Procedural Query Language (PQL)

This document describes the basic concepts of the Procedural Query Language (PQL) of Odysseus and shows how to use the language. In contrast to languages SQL based languages like the Continuous Query Language (CQL) or StreamSQL, PQL is more procedural and functional than declarative. This document shows how to formulate queries with PQL.

- 1 Using PQL in Queries
  - 1.1 Define an Operator
  - 1.2 Intermediate Names, Views and Sources
  - 1.3 Parameters – Configure an Operator
  - 1.4 Ports – What if the Operator Has More Than One Output?
  - 1.5 The Full Grammar of PQL
- 2 List of available PQL Operators
  - 2.1 Machine Learning / Data Mining

Using PQL in Queries

PQL is an operator based language where an operator can be seen as a logical building block of the query. Thus, PQL is the connection of several operators. Since Odysseus differentiates between logical operators and their physical operators, which are the implementing counterpart, PQL is based upon logical operators. Therefore, it may happen that the query gets changed during the transformation from the logical query plan into the physical query plan. This includes also logical optimization techniques like the restructuring of the logical query plan. To avoid this, you can explicitly turn off the query optimization.

Define an Operator

An operator can be used in PQL via its name and some optional settings, which can be compared with a function and the variables for the function:

```
OPERATORNAME(parameter, operator, operator, ...)
```

The first variable (parameter) describes operator dependent parameters and is used for configuring the operator. Note, that there is only one parameter variable! The other variables (operator) are input operators, which are the preceding operators that push their data into this operator. The inputs of an operator can be directly defined by the definition of another operator:

```
OPERATOR1(parameter1, OPERATOR2(Parameter2, OPERATOR3(...)))
```

Except for source operators (usually the first operator of a query) each operator should have at least one input operator. Thus, the operator can only have parameters:

```
OPERATOR1(parameter1)
```

Accordingly, the operator may only have input operators but no parameters:

```
OPERATOR1(OPERATOR2(OPERATOR3(...)))
```

Alternatively, if the operator has neither parameters nor input operators, the operator only exists of its name (without any brackets!), so just:

```
OPERATORNAME
```

It is also possible to combine all kinds of definitions, for example:

```
OPERATOR1(OPERATOR2(Parameter2, OPERATOR3))
```

Intermediate Names, Views and Sources

Since the nesting of operators may lead to an unreadable code, it is possible to name operators to reuse intermediate result. This is done via the "=" symbol. Thus, we can temporary save parts of the query, for example **(it is important to place blanks before and after the "=" symbol!)** :

```
Result2 = OPERATOR2(Parameter2, OPERATOR3)
```

The defined names can be used like operators, so that we can insert them as the input for another operator, for example:

```
Result2 = OPERATOR2(Parameter2, OPERATOR3)OPERATOR1(Result2)
```

There could be also more than one intermediate result, if they have different names:

```
Result1 = OPERATOR1(Parameter1, …)Result2 = OPERATOR2(Parameter2,
Result1)Result3 = OPERATOR3(Parameter3, Result2)
```

And you can use the intermediate name more than one time, e.g. if there are two or more operators that should get the same preceding operator:

```
Result1 = OPERATOR1(Parameter1, …)OPERATOR2(Parameter2, Result1)OPERATOR3
(Parameter3, Result1)
```

All intermediate results that are defined via the "=" are only valid within the query. Thus, they are lost after the query is parsed and runs. This can be avoided with views.
A **view** is defined like the previous described intermediate results but uses ":=" instead of "=", e.g.:

```
Result2 := OPERATOR2(Parameter2, OPERATOR3)
```

Such a definition creates an entry into the data dictionary, so that the view is globally accessible and can be also used in other query languages like CQL.
Alternatively, the result of an operator can also be stored as a **source** into the data dictionary by using "::="

```
Result2 ::= OPERATOR2(Parameter2, OPERATOR3)
```

The difference between a view and a source is the kind of query plan that is saved into the data dictionary and is reused. If a view is defined, the result of the operator is saved as a logical query plan, which exists of logical operators. Thus, if another query uses the view, the logical operators are fetched from the data dictionary and build the lower part of the new operator plan or query. If an operator is saved as a source, the result of the operator is saved as a physical query plan, which exists of already transformed and maybe optimized physical operators. Thus, reusing a source is like a manually query sharing where parts of two or more different queries are used together. Additionally, the part of the source is not recognized if the new part of the query that uses the source is optimized. In contrast, the logical query plan that is used via the a view is recognized, but will not compulsorily lead to a query sharing.
Finally, all possibilities gives the following structure:

```
QUERY = (TEMPORARYSTREAM | VIEW | SHAREDSTREAM)+
TEMPORARYSTREAM = STROM "=" OPERATOR
VIEW = VIEWNAME ":=" OPERATOR
SHAREDSTREAM = SOURCENAME "::=" OPERATOR
```

Parameters – Configure an Operator

As mentioned before, the definition of an operator can contain a parameter. More precisely, the parameter is a list of parameters and is encapsulated via two curly brackets:

```
OPERATOR({parameter1, paramter2, …}, operatorinput)
```

A parameter itself exists of a name and a value that are defined via a "=". For example, if we have the parameter port and want to set this parameter to the 1234, we use the following definition:

```
OPERATOR({port=1234}, …)
```

The value can be one of the following simple types:

- Integer or long: `OPERATOR({port=1234}, …)`
- Double: `OPERATOR({possibility=0.453}, …)`
- String: `OPERATOR({host='localhost'}, …)`

Furthermore, there are also some complex types:

- Predicate: A predicate is normally an expression that can be evaluated and returns either true or false. In most cases a predicate is simple a string, e.g.:

  ```
  OPERATOR({predicate='1<1234'}, …)
  ```

  Hint: In some cases the predicate must be in this form PREDICATE_TYPE('1<1234'), where PREDICATE_TYPE can be something like RelationalPredicate.

