



Nile University

School of Information Technology and Computer Science

Program of Informatics

Row Pattern Matching Analytics: Bringing SQL MATCH_RECOGNIZE to Pandas DataFrames

A thesis submitted to the School of Information Technology and Computer Science at
Nile University in Partial Fulfillment of the Requirements for the degree of Master of
Science in Informatics.

By

Monier Ashraf Monier Lawande

Under Supervision of

Prof. Mohamed ElHelw

Prof. Ahmed Awad

Nile University

August, 2026

Thesis Supervision Agreement

Semester: Fall ☒ Winter ☐ Spring ☐ Summer ☐ Year: 2023 Date: 4/10/2023

Student Name: Monier Ashraf Monier Lawande

Student ID: 22-1002188

Program: In-Formatics

Thesis Topic: Row pattern Matching analytics on PANDAS
data Frames -

Supervision Committee: 1. prof. Mohamed E. L Helw

2. prof. Ahmed Awad

Student Signature:

Date

Monier Ashraf Monier Lawande 4 OCT 2023

Main Supervisor Signature:

Date

Mohamed A. El Helw 8 Oct '23

Supervisor Signature:

Date

Ahmed Awad 8/10/2023

Program Director Signature

Date

S. Elwan 8/10/2023

COPYRIGHT

Attention is drawn to the fact that copyright of this thesis rests with the author and copyright of any previously published materials included may rest with third parties. A copy of this thesis has been supplied on condition that anyone who consults it understands that they must not copy it or use material from it except as licensed, permitted by law, or with the consent of the author or other copyright owners.

يجب التنويه إلى أن حقوق الطبع والنشر لهذه الأطروحة تعود إلى المؤلف وأن حقوق الطبع والنشر لأي مواد منشورة مسبقاً متضمنة قد تكون مع أطراف ثالثة. تم توفير نسخة من هذه الأطروحة بشرط أن يجب على أي شخص يستشيرها عدم نسخها أو استخدام مواد منها باستثناء ما هو مرخص أو يسمح به القانون أو بموافقة المؤلف أو مالكي حقوق الطبع والنشر الآخرين.

This thesis was prepared according to:

- Egyptian Standard No. 2609-2005, which is technically identical to ISO 7144:1986 for the preparation of scientific theses.
- References follow the IEEE citation style.

تم إعداد وإخراج هذه الرسالة وفقاً ل :

- المواصفة القياسية المصرية رقم 2005-2609 المتماثلة فنياً مع المواصفة الدولية رقم ISO 7144:1986 الخاصة بإعداد الأطروحات العلمية
- تم صياغة الاستشهادات المرجعية المقتن وفقاً لأسلوب (IEEE Style)

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to Professor Ahmed Awad for his continuous academic, technical, and personal support throughout this work. His guidance, encouragement, and unwavering support at every stage meant more to me than words can express, and I am sincerely thankful for everything he has done

I am also deeply grateful to Professor Mohamed El-Helw for his valuable guidance, constant support, and understanding, particularly during difficult and unusual circumstances.

My heartfelt gratitude goes to my family—my mother, father, and brother—for their endless love, patience, encouragement, and support throughout all my years of study. Their belief in me has always been a source of strength and motivation.

I would also like to thank my friends and colleagues for their support and encouragement and kindness throughout this journey.

ABSTRACT

SQL:2016 introduced the `MATCH_RECOGNIZE` clause to search ordered rows for user-defined patterns. It is useful when the important information is a sequence rather than one isolated row. It can express V-, W-, and U-shaped patterns, repeated behavior, and changes over time. Applications include fraud detection, IoT monitoring, market analysis, and e-commerce clickstreams. Database Systems such as Oracle and Trino provide this feature, but Pandas has no native equivalent. Analysts therefore use shifted columns, loops, and custom state handling, making the code longer and harder to verify.

This thesis addresses that gap with an in-process engine for an evaluated R010-style subset: row pattern recognition in the `FROM` clause. The engine parses and validates a query, compiles its row pattern into finite automata, and executes it over ordered `DataFrame` partitions. The supported scope includes the main clauses, direct-column ordering, output and skip modes, the listed pattern constructs, navigation, and selected aggregates. The evaluation covers correctness, feature support, usability, performance, and scalability. All 709 regression tests passed. Trino and Oracle served as cross-system references; usability compared the SQL form with hand-written Pandas.

The main benchmark used five patterns and six Amazon UK Products 2023 sizes from 100 K to 2.22 million rows. All three systems returned the same normalized results in all thirty pattern–size combinations. Under one CPU core and a 32 GB memory limit, mean time across those combinations was 0.17 s for the engine, 1.04 s for Oracle, and 1.98 s for Trino. The scalability study using Amazon Reviews 2023 extended the engine to 227.9 million rows. The engine and Oracle completed all 35 cases; Trino completed 33. Engine time remained close to linear, while query-time working memory grew with

ABSTRACT

the input. The results show that row-pattern recognition can connect declarative SQL with in-process Pandas analysis while keeping sequential queries readable.

Keywords: MATCH_RECOGNIZE; Row Pattern Recognition; Pandas; SQL:2016; Finite Automata; Pattern Matching.

CONTENTS

	Page
Acknowledgments	iv
Abstract	v
List of Figures	x
List of Tables	xii
List of Abbreviations and Acronyms	xiv
Chapters:	
1. Introduction	1
1.1 Problem Statement	1
1.2 Research Objectives and Contributions	3
1.3 Organization of the Thesis	5
1.4 Publications in the Course of this Thesis	5
1.5 Chapter Summary	6
2. Background and Concepts	7
2.1 Pattern Matching in Sequential Data Analysis	7
2.1.1 Real-World Applications	7
2.2 The SQL:2016 MATCH_RECOGNIZE Clause	8
2.2.1 Clause Structure and Components	9
2.2.2 Illustrative Example: Price V-Pattern	11
2.3 Chapter Summary	12
3. Related Work	13
3.1 Existing Systems and Approaches	13
3.1.1 Comparative Overview of Existing Implementations	14
3.1.2 Oracle	14
3.1.3 Trino	14
3.1.4 Snowflake	15
3.1.5 Apache Flink	15
3.1.6 Bringing MATCH_RECOGNIZE to DuckDB	15
3.1.7 Optimization Research	16

3.1.8	Implementation Restrictions Across Systems	16
3.2	Pandas and the Python Ecosystem Gap	16
3.2.1	Research Gap in the Pandas Ecosystem	17
3.3	Contributions of This Work	17
3.4	Chapter Summary	18
4.	System Design and Methodology	19
4.1	Architecture Overview	19
4.1.1	Worked Pipeline Example	20
4.2	SQL Parsing	22
4.3	Semantic Validation	24
4.4	Logical Plan Construction	25
4.5	Chapter Summary	26
5.	Implementation and Optimization	27
5.1	Pattern Compilation with Finite Automata	27
5.2	Pandas DataFrame Execution Model	29
5.2.1	Exact Search for State-Dependent Conditions	30
5.2.2	Skip Modes, Output Modes, and Measures	30
5.3	Performance Optimization Techniques	32
5.4	Chapter Summary	35
6.	Experimental Evaluation	36
6.1	Evaluation Methodology	36
6.1.1	Evaluation Criteria and Metrics	36
6.1.2	Environment and Configuration	37
6.2	Dataset and Test Patterns	38
6.2.1	Datasets	38
6.3	Validation of Concept	40
6.3.1	Example 1: Dip Pattern on the orders Dataset	40
6.3.2	Example 2: Stock-Price Analysis on the stock_price Dataset	43
6.3.3	Example 3: V-Shape Recovery on the stock_price Dataset	47
6.3.4	Usability and Developer Effort	50
6.4	Coverage and Complexity Results	50
6.4.1	SQL Feature Coverage	50
6.4.2	Complexity Analysis	52
6.5	Empirical and Cross-System Analysis	55
6.5.1	Pattern Definitions	55
6.5.2	Cross-System Comparison Under a Common Outer Resource Limit	57
6.5.3	Throughput by Pattern, Size, and System	58
6.5.4	Memory Metrics by Pattern, Size, and System	58

6.5.5	Cross-System Summary	61
6.5.6	Sensitivity and Boundary Analysis	64
6.5.7	Scalability	67
6.5.8	Evaluation Validity	70
6.5.9	Key Findings	71
6.6	Chapter Summary	72
7.	Conclusions and Future Work	73
7.1	Discussion	73
7.2	Validation of Chapter Claims	74
7.3	Limitations	76
7.4	Conclusion	77
7.5	Future Work	77
7.6	Chapter Summary	78
	Bibliography	79
	Appendices:	
A.	Detailed Benchmark Tables	85
A.1	Measurement Provenance and Reproducibility	85
A.2	Match Counts	86
A.3	Performance Summary by Pattern and by Dataset Size	87
A.4	Detailed Cross-System Benchmark Matrices	87
A.5	Ordered vs. Shuffled Input (Sorting Scenarios)	92
A.6	Stress-Test Results	93

LIST OF FIGURES

Figure	Page
4.1 Five-stage architecture of the proposed engine.	19
4.2 Worked orders pipeline.	21
4.3 Resolving RUNNING and FINAL semantics.	26
5.1 Language-equivalent NFA for $A^+ B$?	27
5.2 Inside the DataFrame execution stage.	29
6.1 Data-size complexity of the compiled benchmark path	54
6.2 Performance summary by pattern	57
6.3 Performance summary by dataset size	58
6.4 Execution time by pattern, size, and system	59
6.5 Throughput by pattern, size, and system	59
6.6 Query memory by pattern, size, and system	60
6.7 Footprint memory by pattern, size, and system	61
6.8 Overall benchmark statistics by system	62
6.9 Average query memory by dataset size	62
6.10 Average footprint memory by dataset size	63
6.11 Proposed engine relative to each SQL engine	63
6.12 Ordered vs shuffled input by system	64
6.13 Ordered and shuffled engine time	65
6.14 Pattern-driven worst case, all systems	66
6.15 Volume scaling on real data to 227.9M rows	68

6.16	Throughput scaling on real data	68
6.17	Per-pattern time at the full 227.9M-row corpus	69
6.18	Diagnostic query-memory counters on real data	70
A.1	Per-pattern execution-time scaling	94

LIST OF TABLES

Table	Page
3.1 Scope and deployment of selected row-pattern systems	14
4.1 Principal fields in the logical execution configuration	25
5.1 The four AFTER MATCH SKIP modes and the resulting next scan position	31
5.2 Output modes and their row-count and measure-semantics behavior . .	31
5.3 Performance optimizations and their activation conditions	34
6.1 Sample orders dataset (customer_id, order_date, price)	38
6.2 Stock price table	39
6.3 Dip-pattern result	43
6.4 Stock-price result	47
6.5 V-shape result	49
6.6 Lines-of-code comparison for the three worked examples.	50
6.7 Support status of MATCH_RECOGNIZE features in the proposed engine . .	51
6.8 Pattern-type coverage	52
6.9 Theoretical and empirical views of complexity analysis	52
6.10 SQL Pattern Definitions and Detection Goals	55
7.1 Validation mapping for the thesis claims	75
A.1 Returned rows by pattern and size under ONE ROW PER MATCH	86
A.2 Performance summary by pattern across the three systems	87
A.3 Performance summary by dataset size across the three systems	87

A.4	Execution time by pattern, size, and system (seconds; mean $\pm$ sample standard deviation)	87
A.5	Throughput (rows/s) by pattern, dataset size, and system	88
A.6	Incremental query memory by pattern, size, and system (MB)	89
A.7	Absolute footprint memory by pattern, size, and system (MB)	90
A.8	Overall benchmark statistics by system	90
A.9	Average query memory by dataset size, all three systems	91
A.10	Average footprint memory by dataset size, all three systems	91
A.11	Cross-system correctness by pattern across six dataset sizes	91
A.12	Relative difference between the proposed engine and each SQL engine .	92
A.13	Average execution time for ordered and shuffled input	92
A.14	Execution time and slowdown by size for ordered (“ord”) and shuffled (“shuf”) input	92
A.15	Observed ordered–shuffled time difference for the proposed engine by dataset size	93
A.16	Mean execution time for the state-dependent aggregate pattern	93
A.17	Stress-test execution time (seconds; mean $\pm$ SD)	95
A.18	Stress-test absolute peak memory (GB)	95
A.19	Stress-test throughput (million rows/s)	96
A.20	Stress-test incremental peak memory above the pre-query baseline (GB)	96
A.21	Database-native per-query memory counters (GB)	97
A.22	Stress-test returned rows by pattern and size under ONE ROW PER MATCH	97

LIST OF ABBREVIATIONS AND ACRONYMS

Abbreviation	Full Name
ANTLR	ANother Tool for Language Recognition
API	Application Programming Interface
AST	Abstract Syntax Tree
CEP	Complex Event Processing
CPU	Central Processing Unit
CSV	Comma-Separated Values
CTE	Common Table Expression
DFA	Deterministic Finite Automaton
EE	Enterprise Edition (Oracle)
GB	Gigabyte
IQR	Interquartile Range
JIT	Just-In-Time (compilation)
JVM	Java Virtual Machine
LOC	Lines of Code
LRU	Least Recently Used
MB	Megabyte
NFA	Nondeterministic Finite Automaton
PGA	Program Global Area (Oracle)
RAM	Random Access Memory
SGA	System Global Area (Oracle)
SQL	Structured Query Language

CHAPTER 1

INTRODUCTION

Row-pattern recognition searches an ordered series of table rows for relationships that span several events. It can detect shapes such as V, W, and U, as well as trends and repeated behavior. These patterns support analysis in many domains. Examples include money-laundering transaction chains [1], failures in IoT sensor data, and user journeys through e-commerce clickstreams. Unlike an ordinary SQL filter, which evaluates one row at a time, a row pattern uses event order and relationships between rows. Its syntax resembles a regular expression applied to rows rather than characters. Analysts can therefore describe and extract meaningful sequences from ordered partitions.

1.1 PROBLEM STATEMENT

SQL:2016 standardized row-pattern recognition through the `MATCH_RECOGNIZE` clause [2], [3]. The clause lets a query define and search patterns within ordered rows. Oracle had already introduced it in Oracle Database 12c in 2013 [4]–[6]. Support nevertheless remains uneven.

Trino [7], [8], Snowflake [9], and Apache Flink [10] provide their own implementations. However, support for the optional SQL:2016 features R010, R020, and R030 varies, as do practical restrictions. PostgreSQL 18 lists these features as unsupported, while DuckDB lists `MATCH_RECOGNIZE` as planned work [11], [12].

This gap matters because many analytical workflows begin in Python. Users prepare data, test hypotheses, and build models before moving to a larger database or cloud platform. For data that already fits in memory, a separate SQL engine adds deployment, integration, and data-transfer work. This thesis does not aim to replace an enterprise database. It brings a practical subset of declarative row-pattern recognition to the Pandas environment where these workflows often begin.

Without this declarative primitive, SQL users must combine window functions, self-joins, or loops outside the database. Beyond two or three steps, the logic accumulates row-number calculations, timestamp checks, and `JOIN` conditions. It is difficult to maintain and may require repeated table scans [13].

The same limitation appears in Python. Pandas [14] is a popular tabular-data library, but it has no built-in SQL row-pattern primitive. Analysts instead create shifted columns such as `df['value'].shift(-1)` and `shift(-2)`, combine Boolean masks, use `groupby().apply()`, and then pivot or rename intermediate columns. Explicit

loops are also common. Listing 1.1 shows this approach for a three-row price decrease (A B C).

```
1 import pandas as pd
2
3 df['order_date'] = pd.to_datetime(df['order_date'])
4 df = df.sort_values(['customer_id', 'order_date']).
    reset_index(drop=True)
5
6 def match_pattern_ABC(group: pd.DataFrame):
7     matches = []
8     for i in range(len(group) - 2):
9         A, B, C = group.iloc[i], group.iloc[i+1], group.iloc[
            i+2]
10        if (B['price'] < A['price']) and (C['price'] < B['
            price']):
11            matches.append({
12                'customer_id': A['customer_id'],
13                'start_date': A['order_date'],
14                'end_date': C['order_date'],
15                'bottom_price': C['price']
16            })
17    return matches
18
19 all_matches = []
20 for _, grp in df.groupby('customer_id'):
21     all_matches.extend(match_pattern_ABC(grp))
22
23 result_df = pd.DataFrame(all_matches)
```

Listing 1.1: Detecting a three-step price-decrease pattern in Pandas without MATCH_RECOGNIZE

Even this fixed three-row pattern needs several procedural steps. The hand-written logic is vulnerable to alignment errors and does not extend cleanly to longer patterns or patterns with more conditions. Such cases also require custom indexing, overlap handling, and output logic.

The same query expressed with MATCH_RECOGNIZE is shown in Listing 1.2: a single declarative statement that specifies the partition, ordering, pattern grammar, variable definitions, and output measures.

```
1 import pandas as pd
```

```
2 from pandas_match_recognize import match_recognize
3
4 df['order_date'] = pd.to_datetime(df['order_date'])
5
6 query = """
7 SELECT customer_id, start_date, end_date, bottom_price
8 FROM orders
9 MATCH_RECOGNIZE (
10     PARTITION BY customer_id
11     ORDER BY      order_date
12     MEASURES
13         A.order_date AS start_date,
14         C.order_date AS end_date,
15         C.price       AS bottom_price
16     ONE ROW PER MATCH
17     AFTER MATCH SKIP TO NEXT ROW
18     PATTERN (A B C)
19     DEFINE
20         B AS B.price < A.price,
21         C AS C.price < B.price
22 )
23 """
24
25 output_df = match_recognize(query, df)
```

Listing 1.2: The same pattern expressed with MATCH_RECOGNIZE on a Pandas DataFrame

The declarative form is not only shorter. It separates what to find from how the engine finds it. The query is easier to audit, compose, and extend. A four- or five-step sequence with conditional alternation normally changes the PATTERN and DEFINE clauses, not the surrounding Python traversal code [15], [16]. The implementation can then compile the pattern into automata, cache repeated pattern structures, and process ordered partitions without exposing those details to the user. Chapter 2 introduces the full structure and sub-clauses of MATCH_RECOGNIZE.

1.2 RESEARCH OBJECTIVES AND CONTRIBUTIONS

This thesis brings a practical R010-style subset of SQL:2016 row-pattern recognition into Pandas. It connects relational pattern queries with in-memory analytics and asks whether they can run without a database server. Three objectives guide the work.

Research objectives.

1. **Build a Pandas-native engine.** The engine parses a MATCH_RECOGNIZE query, compiles its row pattern, and runs it directly on a DataFrame. It works alongside Pandas and needs no separate database or cloud service. It replaces procedural code such as Listing 1.1 with the declarative call in Listing 1.2.
2. **Support and verify useful row-pattern semantics.** The target is the evaluated R010-style subset. It includes the evaluated forms of the main clauses, output and skip modes, and navigation and aggregate functions. Its pattern constructs include quantifiers (*, +, ?, and {n,m}), alternation (|), grouping, anchors, exclusions ({- . . . -}), and PERMUTE. Automated and cross-system tests verify the supported behavior.

For example, a fraud analyst can express a probe-and-drain sequence as:

```
1 PATTERN (PROBE+ (TRANSFER | WITHDRAW){2,5} DRAIN)
```

3. **Evaluate the system.** The study assesses the five criteria defined in Chapter 6: correctness, SQL feature coverage, usability, performance, and scalability. It measures output agreement, developer effort, execution time, throughput, memory, and time-versus-size R^2 . The main evaluation uses Amazon UK Products 2023 [17], which contains about 2.2 million e-commerce product records. A separate scalability test uses Amazon Reviews 2023 up to 227.9 million rows [18].

Contributions. Together, these objectives produce three contributions: an in-process Pandas engine with tested row-pattern semantics, an optimized and guarded execution design, and a theoretical and empirical evaluation. Section 3.3 explains these contributions and positions them against related systems. Chapter 4 describes the engine’s five core stages.

Evidence at a glance. At the time of final validation, the regression suite contained 709 passing test cases [19]. The cases cover pattern families, constructs, output modes, navigation, and aggregation behavior. The controlled cross-system benchmark adds direct comparison with Trino 473 and Oracle 21c EE. It uses five pattern families and six nested prefixes of the Amazon UK Products dataset, from 100 K rows to the full 2,222,742 rows. All 90 system–pattern–size cells completed. For each of the 30 pattern–size combinations, all three systems returned the same normalized result set. Under the shared one-core, 32 GB memory limit, mean time was 0.17 s for the engine, 1.04 s for Oracle, and 1.98 s for Trino. Engine time grew almost linearly. Its measured query-time memory was higher than either database, but its process footprint was lower.

The larger Amazon Reviews test extends the range to 227.9 million rows, about $100\times$ the main dataset. Under a shared one-core, 58 GB memory limit, the engine and Oracle 21c Enterprise Edition completed all 35 cases. Trino 473 completed 33. Its complex nested cases at 160 and 227.9 million rows reached the 600 s limit. Engine time remained close to linear. Across the four pattern families completed by all three systems, Oracle’s per-size mean time was $5.1\text{--}10.1\times$ the engine’s. Trino’s was $11.4\text{--}13.8\times$ the engine’s. Chapter 6 reports the complete method, trade-offs, and limits.

1.3 ORGANIZATION OF THE THESIS

The rest of the thesis is organized as follows.

Chapter 2 — Background and Concepts presents the theoretical and technical foundations of the work. It introduces sequential pattern matching and its applications, explains the eight MATCH_RECOGNIZE sub-clauses, and presents the SQL:2016 R010–R030 feature classification used in this thesis.

Chapter 3 — Related Work reviews Oracle, Trino, Snowflake, Apache Flink, and DuckDB. It compares their feature scope, execution strategies, deployment models, and practical restrictions. It then examines the Python and Pandas gap and positions this work within the row-pattern literature.

Chapter 4 — System Design and Methodology describes the architecture and methodology. It covers the SQL grammar, parser, AST, semantic validation, logical plan, and the design that connects a declarative query to automata-based execution.

Chapter 5 — Implementation and Optimization explains NFA and DFA construction, Pandas execution, output construction, and measure evaluation. It also presents the performance layer: vectorized conditions, compiled fast paths, pattern caching, compact output construction, and memory management.

Chapter 6 — Experimental Evaluation presents the methodology, datasets, and results. It evaluates correctness, feature and pattern coverage, usability, pattern complexity, performance, memory, and scalability. The chapter compares five pattern families across six Amazon UK Products sizes and then presents a larger Amazon Reviews scalability test. Trino and Oracle provide the cross-system references.

Chapter 7 — Conclusions and Future Work summarizes the contributions and current limitations. It also identifies directions for future research and extension.

1.4 PUBLICATIONS IN THE COURSE OF THIS THESIS

This thesis produced two scientific outputs in data systems and sequential pattern recognition: one published paper and one submitted manuscript.

First Publication. The first paper was published in the *2026 IEEE 23rd Mediterranean Electrotechnical Conference (MELECON)*. The paper, entitled “Row Pattern Recognition for Pandas: Bringing MATCH_RECOGNIZE to Python DataFrames” [20], addresses the lack of native row-pattern recognition in Python data-analysis frameworks. It presents an SQL-in-Pandas engine for running MATCH_RECOGNIZE-style queries directly on DataFrames. The paper describes its supported row-pattern features, NFA-based matching method, and initial tests. Those tests show that the declarative interface reduces procedural Pandas code while preserving the expected results in the studied cases.

Second Publication. A second manuscript extending this work has been submitted to *Scientific Reports* and is currently under review.

1.5 CHAPTER SUMMARY

This chapter introduced row-pattern recognition and the lack of a native, declarative equivalent in Pandas. It explained why hand-written multi-row solutions become verbose, fragile, and difficult to extend. It then stated the research objectives, summarized the contributions and evidence, outlined the thesis, and listed the research outputs. Chapter 2 next introduces the theory and SQL semantics used by the system.

CHAPTER 2

BACKGROUND AND CONCEPTS

This chapter introduces the foundations needed to understand the MATCH_RECOGNIZE engine for Pandas DataFrames. Section 2.1 explains pattern matching in sequential data. Section 2.2 introduces the SQL:2016 clause and the terms used in this thesis. A V-pattern example then connects these concepts to the proposed system. The next chapter reviews related work and identifies the research gap.

2.1 PATTERN MATCHING IN SEQUENTIAL DATA ANALYSIS

Sequential pattern detection identifies ordered events or states in a dataset. A conventional filter tests each row against one or more predicates. A row-pattern query also tests the order and relationships between rows. It returns a group when the rows follow the requested sequence. This makes it possible to find shapes such as V, W, and U, as well as repeated or changing behavior. Common applications include security monitoring, stock-price analysis, sensor analysis, fraud detection, and complex event processing [4].

The important distinction is that the “symbols” in row-pattern recognition are not characters in a string. They are data rows that become associated with pattern variables such as A, B, or DOWN when their predicates hold. A row sequence can therefore be read as a sequence of variable assignments. This explains why regular-expression-like syntax and finite automata fit the problem. Later chapters use this connection to compile row patterns into automata and execute them over ordered Pandas partitions.

Relational databases are strong at set-based operations such as selection, projection, joins, and aggregation. Sequence analysis was traditionally harder. Analysts often exported data to a stream processor, used cursors and temporary tables, or wrote the logic in an application. SQL:2016 introduced row-pattern recognition so that this logic could be expressed directly in SQL [21].

2.1.1 Real-World Applications

Financial Market Analysis. Detecting a V-shaped price pattern—a decline followed by a recovery—requires an ordered sequence of price movements. Successive higher troughs, for example, can indicate an upward trend. The worked example later in this chapter applies a V-pattern to ordered price records [4].

Fraud Detection. Credit card fraud can appear as a sequence, such as several small test charges before a large purchase. Another example is a series of transactions in

locations that are too far apart to be realistic. Writing these rules in SQL can make them easier to manage than procedural code [1].

IoT and Sensor Data. Manufacturing systems monitor sensor readings for signs of equipment degradation. One example is a gradual temperature rise followed by a sudden drop. Pattern queries over ordered sensor readings are a natural use of `MATCH_RECOGNIZE` [4].

Clickstream and Funnel Analysis. E-commerce platforms study the paths that lead to a purchase or an exit. A funnel query may look for “view → add-to-cart → remove → exit.” With `MATCH_RECOGNIZE`, one query can count repeated actions, return their timestamps, and measure the elapsed time. The same analysis otherwise needs several subqueries or custom session logic [22].

Network Security and Intrusion Detection. Cyber-attacks often appear as a series of events rather than one isolated event. A multi-stage intrusion may start with port scanning, continue with failed logins, and end with a successful login and privilege escalation. One row-pattern query can describe this sequence and return the relevant session, timestamps, port, and account [1].

Healthcare and Patient Monitoring. Early-warning systems can look for a sequence of changes in clinical measurements. An alert may combine a rising heart rate, falling blood pressure, and a later laboratory change. A row-pattern query can describe this ordered sequence. The actual alerting rule depends on the clinical domain and system.

Supply Chain and Logistics. One late shipment may lead to partial deliveries and then a stockout. A row-pattern query over a fulfillment log can find this sequence for each supplier or warehouse. It can also measure the time from the first delay to the final stockout.

Financial Compliance and Trade Surveillance. Trade-surveillance systems look for sequences that may indicate market manipulation. In a layering example, a trader places a large resting order, trades on the other side, and then cancels the first order. `MATCH_RECOGNIZE` can express this timed sequence as an auditable SQL query.

2.2 THE SQL:2016 `MATCH_RECOGNIZE` CLAUSE

The SQL:2016 standard introduced `MATCH_RECOGNIZE` [2], [21]. It finds ordered rows in which each part of a pattern is defined by a row condition. The regular-expression-like pattern describes event sequences in a clear, declarative form [23]. Similar logic can be

written with joins, window functions, recursion, or procedural code. The clause makes that logic shorter and easier to maintain.

The SQL standard distinguishes three optional features for row pattern recognition [24]. **Feature R010** covers row pattern recognition in the FROM clause. **Feature R020** covers row patterns in WINDOW specifications. **Feature R030** covers full aggregate support. They are distinct parts of the standard, not successive levels [3], [15], [25].

This thesis uses the SQL:2016 terminology as the conceptual frame, but the implementation claims are intentionally scoped. The proposed engine targets core R010-style behavior in the FROM clause and evaluates selected row-pattern constructs that are relevant to in-memory Pandas analytics. Features outside this scope, including full window-clause row-pattern recognition and complete vendor-specific extensions, are treated as related work or future work rather than as requirements of the implemented system.

