Celonis PQL: A Query Language for Process Mining

Authors:

Thomas Vogelgesang, Jessica Kaufmann, David Becher, Robert Seilbeck, Jerome Geyer-Klingeberg, Martin Klenk

Editor:

Artem Polyvyanyy

Chapter from:
Process Querying Methods

Acknowledgments

This chapter will be published in the upcoming book “Process Querying Methods”
by Artem Polyvyanyy, Springer Nature Switzerland AG, 2020.

Abstract

Process mining studies data-driven methods to discover, enhance and monitor business processes by gathering knowledge from event logs recorded by modern IT systems. To gain valuable process insights, it is essential for process mining users to formalize their process questions as executable queries. For this purpose, we present the Celonis Process Query Language (Celonis PQL), which is a domain-specific language tailored towards a special process data model and designed for business users. It translates process-related business questions into queries and executes them on a custom-built query engine. Celonis PQL covers a broad set of more than 150 operators, ranging from process-specific functions to machine learning and mathematical operators. Its syntax is inspired by SQL, but specialized for process-related queries. In addition, we present practical use cases and real-world applications, which demonstrate the expressiveness of the language and how business users can apply it to discover, enhance and monitor business processes. The maturity and feasibility of Celonis PQL is shown by thousands of users from different industries, who apply it to various process types and huge amounts of event data every day.

Thomas Vogelgesang
Celonis SE, Munich, Germany

Jessica Kaufmann
Celonis SE, Munich, Germany

David Becher
Celonis SE, Munich, Germany

Robert Seilbeck
Celonis SE, Munich, Germany

Jerome Geyer-Klingeberg (corresponding author)
Celonis SE, Munich, Germany, e-mail: j.geyerklingeberg@celonis.com

Martin Klenk
Celonis SE, Munich, Germany

1 Introduction

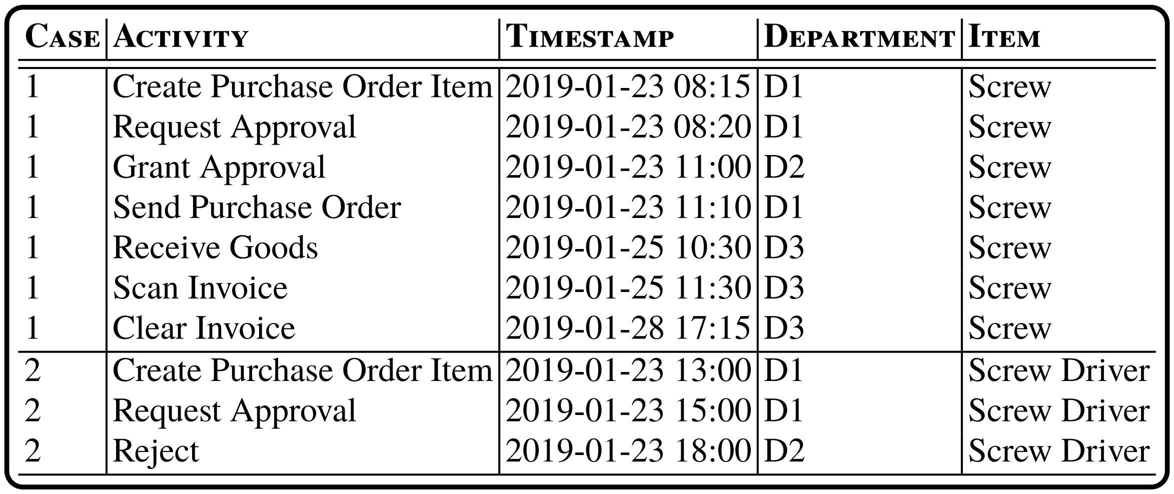
Process mining is a data-driven approach to reconstruct, analyze and improve business processes using log data recorded by modern IT systems, like enterprise resource planning (ERP) or customer relationship management (CRM) systems [15]. The starting point of process mining is an event log, which is a record of what happens when a business process is performed. Process mining applies special algorithms on the event log data to uncover the actual process execution, to reveal undesired process behavior and to identify inefficiencies [15]. Typical business use cases for process mining are procurement (e.g. purchase-to-pay), sales (e.g. order-to-cash), accounting (e.g. accounts payable) or production (e.g. make-to-order).

Due to their strong ability to provide transparency across complex business processes, process mining capabilities have been adopted by many software vendors and academic tools. A key success factor for any process mining tool is the ability to translate business questions into executable process queries and to make the query results accessible to the user. To this end, we developed Celonis Process Query Language (Celonis PQL). It takes the input from the user and executes the queries in a custom-built query engine. This allows the users to analyze all facets of a business process in detail, as well as to detect and employ process improvements. Celonis PQL is a comprehensive query language that consists of more than 150 (process) operators. The language design is strongly inspired by the requirements of business users. Therefore, Celonis PQL achieved a wide adoption by thousands of users across various industries and process types.

This chapter is organized as follows: Section 2 provides the background knowledge, which is required to understand the specifics of our process query language. Section 3 gives an overview of the various application scenarios of Celonis PQL. Section 4 presents the query language, its syntax and operators. Section 5 demonstrates the applicability of Celonis PQL to solve widespread business problems. Section 6 outlines the implementation of the query language. Section 7 positions Celonis PQL within the Process Querying Framework (PQF). Finally, Section 8 concludes the chapter.

2 Background

In this section, we introduce the general concept of process mining and how the Celonis software architecture enables process mining through our query language. We also present the history of Celonis PQL as well as the design goals that were considered during the development of the query language.



Case	Activity	Timestamp	Department	Item
1	Create Purchase Order Item	2019-01-23 08:15	D1	Screw
1	Request Approval	2019-01-23 08:20	D1	Screw
1	Grant Approval	2019-01-23 11:00	D2	Screw
1	Send Purchase Order	2019-01-23 11:10	D1	Screw
1	Receive Goods	2019-01-25 10:30	D3	Screw
1	Scan Invoice	2019-01-25 11:30	D3	Screw
1	Clear Invoice	2019-01-28 17:15	D3	Screw
2	Create Purchase Order Item	2019-01-23 13:00	D1	Screw Driver
2	Request Approval	2019-01-23 15:00	D1	Screw Driver
2	Reject	2019-01-23 18:00	D2	Screw Driver

Fig. 1: Example event log

2.1 Process Mining

In the course of digitization, an increasing number of log data is recorded in IT systems of companies worldwide. This data is of high value, as it represents how processes are running inside a company. Process mining technology can be applied to such data to gain insights about business processes, discover inefficiencies, and find potential improvements.

Process mining is based on *event logs*. An event log is a collection of *events*. An event is described by a number of attributes. The following three event attributes are always required for process mining:

Case.

The case attribute indicates which process instance the event belongs to. A process instance is called a *case*. A case usually consists of multiple events.

Activity.

The activity attribute describes the action that is captured by the event.

Timestamp.

The timestamp indicates the time when the event took place.

A sequence of events, ordered by their timestamps, that belong to the same case is called a trace. The *traces* of all the different cases with the same activity sequence represent a *variant*. The *throughput time* between two events of a case is the time difference between the corresponding timestamps. Accordingly, the throughput time of a case is equal to the throughput time between the first and the last event of the corresponding trace.

Figure 1 shows an example event log in procurement. Each case represents a process instance of one purchase order. In the first case, the order item is created in the system, an approval for purchasing is requested and approved. After the approval, the order is sent to the vendor. Two days later, the ordered goods are received, the invoice is registered and eventually paid. In the second case, an order item is created, but the approval to actually order it is rejected. Besides the three required attributes of an event log mentioned above, the example also includes attribute **Department**, which specifies the executing department for each event, as well as attribute **Item** containing the description of the corresponding order item.

