduced the idea of using partial match count to model the cost of CEP queries in streaming setting for both ordered executors (NFA) and tree-based executors (ZStream [39]). We apply the same idea to gain insights into Join versus NFA in historical analysis setting: for NFA, partial match count is the accumulated number of times that a partial match is created or extended during the life circle of the operator; for Join, it is the accumulated number of intermediate and final Join results.

We observed that Join tends to produce less partial matches due to (1) deferred evaluation of symbols, (2) flexible order of evaluation of symbols, and (3) use of indexes. Our observations were made in historical analysis setting but confirm previous studies for streaming settings [31–33, 39]. Let us use the query in Figure 1 as the basis for our discussion.

Deferred evaluation of symbols in our context means we use Join to execute row-matching for the selected symbol set (`{R, M}`) first and defer the execution for other symbols (`B` and `Z*`) to the `MATCH_RECOGNIZE` operator, which only runs on the rows that fall into the ranges marked by the prefilter. Put this into numbers: the original query produced in total 14,506,619 partial matches (with window condition propagated to `Z*`); for the rewrite using `{R, M}`, there are 244,101 rows matching `R`, 293,242 rows matching `M`, 533,508 ranges produced by joining `R` and `M`, 946,245 rows produced by the prefilter, and lastly 3,041,019 partial matches produced in `MATCH_RECOGNIZE`– total 5,058,115, about 2.8X less. At a higher level, deferred evaluation is similar to using indexes, which allows dependent conditions to be used for pruning rows of selected symbols. The difference is that deferred evaluation prunes the rows of deferred symbols using sequential and window conditions only.

Flexible order of evaluation means we let the host system to decide the order of the Joins in bucket generation, in contrast to NFA which forces a fixed order given by its state transition graph. If we swapped the `primary_type` of `R` and `B` so `R.primary_type = 'BATTERY'` and `B.primary_type = 'ROBBERY'`, the original query produces 40,956,087 partial matches, that is 2.8X more than before, because `'BATTERY'` is less selective comparing to `'ROBBERY'`. In comparison, the rewrite using `{R, B, M}` produces 7,757,062 partial matches before the swap and 9,601,788 partial matches after – not a big difference. It can be explained partially by looking at the Join order: before the swap the host system first joins `R` and `M` which have smaller cardinalities, then joins `B`; after the swap, `R` has higher cardinality, so the host system choose to join `M` and `B` first instead. This example shows that the number of partial matches produced by NFA can be heavily affected by the ordering of symbols, while for Join it is less so due to the flexible order of evaluation.

F.2 Parallelism

Support for operator-level parallelism [22] is also an important reason for using Join over pure NFA. For example, the rewrite using `{R, B, M}`'s elapsed time in Trino went from 9.8s on a single thread to 3.8s with multi-threading.

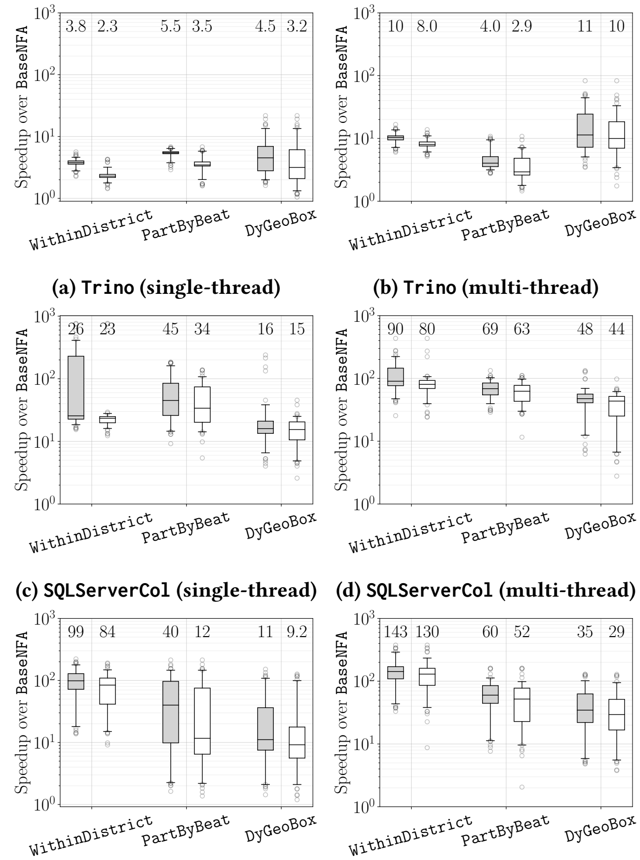
G EXPERIMENTS (CONTINUED)

This section reports additional experimental results supplementing Section 5.

