Petriflow Query Language

Gabriel Juhás[1], Juraj Mažári[2], Tomáš Kováčik[2], Milan Mladoniczky[1] and Martin Širáň[2]

[1]Faculty of Informatics, Pan-European University Tematínska 10, 851 05 Bratislava, Slovakia
[2]NETGRIF, s.r.o., Dúbravská cesta 14, 841 04 Bratislava, Slovakia, netgrif@netgrif.com , Home Page: https://netgrif.com

Abstract

Querying process models, running instances of processes, and logs of their execution is important for developers and analysts of process driven applications. In this paper we present the design and implementation of a domain-specific query language for the object-centric language Petriflow. After providing a brief contextual positioning of existing process query languages and establishing the specific requirements for an object-centric runtime environment, we define a simplified version of the extended Backus–Naur form of the query language over objects of Petriflow classes. We also provide details of the query language implementation in the Netgrif Platform.

Keywords

Petriflow, Query Language, Case, Task, Petri nets

1. Introduction

Searching and querying a database is an essential part of every application. It is important for developers to have a single, powerful query language to use. Process-driven applications have three distinct concerns for such query languages:

1. Querying process models - answering questions like *"In which processes is the 'Person verification' REST service called?"* or *"In which processes does the legal department participate?"*.
2. Querying instances of processes - answering questions like *"Which instances of the claim process were made using this email address?"* or *"How many instances of the claim process are in the state 'waiting for answer from legal department'"*.
3. Querying execution history of processes - answering questions like *"How many times was a claim made?"* or *"What is the average time for an employee to answer a customer's request?"*.

Many process query languages exist, but they mainly focus on querying process definitions (1.) or their retrospective execution history (3.). For instance, the Business Process Query Language (BP-QL) focuses on querying business process schemas, answering questions like *"Can I get a price quote without giving my credit card details first?"* [1]. Another language, BPMN-Q, addresses process topologies and searches for structural patterns within a process repository, using queries like *"Activity B is immediately followed by activity C"* [2]. On the analytical end of the spectrum, Celonis PQL (Process Query Language) was designed for business users to translate process-related questions into aggregations over historical logs, such as *"How many purchase orders violated segregation of duties for activities 'Request Approval' and 'Grant Approval'?"* [3].

To position our work within this existing landscape, these prominent query languages can be broadly categorized into two paradigms: structural querying of process models or retrospective analysis of event logs. Languages such as BP-QL and BPMN-Q are designed to query the topology and paths of process schemas (e.g., verifying compliance at the design stage), without addressing the live runtime state or data attributes of individual instances. Similarly, while highly optimized for aggregations, process discovery, and conformance checking over read-only data, Celonis PQL operates independently

PNSE'26, International Workshop on Petri Nets and Software Engineering, 2026
gabriel.juhas@paneurouni.com (G. Juhás); mazari@netgrif.com (J. Mažári); kovacik@netgrif.com (T. Kováčik); milan.mladonczizky@stuba.sk (M. Mladoniczky)
0000-0001-8302-5112 (G. Juhás); 0000-0002-8015-7213 (J. Mažári); 0000-0003-3157-8609 (T. Kováčik); 0000-0002-1385-2304 (M. Mladoniczky)

of a live execution engine. Because these existing languages were designed for post-execution analytics or static design-time analysis respectively, they cannot be used to interact with a live process engine, as they lack the capabilities to query current execution states and active data variables in real time. A comprehensive comparison of these paradigms is summarized in Table 1.

Table 1
Comparison of Process Query Languages

Feature / Dimension	**BP-QL / BPMN-Q**	**Celonis PQL**	**Petriflow Query Language (PFQL)**
Primary Goal	Structural querying of process models/graphs.	Analytical querying of historical event logs (Mining).	Operational querying of live process instances (Execution).
Target Objects	Process definitions, tasks, paths.	Activity logs, timestamps, event data columns.	Active cases, execution states, object-centric attributes.
Runtime Context	Design-time / Static analysis.	Post-execution / Read-only analytics.	**Real-time execution / Read-Write application logic.**
Underlying Engine	Graph databases or XML parsers.	Column-oriented in-memory analytical databases.	**Distributed transactional databases (MongoDB, Elasticsearch).**

In contrast, PFQL addresses a fundamentally different operational requirement: the real-time querying of active, running instances (cases) within an object-centric process execution platform. Rather than analyzing static structures or historical logs, PFQL provides a unified low-code syntax to query the intersection of process control-flow (e.g., currently enabled tasks) and live business data (object-centric attributes) across multiple dynamically linked process instances.

A primary motivation for a query language specifically designed for Petriflow objects (classes) is the necessity to establish and manage dynamic relationships between different process instances (cases). At this stage, PFQL is intentionally focused on the static state and the current marking of objects. This functionality is essential for identifying and retrieving specific cases based on their current data and state, as well as for locating tasks (transitions) required for synchronization. Furthermore, such querying capabilities are vital for UI composition, where retrieved tasks are dynamically integrated as sub-forms within parent process forms.