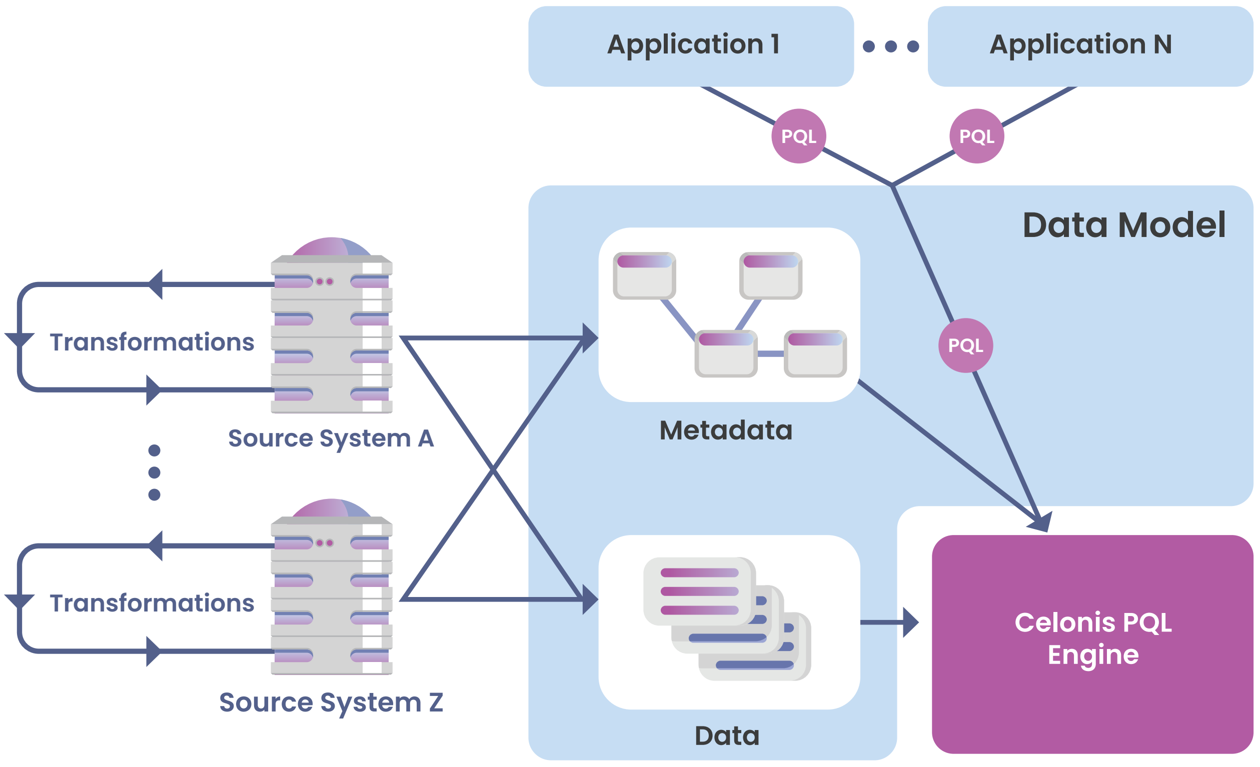


Fig. 2: Celonis architecture overview

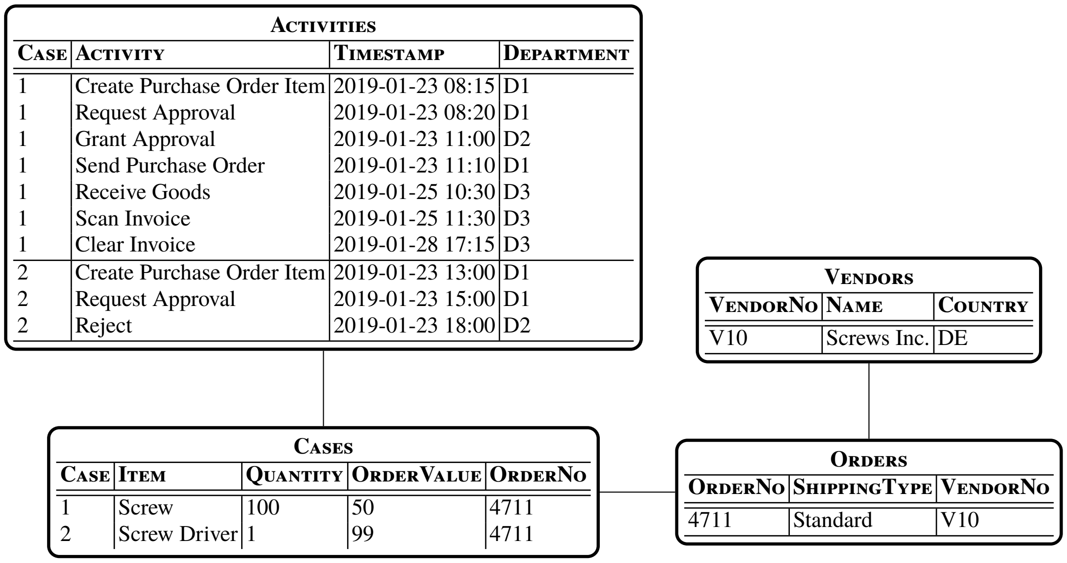
Process mining techniques that are applied on an event log to understand and improve the corresponding process can be assigned to three groups [15]: discovery, conformance and enhancement. Discovery uses the event log as input and generates a business process model as output. Conformance takes the event log and an a priori process model to detect discrepancies between the log data and the a priori model. Enhancement takes the event log and an a priori model to improve the model with the insights generated from the event log.

2.2 Architecture Overview

Celonis PQL is an integral component of the Celonis software architecture, which is shown in Figure 2. All Celonis applications use this language to query data from a *data model*. The data model contains metadata like schema information and the foreign key relationships between the tables, as well as the actual data from the source systems. Celonis PQL queries are evaluated by the Celonis PQL Engine.

Source system.

A source system is the system containing the business data to be analyzed by the Celonis applications. ERP systems like SAP, CRM systems like Salesforce, and many other standard systems are supported. Celonis applications can also connect to a variety of database systems on the customer's premises like PostgreSQL. It is also possible to upload Excel or CSV files. Data from multiple source systems can be combined in one data model.



Activities

Case	Activity	Timestamp	Department
1	Create Purchase Order Item	2019-01-23 08:15	D1
1	Request Approval	2019-01-23 08:20	D1
1	Grant Approval	2019-01-23 11:00	D2
1	Send Purchase Order	2019-01-23 11:10	D1
1	Receive Goods	2019-01-25 10:30	D3
1	Scan Invoice	2019-01-25 11:30	D3
1	Clear Invoice	2019-01-28 17:15	D3
2	Create Purchase Order Item	2019-01-23 13:00	D1
2	Request Approval	2019-01-23 15:00	D1
2	Reject	2019-01-23 18:00	D2

Vendors

VendorNo	Name	Country
V10	Screws Inc.	DE

Cases

Case	Item	Quantity	OrderValue	OrderNo
1	Screw	100	50	4711
2	Screw Driver	1	99	4711

Orders

OrderNo	ShippingType	VendorNo
4711	Standard	V10

Fig. 3: Example data model with four tables, including activity and case tables

Data model.

A data model combines all tables from the source system (or multiple source systems) which contain the data about a process that a user wants to analyze. In the data model, the foreign key relationships between the source tables can be defined. This is performed here because specifying joins is not part of the query language itself. The tables are arranged in a snowflake schema, which is common for data warehouses, and the schema is centered around explicit case and activity tables. Other data tables provide additional context. Figure 3 shows an example data model. It contains the event log of Figure 1 in the **Activities** table, including the **Department** column. It is linked to the **Cases** table, containing information about each order item. The **Item** attribute from the example event log of Figure 1 is contained in the **Cases** table, as the data model should contain normalized table schemas. Both order items (i.e. both cases) belong to the same purchase order which is sent to one vendor. Details about the purchase orders and the vendors are available in the **Orders** and **Vendors** tables of the data model.

Activity table.

The data model always contains an event log, which we call the *activity table*. The activity table always contains the three columns of the core event log attributes, while additional columns may be present. Within one case, the corresponding rows in the activity table are always sorted based on the timestamp column.

Usually, the activity table is not directly present in the source systems and therefore needs to be generated depending on the business process being analyzed. Since the source system is a relational database in most cases, this is usually done in SQL in the so-called transformation step. The transformation result can be a database view. However, a persisted table is usually created for performance reasons. This procedure is comparable to the extract, transform, load procedure (ETL) in data warehouses. Like all the other tables, the resulting activity table is then imported into the data model. The user can specify the case and activity table in a graphical user interface (GUI) and mark the corresponding columns of the activity table as the case, activity and timestamp columns.

Case table.

The case table provides information about each case and acts as the fact table in the snowflake schema. It always includes the case column, containing all distinct case IDs, while other columns provide additional information on the cases. There is a 1:N relationship between the case table and the activity table. If the case table is not specified in the data model, it will be generated automatically during the data model load. The case table then consists of one column containing all distinct case IDs from the activity table. This guarantees that a case table always exists, and the Celonis PQL functions and operators can rely on it.

Celonis PQL Engine.

Celonis PQL Engine is an analytical column-store main memory database system. It evaluates Celonis PQL queries over a defined data model. Section 6 describes the Celonis PQL Engine in more detail.

Applications.

Celonis applications provide a variety of tools for the business user to discover, monitor and enhance business processes. All applications use Celonis PQL to query the required data. They include easy-to-use GUIs, providing a convenient way for the users to interact with the process and business data. In the applications, the users can specify custom Celonis PQL queries. There are also many auto-generated queries sent by the applications to retrieve various information, which is then presented to the user in the graphical interface. An overview of the different applications that use Celonis PQL is given in Section 3.

2.3 History of Celonis PQL

The first version of Celonis PQL was introduced with version 2.4 of Celonis Process Mining, which was released in 2014. Celonis PQL was an extension to SQL, providing commands to query and filter process flows and patterns in addition to the standard SQL commands. For example, the process filter `process# START 'Activity XYZ'` could be used to filter all cases that start with activity `'Activity XYZ'`. The custom Celonis PQL commands were translated into standard SQL in the background and executed directly on the source database system. Multiple source systems like SAP HANA, MSSQL and Oracle were supported.

In SQL, different database systems support different SQL dialects. For example, function names and syntaxes for certain functionalities like date and time calculations are database specific. However, Celonis PQL was independent of the SQL dialect of the underlying database system. If necessary, Celonis PQL functions were mapped to equivalent SQL functions based on the appropriate dialect.

For version 4.0 of Celonis Process Mining, which was released in February 2016, the concept of Celonis PQL was fully redesigned based on the experiences gained from the first version of the language. Instead of being an extension to SQL, Celonis PQL became an independent language inspired by SQL. Now, Celonis PQL queries are executed on the custom-built Celonis PQL Engine, which is a query engine that is highly optimized for process mining capabilities. In comparison to the previous approach, this enables much better performance. Also, all functionalities are fully independent of the underlying database dialect, which makes it easier to support a wider range of source systems. Several patents were registered as part of the development efforts.

In October 2018, Celonis Intelligent Business Cloud (IBC) was released. This transition from an on-premises product to a modern native cloud solution provides easier access to process mining and, consequently, IBC increased the number of Celonis PQL users significantly. Many new applications, all using Celonis PQL to query process data, are included in IBC, as described in Section 3.

The language is continuously extended with new functionalities. This is mostly driven by customers who use Celonis PQL on a daily basis to explore their data. Due to the rich functionality and possibility to use it in various Celonis applications, Celonis PQL is used by a high number of users in many production systems.

2.4 Design Goals

The first version of Celonis PQL was an extension to SQL. While it had several convenient process mining functions, the actual queries evaluated on the database were complicated. Furthermore, it was difficult to extend the language with new functionality. To overcome these issues, the query language was redesigned based on previous experiences, with the following design goals in mind:

Simplicity.

The query language should be easy to use for business users. Providing an easy way to translate complex process questions into data queries should make process mining accessible for business users.

Flexibility.

The query language should not include specialized functions. Instead, the goal is to provide a set of generic functions and operators that can be combined in a wide range of queries. This flexibility is very important, since the users should be able to formulate all their questions in the query language, regardless of the processes they address.

Event log-centered.

In contrast to SQL, the language should be designed to support dedicated process mining functionality. This should be reflected in the query language by process functions, which operate on the given event log.

Business focus.

Event data can be augmented with additional business information. It is therefore important to combine process mining and business intelligence (BI) capabilities within one query language. To achieve this, besides specific process mining functionality, the query language should also provide a variety of functions known from SQL, like aggregations, string modifications and mathematical functions.

Frontend interaction.

To simplify the use of the query language, the user should be able to formulate queries with support of a GUI. Consequently, the goal is to design a language that provides easy integration via GUI components. The simple query creation using a GUI is a key factor for the usability of a product, which results in high acceptance, usage, and adoption by the users.

3 Applications

As a result of emerging technologies, the requirements on tools used for analyzing processes within different business departments go beyond the simple tracking of performance. For this reason, process mining at Celonis is evolving into a holistic approach that serves as a performance accelerator for business processes. Necessary steps that are included within this approach are the discovery, enhancement and monitoring of processes. Within discovery, process mining can capture digital footprints from all source systems involved in the process, visualize the respective processes and understand the root causes of deviations between the as-is process and the to-be process. Thus, the discovery step serves as the starting point for process improvements. During enhancement, process mining supports the automation of tasks, proposes intelligent actions and proactively drives process interventions and improvements. Monitoring allows the user to continuously track the development of key figures that are defined during the discovery step. This enables the ongoing benchmarking of processes – internally, as well as externally. Celonis PQL enables all these activities and tasks, and it is used in all Celonis products as depicted in Figure 4.

