Democratize MATCH_RECOGNIZE!

Louisa Lambrecht
louisa.lambrecht@uni-tuebingen.de
University of Tübingen, Germany

Tim Findling*
Samuel Heid*
Marcel Knüdeler*
University of Tübingen, Germany

Torsten Grust
torsten.grust@uni-tuebingen.de
University of Tübingen, Germany

ABSTRACT

Row pattern matching in terms of the MATCH_RECOGNIZE clause is a powerful and relatively recent feature in SQL that allows users to define regular patterns over ordered rows in a table. As of today, few database systems offer support for *match recognize*, making it unaccessible to a wide range of users. We demonstrate the implementation of a transpiler that translates *match recognize* into a plain SQL query executable by any database system that supports window functions and recursive common table expressions—no changes to the underlying database systems are required.

We evaluate the performance of this approach on the running example to show that the transpiler generates code competitive with contemporary database systems that implement row pattern matching natively. The on-site demonstration is based on DuckDB.

PVLDB Reference Format:
Louisa Lambrecht, Tim Findling, Samuel Heid, Marcel Knüdeler, and Torsten Grust. Democratize MATCH_RECOGNIZE!. PVLDB, 18(12): 5251 - 5254, 2025.
doi:10.14778/3750601.3750644

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://db.cs.uni-tuebingen.de/matchrecognize.

1 DEMOCRATIZING MATCH_RECOGNIZE

MATCH_RECOGNIZE is a SQL clause that was first added to the SQL Standard in 2016 [6]. It provides an expressive tool for *row pattern matching*, *i.e.*, finding sequences of ordered rows that follow a user-defined regular pattern. Possible usage involves pattern detection for applications in security and finance or, generally, the analysis of time series data [9, 13].

Examples of database systems implementing row pattern matching include Oracle [7], Snowflake [12], Trino [2], Azure Stream Analytics [1], and Apache Flink [3]. They all provide at least a substantial subset of *match recognize*'s specification but may not follow the SQL Standard in detail. Almost ten years after the introduction of *match recognize*, the majority of database systems entirely lack an implementation, however.

This is where the present work comes in. We argue that row pattern matching definitely is useful and should be part of contemporary SQL dialects. An integration into the database kernel reaches deep and may be non-trivial. We thus demonstrate a transpiler[1] that translates MATCH_RECOGNIZE into plain SQL code:

- We compile the *match recognize*-based table expression into a pipeline of (recursive) common table expressions (CTEs). The first CTE uses the given table as input. With every step of the pipeline the data undergoes transformation until the last CTE mimics the content of the full table expression.
- The generated code purely relies on *recursive CTEs* and window functions: any DBMS that supports these established SQL constructs will thus also be able to evaluate *match recognize* queries.

The on-site demonstration offers a web interface with an input editor for *match recognize* queries. Transpilation to regular SQL is live and happens on every keystroke. Our setup relies on DuckDB [11] as the target DBMS. A DuckDB web shell can be opened to immediately execute the compiled SQL code on sample data. We chose DuckDB as it has rich support of recursive CTEs and window functions, and as of now does not implement *match recognize* natively. Trino's recursive CTE feature is experimental only.

Row Pattern Matching in SQL. Row pattern matching is a powerful SQL feature to detect patterns in ordered tables. A comprehensive explanation of the syntax can be found in [10]. For a brief sketch of the most important features, see the running example in Figure 1. The MATCH_RECOGNIZE subclauses allow the user to define ❶ *ordering* and *partitioning* of the input data, ❷ *tags* (here: A, B, and C) that are attached to those rows that fulfill the given conditions, and ❸ the *pattern* to be matched. The pattern is defined as a *regular expression over row tags*. Finally, the MEASURES subclause ❹ performs computations over the rows that constitute a match.