2.2.1 Clause Structure and Components

The clause works with a table or a derived table. It is commonly described through the eight specifications shown in Listing 2.1. PATTERN is required. In the evaluated engine, DEFINE and the other specifications are optional. Together, they control partitioning, order, output, row classification, and overlap between matches.

```

1 SELECT <columns>
2 FROM   <table>
3 MATCH_RECOGNIZE (
4     [PARTITION BY <partition_columns>] -- 1. Data
        organization
5     [ORDER BY      <ordering_columns>] -- 2. Row sequencing
6     [MEASURES      <expressions>]      -- 3. Output
        computations
7     [<rows_per_match>]                  -- 4. Result
        granularity
8     [AFTER MATCH SKIP <option>]         -- 5. Overlap handling
9     PATTERN ( <pattern_regex> )         -- 6. Sequence
        specification
10    [SUBSET <union_variable_defs>]      -- 7. Union variables
11    [DEFINE <variable_definitions>]    -- 8. Variable
        predicates
12 ) [AS <alias>]
```

Listing 2.1: General clause structure used in this thesis

1. **PARTITION BY.** If present, this specification splits the input into independent partitions, similar to a window function. Matching then runs within each customer, sensor, or other partition. Independent partitions may also support parallel processing. If it is omitted, the input forms one partition.
2. **ORDER BY.** This determines the row order within each partition. The matcher processes rows in that sequence. An explicit and deterministic order is normally needed for repeatable results. Several sort keys may be combined, each with its own ASC or DESC direction. This describes the standard syntax. The evaluated R010 execution path in this thesis is narrower and uses ascending pattern order.
3. **MEASURES.** Optionally defines computed expressions that extract values from matched row sequences using pattern variables and navigation functions (FIRST, LAST, PREV, NEXT). Each measure expression becomes a column in the output, allowing a match to be summarized without returning every raw input row.
4. **Output options.**
 - ONE ROW PER MATCH: one summary row per complete match.
 - ALL ROWS PER MATCH: one output row for each included row in a match.
 - SHOW EMPTY MATCHES: include an output row when a pattern completes without consuming an input row.
 - OMIT EMPTY MATCHES: suppress such empty-match rows.
 - WITH UNMATCHED ROWS: an ALL ROWS PER MATCH modifier that includes non-matching rows in the result where supported.
5. **AFTER MATCH SKIP.** Controls how the engine resumes after finding a match:
 - SKIP TO NEXT ROW – resume from the row after match start (maximum overlaps).
 - SKIP PAST LAST ROW – resume from the row after match end (non-overlapping matches).
 - SKIP TO FIRST *var* – resume from the first row classified as *var*.
 - SKIP TO LAST *var* – resume from the last row classified as *var*.
6. **PATTERN.** Specifies the sequence using regular-expression-like syntax with pattern variables as terminals:
 - Concatenation: A B C (A followed by B followed by C).

- Alternation: $A \mid B$ (either A or B).
- Quantifiers: A^* (zero or more), A^+ (one or more), $A?$ (zero or one), $A\{n\}$ (exactly n), $A\{n,m\}$ (between n and m). A trailing question mark gives the reluctant form, such as $A^*?$, $A^+?$, $A??$, or $A\{n,m\}?$.
- Grouping: $(A \ B)^+$ (one or more AB pairs).
- Anchors: $\text{\textasciitilde}A \ B \ C\text{\textasciitilde}$ (pattern must span the entire partition).
- Exclusions: $\{- \ A^+ \ B^+ \ -\}$ (rows matching the enclosed sub-pattern are excluded from row-level output).
- Permutations: $\text{PERMUTE}(A, \ B, \ C)$ (any ordering of A , B , and C).

7. **SUBSET.** This optional specification defines a union variable from variables already used in **PATTERN**. It does not change the match. Instead, it gives one name to rows classified by any of its component variables. It can then be referenced in **MEASURES**, **DEFINE**, and supported skip targets.

8. **DEFINE.** This optional specification assigns Boolean predicates to pattern variables. The matcher tests a candidate row in the current match context to decide whether it can be classified as a variable. Predicates may use current-row columns, navigation functions, aggregations over matched rows, and Boolean or comparison operators [24]. A variable without a **DEFINE** predicate uses the implicit true predicate.

The clause forms a table expression inside a **SELECT** query. An outer query can therefore filter, group, aggregate, or order its result where the SQL dialect allows [24].

These specifications separate the sequence to detect from the way matches are classified and returned. **PATTERN** defines the sequence, while **DEFINE** gives the row-classification rules. The output and skip options decide how matches are shown and where the next search begins. This separation guides the system design in later chapters: parsing creates a structured query, validation checks its variables and columns, and compilation builds an automaton for execution.

2.2.2 Illustrative Example: Price V-Pattern

Listing 2.2 shows a **MATCH_RECOGNIZE** query that detects a V-shaped sequence in customer order prices.

```
1 SELECT customer_id ,
2         start_price , bottom_price , final_price ,
3         start_date ,  final_date
```

```
4 FROM orders
5 MATCH_RECOGNIZE (
6     PARTITION BY customer_id
7     ORDER BY order_date
8     MEASURES
9         START.price AS start_price,
10        LAST(DOWN.price) AS bottom_price,
11        LAST(UP.price) AS final_price,
12        START.order_date AS start_date,
13        LAST(UP.order_date) AS final_date
14    ONE ROW PER MATCH
15    AFTER MATCH SKIP PAST LAST ROW
16    PATTERN (START DOWN+ UP+)
17    DEFINE
18        DOWN AS price < PREV(price),
19        UP AS price > PREV(price)
20 );
```

Listing 2.2: Detecting V-shaped price patterns with MATCH_RECOGNIZE

The query groups rows by `customer_id` and orders each group by `order_date`. A match starts with any row. It then has one or more price decreases followed by one or more price increases. For each match, MEASURES returns the starting value and date, the value at the last DOWN row, and the final UP value and date.

2.3 CHAPTER SUMMARY

This chapter explained sequential pattern matching and why ordinary row-by-row predicates are difficult to use for ordered events. It covered the main MATCH_RECOGNIZE specifications, including partitioning, ordering, measures, output modes, skip modes, patterns, subsets, and variable predicates.

The V-pattern example showed how these parts fit together in one query and why the abstraction works with automata-based execution. Chapter 3 next compares existing implementations, their deployment models, and their practical limits.

CHAPTER 3

RELATED WORK

This chapter positions the thesis within existing work on SQL row-pattern recognition and Python-based tabular analytics. Section 3.1 reviews systems that provide native `MATCH_RECOGNIZE` or related row-pattern support. Section 3.2 discusses the gap in Pandas and similar DataFrame libraries. Section 3.3 summarizes how this thesis addresses that gap.

Row pattern recognition was first proposed as a SQL standard feature by Zemke et al. [26] and was officially adopted in SQL:2016 [2], [21]. Later standards guidance and technical literature documented its semantics in more detail [3], [15], [25]. Further work studied practical implementations and applications [1], [16].

The `MATCH_RECOGNIZE` clause is supported natively by several SQL and stream-processing engines, including Oracle [4]–[6], [27], Snowflake [9], and Trino [7], [8]; Apache Flink also exposes row pattern recognition through Flink SQL [10]. By contrast, PostgreSQL 18 still lists the related SQL features as unsupported, while DuckDB lists native `MATCH_RECOGNIZE` as planned work [11], [12], [28], [29]. In systems without native support, users typically emulate row-pattern matching using window functions, recursive queries, user-defined aggregates, procedural layers, or query translation [13], [30]–[32]. These alternatives can express many of the same ideas, but they are usually more verbose and harder to maintain than a declarative row pattern clause.

3.1 EXISTING SYSTEMS AND APPROACHES

Oracle [4]–[6], Trino [7], [8], Snowflake [9], and Flink [10] provide SQL-based row-pattern recognition. Their scope and execution models differ. Oracle and Snowflake target database or warehouse workloads. Trino is a distributed query engine, while Flink targets streams. This variety shows the value of the abstraction, but it does not give Pandas users an in-process implementation.

The systems reviewed here are representative rather than exhaustive. They were selected because they provide semantic references, performance baselines, streaming alternatives, or in-process designs that are relevant to this thesis. Other platforms also provide the clause. BigQuery documents `MATCH_RECOGNIZE`, and Databricks introduced beta support in Runtime 19.0 [33], [34].

3.1.1 Comparative Overview of Existing Implementations

The SQL standard defines three optional row-pattern features—R010 (FROM-clause patterns), R020 (window patterns), and R030 (full aggregate support)—introduced in Section 2.2. In practice, vendor documentation normally describes the supported syntax instead of declaring conformance by feature code. Table 3.1 therefore describes the documented scope directly. It does not claim that all systems provide identical SQL semantics.

Table 3.1: Documented scope and deployment of selected row-pattern systems and approaches

System	Documented scope	Deployment	License
Oracle	FROM-clause row patterns	On-premises / cloud	Proprietary
Trino	FROM and window row patterns	Server / cluster	Apache 2.0
Snowflake	FROM-clause row patterns	Cloud service	Proprietary
Apache Flink	Streaming SQL subset backed by CEP	Streaming cluster	Apache 2.0
DuckDB	No native clause; external research transpiler	In-process	MIT
This work	Evaluated R010-style subset	In-process Python	MIT

3.1.2 Oracle

Oracle introduced SQL row-pattern recognition in Database 12c. Its implementation is a main reference for relational MATCH_RECOGNIZE semantics [4]–[6]. The Oracle 21c documentation considered in this thesis places MATCH_RECOGNIZE in a table expression; it does not document the R020 window form. It also prohibits some nesting and correlated-subquery forms in MEASURES and DEFINE. Its documented aggregate support is limited to COUNT, SUM, AVG, MAX, and MIN, without DISTINCT [5]. Oracle uses a proprietary license and requires a separate database deployment. This differs from an in-process, open-source package.

3.1.3 Trino

Trino is an open-source distributed query engine designed for parallel, federated query execution across large data sources [7], [8]. Its current documentation covers MATCH_RECOGNIZE in the FROM clause and also documents row pattern recognition in window structures. This gives Trino broad documented coverage among the reviewed open-source systems. It is still a query server rather than an in-process DataFrame library.

In this thesis, Trino 473 is a semantic reference for selected results and a performance baseline.

3.1.4 Snowflake

Snowflake is a managed cloud data platform. It documents `MATCH_RECOGNIZE` support, with rich `DEFINE` and `MEASURES` expressions for row-pattern queries [9], [22]. Its documentation also notes implementation-specific behavior. The engine uses backtracking, and some pattern and data combinations can take a long time to execute. Some row-pattern functions are also restricted to particular clauses. Snowflake therefore recommends selective conditions and bounded quantifiers when possible. Because it is a managed cloud warehouse, computational cost also depends on warehouse size and query workload [9].

3.1.5 Apache Flink

Apache Flink is designed for real-time stream processing. It exposes `MATCH_RECOGNIZE` through Flink SQL, backed by its Complex Event Processing (CEP) runtime library [10], [35], [36]. Its documented implementation is a SQL subset oriented toward streaming use cases. The current Flink documentation lists unsupported features including pattern groups, subsequence alternation, `PERMUTE`, anchors, exclusions, `ALL ROWS PER MATCH`, `SUBSET`, physical offsets (`PREV/NEXT`), and `DISTINCT` aggregations [10]. Flink fits continuous event streams, but it assumes a streaming runtime rather than a lightweight, batch-oriented `DataFrame` workflow.

When pattern output is later used in Python-based machine learning, server and cluster systems may require additional integration between SQL execution and modeling tools [37], [38]. In many practical pipelines, this can introduce data movement, duplicated intermediate data, or more complicated orchestration steps [39].

3.1.6 Bringing `MATCH_RECOGNIZE` to DuckDB

DuckDB is an in-process analytical database [40]. It needs no separate server, and its Python API can query a Pandas `DataFrame` directly from SQL [41].

At the time of review, `MATCH_RECOGNIZE` has not yet been implemented in DuckDB itself; its roadmap lists the clause as planned work [12]. Lambrecht et al. instead translate row-pattern queries into SQL that DuckDB already supports. Their transpiler rewrites a query into non-recursive and recursive common table expressions and then executes the translated SQL in DuckDB [29], [32], [42]. This approach expresses pattern variables,

ordering, skip behavior, and measures without adding a native operator to DuckDB. It still requires translation and execution through a second engine.

3.1.7 Optimization Research

Zhu et al. study how row-pattern queries over historical relational data can be optimized [43]. Their approach finds pattern variables whose conditions can be checked independently. It uses relational joins to remove rows that cannot take part in a match, then runs the NFA matcher on the smaller input. Cost estimates for the joins and the NFA guide this choice.

This thesis addresses a different execution environment. It compiles conditions and match plans for in-memory arrays, then uses the general execution path when an optimization does not apply. Both approaches have the same broad goal: reduce unnecessary work before or during the automaton scan without changing the query result.

3.1.8 Implementation Restrictions Across Systems

The comparison shows three recurring differences [5], [7], [9], [10], [16]. First, some systems support only the FROM-clause form, while others also support window patterns. Second, function and aggregate coverage differs. Third, empty matches, exclusions, reluctant quantifiers, and skip modes are not uniform across implementations.

Oracle, Trino, Snowflake, and Flink also require a database, cluster, stream runtime, or cloud service. That setup is reasonable for workloads already in those environments, but it adds friction when the data already fits in memory [44]–[46].

3.2 PANDAS AND THE PYTHON ECOSYSTEM GAP

Pandas is a popular open-source Python library for data analysis [14]. Easy installation, broad adoption, and tight integration with the Python data-science stack make it a common choice for tabular analysis without a separate database. It provides primitives for sorting, filtering, grouping, rolling windows, shifting columns, and time-series analysis.

But those are low-level building blocks, not a declarative row-pattern language. Expressing a multi-step sequence usually means stitching together groupby, shift, rolling, custom loops, or intermediate Boolean columns. The code becomes verbose and error-prone as patterns lengthen, need overlap control, or refer to already-matched rows.

The Pandas 2.3.3 API used in this thesis has no native SQL:2016 MATCH_RECOGNIZE-style primitive [47]. Equivalent logic must therefore be assembled from operations such

as groupby, shift, rolling, and custom iteration, or sent to another engine. Other DataFrame libraries, including Dask, Polars, and Modin, are outside the Pandas-specific implementation and evaluation scope of this thesis [45].

3.2.1 Research Gap in the Pandas Ecosystem

The reviewed systems expose a deployment and workflow gap. Each fits when the data already lives in its environment or needs distributed or stream processing. None is the direct execution path for a Pandas DataFrame already held in a notebook or script.

DuckDB reduces the setup cost because it runs in-process. The reviewed row-pattern approach still translates the query and executes it inside DuckDB. The gap addressed here is more specific: a Pandas-native interface that keeps the declarative SQL structure without a second engine or a transpilation step.

That sets the scope of this work. The engine targets in-process, offline DataFrame analytics and an evaluated R010-style subset with selected SQL:2016 constructs. It is not intended to replace a distributed database or stream processor.

3.3 CONTRIBUTIONS OF THIS WORK

The three objectives in Section 1.2 lead to three connected contributions: the engine and its tested semantics, its optimized execution design, and the evaluation.

1. A **Pandas-native engine** for the evaluated SQL:2016 R010-style subset. It supports the evaluated forms of the main clauses, output modes, skip modes, navigation functions, and aggregates. Its pattern constructs include quantifiers, alternation, grouping, anchors, exclusions, and PERMUTE. It runs in-process and needs no database server or cloud warehouse. Chapter 4 presents its five-stage design.
2. An **optimized and guarded execution design** with compiled fast paths, vectorized predicate evaluation, compact output construction, bounded caching, and partition-scoped data handling. Conditions outside a fast path use the general evaluator.
3. **Theoretical and empirical evaluation.** The theoretical analysis gives time and space bounds for ordering, matching, output construction, and pattern complexity. The empirical analysis evaluates correctness, SQL feature coverage, usability, performance, and scalability. Performance includes execution time, throughput, and memory. The study uses synthetic reference cases, Amazon UK Products 2023 up to 2.22 million rows, and Amazon Reviews 2023 up to 227.9 million rows [17], [18]. Chapter 6 gives the full method and results.

3.4 CHAPTER SUMMARY

Oracle, Trino, Snowflake, and Flink show how row patterns work in database and streaming environments. DuckDB research shows a translation-based, in-process alternative. Optimization research shows how early filtering can reduce matching work.

Pandas 2.3.3 still has no direct row-pattern interface. Users must write procedural DataFrame logic or call another engine. This leaves the focused gap addressed by the thesis: declarative row-pattern recognition inside an in-memory Pandas workflow.

CHAPTER 4

SYSTEM DESIGN AND METHODOLOGY

This chapter describes how the engine turns a MATCH_RECOGNIZE query into an executable plan for a pandas DataFrame [48]. It also explains the main design decisions. The focus is the front end: the stages that turn raw SQL into a validated execution configuration for the implemented row-pattern subset [2]. Chapter 5 continues with automata compilation, execution, and optimization.

4.1 ARCHITECTURE OVERVIEW

The engine takes two inputs: an SQL query and a pandas DataFrame. It returns a result DataFrame. The work has five stages (Figure 4.1). *Parsing* creates an abstract syntax tree (AST). *Semantic validation* checks references and supported clause combinations while the tree is built. *Logical-plan construction* resolves the execution configuration. *Pattern compilation* builds the automata. Finally, *execution* combines the compiled pattern with the input and builds the result.

These stages are design boundaries. They do not require five separate runtime objects: parsing and validation share one traversal, and the logical plan is a group of resolved attributes. The separation still keeps errors close to their source and lets each boundary be tested before matching begins.

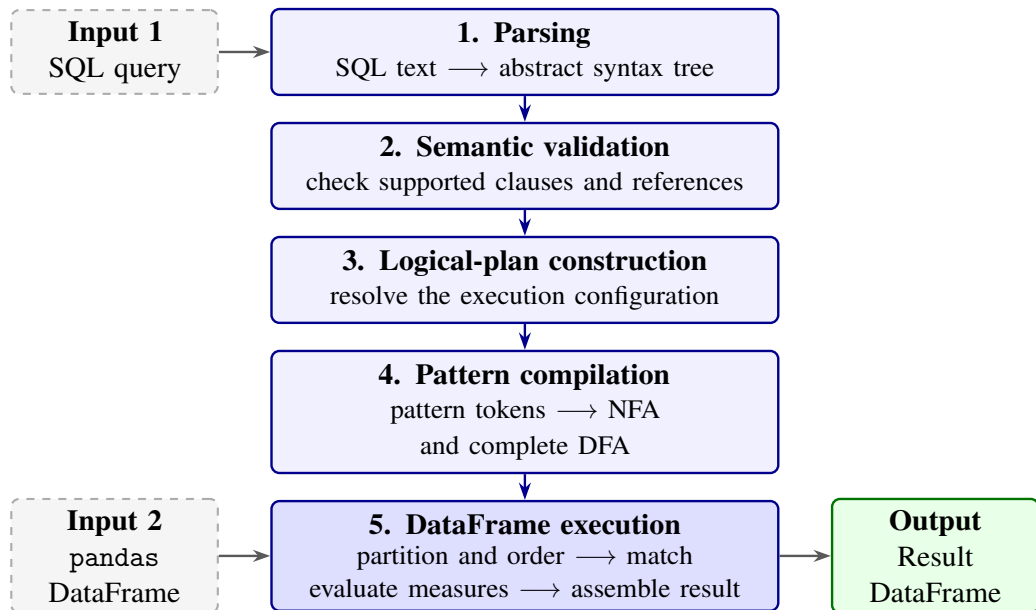


Figure 4.1: Five-stage architecture of the proposed engine.

The compiler converts the validated pattern into a non-deterministic finite automaton (NFA) and then a complete deterministic finite automaton (DFA) [36]. If complete construction exceeds a resource limit, compilation fails explicitly; it never executes a partial automaton. Chapter 5 explains compilation and DataFrame execution in detail. After the worked example, the remaining sections focus on the first three stages.

4.1.1 Worked Pipeline Example

Listing 6.2 applies the architecture to the `orders` example. Its two inputs are the SQL query and the eight-row DataFrame in that listing. Figure 4.2 traces both inputs through the architecture in Figure 4.1.

The input table accounts for all eight rows. Stage 4 shows the four reachable DFA states produced for this pattern. Each transition is guarded by the predicate of its labeled variable. During execution, `cust_1` satisfies $A \rightarrow B \rightarrow C$ with prices $200 > 100 > 50$ and produces the output row. The `cust_2` candidate fails at C because $6 < 4$ is false.

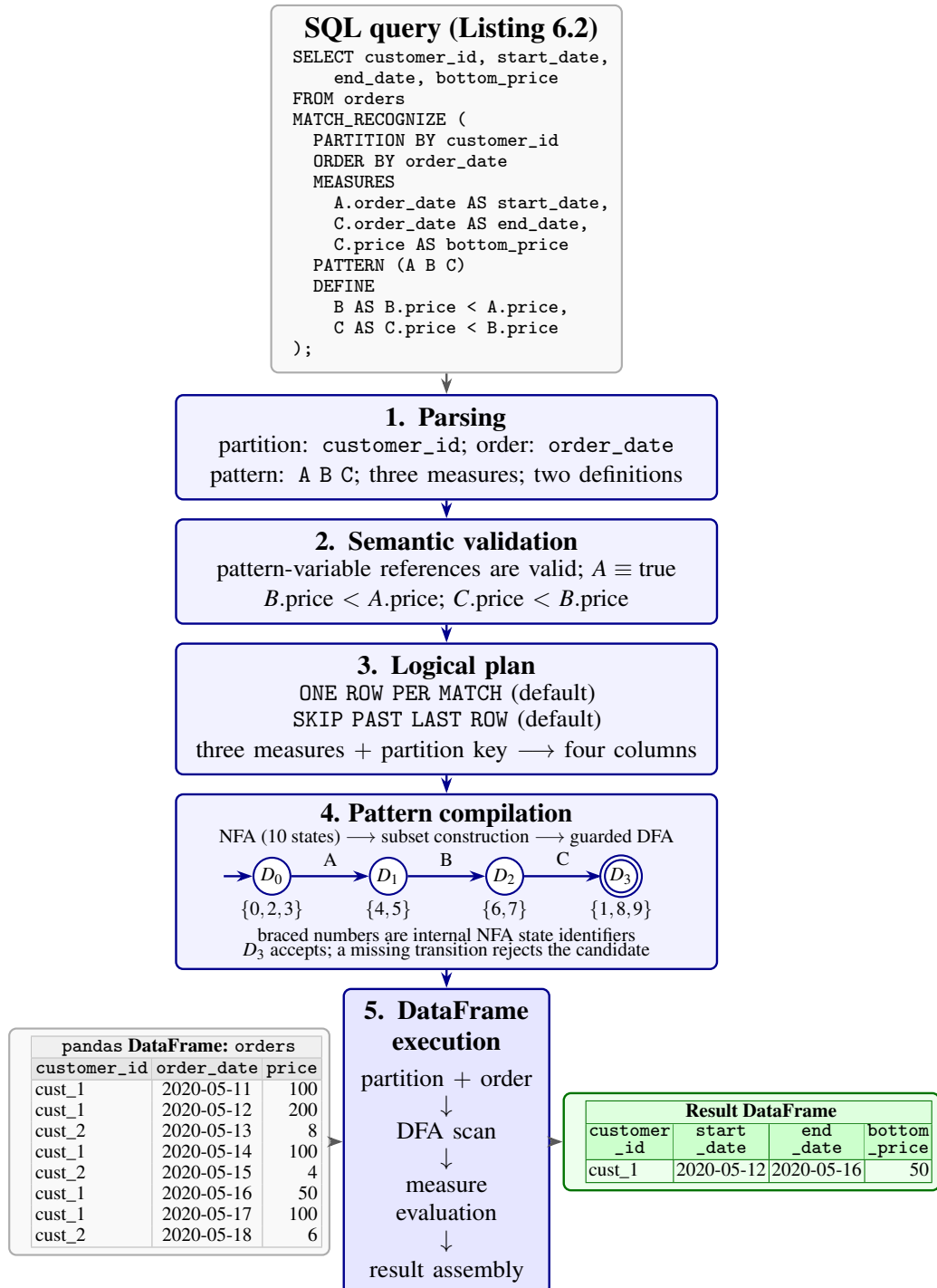


Figure 4.2: Worked orders pipeline.

4.2 SQL PARSING

A `MATCH_RECOGNIZE` query mixes ordinary SQL with an extended row-pattern language. It has eight sub-clauses and, inside `PATTERN`, its own grammar of quantifiers, alternation, grouping, anchors, exclusions, and `PERMUTE`. The parser must understand both. Listing 4.1 summarizes the clause structure accepted by the core path.

```

1 SELECT <expression> [, ...]
2 FROM   <table>
3 MATCH_RECOGNIZE (
4   [ PARTITION BY <column> [, ...] ]
5   [ ORDER BY <column> [ASC|DESC]
6     [NULLS FIRST|NULLS LAST] [, ...] ]
7   [ MEASURES <expression> AS <alias> [, ...] ]
8   [ ONE ROW PER MATCH
9     | ALL ROWS PER MATCH
10    [ SHOW EMPTY MATCHES
11      | OMIT EMPTY MATCHES
12      | WITH UNMATCHED ROWS ] ]
13   [ AFTER MATCH SKIP
14     { PAST LAST ROW
15       | TO NEXT ROW
16       | TO FIRST <variable>
17       | TO LAST  <variable> } ]
18   PATTERN ( <row_pattern> )
19   [ SUBSET <subset_name> = ( <variable> [, ...] ) [, ...] ]
20   [ DEFINE <variable> AS <condition> [, ...] ]
21 ) [ AS <alias> ]
22 <rest>;

```

Listing 4.1: Clause structure used by the supported `MATCH_RECOGNIZE` path

The core `DataFrame` path targets one `MATCH_RECOGNIZE` clause inside a top-level `SELECT`, with an optional outer `ORDER BY`. A narrow separate path supports multiple independent clauses over self-contained inline relations. It is not a general relational planner. The `DEFINE` clause is optional, and a pattern variable without a predicate is treated as always true. Direct ordering keys support per-key `ASC/DESC` direction and `NULLS FIRST/NULLS LAST`; omitted null placement defaults to `NULLS LAST`. General ordering expressions, full CTE rewriting, set-operation planning, and general nested subqueries remain outside the supported path.

Choosing a parser. The requirement is a typed tree for both languages at once. At the time of design, none of the reviewed Python SQL libraries covered the needed row-pattern syntax. `sqlparse` [49] produces only a shallow token stream. `sqlglot` [50] and `mo_sql_parsing` [51] expose a typed AST but omit constructs such as exclusions, anchors, `PERMUTE`, and bounded quantifiers. `pglast` [52] cannot parse `MATCH_RECOGNIZE` at all. Rather than extend one of these, the engine reuses the community ANTLR4 grammar for the Trino SQL dialect [53], [54]. Its `patternRecognition` rule already covers all eight sub-clauses. ANTLR4 generates the lexer, parser, and traversal scaffolding directly from that grammar, so no grammar changes were needed.

Two ANTLR4 properties matter here. Its adaptive $LL(*)$ prediction examines more input when a decision needs it [55]. It also rewrites direct left-recursive expression rules internally. This lets the Trino grammar keep its natural Boolean and arithmetic structure.

Syntax errors go through a custom *error listener* instead of ANTLR’s console default. Each error carries a line, column, and source snippet. A hand-written recursive-descent parser would duplicate a large SQL grammar. Regular expressions also cannot model nested pattern groups.

From parse tree to a typed AST. ANTLR4’s raw parse tree is a *concrete syntax tree*. It keeps every matched token, including keywords and punctuation that execution does not need. Lowering it to an abstract form follows the *Visitor* pattern [56]. ANTLR4 emits the visitor base class. Two cooperating visitors handle the outer query and the `MATCH_RECOGNIZE` clause. They produce a compact, typed AST. Its root holds the outer `SELECT/FROM` and one `MatchRecognizeClause`. Each sub-clause has a typed dataclass for its resolved fields. Later stages can dispatch on node type instead of reading the raw SQL again. Expression leaves remain textual by design. `DEFINE` predicates and `MEASURES` expressions are extracted as strings and parsed later by the restricted evaluator. This keeps the outer query AST compact.

Safe expression evaluation. Those strings come from the user, so the engine never runs them as Python code. It reads each one into a small tree of operations and walks that tree itself. The walk uses an *allowlist*: an operation is evaluated only when the engine has a handler for it. Safety therefore comes from the evaluator design, not from blocking unsafe cases one at a time.

The allowlist includes comparisons, Boolean and arithmetic operators, literals, `IN` lists, conditional expressions, qualified references such as `A.price`, navigation functions, `CLASSIFIER`, `MATCH_NUMBER`, and the supported aggregates. Imports, assignments, and access to Python built-ins are not evaluated. An expression sees only its explicit evaluation context: ordered rows, pattern bindings, and match metadata.

The front end is exercised by dedicated tests. They include adversarial inputs—undefined variables, prohibited function forms, invalid skip targets, and incompatible output-mode combinations—and end-to-end scenarios adapted from Trino’s row-pattern tests. General subquery planning and SQL features outside the supported row-pattern subset are not handled by this path. Vendor extensions may also need parser and evaluator changes.

4.3 SEMANTIC VALIDATION

Parsing proves that a query is well formed, but not that it is semantically valid. For example, `PATTERN (A B)` with `DEFINE B AS B.price > Q.price` parses correctly, but `Q` is not declared by the pattern.

A validation pass runs while the AST is completed and before any automaton is built. It rejects invalid references and unsupported clause combinations. The next stage then supplies defaults and builds the execution configuration. At this point no input row has been matched, so errors can still identify the clause that caused them.