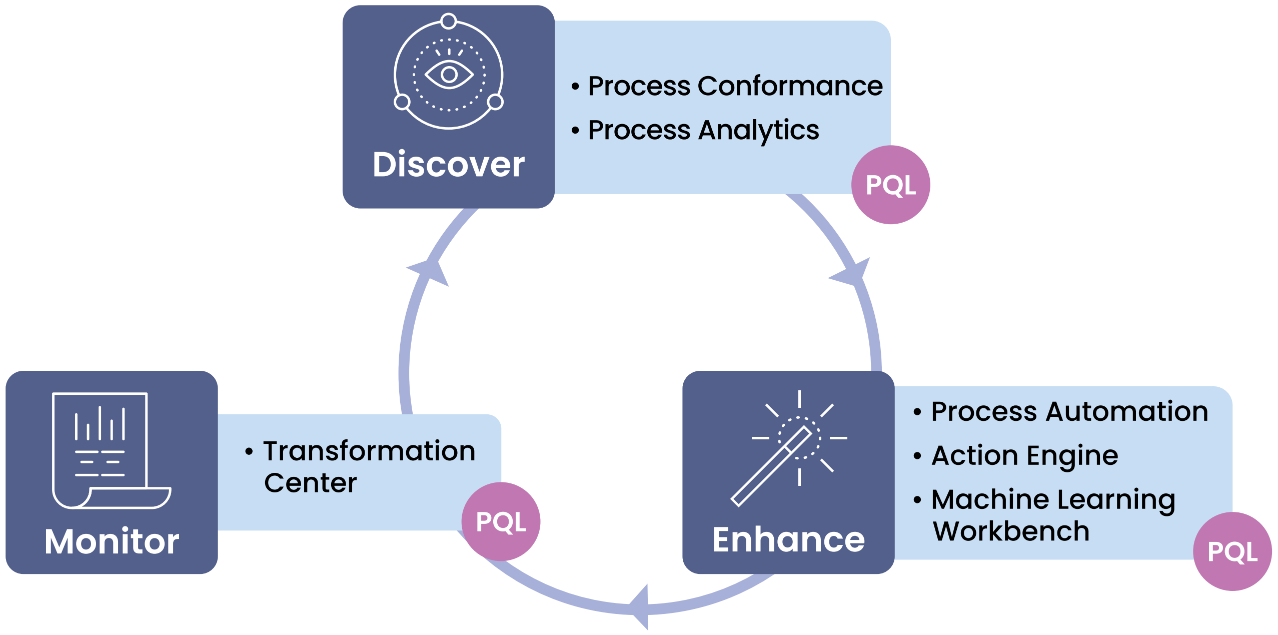


Fig. 4: The Discover, Enhance, Monitor approach

- **Process Analytics** is part of discovery and can be used to visualize and identify the root causes of issues within a process. Furthermore, it identifies the specific actions that have the greatest impact on solving the issue. For that purpose, Celonis PQL is used to obtain performance metrics, such as the average case duration, the change rate in the process or the degree of process automation. In addition, Celonis PQL enables the user to enrich the event log data with data related to the problematic cases, such as finding the vendor causing the issue, or identifying sub-processes that prolong throughput times.
- **Process Conformance** is, as shown in Figure 4, also part of the discovery step. It is used to identify deviations from the defined to-be process and to uncover the root causes of process deviations. In this context, Celonis PQL allows utilizing the calculated process conformance to obtain metrics leading to the discovery of root causes for deviations [16].
- **Action Engine** is part of the enhancement step and uses insights gained from the discovery step to recommend actions to improve process performance [1]. The foundation to generate these findings is Celonis PQL. The findings can include all the relevant information that impacts a decision about whether or how an action, like executing a task in the source system or triggering a bot, is performed. In addition, Celonis PQL assists in prioritizing necessary actions.
- **Machine Learning Workbench** enables the usage of Jupyter notebooks within Celonis IBC for creating and using Python predictive models. As part of the enhancement step, it supports building predictive models on process data to proactively avoid downstream friction like long running process steps, leading to e.g. late payments in finance. Celonis PQL is crucial for querying the process data necessary as input for the predictive models.
- **Process Automation** allows to automatically trigger specific actions in downstream applications, like SAP, Gmail, and Salesforce, based on predefined process conditions. Therefore, Process Automation is part of the enhancement step. In this context, Celonis PQL is used for process condition querying.

- **Transformation Center** is part of the monitoring step. It measures and monitors the progress of the process metrics defined within the discovery step. Therefore, it assists in achieving Key Performance Indicators (KPIs) and business outcomes. As with Process Analytics, Celonis PQL is used in this context to obtain the performance metrics by querying the underlying event log data and to enrich event log data with relevant business context.

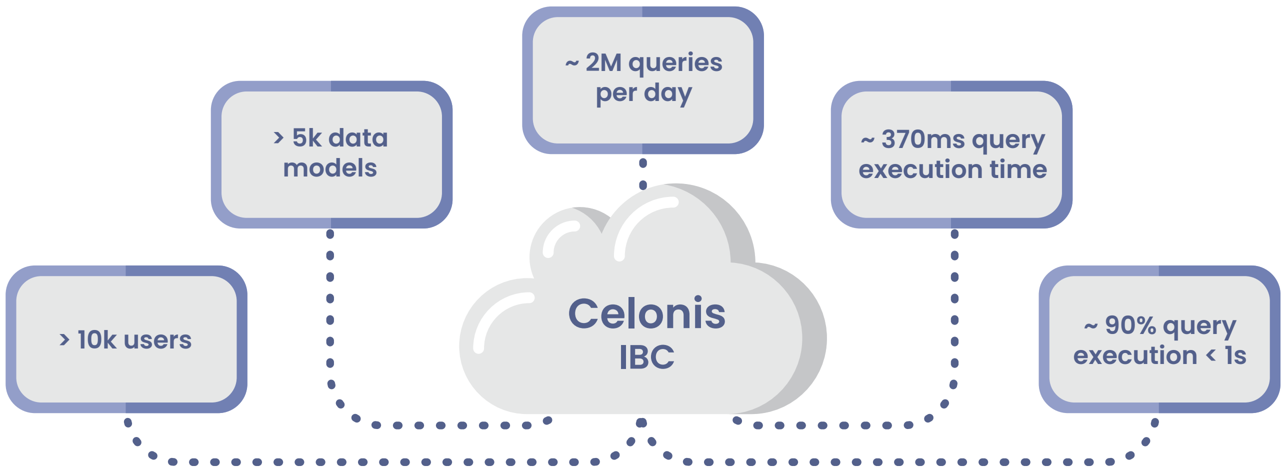


Fig. 5: Celonis IBC statistics (as of October 2019)

Celonis IBC enables customers to use the entire product range on-demand, containing all products described above. As shown in Figure 5, Celonis IBC is currently (as of October 2019) managing more than 10,000 users and more than 5,000 data models. Two million queries are executed per day, written in Celonis PQL, and processed by the Celonis PQL Engine with an average execution time of 370 milliseconds per query. In reality, the execution time per query is often much lower, but especially complex Celonis PQL statements on huge datasets lead to outliers in execution time. In around 90% of the cases, the execution time is less than a second. Nevertheless, we still aim for continuous performance improvements in order to further reduce the execution time of complex Celonis PQL statements, like heavily nested queries, in the future.

4 The Celonis Process Query Language

The intention of Celonis PQL is to provide a query language for performing process mining tasks on large amounts of event data. As described in Section 2.2, it is based on a relational data model. The event and business data as well as all results (including the mined process models) are represented as relational data.

Currently, the supported data types comprise STRING, INT, FLOAT, and DATE. Boolean values are not directly supported, but can be represented as integers. Each data type can hold NULL values. In general, Celonis PQL treats NULL values as non-existing and ignores them in aggregations. Also, row-wise operations like adding the values of two columns will return NULL if one of its inputs is NULL.

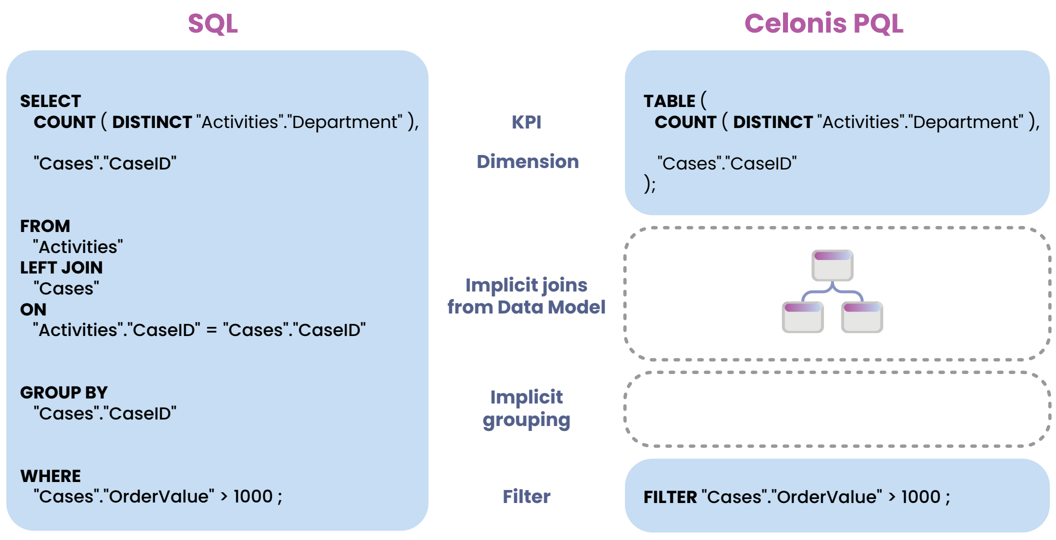
1 https://www.celonis.com/snap-signup/ (registration required)

Operators usually create and return a single column that is either added to an existing table (e.g. the case or activity table) or to a new, temporary result table. Only a few operators (e.g. for computing a process graph) create and return one or more tables with multiple columns. However, these operators are only used internally by GUI components and are not exposed to the end-user.

Currently, Celonis PQL provides more than 150 different operators to process the event data. Due to space limitations, we cannot sketch the full language. However, we can offer a brief overview of the major language features before we present selected examples to showcase the expressiveness of the language. Comprehensive documentation of the Celonis PQL operators can be accessed via the free process mining platform Celonis Snap[1].

4.1 Language Overview

Even though Celonis PQL is inspired by SQL, there are major differences between the two query languages. Figure 6 shows these differences by comparing how to query the cases and the number of involved departments for all orders with a value of more than 1000 euros in both languages. Furthermore, it also illustrates the key concepts of Celonis PQL.



```
SELECT
  COUNT ( DISTINCT "Activities"."Department" ),

  "Cases"."CaseID"


FROM
  "Activities"
LEFT JOIN
  "Cases"
ON
  "Activities"."CaseID" = "Cases"."CaseID"


GROUP BY
  "Cases"."CaseID"


WHERE
  "Cases"."OrderValue" > 1000 ;
```

```
TABLE (
  COUNT ( DISTINCT "Activities"."Department" ),

  "Cases"."CaseID"
);

FILTER "Cases"."OrderValue" > 1000 ;
```

Fig. 6: Comparing SQL and Celonis PQL by an example query

Similar to SQL, Celonis PQL enables the user to specify the data columns to retrieve from the data model. This can either be an aggregation, which we call a KPI, or an unaggregated column, which we call a dimension. While the data columns are part of the SELECT statement in SQL, Celonis PQL requires them to be wrapped in the TABLE operator, which combines the specified columns into a common table.