[1] a source-to-source-compiler with a human-readable language as a target language

```
SELECT mr.station, mr.match_no, mr.tstamp, mr.diesel,
       mr.e5, mr.tag, mr.duration, mr.diff
FROM   gas_prices MATCH_RECOGNIZE (
  PARTITION BY station                                          }❶
  ORDER BY     tstamp                                           }
  MEASURES MATCH_NUMBER()                      AS match_no,    }
           CLASSIFIER()                        AS tag,         }❹
           LAST(D.tstamp) - FIRST(D.tstamp)    AS duration,    }
           abs(AVG(C.diesel) - A.diesel)       AS diff         }
  ALL ROWS PER MATCH
  AFTER MATCH SKIP TO LAST B
  PATTERN( A (B+ C*?)+ A )                                      ❸
  SUBSET D = (B, C)
  DEFINE A AS A.diesel <= A.e5,                                 }
         B AS B.diesel > B.e5                                   }
          AND B.diesel > A.diesel AND B.e5 < A.e5,              }❷
         C AS C.diesel > C.e5                                   }
) AS mr
ORDER BY mr.station, mr.match_no, mr.tstamp;
```

Figure 1: A *match recognize* query collecting information about atypical diesel fuel prices that exceed gasoline prices.

*(Former) students at University of Tübingen, listed alphabetically.

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750644

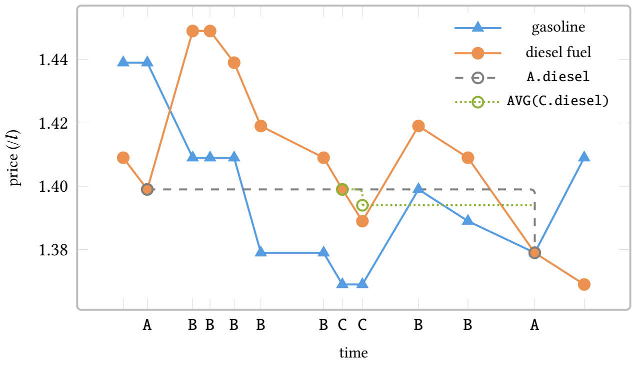


Figure 2: The time series (blue and orange) show the price development of gasoline and diesel fuel. The match starts with the second data point and ends with the penultimate one. On the x axis, the row tags are given. The dashed gray line denotes the value of `A.diesel` within this match and the dotted green line the average diesel value of `C`-tagged rows.

The example in Figure 1 demonstrates how to find segments in a time series containing atypical gasoline prices that drop below the price of diesel fuel. A match found by the regular row pattern `A (B+ C*?)+ A` is shown in Figure 2.

Some conditions (`A`, `C`) are easily determined while others (`B`) can only be evaluated during the matching process as they depend on values of previously matched rows. This ability to inspect the vicinity of rows renders *match recognize* versatile but may affect matching performance. Optimization techniques for row pattern matching are addressed in [8, 14].

Apart from the matching itself, *match recognize* can perform computations (`MEASURES`) over specific rows contained in a match. In Figure 1 (Line 8), the example query calculates the duration of the price anomaly through a subtraction of timestamps (rows tagged either with `B` or `C` constitute a match—the `SUBSET` definition in Line 13 enables us to refer both tags using the common tag `D`).

We are barely scratching the surface of `MATCH_RECOGNIZE` syntax here—a full implementation of the construct is a rather complex undertaking.

2 FROM `MATCH_RECOGNIZE` TO `WITH RECURSIVE`

Our compilation approach relies on recursive CTEs that were introduced with SQL:1999. We use a data transformation pipeline of various CTEs to emulate the behavior of *match recognize*. In the core matching process, a recursive CTE is used to drive a finite

```
WITH RECURSIVE
  automaton AS(...),
  preprocessing AS(...),
  bfs AS(...),
  matches AS(...),
  after_match AS(...),
  collect AS(...),
  postprocessing AS(...)
SELECT ...
```

Figure 3: CTE-based transpilation scheme.

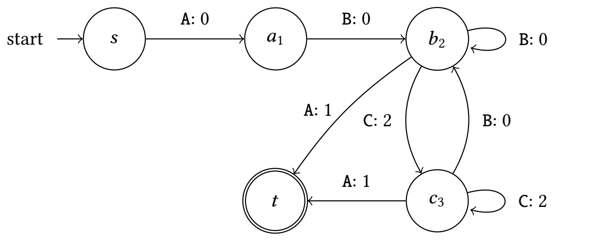


Figure 4: The finite state automaton for the pattern `A (B+ C*?)+ A`. The transitions of the automaton are the conditions that result in a row tag: `A`, `B`, and `C`. Additionally, each transition is weighted with a penalty: 0, 1, and 2.

state automaton that iteratively draws rows from the input table. Pre- and post-processing steps build on SQL window functions and ensure that matches are filtered according to the specification of the `MATCH_RECOGNIZE` subclauses. Figure 3 shows the structure of the CTE pipeline. In what follows we describe each step of the pipeline in more detail.

