

# Persistence of workflow control data in temporal databases

DIPLOMARBEIT

zur Erlangung des akademischen Grades

**Diplom-Ingenieur**

im Rahmen des Studiums

**Wirtschaftsinformatik**

eingereicht von

**Thomas Sonnleitner**

Matrikelnummer 00751500

an der Fakultät für Informatik  
der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Inf. Dr.-Ing. Jürgen Dorn

Wien, 19. September 2019

---

Thomas Sonnleitner

---

Jürgen Dorn



# Persistence of workflow control data in temporal databases

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

**Diplom-Ingenieur**

in

**Business Informatics**

by

**Thomas Sonnleitner**

Registration Number 00751500

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Inf. Dr.-Ing. Jürgen Dorn

Vienna, 19<sup>th</sup> September, 2019

---

Thomas Sonnleitner

---

Jürgen Dorn



# Erklärung zur Verfassung der Arbeit

Thomas Sonnleitner

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 19. September 2019

---

Thomas Sonnleitner



# Danksagung

An dieser Stelle möchte ich all jenen danken, die durch ihre fachliche und persönliche Unterstützung zum Gelingen dieser Diplomarbeit beigetragen haben.

Dabei gebührt insbesondere Herrn Prof. Dorn mein Dank, welcher meine Diplomarbeit betreut und begutachtet hat. Für die hilfreichen Anregungen und die konstruktive Kritik bei der Erstellung dieser Arbeit möchte ich mich herzlich bedanken.

Mein besonderer Dank gilt meiner Familie, die mir mein Studium ermöglicht und mich in all den Jahren während und auch bereits vor dem Studium bedingungslos unterstützt hat. Meine Eltern, Stiefeltern, Großeltern und Firmpaten haben mir dadurch den Weg zur Matura und einen Abschluss an einer Universität ermöglicht.

Ebenso herzlich bedanken möchte ich mich auch bei meiner Lebensgefährtin, die mich während des Studiums unterstützt und in dieser Zeit auch viele Entbehrungen hinnehmen musste. Beim Schreiben der Diplomarbeit hat sie mich immer wieder ermutigt und durch die Korrekturlesung auch einen Teil zum Gelingen beigetragen.

Schließlich danke ich meinen Freundinnen und Freunden während der Studienzeit für sehr schöne Jahre an der Technischen Universität Wien. Ohne deren Geduld, Interesse und Hilfsbereitschaft wäre mein Studium nicht ebenso spannend und lehrreich gewesen. Besonderer Dank gilt auch meiner Schwester, die mit mir gemeinsam viele Tage intensiver Schreibarbeit durchgestanden hat.





# Acknowledgements

At this point I would like to thank all those who contributed to the success of this diploma thesis with their professional and personal support.

In particular, I would like to thank Prof. Dorn, who supervised and reviewed my diploma thesis. For the helpful suggestions and the constructive criticism with the production of this work I would like to thank him cordially.

My special thanks go to my family, who made my studies possible and supported me all the years during and even before my studies. My parents, stepparents, grandparents and godfathers made it possible for me to graduate from university.

I would also like to thank my girlfriend, who supported me unconditionally during my studies and also had to accept many hardships during this time. She has constantly motivated me to write my thesis and also contributed to its success by proofreading it.

Finally, I would like to thank my friends and colleagues for wonderful years at the Vienna University of Technology. Without their patience, interest and helpfulness my studies would not have been as exciting and educative. Special thanks go to my sister, who has spent many days of intensive writing with me.



# Kurzfassung

Diese Arbeit zeigt, dass die Verwendung eines temporalen Datenbankmanagementsystems (DBMS) im Geschäftsprozessmanagement (GPM) von Vorteil sein kann. Im Vergleich zur Verwendung eines herkömmlichen DBMS ist die Unterstützung von temporalen Datenabfragen ein großer Vorteil. Diese sind deutlich kürzer und in der Struktur einfacher als herkömmliche Abfragen nach Periodendaten. Zusätzlich wurden Planungs- und Ausführungszeiten der Datenbankabfragen ausgewertet. Teils konnte auch eine bessere Leistung in der Verarbeitung von Abfragen in der temporalen Datenbank beobachtet werden. In dieser Hinsicht konnte jedoch kein allgemeingültiger Leistungsvorsprung festgestellt werden.

Um beide Datenbanken zu vergleichen, wurden dreizehn GPM-relevante Datenbankabfragen in Standard- und temporalem SQL implementiert, die dann in einer Standard- und einer temporären PostgreSQL-Installation ausgeführt wurden. Zweitens ist eine PostgreSQL-Erweiterung, die als Forschungsprototyp von Dignös, Böhlen, Gamper und Jensen entwickelt wurde und in deren Arbeit *Extending the Kernel of a Relational DBMS with Comprehensive Support for Sequenced Temporal Queries* näher beschrieben ist.

