One Language to Rule them All: Behavioural Querying of Process Data using SQL

Jakob Brand[1], Timotheus Kampik[2,3], Cem Okulmus[3], and Matthias Weidlich[1,2]

[1] Humboldt-Universität zu Berlin, Berlin, Germany,
(brandjak,matthias.weidlich)@hu-berlin.de,
[2] SAP Signavio,
(timotheus.kampik,matthias.weidlich)@sap.com,
[3] Umeå University, Umeå, Sweden
(tkampik,okulmus)@cs.umu.se

Abstract. State-of-the-art solutions for process mining rely on proprietary, domain-specific languages to query data recorded during business process execution. To support common analysis tasks, these languages focus on the definition of queries for behavioural patterns. Yet, the use of domain-specific languages for process mining has drawbacks: they require specific user training, lead to a decoupling of the query models for (i) data extraction and transformation, and (ii) the actual analysis, and induce engineering overhead through the development of a dedicated query engine. In this work, we therefore explore the use of standard SQL for process mining tasks. In particular, we demonstrate that the SQL concepts for row pattern recognition as realised by the `MATCH_RECOGNIZE` clause are sufficient to capture queries for behavioural patterns as specified in the SIGNAL language by SAP Signavio as well as the Process Querying Language (PQL) by Celonis. Based on a discussion of the respective language features, we outline a translation of SIGNAL and PQL queries into standard SQL. This way, we provide the basis for the adoption of widely used, general purpose query engines for process mining tasks.

Key words: Process Querying, Process Mining, Pattern Recognition

1 Introduction

Process mining supports business process management through the analysis of event log data that has been recorded during process execution. Over the past decade, process mining has evolved from an academic field of inquiry into a widely adopted practice in large-scale organisations that is supported by special-purpose software products provided by major vendors such as Microsoft and SAP. To support a wide range of analysis tasks, *process querying* has become a cornerstone of existing process mining solutions and a vibrant direction of research within the community [11]. Yet, the assumption so far has mostly been that process querying is executed by special-purpose technologies, i.e., domain-specific languages that facilitate the definition of queries for behavioural patterns [5, 15] (Section 2).

While domain-specific languages for process querying can be tailored to specific analysis needs, their usage also induces certain drawbacks. They require specific training, which narrows the user group. They also lead to a decoupling of the query models for data preparation and analysis. Since process-related data is often stored in mainstream relational database systems, the extraction, transformation and loading (ETL) of the data is typically realized in standard SQL. Finally, the use of domain-specific languages for process querying incurs engineering overhead, through the development of dedicated query engines.

In this paper, we question the need for dedicated languages for process querying. We practically demonstrate that the behavioural querying capabilities of two industry-scale process query languages can be mapped to standard SQL, most notably using the `MATCH_RECOGNIZE` clause as introduced with the 2016 SQL standard revision (Section 3). This means that process behaviour can be analysed using "mainstream" database systems. As such, our work strengthens the bridge between process query languages and database theory and applications (Section 4), while simultaneously raising questions about *i)* the complexity and scalability of `MATCH_RECOGNIZE` for behavioural querying and *ii)* the potential of ubiquitous process querying with mainstream database technologies, e.g., directly on top of enterprise system databases and in the data lake-houses that serve as the data backbone for a wide range of business applications (Section 5).

2 Background

Below, we give an intuitive overview of two state-of-the-art languages for process querying, i.e., SIGNAL by SAP Signavio (Section 2.1) and PQL by Celonis (Section 2.2). Then, we review the SQL `MATCH_RECOGNIZE` clause (Section 2.3).

2.1 SIGNAL by SAP Signavio

SIGNAL [5] is a language for process querying provided by SAP Signavio as part of their process mining offering. The central data model in SIGNAL is a nested table, as illustrated in Table 1. It contains information on process executions on two levels. The outer level includes attributes for a case identifier (**case_id** in Table 1) and additional case properties, if available (**customer_name** and **order_value**). For each tuple of the outer level, the inner level contains tuples that describe the individual events recorded for a case, with attributes capturing an event type (**event_name**) and timestamp (**end_time**), and potentially further properties of events (**department**).