Every reference must resolve. This is ordinary *name resolution*, with the pattern as its scope. For ordinary patterns, the front end checks qualified references in `DEFINE` and `MEASURES` against the pattern variables and `SUBSET` aliases. Specialized `PERMUTE` validation continues during pattern handling. A subset such as `SUBSET U = (A, B)` lets `U.price` refer to rows matched by either member. A variable without `DEFINE` is valid because its predicate is implicitly true. `DataFrame` column names are resolved later, when the execution configuration is applied to the input.

Calls must use a supported form. The front end checks the syntax of navigation functions and selected aggregates. Detailed aggregate argument checks also run before an aggregate is evaluated. For example, `COUNT_IF` takes one condition, while `SUM_IF` and `AVG_IF` take a value and a condition. This validation does not try to prove that a predicate can become true; a valid predicate may simply produce no match.

Settings must work together. Some combinations ask for contradictory output. An exclusion, written `{- ... -}`, keeps matched rows out of the result, while the `WITH UNMATCHED ROWS` form of `ALL ROWS PER MATCH` asks for every row to appear. One removes what the other requires, so the standard forbids the pair. The validator therefore rejects it. An `AFTER MATCH SKIP` target must also name a pattern variable or subset. Whether the target occurs in a particular match is known only at runtime. The matcher therefore rejects an absent target or a skip position that would make no progress. The output mode, skip mode, implicit predicates, and measure modes then pass to logical-plan construction (Section 4.4).

4.4 LOGICAL PLAN CONSTRUCTION

The logical plan is a resolved execution configuration. It collects the typed AST fields pulled from the `MatchRecognizeClause` that compilation and matching need (Table 4.1). Later stages do not reparse the full SQL query. Pattern, predicate, and measure text remain available to their dedicated compilers and evaluators.

Table 4.1: Principal fields in the logical execution configuration

Attribute	Type	Semantics
<code>partition_by</code>	<code>List[str]</code>	Columns for DataFrame partitioning; empty means a single global partition
<code>order_by</code>	<code>List[str]</code>	Direct columns used for intra-partition ordering
<code>order_ascending</code>	<code>List[bool]</code>	Per-key ascending or descending direction
<code>order_nulls</code>	<code>List[str]</code>	Per-key null placement; omitted placement defaults to <code>NULLS LAST</code>
<code>pattern_text</code>	<code>str</code>	Normalized pattern string forwarded to the tokenizer
<code>define</code>	<code>Dict[str, str]</code>	Explicit variable $\rightarrow$ predicate mappings; absent mappings are treated as implicit true during matching
<code>subset_dict</code>	<code>Dict[str, List[str]]</code>	Subset alias $\rightarrow$ constituent variables
<code>measures</code>	<code>Dict[str, str]</code>	Measure alias $\rightarrow$ expression string
<code>measure_semantics</code>	<code>Dict[str, str]</code>	Measure alias $\rightarrow$ "RUNNING"/"FINAL", per the rules in Figure 4.3
<code>rows_per_match</code>	<code>RowsPerMatch</code>	Output-row multiplicity
<code>skip_mode</code>	<code>SkipMode</code>	After-match skip strategy
<code>skip_var</code>	<code>Optional[str]</code>	Target variable/alias for <code>TO FIRST/TO LAST</code>

Two defaults matter. A variable without `DEFINE` uses an implicit true predicate. Measure semantics follow the priority rules in Figure 4.3. An explicit `RUNNING` or `FINAL` prefix wins. `ONE ROW PER MATCH` uses final values. Under `ALL ROWS PER MATCH`, navigation rules are resolved before aggregate defaults. `RUNNING` sees rows through the current output row; `FINAL` sees the complete match.

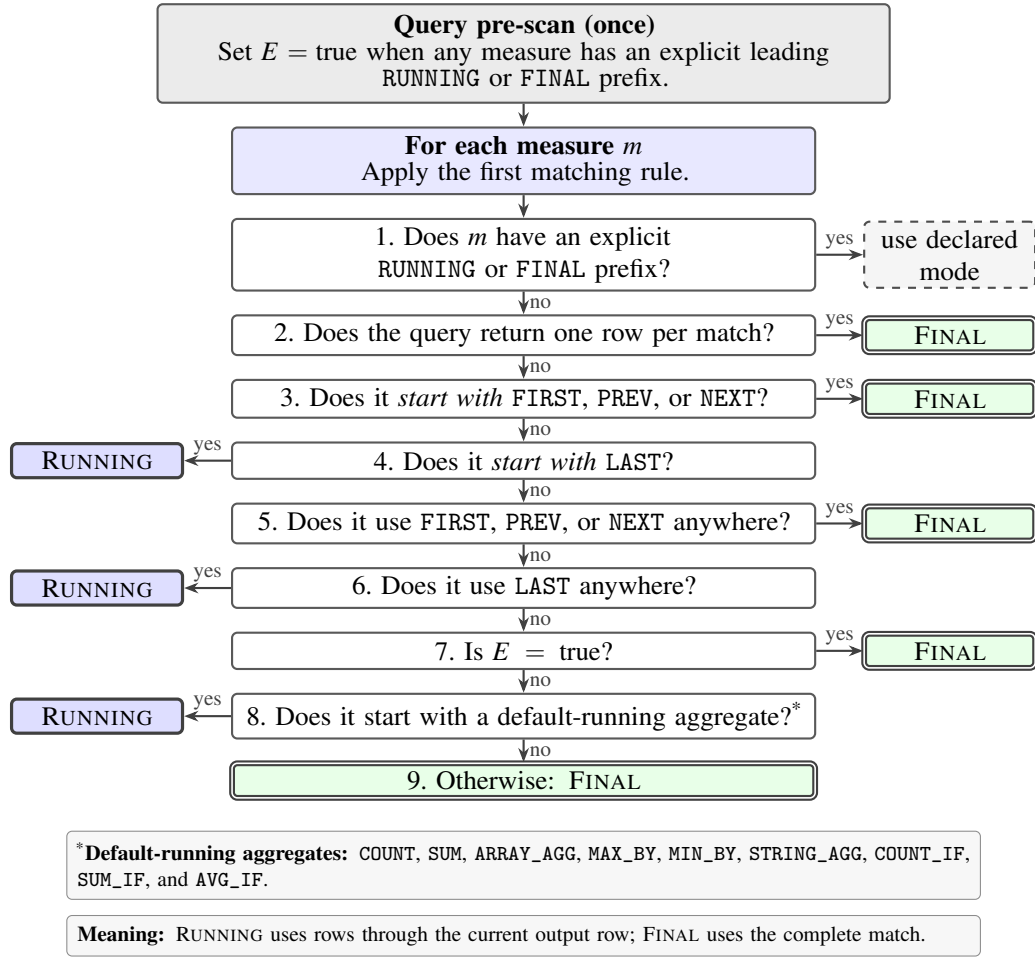


Figure 4.3: Resolving RUNNING and FINAL semantics.

With these attributes resolved, the execution configuration is handed to the pattern-compilation phase of Chapter 5.

4.5 CHAPTER SUMMARY

This chapter presented the five-stage design and its front-end boundaries. Parsing and validation create a checked AST. Logical-plan construction resolves the execution configuration, including output, skip, and measure modes. The design reuses the Trino ANTLR4 grammar and evaluates user expressions through a restricted AST evaluator. Chapter 5 continues with automata compilation, pandas execution, and optimization.

CHAPTER 5

IMPLEMENTATION AND OPTIMIZATION

This chapter shows how the validated logical plan from Chapter 4 becomes an executable query. A pattern is compiled into a finite automaton and applied to ordered pandas partitions. Guarded fast paths reduce repeated Python work without changing the supported semantics. Section 5.1 covers automata compilation, Section 5.2 the DataFrame execution model, and Section 5.3 the optimization layer.

5.1 PATTERN COMPILATION WITH FINITE AUTOMATA

Row-pattern matching is a sequence-recognition problem. A row may satisfy one or more pattern variables. A match assigns consecutive rows to those variables in an order accepted by the pattern. Finite automata provide a natural representation for this ordered recognition task [57]. The engine therefore compiles the pattern before it scans the data.

From pattern to NFA. The first automaton is a *non-deterministic finite automaton* (NFA). It may follow several branches and take ϵ -transitions, which consume no row. These properties represent alternation, optional parts, and repetition directly. The builder follows Thompson’s construction. Each pattern part becomes a small fragment, and the fragments are joined to form the whole pattern [58], [59]. The implementation also carries SQL-specific metadata for conditions, exclusions, anchors, and preference order. Figure 5.1 shows a simplified NFA for $A^+ B?$ that accepts the same row sequences as the compiled pattern. Intermediate wrapper states used during construction are not shown.

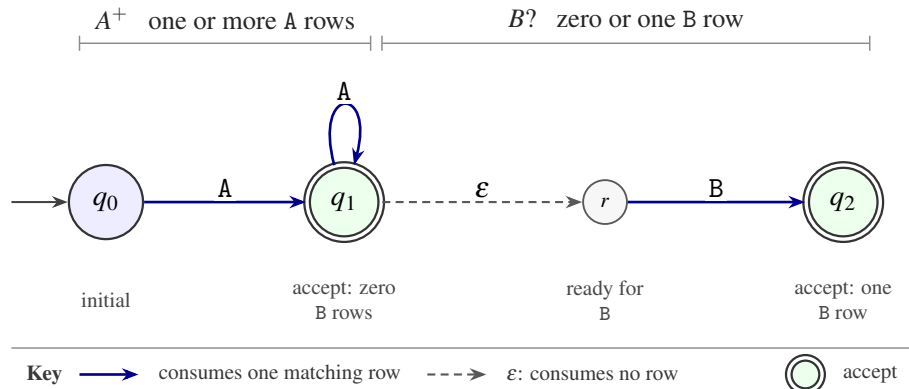


Figure 5.1: Language-equivalent NFA for $A^+ B?$: q_1 accepts with zero B rows, q_2 after one.

An A or B transition consumes one row for which that variable's DEFINE predicate is true. State q_1 is reached after one or more A rows and is accepting because B is optional. From q_1 , the NFA may accept, repeat A, or enter the optional fragment through the ε -transition. The transition into the connector state r consumes no row. Consuming one B row then reaches the second accepting state, q_2 . These alternatives show nondeterminism. The default greedy preference is stored as transition metadata and is not represented by the simplified state graph.

Reading the pattern. A left-to-right tokenizer reads variables, alternation, grouping, anchors, exclusions, and PERMUTE. It normalizes $*$, $+$, $?$, and $\{n, m\}$ into minimum and maximum counts together with a greedy or reluctant preference. The builder can then handle every quantifier through one internal representation.

Building the automaton. Each token contributes an NFA fragment. When several continuations are possible, transition metadata preserves their pattern order. PERMUTE over n variables represents all $n!$ orders. Construction is therefore guarded at eight variables and an estimated 50,000 branches. Exceeding either guard raises an explicit complexity error. Rows inside $\{- \dots -\}$ still take part in matching; only their output is excluded.

From NFA to DFA. The NFA is converted into a deterministic finite automaton (DFA) by subset construction. Each DFA state represents a complete set of NFA states. Its ε -closure includes every state reachable without consuming a row. Reusing a previously seen set prevents duplicate DFA states [57].

The resulting automaton has no ε -transitions. For eligible row-local pattern structures, the matcher can advance through the ordered partition without keeping the complete NFA frontier at every row. Predicates or structural choices that depend on tentative assignments use the exact search described in Section 5.2.1.

Subset construction is exponential in the worst case. The engine therefore applies a memory-aware state budget, an iteration limit, and a structural ceiling of 50,000 states. The memory budget may lower that ceiling on a constrained system. Reaching any limit raises a construction error. The engine never executes or caches a partial DFA, and it never truncates an NFA subset. None of the evaluated patterns reached these limits.

After construction, unreachable states are removed and transitions are normalized. The current reduction is conservative; it does not perform full DFA minimization. Transition metadata is preserved so the runtime can apply the required greedy and leftmost preference.

5.2 PANDAS DATAFRAME EXECUTION MODEL

pandas [14] stores each DataFrame column as an array. This suits both row-by-row matching and the vectorized fast paths in Section 5.3. Figure 4.1 shows the system boundary, while Figure 5.2 expands Stage 5. Its two inputs are the compiled execution plan and the pandas DataFrame. The SQL query is shown only as the upstream source of that plan. The figure presents five logical execution steps: partitioning, ordering, match enumeration with predicate evaluation, measure evaluation, and result materialization. An optimized physical path may fuse some operations without changing this logical flow.

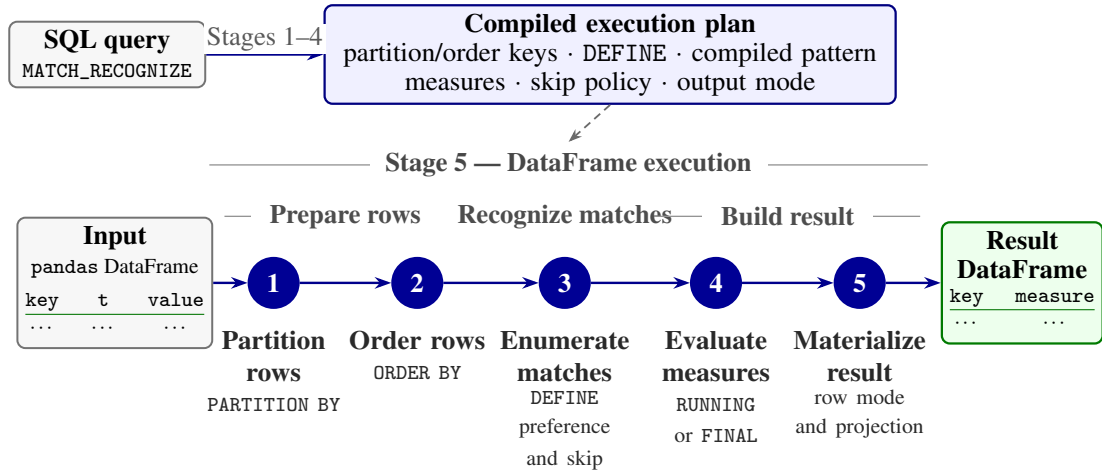


Figure 5.2: Inside the DataFrame execution stage.

Preparing the data. A match never crosses a partition boundary, so the DataFrame is first split on the PARTITION BY columns—or treated as one partition when there are none. The engine checks the ORDER BY columns. It keeps a partition that already follows the complete ordering specification and sorts it otherwise. Each key may use ASC or DESC and its own null placement. The sort is stable, so rows with equal keys keep their DataFrame order. This avoids repeating a sort whose result is already present. Because matches do not cross partitions, the remaining steps can process one ordered partition at a time.

Each DEFINE condition is turned into a small function once, at the start of the query. A variable used in PATTERN but not in DEFINE receives an always-true condition, as required by the supported semantics [7]. Row-local conditions may be evaluated for the whole partition. Other conditions are evaluated only when a candidate assignment needs them.

Finding the matches. Matching walks the rows in order. At each position, it tests the possible variable transitions and applies the pattern’s preference. Greedy quantifiers prefer a longer valid match. Reluctant quantifiers prefer a shorter one. Alternation follows declaration order. The automata carry this ordering information, while the runtime checks that the preferred path can still complete.

Each completed match records its start, end, variable assignments, and excluded rows. This state drives skip handling and measure evaluation. Exclusion changes only the rows emitted; it does not change the rows used by conditions or measures.

5.2.1 Exact Search for State-Dependent Conditions

The deterministic walk is used only when the predicates and pattern structure can be decided without revisiting tentative assignments. Other conditions depend on the tentative match. For example, accepting a row changes a running average used by a later condition. Some nested quantified alternatives also need a choice to be tested before their preference can be decided.

These cases use a preference-ordered depth-first search with backtracking. The search tentatively assigns rows, evaluates the affected conditions, and tries the next legal choice when a later step fails. It is iterative, so Python’s recursion depth does not limit the match length. A state budget and a search budget bound the explored states and steps. If either is exhausted, the query raises an explicit error instead of returning a partial or different match. Eligible row-local patterns remain on deterministic fast paths. Section 6.5.6 measures one supported state-dependent case.

5.2.2 Skip Modes, Output Modes, and Measures

After recording a match the engine must decide where to resume, which controls whether matches can overlap. SQL:2016 defines four `AFTER MATCH SKIP` modes, summarized with their next-start computation in Table 5.1. Empty matches always advance by one row. A variable-based target that does not occur in the match, or that points back to the current start, raises an error. This prevents a silent loop or a changed match sequence.

The engine supports `ONE ROW PER MATCH` and `ALL ROWS PER MATCH`. The latter includes `SHOW EMPTY MATCHES`, `OMIT EMPTY MATCHES`, and `WITH UNMATCHED ROWS` [7]. Empty matches are shown by default; omitting them removes their output rows, while unmatched mode also emits input rows not covered by a non-empty match. `CLASSIFIER()` can expose the variable assigned to an output row. Table 5.2 summarizes the row counts and measure contexts.

Table 5.1: The four *AFTER MATCH SKIP* modes and the resulting next scan position

Mode	Next-start computation
PAST LAST ROW (default)	$i \leftarrow best.end + 1$. No overlap is possible; the scan resumes immediately after the last matched row.
TO NEXT ROW	$i \leftarrow best.start + 1$. Overlapping matches are produced because only a single row is skipped regardless of match length.
TO FIRST V	$i \leftarrow \min(bindings[V])$. The scan resumes at the <i>first</i> row assigned to variable V (V may also be a SUBSET alias).
TO LAST V	$i \leftarrow \max(bindings[V])$. The scan resumes at the <i>last</i> row assigned to variable V (or SUBSET alias).

Table 5.2: Output modes and their row-count and measure-semantics behavior

Mode	Rows out	Measure evaluation context
ONE ROW PER MATCH	1 per match	Measures use the completed variable bindings.
ALL ROWS PER MATCH	One per visible matched row; SHOW EMPTY adds one row for each empty match	Measures use their resolved RUNNING or FINAL semantics for each emitted row.
ALL ROWS ... WITH UNMATCHED	Visible matched rows, plus each unmatched input row; empty matches follow the selected empty-match option	Matched rows carry measure values. Unmatched rows emit NULL for the measure columns.

Measure semantics are resolved before matching. Common navigation and aggregate forms are compiled once; other supported expressions use the general evaluator. A qualified aggregate uses rows assigned to its variable or SUBSET. An unqualified aggregate uses the applicable RUNNING match prefix or the complete FINAL match. Direct navigation plans use bounds-checked lookups into the ordered partition, and an out-of-range reference returns NULL. MATCH_NUMBER() numbers matches within each partition, while CLASSIFIER() reports the variable assigned to the output row. Under ALL ROWS PER MATCH, a RUNNING measure sees the match prefix through the current row. A FINAL measure sees the complete match.

5.3 PERFORMANCE OPTIMIZATION TECHNIQUES

One rule governs this whole layer: a faster path may never change the answer. Each technique below is therefore switched on only when the engine can be sure it applies, and the ordinary path takes over otherwise. Differential tests compare eligible fast paths with the general path on the same input and output mode.

The techniques remove repeated predicate work, avoid unnecessary sorting and materialization, and reuse compiled structures. Table 5.3 gives their activation boundaries.

Deciding conditions ahead of time. The main cost in the general path is repeated condition evaluation. Predicates that depend only on the current row, literals, and eligible fixed-offset PREV/NEXT forms can be evaluated as column operations. Their Boolean results are stored once per partition. The matcher then reads a mask instead of calling the expression evaluator for every candidate row.

The planner is conservative. FIRST, LAST, CLASSIFIER(), aggregates, and cross-variable references depend on the tentative match and cannot use a row-local result. A mixed AND predicate may still use its proven row-local conjuncts as a rejection mask; the remaining conjuncts are evaluated exactly. Supported expressions outside the compiled forms use the general evaluator. Unsupported forms are rejected.

The running-average condition used in the worst-case experiment receives a separate incremental plan. The aggregate is updated as the matched prefix grows instead of being recomputed from the beginning. This removes the quadratic repeated-aggregation cost for that supported form. It does not make every state-dependent predicate vectorizable (Section 6.5.6).

Turning the search into a lookup. When all variable predicates have masks, the engine builds a compact transition index for the DFA states. Matching becomes array lookup and integer comparison rather than repeated dictionary and function work. Assignments can be stored as row segments instead of one Python object per row.

A linear pattern such as `A B{2,4} C+` allows a second plan. Run-length arrays record how many consecutive rows satisfy each variable. Failed token–position pairs are memoized, so the same impossible suffix is not explored again. Patterns with alternation use the row-local DFA path when eligible. State-dependent choices use the exact matcher. Neither case enters the linear run-length plan.

For eligible output modes, matching and materialization are fused. One path emits `ONE ROW PER MATCH` results directly. Another publishes compact segments for `ALL ROWS PER MATCH`. Both require the default skip mode and fully compiled measures. Any anchor, empty-match rule, measure, or output option outside their proof boundary returns to the complete path.

Computing measures once. Common measures are compiled once into direct plans. Simple `SUM`, `AVG`, `COUNT`, statistical aggregates, and direct navigation can then use arrays or prefix state. A `RUNNING` aggregate over a long match is updated incrementally instead of being recomputed for every output row. The compiled layer also covers selected arithmetic combinations and distinct aggregates. Filters and expressions outside these proven forms use the general measure evaluator.

Reducing repeated work and memory. Compiled automata are stored in a bounded, thread-safe *least-recently-used* (LRU) cache. The key includes the pattern, its `DEFINE` conditions, and compilation-relevant `SUBSET` data. Repeating the same structure therefore avoids NFA and DFA construction. Condition caches are also bounded. For a large partition, a small row sample warms common evaluation paths.

The executor avoids copying all grouped partitions before matching. For eligible default-skip, non-`PERMUTE` queries, it exposes one partition through a lazy row view with a bounded row cache. Other paths materialize partition rows and may retain them until output assembly. Match membership uses compact masks and segments where the semantics allow it. A conservative final-output projection gathers only source columns needed by the result; a wildcard or an expression it cannot prove safe keeps all columns. Eligible fast paths assemble output columns directly before creating the result `DataFrame`. These choices lower temporary memory, but they do not make execution streaming: it retains the input and result, and some paths also retain materialized rows or whole-column working arrays. A cache budget bounds cache capacity, a state budget bounds automaton construction, and a search budget bounds exact search.

Table 5.3: Performance optimizations and their activation conditions

Technique	Activation condition	Primary benefit
Ordered-input check	Requested ORDER BY sequence already present	Avoids an unnecessary sort and DataFrame copy
Vectorized condition masks	Row-local expression, or safe row-local conjuncts in a mixed AND	Replaces repeated Python evaluation with column operations and exact pre-filters
Row-local DFA decision tables	All predicates are vectorized, with no empty-match, anchor, exclusion, PERMUTE, SUBSET, reluctant, or state-dependent requirement	Replaces repeated transition dispatch with compact indexing and segment assignments
Linear run-length plan	Default skip, a plain linear variable sequence, supported quantifiers, and row-local masks	Skips complete runs and memoizes failed suffixes
Fused output drivers	Default skip, no unmatched rows, anchors, or empty matches, an eligible matcher, and fully compiled measures	Avoids intermediate match and row dictionaries
Compiled measure and prefix plans	Supported direct navigation or aggregate form	Reuses prepared array state; avoids repeated prefix aggregation
LRU pattern cache	Caching enabled and the complete automaton fits the cache budget	Eliminates redundant NFA/DFA construction across repeated queries
Lazy rows and output projection	Partition path supports lazy access; selected source columns can be resolved safely	Reduces row copies, retained Python objects, and output width

5.4 CHAPTER SUMMARY

This chapter followed a query from a compiled pattern to a result. The pattern becomes an NFA and then a complete, guarded DFA. The input is partitioned and ordered before matching. Row-local queries use the deterministic path only when their predicates and structure are eligible. Predicates or structural choices that depend on tentative assignments use exact search and fail explicitly if its search budget is exhausted.

Major speed gains come from safe column-wise condition evaluation, linear run-length plans, compiled measures, fused output, and pattern caching. Lazy row access, projection, compact masks, and the cache and state budgets reduce temporary memory. Every fast path is guarded. If its conditions are not met, the complete path remains responsible for the result. Chapter 6 measures the resulting correctness, performance, scalability, and memory behavior.

CHAPTER 6

EXPERIMENTAL EVALUATION

This chapter evaluates the implementation on five criteria: correctness against Trino and Oracle, SQL:2016 feature coverage, performance, scalability, and usability. Section 6.1 defines the methodology, Section 6.2 the datasets and patterns, Section 6.3 validates the concept example by example, Section 6.4 reports coverage and complexity results, and Section 6.5 presents the cross-system performance analysis.

6.1 EVALUATION METHODOLOGY

Section 6.1.1 defines the five criteria and how each is measured; Section 6.1.2 describes the hardware and software environment.

6.1.1 Evaluation Criteria and Metrics

Correctness. The result DataFrame is compared against two independent SQL references, Trino 473 and Oracle 21c EE, by exact row-level equality (same rows, values, and order where deterministic). This covers the two small synthetic tables—orders (Table 6.1) and stock_price (Table 6.2)—and the 30 pattern–size combinations of the Amazon benchmark (Section 6.5.2). The SQL engines are the correctness references. The hand-written Pandas functions provide an additional output cross-check and the usability baseline (Section 6.3.4).

SQL feature coverage. Two views are recorded. *Feature coverage* marks each main MATCH_RECOGNIZE clause—partitioning, ordering, PATTERN, DEFINE, MEASURES, SUBSET, the output and skip modes, and navigation functions [2], [3], [48]—as *supported*, *partially supported*, or *unsupported* against the reference tests (Table 6.7). *Pattern-type coverage* groups the test cases by row-pattern type (simple sequences, quantified repetitions, trends, recovery, alternation, exclusions, anchors, and PERMUTE); a type is covered when at least one case matches the reference output (Table 6.8).

Performance. Execution time is the client-observed end-to-end latency per pattern–size run on the Amazon UK Products dataset (Section 6.5): five pattern families, six sizes, result transfer included. Each cell runs five warmups to reduce cold-start and JIT effects, followed by twenty measured runs, each preceded by a five-second idle window that sets the pre-query memory baseline. Outliers are removed per metric with the $1.5 \times \text{IQR}$ rule; cells report the mean of the retained runs with its standard deviation.

Throughput is computed per run—dataset size divided by that run’s time—and then averaged the same way, so it is the mean of the rates rather than the rate of the mean. For the main benchmark, the two memory metrics use one instrument for all three systems. Their scope is defined in Section 6.5.4.

Scalability. Mean execution time is regressed against row count for each pattern family. The reported R^2 shows how closely the measured points fit a straight line. It is reported for both the six-size benchmark and the larger stress test. A high R^2 describes the measured range; it does not prove the algorithm’s asymptotic complexity.

Usability (developer effort). For each worked example, the line count of the idiomatic Pandas solution is compared with the single `MATCH_RECOGNIZE` statement (Table 6.6); the reduction measures developer effort [2], [3], [25], [60].

6.1.2 Environment and Configuration

The experiments ran on a workstation with an Intel Core i7-11800H processor and 64 GB of RAM under Ubuntu 24.04.4 LTS. For the cross-system benchmark, each system had the same outer control-group limit of one CPU core and 32 GB. Section 6.5.1 details the enforcement mechanisms.

The scalability stress test (Section 6.5.7) instead uses a 58 GB memory limit. Reaching hundreds of millions of rows on equal terms also meant changing the Trino deployment: it reads from a disk-backed Hive/Parquet connector rather than its in-memory connector. All database containers run on the host’s native Docker daemon, so container limits map directly to host control groups—the same instrument used for the engine.

The software stack comprised the two baseline SQL engines, Trino 473 and Oracle 21c EE, each run as a Docker container. It also included the Java 11.0.25 LTS and Python 3.12.7 runtimes and the core Python libraries pandas 2.3.3, numpy 2.1.3, antlr4-python3-runtime 4.13.2, antlr4-tools 0.2.1, and pytest 8.3.4. The implementation itself uses ANTLR4 to parse Trino SQL, pandas for DataFrame management, and custom AST/NFA/DFA modules for pattern detection (Chapter 4).

Unless stated otherwise, all experiments use this configuration. For correctness comparisons, identical data are loaded into all three systems and result sets are compared by row order and measure values (after normalizing case-only column-name differences).

6.2 DATASET AND TEST PATTERNS

6.2.1 Datasets

Four datasets are used. Two are small synthetic tables that can be verified by hand, `orders` and `stock_price`. The real-world Amazon UK Products dataset, up to 2.2 million rows, carries the cross-system timing and memory benchmark. The Amazon Reviews 2023 corpus, up to 227.9 million rows, carries the scalability test (Section 6.5.7).

Dataset 1: the `orders` table. The `orders` table supports the first worked example. It comprises three attributes [8]: `customer_id` (VARCHAR), which identifies the customer (two distinct values, `cust_1` and `cust_2`); `order_date` (DATE), the date the order was placed (ranging from 2020-05-11 to 2020-05-18); and `price` (INTEGER), the purchase amount on that date. Each customer appears multiple times in chronological order—`cust_1` spans five dates and `cust_2` spans three—which enables focused validation of partitioning, ordering, and pattern detection (e.g. sequences of rising or falling prices per customer). Table 6.1 shows the eight records used.