Automaton. A finite state automaton forms the core of the row pattern matcher. Every transition in the automaton denotes a row matching the condition that corresponds to the tag of its transition. Figure 4 shows the automaton for the pattern `A (B+ C*?)+ A` of the running example.

Since condition `B` is a stricter version of condition `C`, every row matching condition `B` also matches condition `C`. This introduces many matching possibilities, *e.g.*, states b_2 and c_3 have `B` and `C` on the outgoing edges. To ensure a deterministic result, *match recognize*'s evaluation rules prefer matching greedy over lazy quantifiers (*e.g.*, `*` dominates `*?`), and alternatives in their declared order (*e.g.*, `a` dominates `c` in `a|c`).

We developed an algorithm that constructs a non-deterministic finite state automaton (NFA) that implements this evaluation logic. We do so by annotating transitions with penalties making the NFA a weighted graph-automaton hybrid. Penalties must not repeat within all outgoing edges of one state. Thus, the first match reaching terminal state t by always taking the transition of the lowest penalty possible adheres to the matching logic. This constitutes the *depth-first search result.*

The construction algorithm was inspired by Glushkov's construction [4]. The automaton in Figure 4 shows that matching a greedy `B+` results in the lowest penalties. The penalties for lazy `C*?` are higher than simply continuing the pattern by returning to `B` or moving on to the last `A`.

Preprocessing using window functions. The first CTE applies `SELECT ROW_NUMBER() OVER (`❶`)` to the input table to make row ordering as well as partitioning explicit in columns `rid` and `station`. Table ⊞ prep below shows this data transformation for our example. Columns `rid` and `station` are essential in every subsequent step of the pipeline. Any references to rows with relative offsets (`PREV`/`NEXT`) are replaced by references to the window function results that are calculated here.

Matching—Breadth first search comes naturally with SQL. Maintaining state in a recursive CTE is cumbersome and comes along with performance issues which makes it impractical to prep

*	station	rid
…	stat1	1
…	stat1	2
…	…	…
…	stat2	1
…	…	…

bfs

*	station	rid	start	tag	path	skip	measures
…	stat1	1	1	A	[A:0]	□	…
…	stat1	2	1	B	[A:0,B:0]	2	…
…	…	…	…	…	…	…	…
…	stat2	1	1	A	[A:0]	□	…
…	…	…	…	…	…	…	…

leverage a depth-first search [5]. We thus use a breadth-first search to find all possible matches using the automaton. The search is implemented by iteratively joining the preprocessed input table with itself and the automaton. During the matching process each row belonging to a match receives its `tag`. Such late assignment of tags during the matching process allows for conditions (like `B`, see Figure 1, Line 15–16) that depend on rows that have been tagged earlier (with `A`). Since penalties appear just once for each state, the path of a match is unambiguously defined by its *sequence of penalties*.

Incomplete matches are discontinued automatically as the join condition with the automaton does not result in more rows. All *measures* of the `MATCH_RECOGNIZE` clause are computed in this step by adding at least one column per measure item. The result is a table like `bfs` above that holds all rows of full and incomplete matches with their corresponding measures, start and row index, and automaton path taken.

While this approach may result in computational overhead since we possibly compute many matches that are not part of the final result, it still results in better performance than maintaining a state for a depth-first search in SQL. We intend to further explore improved performance options by using alternatives to vanilla recursive CTEs [5].

Retrieving the depth-first matches. We collect rows that reach the final state of the NFA which returns one row per match for every *complete* match.

The breadth-first search produces *all* possible matches, *i.e.*, the search also produces different matches for the running example (see Figure 2) where `C`-tagged rows should have been tagged as `B`. For example, the match `ABBCA` has the alternative tagging solution `ABCCA`. These matches have higher penalties at the sequence position that represents that tagging decision. In order to retrieve the depth-first matches among all matches, we `arg_min` with respect to the sequence of penalties. This entails the depth-first search result `ABBCA` we desire.