Um einen umfassenden Vergleich zu ermöglichen, wurden innerhalb einer Simulationsanwendung unter Verwendung der Activiti Workflow Engine drei unterschiedlich große Datensätze von Prozessausführungsdaten erzeugt und im Activiti Datenmodell gespeichert. Bis zu 4.000 Prozessiterationen eines beispielhaften Kreditgenehmigungsprozesses wurden simuliert. Die generierten Daten wurden wiederholt unter Verwendung von Standard- und temporalen SQL Statements in dem jeweiligen DBMS abgefragt.



# Abstract

This work shows that the use of a temporal Database Management System (DBMS) can be advantageous in Business Process Management (BPM). Compared to the use of a conventional DBMS, the support of a temporal query language and the respective temporal query processing is a major advantage. It enables a less difficult retrieval of process execution data. Additionally, query planning and execution times have been evaluated. In some cases, also a better execution performance could be observed in the temporal database. However, in this respect no general advantage could be identified.

To evaluate the execution performance of both DBMS, thirteen BPM-relevant query statements have been implemented in standard and temporal SQL and executed in a standard and a temporal PostgreSQL installation. Second is a PostgreSQL extension developed as research prototype by Dignös, Böhlen, Gamper and Jensen and described in their work *Extending the Kernel of a Relational DBMS with Comprehensive Support for Sequenced Temporal Queries*.

In order to enable a comprehensive comparison of the query execution performance, three differently sized sets of process execution data were generated within a simulation application, utilizing the Activiti workflow engine and its underlying data model. Up to 4,000 process iterations of a sample credit approval process have been simulated. The generated data has been queried repeatedly utilizing standard and temporal SQL statements being executed in the respective DBMS.

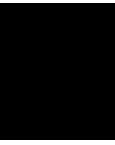


# Contents

<b>Kurzfassung</b>	<b>xi</b>
<b>Abstract</b>	<b>xiii</b>
<b>Contents</b>	<b>xv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation and problem statement . . . . .	1
1.2 Aim of the work . . . . .	2
1.3 Structure and methodological approach . . . . .	3
<b>2 State-of-the-art</b>	<b>5</b>
2.1 Relevance of process execution data in BPM . . . . .	5
2.2 Process modelling with BPMN . . . . .	9
2.3 Activiti framework . . . . .	12
2.4 Managing temporal data in databases . . . . .	16
2.5 Complexity metrics for SQL queries . . . . .	19
<b>3 Methodology</b>	<b>21</b>
3.1 Benchmark approach and setup . . . . .	22
3.2 Benchmark architecture . . . . .	25
3.3 Requirements on supportive artifacts . . . . .	27
<b>4 Implementation</b>	<b>31</b>
4.1 Application architecture and shared resources . . . . .	31
4.2 Simulation application: Generating Activiti process execution data . .	35
4.3 Simulation app input: Business process definition (BPMN model) . . .	43
4.4 Benchmark application: Querying process execution data . . . . .	48
4.5 Benchmark app input: BPM relevant database queries . . . . .	55
<b>5 Benchmark: Processing queries on workflow control data</b>	<b>65</b>
5.1 Benchmark configuration and technical setup . . . . .	65
5.2 Results and analysis . . . . .	67
5.3 Summary of benchmark results . . . . .	74
	xv

<b>6</b>	<b>Conclusion</b>	<b>75</b>
6.1	Limits of work and results . . . . .	76
6.2	Future work . . . . .	76
<b>A</b>	<b>Query Code Listings</b>	<b>79</b>
A.1	Query Q01 Count of open processes over time . . . . .	80
A.2	Query Q02 Count of active processes over time . . . . .	81
A.3	Query Q03 Non-active (idle) periods per process . . . . .	83
A.4	Query Q04 Periods with no open processes . . . . .	84
A.5	Query Q05 Count of assigned tasks to user over time . . . . .	85
A.6	Query Q06 Count of claimed user tasks over time . . . . .	86
A.7	Query Q07 Count of not-yet claimed tasks per user . . . . .	87
A.8	Query Q08 Periods with no assigned tasks to user . . . . .	88
A.9	Query Q09 Count of assigned tasks to department . . . . .	89
A.10	Query Q13 Count of assigned tasks to service . . . . .	90
A.11	Query S01 Distinct periods with claimed tasks per user . . . . .	91
A.12	Query S02 - Special: Parallel processing of specific user tasks . . . . .	92
A.13	Query S03 - Parallel processing of activities . . . . .	93
<b>B</b>	<b>Results</b>	<b>95</b>
B.1	Halstead metrics of queries . . . . .	96
B.2	Execution times querying data of simulation A . . . . .	97
B.3	Execution times querying data of simulation B . . . . .	98
B.4	Execution times querying data of simulation C . . . . .	99
B.5	Planning times querying data of simulation A . . . . .	100
B.6	Planning times querying data of simulation B . . . . .	101
B.7	Planning times querying data of simulation C . . . . .	102
	<b>List of Figures</b>	<b>103</b>
	<b>List of Tables</b>	<b>105</b>
	<b>Listings</b>	<b>107</b>
	<b>List of Listings</b>	<b>107</b>
	<b>Acronyms</b>	<b>109</b>
	<b>Bibliography</b>	<b>111</b>





# Introduction

## 1.1 Motivation and problem statement

Every organization has to manage a number of processes of different kinds such as order, purchase, issue-resolution or other internal processes. A process contains activities, tasks, events, decision points, a number of actors as well as physical or immaterial objects and leads to one or several outcome(s). It might be rather simple or more complex.