Table 6.1: Sample orders dataset (*customer_id, order_date, price*)

customer_id	order_date	price
cust_1	2020-05-11	100
cust_1	2020-05-12	200
cust_2	2020-05-13	8
cust_1	2020-05-14	100
cust_2	2020-05-15	4
cust_1	2020-05-16	50
cust_1	2020-05-17	100
cust_2	2020-05-18	6

Dataset 2: the `stock_price` table. The `stock_price` table supports the second worked example. It represents a history of daily stock prices for market analysis and includes three columns: `company` (the company name), `price_date` (the trading date), and `price` (the stock’s value on that day) [16]. Two companies, A and B, are recorded over ten trading days, yielding twenty rows that support per-company trend detection. Table 6.2 shows the data.

Table 6.2: The tabular form of the *stock_price* table

Company	Price_date	Price
A	2020-10-01	50
B	2020-10-01	89
A	2020-10-02	36
B	2020-10-02	24
A	2020-10-03	39
B	2020-10-03	37
A	2020-10-04	42
B	2020-10-04	63
A	2020-10-05	30
B	2020-10-05	65
A	2020-10-06	47
B	2020-10-06	56
A	2020-10-07	71
B	2020-10-07	50
A	2020-10-08	80
B	2020-10-08	54
A	2020-10-09	75
B	2020-10-09	30
A	2020-10-10	63
B	2020-10-10	32

Dataset 3: the Amazon UK Products dataset. The Amazon UK Products dataset [17] holds roughly 2.2 million product records from Amazon’s UK marketplace as one CSV file. The benchmark uses four of its columns: *stars* (average rating, 1–5), *reviews* (review count), *price* (GBP), and *categoryName*; the remaining columns (IDs, titles, URLs, flags) are not needed. It supplies the data volume for the performance benchmark; the derived category column and the six dataset sizes are described in Section 6.5.1.

Dataset 4: the Amazon Reviews 2023 corpus. The scalability test (Section 6.5.7) needs far more rows than the product tables provide, so it uses the Amazon Reviews 2023 corpus [18] from the McAuley Lab at UC San Diego—roughly 571 million genuine review events across product categories. Only the pattern-relevant fields are taken: *user_id*, *parent_asin*, *rating*, and *timestamp*. Review text, images, and product metadata are excluded. Five categories are merged in global timestamp order into nested prefixes of 5 M to 227.9 M rows. Each review’s 1–5 rating is mapped to the same A–E category alphabet as the benchmark. The same five pattern families are used. Four keep the main benchmark structure with different rating labels; the stress test’s optional

family uses E D? C+ B*. Unlike a tiled copy of a smaller table, every row is a distinct real event, which makes this the more honest test of scaling behavior.

6.3 VALIDATION OF CONCEPT

Each worked example compares the proposed engine, Trino, Oracle, and an equivalent hand-written Pandas function. The comparison covers output, exercised clauses, and developer effort. Beyond these examples, the engine is checked by the regression suite (709 passing test cases [19]). The 30 benchmark pattern–size combinations in Section 6.5.2 also agree across all three systems.

6.3.1 Example 1: Dip Pattern on the orders Dataset

The first example detects a ‘dip’—three consecutive orders with successively lower prices—on the orders dataset (Dataset 1, Table 6.1). The same detection is expressed three ways: as a Trino SQL query (Listing 6.1), as a call to the proposed engine (Listing 6.2), and as a hand-written Pandas function (Listing 6.3). Oracle provides the second SQL reference. All four return the same match (Table 6.3).

```

1 CREATE TABLE memory.default.orders (
2     customer_id VARCHAR,
3     order_date   DATE,
4     price        INTEGER
5 );
6
7 INSERT INTO memory.default.orders VALUES
8     ('cust_1', DATE '2020-05-11', 100),
9     ('cust_1', DATE '2020-05-12', 200),
10    ('cust_2', DATE '2020-05-13',   8),
11    ('cust_1', DATE '2020-05-14', 100),
12    ('cust_2', DATE '2020-05-15',   4),
13    ('cust_1', DATE '2020-05-16',  50),
14    ('cust_1', DATE '2020-05-17', 100),
15    ('cust_2', DATE '2020-05-18',   6);
16
17 SELECT customer_id, start_date, end_date, bottom_price
18 FROM memory.default.orders
19 MATCH_RECOGNIZE (
20     PARTITION BY customer_id
21     ORDER BY order_date

```

```

22 MEASURES
23     A.order_date AS start_date,
24     C.order_date AS end_date,
25     C.price      AS bottom_price
26 PATTERN (A B C)
27 DEFINE
28     B AS B.price < A.price,
29     C AS C.price < B.price
30 );

```

Listing 6.1: Trino SQL: create and load the orders table, then detect the dip pattern with MATCH_RECOGNIZE.

```

1 import pandas as pd
2 from pandas_match_recognize import match_recognize
3 orders = pd.DataFrame({
4     "customer_id": ["cust_1", "cust_1", "cust_2", "cust_1",
5     "cust_2", "cust_1", "cust_1", "cust_2"],
6     "order_date": ["2020-05-11", "2020-05-12", "2020-05-13", "
7     2020-05-14",
8     "2020-05-15", "2020-05-16", "2020-05-17", "
9     2020-05-18"],
10    "price":      [100, 200, 8, 100, 4, 50, 100, 6],
11 })
12
13 query = """
14 SELECT customer_id, start_date, end_date, bottom_price
15 FROM orders
16 MATCH_RECOGNIZE (
17     PARTITION BY customer_id
18     ORDER BY order_date
19     MEASURES
20     A.order_date AS start_date,
21     C.order_date AS end_date,
22     C.price      AS bottom_price
23     PATTERN (A B C)
24     DEFINE
25     B AS B.price < A.price,
26     C AS C.price < B.price
27 ); """

```

```

26 result = match_recognize(query, orders)
27 print(result)

```

Listing 6.2: Proposed engine: the same query as a MATCH_RECOGNIZE string over an orders DataFrame.

```

1  import pandas as pd
2
3  def detect_dip(df, cust_col="customer_id",
4                date_col="order_date", price_col="price"):
5      g = df.copy()
6      g[date_col] = pd.to_datetime(g[date_col])
7      g = g.sort_values([cust_col, date_col]).reset_index(drop=
8          True)
9      rows = []
10     for cust, grp in g.groupby(cust_col, sort=False):
11         grp = grp.reset_index(drop=True)
12         i = 0
13         while i + 2 < len(grp):
14             a, b, c = (grp.loc[i, price_col],
15                       grp.loc[i + 1, price_col],
16                       grp.loc[i + 2, price_col])
17             if b < a and c < b:          # A B C:
18                 successively lower prices
19                 rows.append({
20                     "customer_id": cust,
21                     "start_date": grp.loc[i, date_col].
22                         date(),
23                     "end_date": grp.loc[i + 2, date_col].
24                         date(),
25                     "bottom_price": c,
26                 })
27                 i += 3                    # AFTER MATCH SKIP
28                                     PAST LAST ROW
29             else:
30                 i += 1
31     return pd.DataFrame(rows)
32
33 result = detect_dip(orders)
34 print(result)

```

Listing 6.3: Hand-written Pandas function for the same ‘dip’ detection.

For the *correctness* criterion, all four implementations return the same single match, shown in Table 6.3. The query also exercises the `PARTITION BY`, `ORDER BY`, `PATTERN`, `DEFINE`, and `MEASURES` clauses, and covers the simple-sequence pattern type.

Table 6.3: Dip-pattern result: all four systems return the identical match [48]

Implementation	customer_id	start_date → end_date	bottom_price
Trino 473 (SQL)	cust_1	2020-05-12 → 2020-05-16	50
Oracle 21c EE (SQL)	cust_1	2020-05-12 → 2020-05-16	50
Proposed engine	cust_1	2020-05-12 → 2020-05-16	50
Pandas function	cust_1	2020-05-12 → 2020-05-16	50

6.3.2 Example 2: Stock-Price Analysis on the `stock_price` Dataset

The second example detects bullish momentum on the `stock_price` table: a three-row window where the second and third rows each rise. It partitions by company, orders by date, and uses `PATTERN (A B C)` with `PREV`-based `DEFINE` predicates (A is implicitly always true). The same detection is written as Trino SQL (Listing 6.4), a call to the proposed engine (Listing 6.5), and a hand-written Pandas function (Listing 6.6); Table 6.4 confirms all four implementations return the identical three matches.

```

1 CREATE TABLE memory.default.stock_price (
2     Company    VARCHAR,
3     Price_date DATE,
4     Price      INTEGER
5 );
6
7 INSERT INTO memory.default.stock_price (Company, Price_date,
8     Price) VALUES
9     ('A', DATE '2020-10-01', 50), ('B', DATE '2020-10-01',
10    89),
11    ('A', DATE '2020-10-02', 36), ('B', DATE '2020-10-02',
    24),
    ('A', DATE '2020-10-03', 39), ('B', DATE '2020-10-03',
    37),
    ('A', DATE '2020-10-04', 42), ('B', DATE '2020-10-04',
    63),

```

```

12      ('A', DATE '2020-10-05', 30), ('B', DATE '2020-10-05',
13      65),
14      ('A', DATE '2020-10-06', 47), ('B', DATE '2020-10-06',
15      56),
16      ('A', DATE '2020-10-07', 71), ('B', DATE '2020-10-07',
17      50),
18      ('A', DATE '2020-10-08', 80), ('B', DATE '2020-10-08',
19      54),
20      ('A', DATE '2020-10-09', 75), ('B', DATE '2020-10-09',
21      30),
22      ('A', DATE '2020-10-10', 63), ('B', DATE '2020-10-10',
23      32);
24
25 SELECT *
26 FROM memory.default.stock_price
27 MATCH_RECOGNIZE (
28   PARTITION BY Company
29   ORDER BY Price_date
30   MEASURES
31     FIRST(A.Price) AS start_price,
32     LAST(C.Price) AS end_price,
33     MATCH_NUMBER() AS match_id
34   PATTERN (A B C)
35   DEFINE
36     B AS B.Price > PREV(B.Price),
37     C AS C.Price > PREV(C.Price)
38 );

```

Listing 6.4: Trino SQL: create and load the `stock_price` table, then detect the increasing-window pattern with `MATCH_RECOGNIZE`.

```

1 import pandas as pd
2 from pandas_match_recognize import match_recognize
3 data = {
4     "Company": ["A", "B", "A", "B", "A", "B", "A", "B", "A", "B",
5     "A", "B", "A", "B", "A", "B", "A", "B", "A", "B"],
6     "Price_date": ["2020-10-01", "2020-10-01", "2020-10-02", "
7     2020-10-02",
8     "2020-10-03", "2020-10-03", "2020-10-04", "
9     2020-10-04",

```

```

8         "2020-10-05", "2020-10-05", "2020-10-06", "
          2020-10-06",
9         "2020-10-07", "2020-10-07", "2020-10-08", "
          2020-10-08",
10        "2020-10-09", "2020-10-09", "2020-10-10", "
          2020-10-10"],
11        "Price": [50, 89, 36, 24, 39, 37, 42, 63, 30, 65,
12                  47, 56, 71, 50, 80, 54, 75, 30, 63, 32],
13    }
14    df = pd.DataFrame(data)
15    df["Price_date"] = pd.to_datetime(df["Price_date"])
16
17    query = """
18    SELECT *
19    FROM stock_price
20    MATCH_RECOGNIZE (
21        PARTITION BY Company
22        ORDER BY Price_date
23        MEASURES
24            FIRST(A.Price) AS start_price,
25            LAST(C.Price) AS end_price,
26            MATCH_NUMBER() AS match_id
27        PATTERN (A B C)
28        DEFINE
29            B AS B.Price > PREV(B.Price),
30            C AS C.Price > PREV(C.Price)
31    ); """
32    result = match_recognize(query, df)
33    print(result)

```

Listing 6.5: Proposed engine: the same query as a MATCH_RECOGNIZE string over a stock_price DataFrame.

```

1    import pandas as pd
2
3    def detect_two_consecutive_increases(
4        df: pd.DataFrame,
5        company_col: str = "Company",
6        date_col: str = "Price_date",
7        price_col: str = "Price",

```

```

8  ) -> pd.DataFrame:
9      gdf = df.copy()
10     gdf[date_col] = pd.to_datetime(gdf[date_col])
11     gdf = gdf.sort_values([company_col, date_col]).
         reset_index(drop=True)
12     out_rows = []
13     for company, g in gdf.groupby(company_col, sort=False):
14         g = g.reset_index(drop=True)
15         inc = g[price_col].diff() > 0
16         start_mask = (inc.shift(-1).fillna(False)
17                       & inc.shift(-2).fillna(False))
18         candidates = [int(i) for i, ok in enumerate(
19                       start_mask)
20                       if ok and i + 2 < len(g)]
21         match_id, next_start_pos = 0, 0
22         for i in candidates:
23             if i < next_start_pos:
24                 continue
25             match_id += 1
26             out_rows.append({
27                 "company":      company,
28                 "start_price":  g.loc[i, price_col],
29                 "end_price":    g.loc[i + 2, price_col],
30                 "match_id":     match_id,
31             })
32             next_start_pos = i + 3    # AFTER MATCH SKIP PAST
33                                     LAST ROW
34     return pd.DataFrame(out_rows)
35
36 result = detect_two_consecutive_increases(df)
37 print(result)

```

Listing 6.6: Hand-written Pandas function for the same increasing-window detection.

All four implementations return the same three matches (Table 6.4), covering partitioning, ordering, implicit DEFINE, PREV navigation, MATCH_NUMBER(), and measure materialization.

Table 6.4: Stock-price result: all four systems return these three identical matches [16], [61]

Match	company	start_price	end_price	match_id
1	A	36	42	1
2	A	30	71	2
3	B	24	63	1

6.3.3 Example 3: V-Shape Recovery on the stock_price Dataset

The third example detects a *V-shape recovery* on the same table: a strictly falling run followed by a strictly rising one. Unlike the fixed three-row patterns above, PATTERN (A B+ C+) uses the + quantifier twice, with the fall and rise defined through PREV. The measures report the start, bottom, and recovered prices. As before, the same detection is expressed as a Trino SQL query (Listing 6.7), as a call to the proposed engine (Listing 6.8), and as a hand-written Pandas function (Listing 6.9). Oracle provides the second SQL reference. All four return the same matches (Table 6.5).

```

1 SELECT *
2 FROM memory.default.stock_price
3 MATCH_RECOGNIZE (
4   PARTITION BY Company
5   ORDER BY Price_date
6   MEASURES
7     FIRST(A.Price) AS start_price ,
8     LAST(B.Price)  AS bottom_price ,
9     LAST(C.Price)  AS end_price ,
10    MATCH_NUMBER() AS match_id
11   ONE ROW PER MATCH
12   PATTERN (A B+ C+)
13   DEFINE
14     B AS B.Price < PREV(B.Price),
15     C AS C.Price > PREV(C.Price)
16 );

```

Listing 6.7: Trino SQL: detect the V-shape recovery on the stock_price table (already created and loaded in Listing 6.4).

```

1 from pandas_match_recognize import match_recognize
2
3 query = """
4 SELECT *

```

```

5 FROM stock_price
6 MATCH_RECOGNIZE (
7     PARTITION BY Company
8     ORDER BY Price_date
9     MEASURES
10        FIRST(A.Price) AS start_price,
11        LAST(B.Price)  AS bottom_price,
12        LAST(C.Price)  AS end_price,
13        MATCH_NUMBER() AS match_id
14    ONE ROW PER MATCH
15    PATTERN (A B+ C+)
16    DEFINE
17        B AS B.Price < PREV(B.Price),
18        C AS C.Price > PREV(C.Price)
19 ); """
20 result = match_recognize(query, df)
21 print(result)

```

Listing 6.8: Proposed engine: the same V-shape query over the stock_price DataFrame df of Listing 6.5.

```

1 import pandas as pd
2
3 def detect_v_shapes(df, key="Company",
4                     ts="Price_date", val="Price"):
5     g = df.copy()
6     g[ts] = pd.to_datetime(g[ts])
7     g = g.sort_values([key, ts]).reset_index(drop=True)
8     out = []
9     for company, grp in g.groupby(key, sort=False):
10         v = grp[val].to_numpy()
11         n, i, match_id = len(v), 0, 0
12         while i < n - 2:
13             j = i
14             while j + 1 < n and v[j + 1] < v[j]:    # B+:
15                 strictly falling
16                 j += 1
17             if j == i:                                # need at
18                 least one fall
19                 i += 1

```

```

18         continue
19     k = j
20     while k + 1 < n and v[k + 1] > v[k]:      # C+:
21         strictly rising
22         k += 1
23     if k == j:                                # need at
24         least one rise
25         i += 1
26         continue
27     match_id += 1
28     out.append({
29         "company":      company,
30         "start_price":  int(v[i]),
31         "bottom_price": int(v[j]),
32         "end_price":    int(v[k]),
33         "match_id":     match_id,
34     })
35     i = k + 1                                # AFTER
36     MATCH SKIP PAST LAST ROW
37     return pd.DataFrame(out)
38
39 result = detect_v_shapes(df)
40 print(result)

```

Listing 6.9: Hand-written Pandas function for the same V-shape detection.

For the *correctness* criterion, all four implementations return the same three matches, shown in Table 6.5. Beyond the clauses of the earlier examples, this query additionally exercises the + quantifier (one-or-more) on two pattern variables, confirming that the engine’s greedy consecutive-quantifier handling agrees with the reference engines.

Table 6.5: V-shape result: all four systems return these three identical matches [16], [61]

Match	company	start_price	bottom_price	end_price
1	A	50	36	42
2	B	89	24	65
3	B	56	50	54

6.3.4 Usability and Developer Effort

Each worked example expresses the same detection three ways: SQL query, engine call, and hand-written Pandas function. Table 6.6 counts the detection logic on both sides—the `MATCH_RECOGNIZE` statement against the Pandas detection function—in non-blank, non-comment lines. Data setup, the imports, and the printing are shared, so they are left out of both. The declarative form takes 14 vs 24, 14 vs 30, and 16 vs 32 lines, and the gap widens with pattern complexity.

Table 6.6: Lines-of-code comparison for the three worked examples.

Example	Implementation	Notes	LOC
Dip (orders)	<code>MATCH_RECOGNIZE</code> statement	single declarative query	14
Dip (orders)	Hand-written Pandas function	loop / index / skip bookkeeping	24
Stock (stock_price)	<code>MATCH_RECOGNIZE</code> statement	single declarative query	14
Stock (stock_price)	Hand-written Pandas function	groupby/shift/index bookkeeping	30
V-shape (stock_price)	<code>MATCH_RECOGNIZE</code> statement	quantified declarative query	16
V-shape (stock_price)	Hand-written Pandas function	nested falling/rising run scan	32

The declarative form states *what* to find in one statement and lets the engine decide *how*. Each Pandas function hand-codes grouping, sorting, and skip logic, where one off-by-one silently changes the result. That is why the procedural baselines were more error-prone [60]. Against an external SQL engine the value is *where* it runs: in-process on the DataFrame, with no `CREATE TABLE/INSERT` round-trip and a footprint below 0.6 GB versus 2.0 GB (Oracle) and 14.6 GB (Trino). The trade-off is scope—the supported subset only, without a database query planner or distributed execution. This is a code-level comparison, not a user study (Chapter 7).

6.4 COVERAGE AND COMPLEXITY RESULTS

This section reports feature and pattern coverage, then analyzes theoretical complexity. Usability was covered with the worked examples.

6.4.1 SQL Feature Coverage

Two dimensions are reported: implemented clauses (Table 6.7) and detected pattern types (Table 6.8), both checked against the Trino-aligned reference tests. *Partially*

supported means the common forms are implemented and verified; rarer variants are outside the validated scope.

Table 6.7: Support status of *MATCH_RECOGNIZE* features in the proposed engine

Feature	Status
PARTITION BY	supported
ORDER BY	supported
Quantifiers * + ? {n,m}	supported
Alternation (A B)	supported
Grouping / nesting	supported
Exclusion {- ... -}	supported
Anchors ^, \$	supported
PERMUTE	supported
DEFINE predicates	supported
Navigation PREV/NEXT/FIRST/LAST	supported
MEASURES core aggregates	supported
RUNNING/FINAL semantics	supported
SUBSET	supported
ONE ROW PER MATCH	supported
ALL ROWS PER MATCH	supported
AFTER MATCH SKIP modes	supported
MATCH_NUMBER(), CLASSIFIER()	supported

In summary, the engine supports the evaluated R010-style subset: partitioning; ordering; the listed PATTERN constructs; DEFINE predicates; SUBSET union variables; MEASURES with supported navigation functions and aggregates; RUNNING and FINAL semantics; both output modes; the evaluated AFTER MATCH SKIP modes; and MATCH_NUMBER()/CLASSIFIER().

Inside DEFINE and MEASURES it also evaluates arithmetic, comparisons, IN lists, NULL checks, and common scalar and string functions (LOWER, CAST, COALESCE, CASE, ||). Constant scalar subqueries without a FROM clause are supported inside tested expressions. Table-reading and correlated subqueries are not. General ORDER BY expressions, general relational planning, and distributed execution are also outside the evaluated R010 path. The separate R020 window path is not evaluated in this chapter. These limits define the scope of Chapters 4 and 5.

At the pattern level, Table 6.8 lists the row-pattern types the engine detects, each with a regex example and a simple use case. A type counts as covered when at least one test case exercises it and produces output identical to the reference systems.

Table 6.8: Pattern-type coverage, with a regex example and a simple use case for each type

Pattern type	Pattern (regex)	Simple use case	Covered
Simple sequence	A+ B+	a run of one category then a run of another	yes
Alternation	A (B C)+ D	a fixed start, mixed intermediate states, a fixed end	yes
Bounded quantifier	A{1,5} B* C+	one to five of one event, optional filler, then one or more of another	yes
Optional parts	A+ B? C*	a required run with an optional middle and tail	yes
Grouping / nesting	(A B)+ (C{1,3} D*)+	repeated groups nested in a larger repeated pattern	yes
Rising / falling trend	A B+	values rising (or falling) over several consecutive rows	yes
Recovery (V-shape)	A B+ C+	a fall followed by a rise—a dip that recovers	yes
Exclusion	{- A -} B+	detect a context but omit some of its rows from the result	yes
Anchors	^A, A\$	the match must be at the partition start or end	yes
Permutation	PERMUTE(A, B, C)	the same set of events in any order	yes

6.4.2 Complexity Analysis

Complexity is analyzed twice: theoretically here, empirically in Section 6.5. Theory predicts scaling; experiments measure the implementation on a specific machine and dataset.

Table 6.9: Theoretical and empirical views of complexity analysis

Theoretical Analysis	Empirical Analysis
Mathematical analysis.	Experimental measurement.
Independent of hardware and software environment.	Depends on the hardware, operating system, Python runtime, pandas, and memory behavior.
Describes how the running time grows as the input grows, using asymptotic notation (Big-O, Big- Θ , Big- Ω).	Uses actual execution time, throughput, and memory usage.
Predicts scalability as input size grows.	Measures real-world performance for the evaluated benchmark.

Theoretical analysis. Costs are stated in asymptotic notation: $O(f(n))$ is an upper bound, $\Omega(f(n))$ a lower bound, and $\Theta(f(n))$ both at once. Five symbols recur: n input rows, m pattern symbols (a bounded quantifier $\{i, j\}$ counts as j copies of its fragment, since the builder clones the fragment per repetition), d DEFINE variables, k matches, ℓ average rows per match that are materialized in the output. Thus $\ell = 1$ for ONE ROW PER MATCH; it can be larger for ALL ROWS PER MATCH.

Running a query does five kinds of work. Parsing reads the query and extracts its MATCH_RECOGNIZE clauses. Automaton construction builds the pattern’s automaton, NFA then DFA. Ordering sorts the rows. Condition checking tests the DEFINE conditions on every row. Matching scans for matches and builds the output.

Parsing and automaton construction depend on the pattern, not the data: NFA construction is $O(m)$, and the NFA-to-DFA conversion is $O(2^m)$ states in the worst case but a handful in practice. For a fixed pattern this is a one-time constant. *Ordering* is the only data-dependent cost above linear for the fixed row-local benchmark path. A full comparison sort costs $O(n \log n)$, done per partition—more precisely $\sum_i O(n_i \log n_i)$ over partitions of n_i rows, which the benchmark’s single partition bounds by $O(n \log n)$. But the engine sorts only when needed. It first checks in $O(n)$ whether the rows are already ordered and, if so, skips the sort. *Condition checking* prepares one Boolean result per row and variable on the compiled row-local path, costing $O(nd)$. *Matching and output* then costs $O(n + k\ell)$ on the compiled default-skip path: one scan plus the rows materialized for the k matches. These bounds describe the path used by the main benchmark. They are not a bound for every supported expression and skip mode.

Combining the data-dependent work, the cost has *two independent axes*, and it is important to keep them apart. One is growth with the data size n for a fixed pattern. The other is growth with the pattern complexity for fixed data.

Axis 1—growth with the input size n , for a fixed compiled row-local pattern. This is the axis the benchmark scales. The automaton is fixed, and every data-dependent stage except ordering is linear, so only whether a sort is needed separates the two cases:

- **Best case** (input already ordered, as in the benchmark): the $O(n)$ check skips the sort. With fixed d and the benchmark’s linear-size output, this gives $\Theta(nd + n + k\ell) = \Theta(n)$. The final equality depends on fixed d and $k\ell = O(n)$.
- **Worst case** (input unordered, a real sort required): $O(n \log n + nd + n + k\ell)$. With fixed d and $k\ell = O(n)$, this reduces to $O(n \log n)$.

These two bracket the engine’s realistic scaling. The ordered and shuffled scenarios of Section 6.5.6 measure them directly. One reading note: across the measured 100 K–2.22 M range, $\log_2 n$ grows only from about 17 to 21. The $O(n \log n)$ worst case therefore

looks almost straight in the plots—the two cases separate by a near-constant factor, not by a visible bend. Extra memory on this compiled path is $O(nd + n + k\ell)$, which reduces to $O(n)$ for the fixed benchmark and its linear-size output. Other modes can materialise more output or retain more search state.

Figure 6.1 draws the two cases.

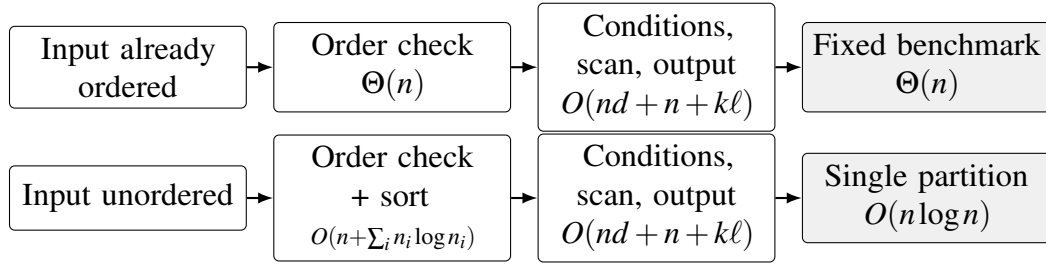


Figure 6.1: Data-size complexity for a fixed compiled row-local pattern.

For this fixed compiled path, ordering determines which branch applies.

Axis 2—growth with pattern complexity, for fixed data. A few constructs are expensive in the pattern size m rather than in the number of rows n . Each is either paid once or held in check by an explicit guard, so none appears as a growth curve in the data size.

The NFA-to-DFA conversion is $O(2^m)$ states in the worst case. But it is a one-time, cached compile step, capped at a state limit the engine derives from its own memory budget at roughly 64 KiB per state, with a conservative default ceiling of 50,000 states. The memory term is what tightens on a small machine; the ceiling is what holds on a large one, and on the evaluation host it is the ceiling that binds. Reaching this limit raises an explicit compilation error, and the engine does not execute or cache a partial DFA. None of the evaluated patterns came close.

A PERMUTE of m variables expands to $O(m!)$ branches, rejected above eight variables or an estimated fifty thousand branches. A DEFINE predicate that reads the match built so far cannot use the row-local Boolean matrix. Ambiguous quantified choices may then require the exact preference-ordered search, whose worst case is exponential. Non-default skip modes are different: they disable some fused enumeration and output paths, but do not by themselves make a predicate state-dependent or force exact search.

The exact search has a partition-level search budget (Section 5.2.1) and uses an explicit stack. It therefore avoids Python’s recursion limit. If the search budget is exhausted, the query raises an error instead of returning a partial result. The main benchmark and the ordering scenarios stay on the compiled row-local path and exercise

Axis 1. The state-dependent test in Section 6.5.6 uses a compiled incremental exact-search plan. It measures one important form, not the worst case of every state-dependent expression.

6.5 EMPIRICAL AND CROSS-SYSTEM ANALYSIS

This section compares the engine with Trino 473 and Oracle 21c EE on the Amazon UK Products dataset: five pattern families, six sizes (100 K to 2.22 M rows), 30 cells per system. Each cell is the IQR-filtered mean of twenty runs after five warmups. All 90 cells completed. The engine averaged about 6.1 M rows/s. Its per-family time fits give $R^2 = 0.9904\text{--}0.9997$, so growth is close to linear over these six sizes. Complex nested patterns cost the most. Section 6.5.1 defines the patterns and the resource policy, Section 6.5.2 the results, Sections 6.5.3 and 6.5.4 throughput and memory, and Section 6.5.5 the summary.