SIGNAL supports read-only queries that are specified in an SQL-like syntax, see Listing 1. The queries refer to a single nested table (`FROM` clause), which is typically derived from the query context (`THIS_PROCESS` in Listing 1). A SIGNAL query may be *flat* and refer only to the information at the case level in the nested table, through standard SQL operators for projection, selection, and aggregation (in `SELECT` and `WHERE` clauses). A query may also be *nested*, such that the outer

Table 1. Example of the SIGNAL columnar data format for event logs.

case_ID	customer_name	order_value	events		
			event_name	**end_time**	**department**
01	C1	599	Order received	2024-03-01 11:15	D1
			Invoice sent	2024-03-01 12:33	D2
			Payment received	2024-03-02 09:01	D2
			Order shipped	2024-03-05 14:39	D4
02	C3	149	Order received	2024-03-02 15:25	D1
			Invoice sent	2024-03-02 17:43	D2
			Timer expired	2024-03-09 17:44	D3
			Order cancelled	2024-03-09 18:02	D4

```
SELECT case_id
FROM THIS_PROCESS
WHERE BEHAVIOUR (event_name = 'Order received' AND 'order_value' > 300)
      AS order_300
MATCHES (^order_300 ~> 'Payment received' -> 'Order shipped'$)
```

Listing 1. Example of a SIGNAL query.

subquery refers to cases, while the inner subquery refers to events within cases. Such a nested query may leverage the order of events within a case as it is inferred from the events' timestamps (which is assumed to be total) in order to detect patterns based on temporal constraints (MATCHES clause). The events to consider for the evaluation of the constraints are either characterized implicitly (e.g., by referring directly to a value of the **event_name** column; 'Payment received' in Listing 1) or defined as so-called behaviours (BEHAVIOUR clause), i.e., subqueries that select the events of a case that satisfy the specified constraints.

2.2 PQL by Celonis

PQL [15] has been developed by Celonis as a query language for process mining tasks. Is adopts a so-called snowflake schema [15], as illustrated in Figure 1. Here, the central relations are an **Activities** table and a **Cases** table. Additional information is stored in further tables with a normalized schema (a **Customers** table and an **Orders** table in our example), which is linked to the **Activities** table and **Cases** table, respectively, by foreign key relationships. These relationships have to be defined when loading data into the respective model.

PQL queries are read-only and also adopt an SQL-like syntax. PQL supports a wide range of operators, from SQL-like aggregation and string modification functions through ML operators (e.g., k-means clustering) to operators for process mining tasks (e.g., a dedicated operator for conformance checking [2]).

In PQL queries, two important operations are performed implicitly based on the interpretation of the aforementioned tables in a process mining context. That is, queries over the **Activities** and **Cases** tables may refer to attributes of the additional tables, which are then joined implicitly according to the foreign keys. In addition, groupings are performed implicitly using all selected non-aggregated columns in a query.

Activities

Case	Activity	Timestamp	Department
01	Order received	2024-03-01 11:15	D1
01	Invoice sent	2024-03-01 12:33	D2
01	Payment received	2024-03-02 09:01	D4
01	Order shipped	2024-03-05 14:39	D4
02	Order received	2024-03-02 15:25	D1
02	Invoice sent	2024-03-02 17:43	D2
02	Timer expired	2024-03-09 17:44	D3
02	Order cancelled	2024-03-09 18:02	D4

Customers

CustomerID	Name
183	C1
121	C3

Cases

Case	Customer	Order
01	183	5431
02	121	1003

Orders

OrderID	OrderAmount
5431	599
1003	149

Fig. 1. Example of the PQL snowflake data model with four tables.

PQL's support for the identification of behavioural patterns primarily relies on three so-called process functions that match cases showing a specific pattern of activity executions: `PROCESS EQUALS` enables matching based on a reduced set of regular expressions. Patterns in `MATCH_PROCESS` are defined in graph structure, in which vertices are activities, or sets thereof, and edges describe behavioural relations between them. The most expressive clause, `MATCH_PROCESS_REGEX`, defines a pattern as a regular expression. The latter resembles the behavioural matching in SIGNAL, so that we will focus on this clause in the remainder.