In contrast to SQL, Celonis PQL does not require the user to define how to join the different tables within the query. Instead, it implicitly joins the tables according to their foreign key relationships which have to be defined only once in the data model. Also, the grouping clause is not needed in

Celonis PQL as each selected column which is not aggregated (i.e. a dimension) is implicitly used as a grouper. According to the design goals, implicit joins and groupings significantly reduce the size and complexity of the queries and make it much simpler to formulate them.

Both languages offer the possibility to filter rows. While SQL requires the user to formulate the filter condition in the `WHERE` clause of the query, Celonis PQL offers the `FILTER` statements which are separated from the `TABLE` statements but executed together. Splitting the data selection and the filters into different statements enables the user to define multiple filter statements in different locations inside an application, which then can be combined into the table statement to query the data.

Beyond this simple structure, Celonis PQL provides a wide range of different operators which can be combined to answer complex business questions. The following list gives an overview of the most important classes of operators.

Aggregations.

Celonis PQL offers a wide range of aggregation functions, from simple standard functions like count and average, to more advanced aggregations like standard deviation and quantiles. Most of the aggregation functions are also available as window-based functions computing the aggregation not over all values but over a user-defined sliding element window.

Data functions.

These are operators like `REMAP_VALUES` (see Section 4.2) and `CASE WHEN` (see Section 5.1), which allow for conditional changes of values.

Date and time functions.

These functions enable the user to modify, project or round a date or time value, e.g. add a day to a date or extract the month from a timestamp. There are also functions to compute date and time difference (e.g. between timestamps of events).

Index functions.

Index functions create indices based on columns. The function `INDEX_ACTIVITY_LOOP`, for example, returns for each activity how many times it has occurred in direct succession in a case. This is useful, e.g., for identifying self-loops and computing their cycle lengths.

Machine learning functions.

There are various machine learning functions available, e.g., to cluster data using the k-means algorithm or learn decision trees.

Math functions.

Celonis PQL offers a wide range of mathematical functions, e.g., for arithmetic computations, rounding float numbers, and computing logarithms.

Predicates and logical operators.

For expressing complex filter conditions, Celonis PQL offers a variety of predicates (like range-based or pattern-based comparison) and standard boolean operators (AND, OR, and NOT).

Process functions.

Process functions comprise all process-specific functions which operate on the activity table and take its configuration into account. Examples are pattern-based process filters, SOURCE and TARGET operators (see Section 4.2), and computation of variants (see Section 4.3). There are also special process mining operators for discovering process models, clustering variants, and checking the conformance of a process model to the event data (see Section 4.4).

String modification.

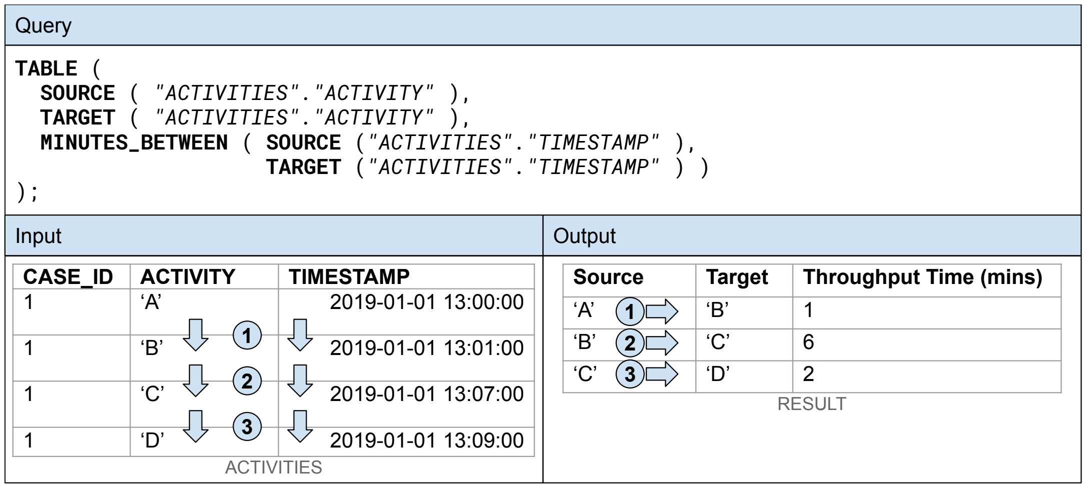
These functions enable the user to modify string values, e.g., trimming whitespaces, changing case, and creating substrings.

A major difference between SQL and Celonis PQL is the different language scope. Hence, Celonis PQL does not support all operators that are available in SQL. This is due to the fact that the development of the language is driven by customer requirements and only operators that are needed for the target use cases are implemented. For example, generic set operators like UNION are not supported, as they have not been required so far.

Another major difference to SQL is the missing support of a data manipulation language (DML). As all updates in the process mining scenario should come from the source systems, there is no need to directly manipulate and update the data through the query language. As the data can be considered to be read-only, this also allows for specific performance optimizations during implementation (see Section 6).

Furthermore, Celonis PQL does not provide any data definition language (DDL). As the data model is created by a visual data model editor and stored internally, there has not been any need for this so far.

In contrast to SQL, Celonis PQL is domain-specific and offers a wide range of process mining operators which are not available in SQL. Consequently, Celonis PQL seamlessly integrates the data with the process perspective. In the following, we explain selected process operators like SOURCE and TARGET (Section 4.2), VARIANT (Section 4.3), and CONFORMANCE (Section 4.4) in more detail.



```
TABLE (
  SOURCE ( "ACTIVITIES"."ACTIVITY" ),
  TARGET ( "ACTIVITIES"."ACTIVITY" ),
  MINUTES_BETWEEN ( SOURCE ("ACTIVITIES"."TIMESTAMP" ),
                    TARGET ("ACTIVITIES"."TIMESTAMP" ) )
);
```

CASE_ID	ACTIVITY	TIMESTAMP
1	'A'	2019-01-01 13:00:00
1	'B'	2019-01-01 13:01:00
1	'C'	2019-01-01 13:07:00
1	'D'	2019-01-01 13:09:00

Source	Target	Throughput Time (mins)
'A'	'B'	1
'B'	'C'	6
'C'	'D'	2

Fig. 7: Example of throughput time computation using SOURCE and TARGET operators

4.2 Source and Target Operators

In process mining applications it is often required to relate an event to another event which directly or eventually follows. For instance, this is required to compute the throughput time between two events by calculating the difference between the corresponding timestamps. Due to the relational data model, the timestamp values to subtract are stored in different rows. However, the operators (e.g. arithmetic operations) usually can only combine values from the same row. Therefore, we need a way to combine values from two different rows into the same row for performing such computations.

To overcome this issue, Celonis PQL relies on the SOURCE and TARGET operators. Figure 7 shows an example that illustrates how SOURCE and TARGET can be used to compute the throughput time between an event and its direct successor. While SOURCE always refers to the actual event, TARGET refers to its following event. Consequently, SOURCE and TARGET can be used to combine an event with its following event in the same row of a table. Both operators accept a column of the activity table as input and return the respective value of the referred event, as illustrated in Figure 7.

For the first event in the Activities table, SOURCE returns the activity name 'A' of the current event, while TARGET returns the activity name 'B' of the following event (refer to ① in Figure 7). For the second event of the input table, SOURCE returns 'B' and TARGET returns 'C' (refer to ② in Figure 7) while they return 'C' and 'D' for the third event (refer to ③ in Figure 7).

The example also demonstrates how the SOURCE and TARGET operators can be used to compute the throughput time. Instead of the activity column, we can use the column containing the timestamp of the events as input. Consequently, SOURCE and TARGET return the timestamps of the referred events. Then, we can pass the result columns of the SOURCE and TARGET operators to the MINUTES_BETWEEN operator to compute the difference between the timestamps of an event and its following event in minutes. In the example of Figure 7, this results in throughput times of 1 minute from 'A' to 'B', 6 minutes from 'B' to 'C', and 2 minutes from 'C' to 'D'.

```
SOURCE ( input_column [, filter_column ] [, edge_config ] )

TARGET ( input_column [, filter_column ] [, edge_config ] )

edge_config ←  ANY_OCCURRENCE[]    TO ANY_OCCURRENCE[]
             | FIRST_OCCURRENCE[] TO ANY_OCCURRENCE[]
             | FIRST_OCCURRENCE[] TO ANY_OCCURRENCE_WITH_SELF[]
             | ANY_OCCURRENCE[]    TO LAST_OCCURRENCE[]
             | FIRST_OCCURRENCE[] TO LAST_OCCURRENCE[]
```

Syntax 1: SOURCE and TARGET operators

Syntax 1 shows the syntax of the SOURCE and TARGET operators, which is similar for both operators. The first parameter is a column of the activity table. Its values are mapped to the referred events and returned as result column. The result column is stored in a temporary result table which can be joined with the case table.

To skip certain events, the SOURCE and TARGET operators accept an optional filter column as a parameter. This column must be of the same size as the activity table. The SOURCE and TARGET operators ignore all events that have a NULL value in the related entry of the filter column. Usually, the filter column is created using the REMAP_VALUES operator.

```
REMAP_VALUES ( input_column ( , [ string , string ] )+ [, other_value ] )

REMAP_INTS ( input_column ( , [ integer , integer ] )+ [, other_value ] )
```

Syntax 2: REMAP_VALUES operator

The syntax of the REMAP_VALUES operator is shown in Syntax 2. The first parameter is an input column of type string that provides the values that should be remapped as input. For creating a filter column for the SOURCE and TARGET operators, this input column is usually the activity column of the activity table. However, REMAP_VALUES can be generally applied to any column of type string. The second parameter is a list of one or more pairs of string values that describe the mapping. Each occurrence of the first value of the pair will be remapped to the second value of the pair. Finally, the operator accepts an optional string value that will replace all values that are not remapped within the mapping. If this optional default replacement value is missing, all values not considered in the mapping will remain unchanged. As the REMAP_VALUES operator is only applicable to columns of type string, REMAP_INTS provides a similar functionality for columns of type integer.

Figure 8 shows a simple example of the REMAP_VALUES operator. It takes the activity column of the activity table as input and maps 'B' and 'C' to NULL. As the optional replacement value is not defined, all the other values ('A' and 'D') remain the same. Figure 9 demonstrates how to use the result of the REMAP_VALUES as filter column for the SOURCE and TARGET operators by an example query.