Positioning. The engine does not aim to replace Oracle, Trino, or streaming platforms. It brings a practical subset of row-pattern matching into in-memory Python and skips the load-into-a-server step. In return it supports only the subset of Chapters 4–5, while the databases have mature optimizers, distributed execution, and broader SQL coverage. Against hand-written Pandas, though, the SQL form states partitioning, ordering, pattern, skip, and measures in one query.

6.5.1 Pattern Definitions

Table 6.10: SQL Pattern Definitions and Detection Goals

Pattern Name	SQL Pattern	Description & Goal
simple_sequence	A+ B+	Goal: Detect transitions from budget to low-mid price listings. Use: Identify runs where budget products (price $\leq$ £10) are immediately followed by low-mid products (£10–25) in the listing order, useful for locating price-band boundaries and clustered cheap listings.

Pattern Name	SQL Pattern	Description & Goal
alternation	$A (B C)^+ D$	Goal: Detect budget-to-upper-mid transitions through mixed middle bands. Use: Find sequences that start with a budget product, continue with a mix of low-mid and mid products (£10–50), and end with an upper-mid product (£50–100). Exercises alternation inside a quantified group.
quantified	$A\{1,5\} B^* C^+$	Goal: Detect bounded budget runs followed by mid-band products. Use: Find sequences with 1–5 budget products, optionally followed by low-mid products, then one or more mid products (£25–50). Exercises bounded, optional, and unbounded quantifiers in one pattern.
optional_pattern	$A^+ B? C^*$	Goal: Flexible budget-run detection with optional band transitions. Use: Broad matching of budget runs with an optional low-mid product and optional mid products. High recall; produces the largest match sets in the benchmark.
complex_nested	$(A B)^+ (C\{1,3\} D^*)^+$	Goal: Nested grouping of low-price runs followed by mid/upper-mid groups. Use: Detect runs of low-priced products ($\text{price} \leq \text{£}25$) followed by one or more groups of 1–3 mid products with optional upper-mid products. Exercises nested alternation combined with bounded repetition; the most expensive pattern family.

All patterns run on a derived category column from the product price: A = budget ($\leq \text{£}10$), B = low-mid ($> \text{£}10$ and $\leq \text{£}25$), C = mid ($> \text{£}25$ and $\leq \text{£}50$), D = upper-mid ($> \text{£}50$ and $\leq \text{£}100$), and E = premium ($> \text{£}100$). Each row carries a sequential `seq_id` in source order, and the six sizes (100 K to 2,222,742 rows) are nested prefixes of the same source. Queries use `ORDER BY seq_id` and no `PARTITION BY`, so each dataset is one logical partition. The sequence is a controlled listing order, not a temporal one; what matters is that all three systems process the identical rows.

Benchmark setup and resource policy. The benchmark is 90 cells (three systems, five families, six sizes; 1,800 measured executions) on the same CSVs and query bodies.

The proposed-engine values describe the fixed benchmark revision recorded in Appendix A.1. The Oracle and Trino values were retained from an earlier campaign under the same one-core, outer 32 GB, five-warmup, twenty-run protocol. The systems ran sequentially, not at the same time. The engine ran in its own control group; Trino and Oracle ran in limited containers that were restarted before each size. The same control-group counter measured all three memory footprints. Reported times are client-observed wall-clock values and include result materialization, but exclude data loading. The database client’s memory is outside the measured containers, which is a limitation of the memory comparison. Every system returned the same normalized rows (Table A.1).

The engine’s throughput is stable across the range (a scan-dominated path) and it records the lowest time on all 30 cells. Memory uses the two unified metrics of Section 6.5.4: incremental peak (per-query working memory) and absolute peak (footprint).

6.5.2 Cross-System Comparison Under a Common Outer Resource Limit

The three systems run the same query bodies; they differ only in the source table. Appendix A.1 records the benchmark protocol and evidence used to reproduce the comparison. Match counts per cell are in Table A.1, identical across the three systems (all use ONE ROW PER MATCH). `optional_pattern` yields the most matches and `alternation` the fewest. Match volume feeds the output cost, but pattern structure drives most of the per-pattern variation below—`complex_nested` is the slowest family despite fewer matches than `simple_sequence`.

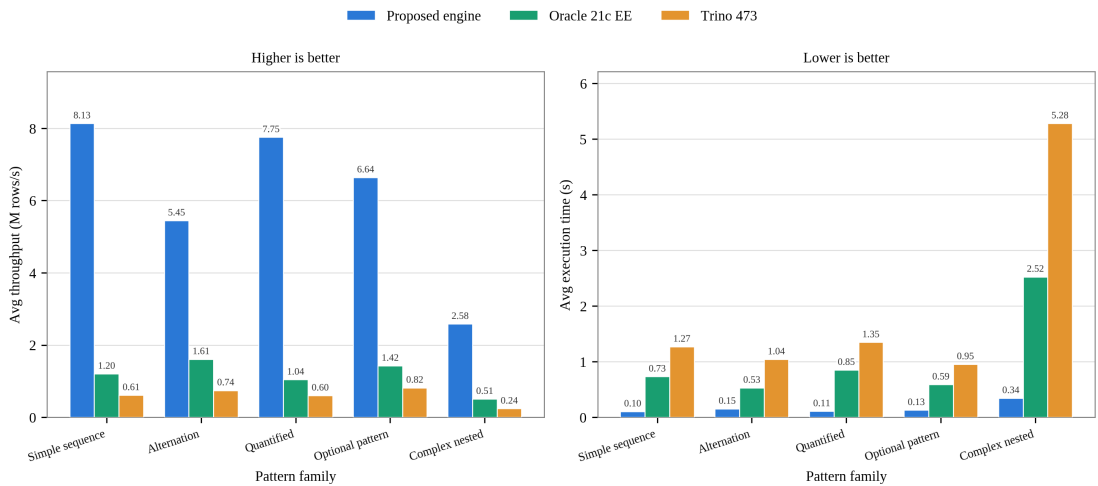


Figure 6.2: Performance summary by pattern family.

In Figure 6.2 each family is averaged over the six sizes: throughput on the left (higher is better), time on the right (lower is better). The engine leads both measures on all five families. The widest gap is `complex_nested`, where Trino averages 5.28 s against the engine’s 0.34 s—15.4 $\times$. The values are in Table A.2.

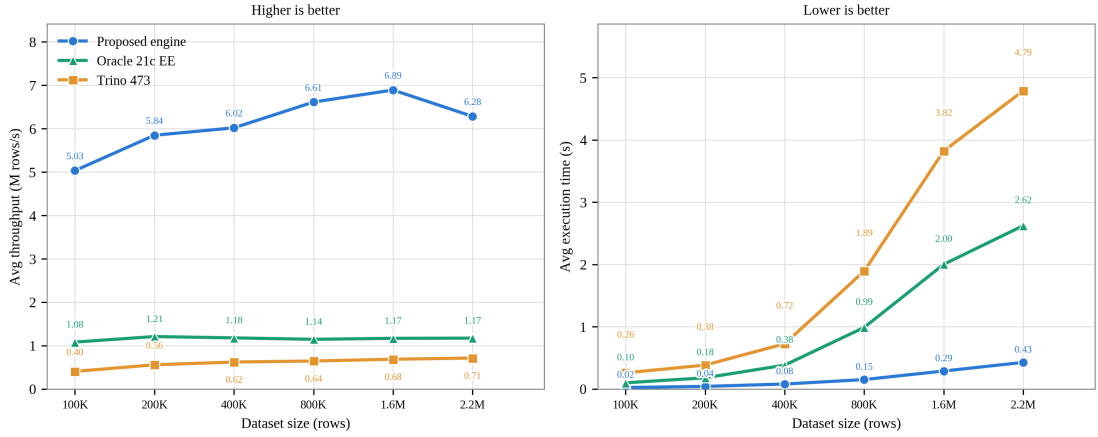


Figure 6.3: Performance summary by dataset size.

Figure 6.3 groups the same data by size. The engine is fastest at every size, holding 5.0–6.9 M rows/s as input grows. Per size it is about 9.3–13.3 $\times$ faster than Trino and 4.3–7.0 $\times$ faster than Oracle. The values are in Table A.3.

Figure 6.4 gives one time-versus-size panel per pattern. The engine (blue) is the lowest curve everywhere. The margin widens most on `complex_nested`, where the databases bend upward past 400 K rows while the engine stays near-linear (Table A.4).

6.5.3 Throughput by Pattern, Size, and System

Figure 6.5 plots throughput, one panel per pattern; higher is better. Throughput is computed for each measured run, then filtered and averaged. The engine is the highest curve in every panel (Table A.5).

6.5.4 Memory Metrics by Pattern, Size, and System

For the main benchmark, both memory metrics use the same `cgroup v2 memory.current` counter. It measures the dedicated engine control group or the complete Trino or Oracle container every 50 ms. The boundaries are therefore operational rather than identical: the engine value covers its Python process, while each database value covers its server container and excludes the external client. Shorter allocation spikes can also fall between samples.

CHAPTER 6. EXPERIMENTAL EVALUATION

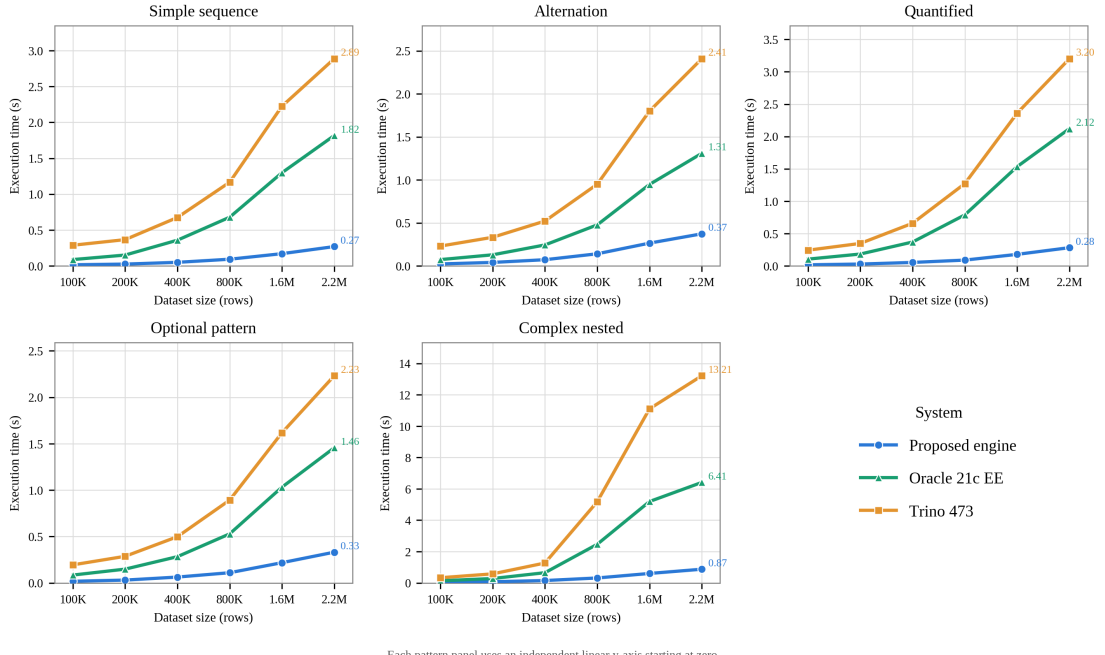


Figure 6.4: Execution time by pattern, size, and system.

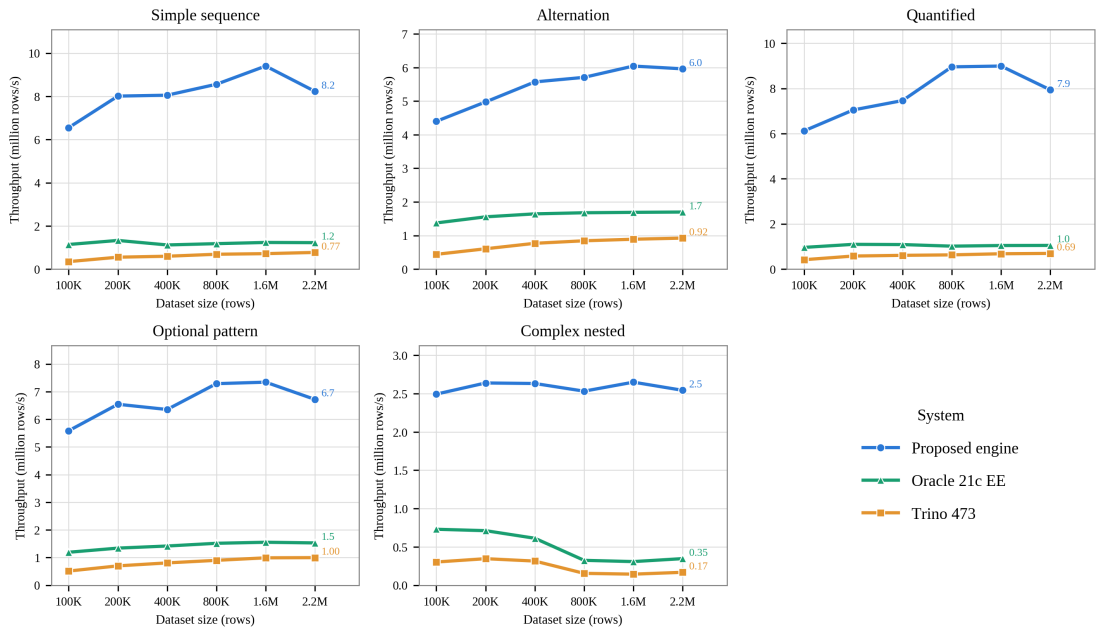


Figure 6.5: Throughput by pattern, size, and system.

The scripts divide byte counters by 1024^2 and 1024^3 . For compact labels, MB and GB in the memory results therefore mean the binary units MiB and GiB.

Each measured run has a five-second idle window. Its average defines the *baseline*. The *absolute peak* is the largest query-time sample, and the *incremental peak* is that sample minus the baseline. The subtraction removes most fixed resident memory, such as the already-loaded DataFrame, Oracle SGA, and Trino JVM heap. It does not turn the values into an exact algorithm-only measurement. As with time, each reported memory value is the IQR-filtered mean over twenty measured runs.

Query memory (Figure 6.6; Table A.6 in Appendix A) reports the incremental peak. *Footprint memory* (Figure 6.7; Table A.7) reports the absolute peak. The footprint documents the practical deployment cost of the resident engine or container, and is *not* a per-query or algorithm-level metric.

Two supporting values are recorded per cell in the benchmark evidence. The first is a *memory-per-million-rows* normalization—the incremental peak divided by the input size in millions, meaningful mainly at the larger sizes. The second is each database’s own per-query counter, Trino’s `peakMemoryBytes` and Oracle’s `session-PGA high-water delta`, which serve as cross-checks below.

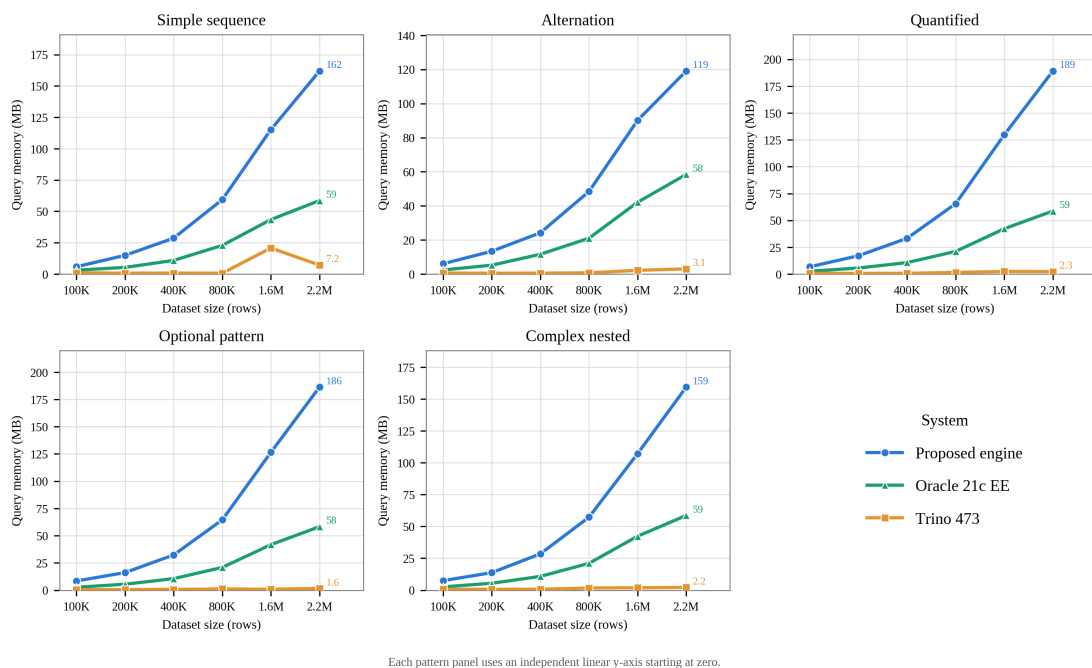
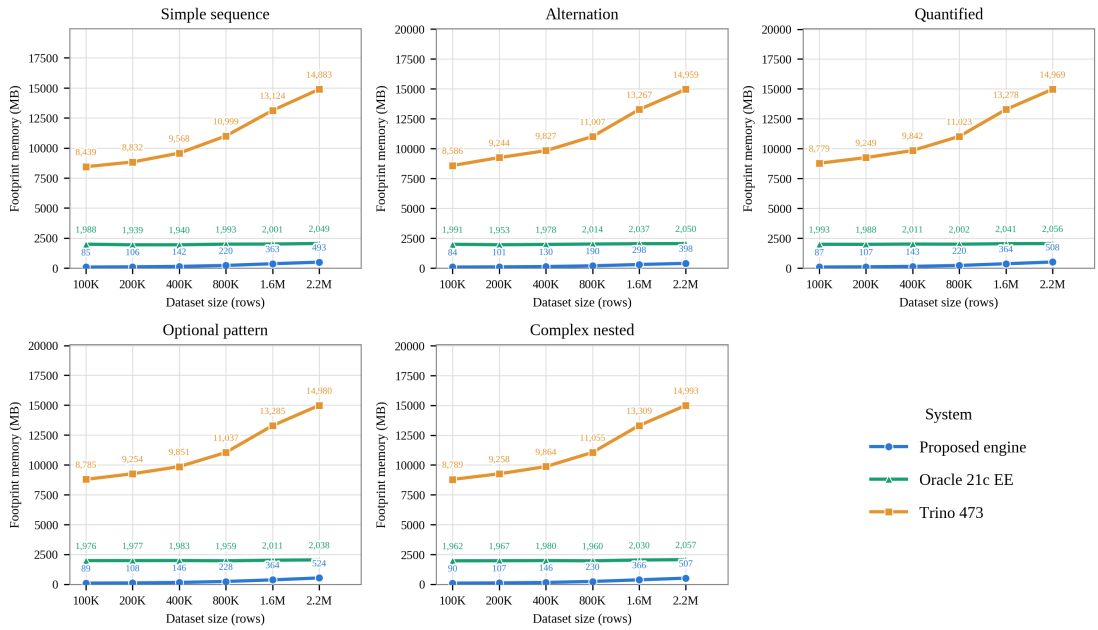


Figure 6.6: Query memory (incremental peak) by pattern, size, and system.

The engine’s and Oracle’s peaks rise roughly with size—the engine 5.9 MB to 189 MB, and Oracle 2.4 MB to 58.7 MB. The engine’s incremental peak is the largest of

the three (mean 64.5 MB versus Oracle’s 23.6 MB, about $2.7\times$). This is consistent with the compiled path creating whole-column masks and intermediate arrays.

One caveat for Trino is that the incremental cgroup measure observes growth above an already-resident JVM heap. It stays below 2.5 MB in 27 of the 30 cells, with a largest value of 20.8 MB. Trino’s native `peakMemoryBytes` counter ranges from 1.3 to 41.5 MB. The two counters have different scopes, so neither should be substituted for the common footprint comparison.



Each pattern panel uses an independent linear y-axis starting at zero.

Figure 6.7: Footprint memory (absolute peak) by pattern, size, and system.

These deployment footprints have different boundaries. The engine is one Python process and peaks at 0.51 GB. The database values cover whole containers: Trino uses 8.2–14.6 GB because the table resides in its JVM heap, while Oracle stays near 2.0 GB. Per-size restarts make the measurements independent across sizes. They show deployment cost; per-query comparisons use Table A.6.

6.5.5 Cross-System Summary

Figure 6.8 gives four bars per system: average time, average throughput, peak query memory, and peak footprint. The engine leads on time, throughput, and total footprint, and trades that for the highest per-query memory (Table A.8).

Averaged over the five patterns (Figure 6.9), the engine’s per-query memory rises with size. It stays the highest of the three (Table A.9).

CHAPTER 6. EXPERIMENTAL EVALUATION

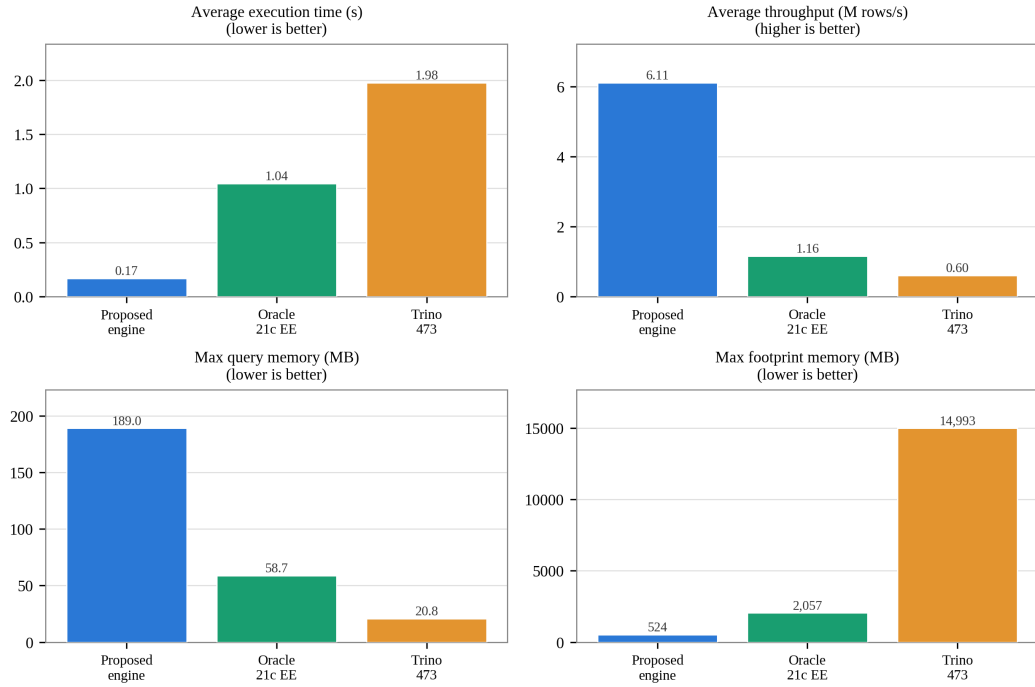


Figure 6.8: Overall benchmark statistics by system.

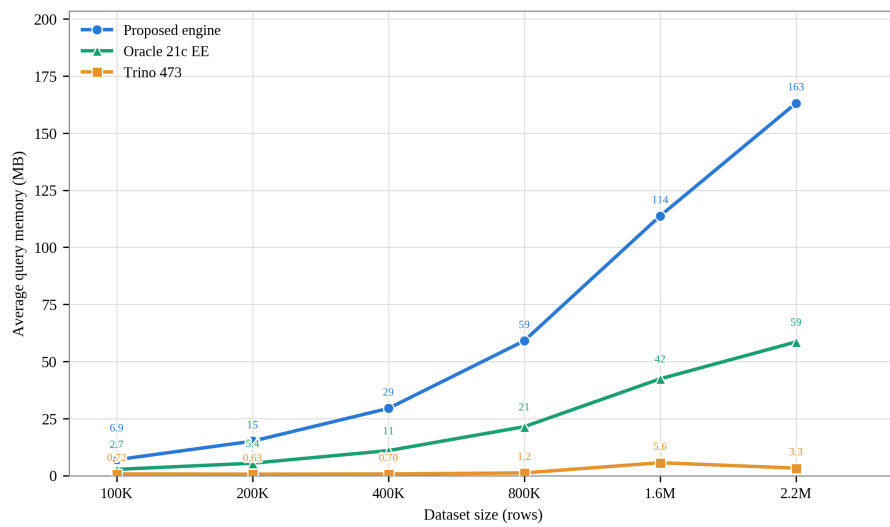


Figure 6.9: Average query memory by dataset size.

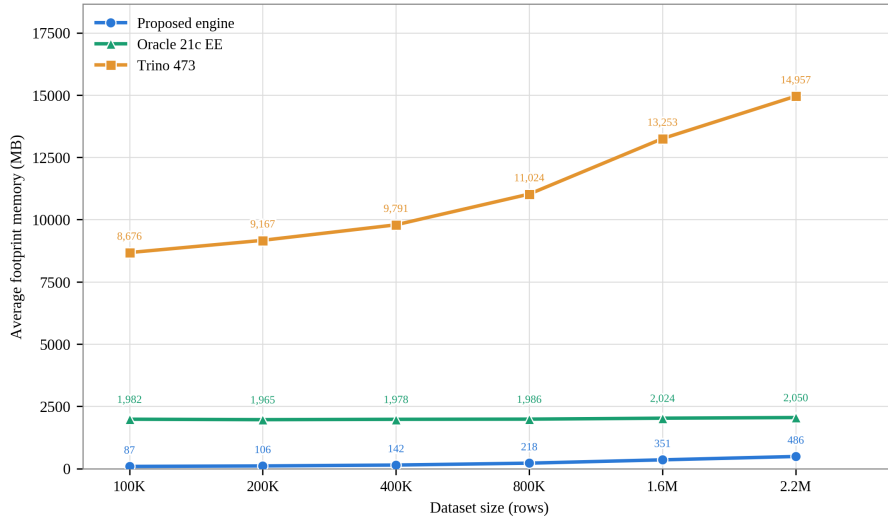


Figure 6.10: Average footprint memory by dataset size.

The footprint picture inverts (Figure 6.10). The engine’s single process stays well below the Trino and Oracle containers at every size (Table A.10).

All 90 system cells completed, and all 30 pattern–size combinations agreed across the three systems. The engine had the lowest mean time (0.17 s, against 1.04 s for Oracle and 1.98 s for Trino). It also had the highest incremental query peak but the lowest total footprint. Run-to-run variation remained low after warmup and IQR filtering. The detailed values are reported once in Appendix A. These results describe the selected workload and resource configuration; they do not establish universal superiority.

Figure 6.11 puts the two databases against the engine, drawn as a $1\times$ reference.

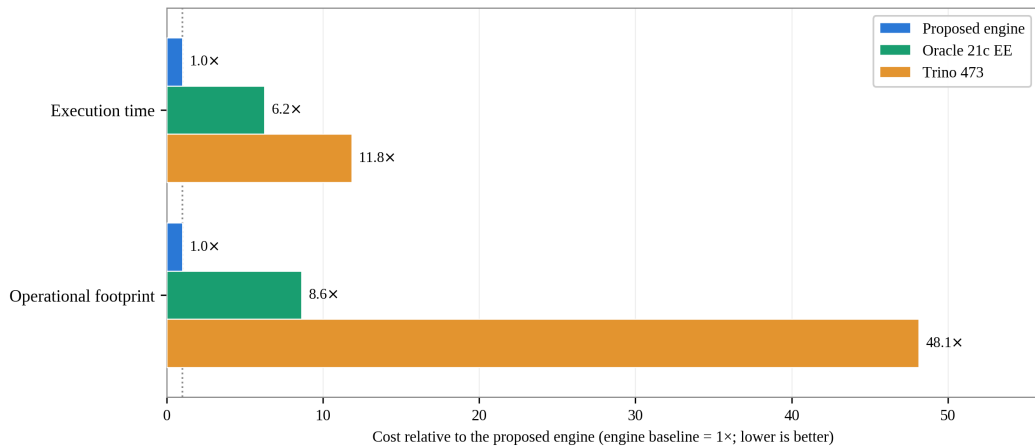


Figure 6.11: Cost of each database relative to the proposed engine (engine = $1\times$).

Each bar is a database’s average time or footprint as a multiple of the engine’s (blue $1\times$ line; lower is better). Oracle needs $6.2\times$ the time and $8.6\times$ the footprint; Trino $11.8\times$ and $48.1\times$. Throughput mirrors the time result. The engine’s one trade-off—higher incremental query memory, because the compiled path creates whole-column working arrays (about $2.7\times$ Oracle’s mean incremental peak)—is shown in Figure 6.6; signed percentages are in Table A.12.

6.5.6 Sensitivity and Boundary Analysis

The main matrix uses ordered input and row-local predicates. Two sensitivity cases remove these advantages: shuffled input and a state-dependent predicate. Both are measured across all three systems.