A `MATCH_PROCESS_REGEX` query is shown in Listing 2. At first, a single string column of the **Activities** table on which the matching is performed is specified. Then, a pattern is defined for behaviours via equality or wildcard matching, and transitions between them. In the example, matching is performed on a **behaviour** column that is constructed by the `CASE`. It has three sub-clauses, one for the each of the behaviours, which define the condition (`WHEN`) of the behaviour as well as its name (`THEN`). The latter represents the values for the **behaviour** column, which are then matched via string equality. The matching clause adds a temporary integer column to the **Cases** table with 0/1 values, indicating whether a case matches the pattern. Wrapping the clause with a `FILTER = 1` condition will return all rows of all matched cases, as seen in the example.

As pattern detection is limited to a single string column, complex patterns on multiple/non-string columns are not directly supported. However, the `CASE WHEN` clause supports evaluation of arbitrary columns and can be used beforehand to create a new string column in the **Activities** table, which indicates the satisfied constraints for each row. This is illustrated in our example in Listing 2, where the string values 'order_300', 'payment_received', and 'order_shipped' indicate the fulfilment of the respective constraints.

2.3 SQL Match Recognize

With the 2016 revision of the SQL standard [7], the `MATCH_RECOGNIZE` clause has been introduced for row pattern recognition. While a concise description can be found in [10], we summarise the main concepts of `MATCH_RECOGNIZE` below.

```
CASE WHEN "Activities"."Activity"='Order Received'
        AND "Activities"."OrderAmount" > 300 THEN 'order_300'
   WHEN "Activities"."Activity"='Payment Received' THEN 'payment_received'
   WHEN "Activities"."Activity"='Order shipped' THEN 'order_shipped'
   ELSE ''
   END
FILTER MATCH_PROCESS_REGEX("Activities"."behaviour", ^'order_300' >>
       ANY* >> 'payment_received' >> 'order_shipped'$) = 1;
```

Listing 2. Example of a PQL query.

Listing 3 illustrates the syntax for the `MATCH_RECOGNIZE` clause. It operates on an input table, as constructed by the `FROM` clause, which may involve joins, and produces an output table, which is then processed by `SELECT` and other clauses (e.g., a `GROUP BY` clause). The `MATCH_RECOGNIZE` clause involves several operators:

`DEFINE`: This operator is mandatory and is used to define *pattern symbols*, which are—seen semantically—matched to a set of rows that satisfy some condition. These symbols may even refer to other symbols in their definition.

`PATTERN`. This operator is mandatory and comprises a regular expression, which may use symbols defined in `DEFINE`. Notably, it is allowed to include undefined symbols, which are given a dummy predicate that is satisfied by all rows. The regular expressions may include Kleene closure (* and $^+$), upper and lower cardinality bounds ($\{n, m\}$), alternatives ($+$), and references to the first and last row of a table (^ and $).

`ONE ROW PER MATCH` / `ALL ROWS PER MATCH`. Upon a "match", understood as the sequence of rows which satisfies the pattern, the content of the output table is derived as follows: `ONE ROW PER MATCH` produces one output row for every match, i.e., provides a certain aggregation. `ALL ROWS PER MATCH` performs no such aggregation, and outputs each row in the sequence making up a match.

`AFTER MATCH`. This optional operator controls, upon a "match", where to continue pattern matching, e.g., after the first or last row (in general, or representing a specific symbol).

`PARTITION BY`. This optional operator groups the rows of the table given a list of columns. The `MATCH_RECOGNIZE` clause is then evaluated per such group.

`ORDER BY`. While the ordering operator is optional, it carries the same meaning as when used outside a `MATCH_RECOGNIZE` clause.