A natural question arises regarding the specific relevance of PFQL to the Petri nets: *Why is a dedicated query language necessary for Petriflow, and why can it not be applied generically to any process formalism like BPMN?* The answer lies in the strict mathematical and structural properties of Petri nets that are natively embedded into the syntax and semantics of PFQL. Unlike traditional process languages that treat data as an isolated payload, Petriflow leverages the formal semantics of Petri nets, allowing PFQL to query the execution space with mathematical precision through three fundamental aspects:

1. **Querying the Active Marking:** In Petri nets, the exact state of a process instance is explicitly defined by its *marking* (the distribution of tokens). PFQL directly utilizes this concept by allowing users and automated agents to query active states based on the presence of tokens in specific places. This enables real-time filtering of cases that are in mathematically well-defined states, a feature unavailable in informal block-structured formalisms.
2. **Transition-Centric Task Execution and Cross-Case Synchronization:** In Petriflow, user actions and system tasks are modeled directly as Petri net *transitions*. PFQL includes specialized operational operators to query the visibility, enablement, and fireability of these transitions in real time. A critical application of this capability is the dynamic synchronization between a parent process and its child instances. To synchronize a task in a parent process with tasks in corresponding child instances, PFQL is used to resolve relational references dynamically. For

instance, by leveraging a data variable acting as a reference (similar to a "foreign key"), PFQL can retrieve all child instances linked to a specific parent case. Crucially, the language filters these instances to identify only those currently in a specific marking where the child task is *enabled*. Once these specific transitions are located via PFQL, the execution engine can programmatically fire them, achieving seamless transaction-safe cross-case coordination.

3. **Dynamic Structural Interaction via Variable Arcs:** Most importantly, data variables in Petriflow are not detached from the process logic; they are functionally integrated into the net configuration. Petriflow utilizes data variable values as weights on *variable arcs*. Consequently, a change in a data attribute directly alters the underlying Petri net structure and immediately affects the enablement (*enabledness*) of transitions (tasks). PFQL unifies this data-control flow interaction within a single atomic query operation.

Consequently, PFQL is not a generic process query language; it is an execution-level interface whose syntax, execution engine, and capability to manage concurrent state synchronization are fundamentally enabled by, and dependent on, the underlying Petri net.

Therefore, if Petri nets are to be practically utilized as a lifecycle model for process-centric objects with integrated data, a dedicated query language like PFQL is essential to support their operational runtime execution and management.

The remainder of this paper is structured as follows. In Section 2, we introduce the Petriflow language and Section 3 provides a motivational example. In Section 4 we provide a formal definition of our Petriflow Query Language (PFQL) using an Extended Backus-Naur Form (EBNF). Section 5 details the architecture and practical implementation of the query language within the Netgrif Platform, followed by evaluation and future work directions in Section 6.

2. Petriflow

Petriflow object-centric processes integrate *data attributes*, *lifecycle* in the form of extended Petri nets, where transitions represent activities and tasks, and *user interfaces* as forms that can be associated with tasks. Data attributes themselves are XML objects with type, id, title, validations, etc. (see petriflow.org for more details). As a lifecycle, place/transition nets [4, 5, 6] with inhibitor [7], read [8], and reset arcs [9] are used with possible variable weights of arcs given by values of data attributes of type number or markings of places, inspired by self-modifying nets [10]. Forms are subsets of data attributes together with an information whether a data attribute on a form is required, editable, or read-only.

Relations between instances of object-centric processes are modelled on the data layer. Data attributes can reference one or many related instances similar to concepts of foreign keys in SQL databases, references in object-oriented programming, or one-to-one and one-to-many relationships in entity-relationship or class diagrams.

Searching and retrieving data of related instances requires a use of a query language. Petriflow language does not strictly define which database should be used to store all data, therefore it needs its own query language. This query language should fulfil following requirements:

- database-agnostic - user should not be required to learn a specific database query language in different implementations that use Petriflow language,
- single data structure - user should not be concerned how the processes and instances are stored and in which format or data structure,
- enhance low-code capabilities - the query language should be easy to learn and intuitive.

With these requirements in mind we designed a query language inspired by SQL as its queries are readable almost like sentences, they have a simple and structured notation and users do not need to know any specific data format like JSON.

3. Motivation example

The primary motivation for a specialized query language in the Petriflow framework is the need to establish and manage dynamic relationships between different process classes. In object-centric process management, a single case often needs to interact with a collection of other cases based on shared data attributes.

For example, consider a *Bulk Approval* process containing a *Bulk Approval* task, designed to approve multiple requests simultaneously. The instance of *Bulk Approval* process may contain data attributes for a specific *name* and *surname*, such as *'John Doe'*. To function correctly, this process must establish a **one-to-many relationship** with all instances of the *request* process depicted in Figure 1, that share the same attributes of *name* and *surname*. To create this link, the system must perform a lookup to identify and retrieve all relevant case objects. In this example, it is realized by an action triggered by pressing *Get requests* button in the *Bulk approval* task of the *Bulk Approval* process. The action containing the query is in Figure 2.