Query

```
TABLE (
  REMAP_VALUES ( "ACTIVITIES"."ACTIVITY", [ 'B', NULL ], [ 'C', NULL ] ) )
);
```

Input

CASE_ID	ACTIVITY	TIMESTAMP
1	‘A’	2019-01-01 13:00:00
1	‘B’	2019-01-01 13:01:00
1	‘C’	2019-01-01 13:07:00
1	‘D’	2019-01-01 13:09:00

ACTIVITIES

Output

Remapped Values
‘A’
NULL
NULL
‘D’

RESULT

Fig. 8: Example of REMAP_VALUES operator

The query returns the activity names of the source and target events given in the input table **Activities.** However, the **Result** table only shows one row relating ‘A’ to ‘D’ because the activities ‘B’ and ‘C’ are filtered out. This is achieved by passing the result of the REMAP_VALUES operator as shown in Figure 8 to the SOURCE operator as filter column. As both activities ‘B’ and ‘C’ are mapped to NULL, the next subsequent activity of ‘A’ is ‘D’ with a throughput time of 9 minutes.

To define which relationships between the events should be considered, the operators offer the optional edge configuration parameter. Figure 10 illustrates the different edge configuration options. The first option (a) is the default and only considers the direct follow relationships between the events, while option (b) only considers relationships from the first event to all subsequent events. Option (c) is similar to option (b) but also considers self-loops of the first event. Option (d) is the opposite of option (b) and only considers relationships going from any event to the last event.

Query

```
TABLE (
  SOURCE ( "ACTIVITIES"."ACTIVITY",
         REMAP_VALUES ( "ACTIVITIES"."ACTIVITY", [ 'B', NULL ], [ 'C', NULL ] ) ),
  TARGET ( "ACTIVITIES"."ACTIVITY" ),
  MINUTES_BETWEEN ( SOURCE ("ACTIVITIES"."TIMESTAMP" ),
                    TARGET ("ACTIVITIES"."TIMESTAMP" ) )
);
```

Input

CASE_ID	ACTIVITY	TIMESTAMP
1	‘A’	2019-01-01 13:00:00
1	‘B’	2019-01-01 13:01:00
1	‘C’	2019-01-01 13:07:00
1	‘D’	2019-01-01 13:09:00

ACTIVITIES

Output

Source	Target	Throughput Time (mins)
‘A’	‘D’	9

RESULT

Fig. 9: Example for omitting activities ‘B’ and ‘C’ in SOURCE and TARGET operators

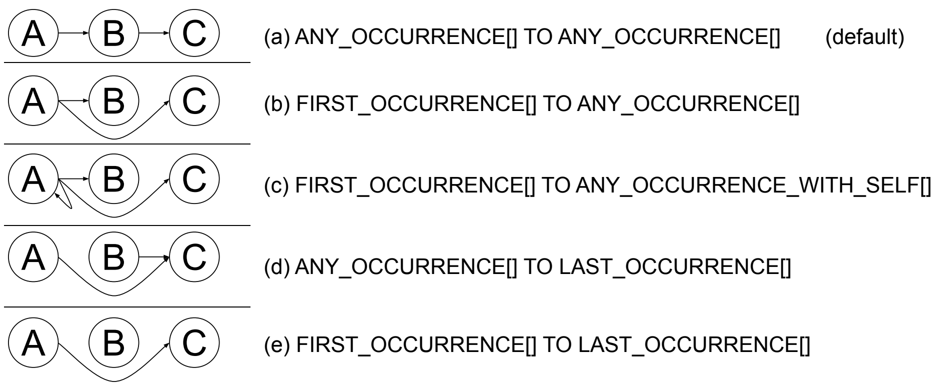
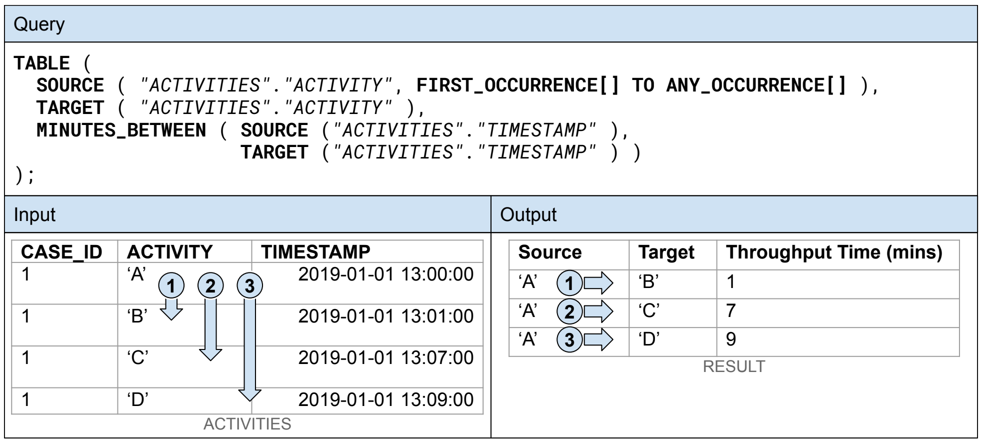

A B C
(a) ANY_OCCURRENCE[] TO ANY_OCCURRENCE[] (default)
(b) FIRST_OCCURRENCE[] TO ANY_OCCURRENCE[]
(c) FIRST_OCCURRENCE[] TO ANY_OCCURRENCE_WITH_SELF[]
(d) ANY_OCCURRENCE[] TO LAST_OCCURRENCE[]
(e) FIRST_OCCURRENCE[] TO LAST_OCCURRENCE[]

Fig. 10: Available edge configuration options of the SOURCE and TARGET operators


Query

```
TABLE (
  SOURCE ( "ACTIVITIES"."ACTIVITY", FIRST_OCCURRENCE[] TO ANY_OCCURRENCE[] ),
  TARGET ( "ACTIVITIES"."ACTIVITY" ),
  MINUTES_BETWEEN ( SOURCE ("ACTIVITIES"."TIMESTAMP" ),
                    TARGET ("ACTIVITIES"."TIMESTAMP" ) )
);
```

Input

CASE_ID	ACTIVITY	TIMESTAMP
1	‘A’	2019-01-01 13:00:00
1	‘B’	2019-01-01 13:01:00
1	‘C’	2019-01-01 13:07:00
1	‘D’	2019-01-01 13:09:00

ACTIVITIES

Output

Source	Target	Throughput Time (mins)
‘A’ ① ⇨	‘B’	1
‘A’ ② ⇨	‘C’	7
‘A’ ③ ⇨	‘D’	9

RESULT

Fig. 11: Example for computing how many minutes after the start of the process an activity was executed

Finally, option (e) only considers the relationship between the first and the last event. The different options enable the user to compute KPIs between different activities of the process. For example, you can use option (b) to compute how many minutes after the start of the process (indicated by the first activity ‘A’) an activity was executed. This is illustrated in Figure 11 where SOURCE always refers to the first event of the case (activity ‘A’) while TARGET refers to any other event (activities ‘B’, ‘C’, and ‘D’). Consequently, MINUTES_BETWEEN computes the minutes elapsed between the occurrence of ‘A’ and all the other activities of the case. For computing the remaining process execution time for each activity of the process, you can simply adapt the edge configuration in the query from Figure 11 to option (d).

To simplify the query, the optional edge configuration and the filter column need to be defined in only one occurrence of SOURCE or TARGET per query. The settings are implicitly propagated to all other operators in the same query. This can be seen in the query in Figure 9, where the TARGET operator inherits the filter column from the SOURCE operator.

Besides the computation of custom process KPIs, like the throughput time between certain activities, SOURCE and TARGET also enable more advanced use cases, like the segregation of duties, as we will demonstrate in Section 5.3. A concept similar to the SOURCE and TARGET operators has recently been proposed in [3].

4.3 Variant Computation

The computation of variants is a vital task in process mining. Most process discovery algorithms, like the Inductive Miner [8] or the Heuristics Miner [17], use them as input instead of the raw events and cases to significantly speed up the computation. To compute variants, Celonis PQL provides the VARIANT operator that aggregates all events of a case into a string which represents the variant of the case. The resulting column is added to the case table such that each case is related to its respective variant.

```
VARIANT ( input_column )

SHORTENED ( VARIANT ( input_column ) [, max_cycle_length ] )
```

Syntax 3: VARIANT operator and VARIANT operator with reduced self-loops

The syntax of the VARIANT operator is shown in Syntax 3. As input, the operator uses a column of type string from the activity table. The operator concatenates the string values of the given column into a single string delimited by comma, and adds the result to the row of the related case. Usually, the activity column is used as input, however, other columns of the activity table, like the name of the executing department or user, can be used.

Sometimes, different cases may have self-loops of the same activity but with a different number of activities. Consequently, these cases are related to different variants. However, in some applications it is not of interest how often an activity is repeated but only if there is a self-loop or not. For such cases, the VARIANT operator can be wrapped by the SHORTENED command which shortens self-loops to a maximum number of occurrences. In this way, it is possible to abstract from repeated activities and reduce the number of distinct variants. The limit for the length of the self-loops can be specified by an optional parameter. The default value for the maximum cycle length is 2.

Figure 12 shows an example query for the variant computation. The input data consists of an activity table and a case table which can be joined by the foreign key relationship between the **Case_ID** columns of both tables. For each case, the query result shows the variant string (**Variant** column) and the variant string with reduced self-loops (**Shortened** column). Column **Variant** of the **Result** table shows individual variants (with a varying number of 'B' activities) for each case, while column **Shortened** shows equal variants for the cases 2 and 3 where the third 'B' activity of case 3 is omitted.

Query

```
TABLE (
  "CASES"."CASE_ID",
  VARIANT ( "ACTIVITIES"."ACTIVITY" ),
  SHORTENED ( VARIANT ( "ACTIVITIES"."ACTIVITY" ) )
);
```

Input

CASE_ID	ACTIVITY	TIMESTAMP
1	‘A’	2019-01-01 13:00:00
1	‘B’	2019-01-01 13:01:00
1	‘C’	2019-01-01 13:02:00
2	‘A’	2019-01-01 13:03:00
2	‘B’	2019-01-01 13:04:00
2	‘B’	2019-01-01 13:05:00
2	‘C’	2019-01-01 13:06:00
3	‘A’	2019-01-01 13:07:00
3	‘B’	2019-01-01 13:08:00
3	‘B’	2019-01-01 13:09:00
3	‘B’	2019-01-01 13:10:00
3	‘C’	2019-01-01 13:11:00

ACTIVITIES

CASE_ID
1
2
3

CASES

ACTIVITIES.CASE_ID	CASES.CASE_ID

Foreign Keys

Output

CASE_ID	Variant	Shortened
1	‘A, B, C’	‘A, B, C’
2	‘A, B, B, C’	‘A, B, B, C’
3	‘A, B, B, B, C’	‘A, B, B, C’

RESULT

Fig. 12: Example for the VARIANT operator with and without reduced self-loops

4.4 Conformance Checking

Besides process discovery, conformance checking is another important process mining technique which relates a process model to an event log [4]. It enables the identification of deviations of the as-is process – as reflected by the data in the event log – from the to-be process as defined by a prescriptive process model. Celonis PQL offers such conformance checking capability via the CONFORMANCE operator.