The way of how a process is designed and executed affects the quality of service which is perceived by customers, employees or other involved persons. Therefore, organizations do have interest to set up their processes in the most effective way. A process analysis founds the basis for process improvements (or redesign). “Typical example of improvement objectives include reducing costs, reducing execution times and reducing error rates” [DRMR13].

The possibility to analyze but also monitor and execute processes more efficiently motivates organizations to digitalize their processes, or at least support process execution with IT-systems. This happens e.g. in the financial services industry. Financial institutions expect ongoing digitalization to result not only in lower costs but also in faster execution times. On the one hand, there is a need to because of regulatory requirements on reporting timeliness such as demanded by IASB financial reporting standards IFRS 9 [Int14]. On the other hand, also clients demand fast processing times of loan applications, account openings or other services, as they are already used to in other (online) services.

A digitalized process management is supported by a business process or Workflow Management System (WfMS), which governs or automatizes the process flow, supports the approval management and furthermore tracks and monitors process execution. Digitalization of processes and the availability of process metadata (e.g. information about process executions) enables organizations to analyze and improve processes.

However, conventional database systems, in which process metadata is usually stored, do not adequately support the handling of time-related data. Combi and Pozzi discussed architectures of temporal workflow management systems and claimed, that no WfMS was built on top of a temporal Database Management System (DBMS) "due to the lack of a high-performance and reliable temporal database system" [CP04], even though they identified various aspects on how a WfMS can benefit from a temporal database [CP03, CP09]. In their work *From time to Temporal Information*, Tang et al. recognized the "expression and inference of temporal knowledge and temporal algebra" as well as the application of research to fields of "temporal workflow, temporal data mining, etc." as relevant research areas [TPLZ11].

In particular, querying data in conventional database systems with regard to time-related aspects can be complex and associated with long run-times during execution, "in particular for [data] aggregation" [JG18]. Temporal databases focus on temporal aspects in data and promise easier querying and faster execution times. Therefore, they offer various kinds of time oriented statements, specialized data types and a more applicable data organization [Sno00].

To get better insights into the benefit of using such temporal databases, this work will deal with the following research questions:

1. What are typical queries executed on process execution data during process analysis, considering temporal aspects?
2. Can the use of temporal databases and temporal queries be beneficial for querying time-related data in terms of time efficiency or simplicity in context of business process management?

### 1.2 Aim of the work

To improve a process, the current process and thus its process execution data needs to be analyzed. In a first step, this thesis will determine possible information needs which arise during process analysis and their according database queries on process execution data. The result will be a list of typical queries which are executed during business process analysis.

Furthermore, the thesis will identify features of temporal databases which improve the handling of workflow related data, being relevant for process management. Thus, extended query features, but also data types and other characteristics of temporal databases will be reflected in context of process management. Additionally, dimensions that could represent such an improvement will be selected and respective metrics defined, e.g. dimension time efficiency with metric execution time of queries.

The thesis will perform an evaluation of a temporal database in comparison to a conventional databases in context of process management, considering previously defined benchmark dimensions. Therefore, a sample process as well as sample queries on process execution data will be defined. A prototype will be developed which generates mock-up process execution data, performs sample query executions and captures metrics of benchmark dimensions. The thesis will document and interpret captured metrics to subsequently determine if and how temporal databases be beneficial for process management.

## 1.3 Structure and methodological approach

In addition to the introductory chapter and the concluding chapter, the work is divided into five chapters with the following contents:

### 1. State-of-the-art

Literature provides the theoretical background in the field of process and workflow management, including approaches and metrics for quantitative process analysis and notations for process modelling. Furthermore, literature will be reviewed regarding features and capabilities of temporal databases. Additionally, already existing approaches on how to compare different database technologies and measure performance of query executions will be screened as basis for determination of benchmark dimensions and metrics.

### 2. Methodology

A sample process will be designed using BPMN 2.0 in order to generate sample process execution data.

### 3. Implementation of a simulation application: Creation of sample process execution data

Sample process execution data of various process runs will be generated with the help of a workflow engine. The created data should be representative for Business Process Management (BPM) related data sets with focus on temporal data.

### 4. Implementation of benchmark application: Definition and execution of queries on process execution data

Based on literature review, a variety of different queries on process execution data will be defined. These queries should cover the whole range of different kinds of data requirements for process analysis. A sample process will be designed using BPMN 2.0 in order to generate sample process execution data. The sample process should also be used as scenario to show how process execution data can be used to find process optimization potential. The process execution data will be stored using two different database management systems: PostgreSQL as a conventional relational database system and a research prototype by the University of Bozen-Bolzano as temporal database system.

### 5. **Evaluation of query execution performance**

For the benchmark defined queries will be executed on a conventional and a temporal database system. The performance of these database systems in handling the query requests will be evaluated in the defined dimensions with the defined metrics.