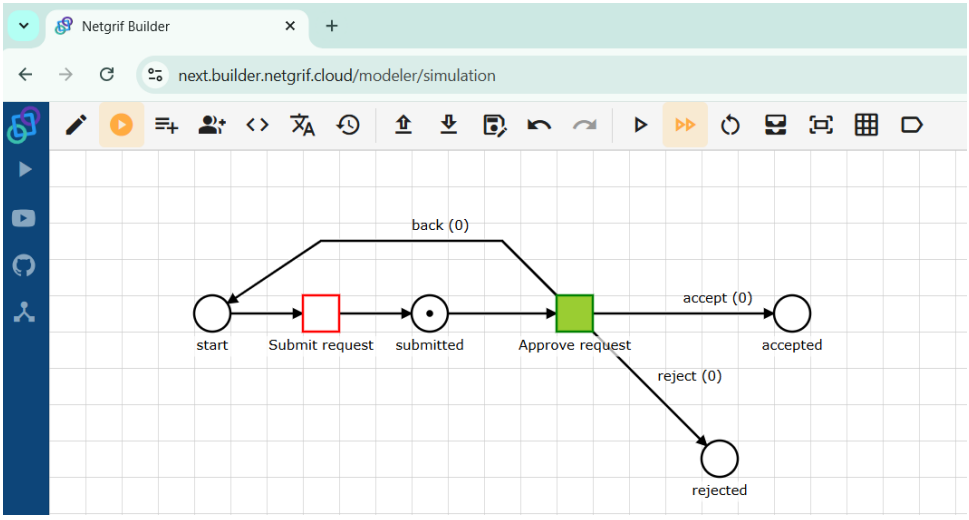

Figure 1: Request process with a token in the place *submitted* waiting for approval.

To support these complex cross-case synchronization requirements, the underlying Netgrif execution platform relies on a hybrid storage architecture consisting of MongoDB and Elasticsearch. MongoDB acts as the primary transactional database to guarantee strict compliance and persistence of case states, while active data variables and process-centric objects are continuously synchronized into Elasticsearch to enable high-performance text searching. However, before the introduction of a unified query language, this architectural separation forced developers to interact with two completely different database interfaces and query syntaxes within the application logic.

To illustrate this issue within our example, where the system must search for process instances with a *processIdentifier* equal to *'request'*, a *name* attribute of *'John'*, and a *surname* attribute of *'Doe'*, a developer would traditionally have to write a backend query using the *QueryDSL* framework for MongoDB:

```
findCases{
    it.processIdentifier.eq('request')
    .and(it.dataSet.get('name').value.eq(name.value))
    .and(it.dataSet.get('surname').value.eq(surname.value))
}
```

This query can be hard to read for a non-programmer and brings a risk of slow performance because data attributes (*dataSet*) are not indexed; in the worst case, MongoDB has to iterate over all records. To solve this problem, we would need to use Elasticsearch with its own *Query string* query:

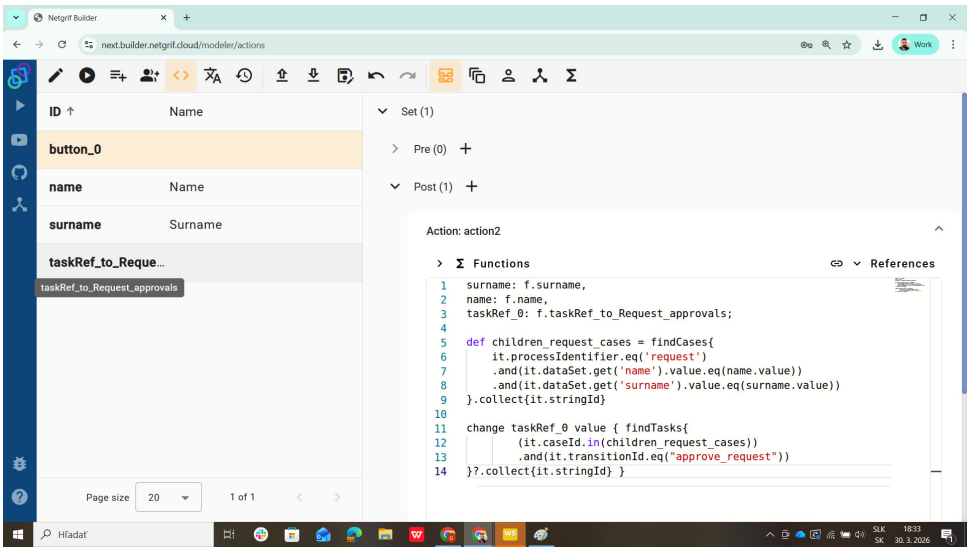

Figure 2: Action with a search query executed by pressing a button within *Bulk approval* task of a *Bulk approval* process. The implementation performs two main steps. First, it searches for instances of the `request` process where the *name* and *surname* attributes match those of the current *Bulk approval* case. Second, it retrieves the corresponding *Approve request* tasks (with ID equals *approve_request*) for the identified cases.