`MEASURES`. This optional operator enables access to pre-defined internal functions to populate the output with additional columns, accessible *outside* the `MATCH_RECOGNIZE` clause, such as `match_number()` and `first()`.

`SUBSET`. This optional operator, given a list of pattern symbols, groups them to refer to them collectively (e.g., to compute aggregates).

The `MATCH_RECOGNIZE` clause is available in various database management systems, such as Oracle, Snowflake, and Trino, as well as data stream processing frameworks, such as Azure Stream Analytics, Flink, and Esper.

```
SELECT  <select list>
FROM    <source table>
MATCH_RECOGNIZE (
        [ PARTITION BY <partition list> ]
        [ ORDER BY <order by list> ]
        [ MEASURES <measure list> ]
        [ ONE ROW PER MATCH | ALL ROWS PER MATCH ]
        [ AFTER MATCH <skip to option> ]
        PATTERN ( <row pattern> )
        [ SUBSET <subset list> ]
        DEFINE <definition list> ) AS <table alias>;
```

Listing 3. The syntax of the `MATCH_RECOGNIZE` clause, as given in [10].

3 Language Comparison and Translation

In this section, we compare the above languages, and outline how SIGNAL and PQL queries can be translated to SQL using the `MATCH_RECOGNIZE` clause.

3.1 Query Input

The input data for process querying is saved either as a nested table (in SIGNAL) or in a pre-defined schema comprising an **Activities** table, a **Cases** table and optional additional dimensions (in PQL). In SIGNAL, specifying a process identifier as input in the `FROM` subclause is sufficient. In PQL, the **Activities** table is specified to identify the input data. When referencing data from the dimension tables in PQL, the necessary joins are performed implicitly.

In SQL, the `FROM` subclause specifies the tables or views based on which the table for the evaluation of the `MATCH_RECOGNIZE` clause is derived. The construction of this table follows common SQL semantics. That is, a listing of multiple tables leads to the (implicit) construction of a Cartesian product, which may be avoided by specifying explicit joins or subqueries.

When translating SIGNAL and PQL queries to SQL, therefore, all tables that capture relevant process data need to be included in the `FROM` clause, potentially joining them over the attributes for the activity or case identifiers.

3.2 Behaviour Definition

Process querying concerns the identification of patterns over some behavioural abstraction. Using the terminology of SIGNAL, we call these abstractions *behaviours*. A behaviour is defined by a Boolean condition that is evaluated against each event of the process data, i.e., against each row of a respective table.

Patterns are then constructed by specifying relations over behaviours, incorporating the ordering of rows as established by timestamp attributes. As detailed later, a pattern resembles a regular expression (regex), i.e., the behaviours can be seen as the alphabet over which to define the regex.

To differentiate the expressiveness of behaviours, we adopt the classification of conditions as presented for `MATCH_RECOGNIZE` in [16]: if a condition can be evaluated on a single row, it is called an *independent condition*. For behaviours that need to be evaluated across multiple rows, the condition is called a *dependent condition*. If all rows that need to be evaluated in a dependent condition are located in the same pattern match, the condition is classified as *self-contained*.

SIGNAL Behaviours can be defined implicitly with a string that is matched with a specified column (by default the **event_name** column). For an explicit definition, a `WHERE BEHAVIOUR` clause is part of the language. It supports a wide range of operators, such as comparison, logical, `LIKE`/`ILIKE`, `IS NULL`, and durations on event-level columns from the table. In the SIGNAL query in Listing 1, the first behaviour **order_300** is defined explicitly and matches all rows with the **event_name** 'Order received' and an **order_value** larger than 300. The second and third behaviours are implicitly defined in the `MATCHES` subclause.

Queries that include `BEHAVIOUR` and `MATCHES` clauses are restricted to nested tables. Therefore, the `BEHAVIOUR` clause operates at the inner (i.e., event) level, and comparisons to case-level aggregations like `SUM` or `AVG` are not possible. While the SIGNAL language includes `LAG`/`LEAD` operators to navigate to previous and subsequent rows in a match, they are defined as window functions that work on flattened tables and, therefore, are not applicable for row navigation in nested tables. Hence, behaviours in SIGNAL contain only independent conditions.