# State-of-the-art

This chapter provides an overview of related work. At first, a brief introduction into BPM is followed by more detailed information regarding BPM phase process analysis. Subsequently, data requirements on process execution data and its availability in WfMS are outlined. Furthermore, a brief introduction into modelling in Business Process Model and Notation (BPMN) is provided. On the more technological side, the WfMS tool and its capabilities are described. Furthermore, temporal databases and possibilities for querying temporal data in SQL are discussed. Lastly, metrics for determining the complexity of Structured Query Language (SQL) statements are described.

## 2.1 Relevance of process execution data in BPM

### 2.1.1 A brief introduction into BPM

In the 90s, processes and process design moved into the focus of companies for the first time. In the period before, management's attention was primarily dedicated to the functional organization of companies. As this led to inefficient and bureaucratic way of work, the management's attention moved from the distribution of responsibilities and tasks along a hierarchical organization to the design of more efficient processes [DRMR13].

As not only processes have been re-designed, but also organizations it-self became process-oriented, BPM arose. "BPM provides concepts, methods, techniques and tools that cover all aspects of managing a process - plan, organize, monitor, control - as well as its actual execution" [DRMR13]. So BPM is not only about planning and organizing processes, but also about managing a company process-centered.

Thereby, BPM can be seen as continuous cycle including the phases process identification, process discovery, process analysis, process redesign, process implementation as well as

process monitoring and controlling as described by Dumas et al [DRMR13]. The process as defined by Dumas et al. is visualized in figure 2.1.

In literature, “there are many views of the generic BPM life cycle” [KLWL09]. The design and the concrete steps of the BPM cycle vary in nuances, but the principle idea stays the same: Identified processes are documented in their current form in a first step. Within process analysis, processes are evaluated qualitatively as well as based on quantitative metrics. Observations are compared with target values, historical process analysis and/or with similar processes in other environments (e.g. other but comparable organizations). Based on the findings from the process analysis, the process is redesigned and implemented. During execution, the process is then monitored and controlled in order to draw further conclusions in a new process improvement iteration.

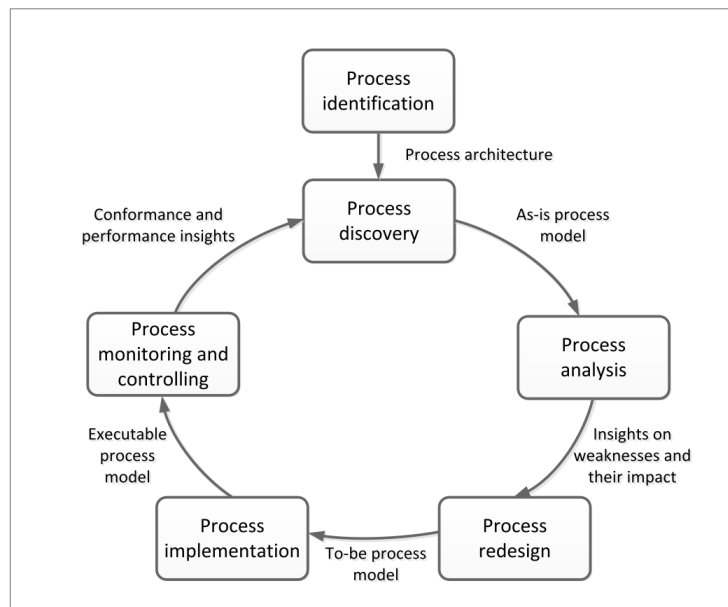


Figure 2.1: BPM life cycle according to [DRMR13]

### 2.1.2 BPM phase process analysis

BPM phase process analysis refers to the systematic analysis of processes in order to gain an understanding of the process and to identify weak points and potential for improvement. On the one hand, it includes a qualitative analysis of the process. On the other hand, and this is why the phase is particularly relevant to this thesis, a quantitative evaluation is done based on historic process execution data.

Not only the calculation of classical performance metrics, but also other process analysis techniques, such as flow analysis, queuing theory and simulation rely on process execution data. Furthermore process execution data is needed to perform process mining. Process mining refers to the extraction of knowledge on event logs to discover, monitor and

improve processes [Vai13] which can also be performed on data from systems which do not execute an explicit process model, e.g. ticketing or Enterprise Resource Planning (ERP) systems.

In this thesis we will focus on classic performance measures of a process which are quantified based on process execution data. This is done in order to determine how well a process is performing with respect to its performance targets or reference Key Performance Indicators (KPIs) from process execution data of similar processes. Typical performance measures are e.g. processing time and waiting time but also quantifications of the dimensions quality, cost and flexibility [DRMR13]. As this work examines potential advantages of using temporal databases in BPM, aspects of process analysis and monitoring which involve a time dimension are specifically of interest.

The consideration of KPIs does not necessarily have to be static, in sense of aggregated aggregated over the entire observation period. Changes or fluctuations over time are often also relevant in process analysis.

### 2.1.3 Temporal data in WfMS

The development of BPM went hand in hand with technological progress. “Information technology in general and information systems in particular deserve an important role in business process management, because more and more activities that a company performs are supported by information systems” [Wes12]. Different types of IT system emerged, most notably Enterprise Resource Planning (ERP) systems and Workflow Management Systems (WfMSs), which enabled a central data retention and easy control but also monitoring of business processes [DRMR13]. Already in 1990, Davenport highlighted the importance of IT for process-centered management and described a “recursive relationship between IT capabilities and business process redesign” [DES89].