Sensitivity case 1: shuffled input. The benchmark orders by an already-monotonic `seq_id`, so the engine’s $O(n)$ check skips its sort. The matrix was rerun with every input deterministically shuffled, which makes the engine enter its $O(n \log n)$ sorting branch. The query still restores the same logical order. All 90 system cells completed, and all 30 pattern–size combinations still agreed across systems. Scenario A is ordered and Scenario B is shuffled. The difference measures sensitivity to physical input order; it is not a direct profile of the sort operator inside each database.

The proposed-engine values describe the benchmark revision recorded in Appendix A.1. The Oracle and Trino values were retained from the earlier unordered campaign under the same one-core, 32 GB, five-warmup, twenty-run protocol. As in the main benchmark, the systems ran sequentially.

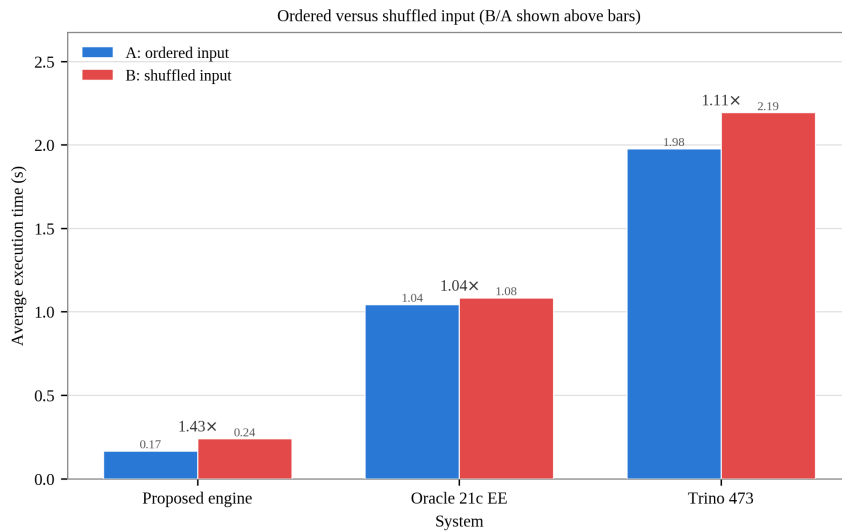


Figure 6.12: Average execution time on ordered vs shuffled input, by system.

Each pair is a system’s ordered (A) and shuffled (B) average; the number above is the B/A slowdown. Scenario A is the main matrix and Scenario B repeats it with every input shuffled. Each A/B comparison uses the same logical rows and query, but the changed physical layout can also affect caching and compression. The engine shows the largest slowdown ($1.43\times$): $1.70\times$ for `simple_sequence`, $1.69\times$ for `quantified`, and $1.19\times$ for `complex_nested`. Oracle and Trino change by 1.04 – $1.11\times$. Their internal plans were not profiled, so this smaller change is reported without assuming that they used the same sort plan in both scenarios. Most important, the ranking holds: at 2.22 M rows the engine is still about $4.5\times$ faster than Oracle and $8.8\times$ than Trino under shuffled input (against $6.2\times$ and $11.2\times$ ordered). So the advantage is not an artifact of the ordered shortcut—it survives when the engine must sort (Tables A.13–A.14).

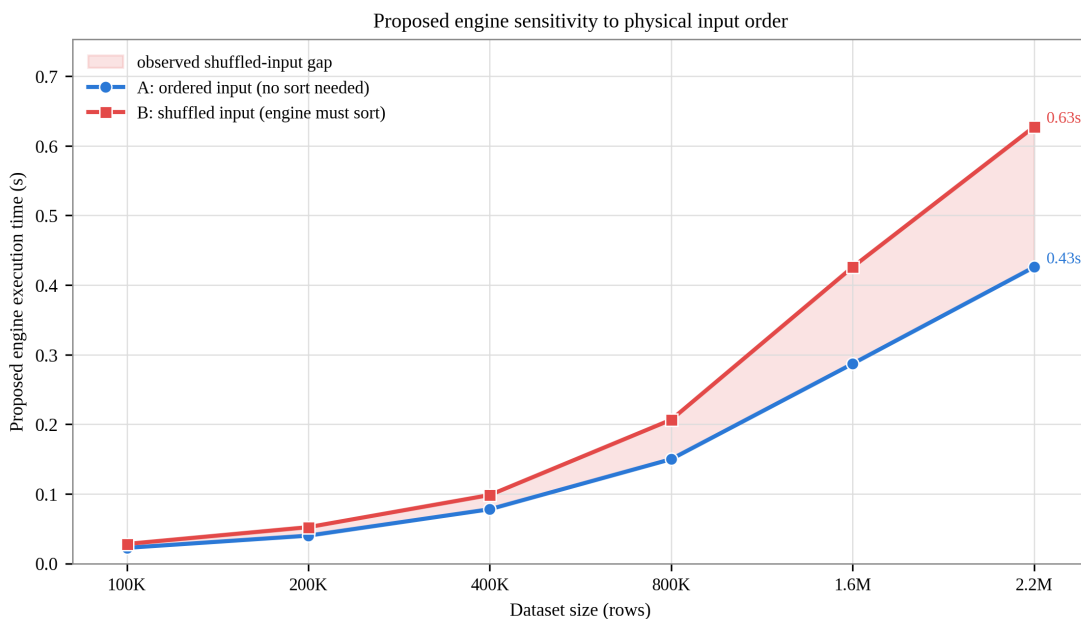


Figure 6.13: Engine time on ordered vs shuffled input, across sizes.

Figure 6.13 compares the engine on ordered (no sort) and shuffled (forced sort) input. Both grow close to linearly over these six sizes—exactly what the reading note of Section 6.4.2 predicts, since the log factor barely moves over this range. The shaded gap is the extra latency associated with shuffled input, including the engine’s sort. It is 0.20 s at 2.22 M rows (0.43 s ordered versus 0.63 s shuffled), and its normalization by $n \log_2 n$ stays within the same order of magnitude (Table A.15). Over this measured range, the shuffled-input penalty remains smaller than the total latency.

Sensitivity case 2: a state-dependent predicate. Shuffling changes ordering work but leaves the predicates row-local. A state-dependent predicate cannot use the same row-local Boolean matrix. To show this boundary, the `simple_sequence` query’s `DEFINE` becomes `B AS category='B' AND price > AVG(A.price)`—a running aggregate that depends on the match so far. Trino and Oracle evaluate it natively. All three produce identical output, so the semantic comparison is valid.

The repetition counts differ from the main matrix. The recorded engine values use five warmups and twenty measured runs per size. The retained Trino and Oracle values use one warmup and two measured runs per size. The systems ran sequentially. This test therefore shows the performance boundary, but its database means have less repetition than the main benchmark.

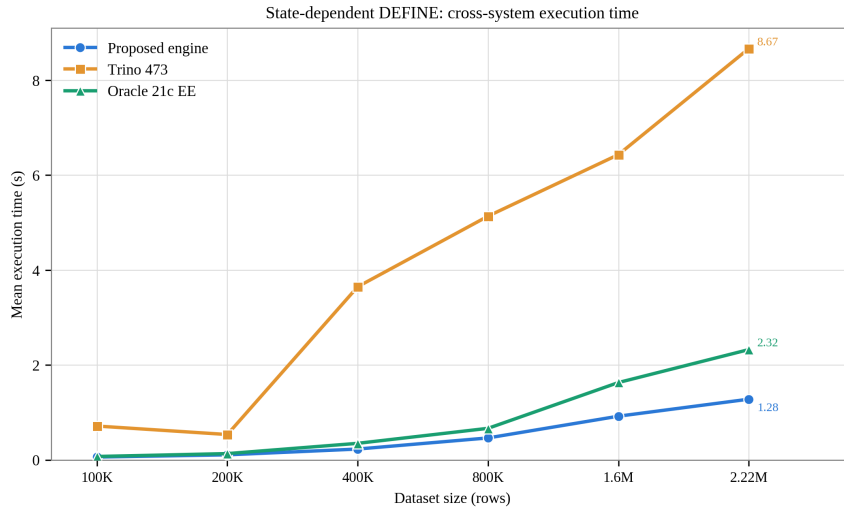


Figure 6.14: Execution time on a state-dependent *DEFINE* query, all systems.

An earlier recorded run reversed the ranking. The engine was the *slowest* system at every size—64.41 s at 2.22 M rows—because the generic exact evaluator repeatedly recomputed `AVG(A.price)` as the tentative match grew. The exact matcher now compiles this form into an incremental plan. It updates the A prefix state on append and rollback instead of recomputing the aggregate. This evaluated query is linear, but it is not a row-local vectorized predicate. It takes 1.28 s at 2.22 M rows, about 50× faster than the earlier 64.41 s result, with identical output (181,147 matches). The engine is now fastest here too—1.2–1.8× ahead of Oracle and 5.0–15.9× ahead of Trino (Table A.16).

This is one compiled incremental form, not a general solution for all state-dependent expressions. Supported forms outside the compiled plans still use the generic exact

evaluator, whose search can be exponential and can stop with an explicit resource-limit error.

The result is therefore scoped. The fixed compiled row-local benchmark is $\Theta(n)$ when ordered and $O(n \log n)$ when sorting is required. The tested running-AVG query is also linear after incremental compilation. No linear bound is claimed for generic exact search.

6.5.7 Scalability

Everything so far lives inside the range all three systems handle comfortably; this section asks how far each scales. The input is the real Amazon Reviews 2023 corpus. It uses nested prefixes of 5, 10, 20, 40, 80, 160, and 227.9 million rows—about $100\times$ the main dataset—with the five pattern families re-run on each. All three share one envelope at scale: one CPU core and 58 GB of RAM. Each reads the data in its own scalable mode. The engine uses RAM, Trino a disk-backed Hive/Parquet table, and Oracle 21c EE a native heap loaded by direct path. A query over the 600 s per-query budget is recorded as a wall; every completed database cell matched the engine’s output. Each stress-test cell uses two warmups and five measured runs. The proposed-engine values describe the benchmark revision recorded in Appendix A.1. The Oracle and Trino measurements were retained from the earlier campaign under the same one-core, 58 GB, two-warmup, five-run protocol. The systems ran sequentially. Figures 6.15 and 6.17 carry the results; full values are in Tables A.17–A.18 (Appendix A).

Time and throughput. Execution time is close to linear in n over the measured range. The per-pattern regressions give $R^2 = 0.9966$ – 0.9998 for the engine, 0.9824 – 0.9992 for Oracle, and 0.9912 – 0.9999 for Trino. Trino’s `complex_nested` fit uses its five completed sizes because the last two reached the time limit. These values quantify the observed trend; they do not replace the theoretical bounds in Section 6.4.2.

Left is execution time and right is absolute peak memory. Each size average uses the same four families completed by all three systems at all seven sizes. The engine is fastest on every completed cell—about 5.1 – $10.1\times$ ahead of Oracle 21c EE and 11.4 – $13.8\times$ ahead of Trino on the size averages. It also finishes the full corpus on every pattern: 27.8 s for `simple_sequence`, 106.0 s for `complex_nested`. The engine and Oracle 21c EE each complete 35/35 cells. Trino completes 33/35: `complex_nested` crosses the 600 s time budget at 160 M and 227,899,533 rows. Because Trino stops completing that family there, every size average in this section is taken over the four families all three systems finished at every size. Otherwise the composition of the mean would change along the axis.

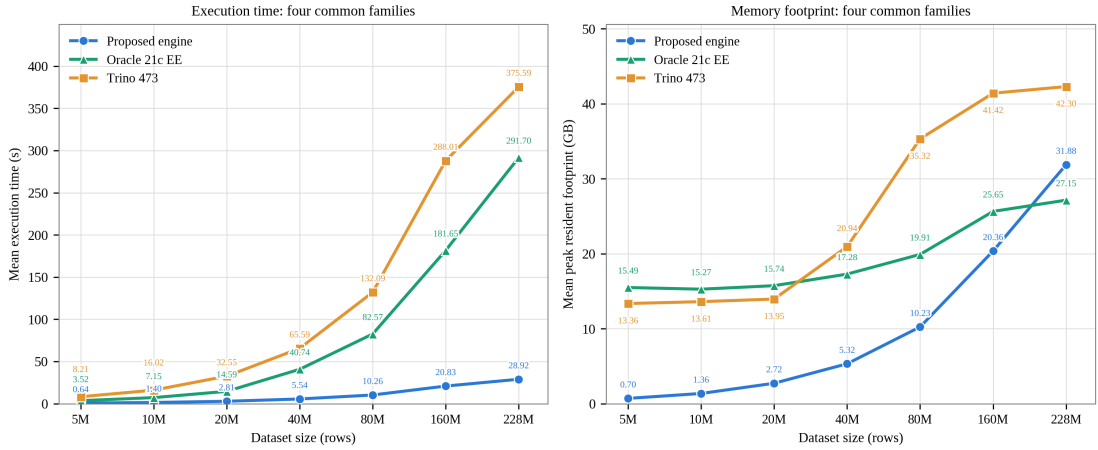


Figure 6.15: Volume scaling on real data: time (left) and peak memory (right).

Total resident memory tells a different story from the per-query figure below. The engine starts far leaner than either database—0.7 GB at 5 M rows against 13.4 GB for Trino and 15.5 GB for Oracle. The database footprints include their resident JVM or SGA baselines. The engine stays the leanest of the three up to 160 M. Only at the full corpus does its resident data pass Oracle: 31.9 GB against Oracle 21c EE’s 27.1 GB, still well under Trino’s 42.3 GB (Tables A.17–A.18).

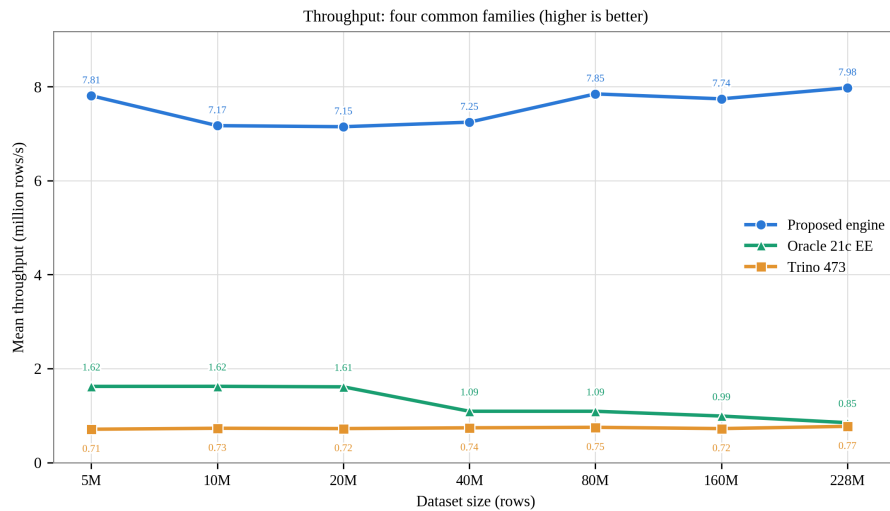


Figure 6.16: Throughput scaling on real data.

Figure 6.16 plots throughput, again averaged over the same four common families. The engine holds 7.1–8.0 M rows/s across the whole range. At the full size its patterns

range from 2.2 Mrows/s on `complex_nested` to 9.2 Mrows/s on `alternation`. Oracle 21c EE records 0.8–1.6 Mrows/s and 0.7–0.8 M for Trino. Oracle 21c EE drifts down as its fixed budgets spread over more data, while Trino stays near-flat, so the two converge at the full corpus. The per-pattern curves remain close to linear for all three (Figure A.1), and every completed cell returned identical match counts (Tables A.19, A.22).

Ranking. At the full corpus, the four-family average ranks the engine first, Oracle 21c EE second, and Trino third. The order is not the same for every pattern.

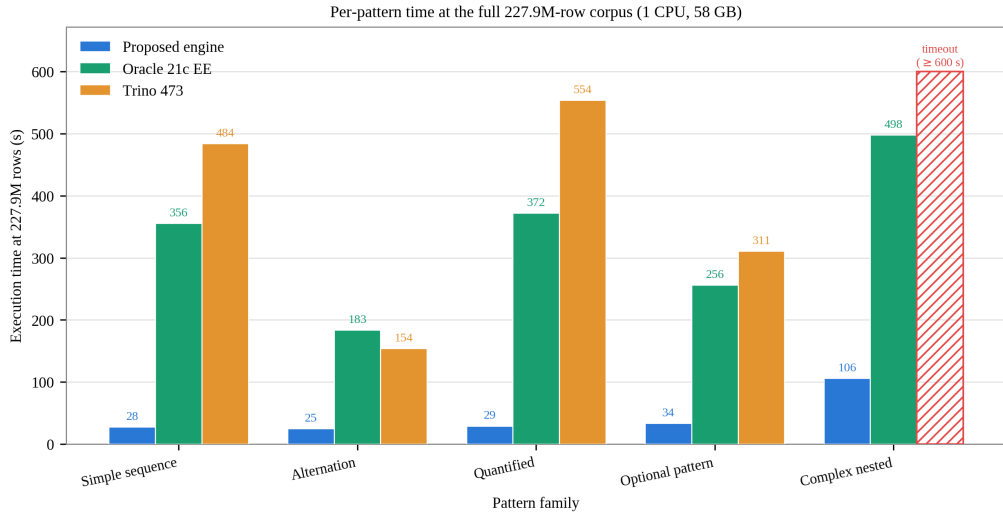
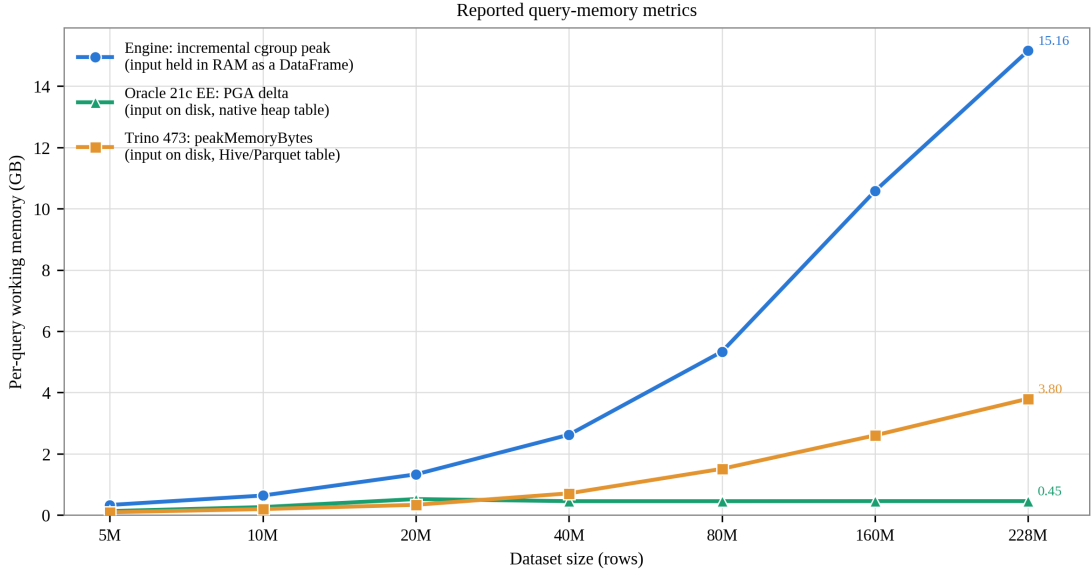


Figure 6.17: Per-pattern time at the full 227.9 M-row corpus.

The engine leads on all five patterns. Oracle 21c EE beats Trino on the three heavy families; for quantified, the times are 372 s and 554 s. Oracle is also the only database to finish `complex_nested`, at 498 s. Trino wins only on the cheapest family, `alternation`, at 154 s versus 183 s. It reaches the time wall on `complex_nested`. Thus the compiled engine leads, Oracle 21c EE is the more robust database in this stress test, and Trino is the first to reach the time wall (Table A.17).

Memory. The systems trade speed for memory in opposite directions.

Figure 6.18 is a diagnostic view, not a same-instrument comparison. It places the engine’s cgroup incremental peak beside the database-native counters (Oracle PGA and Trino `peakMemoryBytes`). Each series is averaged over the same four common families. The engine reaches about 15 GB on average and 18.5 GB on the largest single cell at 227.9 M rows. Oracle stays near 0.5 GB and Trino below 4 GB according to their own counters. These counters have different accounting boundaries, so their magnitudes



All three systems share one envelope: 1 CPU core and 58 GB of RAM. Metric definitions and storage modes differ by system; values are reported without a causal interpretation.

Figure 6.18: Diagnostic query-memory counters; the accounting boundaries differ.

should not be read as an exact memory ratio. The same-instrument deployment comparison is the absolute cgroup footprint in Figure 6.15 (Tables A.20–A.21). The largest observed engine footprint was 35.7 GB on one pattern, below the 58 GB limit. The experiment does not establish an exact maximum beyond this range.

6.5.8 Evaluation Validity

A cross-system comparison only means something if the advantage cannot be an artifact of the measurement. Six checks define how the results should be read.

Correctness first. Times are compared only where the systems return the same answer; every cell was checked for row-level equality (Table A.11). The 30/30 agreement is a correctness result and a precondition for comparing execution time.

Timer scope. Initial data loading is excluded for all systems. Query execution and result materialization are included, including database result transfer to the client. In the main matrix, five warmups reduce cold-start and JIT effects. The values are end-to-end query latencies, not isolated matcher-kernel times.

Stable main matrix. Each main-matrix cell is the IQR-filtered mean of twenty runs. The median coefficient of variation is 3.5% (engine), 2.3% (Oracle), and 3.8% (Trino). This supports the stability of the main-matrix means.

Narrow scope. The claim is only that the engine is faster *on this workload*—one partition, in-memory input, the supported subset—on one machine. The systems had

the same outer CPU and memory limits in the main matrix. Their execution models and measured process boundaries also differ. Database tuning could change the gap.

6.5.9 Key Findings

The main findings are as follows.

Speed. The engine was fastest on all 30 cross-system cells, and it stayed $5.1\text{--}10.1\times$ ahead of Oracle 21c EE and $11.4\text{--}13.8\times$ ahead of Trino on the four common stress-test families to 227.9 M rows.

Why it is faster on this workload. The engine is specialized for in-process DataFrame matching. It avoids database planning, catalog, transaction, client-protocol, and data-transfer costs. Its compiler turns supported predicates into column operations and integer-code comparisons, then uses decision tables and grouped NumPy reductions to reduce per-row Python work. Ordered input also lets it skip sorting, although Section 6.5.6 shows that it remains fastest when sorting is forced. This advantage applies to the supported single-node workload, not to every pattern or deployment.

Correctness. All 30 main pattern–size combinations matched across the three systems. Every completed stress database cell also matched the engine. The engine and Oracle completed 35/35 stress cells; Trino completed 33/35.

Scalability. Execution time was close to linear for all three systems over the tested range. The engine’s measured memory was also close to linear, and it finished the full 227.9 M-row corpus. It did not reach an internal size or memory limit; RAM remains the likely boundary for larger in-memory inputs. Oracle 21c EE also reached the full size; Trino’s hardest pattern exceeded the time budget at 160 M and 227.9 M rows.

The trade-off. The engine keeps its DataFrame in RAM and builds whole-column working arrays. The input is already part of the memory baseline, while the incremental peak measures additional query-time growth. Its total footprint is lower than the resident database containers in the main benchmark, but that relationship reverses against Oracle at the largest stress-test size.

Honest limits. The lead is bounded by what the compiler can turn into column operations. The state-dependent DEFINE that once made both databases faster now uses a compiled incremental plan. This solves one aggregate form, not the full class. Other state-dependent or ambiguous forms can still use the generic exact evaluator and may reach its explicit work limit. At scale, the resident DataFrame makes RAM a likely boundary. The study maps where each system’s usable region lies; it does not crown a universal winner.

Protocol. Across the aligned records, the systems used the same host, logical queries, input rows, CPU allocation, and outer memory limit. Measurements were collected

sequentially in separate campaigns. The main benchmark used a 32 GB limit and resident inputs. The stress study used a 58 GB limit and system-appropriate scalable storage.

6.6 CHAPTER SUMMARY

This chapter evaluated correctness, coverage, complexity, usability, performance, scalability, and memory. All three systems agreed in the 30 main pattern–size combinations, and every completed stress database cell matched the engine. The engine and Oracle completed all 35 stress cells; Trino completed 33. The engine recorded the lowest time on the evaluated workload and scaled close to linearly, while its query-time memory grew with the input. These findings apply to the supported single-node workload and stated resource limits. The next chapter summarizes the contributions, limitations, and future work.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

This chapter brings the findings together. It interprets the results, states the limits of the work, and identifies the next steps. Section 7.1 discusses the evidence. Section 7.2 maps the main claims to that evidence, and Section 7.3 defines their boundaries. Sections 7.4 and 7.5 give the final conclusion and future work.

7.1 DISCUSSION

The results show that a practical subset of SQL row-pattern recognition can run directly on pandas DataFrames. For the reported correctness queries, the engine agreed with the applicable Trino 473, Oracle, and hand-written pandas baselines. The cases covered partitioning, ordering, implicit DEFINE conditions, navigation, match numbering, and measure output. This supports the evaluated subset. It does not claim full SQL:2016 or vendor parity.

The usability comparison gives a second result. The three SQL examples used 14–16 lines of detection logic, compared with 24–32 lines for their hand-written pandas versions. Those functions also needed explicit grouping, shifting, candidate filtering, index tracking, and skip handling. The SQL form keeps this logic in one declarative statement. That lowers developer effort, makes the pattern easier to read, and travels between notebook analysis and SQL workflows—as long as the query stays inside the supported subset.

The performance benchmark contained 30 pattern–size combinations: five pattern families over six sizes from 100 K to the full 2.22 M rows. Each cell is the mean of twenty measured runs after five warmups and the stated outlier rule. Each system used the same input and query logic under one CPU core and the same outer 32 GB memory limit. The proposed engine averaged 6 110 392 rows/s and 0.17 s per cell. Oracle averaged 1.04 s and Trino 1.98 s. The proposed engine had the lowest time in all 30 cells, and all 30 normalized outputs agreed across the three systems. Pattern structure remained important: `complex_nested` was slower than the simpler families.

Memory shows the trade-off. Across the main benchmark, the maximum incremental query peaks were 189.05 MB for the engine, 58.71 MB for Oracle, and 20.75 MB for Trino. The maximum total footprints showed the reverse order: 523.97 MB, 2,057.17 MB, and 14,992.90 MB, respectively. All six values use the same cgroup-based metric, but the process boundaries differ: one engine process versus a complete database

container. Vendor-native counters have different scopes and are not treated as equivalent measurements.

This result has a clear boundary. The engine is specialized for one in-process DataFrame workload. It avoids database planning, storage, and client-transfer layers, and it compiles supported conditions to column operations. The evaluated running-average predicate is also updated incrementally. Supported expressions outside a compiled form use the general evaluator; unsupported forms are rejected. State-dependent conditions and some ambiguous pattern choices may require exact search. In the separate state-dependent test, the engine processed the full 2.22 M rows in 1.28 s and preserved the reference output. These results describe the tested subset on one machine. They are not a general ranking of SQL systems (Sections 6.5.9 and 6.5.6).

A separate scalability test used Amazon Reviews 2023 from 5 M to 227.9 M rows. Execution time was close to linear for all three systems. The engine’s measured memory was also close to linear. On the size averages, the engine remained $5.1\text{--}10.1\times$ faster than Oracle 21c EE and $11.4\text{--}13.8\times$ faster than Trino over the four pattern families completed by all three systems. Oracle 21c EE completed every cell. Trino’s `complex_nested` pattern reached the 600 s limit at the last two sizes. The engine completed the full corpus, but its maximum process footprint reached about 35.7 GB. Speed and memory residency must therefore be read together. The engine values describe the fixed benchmark revision recorded in Appendix A.1. The retained Oracle and Trino measurements came from separate campaigns under the same stated protocol. They were not run simultaneously.

The architecture separates parsing, validation, logical planning, automata construction, matching, and output assembly. The optimized paths use row-local masks, compact DFA and run-length plans, compiled measures, fused output, bounded caches, lazy row access, and conservative output projection. Partition processing remains sequential. The study therefore evaluates a single-node pandas engine, not a distributed SQL system. Trino and Oracle retain wider SQL coverage, mature optimizers, and production execution infrastructure.

7.2 VALIDATION OF CHAPTER CLAIMS

Table 7.1 lists the chapter’s main claims, the evidence behind each, and the boundary within which it holds. The purpose is to keep each conclusion within the evidence reported in the thesis.

Table 7.1: Validation mapping between thesis claims, supporting evidence, and scope boundaries

Claim area	Supporting evidence	Validated conclusion	Scope boundary
Correctness	Selected comparisons with Trino, Oracle, and hand-written pandas reported in Chapter 6.	The reported reference queries produce the same rows and measure values across the compared systems.	Selected supported scenarios only.
Feature support	The coverage tables and reference tests in Chapter 6, supported by the design in Chapters 4 and 5.	The system supports a practical row-pattern subset suitable for DataFrame-centered analysis.	This is not full SQL:2016 or vendor coverage.
Performance	The repeated 30-cell benchmark to 2.22 M rows and the separate scalability sweep to 227.9 M rows.	The engine has the lowest time on the evaluated cells and shows near-linear measured scaling.	One host, selected patterns, and different repetition protocols for the main and scalability studies.
Memory use	Incremental-peak and total-footprint measurements in both benchmarks.	The measurements show the memory cost of the evaluated workloads.	Empirical measurements, not a formal upper bound.
Developer effort	Lines of detection code in the three worked examples.	The SQL form uses fewer lines and removes manual traversal logic.	Code comparison only, not a user study.
Architecture	The five-stage design and its automated tests.	The implementation separates parsing, planning, compilation, and execution concerns.	The system remains a pandas-centered prototype.