PQL To define behaviours, a user may specify a single string column (by default the **activity** column) for matching it against a set of given strings. In addition, string matching may incorporate `LIKE` with wildcards and grouped matching.

While such behaviour definition based on string matching offers only limited expressiveness, more complex behaviours may be derived using some limited data manipulation capabilities in PQL. That is, the `CASE_WHEN` operator enables the creation of a temporary table based on the **Activities** table that features an additional string column **behaviour**. The latter indicates the satisfied behaviour for each row and can be incorporated in the matching operator, as discussed already for the example given in Listing 2.

In a `CASE` clause, multiple conditions on columns and corresponding output values (i.e., behaviour names) can be specified. The conditions are evaluated on each row individually and may include comparisons, logical operators, `LIKE`/`ILIKE`, `IS NULL` and `BETWEEN` for time intervals; they may refer to neighbouring rows using `LAG`/`LEAD`; and they can include aggregations. Note that aggregates are by default applied to groups of all non-aggregated columns and, hence, computed at least on case groups. Row navigation with `LAG`/`LEAD`, in turn, operates on the whole table by default, so that a condition for a row can refer to rows from other cases. In pattern matching, this means that a behaviour can include dependent, not self-contained conditions. By using row navigation with a `PARTITION BY` clause on the **case-id** (or, equivalently for case partitioning, `ACTIVITY_LAG`/`ACTIVITY_LEAD`), one can ensure self-contained, dependent conditions. If neither aggregates nor row navigation is used, the behaviour conditions are always independent.

```
SELECT case_id
FROM events
MATCH_RECOGNIZE (
   PARTITION BY case_id
   ORDER BY end_time
   ONE ROW PER MATCH
   PATTERN (^order_300 ANY* payment_received order_shipped$)
   DEFINE order_300 AS event_name = ’Order received’ AND order_value > 300,
          payment_received AS event_name = ’Payment received’,
          order_shipped AS event_name = ’Order shipped’)
```

Listing 4. Example of an SQL query with `MATCH_RECOGNIZE`.

However, when using a `CASE` clause for behaviour definition, each row is assigned exactly *one* behaviour. To work around this limitation, one would need to leverage the string processing capabilities of PQL. That is, for each row, each behaviour is evaluated with a separate `CASE` clause and the resulting string values are concatenated (`CONCAT` or short `||`) in the **behaviour** column. In the pattern matching, the presence of a behaviour in this string is assessed using the `LIKE` operator, which we illustrate with the query in Listing 5 that is discussed later.

MATCH_RECOGNIZE In the `MATCH_RECOGNIZE` clause, behaviours are constructed in the `DEFINE` clause that includes a name followed by (`AS`) by the respective conditions. An example is given in Listing 4, where the behaviours **order_300**, **payment_received**, and **order_shipped** are defined. The conditions may include comparisons, logical operators, `LIKE`/`ILIKE`, `IS NULL`; they may refer to aggregates, and neighbouring rows `PREVIOUS`/`NEXT`. Therefore, depending on the partitioning of aggregates and row navigation functions, such a query can contain either not self-contained or self-contained conditions. Only if aggregates and row navigation are not used, the behaviour conditions are independent.

By default, undefined behaviours assign the value `TRUE` to any row. However, a placeholder behaviour that matches any row can also be modelled more explicitly using `ANY`, which we use in our examples.

Turning to the translation of SIGNAL queries to SQL, the behaviours defined in a `WHERE BEHAVIOUR` clause and in a `MATCHES` clause, need to be specified in the `DEFINE` clause of `MATCH_RECOGNIZE` (as illustrated for the exemplary queries in Listing 1 and Listing 4). The same translation needs to be applied for the behaviours defined in PQL queries as part of one or more `CASE` clauses (see Listing 2 and Listing 4). If `MATCH_PROCESS_REGEX` is used without `CASE` clauses, each behaviour in the PQL pattern is translated into an SQL behaviour using string equality or `LIKE`/`ILIKE` operators on the respective column.