For WfMSs relevant process execution data<sup>1</sup> is usually stored in DBMSs. Workflow control data in particular, but also so called workflow relevant data, is “necessary for the operation of the workflow and the realisation of routing” [Jó06]. Whereas workflow relevant data may be manipulated by workflow applications as well as by the WfMS, workflow control data is exclusively “managed by the WfMS and/or a workflow engine. Such data is internal to the WfMS and is not normally accessible to applications” [Wor99].

Different kinds of time-related process execution data have been identified and listed in the following subsections. They might be used as input in process analysis or as basis for the calculation of KPIs in Business Process Management.

---

<sup>1</sup>[Wor99] defines application data as one kind of BPM relevant data. Application data is only known to evoked applications and not relevant for process control and therefore not in focus of the thesis.

### **Event data**

Events within a process execution occur to a certain point in time. This timestamp is captured.

**Examples:** Login to a system or delivery of goods.

### **Duration of processes or process steps**

Usually, a WfMS keeps track of the duration of a process or activity execution. This information can contribute to BPM in many ways. The duration of executions might be analyzed along particular process characteristics such as values of input variables, the actual process path or involved capacities. The duration might also correlate with or even influence defined quality indicators and therefore provides information on potential improvement approaches.

Furthermore, duration is used as basis to calculate various KPIs. The duration is also needed to determine adherence to delivery dates or schedules. Additionally, the identification of tasks with the highest / lowest share on total execution time helps to identify process improvement strategies with the potentially highest impact.

**Examples:** Based on the duration, it might be discovered that whenever a certain resource is involved in a process, the duration is significantly shorter. The customer satisfaction with the process execution might drop at a certain duration or the delivered quality of service might be low when certain process steps are executed too fast (e.g. due to a higher error rate).

### **Validity periods of data**

Instead of deleting data, its period of validity is documented as ended. This helps to analyse changes in data over time or to join the correct process-execution-independent data to time-related execution data.

**Examples:** Process variables might change over time such as the status of a credit approval decision. Another example are changes to the organizational structure of a company, such as assigning employees to a department or taking over the management of a department.

### **Execution periods**

The count of active process executions or execution times of activities might vary over time. The determination of an execution curve helps to recognize seasonal or time-of-day-dependent variations.

**Examples:** The number of unattended customers of an ice cream shop varies over time. In the afternoon, there is a long queue of customers in front of an ice cream shop. In terms of BPM: A lot of processes are started and executed in the afternoon.



### Utilization periods

The utilization of a resource or service might vary over time. The determination of a utilization curve helps to recognize seasonal or time-of-day-dependent variations. Knowing demand for resources helps to make resources available at the right times.

**Examples:** Automatic batch processes of a financial institution run over night. Required computing power needs to be available.

### Idle periods of resources or processes

Idle times of resources or periods of times where no process is executed give insights into possible savings potentials or the possibility to re-allocate resources.

**Examples:** Ice cream shop workers have little to do in the morning. The shop owner might ask them to better work in the afternoon.

## 2.2 Process modelling with BPMN

In BPM process models are often used to document current but also target processes. E.g. in phase process discovery, identified processes are documented in form of as-is process models. In comparison to textual descriptions of processes, a graphical model allows stakeholders to more easily comprehend a process and avoids misinterpretations originating in the ambiguity of text [DRMR13].

Before 2002, only proprietary process execution language definitions existed, such as IBM's Web Service Flow Language (WSFL) and Microsoft's XLANG specification. With the final release of a standardized specification of WS-BPEL 2.0 by OASIS in 2007, a first and quite successful industry wide standard to define execution models for business processes was born. However, it was rather focused on the actual execution of processes and lacking relevant features, such as human tasks or cyclic control flows, to be also used in a more business related context [Rad12].

At similar time, the Business Process Management Initiative (BPMI) released BPMN 1.0 in 2004, "which was widely used as a modeling notation for business processes. As a process developer, you may have received a BPMN 1.x model for requirements or documentation purpose from information or business analysts. But then, you had to convert those models into an execution language, such as WS-BPEL" [Rad12].

With the release of BPMN 2.0 in 2011 as the first standard for modeling *and* implementing a business process execution model, there was no need to convert business-defined process models into implementable execution models anymore. "New features of BPMN 2.0 created a standardized bridge between the business process design and process implementation." [WB11]. Subsequently, it became one of the most well known graphical notations for business process modelling [Wes12]. BPMN 2.0 has been defined by the Object Management Group (OMG) [Obj11], which "is a well-known standardization organization that develops and maintains the Unified Modeling Language (UML) standard,

for example“ [Rad12]. The implementation of BPMN 2.0 processes is supported by a broad range of WfMS, such as Activiti.