Three validity limits remain. *Internal validity* depends on the reference outputs and their normalization. *Construct validity* depends on row equality, time, throughput, memory, and line count representing the claims made from them. *External validity* is limited by one workstation, two performance datasets, and the selected patterns. The main benchmark uses twenty measured runs per cell; the larger sweep uses five. Its recorded engine and retained database results were not collected in one simultaneous

rerun. Trino and Oracle were not expert-tuned, so other configurations may narrow the measured gaps.

7.3 LIMITATIONS

The following limitations define how far the results can be generalized.

Supported subset, not exhaustive SQL. The engine targets a substantial but bounded slice of `MATCH_RECOGNIZE: PARTITION BY, ORDER BY, PATTERN, DEFINE`, selected `MEASURES`, supported skip and output modes, navigation, subsets, exclusion blocks, anchors, and `PERMUTE`. It does not claim full SQL:2016 coverage or parity with Trino or Oracle. General ordering expressions, vendor-specific extensions, and other unsupported expression forms remain outside the validated scope.

Complex pattern and quantifier interactions. The automata design handles ordinary sequences, alternation, grouping, bounded repetition, and common quantifiers. Deeply nested expressions with several greedy or unbounded quantifiers can blow up the state and transition count, and `PERMUTE` is especially costly because its semantics are factorial. The engine therefore uses explicit NFA, DFA, and `PERMUTE` guards. If complete construction exceeds the state budget, compilation fails. It does not execute or cache an incomplete automaton. This preserves correctness, but a very large pattern may be rejected.

Predicates set a second limit. A `DEFINE` that cannot become column operations may need the exact search, which is the engine’s slowest path. The evaluated running `AVG` form is updated incrementally, but this does not cover every state-dependent expression. Exact search also has an explicit search budget. Its exhaustion is reported as an error rather than a changed result (Section 6.5.6).

Measure and aggregate scope. The measure evaluator supports common functions—`SUM`, `COUNT`, `MIN`, `MAX`, `AVG`, statistical and collection/string aggregates, conditional aggregates, and supported `RUNNING/FINAL` forms. Arbitrary user-defined aggregates and every possible expression combination remain out of scope—a natural limit of an in-memory engine with its own expression evaluator rather than a full SQL engine.

Performance boundaries. The core benchmark is predictable from 100 K to 2.22 M rows, and a separate sweep confirms near-linear execution time to 227.9 M rows. Measured engine memory also increased close to linearly on one workstation and one CPU core. Wider schemas, many small partitions, and highly ambiguous patterns were not tested at that scale. For large row-local scans, available RAM is the practical limit. Pattern compilation and exact search also have explicit algorithmic guards.

Memory behavior. Memory stayed within the configured limits, and query memory increased with input size. Small deltas still vary with allocator reuse, garbage collection,

result materialization, and the 50 ms sampling interval. The reported values are therefore empirical measurements, not a formal upper bound. The input DataFrame and result must remain resident, so the current engine is not an out-of-core system.

Optimizer integration. The engine runs independently of database optimizers. It does no cost-based plan selection, predicate pushdown into external storage, join reordering, or distributed scheduling, so some plan-level optimizations that databases offer are simply absent here.

7.4 CONCLUSION

Pandas has no native row-pattern primitive: detecting a multi-row pattern means shifted columns, manual loops, and custom state handling, or a round trip through an external SQL engine. This thesis addressed that gap with a SQL-in-pandas implementation that runs a supported subset of MATCH_RECOGNIZE over DataFrames.

The work makes three contributions. First, it provides an in-process Pandas engine with verified semantics for the evaluated R010-style subset. Second, it provides an optimized and guarded execution design for conditions, matching, measures, caching, output construction, and memory use. Third, it provides theoretical and empirical evaluation of correctness, coverage, usability, performance, and scalability.

The evidence supports three conclusions. The selected correctness scenarios agree with Trino, Oracle, and hand-written pandas. The declarative examples use less detection code than their procedural baselines. The main and scalability benchmarks show predictable growth under the stated protocols, with an increasing in-memory cost.

The approach is a bridge between SQL row-pattern recognition and DataFrame analytics. It does not replace mature SQL engines; it brings a useful slice of their row-pattern power into Python workflows where the data already lives in pandas. The conclusion holds within the limits stated in Section 7.3.

7.5 FUTURE WORK

Five directions would extend the work.

Broader SQL and dialect coverage. The most direct step is wider SQL:2016 and vendor coverage: more navigation combinations, broader expressions, richer aggregates, fuller RUNNING/FINAL semantics, and a larger conformance suite drawn from Trino, Oracle, and the standard.

Parallel and distributed execution. Independent partitions are a natural unit of parallel work. The current sequential partition loop could use independent workers with

separate matcher state while preserving deterministic output order. Distributed execution on Dask or Spark would also need a rule for a partition split across chunks.

Backend abstraction and columnar acceleration. The path is pandas-centered today. A backend abstraction could target Polars or Apache Arrow, pushing more predicate evaluation into compiled vectorized kernels and reducing Python iteration. An out-of-core backend could also reduce the resident-memory limit, but it would have to preserve partition order and matches across batch boundaries.

Optimization and diagnostics. The automata pipeline can gain from stronger state pruning, guard pushdown, better selectivity estimates, and a faster exact search for state-dependent forms not covered by compiled plans. This remains a performance boundary (Section 6.5.6). Better EXPLAIN output, match traces, and per-pattern counters would make each execution path easier to verify.

Usability and integration. Clearer error messages, notebook-friendly visualizations, reusable test fixtures, and integration with existing DataFrame query interfaces would make the engine easier to adopt in exploratory analysis and teaching.

7.6 CHAPTER SUMMARY

This chapter drew the findings together in a scoped context. It read the evaluation in practical terms: supported semantics, developer effort, performance, and memory. It also separated measured results from their limits.

The conclusion is that a practical subset of SQL row-pattern recognition can run directly over DataFrames. This is supported by the reported correctness and performance evidence, not by a claim of full SQL parity. Future work follows the remaining limits: wider conformance, parallel execution, alternative backends, exact-search optimization, and better diagnostics.

BIBLIOGRAPHY

- [1] D. Petković, “Identifying possible financial frauds using SQL row pattern recognition,” *International Journal of Computer Applications*, vol. 184, no. 35, pp. 31–34, Nov. 2022, ISSN: 0975-8887. DOI: 10.5120/ijca2022922446. [Online]. Available: <https://ijcaonline.org/archives/volume184/number35/32542-2022922446/>.
- [2] International Organization for Standardization, *ISO/IEC TR 19075-5:2016 information technology – database languages – sql technical reports – part 5: Row pattern recognition in sql, feature r010: Row pattern recognition in from clause*, <https://www.iso.org/standard/65143.html>, 2016.
- [3] International Organization for Standardization and International Electrotechnical Commission, *Information technology — guidance for the use of database language sql — part 5: Row pattern recognition*, <https://www.iso.org/standard/78936.html>, ISO/IEC 19075-5:2021, Edition 1, 2021.
- [4] Oracle Corporation, *Pattern matching in oracle database 12c*, https://docs.oracle.com/database/121/DWHSG/pattern_match.htm, Accessed: 2024, 2013.
- [5] Oracle, *Sql pattern matching in oracle database*, <https://docs.oracle.com/en/database/oracle/oracle-database/21/dwhsg/sql-pattern-matching-data-warehouses.html>, Oracle Database 21c Data Warehousing Guide, 2021. (visited on 06/15/2025).
- [6] K. Hansen, *Practical Oracle SQL: Mastering the Full Power of Oracle Database*. Apress Berkeley, CA, Jan. 2020, ISBN: 978-1-4842-5616-9. DOI: 10.1007/978-1-4842-5617-6.
- [7] Trino, *MATCH_RECOGNIZE*, <https://trino.io/docs/current/sql/match-recognize.html>, 2022.
- [8] Trino, *Row pattern recognition with MATCH_RECOGNIZE*, https://trino.io/blog/2021/05/19/row_pattern_matching.html, 2021.
- [9] Snowflake Inc., *MATCH_RECOGNIZE clause*, Snowflake Documentation, 2025. [Online]. Available: https://docs.snowflake.com/en/sql-reference/constructs/match_recognize (visited on 06/18/2025).

BIBLIOGRAPHY

- [10] Apache Flink, *MATCH_RECOGNIZE*, https://nightlies.apache.org/flink/flink-docs-stable/docs/dev/table/sql/queries/match_recognize/, 2026.
- [11] PostgreSQL Global Development Group, *Unsupported features*, PostgreSQL 18 documentation, 2025. [Online]. Available: <https://www.postgresql.org/docs/18/unsupported-features-sql-standard.html> (visited on 07/21/2026).
- [12] DuckDB Foundation, *Development roadmap*, 2026. [Online]. Available: <https://duckdb.org/roadmap> (visited on 07/21/2026).
- [13] Y. Cao, C.-Y. Chan, J. Li, and K.-L. Tan, *Optimization of analytic window functions*, 2012. arXiv: 1208.0086 [cs.DB]. [Online]. Available: <https://arxiv.org/abs/1208.0086>.
- [14] W. McKinney, “Data structures for statistical computing in Python,” in *Proceedings of the 9th Python in Science Conference*, 2010, pp. 56–61. DOI: 10.25080/Majora-92bf1922-00a.
- [15] J. Michels, K. Hare, K. Kulkarni, *et al.*, “The new and improved SQL: 2016 standard,” *ACM SIGMOD Record*, vol. 47, no. 2, pp. 51–60, 2018.
- [16] D. Petković, “Specification of row pattern recognition in the SQL standard and its implementations,” *Datenbank-Spektrum*, vol. 22, no. 2, pp. 163–174, Jul. 2022, ISSN: 1610-1995. DOI: 10.1007/s13222-022-00404-3.
- [17] asaniczka, *Amazon UK Products Dataset 2023*, Kaggle dataset, <https://www.kaggle.com/datasets/asaniczka/amazon-uk-products-dataset-2023> (accessed 2025-07-16), 2023.
- [18] Y. Hou, J. Li, Z. He, A. Yan, X. Chen, and J. McAuley, *Bridging Language and Items for Retrieval and Recommendation*, Amazon Reviews 2023 dataset, McAuley Lab, UC San Diego, <https://amazon-reviews-2023.github.io/> (accessed 2026-07-19), 2024.
- [19] MonierAshraf, *Row_match_recognize: Regression test suite*, GitHub repository, 2025. [Online]. Available: https://github.com/MonierLawande/Row_match_recognize/tree/master/tests.
- [20] M. Lawande, M. El-helw, and A. Awad, “Row pattern recognition for pandas: Bringing match_recognize to python dataframes,” in *2026 IEEE 23rd Mediterranean Electrotechnical Conference (MELECON)*, IEEE, 2026. DOI: 10.1109/MELECON64486.2026.11418885.

- [21] ISO/IEC, “ISO/IEC 9075-2:2016 information technology — database languages — SQL — part 2: Foundation (SQL/Foundation),” International Organization for Standardization, International Standard, 2016.
- [22] F. Hoffa, *Funnel analytics with SQL: MATCH_RECOGNIZE() on Snowflake*, Medium, Towards Data Science, Mar. 2021. [Online]. Available: <https://medium.com/data-science/funnel-analytics-with-sql-match-recognize-on-snowflake-8bd576d9b7b1> (visited on 06/15/2025).
- [23] M. Winand, *What’s new in SQL:2016*, Modern SQL: A lot has changed since SQL-92, Jun. 2017. [Online]. Available: <https://modern-sql.com/blog/2017-06/whats-new-in-sql-2016> (visited on 06/15/2025).
- [24] M. Winand, *match_recognize — regular expressions over rows*, Modern SQL: A lot has changed since SQL-92. [Online]. Available: https://modern-sql.com/feature/match_recognize (visited on 06/15/2025).
- [25] J. Melton and A. Eisenberg, “SQL:2016 support for row pattern recognition,” *SIGMOD Record*, vol. 46, no. 2, pp. 39–48, 2017. DOI: 10.1145/3105906.3105914.
- [26] F. Zemke, A. Witkowski, M. Cherniack, and L. Colby, “Pattern matching in sequences of rows,” ANSI Standard Proposal, Technical Report, 2007.
- [27] Oracle, *Pattern recognition with MATCH_RECOGNIZE*, https://docs.oracle.com/en/middleware/fusion-middleware/osa/19.1/cqlreference/pattern-recognition-match_recognize.html, 2022.
- [28] G. Howe, *Any interest in adding match_recognize?* PostgreSQL General Mailing List, Message-ID: 7e495cb8-ca03-4e3e-aa7e-45bf75212da8@Spark, 2020. [Online]. Available: <https://www.postgresql.org/message-id/7e495cb8-ca03-4e3e-aa7e-45bf75212da8@Spark> (visited on 06/18/2025).
- [29] T. Findling, *Bringing row pattern recognition to DuckDB: Translating MATCH_RECOGNIZE to WITH RECURSIVE*, Bachelor’s thesis, Eberhard Karls Universität Tübingen, 2025.
- [30] University of Tübingen, Database Systems Group, *Match_Recognize research project*, Database Systems Teaching Notes. [Online]. Available: <https://db.cs.uni-tuebingen.de/teaching/research-projects/match-recognize/> (visited on 07/07/2025).
- [31] University of Tübingen, Database Systems Group, *Research projects*, Database Systems Teaching Notes. [Online]. Available: <https://db.cs.uni-tuebingen.de/teaching/research-projects/> (visited on 07/07/2025).

- [32] L. Lambrecht, T. Findling, S. Heid, M. Knüdel, and T. Grust, “Democratize MATCH_RECOGNIZE!” *Proceedings of the VLDB Endowment*, vol. 18, no. 12, pp. 5251–5254, 2025. DOI: 10.14778/3750601.3750644.
- [33] Google Cloud, *MATCH_RECOGNIZE clause in GoogleSQL*, BigQuery documentation, 2025. [Online]. Available: https://cloud.google.com/bigquery/docs/reference/standard-sql/query-syntax#match_recognize_clause (visited on 07/21/2026).
- [34] Databricks, *Databricks SQL release notes 2026*, MATCH_RECOGNIZE beta in Databricks SQL and Runtime 19.0, 2026. [Online]. Available: <https://docs.databricks.com/gcp/en/sql/release-notes/2026#match-recognize-is-now-in-beta> (visited on 07/21/2026).
- [35] Š. Šprem, N. Tomažin, J. Matečić, and M. Horvat, “Building advanced web applications using data ingestion and data processing tools,” *Electronics*, vol. 13, no. 4, p. 709, 2024.
- [36] E. Georgiou, *Evaluating Apache Flink, Kafka Streams, and Spark Streaming for scalable machine learning workflows*, 2025.
- [37] M. Schule, H. Lang, M. Springer, A. Kemper, T. Neumann, and S. Gunnemann, “In-database machine learning with sql on gpus,” in *Proceedings of the 33rd International Conference on Scientific and Statistical Database Management*, ser. SSDBM ’21, New York, NY, USA: Association for Computing Machinery, 2021, pp. 25–36, ISBN: 9781450384131. DOI: 10.1145/3468791.3468840. [Online]. Available: <https://doi.org/10.1145/3468791.3468840>.
- [38] N. Makrynioti, R. Ley-Wild, and V. Vassalos, “Sql4ml: A declarative end-to-end workflow for machine learning,” *arXiv preprint arXiv:1907.12415*, 2019. [Online]. Available: <https://api.semanticscholar.org/CorpusID:198968138>.
- [39] D. Naphade, “The evolution and modernization of data pipeline architectures,” *European Journal of Computer Science and Information Technology*, vol. 13, no. 6, pp. 42–53, 2025.
- [40] M. Raasveldt and H. Mühleisen, “DuckDB: An embeddable analytical database,” in *Proceedings of the 2019 International Conference on Management of Data*, Amsterdam, Netherlands: Association for Computing Machinery, 2019, pp. 1981–1984. DOI: 10.1145/3299869.3320212.
- [41] DuckDB Foundation, *SQL on pandas*, DuckDB documentation, 2026. [Online]. Available: https://duckdb.org/docs/current/guides/python/sql_on_pandas (visited on 07/21/2026).

BIBLIOGRAPHY

- [42] T. Æ. Mogensen, *Introduction to Compiler Design*, 3rd. Springer Nature, 2024, pp. 105–113.
- [43] E. Zhu, S. Huang, and S. Chaudhuri, “High-performance row pattern recognition using joins,” *Proceedings of the VLDB Endowment*, vol. 16, no. 5, pp. 1181–1194, 2023, ISSN: 2150-8097. DOI: 10.14778/3579075.3579090.
- [44] J. Lucas, Y. Idris, B. Contreras-Rojas, J.-A. Quiane-Ruiz, and S. Chawla, “Rheem-studio: Cross-platform data analytics made easy,” in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018, pp. 1573–1576. DOI: 10.1109/ICDE.2018.00179.
- [45] A. Mozzillo, L. Zecchini, L. Gagliardelli, A. Aslam, S. Bergamaschi, and G. Simonini, *Evaluation of dataframe libraries for data preparation on a single machine*, 2025. DOI: 10.48786/EDBT.2025.27. [Online]. Available: <https://openproceedings.org/2025/conf/edbt/paper-96.pdf>.
- [46] S. Chawla, B. Contreras Rojas, Z. Kaoudi, S. Kruse, and J.-A. Quiané-Ruiz, “Building your cross-platform application with RHEEM,” *arXiv preprint arXiv:1805.11723*, 2018. DOI: 10.48550/arXiv.1805.11723. [Online]. Available: <https://arxiv.org/abs/1805.11723>.
- [47] The pandas development team, *pandas 2.3.3 API Reference*, Official pandas documentation, 2025. [Online]. Available: <https://pandas.pydata.org/pandas-docs/version/2.3/reference/index.html> (visited on 07/22/2026).
- [48] MonierAshraf, *Row_match_recognize*, GitHub repository, 2025. [Online]. Available: https://github.com/MonierLawande/Row_match_recognize/blob/master/Examples%20Match_reocginze/Examples.ipynb.
- [49] A. Albrecht and contributors, *Python-sqlparse documentation*, <https://sqlparse.readthedocs.io/en/latest/>, Accessed: 25 June 2026, 2026.
- [50] T. Mao and contributors, *SQLGlot: Python sql parser and transpiler*, <https://github.com/tobymao/sqlglot>, Accessed: 25 June 2026, 2026.
- [51] E. Klahn and contributors, *mo-sql-parsing: More sql parsing*, <https://github.com/klahnakoski/mo-sql-parsing>, Accessed: 25 June 2026, 2026.
- [52] L. Gaifax and contributors, *pglast: Postgresql languages ast and statements prettifier*, <https://pglast.readthedocs.io/en/latest/>, Accessed: 25 June 2026, 2026.
- [53] ANTLR Team and Contributors, *Trino sql grammar for antlr*, <https://github.com/antlr/grammars-v4/tree/master/sql/trino>, 2022.

BIBLIOGRAPHY

- [54] T. Parr, *The Definitive ANTLR 4 Reference*. Lewisville, TX, USA: Pragmatic Bookshelf, 2013, ISBN: 978-1-934356-99-9.
- [55] T. Parr, S. Harwell, and K. Fisher, “Adaptive LL(*) parsing: The power of dynamic analysis,” in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA ’14, ACM, 2014. DOI: 10.1145/2660193.2660202.
- [56] E. Robson and E. Freeman, *Entwurfsmuster von Kopf bis Fuß*. O’Reilly, 2022, pp. 614–615.
- [57] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd. Boston, MA, USA: Addison-Wesley, 2006, ISBN: 978-0-321-45536-9.
- [58] K. Thompson, “Programming techniques: Regular expression search algorithm,” *Communications of the ACM*, vol. 11, no. 6, pp. 419–422, 1968. DOI: 10.1145/363347.363387.
- [59] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd. Boston, MA, USA: Addison-Wesley, 2006, ISBN: 978-0-321-48681-3.
- [60] M. Ashraf, *Python udf example for row pattern recognition*, GitHub repository, 2025. [Online]. Available: https://github.com/MonierLawande/Row_match_recognize/blob/master/Python_Examples/python_udf.ipynb.
- [61] M. Ashraf, *Oracle MATCH_RECOGNIZE — stock price examples*, GitHub repository, 2025. [Online]. Available: https://github.com/MonierLawande/Row_match_recognize/blob/master/Examples%20Match_reocginze/Oracle_data_tables.sql.

CHAPTER A

DETAILED BENCHMARK TABLES

This appendix presents the detailed benchmark tables used in Chapter 6. The chapter explains the main results, while this appendix keeps the full values for verification. Unless stated otherwise, the tables use the Amazon UK Products benchmark and the cross-system protocol below.

A.1 MEASUREMENT PROVENANCE AND REPRODUCIBILITY

All systems ran sequentially, one at a time. No two systems ran together, so they never competed for the same CPU core or memory. Proposed-engine runs were refreshed on revision 179c286. Trino and Oracle values were retained from protocol-aligned earlier campaigns; the state-dependent repetition difference is disclosed below. The final regression run passed 709 tests.

Benchmark protocols. The evaluation used four fixed protocols:

- **Cross-system matrix.** One CPU core, an outer 32 GB memory limit, five warm-up runs, and twenty measured runs. The input was ordered, and the warm-up runs preceded measurement. The matrix contains five patterns and six sizes (100 K–2 222 742 rows), giving 30 cells per system. All 90 cells completed. The 30 proposed-engine cells are from the recorded engine campaign. The 60 database cells are retained results.
- **Stress test.** One CPU core, a 58 GB memory limit, two warm-up runs, and five measured runs. Each query had a 600 s limit. The test contains five patterns and seven sizes (5 M–227 899 533 rows). The proposed engine and Oracle completed 35 of 35 cells. Trino completed 33 of 35; `complex_nested` reached the time limit at 160 M and 227 899 533 rows.
- **Ordering test.** The matrix protocol was repeated after a deterministic shuffle. This removed physical pre-ordering while keeping the same logical `ORDER BY` key. All 90 shuffled cells completed.
- **State-dependent test.** One CPU core and a 32 GB memory limit were used with ordered input and `B AS category='B' AND price > AVG(A.price)`. The proposed engine used five warm-up runs and twenty measured runs per size. The retained Trino and Oracle reference values contain two measured runs per size.

Their means are useful for comparison, but provide less evidence about variation. All 18 cells completed.

Memory was sampled from the control group’s cgroup v2 counter every 50 ms. The incremental peak subtracts a five-second pre-query baseline. Timing includes query execution and result construction or retrieval. For the proposed engine, the absolute peak covers its process. For Trino and Oracle, it covers the full database container. Database-native query counters are reported separately where available. Data generation and loading are excluded. The scripts divide byte counters by 1024^2 and 1024^3 ; MB and GB in the memory tables therefore mean MiB and GiB.

The saved benchmark bundle contains the result tables and provenance records for the main matrix, ordering, state-dependent, and stress campaigns.

Measurement exclusions. No complete benchmark cell was omitted. The stated $1.5 \times$ IQR rule could remove individual samples from a cell. The number removed was retained with the measurements. A complete cell may be rejected only when there is recorded evidence of an invalid run, such as a wrong revision, incorrect resource limits, host contention, sampler failure, process failure, or incomplete output.

Known limitations. The engine is single-node and in-memory. Its practical limit depends on available RAM, input width, output size, and pattern structure. The executor selects among compiled row-local predicates, supported incremental aggregate paths, and general exact search. The selected path depends on both the expression and the pattern. The absence of a simple column operation does not by itself imply exact search.

A.2 MATCH COUNTS

Runtime and memory also depend on output size. Table A.1 therefore reports the returned rows under ONE ROW PER MATCH. The counts are identical across the three systems. In this dataset, optional_pattern produces the largest output and alternation the smallest.

Table A.1: Returned rows by pattern and size under ONE ROW PER MATCH

Size (rows)	simple_sequence	alternation	quantified	optional_pattern	complex_nested
100,000	9,067	1,828	5,643	15,247	9,420
200,000	16,093	3,692	10,064	27,172	17,759
400,000	34,391	5,803	19,012	52,950	32,360
800,000	66,330	9,782	32,804	99,214	58,616
1,600,000	130,338	20,276	67,358	197,253	119,299
2,222,742	181,147	28,776	95,650	276,776	167,641

A.3 PERFORMANCE SUMMARY BY PATTERN AND BY DATASET SIZE

Tables A.2 and A.3 summarize the same 30 cells. The first groups results by pattern. The second groups them by dataset size. Each displayed time is an arithmetic mean of cell mean times. Each displayed throughput is an arithmetic mean of cell throughputs; it is not recomputed from the displayed mean time.

Table A.2: Performance summary by pattern across the three systems

Pattern	Proposed engine		Trino 473		Oracle 21c EE	
	Thr. (rows/s)	Time (s)	Thr. (rows/s)	Time (s)	Thr. (rows/s)	Time (s)
simple_sequence	8,134,451	0.10	610,551	1.27	1,202,967	0.73
alternation	5,445,348	0.15	744,335	1.04	1,606,719	0.53
quantified	7,751,889	0.11	600,410	1.35	1,041,152	0.85
optional_pattern	6,637,948	0.13	816,795	0.95	1,424,079	0.59
complex_nested	2,582,321	0.34	238,402	5.28	505,612	2.52

Table A.3: Performance summary by dataset size across the three systems

Size (rows)	Proposed engine		Trino 473		Oracle 21c EE	
	Thr. (rows/s)	Time (s)	Thr. (rows/s)	Time (s)	Thr. (rows/s)	Time (s)
100,000	5,029,576	0.02	402,647	0.26	1,078,819	0.10
200,000	5,843,626	0.04	555,411	0.38	1,206,014	0.18
400,000	6,015,052	0.08	618,312	0.72	1,175,201	0.38
800,000	6,608,913	0.15	642,407	1.89	1,141,999	0.99
1,600,000	6,886,390	0.29	683,152	3.82	1,165,356	2.00
2,222,742	6,278,793	0.43	710,662	4.79	1,169,247	2.62

A.4 DETAILED CROSS-SYSTEM BENCHMARK MATRICES

This section gives the full values behind the benchmark figures in Chapter 6. Execution time includes result construction or retrieval. For each measured run, throughput is input size divided by that run’s time. Cell throughput is the arithmetic mean of the retained per-run rates after applying the same $1.5 \times$ IQR rule independently to throughput. It is not input size divided by the cell’s mean time. Query memory is the incremental cgroup peak above the five-second baseline. Footprint memory is the absolute cgroup peak. Time and memory are also filtered independently and reported as means of the retained samples from twenty measured runs.