We conclude that the definition of behaviours as realised in SIGNAL and PQL can be mapped to `MATCH_RECOGNIZE`.

Table 2. Operators for the definition of a pattern.

Operator	**Semantics** ($a, b \in E, a \neq b$)	**SIGNAL**	**PQL**	**SQL**
Directly follows	$a \succ b, \nexists\, c \in E \setminus \{a, b\}[a \succ c \wedge c \succ b]$	a -> b	a $\gg$ b	a b
Follows	$a \succ b$	a ~> b	a $\gg$ (ANY)* $\gg$ b	a (ANY)* b
Starts with	$\forall\ c \in E \setminus \{a\} : a \succ c$	^a	^a	^a
Ends with	$\forall\ c \in E \setminus \{a\} : c \succ a$	a$	a$	a$
Contains any	$\exists\ c \in E : a \succ c, c \succ b$	a ANY b	a $\gg$ ANY $\gg$ b	a ANY b
Does not contain	$\nexists\ c \in E$	NOT c	[!c]	/
Alternation	$a \vee b$	a | b	a | b	a | b
Repetition (≥ 0)	$\bigcup_{i=0}^{\infty} a^i$	a*	a*	a*
Repetition (≥ 1)	$\bigcup_{i=1}^{\infty} a^i$	a+ †	a+	a+
0 - 1 occurrences	$a \vee \epsilon$	a? †	a?	a?
x - y occurrences	$\bigcup_{i=x}^{y} a^i$	a{x, y} †	a{x, y}	a{x, y}
One from set	$a \vee b \vee c$ with $c \in E \setminus \{a, b\}$	(a | (b | c))	[a,b,c]	(a | (b | c))

† The operators are available in SIGNAL, but not yet described in the public documentation.

3.3 Pattern Definition

In SIGNAL and PQL (using the `MATCH_PROCESS_REGEX` clause), patterns are matched per case, considering the order of events as inferred from their timestamps. These notions of events and cases need to be translated to SQL using the `PARTITION BY` clause, to group rows of the input table by the attribute denoting the case identifier, and the `ORDER BY` clause, to order events by timestamps.

To define a pattern, all languages offer operators that are similar to regular expressions, as summarized in Table 2. Here, we first illustrate the operator semantics, using E to denote a set of events (rows) of a single case and $\succ \subseteq E \times E$ as the temporal order over E, before giving the pattern definitions in SIGNAL, PQL, and `MATCH_RECOGNIZE` in SQL. Table 2 highlights many similarities among the languages. As such, a translation of SIGNAL and PQL patterns into the `PATTERN` clause of `MATCH_RECOGNIZE` is straight-forward, except for two aspects.

First, to ensure that only a single match of a pattern per case is returned, partition-wise maximal matching needs to be enforced in SQL. That is, if a SIGNAL/PQL query does not include the *starts/ends with* operators, they need to be added in the `MATCH_RECOGNIZE` pattern as ^`ANY`* and `ANY`*$.

Second, SQL lacks a pattern operator for *does not contain*. To achieve the respective semantics, an auxiliary behaviour needs to be defined with the logical `NOT` operator, which is then used in the pattern definition.

We illustrate these aspects of the translation with the PQL query in Listing 5. It exemplifies the aforementioned approach to represent multiple behaviours per row through string concatenation. That is, the two `CASE` clauses yield a **behaviour** column that contains a concatenated string 'beh_invoice_d2,beh_d2,' as value if a row satisfies both conditions. In addition, the PQL example includes a *does not contain* operator, which is realized by a check for the negated behaviour **not_d2** in the corresponding SQL query in Listing 6. Finally, the example highlights that the absence of *starts/ends with* operators in the PQL query requires the insertion of ^`ANY`* and `ANY`*$ in the SQL query to achieve an equivalent expression.

```
CASE WHEN "Activities"."Activity" = 'Invoice sent'
     AND "Activities"."Department" = 'D2' THEN 'beh_invoice_d2,',
ELSE ''
END ||
CASE WHEN "Activities"."Department" = 'D2' THEN 'beh_d2,',
ELSE ''
END
FILTER MATCH_PROCESS_REGEX("Activities"."behaviour",
LIKE '%beh_invoice_d2%' >> [! 'beh_d2,']) = 1;
```