Table `matches` shows the result of this pipeline step.

matches

station	start	end	skip	path
stat1	1	5	3	[A:0,B:0,B:0,C:2,A:1]
stat1	5	7	6	[A:0,B:0,A:1]
…	…	…	…	…
stat2	1	4	3	[A:0,B:0,B:0,A:1]
…	…	…	…	…

After finding a match. In this step of the pipeline, we additionally filter the matches according to the `AFTER MATCH` clause. It determines if and how much matches may overlap. There are four different *skip options*. The retrieval of the depth-first match implements `SKIP TO NEXT ROW` per default. The default skip option in row pattern matching is `SKIP PAST LAST ROW`. The other alternatives comprise jumping to a certain row tag using `SKIP TO FIRST`/`LAST` *tag*. When skipping to the first/last occurrence of a row tag, we gather the row index of said row during the matching process and carry it through the pipeline. This is displayed in the examplary tables `bfs` and `matches` with the `skip` column. When overlapping matches are not allowed, the match's last row index +1 is used instead of the `skip` value. A simple loop over all matches suffices to implement these filters.

Collecting all rows per match. The previous two CTEs produce one row per match as their output. In case the specification states `ALL ROWS PER MATCH`, the rows that belong to each match have to be retrieved from `bfs`. This pipeline step outputs all rows for every match with their measures, the values of the original input table and the meta data (*i.e.*, `start`, `rid`, `path`) needed to collect all rows per match. The examplary result table is shown in `collect`.

collect

*	station	start	rid	path	measu…
…	stat2	1	1	[A:0]	…
…	stat2	1	2	[A:0,B:0]	…
…	stat2	1	3	[A:0,B:0,B:0]	…
…	stat2	1	4	[A:0,B:0,B:0,A:1]	…
…	…	…	…	…	…

post

*	station	measu…
…	stat2	…
…	stat2	…
…	stat2	…
…	stat2	…
…	…	…

Postprocessing. The last CTE is responsible for postprocessing—usually these are simple projections. The columns holding meta data are removed such that the table mimics the output table of the `MATCH_RECOGNIZE` clause. This pipeline step also leaves room for implementing the `FINAL` semantics. The result of the output is Table `post`, containing the columns of the input table as well as the computed measures. Since this is exactly the table format defined by the *match recognize* specification, the surrounding query (Figure 1, Lines 1–2+19) can be run against the `postprocessing` CTE to process the results of row pattern matching. The overall query results will match those obtained on a system that features native support for *match recognize*.

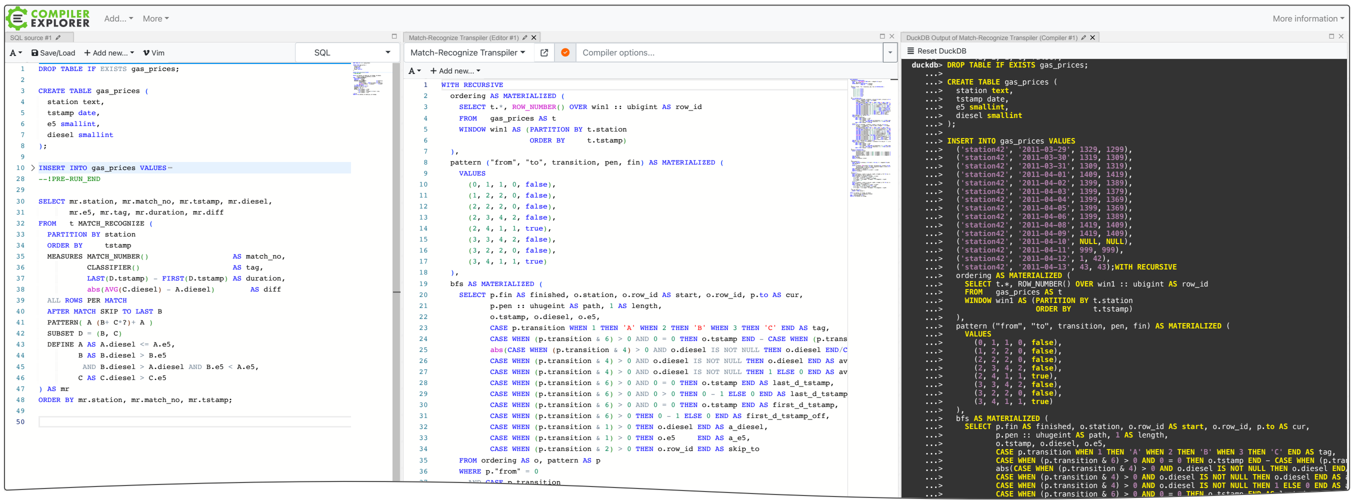
Performance. We evaluated the original query (Figure 1) on Trino (version 433) and the compiled output code on DuckDB (version 1.1.3). The data set comprises ≈71M rows of data about the price development at German gas stations in 2020[2]. The original query runs 426 seconds on Trino whereas the compiled code runs 40 seconds on DuckDB (median of five measured runs). While this is not a comprehensive evaluation, a speedup of more than ten is very promising.