The syntax of the CONFORMANCE operator is shown in Syntax 4. It accepts an activity column and a description of a process model, as a Petri net, as input. The first part of the model description is a list containing all places of the Petri net, each specified by a unique string ID. It is followed by a similar list of all transitions.

```
CONFORMANCE ( activity_column, model )

READABLE ( CONFORMANCE ( activity_column, model ) )

model         ←  places, transitions, flows, mapping, start_places, end_places
places        ←  node_list
transitions   ←  node_list
start_places  ←  node_list
end_places    ←  node_list
node_list     ←  [ ( string_id )+ ]
flows         ←  [ ( [ string_id string_id ] )+ ]
mapping       ←  [ ( [ string string_id ] )+ ]
```

Syntax 4: CONFORMANCE operator

The third part of the model description is a list of flow relations, where each flow relation is specified as a pair of the source place and the target transition or a pair of the source transition and the target place, respectively. After that, a list of value pairs defines the mapping of activity names to the related transitions. The first value in such a pair is an activity name as a string while the second value is the ID of a transition which must be defined in the list of transitions. The last two parts of the model description are the lists of start and end places, respectively. Both lists consist of place IDs which must be specified in the first part of the model description.

The CONFORMANCE operator replays the activities' names from the input column on the process model. As a result, it adds a temporary column of type integer to the activity table. The value of a row in this fresh column indicates if there is a conformance issue or not. Also, the type of violation and the related activities in the process model are encoded in this value.

As the integer encoding is not suitable for the end-user, the CONFORMANCE operator can be wrapped in the READABLE command. If there is a violation, this will translate the encoding into a message explaining the violation.

Figure 13 shows an example query that uses the CONFORMANCE operator which takes an activity table with three different cases as input. The process model that should be related to the event log is illustrated in Figure 14. It is a simple Petri net consisting of two transitions and three places forming a trivial sequence of two activities 'A' and 'B'.

The result of the query consists of four columns with **Case_ID** as the first, and the **Activity** as the second column. The third column (**Conformance**) shows the integer encoded result of the CONFORMANCE operator, while the fourth column (**Readable**) shows intuitive messages explaining the deviations. Even though activity 'A' matches the model, the first row is marked as *incomplete* because it is the last activity of case 1 which does not reach the end of the process due to the missing activity 'B'. For case 2, the first activity (row 2) conforms, but the second activity ('C' in row 3) is not part of the process model and, therefore, is marked as an *undesired activity*. In contrast to that, case 3 fully conforms, which is indicated for all its activities (rows 4 and 5) of the output.

Query

```
TABLE (
  "ACTIVITIES"."CASE_ID",
  "ACTIVITIES"."ACTIVITY",
  CONFORMANCE ( "ACTIVITIES"."ACTIVITY" , [ "P_0" "P_1" "P_2"] , [ "T_01" "T_12"] ,
    [ [ "P_0" "T_01" ] [ "T_01" "P_1" ] [ "P_1" "T_12" ] [ "T_12" "P_2" ] ],
    [ [ 'A' "T_01" ] [ 'B' "T_12" ] ], [ "P_0" ] , [ "P_2" ]
  ),
  READABLE (
    CONFORMANCE ( "ACTIVITIES"."ACTIVITY" , [ "P_0" "P_1" "P_2"] ,[ "T_01" "T_12"] ,
      [ [ "P_0" "T_01" ] [ "T_01" "P_1" ] [ "P_1" "T_12" ] [ "T_12" "P_2" ] ],
      [ [ 'A' "T_01" ] [ 'B' "T_12" ] ], [ "P_0" ] , [ "P_2" ]
    )
  )
);
```

Input

CASE_ID	ACTIVITY	TIMESTAMP
1	'A'	2019-01-01 13:00:00
2	'A'	2019-01-01 13:01:00
2	'C'	2019-01-01 13:02:00
3	'A'	2019-01-01 13:03:00
3	'B'	2019-01-01 13:04:00

ACTIVITIES

Output

CASE_ID	ACTIVITY	Conformance	Readable
1	'A'	2147483647	'Incomplete'
2	'A'	0	'Conforms'
2	'C'	-2	'C is an undesired activity'
3	'A'	0	'Conforms'
3	'B'	0	'Conforms'

RESULT

Fig. 13: CONFORMANCE operator example with integer encoding and readable explanation

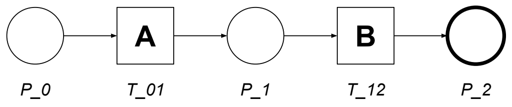


Fig. 14 Simple Petri net used in conformance checking example

As the example illustrates, the model description is quite extensive even for such a small model which seems to contradict the design goal of keeping the language as simple as possible. However, the conformance operator is usually called from a GUI component. Using this component, the user can upload, automatically discover or manually model a process model in Business Process Model and Notation (BPMN)[12] representation, which is automatically translated into the required string description. The GUI component can also bind the model description string to a variable. Instead of defining the process model in the query, the user can simply insert the variable, which makes it much easier to use the CONFORMANCE operator in other GUI components. For example, the user can apply the CONFORMANCE operator in a filter in order to restrict a data table or chart only to cases that are marked as *incomplete*.

5 Use Cases

This section demonstrates the applicability of Celonis PQL for solving real-world problems of business users. First, we show how Celonis PQL is used to discover working capital optimizations. In our example, we identify early invoice payments to improve the on-time payment rate (Section 5.1). Second, we demonstrate how Celonis PQL is used to identify ping-pong-cases in IT service management processes in order to reduce ticket resolution times (Section 5.2). Third, we show the application of Celonis PQL for detecting segregation of duties violations to prevent fraud and errors in procurement (Section 5.3).

5.1 Working Capital Optimization by On-time Payment of Invoices

Working capital is defined as the difference between a company's current assets and its current liabilities essential for the smooth operation of a business, and is a key figure for measuring a company's liquidity and its short-term financial health. Working capital management aims to optimize liquidity while ensuring sustained operations in the long term. Typical ways to optimize the working capital are inventory reduction, faster collection of receivables and lengthening of the payable cycle. Activities for optimization within these areas are manifold. One example for lengthening the payable cycle is on-time payment of invoices by avoiding both early and late payments. Eradicating early payments can improve working capital by keeping assets until the day they are due. Preventing late payments can stop late payment penalties and allows to take advantage of cash discounts.

```
 1 FILTER PROCESS EQUALS 'Clear Invoice';
 2 FILTER PROCESS EQUALS 'Due Date passed';
 3
 4 TABLE(
 5   "Invoice"."VendorName",
 6   AVG(
 7     CASE
 8       WHEN COALESCE(
 9         CALC_THROUGHPUT(
10           FIRST_OCCURRENCE['Clear Invoice'] TO
11           FIRST_OCCURRENCE['Due Date passed'],
12           REMAP_TIMESTAMPS("Activities"."Eventtime", DAYS)),
13           0
14         ) > 3
15       THEN 1.0
16       ELSE 0.0
17     END
18   ) AS "TooEarlyRatio"
19 ) ORDER BY "TooEarlyRatio" DESC;
```

Query 1: Average days of early payments per vendor

Query 1 shows a Celonis PQL statement for the calculation of the early payment ratio per vendor. Using this query, the user is able to discover the vendors which have the highest ratio of invoices paid more than three days early.

The distinction whether an invoice was paid more than three days before the due date is made within the CASE WHEN statement (lines 7–17) by calculating the throughput time with the CALC_THROUGHPUT function (lines 9–12). The CALC_THROUGHPUT operator takes the timestamp of the first occurrence of activity 'Clear Invoice' and the timestamp of the first occurrence of activity 'Due Date passed' and calculates the difference. The second parameter, given as REMAP_TIMESTAMPS operator (line 12), counts time units in the specified interval DAYS based on the timestamps in the activity table to enable the calculation of the throughput time. As the CALC_THROUGHPUT operator returns NULL if the end date is before the start date, the result of the calculation is wrapped in the COALESCE (lines 8–14) operator to return 0 in these cases. The result of the COALESCE operator is then compared to the specified three days (line 14). If the result is greater than 3, the CASE WHEN statement returns 1; otherwise 0.

The whole CASE WHEN statement is wrapped in the AVG operator (lines 6–18), allowing to calculate the ratio of invoices paid more than three days early. By specifying the vendor name (" Invoice"."VendorName") as a dimension in the TABLE statement (lines 4–19), the ratio is calculated per vendor. To get the vendors with the highest ratio of early invoice payments, the result of the AVG calculation is sorted in descending order by the ORDER BY statement (line 19). The two

FILTER statements (lines 1 and 2) at the beginning of the query ensure that only cases with an already paid invoice and a specified due date are considered within the calculation.

5.2 Identifying Ping-Pong-Cases for Ticket Resolution Time Reduction

IT service management (ITSM) refers to the measurements and methods performed by an organization to ensure the optimal support of IT services provided to customers. Service-level agreements (SLAs) between the organization (also referred to as service provider) and the customers (also referred to as service user) define particular aspects of the provided support like availability, responsibility, and most important quality. SLAs are important factors influencing service quality levels and customer happiness. Therefore, compliance with defined SLAs is essential.

Customer support within ITSM systems is usually carried out by creating a ticket for each customer inquiry in the system and solving these tickets. Thus, an important key figure for ITSM is the resolution time of a ticket. A ticket is ideally resolved without the interference of many departments or teams. However, in so-called ping-pong-cases, a ticket is repeatedly going back and forth between departments or teams. This is massively slowing down the resolution time. To prevent this, the identification of ping-pong-cases is crucial.

Query 2 shows a Celonis PQL query to identify direct ping-pong-cases. A case in this context is equivalent to a ticket. Direct ping-pong refers to tickets in which the same activity appears (at least) two times with only one other activity in between, e.g. ‘Change Assigned Group’ directly followed by ‘Review Ticket’ directly followed by ‘Change Assigned Group’.