Listing 5. PQL query with NOT operator and concatenated CASE clauses.

```
SELECT case_id
FROM events
MATCH_RECOGNIZE (
   PARTITION BY case_id
   ORDER BY end_time
   ONE ROW PER MATCH
   PATTERN (^ANY* invoice_d2 not_d2 ANY*$)
   DEFINE invoice_d2 AS event_name = 'Invoice sent' AND department = 'D2'
   not_d2 AS NOT(department = 'D2'))
```

Listing 6. SQL query with NOT operator.

3.4 Query Output

Turning to the capabilities of the languages to define the structure of the generated output, we first note that SIGNAL and PQL operate on cases as output instances. That is, if a pattern is matched at least once, the entire corresponding case is included in the construction of the result, as detailed below.

SIGNAL The `SELECT` clause may contain attributes on the case or event level, as well as aggregates over them. The output is a nested table with all specified attributes and aggregates for all matched cases. Matching in SIGNAL is existential, i.e., one satisfied match of behaviours per case is sufficient [5].

PQL A temporary column is added to the **Case** table, which contains 1 if a pattern is found in a case; and 0 otherwise. Using a `FILTER=1` statement, all rows of all matched cases may be selected.

`MATCH_RECOGNIZE` The result structure is defined in the `SELECT` clause, while the `MEASURES` clause of `MATCH_RECOGNIZE` further facilitates the computation of aggregates and the use of match-specific functions. When translating a SIGNAL query to SQL, columns at either case or event level as well as aggregates over them need to be included in SQL's `SELECT` statement. If the chosen columns are only on the case level, `MATCH_RECOGNIZE` is used with `ONE ROW PER MATCH`; otherwise `ALL ROWS PER MATCH` has to be selected. In PQL, when selecting matching cases by `FILTER=1`, all attributes from all rows of the matched cases are returned. In SQL, `SELECT *` with `ALL ROWS PER MATCH` mirrors this behaviour.

4 Related Work

Academic process query languages typically have their roots in process modelling and mining, and may thus query either process models [11, 1, 4] or process event data [9, 8], in the latter case typically in the form of event logs. Industry-scale process query languages tend to focus on the querying of event data, presumably because process models are queried using mainstream relational and document-based approaches, where aspects specific to the domain of BPM may be lifted to the business logic level. For (process) event data, the two languages described above are *the* two key examples of domain-specific process query languages that have already been described in the literature.

However, process querying is rarely integrated into the wealth of database management research. Notable examples include approaches for the discovery of declarative process specifications, which employ standard SQL to query for behavioural patterns [12, 13, 14]. Here, the conditions that need to be verified to instantiate constraint templates are particularly suitable to be expressed as declarative queries. For imperative models, the efficient extraction of control-flow dependencies is less straight-forward, which led to efforts to implement dedicated operators directly in the database management system [3].

Turning to generic languages for process querying, little work focused on a comparison of these languages with mainstream database languages such as SQL, or with languages that are theoretically well understood, such as Datalog. In [6], the analysis of expressive power and data complexity of SIGNAL is based on a characterisation of the core of SIGNAL using semi-positive Datalog; i.e., here the mapping from process query language to a (theoretically well understood) database query language aids formal analysis.

In contrast, our focus has been the use of standard SQL for process querying, showing that the `MATCH_RECOGNIZE` clause is sufficient to query for behavioural patterns as supported by SIGNAL and PQL. We believe that our results make a compelling case for the value of inquiry also in this direction, with the objective of making process querying more straightforwardly applicable, using the technologies that tend to be readily available in large-scale (enterprise) information systems.