- List:It is also possible to pass a list of values. For that, the values have to be surrounded with squared brackets:

  ```
  OPERATOR({color=['green', 'red', 'blue']}, …)
  ```
  (Type of elements: integer, double, string, predicate, list, map).

- Map: This one allows maps like the HashMap in Java. Thus, one parameter can have a list of key-value pairs, where the key and the value are one of the described type. So, you can use this, to define a set of pairs where the key and the value are strings using the "=" for separating the key from the value:

`OPERATOR({def=['left'='green', 'right'='blue']}, …)`
It is also possible that values are lists:
`OPERATOR({def=['left'=['green','red'],'right'=['blue']]}, …)`
Remember, although the key can be another data type than the value, all keys must have the same data type and all values must have the same data type
Notice, that all parameters and their types (string or integer or list or…) are defined by their operator. Therefore, maybe it is not guaranteed that the same parameters of different operators use the same parameter declaration – although we aim to uniform all parameters.

Ports – What if the Operator Has More Than One Output?

There are some operators that have more than one output. Each output is provided via a port. The default port is 0, the second one is 1 etc. The Route for example, allows to split a stream according to predefined predicates to different output ports. So, if you want to use another port, you can prepend the port number with a colon in front of the operator. For example, if you want the second output (port 1) of the select:

```
PROJECT({…}, 1:SELECT({predicate='1<x'}, …))
```

The Full Grammar of PQL

```
QUERY           = (TEMPORARYSTREAM | VIEW | SHAREDSTREAM)+
TEMPORARYSTREAM = STREAM "=" OPERATOR
VIEW            = VIEWNAME ":=" OPERATOR
SHAREDSTREAM    = SOURCENAME "::=" OPERATOR
OPERATOR        = QUERY | [OUTPUTPORT ":"] OPERATORTYPE "(" (PARAMETERLIST [
"," OPERATORLIST ] | OPERATORLIST) ")"
OPERATORLIST    = [ OPERATOR ("," OPERATOR)* ]
PARAMETERLIST   = "{" PARAMETER ("," PARAMETER)* "}"
PARAMETER       = NAME "=" PARAMETERVALUE
PARAMETERVALUE  = LONG | DOUBLE | STRING | PREDICATE | LIST | MAP
LIST            = "[" [PARAMETERVALUE ("," PARAMETERVALUE)*] "]"
MAP             = "[" [MAPENTRY ("," MAPENTRY*] "]"
MAPENTRY        = PARAMETERVALUE "=" PARAMETERVALUE
STRING          = "'" [~']* "'"
PREDICATE       = PREDICATETYPE "(" STRING ")"
```

List of available PQL Operators

Odysseus has a wide range of operators build in and are explained here.

- Base operators
  - Aggregate (and Group) operator
  - Difference operator
  - Distinct operator
  - Existence operator
  - FastMedian
  - Filter operator
  - Intersection operator
  - Join operator
  - LeftJoin operator
  - Map operator
  - Merge operator
  - Project operator
  - Rename operator
  - Select operator
  - Sort operator
  - StateMap operator
  - Synchronize operator
  - TupleAggregate
  - Union operator
  - Window operator
- Advanced operators
  - AssociativeStorage operator
  - BufferedFilter operator
  - Coalesce operator
  - Combine
  - Command Operator
  - Convolution operator
  - TOPK
  - UDO operator
- Source operators
  - Access operator

    - DatabaseSource operator
    - QuerySource
    - Receive operator
    - Retrieve operator
    - Stream operator
- Sink operators
    - DatabaseSink operator
    - Sender operator
    - Sink operator
    - Store operator
- Database operators
    - DatabaseSink
    - DatabaseSource
- Enrich operators
    - ContextEnrich operator
    - DBEnrich operator
    - Enrich operator
    - WSEnrich operator
    - XmlRpcEnrich
- Pattern operators
    - ChangeCorrelate operator
    - ChangeDetect operator
    - Pattern operator
    - SASE operator
- Mining operators
    - Classification_learn operator
    - Classify operator
    - Clustering operator
    - FrequentItemset operator
    - GenerateRules operator
    - SentimentDetection operator
- Anomaly Detection Operators
    - DeviationAnomalyDetection operator
    - DeviationLearn operator
    - DeviationSequenceAnomalyDetection operator
    - DeviationSequenceLearn operator
    - FrequencyCompare operator
    - LOFAnomalyDetection operator
    - RareSequence operator
    - ValueAnomalyDetection operator
- Recommender System operators
- Probabilistic operators
    - EM operator
    - ExistenceToPayload operator
    - KalmanFilter operator
    - KDE operator
    - LinearRegressionMerge operator
    - LinearRegression operator
    - SampleFrom operator
- Order operators
    - ReOrder operator
    - TimestampOrderValidate Operator
- Plan operators
    - AppendTo operator
    - Close Stream Operator
- Processing operators
    - BloomFilter Operator
    - Buffer operator
    - Cache operator
    - Heartbeat operator
    - Metadata Operator
    - NewFilenamePunctuation
    - Replacement operator
    - Route operator
    - Sample operator
    - Synchronize
    - SyncWithSystemTime operator
    - Timeshift operator
- Transform operators
    - Converter operator
    - TimestampToPayload operator
    - ToKeyValue operator
    - ToTuple operator
    - UnNest operator
- Benchmark operators
    - CalcLatency operator
    - LatencyToPayload operator
    - Sleep Operator
    - Systemload Operator

Machine Learning / Data Mining

Available mining or machine learning operators are described here: Machine Learning