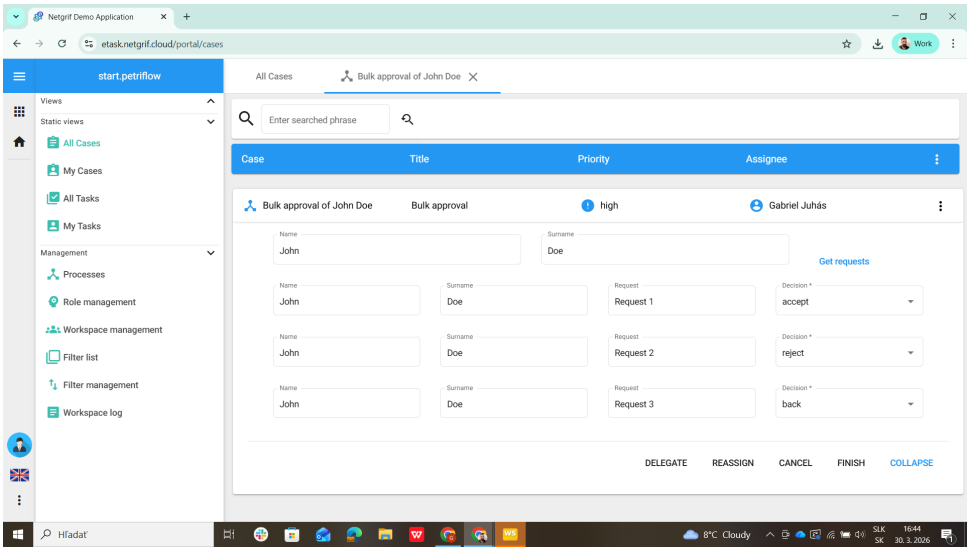

Figure 3: After executing action in *Get requests* button the forms of *Approve request* tasks of cases with name and surname equal to name and surname of the *Bulk approval* are displayed as subforms of the *Bulk approval* task

```
findCasesElastic(
    "processIdentifier: 'request'
    AND dataSet.name.textValue.keyword: '${name.value}'
    AND dataSet.surname.textValue.keyword: '${surname.value}'")
```

While the Elasticsearch query is more readable, the fundamental problem remains: it tightly couples the application logic to the internal structural mechanics of a specific storage engine. A developer must possess deep infrastructure-level knowledge, specifically, that the *processIdentifier* field can be queried directly, but custom data attributes require appending the technical *.keyword* suffix to match exact text values.

Furthermore, simply shifting from MongoDB QueryDSL to Elasticsearch string queries does not solve

the core usability issue; in both cases, the user is forced to learn a database-specific syntax. Even if one of these existing syntaxes (such as the Elasticsearch query string) were adopted as an integral part of the Petriflow language and programmatically translated to other formats when necessary, it would introduce a significant architectural antipattern. This approach would forcefully inject technology-specific implementation details (like *.keyword* indices, full-text analyzers, or search relevance scores) into the clean, formal specification of Petri nets.

This creates a critical vulnerability for the ecosystem: since Petriflow is designed as an open and platform-independent formalism, any third-party developer should be able to implement their own execution engine using completely different data persistence layers. If the language specification were tightly coupled to Elasticsearch syntax, migrating the platform to alternative technologies, such as Apache Solr, OpenSearch, standard relational databases, or modern vector/graph databases, would render the query grammar obsolete and break backward compatibility.

The true advantage of the proposed approach is therefore the introduction of a **neutral, domain-specific query language (DSL)** as an integral part of the Petriflow language. PFQL completely abstracts the shifting hybrid infrastructure away, allowing developers and domain analysts to formulate queries using concepts natively grounded in the application domain itself (e.g., cases, transitions, markings, and process identifiers). By decoupling the query formulation from the data-at-rest storage formats, PFQL ensures that the query logic remains stable, readable, and strictly aligned with the Petri net formalism, shielding both current and future implementations from any technology-specific indexing requirements.

4. Petriflow Query Language

To showcase the composition of a PFQL query we provide a simplified version of the PFQL EBNF. This simplified version omits differences between resources and their attributes and uses special sequences (denoted by a pair of question marks) to define attributes and their values.

Searching for resources is the only goal of PFQL each query needs to define resource its searching and predicates that it needs to fulfil. Optionally the query can define paging and sorting.

```
query = statement resource where predicates [ paging ] [ sorting ]
```

PFQL defines statements *search* for retrieving resources, *count* for getting the number of resources that fulfil the query, and *exists* for finding if any resource fulfil the query.

```
statement = 'search' | 'count' | 'exists' ;
```

Petriflow defines the following *resources*: *process*, instance of a process (*case*), *task* of an instance, *user*, and *role* [11, 12]. To distinguish between querying a single resource and searching for multiple resources, the language explicitly defines both singular and plural forms for each resource type. While plural forms return entire matching result sets, singular forms are deterministically optimized to restrict the output to a single instance without causing runtime errors when multiple matches exist. The precise operational mechanics and aggregation behaviour of these forms are detailed in Section 4.3.

```
resource            = single_resource | multiple_resources ;
single_resource     = 'process' | 'case' | 'task' | 'user' | 'role' ;
multiple_resources  = 'processes' | 'cases' | 'tasks' | 'users' | 'roles' ;
```

The traditional `where` clause can be simplified to a single `:` character. While replacing the `where` keyword with a colon (`:`) can be considered a form of syntactic sugaring, it significantly aligns the language with shorthand query conventions that users and automated agents are already accustomed to in platforms like Lucene, Elasticsearch, or GitHub search filters. This reduction in verbosity improves readability and ease of use in low-code environments without altering the underlying query semantics.

```
where = 'where' | ':' ;
```

Furthermore, PFQL supports two complementary API styles depending on whether a generic or resource-specific search function is utilized. When calling the universal `search()` function, the developer must explicitly define both the target resource type and the predicate boundary using the full grammar specification, for example:

```
search("cases: processIdentifier eq 'request'")
```

Alternatively, if the Petriflow interpreter provides specific shorthand methods for direct resource querying, both the resource name and the condition keyword become redundant and can be omitted within the string argument. In such cases, evaluating the full-syntax query above is programmatically equivalent to passing only the predicate string to a specialized method:

```
searchCases("processIdentifier eq 'request'")
```

where the execution engine automatically infers the target resource (`cases`) from the method name itself.

The grammar production `query = statement resource where predicates [ paging ] [ sorting ]` strictly defines both the `resource` and the `where` clauses as mandatory components of the language syntax to ensure deterministic parsing.