BPMN 2.0 describes various constructs to graphically documented business processes e.g. to describe parallel activities, the handling of events and exceptions, gateways influencing the process flow as well as message flows and data objects. Since the standard is very comprehensive and distinguishes a large number of graphic elements, these are often grouped in secondary literature. E.g. Robert Shapiro from Workflow Management Coalition (WfMC) classifies the BPMN 2.0 elements into four categories [Sha10]:

- Simple (8 elements) for high-level process modeling
- Descriptive (+17 elements) for more extended process modeling including data input and -output as well as timer
- DoDAF (+29 elements) for detailed modeling containing a wide range of BPMN 2.0 elements
- Complete palette (+50 elements), offering numerous more specific modeling elements

As part of this work, we limit ourselves to the use of the notations categorised as simple elements, extended by the differentiation of tasks to service and user tasks. All utilized notations can be found in table 2.1.



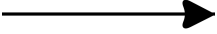
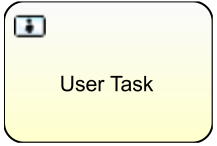
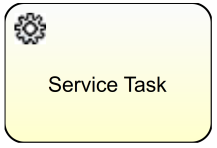


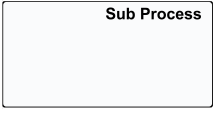
Symbol	Name	Description
	Start event	The start event is the trigger to start a new process instance.
	End event	The end event causes the process execution to terminate.
	Sequence flow	The sequence flow connects activities, gateways and events and therefore represents the orchestration of the process.
	User task	A user task is performed by a human with help of a computer interface. It might be assigned to a single user or a user group.
	Service task	A service task represents an automatic activity, e.g. a web service call or a simple script.
	Exclusive gateway	An exclusive gateway is used for conditional logic. Only one of the outgoing sequences will be followed, based on the condition.
	Parallel gateway	A parallel gateway is used to start simultaneous execution of activities or to indicate that all ingoing process flows need to be ended before continuing the overall process flow.
	Sub Process	A sub process is a compound process containing various BPMN elements such as activities or gateways and can be called from a parent process.

Table 2.1: BPMN notations used in context of this work [Rad12]. Notation according to BPMN 2.0 [Obj11], selection of simple elements according to [Sha10], design as available in Activiti Modeler.

## 2.3 Activiti framework

Activiti is an open-source Workflow Management System (WfMS) to create and execute BPMN 2.0 processes. It is mainly funded by company Alfresco but developed by a broad community of developers. An overview of the framework is provided in figure 2.2, as summarized by Tijs Rademakers in his book *Activiti in Action* [Rad12].

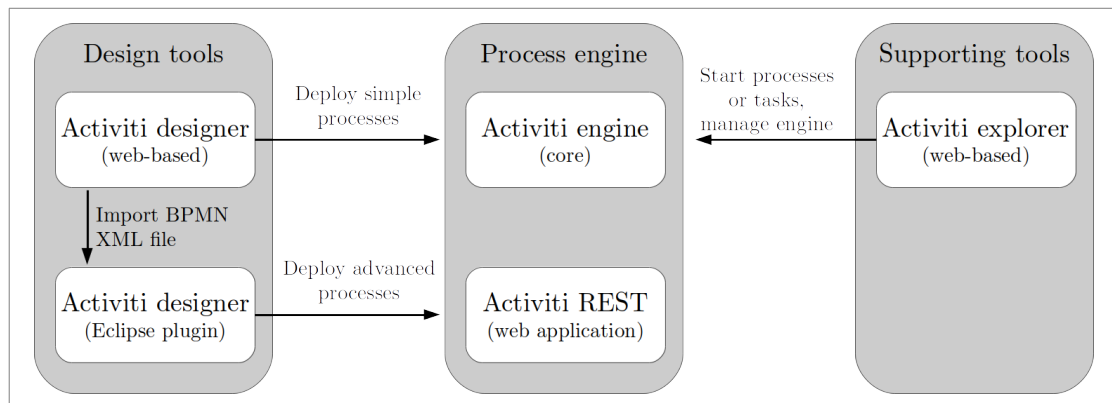


Figure 2.2: Overview of the Activiti tool stack [Rad12]

**Design tools** Activiti offers a web-based graphical modeler to create simple BPMN 2.0 processes and a more powerful, yet more technical, graphical designer as plugin for the integrated development environment (IDE) Eclipse.

**Process engine** "The core component of the Activiti framework is the process engine. The process engine provides the core capabilities to execute BPMN 2.0 processes" [Rad12]. It might be used as stand alone application, but may also be embedded in Java project. The component Activiti REST offers a REST API to interact with the process engine within a distributed system.

**Supporting tool** Additionally, a web-based supporting tool called Activiti explorer is offered to start processes and tasks as well as to manage the Activiti process engine in a web browser.

### 2.3.1 Business process modeling in Activiti modeler

With the Eclipse plugin Activiti modeler, BPMN 2.0 diagrams can be created from scratch or imported from Activiti designer. The modeler offers a graphical interface to edit BPMN 2.0 XML files and, in comparison to the Activiti designer, it offers functionality to add technical details to process models. Thereby, a wide range of BPMN 2.0 XML standard elements is supported.