Table A.4: Execution time by pattern, size, and system (seconds; mean $\pm$ sample standard deviation)

Pattern	Size (rows)	Proposed	Trino 473	Oracle 21c EE
simple_sequence	100,000	0.015 $\pm$ 0.001	0.288 $\pm$ 0.040	0.088 $\pm$ 0.004
simple_sequence	200,000	0.025 $\pm$ 0.001	0.366 $\pm$ 0.033	0.151 $\pm$ 0.004

APPENDIX A. DETAILED BENCHMARK TABLES

Table A.4 (continued): Execution time (s, mean $\pm$ std) by pattern, size, and system

Pattern	Size (rows)	Proposed	Trino 473	Oracle 21c EE
simple_sequence	400,000	0.049 $\pm$ 0.002	0.675 $\pm$ 0.040	0.358 $\pm$ 0.004
simple_sequence	800,000	0.093 $\pm$ 0.007	1.166 $\pm$ 0.029	0.678 $\pm$ 0.012
simple_sequence	1,600,000	0.170 $\pm$ 0.004	2.225 $\pm$ 0.144	1.298 $\pm$ 0.011
simple_sequence	2,222,742	0.269 $\pm$ 0.015	2.887 $\pm$ 0.117	1.816 $\pm$ 0.015
alternation	100,000	0.023 $\pm$ 0.001	0.230 $\pm$ 0.027	0.073 $\pm$ 0.006
alternation	200,000	0.040 $\pm$ 0.001	0.333 $\pm$ 0.026	0.129 $\pm$ 0.006
alternation	400,000	0.072 $\pm$ 0.001	0.521 $\pm$ 0.013	0.244 $\pm$ 0.009
alternation	800,000	0.141 $\pm$ 0.009	0.950 $\pm$ 0.016	0.477 $\pm$ 0.006
alternation	1,600,000	0.265 $\pm$ 0.005	1.800 $\pm$ 0.024	0.947 $\pm$ 0.011
alternation	2,222,742	0.373 $\pm$ 0.006	2.409 $\pm$ 0.015	1.309 $\pm$ 0.013
quantified	100,000	0.016 $\pm$ 0.001	0.242 $\pm$ 0.019	0.105 $\pm$ 0.006
quantified	200,000	0.028 $\pm$ 0.001	0.347 $\pm$ 0.015	0.183 $\pm$ 0.008
quantified	400,000	0.053 $\pm$ 0.004	0.659 $\pm$ 0.030	0.369 $\pm$ 0.009
quantified	800,000	0.089 $\pm$ 0.001	1.272 $\pm$ 0.032	0.788 $\pm$ 0.008
quantified	1,600,000	0.178 $\pm$ 0.004	2.360 $\pm$ 0.049	1.534 $\pm$ 0.017
quantified	2,222,742	0.281 $\pm$ 0.020	3.202 $\pm$ 0.060	2.121 $\pm$ 0.014
optional_pattern	100,000	0.018 $\pm$ 0.001	0.195 $\pm$ 0.011	0.085 $\pm$ 0.008
optional_pattern	200,000	0.031 $\pm$ 0.001	0.287 $\pm$ 0.010	0.149 $\pm$ 0.007
optional_pattern	400,000	0.063 $\pm$ 0.003	0.498 $\pm$ 0.026	0.283 $\pm$ 0.012
optional_pattern	800,000	0.110 $\pm$ 0.005	0.892 $\pm$ 0.046	0.528 $\pm$ 0.016
optional_pattern	1,600,000	0.218 $\pm$ 0.003	1.617 $\pm$ 0.019	1.031 $\pm$ 0.027
optional_pattern	2,222,742	0.331 $\pm$ 0.011	2.231 $\pm$ 0.035	1.455 $\pm$ 0.041
complex_nested	100,000	0.040 $\pm$ 0.001	0.332 $\pm$ 0.025	0.138 $\pm$ 0.010
complex_nested	200,000	0.076 $\pm$ 0.002	0.579 $\pm$ 0.027	0.281 $\pm$ 0.006
complex_nested	400,000	0.152 $\pm$ 0.005	1.271 $\pm$ 0.026	0.653 $\pm$ 0.010
complex_nested	800,000	0.317 $\pm$ 0.018	5.180 $\pm$ 0.177	2.466 $\pm$ 0.023
complex_nested	1,600,000	0.604 $\pm$ 0.013	11.108 $\pm$ 0.094	5.193 $\pm$ 0.032
complex_nested	2,222,742	0.875 $\pm$ 0.029	13.213 $\pm$ 0.050	6.411 $\pm$ 0.045

Table A.5: Throughput (rows/s) by pattern, dataset size, and system

Pattern	Size (rows)	Proposed	Trino 473	Oracle 21c EE
simple_sequence	100,000	6,548,744	345,160	1,139,312
simple_sequence	200,000	8,011,331	550,642	1,326,602
simple_sequence	400,000	8,052,822	594,845	1,117,414
simple_sequence	800,000	8,559,369	686,654	1,177,723
simple_sequence	1,600,000	9,403,104	714,822	1,232,833
simple_sequence	2,222,742	8,231,338	771,181	1,223,919
alternation	100,000	4,400,765	439,363	1,374,150
alternation	200,000	4,977,878	604,704	1,555,973
alternation	400,000	5,573,944	767,586	1,644,350
alternation	800,000	5,709,821	842,631	1,677,404
alternation	1,600,000	6,046,363	888,861	1,689,728
alternation	2,222,742	5,963,317	922,867	1,698,711
quantified	100,000	6,119,628	414,663	959,927
quantified	200,000	7,046,915	577,203	1,095,056
quantified	400,000	7,462,262	608,481	1,085,087
quantified	800,000	8,955,091	629,296	1,015,188
quantified	1,600,000	8,990,669	678,342	1,043,489

APPENDIX A. DETAILED BENCHMARK TABLES

Table A.5 (continued): Throughput (rows/s) by pattern, size, and system

Pattern	Size (rows)	Proposed	Trino 473	Oracle 21c EE
quantified	2,222,742	7,936,771	694,474	1,048,168
optional_pattern	100,000	5,584,402	511,386	1,190,721
optional_pattern	200,000	6,542,233	698,482	1,340,712
optional_pattern	400,000	6,353,474	805,792	1,416,523
optional_pattern	800,000	7,287,498	898,860	1,515,186
optional_pattern	1,600,000	7,341,539	989,684	1,552,628
optional_pattern	2,222,742	6,718,543	996,563	1,528,702
complex_nested	100,000	2,494,341	302,662	729,985
complex_nested	200,000	2,639,771	346,023	711,728
complex_nested	400,000	2,632,758	314,855	612,631
complex_nested	800,000	2,532,787	154,594	324,492
complex_nested	1,600,000	2,650,277	144,049	308,102
complex_nested	2,222,742	2,543,994	168,226	346,736

Table A.6: Incremental query memory by pattern, size, and system (MB)

Pattern	Size (rows)	Proposed	Trino 473	Oracle 21c EE
simple_sequence	100,000	5.87	1.05	3.14
simple_sequence	200,000	14.94	0.73	5.32
simple_sequence	400,000	28.68	0.79	10.91
simple_sequence	800,000	59.37	0.71	22.88
simple_sequence	1,600,000	115.20	20.75	43.45
simple_sequence	2,222,742	161.89	7.20	58.65
alternation	100,000	6.06	0.64	2.42
alternation	200,000	13.38	0.49	5.22
alternation	400,000	24.22	0.56	11.62
alternation	800,000	48.40	0.73	21.03
alternation	1,600,000	90.12	2.20	42.12
alternation	2,222,742	119.05	3.05	58.41
quantified	100,000	6.83	0.68	2.65
quantified	200,000	17.07	0.68	5.65
quantified	400,000	33.31	0.73	10.69
quantified	800,000	65.55	1.55	21.13
quantified	1,600,000	129.68	2.42	42.21
quantified	2,222,742	189.05	2.33	58.71
optional_pattern	100,000	8.45	0.53	2.72
optional_pattern	200,000	16.17	0.57	5.49
optional_pattern	400,000	32.35	0.71	10.62
optional_pattern	800,000	64.51	1.12	20.88
optional_pattern	1,600,000	126.43	0.86	41.76
optional_pattern	2,222,742	186.24	1.59	58.26
complex_nested	100,000	7.50	0.71	2.63
complex_nested	200,000	13.77	0.70	5.42
complex_nested	400,000	28.52	0.71	10.84
complex_nested	800,000	57.29	1.75	21.07
complex_nested	1,600,000	107.09	1.92	42.33
complex_nested	2,222,742	159.30	2.17	58.61

APPENDIX A. DETAILED BENCHMARK TABLES

Table A.7: Absolute footprint memory by pattern, size, and system (MB)

Pattern	Size (rows)	Proposed	Trino 473	Oracle 21c EE
simple_sequence	100,000	84.69	8,439.12	1,988.23
simple_sequence	200,000	105.82	8,832.13	1,938.77
simple_sequence	400,000	142.06	9,567.85	1,939.79
simple_sequence	800,000	219.85	10,999.08	1,993.00
simple_sequence	1,600,000	363.39	13,124.03	2,000.77
simple_sequence	2,222,742	492.84	14,882.81	2,048.95
alternation	100,000	84.35	8,585.77	1,991.35
alternation	200,000	101.29	9,243.71	1,953.46
alternation	400,000	129.50	9,826.87	1,978.45
alternation	800,000	190.33	11,007.22	2,013.57
alternation	1,600,000	298.20	13,266.95	2,037.13
alternation	2,222,742	397.95	14,958.94	2,049.51
quantified	100,000	86.84	8,778.67	1,992.95
quantified	200,000	107.49	9,248.96	1,988.29
quantified	400,000	143.42	9,842.49	2,011.17
quantified	800,000	219.66	11,023.44	2,001.70
quantified	1,600,000	363.83	13,277.69	2,040.85
quantified	2,222,742	507.62	14,969.31	2,055.91
optional_pattern	100,000	89.03	8,785.11	1,975.58
optional_pattern	200,000	107.90	9,253.51	1,976.71
optional_pattern	400,000	146.19	9,851.34	1,983.32
optional_pattern	800,000	228.40	11,037.41	1,959.17
optional_pattern	1,600,000	363.50	13,284.68	2,010.72
optional_pattern	2,222,742	523.97	14,980.42	2,037.57
complex_nested	100,000	89.84	8,789.27	1,961.78
complex_nested	200,000	107.47	9,257.82	1,966.53
complex_nested	400,000	146.49	9,864.07	1,979.70
complex_nested	800,000	229.89	11,054.94	1,960.23
complex_nested	1,600,000	366.45	13,309.19	2,030.26
complex_nested	2,222,742	507.50	14,992.90	2,057.17

The engine footprint is its process peak. The Trino and Oracle footprints cover their full containers. This difference matters when reading the absolute values in Tables A.8 and A.10.

Table A.8: Overall benchmark statistics by system

System	Cells	Cell-Mean Time Total (s)	Avg Time (s)	Avg Thr. (rows/s)	Thr. Range (rows/s)	Max Query Mem. (MB)	Max Footpr. Mem. (MB)
Proposed engine	30	5.02	0.17	6,110,392	2,494,341–9,403,104	189.05	523.97
Trino 473	30	59.33	1.98	602,098	144,049–996,563	20.75	14,992.90
Oracle 21c EE	30	31.34	1.04	1,156,106	308,102–1,698,711	58.71	2,057.17

The cell-mean time total adds the 30 cell mean times; it is not the duration of one workload run. Average time is the arithmetic mean of those 30 cell means. Average throughput is the arithmetic mean of the 30 cell throughputs defined above.

APPENDIX A. DETAILED BENCHMARK TABLES

Table A.9: Average query memory by dataset size, all three systems

Size (rows)	Proposed (MB)	Trino 473 (MB)	Oracle 21c EE (MB)
100,000	6.94	0.72	2.71
200,000	15.07	0.63	5.42
400,000	29.42	0.70	10.93
800,000	59.02	1.17	21.40
1,600,000	113.70	5.63	42.37
2,222,742	163.11	3.27	58.53

Table A.10: Average footprint memory by dataset size, all three systems

Size (rows)	Proposed (MB)	Trino 473 (MB)	Oracle 21c EE (MB)
100,000	86.95	8,675.59	1,981.98
200,000	106.00	9,167.22	1,964.75
400,000	141.53	9,790.52	1,978.49
800,000	217.63	11,024.42	1,985.53
1,600,000	351.08	13,252.51	2,023.95
2,222,742	485.97	14,956.88	2,049.82

Table A.11: Cross-system correctness by pattern across six dataset sizes

Pattern	Proposed engine	Trino 473	Oracle 21c EE
simple_sequence	reference result	6/6 identical	6/6 identical
alternation	reference result	6/6 identical	6/6 identical
quantified	reference result	6/6 identical	6/6 identical
optional_pattern	reference result	6/6 identical	6/6 identical
complex_nested	reference result	6/6 identical	6/6 identical

For each metric, the relative value is the ratio of the two arithmetic means across the 30 cells:

$$100 \left(\frac{\bar{x}_{\text{proposed}}}{\bar{x}_{\text{reference}}} - 1 \right).$$

It is not the arithmetic mean of 30 cell-wise percentage differences. Lower execution time and memory are better. Higher throughput is better.

Table A.12: Relative difference between the proposed engine and each SQL engine

Metric	vs. Trino 473	vs. Oracle 21c EE
Execution time	−92%	−84%
Throughput	+915%	+429%
Query memory	+3093%	+174%
Operational footprint	−98%	−88%

A.5 ORDERED VS. SHUFFLED INPUT (SORTING SCENARIOS)

These tables support the ordering analysis in Section 6.5.6. Scenario A uses input that is already ordered by `seq_id`. Scenario B shuffles the physical rows before the same logical ordering request. The logical order and match counts remain unchanged in all 90 system–pattern–size cells.

Table A.13: Average execution time for ordered and shuffled input

System	A ordered (s)	B shuffled (s)	Slowdown
Proposed engine	0.167	0.240	1.43×
Trino 473	1.978	2.193	1.11×
Oracle 21c EE	1.045	1.083	1.04×

Table A.14: Execution time and slowdown by size for ordered (“ord”) and shuffled (“shuf”) input

Size (rows)	Proposed engine			Trino 473			Oracle 21c EE		
	ord	shuf	slow	ord	shuf	slow	ord	shuf	slow
100,000	0.022	0.028	1.25×	0.257	0.283	1.10×	0.098	0.086	0.88×
200,000	0.040	0.052	1.30×	0.382	0.402	1.05×	0.178	0.166	0.93×
400,000	0.078	0.098	1.26×	0.725	0.775	1.07×	0.381	0.377	0.99×
800,000	0.150	0.207	1.38×	1.892	1.973	1.04×	0.987	0.986	1.00×
1,600,000	0.287	0.426	1.48×	3.822	4.180	1.09×	2.001	2.077	1.04×
2,222,742	0.426	0.627	1.47×	4.788	5.546	1.16×	2.622	2.810	1.07×

Table A.15: Observed ordered–shuffled time difference for the proposed engine by dataset size

Size (rows)	A (s)	B (s)	Δt (ms)	$\Delta t/n$ (ns)	$\Delta t/(n \log_2 n)$ (ns)
100,000	0.022	0.028	5.6	56.0	3.37
200,000	0.040	0.052	12.2	61.0	3.46
400,000	0.078	0.098	20.5	51.2	2.75
800,000	0.150	0.207	56.6	70.8	3.61
1,600,000	0.287	0.426	138.9	86.8	4.21
2,222,742	0.426	0.627	201.5	90.7	4.30

The difference between Scenarios A and B estimates the extra cost associated with shuffled input. Ordering is the main added operation in the proposed engine, but cache and row-layout effects may also contribute. The difference per row rises slightly, while the value divided by $n \log_2 n$ stays in a narrow range. This is consistent with comparison-based sorting over the tested sizes; it does not prove the asymptotic bound. The Oracle value below one at 100 K rows is within small-run variation and does not show that shuffling improves execution.

Table A.16: Mean execution time for the state-dependent aggregate pattern

Size (rows)	Proposed engine (s)	Trino 473 (s)	Oracle 21c EE (s)
100,000	0.060	0.713	0.073
200,000	0.107	0.535	0.133
400,000	0.229	3.649	0.349
800,000	0.463	5.136	0.666
1,600,000	0.921	6.432	1.630
2,222,742	1.279	8.665	2.324

This pattern uses `price > AVG(A.price)` in `DEFINE`. For this supported form, the engine evaluates the running aggregate incrementally in one scan. Ordered input lets it skip sorting after checking the order. All three systems return the same output.

A.6 STRESS-TEST RESULTS

The stress test in Section 6.5.7 uses real Amazon Reviews 2023 data from 5 M to 227.9 M rows. Within this campaign, the three systems use equivalent queries and logical rows. The storage representations also differ. The proposed engine holds a DataFrame in RAM. Trino scans a Hive/Parquet table. Oracle 21c EE scans a native table. Loading time is excluded. Every completed database result was checked against the engine output.

APPENDIX A. DETAILED BENCHMARK TABLES

The recorded proposed-engine measurements and retained database measurements were collected in separate campaigns under the same stress-test protocol. In total, 103 of 105 cells completed: 35 proposed-engine cells, 35 Oracle cells, and 33 Trino cells. The two incomplete Trino cells reached the stated time limit.

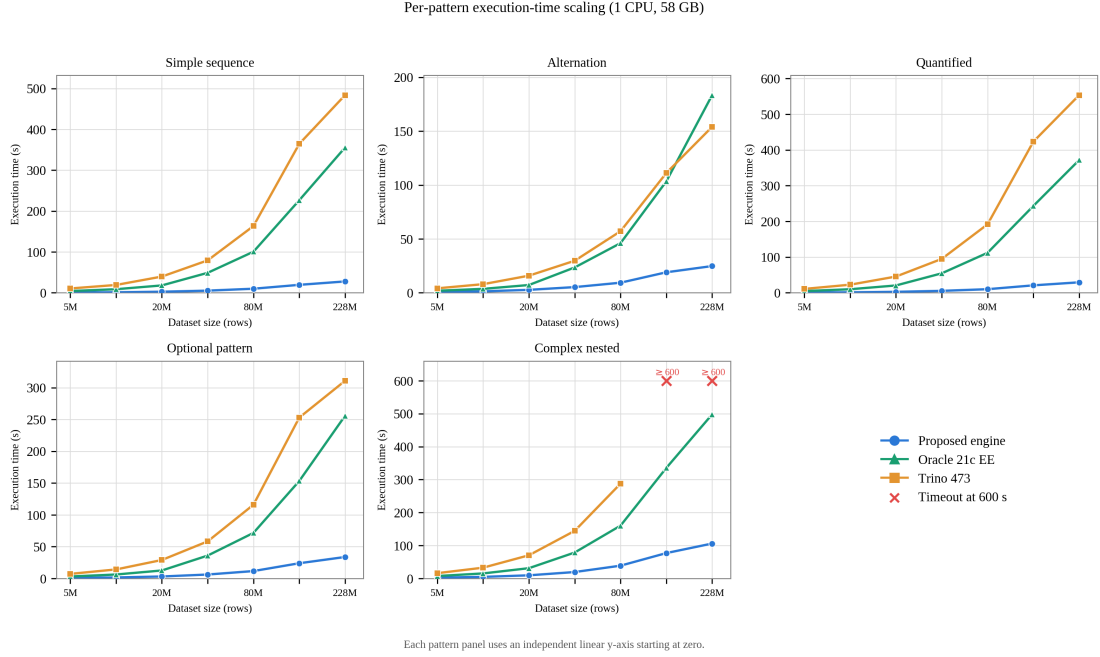


Figure A.1: Execution-time scaling by pattern under one CPU core and a 58 GB memory limit. Red × symbols mark Trino runs that reached the 600-second limit.

Execution time is reported as the mean and sample standard deviation of retained samples from five measured runs. Stress-test throughput is the arithmetic mean of the retained per-run rates, using the same independent $1.5\times$ IQR rule. *wall* means that the query reached the 600 s limit. Absolute memory includes the idle engine or database baseline. It covers the proposed-engine process or the full database container. Incremental memory subtracts that baseline. Oracle’s native value is its session PGA high-water mark. Trino’s native value is its reported query peak. A small incremental Trino value can therefore mean that memory was already reserved by the JVM; it does not mean that the server had no memory footprint.

APPENDIX A. DETAILED BENCHMARK TABLES

Table A.17: Stress-test execution time (seconds; mean $\pm$ SD)

Size (rows)	simple_sequence	alternation	quantified	optional_pattern	complex_nested
<i>Proposed engine</i>					
5,000,000	0.60 $\pm$ 0.01	0.67 $\pm$ 0.00	0.60 $\pm$ 0.01	0.70 $\pm$ 0.01	2.34 $\pm$ 0.02
10,000,000	1.30 $\pm$ 0.00	1.36 $\pm$ 0.01	1.37 $\pm$ 0.02	1.57 $\pm$ 0.00	4.81 $\pm$ 0.09
20,000,000	2.67 $\pm$ 0.02	2.73 $\pm$ 0.02	2.72 $\pm$ 0.03	3.11 $\pm$ 0.02	9.54 $\pm$ 0.06
40,000,000	5.26 $\pm$ 0.01	5.29 $\pm$ 0.01	5.44 $\pm$ 0.05	6.19 $\pm$ 0.07	19.50 $\pm$ 0.38
80,000,000	9.97 $\pm$ 0.14	9.36 $\pm$ 0.02	10.09 $\pm$ 0.25	11.62 $\pm$ 0.09	38.78 $\pm$ 1.38
160,000,000	19.49 $\pm$ 0.02	19.04 $\pm$ 0.27	20.87 $\pm$ 0.19	23.94 $\pm$ 0.06	77.03 $\pm$ 2.24
227,899,533	27.79 $\pm$ 0.04	24.81 $\pm$ 0.15	29.34 $\pm$ 0.08	33.76 $\pm$ 0.06	105.99 $\pm$ 2.28
<i>Oracle 21c EE</i>					
5,000,000	4.27 $\pm$ 0.05	1.88 $\pm$ 0.01	4.83 $\pm$ 0.02	3.10 $\pm$ 0.02	7.42 $\pm$ 0.03
10,000,000	8.78 $\pm$ 0.00	3.65 $\pm$ 0.02	9.95 $\pm$ 0.11	6.22 $\pm$ 0.01	15.25 $\pm$ 0.12
20,000,000	18.05 $\pm$ 0.08	7.22 $\pm$ 0.02	20.55 $\pm$ 0.05	12.55 $\pm$ 0.05	31.18 $\pm$ 0.12
40,000,000	48.89 $\pm$ 0.41	23.54 $\pm$ 0.63	54.54 $\pm$ 0.66	35.98 $\pm$ 0.45	78.75 $\pm$ 0.72
80,000,000	100.55 $\pm$ 1.16	46.08 $\pm$ 1.26	111.99 $\pm$ 1.15	71.67 $\pm$ 1.11	160.26 $\pm$ 1.43
160,000,000	226.56 $\pm$ 2.02	103.42 $\pm$ 3.83	243.30 $\pm$ 4.42	153.30 $\pm$ 3.38	335.53 $\pm$ 4.38
227,899,533	355.62 $\pm$ 8.88	183.46 $\pm$ 9.08	371.81 $\pm$ 8.03	255.91 $\pm$ 7.55	498.04 $\pm$ 9.54
<i>Trino 473 (disk connector)</i>					
5,000,000	10.58 $\pm$ 0.17	4.15 $\pm$ 0.18	10.90 $\pm$ 0.32	7.20 $\pm$ 0.09	16.05 $\pm$ 0.05
10,000,000	19.17 $\pm$ 0.17	8.00 $\pm$ 0.14	22.70 $\pm$ 0.51	14.19 $\pm$ 0.10	32.72 $\pm$ 0.34
20,000,000	39.75 $\pm$ 0.39	15.89 $\pm$ 0.08	45.53 $\pm$ 0.15	29.04 $\pm$ 0.09	70.32 $\pm$ 0.06
40,000,000	79.35 $\pm$ 0.49	29.75 $\pm$ 0.06	94.96 $\pm$ 0.47	58.31 $\pm$ 0.22	144.54 $\pm$ 2.84
80,000,000	163.39 $\pm$ 0.86	57.13 $\pm$ 0.14	191.85 $\pm$ 0.39	115.97 $\pm$ 0.24	287.75 $\pm$ 0.55
160,000,000	364.58 $\pm$ 2.02	111.17 $\pm$ 0.65	423.30 $\pm$ 1.02	253.00 $\pm$ 6.50	wall
227,899,533	483.85 $\pm$ 17.26	154.04 $\pm$ 0.90	553.56 $\pm$ 2.83	310.91 $\pm$ 1.58	wall

Table A.18: Stress-test absolute peak memory (GB)

Size (rows)	simple_sequence	alternation	quantified	optional_pattern	complex_nested
<i>Proposed engine</i>					
5,000,000	0.66	0.59	0.75	0.80	0.77
10,000,000	1.26	1.21	1.44	1.52	1.47
20,000,000	2.53	2.45	2.84	3.06	3.03
40,000,000	4.93	4.78	5.57	6.00	5.97
80,000,000	9.51	9.03	10.75	11.64	9.90
160,000,000	18.92	17.97	21.49	23.06	19.71
227,899,533	30.08	28.36	33.40	35.67	30.89
<i>Oracle 21c EE</i>					
5,000,000	15.49	15.49	15.50	15.50	15.50
10,000,000	15.25	15.28	15.28	15.28	15.26
20,000,000	15.74	15.73	15.76	15.72	15.88
40,000,000	17.18	17.19	17.36	17.39	17.40
80,000,000	19.78	19.94	19.99	19.93	19.99
160,000,000	25.63	25.63	25.63	25.70	25.69
227,899,533	27.02	26.87	26.98	27.72	27.71
<i>Trino 473 (disk connector)</i>					
5,000,000	13.32	13.35	13.38	13.39	13.40
10,000,000	13.59	13.60	13.61	13.62	13.83
20,000,000	13.88	13.90	13.92	14.12	16.23
40,000,000	18.99	20.48	21.45	22.86	27.20
80,000,000	25.31	30.07	38.40	47.51	47.55
160,000,000	30.60	39.23	47.87	47.97	wall
227,899,533	37.83	47.46	47.48	36.42	wall

APPENDIX A. DETAILED BENCHMARK TABLES

Table A.19: Stress-test throughput (million rows/s)

Size (rows)	simple_sequence	alternation	quantified	optional_pattern	complex_nested
<i>Proposed engine</i>					
5,000,000	8.33	7.43	8.31	7.16	2.14
10,000,000	7.68	7.34	7.28	6.39	2.08
20,000,000	7.50	7.32	7.34	6.43	2.10
40,000,000	7.60	7.56	7.36	6.47	2.05
80,000,000	8.03	8.55	7.93	6.88	2.07
160,000,000	8.21	8.41	7.67	6.68	2.08
227,899,533	8.20	9.19	7.77	6.75	2.15
<i>Oracle 21c EE</i>					
5,000,000	1.17	2.66	1.03	1.62	0.67
10,000,000	1.14	2.74	1.00	1.61	0.66
20,000,000	1.11	2.77	0.97	1.59	0.64
40,000,000	0.82	1.70	0.73	1.11	0.51
80,000,000	0.80	1.74	0.71	1.12	0.50
160,000,000	0.71	1.55	0.66	1.04	0.48
227,899,533	0.64	1.24	0.61	0.89	0.46
<i>Trino 473 (disk connector)</i>					
5,000,000	0.47	1.21	0.46	0.69	0.31
10,000,000	0.52	1.25	0.44	0.70	0.31
20,000,000	0.50	1.26	0.44	0.69	0.28
40,000,000	0.50	1.34	0.42	0.69	0.28
80,000,000	0.49	1.40	0.42	0.69	0.28
160,000,000	0.44	1.44	0.38	0.63	wall
227,899,533	0.46	1.48	0.41	0.73	wall

Table A.20: Stress-test incremental peak memory above the pre-query baseline (GB)

Size (rows)	simple_sequence	alternation	quantified	optional_pattern	complex_nested
<i>Proposed engine</i>					
5,000,000	0.321	0.204	0.374	0.417	0.317
10,000,000	0.582	0.468	0.711	0.787	0.620
20,000,000	1.213	1.019	1.457	1.627	1.332
40,000,000	2.373	2.018	2.890	3.208	2.663
80,000,000	4.915	4.023	5.876	6.532	4.246
160,000,000	9.631	7.988	11.758	12.946	8.436
227,899,533	14.018	11.409	16.739	18.459	12.048
<i>Oracle 21c EE</i>					
5,000,000	0.127	0.128	0.128	0.128	0.128
10,000,000	0.257	0.256	0.256	0.256	0.260
20,000,000	0.516	0.512	0.514	0.512	0.524
40,000,000	0.448	0.452	0.448	0.449	0.449
80,000,000	0.463	0.455	0.460	0.452	0.458
160,000,000	0.468	0.455	0.463	0.472	0.487
227,899,533	0.676	0.648	0.641	0.546	0.588
<i>Trino 473 (disk connector)</i>					
5,000,000	0.007	0.003	0.004	0.002	0.004
10,000,000	0.006	0.008	0.004	0.008	0.005
20,000,000	0.006	0.006	0.006	0.004	0.357
40,000,000	0.729	0.007	0.451	0.020	0.719
80,000,000	1.540	0.016	1.532	0.011	0.013
160,000,000	1.605	2.630	0.029	0.015	wall
227,899,533	2.658	1.374	0.776	1.545	wall

APPENDIX A. DETAILED BENCHMARK TABLES

Table A.21: Database-native per-query memory counters (GB)

Size (rows)	simple_sequence	alternation	quantified	optional_pattern	complex_nested
<i>Oracle 21c EE</i>					
5,000,000	0.129	0.129	0.129	0.129	0.129
10,000,000	0.260	0.260	0.260	0.260	0.260
20,000,000	0.522	0.522	0.522	0.522	0.522
40,000,000	0.453	0.453	0.453	0.453	0.453
80,000,000	0.453	0.453	0.453	0.453	0.453
160,000,000	0.453	0.453	0.453	0.460	0.460
227,899,533	0.453	0.453	0.453	0.460	0.460
<i>Trino 473 (disk connector)</i>					
5,000,000	0.091	0.091	0.091	0.091	0.091
10,000,000	0.195	0.194	0.194	0.194	0.194
20,000,000	0.334	0.333	0.334	0.333	0.333
40,000,000	0.708	0.708	0.708	0.708	0.708
80,000,000	1.510	1.509	1.509	1.509	1.509
160,000,000	2.601	2.600	2.601	2.604	wall
227,899,533	3.799	3.798	3.798	3.798	wall

Table A.22: Stress-test returned rows by pattern and size under ONE ROW PER MATCH

Size (rows)	simple_sequence	alternation	quantified	optional_pattern	complex_nested
5,000,000	591,476	103,496	323,805	309,050	351,661
10,000,000	1,150,566	193,114	631,723	605,909	681,288
20,000,000	2,289,658	369,406	1,272,734	1,224,714	1,361,317
40,000,000	4,399,482	665,442	2,470,577	2,389,804	2,617,208
80,000,000	8,216,302	1,190,587	4,727,306	4,597,581	4,974,189
160,000,000	14,293,355	1,960,129	8,716,637	8,536,256	9,102,417
227,899,533	18,836,350	2,656,560	12,355,865	12,134,528	12,900,322