However, from an application development perspective, as mentioned above, the execution platform provides high-level API shorthand methods, such as the method `searchCases(predicates)` or the method `searchTasks(predicates)`. It is critical to clarify that these shorthands do not imply that the language grammar itself treats these clauses as optional. Instead, these runtime wrapper functions programmatically reconstruct the fully compliant PFQL string (e.g., automatically prepending `cases where`) before passing the query string to the ANTLR interpreter. Therefore, the core language parser always enforces the full production rule specification.

Query predicates allow to use logical operators *and* and *or* ('&' and '|' respectively) and their nesting using parenthesis. The nesting of predicates is defined through recursion in the EBNF.

```
predicates = term { or term } ;
term       = predicate { and predicate } ;
predicate  = comparison | ' ( ' predicates ' ) ' ;
comparison = attribute operator value ;
or         = 'or' | '|' ;
and        = 'and' | '&' ;

attribute = ? resource attribute ? ;

operator   = symbol_op | word_op ;
symbol_op  = '==' | '!=' | '<' | '>' | '<=' | '>=' ;
word_op    = 'eq' | 'neq' | 'lt' | 'gt' | 'lte' | 'gte' ;

value      = ? string or number ? ;
```

The list of common comparison operators in the core grammar includes equality (`eq` or `==`), inequality (`neq` or `!=`), less than (`lt` or `<`), greater than (`gt` or `>`), less than or equal (`lte` or `<=`), and greater than or equal (`gte` or `>=`).

To keep the paper comprehensive and focus on the fundamental execution semantics, the EBNF presented above is a simplified core version. The full production grammar includes advanced, type-specific operational operators that lack direct symbolic counterparts. For instance, text attributes support pattern matching via the `like` operator, while numerical and collection structures utilize domain-specific keywords such as `in range` or `in list` to evaluate continuous intervals and set membership. The complete, fully implemented ANTLR v4 grammar file (`.g4`) and its parser rules are publicly available in our open-source repository [13].

The language distinguishes between standard comparison operators and advanced predicates. Common comparison operators can be written either as mathematical symbols (==, !=, <, >, <=, >=) or as their equivalent text-based keywords (`eq`, `neq`, `lt`, `gt`, `lte`, `gte`) to enhance flexibility for different developer preferences.

In contrast, the advanced operational operators—namely `like`, `in_range`, and `in_list`—are inherently text-only and lack symbolic counterparts. This is because they represent higher-level, domain-specific evaluation logic tailored to the integrated data structures of Petriflow classes (as detailed in Table 1), rather than primitive scalar comparisons.

To formalize how core process components are targeted, the language utilizes a structured dot-notation path instead of generic placeholders. Within the Petriflow XML schema specification [12], components are encapsulated as distinct elements, namely data fields, places, and transitions, where each structural element is defined by a unique identifier.

In PFQL, these components are syntactically resolved by prefixing the specific element type (such as `data`, `place`, or `transition`) followed by a dot and the dynamic identifier of the targeted object. This approach accurately reflects how expressions from our motivational example, such as `data.name` or `data.surname`, are mapped and parsed. This dot-notation allows the language to unambiguously differentiate between core system properties and dynamic, user-defined process elements, while the complete tokenization and escape character rules are fully managed by the underlying ANTLR v4 grammar available in the repository [13].

In PFQL, these components are syntactically resolved by prefixing the specific element type (such as `data`, `places`, or `tasks`) followed by a dot and the dynamic identifier of the targeted object. This approach accurately reflects how expressions from our motivational example, such as `data.name` or `data.surname`, are mapped and parsed, where `name` and `surname` are IDs of respective data fields. This dot-notation allows the language to unambiguously differentiate between core system properties and dynamic, user-defined process elements, while the complete tokenization and escape character rules are fully managed by the underlying ANTLR v4 grammar available in the repository.

Iterating over a large amount of resources can be a performance heavy task that will require paging. The *paging* clause define a page number (starting at zero) and optional size of a single page. Both values are non-negative integers.

```
paging      = 'page' page_number [ 'size' page_size ] ;
page_number = ? non-negative integer ? ;
page_size   = ? non-negative integer ? ;
```

Similar to SQL we provide sorting capabilities using the *sort by* following by the name of attribute resources should be sorted by and optional order. The order can be either ascending - *asc*, or descending - *desc*. If not provided the default sorting order is ascending.

```
sorting  = 'sort by' ordering { ',' ordering } ;
ordering = attribute [ order ] ;
order    = 'asc' | 'desc' ;
```

4.1. Resource attributes

Petriflow resources have different data attributes with different data types. In the final PFQL EBNF we need to define different types of predicates, paging, and sorting, for each resource type. In table 2 we provide an overview of all supported operators for all attribute types.

Process resources have following attributes (and types): *id* (ObjectId), *identifier* (String), *version* (Version), *title* (String), and *creationDate* (DateTime).

Case resources have following attributes: *id* (ObjectId), *processId* (ObjectId), *processIdentifier* (String), *title* (String), *creationDate* (DateTime), authorId (ObjectId), list of *places* with *marking* (Number), list of *tasks* with *state* (State) and *userId* (ObjectId), and list of process *data* attributes with *value* (ObjectId, ObjectId[], String, String[], Number, Date, DateTime, or Boolean) and *options* (String[]).