G.1 End-to-End Performance Improvement (Continued)

In this section we continue the discussion in Section 5.2.

G.1.1 How good is the Join-based prefilter strategy? We evaluate the effectiveness of Join-based prefilter by itself without cost model in the picture, over different host systems and pattern definitions. We use two different settings for each host system: single-thread and multi-thread. In single-thread setting, we measure query time as CPU time; in multi-thread setting, we measure query time as elapsed time. Both measurements are obtained directly from the host systems' query statistics history. We find the query time of the true optimal prefilter (`OptJoinNFA`) for every query instances by executing all possible prefilters constructed from all possible symbol sets and find the one with the minimal query time. Figure 21 shows the results for pattern (`A Z* B Z* C`).



(a) `Trino` (single-thread) **(b) `Trino` (multi-thread)**

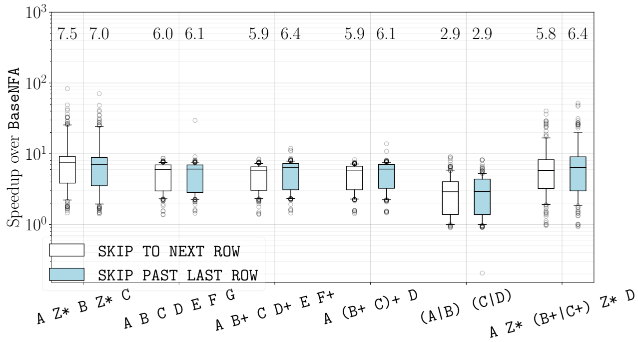
(c) `SQLServerCol` (single-thread) **(d) `SQLServerCol` (multi-thread)**

(e) `SQLServerRow` (single-thread) **(f) `SQLServerRow` (multi-thread)**

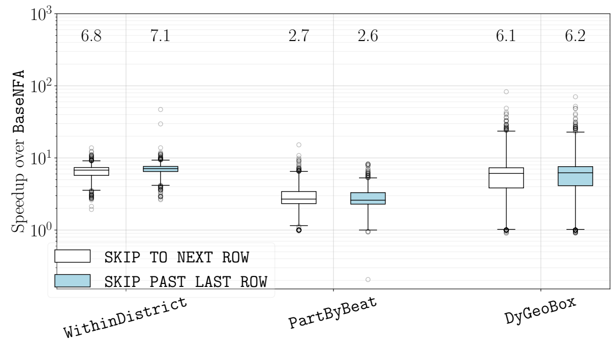
Figure 21: `OptJoinNFA` (shaded) `JoinNFA` (white) speedups for (`A Z* B Z* C`) with Single-thread and Multi-thread

From the single-thread result, the `OptJoinNFA` which uses the true optimal plan achieved up to 5.5× in `Trino`, up to 45× in `SQLServerCol` and 99× in `SQLServerRow`. Since parallelism is not in the picture here, the speedups are purely from the reduction of NFA computation through prefiltering rather than parallelizing the execution of joins using multi-cores. The use of prefiltering effectively reduces the input size to the NFA-based `MATCH_RECOGNIZE` operator, and due to query optimizaiton, the joins in prefilters are reordered and efficiently executed. This result confirms our intuition on why joins can be better than NFA as discussed in Appendix F.

From the multi-thread result, `OptJoinNFA` got higher speedups thanks to parallel execution of joins in the prefilters: up to 11× in `Trino`, up to 90× in `SQLServerCol` and 143× in `SQLServerRow`. Because `Trino`'s native implementation of `MATCH_RECOGNIZE` already executes partitions defined by `PARTITION BY` in parallel, the speedup for `PartByBeat` in `Trino` is limited and lower than that in the single-thread setting.