### 2.3.2 Business process execution in Activiti

Basically, the Activiti process engine is a state machine which executes BPMN elements one by one. It can be easily facilitated in Java applications and therefore e.g. integrated in Maven projects. The process engine uses the standard `java.util.logging` API. Therefore, the Apache Log4J framework can be easily utilized to get better configuration options.

The Activiti API offers various functions to interact with the Activiti process engine. The following core API interfaces are specifically relevant in context of this work and the ones which are most frequently used [Rad12]:

- **RepositoryService** to deploy, query, delete and retrieve process definitions
- **IdentityService** to create users and user groups
- **RuntimeService** to start and query process instances and to retrieve and set process variables
- **TaskService** to retrieve a list of open tasks of a specific user. Additionally, a task can be claimed and completed by a user
- **HistoryService** to retrieve information about completed process instances

After the deployment and start of a process, the Activiti engine “executes the process until a wait state is encountered. A user task is an example of such a wait state” [Rad12]. In this case, additional code needs to constantly query the TaskService to check for queued user tasks. Those can then be handled e.g. by interaction with a user asking him/her for input via the command line or a graphical user interface (GUI) or by using Activiti’s task form within the Activiti Explorer.

Another task type which requires to be handled are service tasks. The execution of programme code is delegated to another class specified in the model definition. This might be a handler class implementing the interface `JavaDelegate`. The according service task business logic needs to be implemented in method `execute`.

In case of script tasks, script code which is defined within the process definition is executed. The integration of business rule tasks with the help of Drools, a business rules management system, is still experimental.

### 2.3.3 Activiti database

The Activiti process stores process execution data in an Activiti database. Activiti can interact with several database technologies, amongst others with the in-memory database H2 and PostgreSQL. The names of all Activiti tables start with prefix `ACT_`, followed by a two-character identification of the use case of the table [AS17]. Please find an overview of table prefixes and their intended meaning in table 2.2.

Prefix	Use Case	Description
ACT_RE_	repository	Static information such as process definitions and process resources.
ACT_RU_	runtime	Runtime data of process instances, user tasks, variables, jobs, etc. Activiti only stores the runtime data during process instance execution, and removes the records when a process instance ends.
ACT_ID_	identity	Identity information, such as users, groups, etc.
ACT_HI_	history	Historic data, such as past process instances, variables, tasks, etc.
ACT_GE_	general data	Data which is used in various use cases.

Table 2.2: Table prefixes in Activiti database [AS17].

For process analysis, tables containing historic data are most relevant, as Activiti removes all information from runtime tables after a process has been completed to keep their size as small as possible. Please find some details on history tables in table 2.3.

Table name	Description
ACT_HI_procinst	For each process-flow, a process instance is created. The table stores information such as processing start and end time. If no end time is set, this means that the process has not yet been completed.
ACT_HI_actinst	Instances of activities (visited process nodes). If no end time is set, this means that the activity has not yet been completed.
ACT_HI_taskinst	Instances of user tasks
ACT_HI_varinst	Latest valid variable values of tasks (no change/update history is stored in this table)
ACT_HI_detail	All updates on process variables; Only filled, if level of historic information to keep has been set to full in <code>activiti.cfg.xml</code> .
ACT_HI_identitylink	Links groups or users as candidates or assignees to tasks.

Table 2.3: History tables in Activiti database.

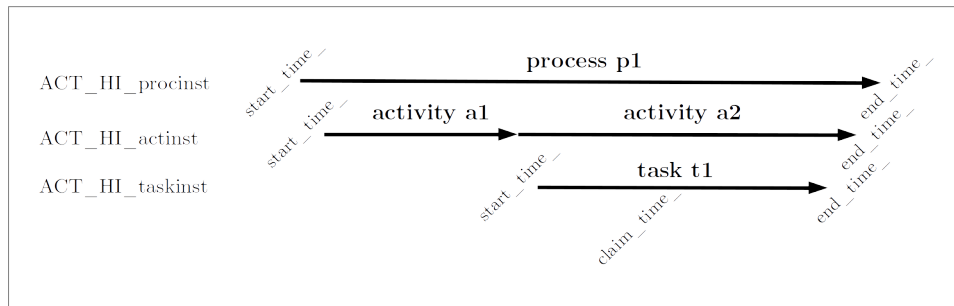
In addition, information about the organization structure is important for the definition of BPMN processes and analysis of workflow control data. Within an Activiti BPMN process definition, users or groups can be defined as potential candidates to complete the task. During process execution, a user is then selected by the process engine and assigned to the task. It is worth to notice that changes to the organizational structure over time cannot be stored (and queried), as data about users, groups and the allocation of users to groups is not historized in Activiti. Please find some details about all relevant identity tables in table 2.4.

Table name	Description
ACT_ID_user	Users and their master data including user id, first name and last name.
ACT_ID_group	User groups that, for example, represent organizational units.
ACT_ID_membership	Allocation of users to one ore more than one group.

Table 2.4: Relevant ID tables in Activiti database.