5 Conclusions

In this paper, we demonstrated that behavioural queries in two industry-scale process query languages can be mapped to standard SQL. Our intuitive analysis raises some technical questions, most notably regarding *i)* the performance of `MATCH_RECOGNIZE` implementations when querying large event logs (e.g., with billions of entries) and *ii)* the theoretical data complexity and expressive power of `MATCH_RECOGNIZE`. Answering these questions may be particularly interesting relative to the characteristics of real-world process querying languages, whose scalability, complexity, and expressive power are (also) understudied. Beyond these technical aspects, our results can serve as a starting point enabling process querying and mining directly in the ecosystem of mainstream database systems.

For example, it may enable process mining with the standard query languages of enterprise systems' relational databases, as well as in data lakehouses that collect process data of multiple organisations for the purpose of benchmarking.

References

1. Awad, A., Sakr, S.: On efficient processing of bpmn-q queries. Computers in Industry **63**(9), 867–881 (2012)
2. Carmona, J., van Dongen, B.F., Solti, A., Weidlich, M.: Conformance Checking - Relating Processes and Models. Springer (2018)
3. Dijkman, R.M., Gao, J., Syamsiyah, A., van Dongen, B.F., Grefen, P., ter Hofstede, A.H.M.: Enabling efficient process mining on large data sets: realizing an in-database process mining operator. Distributed Parallel Databases **38**(1), 227–253 (2020)
4. Francescomarino, C.D., Tonella, P.: The BPMN visual query language and process querying framework. In: Polyvyanyy, A. (ed.) Process Querying Methods, pp. 181–218. Springer (2022)
5. Kampik, T., Lücke, A., Horstmann, J., Wheeler, M., Eickhoff, D.: Signal – the sap signavio analytics query language (2023)
6. Kampik, T., Okulmus, C.: Expressive power and complexity results for signal, an industry-scale process query language. In: BPM Forum 2024. LNBIP, Springer (2024), to appear.
7. Michels, J., Hare, K., Kulkarni, K., Zuzarte, C., Liu, Z.H., Hammerschmidt, B., Zemke, F.: The new and improved sql: 2016 standard. SIGMOD Rec. **47**(2), 51–60 (dec 2018)
8. de Murillas, E.G.L., Reijers, H.A., van der Aalst, W.M.P.: Data-aware process oriented query language. In: Polyvyanyy, A. (ed.) Process Querying Methods, pp. 49–83. Springer (2022)
9. Pérez-Álvarez, J.M., Díaz, A.C., Parody, L., Quintero, A.M.R., Gómez-López, M.T.: Process instance query language and the process querying framework. In: Polyvyanyy, A. (ed.) Process Querying Methods, pp. 85–111. Springer (2022)
10. Petkovic, D.: Specification of row pattern recognition in the SQL standard and its implementations. Datenbank-Spektrum **22**(2), 163–174 (2022)
11. Polyvyanyy, A.: Process query language. In: Process Querying Methods, pp. 313–341. Springer (2022)
12. Riva, F., Benvenuti, D., Maggi, F.M., Marrella, A., Montali, M.: An sql-based declarative process mining framework for analyzing process data stored in relational databases. In: BPM Forum 2023. LNBIP, vol. 490, pp. 214–231. Springer (2023)
13. Schönig, S., Ciccio, C.D., Mendling, J.: Configuring sql-based process mining for performance and storage optimisation. In: ACM/SIGAPP SAC 2019. pp. 94–97. ACM (2019)
14. Schönig, S., Rogge-Solti, A., Cabanillas, C., Jablonski, S., Mendling, J.: Efficient and customisable declarative process mining with SQL. In: CAiSE 2016. LNCS, vol. 9694, pp. 290–305. Springer (2016)
15. Vogelgesang, T., Ambrosy, J., Becher, D., Seilbeck, R., Geyer-Klingeberg, J., Klenk, M.: Celonis PQL: A query language for process mining. In: Process Querying Methods, pp. 377–408. Springer (2022)
16. Zhu, E., Huang, S., Chaudhuri, S.: High-performance row pattern recognition using joins. Proc. VLDB Endow. **16**(5), 1181–1194 (2023)