(a) JoinNFA speedups by patterns



(b) JoinNFA speedups by pattern definitions

Figure 22: JoinNFA speedups in Trino under AFTER MATCH SKIP TO NEXT ROW and AFTER MATCH SKIP PAST LAST ROW for all patterns and pattern definitions.

G.1.2 Does AFTER MATCH SKIP *policy impact performance?* The AFTER MATCH SKIP policy can be optionally defined for every MATCH_RECOGNIZE query to configure the procedural behavior after a valid match is found. For all queries in our earlier experiments in Section 5.2 we set the policy to AFTER MATCH SKIP TO NEXT ROW, which indicates that once a valid match starting at row i is found, the matching procedure is resumed starting from row $i + 1$ – the next row immediately after the first row of the match. Another often used option is AFTER MATCH SKIP PAST LAST ROW, which configures the matching procedure to resume from the first row after the last row of the previous match, i.e., after a match ends at row j is found, restart search from row $j + 1$.
To understand whether different AFTER MATCH SKIP policies impact performance, we measured the speedups of JoinNFA over BaseNFA when trying both aforementioned policies. The result is shown in Figure 22. Based on the result, there is no significant difference in performance for the two policies tried.

G.2 Cost Model Evaluation (Continued)

In this section we continue our experimental evaluation for the cost model in Section 5.3.

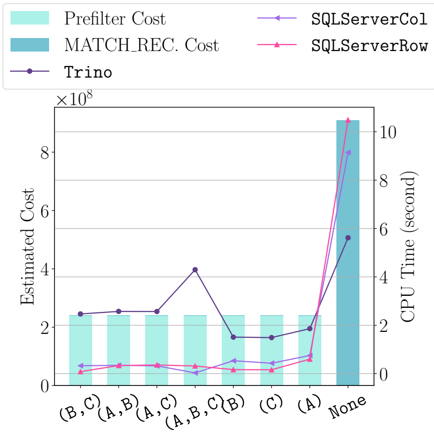
G.2.1 Accuracy Analysis. We dive into the comparison between estimated cost and CPU time. We focus on the pattern (A Z* B Z* C), ran the set of queries used in Section 5.4, and obtained CPU times from query stats in Trino and SQLServer. Table 3 lists the Pearson Correlation Coefficients (PCCs) and p-values measuring the linear correlation between the estimated costs and CPU times on benchmark queries.

Query	Trino		SQLServer			
			Col		Row	
	PCC	p-val	PCC	p-val	PCC	p-val
WithinBeat 1	0.787	$2e^{-2}$	0.998	$2e^{-8}$	0.999	$2e^{-9}$
WithinBeat 2	0.741	$4e^{-2}$	0.999	$8e^{-9}$	0.998	$3e^{-8}$
PartByBeat 1	0.704	$5e^{-2}$	0.847	$8e^{-3}$	0.786	$2e^{-2}$
PartByBeat 2	0.672	$7e^{-2}$	0.831	$1e^{-2}$	0.83	$1e^{-2}$
DyGeoBox 1	0.816	$1e^{-2}$	0.788	$2e^{-2}$	0.858	$6e^{-3}$
DyGeoBox 2	0.777	$2e^{-2}$	0.883	$3e^{-3}$	0.664	$7e^{-2}$

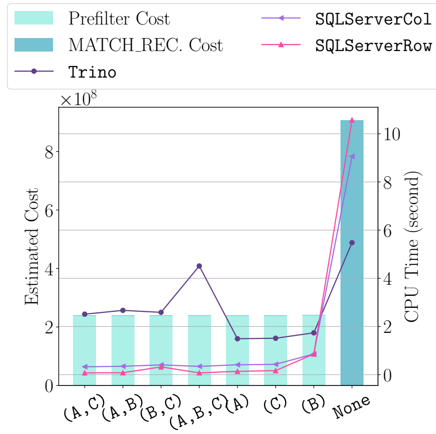
Table 3: Pearson Correlation Coefficient (PCC) and p-value (p-val) between the estimated cost and CPU time for pattern (A Z* B Z* C).