Limitations. This demo focuses on the translation of row pattern matching found in Trino and Oracle to recursive CTEs in DuckDB. Adaptations to the SQL code generated by the transpiler will be required to emit SQL dialects understood by other relational backends. Certain aspects of row pattern matching have been excluded (*e.g.*, empty matches) while others are still under development (`FINAL` semantics, `MATCH_NUMBER`). The length of matches that can be found is limited by the number of bits of DuckDB's largest integer data type.

3 LIVE ON-SITE DEMONSTRATION

Implementation notes. The transpiler is written in Python and uses a parser generator (`antlr4`) to parse the *match recognize*-based input code into an abstract syntax tree (AST). We largely follow the

[2] *Tankerkönig* data set available at https://creativecommons.tankerkoenig.de/.



```sql
DROP TABLE IF EXISTS gas_prices;

CREATE TABLE gas_prices (
  station text,
  tstamp date,
  e5 smallint,
  diesel smallint
);

INSERT INTO gas_prices VALUES…
--!PRE-RUN_END

SELECT mr.station, mr.match_no, mr.tstamp, mr.diesel,
       mr.e5, mr.tag, mr.duration, mr.diff
FROM   t MATCH_RECOGNIZE (
  PARTITION BY station
  ORDER BY     tstamp
  MEASURES MATCH_NUMBER()                    AS match_no,
           CLASSIFIER()                      AS tag,
           LAST(D.tstamp) - FIRST(D.tstamp) AS duration,
           abs(AVG(C.diesel) - A.diesel)        AS diff
  ALL ROWS PER MATCH
  AFTER MATCH SKIP TO LAST B
  PATTERN( A (B+ C*?)+ A )
  SUBSET D = (B, C)
  DEFINE A AS A.diesel <= A.e5,
         B AS B.diesel > B.e5
          AND B.diesel > A.diesel AND B.e5 < A.e5,
         C AS C.diesel > C.e5
) AS mr
ORDER BY mr.station, mr.match_no, mr.tstamp;
```

```sql
WITH RECURSIVE
  ordering AS MATERIALIZED (
    SELECT t.*, ROW_NUMBER() OVER win1 :: ubigint AS row_id
    FROM   gas_prices AS t
    WINDOW win1 AS (PARTITION BY t.station
                    ORDER BY     t.tstamp)
  ),
  pattern ("from", "to", transition, pen, fin) AS MATERIALIZED (
    VALUES
      (0, 1, 1, 0, false),
      (1, 2, 2, 0, false),
      (2, 2, 2, 0, false),
      (2, 3, 4, 2, false),
      (2, 4, 1, 1, true),
      (3, 3, 4, 2, false),
      (3, 2, 2, 0, false),
      (3, 4, 1, 1, true)
  ),
  bfs AS MATERIALIZED (
    SELECT p.fin AS finished, o.station, o.row_id AS start, o.row_id, p.to AS cur,
           p.pen :: uhugeint AS path, 1 AS length,
           o.tstamp, o.diesel, o.e5,
           CASE p.transition WHEN 1 THEN 'A' WHEN 2 THEN 'B' WHEN 3 THEN 'C' END AS tag,
           CASE WHEN (p.transition & 6) > 0 AND 0 = 0 THEN o.tstamp END - CASE WHEN (p.trans
           abs(CASE WHEN (p.transition & 4) > 0 AND o.diesel IS NOT NULL THEN o.diesel END/C
           CASE WHEN (p.transition & 4) > 0 AND o.diesel IS NOT NULL THEN o.diesel END AS av
           CASE WHEN (p.transition & 4) > 0 AND o.diesel IS NOT NULL THEN 1 ELSE 0 END AS av
           CASE WHEN (p.transition & 6) > 0 AND 0 = 0 THEN o.tstamp END AS last_d_tstamp,
           CASE WHEN (p.transition & 6) > 0 AND 0 > 0 THEN 0 - 1 ELSE 0 END AS last_d_tstamp
           CASE WHEN (p.transition & 6) > 0 AND 0 = 0 THEN o.tstamp END AS first_d_tstamp,
           CASE WHEN (p.transition & 6) > 0 THEN 0 - 1 ELSE 0 END AS first_d_tstamp_off,
           CASE WHEN (p.transition & 1) > 0 THEN o.diesel END AS a_diesel,
           CASE WHEN (p.transition & 1) > 0 THEN o.e5     END AS a_e5,