The query calculates whether a ticket is a ping-pong-case or not within the CASE WHEN statement (lines 4–10). If the current activity equals ‘Change Assigned Group’, the second next activity is equal to the current activity and the next activity is not equal to the current activity, the ticket is classified as ping-pong-case and the CASE WHEN statement returns the ticket ID.

```
1  TABLE(
2    "Tickets"."Country",
3    COUNT(DISTINCT
4      CASE
5        WHEN "Activities"."Activity" = 'Change Assigned Group'
6          AND "Activities"."Activity" = ACTIVITY_LEAD("Activities"."Activity",2)
7          AND "Activities"."Activity" != ACTIVITY_LEAD("Activities"."Activity",1)
8        THEN "Activities"."TicketId"
9        ELSE NULL
10     END
11   )
12   /
13   COUNT_TABLE("Tickets")
14   AS "DirectPingPongRatio"
15 ) ORDER BY "DirectPingPongRatio" DESC;
```

Query 2: Direct ping-pong-case ratio per country

The comparison between the current activity, the next and the second next activity is achieved by using the ACTIVITY_LEAD operator (lines 6 and 7). In general, the ACTIVITY_LEAD operator returns the activity from the row that follows the current activity by offset number of rows within a case. As the timestamp column of the activity table is defined in the data model, the ACTIVITY_LEAD operator can implicitly rely on the correct ordering of events. The CASE WHEN statement is wrapped in a COUNT operator (lines 3–11) to count the total number of ping-pong-cases. By adding DISTINCT (line 3) to the COUNT operator, it is guaranteed that a ticket is only counted once although ping-pong activities can occur multiple times within a ticket. The result of the COUNT operator is then divided by the total number of tickets to get the ratio of ping-pong-cases. Thereby, the total number of tickets is calculated using the COUNT_TABLE operator (line 13). COUNT_TABLE is a performance-optimized function for counting the number of rows of a specified table. By specifying the country (" Tickets " . " Country ", line 2) as a dimension in the TABLE statement (lines 1–15), the ratio of ping-pong-cases is calculated per country. In order to get the countries with the highest ratio of ping-pong-cases, the calculated ratio is sorted in descending order by the ORDER BY statement (line 15).

Query 3 shows a Celonis PQL query to identify indirect ping-pong-cases. Indirect ping-pong refers to tickets in which the activity 'Change Assigned Group' appears at least two times with more than one other activity in between, e.g., 'Change Assigned Group', directly followed by 'Review Ticket', directly followed by 'Do some work', directly followed by 'Change Assigned Group'.

The query shown in Query 3 calculates whether a ticket is an indirect ping-pong-case or not by using the operators ACTIVATION_COUNT (line 7) and ACTIVITY_LEAD (lines 8 and 9) within a CASE WHEN statement (lines 6–12).

ACTIVATION_COUNT returns, for every activity, how many times it has already occurred (so far) in the current case. Within the CASE WHEN statement, the ticket ID is returned if the ACTIVATION_COUNT is greater than 1 and the current activity is not equal to the last and the second last activity. The latter comparison is calculated by the ACTIVITY_LAG operator. In general, ACTIVITY_LAG returns the activity from the row that precedes the current activity by o˙set number of rows within a case.

```
 1 FILTER "Activities"."Activity" = 'Change Assigned Group';
 2
 3 TABLE(
 4   "Tickets"."Country",
 5   COUNT(DISTINCT
 6     CASE
 7       WHEN ACTIVATION_COUNT("Activities"."Activity") > 1
 8         AND "Activities"."Activity" != ACTIVITY_LAG("Activities"."Activity",2)
 9         AND "Activities"."Activity" != ACTIVITY_LAG("Activities"."Activity",1)
10       THEN "Activities"."TicketId"
11       ELSE NULL
12     END
13   )
14   /
15   COUNT_TABLE("Tickets")
16   AS "IndirectPingPongRatio"
17 ) ORDER BY "IndirectPingPongRatio" DESC;
```

Query 3: Indirect ping-pong-case ratio per country

If one of the expressions in the WHEN-clause (lines 7–9) is FALSE, the CASE WHEN statement returns NULL. As in the example for direct ping-pong-cases, the CASE WHEN statement is wrapped in a COUNT operator (lines 5–13) to count the total number of ping-pong-cases. By adding DISTINCT (line 5) to the COUNT operator, it is guaranteed that a ticket is only counted once as an indirect ping-pong-case. The result of the COUNT operator is then, again, divided by the total number of tickets to get the ratio of indirect ping-pong-cases. Thereby, the total number of tickets is calculated using the COUNT_TABLE operator (line 15). The country (" Tickets " . " Country ", line 4) is specified as a dimension in the TABLE statement (lines 3–17) to calculate the ratio of indirect ping-pong-cases per country. To get the countries with the highest ratio of indirect ping-pong-cases, the calculated ratio is sorted in descending order by the ORDER BY statement (line 17). The FILTER statement (line 1) at the beginning of the query ensures that the current activity is 'Change Assigned Group'.

5.3 Fraud Prevention by Identifying Segregation of Duties Violations

Segregation of Duties (SoD) is a concept based on shared responsibilities: It ensures that certain activities are not executed by the same person or department. It applies the four-eyes principle and decreases the power of an individual person or department in order to prevent fraud and errors. Therefore, the concept is essential for effective risk management and internal controls. In procurement, unauthorized or unnecessary purchase orders or purchase orders for personal use may occur if duties are not separated properly. With this in mind, it is best practice in procurement to have different people, or departments, for purchase approvals and invoice payment approvals.

```
 1 TABLE(
 2   "PurchaseOrders"."PurchaseOrganization",
 3   AVG(
 4     CASE
 5       WHEN SOURCE("Activities"."Department",
 6           REMAP_VALUES("Activities"."Activity",
 7             [ 'Request Approval', 'Request Approval' ],
 8             [ 'Grant Approval', 'Grant Approval' ],
 9             NULL
10           )
11         ) = TARGET("Activities"."Department")
12       THEN 1.0
13       ELSE 0.0
14     END
15   ) AS "SoDViolationRatio"
16 ) ORDER BY "SoDViolationRatio" DESC;
```

Query 4: Violated SoD ratio per purchase organization

Query 4 shows a Celonis PQL query for the calculation of the ratio of purchase orders in which the SoD for the activities 'Request Approval' and 'Grant Approval' was violated because the same department executed both tasks. The ratio is calculated per purchase organization to discover the ones with the highest violation ratio. Comparing whether the activities 'Request Approval' and 'Grant Approval' were executed by the same department is done within the CASE WHEN statement (lines 4–14). The statement contrasts the source event department to the target event department by using the SOURCE and TARGET operators (lines 5–11). A detailed description of these operators can be found in Section 4.2. The REMAP_VALUES function (lines 6–10) passed as a parameter to the SOURCE operator, allows to extract the activities 'Request Approval' and 'Grant Approval' by mapping them to the same name while mapping all the other activities to NULL. If the comparison between the source department and the target department returns true, the CASE WHEN statement returns 1 (line 12); otherwise 0 (line 13).

The AVG operator (lines 3–15) in which the CASE WHEN statement is wrapped calculates the ratio of violations of the SoD. By specifying the purchasing organization (" PurchaseOrders " . " PurchaseOrganization ") as a dimension in the TABLE statement (lines 1–17), the ratio of violations is calculated per purchase organization. The result of the AVG calculation is sorted in descending order by the **ORDER BY** statement (line 17) to get the organizations with the highest violation rate.

6 Implementation

Celonis PQL is the basis of a commercial product that promises interactive process mining and business intelligence using data sets with hundreds of millions of events. For this reason, the implementation of the language has to fulfill high requirements regarding performance, scalability, and low latency.

The implementation targets business intelligence and process mining because our experience is that process mining unfolds its full potential in combination with classic BI. Many insights into customer data could only be derived by taking into account further dimensional tables, in addition to the event log. For example, to find the country with the most segregation of duties violations (see Section 5.3), information about the countries has to be available.

In the past, different types of software addressed two fields: BI and process mining. BI is the domain of relational database systems. This is also reflected by the TPC-H benchmark [5], which is the de facto standard benchmark for analytical databases. The benchmark portrays a wholesale supplier in a data warehouse and focuses on classic BI questions, but it does not consider any process mining aspects. As a result, databases perform well in answering BI questions, but they are not optimized to answer process mining questions.

Besides the Celonis PQL implementation, process mining is done on relational databases, specialized implementations, or graph databases [6]. Graph databases can be considered as a reasonable choice, because a process instance can be interpreted as a graph. While they deliver a decent performance for process mining, a graph database is not optimized for business intelligence. This is why our objective was not to build upon an existing data processing solution. Instead, we wanted to design a system from scratch which combines techniques from relational and graph databases.

Like most state-of-the-art database systems, the Celonis PQL implementation is a main memory database. This means that it uses main memory as primary storage instead of the disk in order to avoid slow disk access. It is implemented in C++ and Java. C++ is used for all software modules in which active control over the main memory is necessary, like the storage layer and the performance-critical process mining algorithms. Java is used for non-performance critical sections, like the parser, because of its memory safety.

The Celonis PQL Engine uses state-of-the-art techniques from the database research community like just-in-time (JIT) compilation and dictionary encoding. JIT compilation is a technique which generates and compiles code to execute a query. It allows achieving a good cache locality and a low number of CPU instructions resulting in a very high performance as shown by Neumann et al. [11]. Dictionary encoding is a standard compression technique to reduce the memory overhead [10].

Elements that are taken from the graph community are some algorithms and data structures, like the adjacency list. The Celonis PQL Engine thereby exploits features of an event log, like process instances, which in most cases represent graphs with a rather low number of nodes.

The Celonis PQL Engine implementation focuses on analytical queries. It is snapshot-based, which means that the engine answers queries based on the data of the source system at a particular point in time. The data is not constantly updated. Instead, a bulk update mechanism is in place. This avoids some overhead, which would be introduced by a concurrency control mechanism like MVCC [2].

The Celonis PQL Engine implementation is also focused on scaling with the number of CPU cores within one server. The challenge here is that the implementation has to be lightweight enough to run on commodity laptop hardware, while it has to be sophisticated enough to make use of all the power a high-end server provides. This is achieved by parallel intra query execution, refer to Leis et al. [9] for details.

The Celonis PQL Engine implementation, however, is not designed to process queries across multiple servers. This is a conscious decision because the Celonis PQL Engine needs to provide results with low latency. Synchronizing multiple machines across the network to execute a Celonis PQL query adds overhead, which is against the low latency goal. The work of Schüle et al. [14] supports our single-node approach by demonstrating that a single server can handle even large scale applications like Wikipedia. To support such applications, lightweight in-memory compression techniques have to be in place. The Celonis PQL Engine implementation uses an approach for in-memory compression which is inspired by the work of Lang et al. [7].

7 Celonis PQL and the Process Querying Framework

The Process Querying Framework (PQF) [13] is an abstract system consisting of a set of generic components to define a process querying method. Celonis PQL covers many of these components. This section describes the integration of Celonis PQL into the PQF.