Figure 23 illustrates the comparison for all tested queries. The symbol sets in x-axis are ordered by their plan's estimated cost. In general, the CPU time grows with the estimated cost, but with some misalignment. For instance, {A, B, C} in Figure 23d underestimated cost in Trino. This might be due to the ScanFilterProject operator involved in the query plan's input stage for each symbol, i.e., A, B, and C, to load data, is not shared. This overhead is not considered in our cost model.

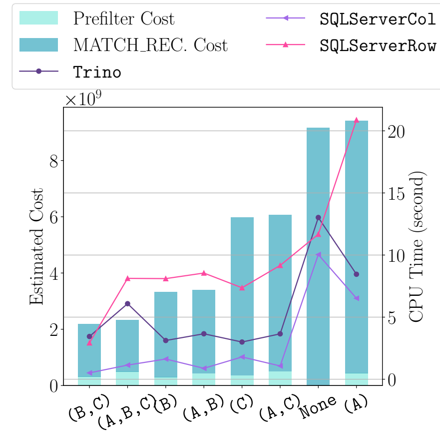
G.2.2 Can we replace the cost model with a heuristic? To investigate this, we turned off the cost model and executed rewrites constructed from all possible symbol sets for pattern (A Z* B Z* C). The results from 300 query instances on every host systems are shown in Figure 24- 32. Based on the results, we found no evidence that any particular symbol set is always optimal for any given host system. For example, Figure 27 and 30 show that on Trino, even though for PartByBeat the best symbol set seemed to be {C} with a median of 4×, {C} was less impressive on DyGeoBox with a median of just over 1×, less than {A,B,C}'s 10×. In comparison, the cost model of JoinNFA made good choices for both queries.



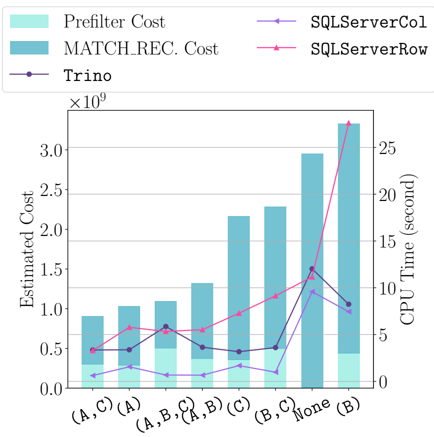
(a) **WithinBeat 1**



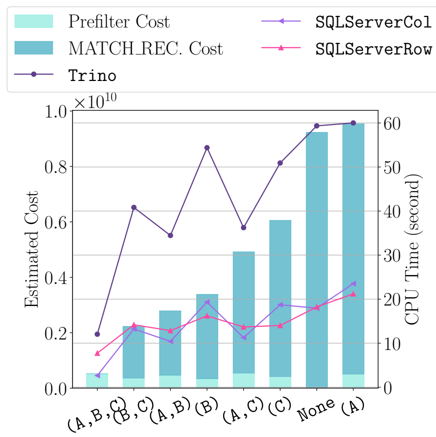
(b) **WithinBeat 2**



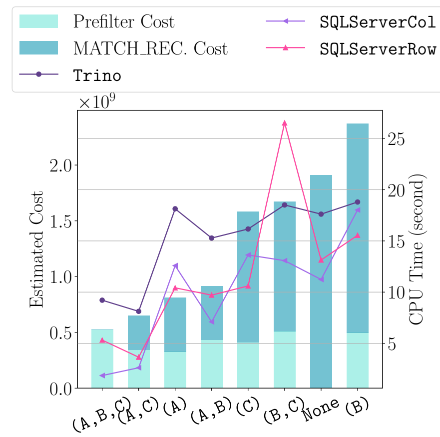
(c) **PartByBeat 1**



(d) **PartByBeat 2**

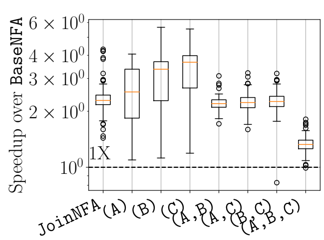


(e) **DyGeoBox 1**

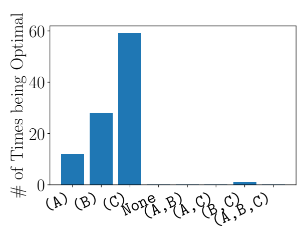


(f) **DyGeoBox 2**

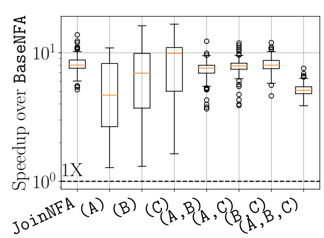
Figure 23: Estimated cost of rewrites created using different symbol sets and their actual CPU time.