Type	eq	neq	lt	gt	lte	gte	like	in range	in list
ObjectId	✓	✓							✓
ObjectId[]	✓	✓							✓
String	✓	✓	✓	✓	✓	✓	✓	✓	✓
String[]	✓	✓							✓
Number	✓	✓	✓	✓	✓	✓		✓	✓
Date	✓	✓	✓	✓	✓	✓		✓	✓
DateTime	✓	✓	✓	✓	✓	✓		✓	✓
Boolean	✓	✓							
State	✓	✓							

Table 2
List of data types and supported operators

Task resources have following attributes: *id* (ObjectId), *transitionId* (String), *title* (String), *state* (State), *userId* (ObjectId), *caseId* (ObjectId), and *processId* (ObjectId).

User resources have following attributes: *id* (ObjectId), *name* (String), *surname* (String), *email* (String), and *roles* (ObjectId[]).

Role resources have following attributes: *id* (ObjectId) and *title* (String).

4.2. PFQL query of motivation example

Using the PFQL and our implemented search service, we can demonstrate the action triggered by the `Get requests` button (see Figure 2). The implementation performs two main steps. First, it searches for instances of the `request` process where the *name* and *surname* attributes match those of the current *Bulk approval* case. Second, it retrieves the corresponding *Approve request* tasks (with ID equals *approve_request*) for the identified cases. The resulting script is implemented as follows:

```
def children_request_cases = searchCases(
    "processIdentifier == 'request'
    & data.name == '${name.value}'
    & data.surname == '${surname.value}'"
)?.collect{it.stringId}

change taskRef_0 value { searchTasks(
    "caseId in ${children_request_cases}
    & transitionId == 'approve_request'"
)?.collect{it.stringId}}
```

4.3. Operational Semantics by Translation

Finally, a formal clarification regarding the execution semantics of PFQL is necessary. To ensure the language specification is transparent and to prevent ambiguous interpretations, it should be noted that a rigorous, purely mathematical (algebraic or denotational) semantics is not detailed within the scope of this paper.

However, because PFQL is structurally designed to be compiled and mapped directly into existing target query languages (specifically MongoDB QueryDSL and Elasticsearch Query String), its operational semantics are implicitly and deterministically defined by this translation process. This translational approach guarantees that the behavior of any PFQL predicate is securely bound to the well-defined, industry-proven execution models of the underlying production databases.

This translational framework strictly resolves the behavior of singular versus plural query formulations. When a query uses a plural resource declaration (such as `search cases` or `search tasks`), the underlying compiler translates the expression into a standard collection-fetch operation, returning a list of all matching documents. Aggregation functions applied to a plural context behave intuitively:

`count` returns the total number of matched items, while `exists` returns a boolean value indicating whether the result set is non-empty.

Conversely, a singular resource declaration (such as `search case` or `search task`) alters the execution mechanics at the database layer. Rather than causing a runtime failure if multiple documents match the predicates, a singular query is deterministically optimized to fetch a single object. Consequently, a singular query always evaluates the collection safely, returning either the first structurally matched instance or an empty result if no match is found. When aggregation functions are used in a singular context, `count` is logically bounded to evaluate to either `0` or `1`, and `exists` remains functionally identical to its plural counterpart.

A fully independent, formal mathematical formalization of the language semantics remains a subject for future research.

5. Implementation

The Netgrif Platform was designed for modelling and execution of object-centric processes (see proceedings of international workshops on object-centric processes in [14, 15]) in a low-code language Petriflow [11]. It uses Netgrif Application Engine (NAE) to interpret and execute Petriflow processes. The NAE backend is written in Java using the Spring Boot framework and stores the data of Petriflow processes, their instances, tasks of process instances, users, and roles in MongoDB as main database and Elasticsearch as a search engine.

MongoDB has a limited number of indexes and only certain metadata of Petriflow resources are indexed in the MongoDB. Data attributes of Petriflow object-centric processes marked as indexed are then indexed in Elasticsearch. This means that developers need to know which database to search in which cases.

The PFQL was implemented as part of Netgrif Platform to address these issues. As you see, syntax of the PFQL is inspired by Elasticsearch queries, which are easier to read compared with QueryDSL. Abstract search service was implemented using the *ANother Tool for Language Recognition* (ANTLR) to generate a parser for the PFQL EBNF.

Each query is parsed into Java objects and analysed to determine which database should be queried. Then the query is transformed into the specific query language of the database and corresponding service is used to evaluate that query. If all queried attributes are indexed in MongoDB then the query is transformed into a MongoDB query. Otherwise the query is transformed into an Elasticsearch query. Security context is also added according to the role-based access control. This ensures that users can access only those resources that they have permission to access.

The search service provides functionality for all PFQL statements search, count, and exists. It also provides a function to explain a query using the parse tree, which can help to understand which database is queried and why. Explanation of the previous query:

```
Searching single instance of resource CASE with Elasticsearch
page number: 0, size: 20
-- AND
   -- processIdentifier == 'request'
   -- data.name.value == 'John'
   -- data.surname.value == 'Doe'
```

The service also provides similar explanation when an error occurs during parsing. For example the error output if instead of *'eq'* we put *'qe'* will be:

```
no viable alternative at input 'processIdentifier qe'
case: processIdentifier qe 'request' and data.name.value == 'John' ...
                        ^^
mismatched input 'qe' expecting {<EOF>, PAGE, SORT_BY}
```

```
case: processIdentifier qe 'request' and data.name1.value == 'John' ...
                         ^^
```

Providing explanation of a query and error output is essential for a wide adoption of the platform as searching instances and tasks is necessary even in small applications with a few processes.