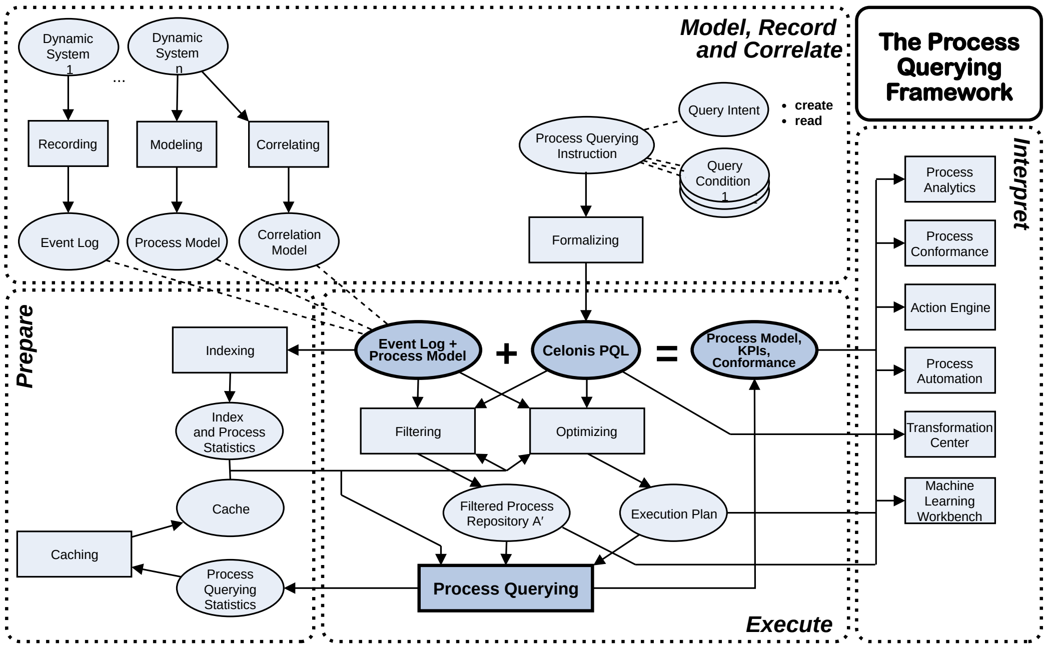


Fig. 15: Celonis PQL in the context of the Process Querying Framework

Figure 15 illustrates how Celonis PQL instantiates the main components of the PQF. The first part of the framework (*Model, Record, and Correlate*) retrieves or creates the behavioral models and formalizes the business questions into process queries. The event logs are recorded by information systems like ERP or CRM systems, and extracted from these source systems into the Celonis IBC platform. The process models are either manually modeled (e.g., in Celonis IBC or in an external tool) or discovered by process mining techniques, like the Inductive Miner [8]. The correlation models are created by the conformance checking operator (see Section 4.4), relating activities of the event log to tasks in the process model. However, the different kinds of model repositories overlap due to their related storage, as relational data within the same data model. For example, the result column of the conformance checking operator (correlation model) is added to the activity table (event log).

The query intent of Celonis PQL is limited to *create* and *read*. While all supported kinds of behavioral models can be read, process models and correlation models can also be created by Celonis PQL queries (e.g., by process discovery and conformance checking). The *update* and *delete* query intents are not included – especially for the event logs – as they should always stem from the source systems. Therefore, event log updates can be achieved by delta loads which regularly extract the latest data from the source systems. The process querying instruction is usually defined by an analyst through a user interface. For example, the user defines the columns to be shown in a table, which can be considered as the query conditions. The selections from the user interface are then formalized into a Celonis PQL query.

The *Prepare* part of the framework focuses on increasing the efficiency of the query processing. The Celonis PQL Engine – that processes the queries – maintains a cache for query results, refer to Section 6. After the application starts, it warms up the cache with the most relevant queries derived from the *Process Querying Statistics* to provide fast response times. According to [13], the *Indexing* component does not only include classical index structures but also all kind of data structures for an efficient retrieval of data records. It is covered by the dictionary encoding of columns, as discussed in Section 6.

The *Execute* part of the framework combines an event log with an optional process model and a Celonis PQL query into a query result which can be either a process model, KPIs, filtered and processed event log data, or conformance information. The concrete input and output of the query depend on the selected query intent and the query conditions. The *Filtering* component reduces the input data of the query. This can either be achieved by the REMAP_VALUES operator and the filter column of the SOURCE and TARGET operators, as described in Section 4.2, or by the general filter statement shown in the example in Figure 6. The *Optimizing* component uses basic database technology to rewrite the query and create the *Execution Plan*, which describes a directed graph of operator nodes. The *Process Querying* component then executes the execution plan on the filtered data. It also retrieves data from the cache to avoid re-computation of either the full query or certain parts of it which are shared with previous queries.

The *Interpret* part of the framework communicates the query results to the user and improves the user's comprehension of them. The applications in the Celonis IBC platform incorporate Celonis PQL and make the results accessible to the user. The *Process Analytics* presents the query results as process graphs, charts and tables. Beyond pure visualization, it is highly interactive with dynamic filtering to drill-down the processes to specific cases of interest. This interactivity offered by all GUI components is achieved through the dynamic creation of Celonis PQL queries.

Process Conformance [16] shows the deviations between process model and event log in a comprehensive view, including a comparison of KPIs between conforming and non-conforming cases. In contrast to this, the focus of the *Action Engine* [1] is not to present query results, but to trigger user actions, for instance, by informing about deliveries that are expected to be late. The Action Engine can also trigger automated workflows which are executed by the *Process Automation* component. Within this component, the workflows can query data from the event log using Celonis PQL.

Transformation Center supports process monitoring. It historicizes the query results to show how the processes evolved over time. Finally, *Machine Learning Workbench* provides a platform for user-defined machine learning analyses over event logs and retrieves the event data using Celonis PQL queries.

8 Conclusion and Future Work

In this chapter, we introduced Celonis PQL, which is an independent query language with a custom-built query engine. It is highly optimized for process mining capabilities, and although it was inspired by SQL, the design of Celonis PQL is mostly driven by requirements of business users. A key difference to SQL is that Celonis PQL is a domain-specific language tailored towards a concrete data model. As a consequence, it does not require the user to explicitly define joins of data tables or groupings within the query, which is done implicitly.

As Celonis PQL comprises more than 150 different operators to process event data, we could only provide an overview of the major language features that are currently offered to users and showcase the expressiveness of the language with a few examples. Besides the description of the language, we illustrated the application of Celonis PQL within the various products available in Celonis IBC. Presented statistics show the extensive usage of Celonis PQL within these products. In addition, we presented the applicability of the query language for solving different real-world problems customers are facing, such as fraud prevention with segregation of duties and speed up of service requests by identifying ping-pong-cases. Finally, we described the position of Celonis PQL within the Process Querying Framework (PQF) [13]. Celonis PQL instantiates all parts of the PQF, except for the capability to *simulate* models. Moreover, *create* and *read* query intents are covered.

Future work on Celonis PQL will focus on the implementation of new operators to further enrich the capabilities and use cases of the query language. Additionally, efforts will be made to improve query performance. New features will be developed in co-innovation projects with academic and commercial partners, and with our customers.

Readers can access a wide range of PQL functionalities for free. Business users can use Celonis PQL in the free Celonis Snap[2] version. Academic users can get free access to the full Celonis IBC technology including the wide range of Celonis PQL capabilities via the Celonis IBC - Academic Edition[3].

2 https://www.celonis.com/snap-signup

3 https://www.celonis.com/academic-signup

References

1. Peyman Badakhshan, German Bernhart, Jerome Geyer-Klingeberg, Janina Nakladal, Steffen Schenk, and Thomas Vogelgesang. The action engine – turning process insights into action. In ICPM Demos 2019, volume 2374 of CEUR Workshop Proceedings. CEUR-WS.org, 2019.
2. Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. Concurrency Control and Recovery in Database Systems. Addison-Wesley, 1987.
3. Alessandro Berti. Increasing scalability of process mining using event dataframes: How data structure matters. CoRR, abs/1907.12817, 2019.
4. Josep Carmona, Boudewijn F. van Dongen, Andreas Solti, and Matthias Weidlich. Conformance Checking - Relating Processes and Models. Springer, 2018.
5. Transaction Processing Performance Council. TPC Benchmark H (Decision Support) Standard Specification Revision 2.18.0, 2018.
6. S. Esser and Dirk Fahland. Storing and querying multi-dimensional process event logs using graph databases. In Process Querying (PQ) Workshop 2019, pages 283–294, 2019.
7. Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In SIGMOD 2016, pages 311–326. ACM, 2016.
8. Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. Discovering block-structured process models from event logs - A constructive approach. In PETRI NETS 2013, volume 7927 of Lecture Notes in Computer Science, pages 311–329. Springer, 2013.
9. Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In SIGMOD 2014, pages 743–754. ACM, 2014.
10. Ingo Müller, Cornelius Ratsch, and Franz Färber. Adaptive string dictionary compression in in-memory column-store database systems. In Proceedings of the EDBT 2014, pages 283–294. OpenProceedings.org, 2014.
11. Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. PVLDB, 4(9):539–550, 2011.
12. OMG. Business Process Model and Notation (BPMN), Version 2.0, January 2011.
13. Artem Polyvyanyy, Chun Ouyang, Alistair Barros, and Wil M. P. van der Aalst. Process querying: Enabling business intelligence through query-based process analytics. Decision Support Systems, 100:41–56, 2017.
14. Maximilian Schüle, Pascal Schliski, Thomas Hutzelmann, Tobias Rosenberger, Viktor Leis, Dimitri Vorona, Alfons Kemper, and Thomas Neumann. Monopedia: Staying single is good enough - the hyper way for web scale applications. PVLDB, 10(12):1921–1924, 2017.
15. Wil M. P. van der Aalst. Process Mining - Data Science in Action, Second Edition. Springer, 2016.
16. Fabian Veit, Jerome Geyer-Klingeberg, Julian Madrzak, Manuel Haug, and Jan Thomson. The proactive insights engine: Process mining meets machine learning and artificial intelligence. In BPM Demos 2017, volume 1920 of CEUR Workshop Proceedings. CEUR-WS.org, 2017.
17. A. J. M. M. Weijters and J. T. S. Ribeiro. Flexible heuristics miner (FHM). In Proceedings of the IEEE CIDM 2011, pages 310–317. IEEE, 2011.

Acronyms

BI	Business Intelligence
BPMN	Business Process Model and Notation
Celonis PQL	Celonis Process Query Language
CRM	Customer Relationship Management
DDL	Data Definition Language
DML	Data Manipulation Language
ERP	Enterprise Resource Planning
ETL	Extract, Transform, Load
GUI	Graphical User Interface
IBC	Intelligent Business Cloud
ITSM	IT Service Management
JIT	Just-In-Time
KPI	Key Performance Indicator
PQF	Process Querying Framework
SLA	Service-Level Agreement
SoD	Segregation of Duties