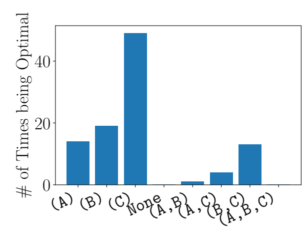
(a) Box plots of symbol sets' speedups (single thread).



(b) Histogram of optimal symbol sets (single thread).

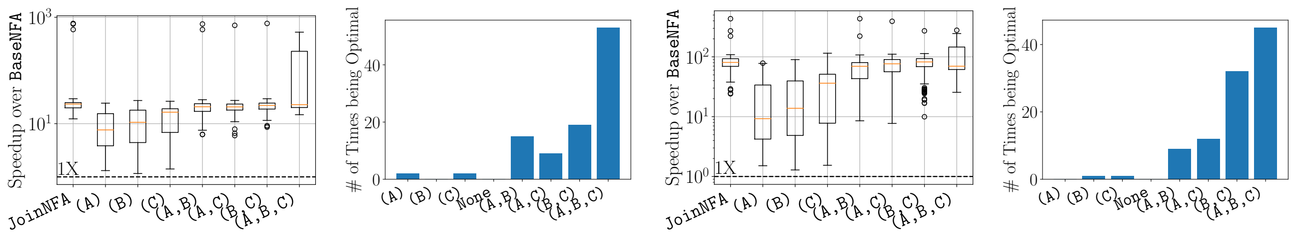


(c) Box plots of symbol sets' speedups (multiple threads).



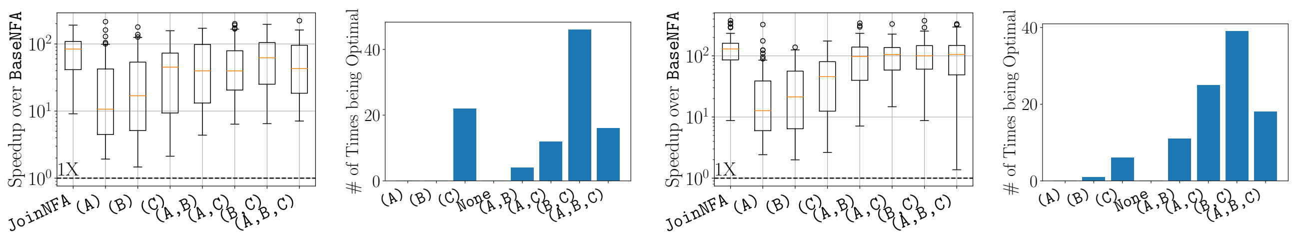
(d) Histogram of optimal symbol sets (multiple threads).

Figure 24: Statistics of each symbol set for WithinDistrict on Trino, over 100 random queries.



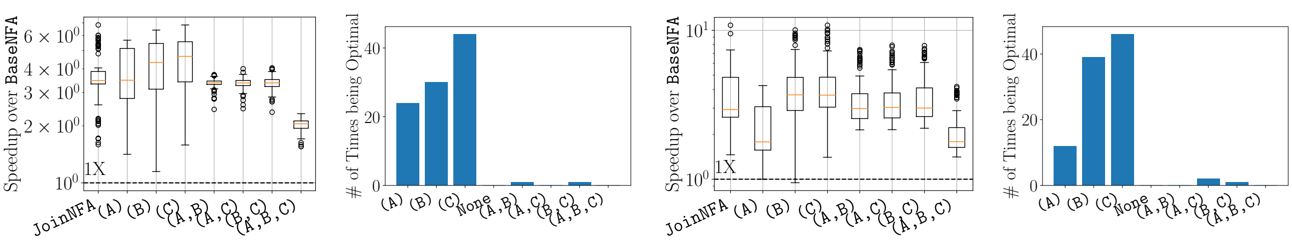
(a) Box plots of symbol sets' speedups (single thread). **(b) Histogram of optimal symbol sets (single thread).** **(c) Box plots of symbol sets' speedups (multiple threads).** **(d) Histogram of optimal symbol sets (multiple threads).**

Figure 25: Statistics of each symbol set for `WithinDistrict` on `SQLServer` + `Col`, over 100 random queries.