5.1. Filters

Query language is not only usable in development of processes but also in creating filters by end users of the platform. These filters can be easily constructed in the graphical user interface (GUI) as seen in Figure 4 where the previous example query is being constructed. The filter building component guides users step-by-step and in each step provides options in the form of a drop-down menu. In the first step users select attribute for the predicate. In the second step users select an operator from the list of available operators for the given data type of the selected attribute. In the last step users provide a value they want to test against. Using the *"+ AND"* and *"+ OR"* buttons it is possible to chain multiple predicates. Each part of the predicate can be changed at any time by simply clicking on it.

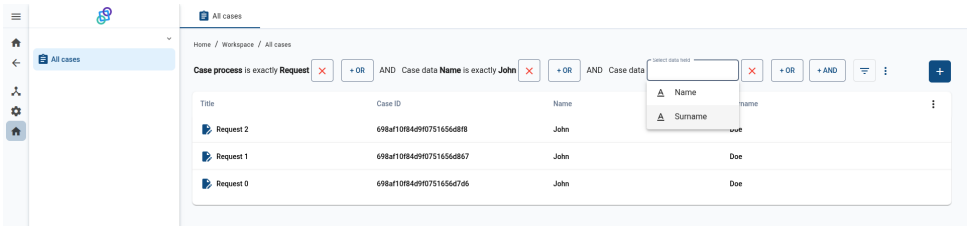

Figure 4: Example of a query language used in custom filter in the user interface of Netgrif Platform.

Filters and PFQL can be also used in processes in data attributes of types task reference (*taskRef*) and case reference (*caseRef*). For example in a task of bulk approval we can specify that only cases waiting for approval and belonging to the users department can be selected into a *caseRef* field.

6. Conclusion and Future Work

In this paper we present the motivation behind the Petriflow Query Language, definition of the main concepts using its simplified EBNF, and showcase its application in the Netgrif Platform using an example query. The Petriflow Query Language not only provides a unified query interface but also takes away the burden of learning different data structures and implementation details of underlying databases, such as MongoDB, and search engines like Elasticsearch.

While a comprehensive, isolated empirical evaluation of query execution runtimes and a quantitative expressiveness benchmark remain outside the scope of this work, the architectural necessity of PFQL is strongly validated by extensive empirical evidence from long-term enterprise deployments. The underlying Netgrif platform has been successfully running in production environments for several years, managing high-throughput systems containing over 100 million process instances (*cases*) for various clients. Initially, the platform relied solely on a monolithic MongoDB persistence layer. However, as data volumes scaled, proprietary full-text searches and complex analytical queries grew computationally unsustainable under standard MongoDB QueryDSL. This practical bottleneck necessitated the evolution of the infrastructure into the current hybrid model by integrating Elasticsearch to offload full-text indexing and real-time analytical evaluation.

This specific production history highlights the exact engineering challenge that PFQL addresses: without a unified abstraction layer, shifting from a pure MongoDB backend to a hybrid MongoDB-Elasticsearch architecture required a complete, manual rewrite of all application-level queries. PFQL successfully decouples developers from these infrastructure shifts, maintaining stable readability and expressiveness regardless of the underlying storage engine's performance limits.

Crucially, this abstraction does not introduce any performance overhead during execution. While routing a query within a hybrid backend traditionally adds analytical latency, the compilation target in PFQL is determined strictly by the query's structural properties rather than its runtime values. Consequently, both the routing decision and the actual translation into native database syntax are performed ahead-of-time during the deployment and parsing of the Petriflow XML process definition. At runtime, the platform directly executes the pre-compiled, native MongoDB or Elasticsearch queries, ensuring that the unified language layer guarantees zero compilation overhead during active user transactions.

To fully leverage this abstraction in production environments and enable seamless integration with any backend, our primary future **engineering objective** is the design and implementation of the *Petriflow Persistence API*. This unified API layer will natively incorporate the compiler components responsible for translating PFQL expressions into various target persistence languages, including MongoDB QueryDSL, Elasticsearch Query String, standard SQL, and distributed NoSQL systems like Cassandra. Furthermore, this API will encompass critical runtime capabilities that extend beyond the query language itself, such as comprehensive transaction management across heterogeneous data sources, providing a fully pluggable and database-agnostic persistence framework for the Petriflow ecosystem. Consequently, we plan to conduct a structured empirical study focusing on precise syntactic verbosity metrics, query formulation times by domain experts, and micro-benchmark runtime comparisons across different database configurations.

Addressing the current limitations provides also a clear roadmap for future research regarding **formal and expressiveness objectives**.

Although the current operational semantics are implicitly and deterministically defined via compilation into target languages, developing a fully independent, **formal** mathematical (algebraic or denotational) formalization of the language semantics remains a primary objective for future theoretical work.

To enhance **expressiveness**, future iterations will extend the language to cover more complex process model querying and the execution history of process instances. This extension will support advanced temporal logic, allowing users to query historical process evolution, state transitions over time, and reachability properties directly within standard PFQL expressions, fully leveraging the underlying Petri net execution models.