           CASE WHEN (p.transition & 2) > 0 THEN o.row_id END AS skip_to
    FROM ordering AS o, pattern AS p
    WHERE p."from" = 0
      AND CASE p.transition
```

```
duckdb> DROP TABLE IF EXISTS gas_prices;
   ...>
   ...> CREATE TABLE gas_prices (
   ...>   station text,
   ...>   tstamp date,
   ...>   e5 smallint,
   ...>   diesel smallint
   ...> );
   ...>
   ...> INSERT INTO gas_prices VALUES
   ...>   ('station42', '2011-03-29', 1329, 1299),
   ...>   ('station42', '2011-03-30', 1319, 1309),
   ...>   ('station42', '2011-03-31', 1309, 1319),
   ...>   ('station42', '2011-04-01', 1409, 1419),
   ...>   ('station42', '2011-04-02', 1399, 1389),
   ...>   ('station42', '2011-04-03', 1399, 1379),
   ...>   ('station42', '2011-04-04', 1399, 1369),
   ...>   ('station42', '2011-04-05', 1399, 1369),
   ...>   ('station42', '2011-04-06', 1399, 1389),
   ...>   ('station42', '2011-04-08', 1419, 1409),
   ...>   ('station42', '2011-04-09', 1419, 1409),
   ...>   ('station42', '2011-04-10', NULL, NULL),
   ...>   ('station42', '2011-04-11', 999, 999),
   ...>   ('station42', '2011-04-12', 1, 42),
   ...>   ('station42', '2011-04-13', 43, 43);WITH RECURSIVE
   ...>   ordering AS MATERIALIZED (
   ...>     SELECT t.*, ROW_NUMBER() OVER win1 :: ubigint AS row_id
   ...>     FROM   gas_prices AS t
   ...>     WINDOW win1 AS (PARTITION BY t.station
   ...>                     ORDER BY     t.tstamp)
   ...>   ),
   ...>   pattern ("from", "to", transition, pen, fin) AS MATERIALIZED (
   ...>     VALUES
   ...>       (0, 1, 1, 0, false),
   ...>       (1, 2, 2, 0, false),
   ...>       (2, 2, 2, 0, false),
   ...>       (2, 3, 4, 2, false),
   ...>       (2, 4, 1, 1, true),
   ...>       (3, 3, 4, 2, false),
   ...>       (3, 2, 2, 0, false),
   ...>       (3, 4, 1, 1, true)
   ...>   ),
   ...>   bfs AS MATERIALIZED (
   ...>     SELECT p.fin AS finished, o.station, o.row_id AS start, o.row_id, p.to AS cur,
   ...>            p.pen :: uhugeint AS path, 1 AS length,
   ...>            o.tstamp, o.diesel, o.e5,
   ...>            CASE p.transition WHEN 1 THEN 'A' WHEN 2 THEN 'B' WHEN 3 THEN 'C' END AS tag,
   ...>            CASE WHEN (p.transition & 6) > 0 AND 0 = 0 THEN o.tstamp END - CASE WHEN (p.tra
   ...>            abs(CASE WHEN (p.transition & 4) > 0 AND o.diesel IS NOT NULL THEN o.diesel END,
   ...>            CASE WHEN (p.transition & 4) > 0 AND o.diesel IS NOT NULL THEN o.diesel END AS
   ...>            CASE WHEN (p.transition & 4) > 0 AND o.diesel IS NOT NULL THEN 1 ELSE 0 END AS
   ...>            CASE WHEN (p.transition & 6) > 0 AND 0 = 0 THEN o.tstamp END AS
```

Figure 5: The web-based interface of the *match recognize*-based *Compiler Explorer*. The input query may be edited while the output is updated live on every keystroke. A built-in DuckDB shell executes DDL statements and the transpiled SQL code.

row pattern matching syntax supported by Oracle and Trino. The extracted regular pattern is translated into the weighted NFA. The automaton's transition table is part of the generated SQL code. The transpiler converts the *match recognize* AST into the CTE pipeline sketched in Figure 3. The SQL code generation phase builds on the `pglast` library which we have tweaked to support a number of constructs specific to DuckDB's SQL dialect. Demo audience members with an interest in the implementation details—*e.g.*, the non-standard NFA construction algorithm—are invited to a guided tour through core sections of the code.