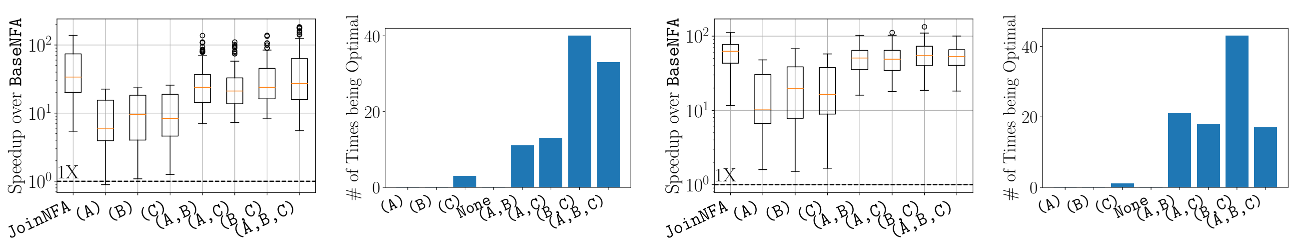
(a) Box plots of symbol sets' speedups (single thread). **(b) Histogram of optimal symbol sets (single thread).** **(c) Box plots of symbol sets' speedups (multiple threads).** **(d) Histogram of optimal symbol sets (multiple threads).**

Figure 26: Statistics of each symbol set for `WithinDistrict` on `SQLServer` + `Row`, over 100 random queries.



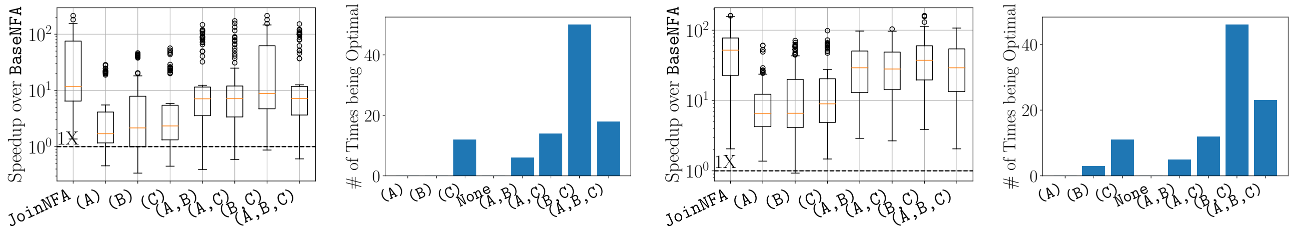
(a) Box plots of symbol sets' speedups (single thread). **(b) Histogram of optimal symbol sets (single thread).** **(c) Box plots of symbol sets' speedups (multiple threads).** **(d) Histogram of optimal symbol sets (multiple threads).**

Figure 27: Statistics of each symbol set for `PartByBeat` on `Trino`, over 100 random queries.



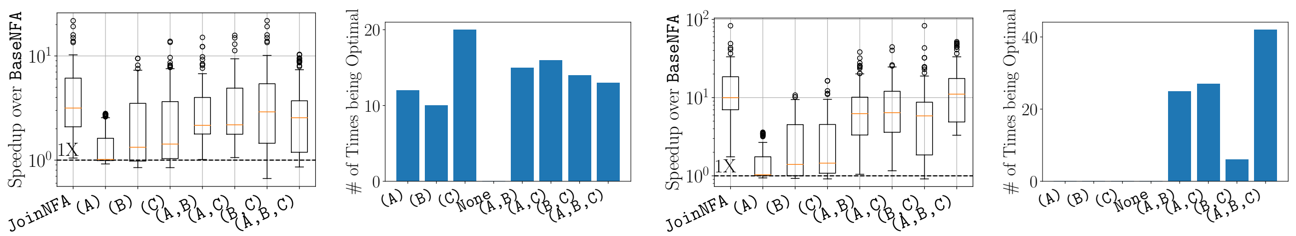
(a) Box plots of symbol sets' speedups (single thread). **(b) Histogram of optimal symbol sets (single thread).** **(c) Box plots of symbol sets' speedups (multiple threads).** **(d) Histogram of optimal symbol sets (multiple threads).**

Figure 28: Statistics of each symbol set for `PartByBeat` on `SQLServer` + `Col`, over 100 random queries.