Petriflow language defines events that create a new instance of a process, assign and finish tasks, and change value of data attributes. Each occurrence of an event in Netgrif Platform is logged and stored in an event log collection. These event logs can be used as a source for business intelligence reports and dashboards by developers. Creating reports and dashboards is currently not a simple task and requires knowledge of Elasticsearch query language and aggregations. Implementing the PFQL for querying event logs could simplify this task and make it available for a wider group of users of the platform with less technical knowledge of Elasticsearch.

In other words, the current implementation of the Petriflow Query Language (PFQL) provides a robust framework for state-based filtering, enabling users to query process instances based on their immediate data attributes and current marking. However, a significant potential for Petri net-based platforms lies in the ability to query not only the current state but the evolution of process instances over time.

Therefore, our future research and development roadmap for the expressiveness of Petriflow query language includes the following key areas:

Temporal Logic Extensions: We plan to extend PFQL with operators from Linear Temporal Logic (LTL). This will allow for complex behavioural queries, such as "Find all cases where an invoice was approved without a prior purchase order" or "Identify processes where a specific task is eventually followed by a notification."

Safety and Liveness Properties: By incorporating LTL and potentially Computation Tree Logic (CTL), we aim to provide built-in support for verifying safety (something bad never happens) and liveness (something good eventually happens) properties directly within the application engine.

Behavioural Compliance: This extension will transform Petriflow from a process execution engine

into a self-verifying system. It will enable on-the-fly compliance checks, ensuring that business cases adhere to formal specifications throughout their entire lifecycle.

Leveraging Petri Net Semantics [4]: Since Petriflow is grounded in formal Petri net theory, these temporal queries will be evaluated against the reachability or coverability graph, or by utilizing state-of-the-art model-checking techniques. To efficiently capture non-sequential behaviour, we plan to adopt causal semantics (such as partial-order [8] or stratified-order based Petri net unfoldings [7]), enabling the query engine to natively reason over various relations of causality and true concurrency, thereby ensuring mathematical precision in query results.

Acknowledgments

Funded by the EU NextGenerationEU through the Recovery and Resilience Plan for Slovakia under the project No. 09I03-03-V04-00493

Declaration on Generative AI

During the preparation of this manuscript, the authors used Gemini for language editing and proof-reading of single paragraphs to improve clarity and readability. The underlying ideas and arguments were made and written solely by the authors. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] C. Beeri, A. Eyal, S. Kamenkovich, T. Milo, Querying business processes with BP-QL, Information Systems 33 (2008) 477–507. URL: https://www.sciencedirect.com/science/article/pii/S0306437908000124. doi:https://doi.org/10.1016/j.is.2008.02.005.

[2] A. Awad, BPMN-Q: A language to query business processes, 2007, pp. 115–128.

[3] T. Vogelgesang, J. Ambrosy, D. Becher, R. Seilbeck, J. Geyer-Klingeberg, M. Klenk, Celonis PQL: A Query Language for Process Mining, Springer International Publishing, Cham, 2022, pp. 377–408. URL: https://doi.org/10.1007/978-3-030-92875-9_13. doi:10.1007/978-3-030-92875-9_13.

[4] J. Desel, G. Juhás, "What Is a Petri Net?" Informal Answers for the Informed Reader, Springer Berlin Heidelberg, Berlin, Heidelberg, 2001, pp. 1–25. URL: https://doi.org/10.1007/3-540-45541-8_1. doi:10.1007/3-540-45541-8_1.

[5] J. Desel, Process Modeling Using Petri Nets, in: Process-Aware Information Systems, Wiley, 2005, pp. 147–177.

[6] J. Desel, W. Reisig, The concepts of Petri nets, Softw. Syst. Model. 14 (2015) 669–683.

[7] M. A. Alqarni, R. Janicki, On Interval Process Semantics of Petri Nets with Inhibitor Arcs, in: Petri Nets, volume 9115 of *Lecture Notes in Computer Science*, Springer, 2015, pp. 77–97.

[8] W. Vogler, Partial order semantics and read arcs, Theor. Comput. Sci. 286 (2002) 33–63.

[9] W. M. P. van der Aalst, K. M. van Hee, A. H. M. ter Hofstede, N. Sidorova, H. M. W. Verbeek, M. Voorhoeve, M. T. Wynn, Soundness of Workflow Nets with Reset Arcs, Trans. Petri Nets Other Model. Concurr. 3 (2009) 50–70.

[10] R. Valk, Self-Modifying Nets, a Natural Extension of Petri Nets, in: ICALP, volume 62 of *Lecture Notes in Computer Science*, Springer, 1978, pp. 464–476.

[11] G. Juhás, T. Kováčik, J. Kovář, M. Kranec, L. Petrovič, Petriflow language and Netgrif Application Builder 2973 (2021) 171–175.

[12] Netgrif, Petriflow Language XML Schema, https://github.com/petriflow/petriflow/blob/main/petriflow.schema.xsd, 2024.

[13] Netgrif, ANTLR v4 Grammar Specification, https://github.com/netgrif/application-engine/blob/NAE-2443/src/main/java/com/netgrif/application/engine/pfql/domain/antlr4/QueryLang.g4, 2024.

[14] K. Gdowska, M. T. Gómez-López, J.-R. Rehse (Eds.), Business process management workshops, Lecture Notes in Business Information Processing, Springer Nature Switzerland, Cham, 2025.
[15] J. De Weerdt, L. Pufahl (Eds.), Business process management workshops, Lecture Notes in Business Information Processing, Springer Nature Switzerland, Cham, 2024.