Demonstration Setup. The on-site demonstration offers a web-based interface which enables live translation of *match recognize* code into plain SQL code. This web interface is based on the *Compiler Explorer* (normally hosted at `godbolt.org`). It re-runs the transpiler in the background while the user edits their query. Figure 5 shows a screenshot of our variant of *Compiler Explorer*. The interface comprises three main components:

- An editor allows the user to author SQL code. Additionally, the user has the opportunity to fill in example data as DDL statements that the *match recognize*-based input query will be able to read. We will bring a number of interesting sample scenarios (data and associated row pattern matching queries) of various levels of complexity, but the audience is encouraged to propose their own row pattern variants or tweak the various `MATCH_RECOGNIZE` clauses.
- An output window displays the pretty-printed result of the transpilation. This SQL code is fit to be copied and pasted into a recent version of DuckDB (v1.1.1 or newer). (An additional window may be opened to visualize the weighted automaton that represents the regular pattern.)
- To enable quick turnaround, the demonstrator features its own WASM-based instance of a DuckDB shell that can execute the supplied DDL statements and generated SQL code. We will also bring a Trino instance for comparison.

ACKNOWLEDGMENTS

This research is supported by the DFG under grant no. GR 2036/6-1.

REFERENCES

[1] Rodrigo Alves. 2019. *Azure Stream Analytics now supports MATCH_RECOGNIZE.* Retrieved 2025-01-30 from https://azure.microsoft.com/en-us/blog/azure-stream-analytics-now-supports-match-recognize/
[2] Kasia Findeisen. 2021. *Row pattern recognition with MATCH_RECOGNIZE.* Retrieved 2025-03-10 from https://trino.io/blog/2021/05/19/row_pattern_matching.html
[3] Apache Flink. 2019. *Pattern Recognition.* Retrieved 2025-03-28 from https://nightlies.apache.org/flink/flink-docs-release-2.0/docs/dev/table/sql/queries/match_recognize/
[4] Victor M. Glushkov. 1961. THE ABSTRACT THEORY OF AUTOMATA. *Russian Mathematical Surveys* 16, 5 (1961).
[5] Denis Hirn and Torsten Grust. 2023. A Fix for the Fixation on Fixpoints. In *Proceedings of the 13th Conference on Innovative Data Systems Research (CIDR '23).*
[6] 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. *ISO: Global standards for trusted goods and services* 22, 2 (2016). https://www.iso.org/standard/65143.html
[7] Keith Laker. 2017. *MATCH_RECOGNIZE and predicates - everything you need to know.* Retrieved 2024-05-28 from https://blogs.oracle.com/datawarehousing/post/match_recognize-and-predicates-everything-you-need-to-know
[8] Kosuke Nakabasami, Hiroyuki Kitagawa, and Yuya Nasu. 2019. Optimization of Row Pattern Matching over Sequence Data in Spark SQL. In *Database and Expert Systems Applications*. Springer, 3–17.
[9] Dušan Petković. 2022. Identifying Possible Financial Frauds using SQL Row Pattern Recognition. *International Journal of Computer Applications* 184 (2022).
[10] Dušan Petković. 2022. Specification of Row Pattern Recognition in the SQL Standard and its Implementations. *Datenbank-Spektrum* 22, 2 (2022).
[11] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19).* 1981–1984.
[12] Snowflake. 2021. *Identifying Sequences of Rows That Match a Pattern.* Retrieved 2025-01-24 from https://docs.snowflake.com/en/user-guide/match-recognize-introduction.html
[13] Fred Zemke, Andrew Witkowski, Mitch Cherniack, and Latha Colby. 2007. *Pattern matching in sequences of rows.* Technical Report. ANSI Standard Proposal.
[14] Erkang Zhu, Silu Huang, and Surajit Chaudhuri. 2023. High-Performance Row Pattern Recognition Using Joins. *Proceedings of the VLDB Endowment* 16, 5 (2023), 1181–1195.