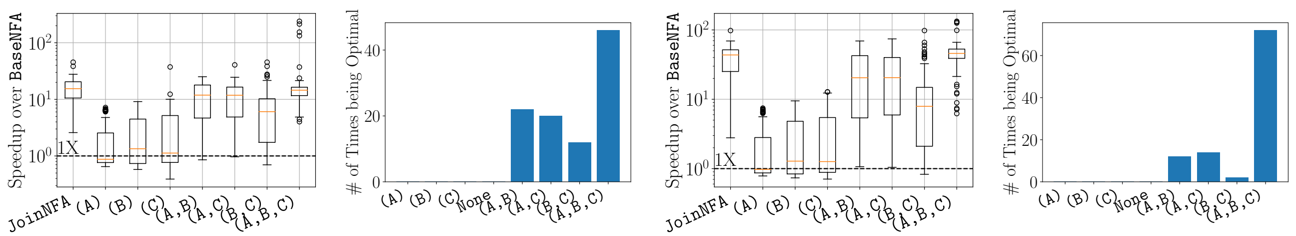
(a) Box plots of symbol sets' speedups (single thread).
(b) Histogram of optimal symbol sets (single thread).
(c) Box plots of symbol sets' speedups (multiple threads).
(d) Histogram of optimal symbol sets (multiple threads).

Figure 29: Statistics of each symbol set for `PartByBeat` on `SQLServer` + `Row`, over 100 random queries.



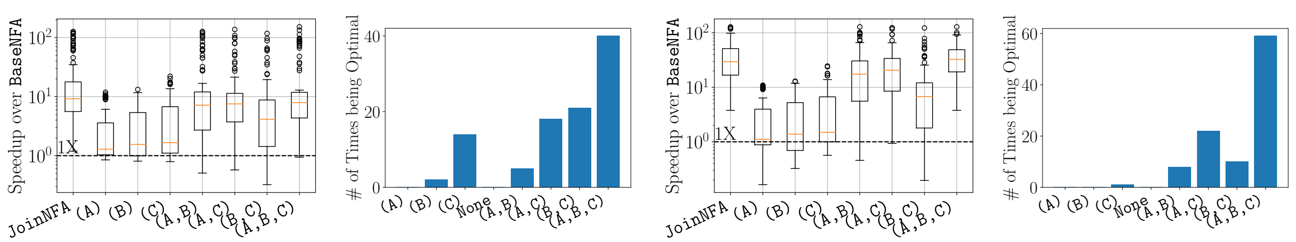
(a) Box plots of symbol sets' speedups (single thread).
(b) Histogram of optimal symbol sets (single thread).
(c) Box plots of symbol sets' speedups (multiple threads).
(d) Histogram of optimal symbol sets (multiple threads).

Figure 30: Statistics of each symbol set for `DyGeoBox` on `Trino`, over 100 random queries.



(a) Box plots of symbol sets' speedups (single thread).
(b) Histogram of optimal symbol sets (single thread).
(c) Box plots of symbol sets' speedups (multiple threads).
(d) Histogram of optimal symbol sets (multiple threads).

Figure 31: Statistics of each symbol set for `DyGeoBox` on `SQLServer` + `Col`, over 100 random queries.



(a) Box plots of symbol sets' speedups (single thread).
(b) Histogram of optimal symbol sets (single thread).
(c) Box plots of symbol sets' speedups (multiple threads).
(d) Histogram of optimal symbol sets (multiple threads).

Figure 32: Statistics of each symbol set for `DyGeoBox` on `SQLServer` + `Row`, over 100 random queries.