### 2.3.4 Period data in Activiti

Activiti stores period data on three different levels of granularity, namely process, activity and task level. A process execution period consists of several activity periods. If an activity is a user task, then the task level forms the lowest in the hierarchy. A period will never end until the lower level of the period has ended. All period start and period end times are stored in column `start_time_` and `end_time_` on all three levels. On user level, additionally a `claim_time_` splits the task execution time in two parts: The period part before the claim time represents the time from assignment of a task to a user to the acceptance of the task. The period part from claim time to end time represents the actual processing time of the task. Please see a visualization of the different period levels in figure 2.3.

Figure 2.3: Periods in Activiti database, with activity  $a_2$  being a user task.

As processes, activities and tasks can be performed in parallel, different process, activity or task periods do usually overlap. So, it is very unlikely that tables store disjoint period data.

## 2.4 Managing temporal data in databases

### 2.4.1 Brief history on research and implementations

Temporal aspects in data are highly relevant for BPM as they are crucial for workflow control data, but also for event logs of process executions. Even though there was a growing need to handle temporal data already in the mid-90s when BPM arose, there was only a weak support of temporal aspects as part of database standards for quite a long time [Zan08].

Recently, however, the interest in temporal aspects of data management has increased in science, but also among commercial providers. On the one hand, data organization of temporal data, such as effective storage and index structures were in focus. On the other hand, research and vendors focused on the development of a query language that better meets the requirements for processing temporal data and on the correspondingly optimized query processing. Böhlen et al. provide a comprehensive overview of the history and current state of database technologies for processing temporal data [BDGJ18a].

In recent times, industrial databases have increasingly devoted themselves to the implementation of temporal aspects. “For instance, Teradata supports ‘temporal modifiers’ where queries can be declared temporal. At implementation level they do query rewriting and currently support temporal joins. SAP is also advancing in this field and is starting to implement temporal aggregation and join. PostgreSQL for now supports range types with predicates and functions that are very useful at single tuple level” [Mos16]. However, commercially available software tools continue to still offer quite limited support for temporal data management [BGJ06]. “Database systems largely remain designed for processing the current state of some modeled reality.” [BDGJ18a].

The implementation of temporal features in commercial databases was also triggered by the release of SQL:2011 [ISO11]. Most DBMS vendors have begun to offer limited support for the new standard. SQL:2011 is “arguably the first SQL standard to introduce explicit support for the storage and manipulation of temporal data” [BDGJ18a]. It introduced the standardization of temporal data types, temporal queries and support to store meta data such as valid time, transaction time or decision time [KM12]. However, it still lacks support of more advanced operations, such as various forms of temporal aggregations or temporal joins [BDGJ18a].

The publication of SQL:2011 was preceded by a scientific discourse in which different proposals for temporal query languages were discussed. Snodgras made a first comprehensive proposal with TSQL2 [Sno95]. Others followed, such as IXSQL, SQL/TP or ATSQL [BDGJ18a]. Some temporal queries are still more easy to formulate in these temporal query languages, than in standard SQL:2011 [DBGJ16].



So, even though standards have been extended and some conventional "SQL-based DBMSs are capable of supporting the management of interval data, the support they offer can be improved considerably". Commercial DBMSs implementations focus "on the representation of intervals and neglecting the implementation of the query evaluation engine. [...] While the querying of temporal data is quite well understood, the key remaining problem is how to achieve an industrial-strength and systematic DBMS implementation of a comprehensive temporal query language" [DBGJ16].

### 2.4.2 Temporal extension of a DBMS at Free University of Bozen-Bolzano

With the *temporal alignment framework*, Dignös, Böhlen, Gamper and Jensen from the Free University of Bozen-Bolzano introduced the "first approach to achieve systematic and comprehensive support for so-called sequenced temporal queries in relational database engines without limiting the use of queries with so-called nonsequenced semantics" [BDGJ18a]. The term sequenced semantics refers to queries that go beyond the consideration of data at a static point in time, i.e. queries that consider the data and its development *over time*. With their research prototype, the Free University of Bozen-Bolzano adapted the evaluation engine of a common DBMS, the proposed solution was integrated into the kernel of a PostgreSQL. Their approach reduces temporal queries to non-temporal queries over data with adjusted intervals. "It integrates, in a systematic and wholesale manner, temporal support into an existing system without affecting the system's support for non-temporal queries" [DBGJ16].

The query processing described in the temporal alignment framework was implemented as PostgreSQL extension. The reduction of temporal queries to non-temporal operators is performed in four steps [BDGJ18b]:

1. **Timestamp propagation:** Period timestamps used as temporal arguments in query statements are replicated, so that the original values can be used in subsequent steps after intervals have been adjusted. Original timestamp values are used to scale attribute values and to evaluate query predicates and functions that reference the original timestamps.
2. **Interval adjustment:** Technically, "interval adjustment is achieved by introducing two new relational operators, a temporal normalizer and a temporal aligner, into the database engine" [DBGJ16]. The normalizer splits relations so that periods of temporal arguments do not overlap. Duplicates of input relations are created and new sub-interval timestamps are assigned in a way that the sum of intervals of the newly created tuples represent the whole period of the original relation. The temporal aligner has been designed for tuple-based operators and "adjusts an argument tuple according to each individual tuple of a group".

3. **Attribute value scaling:** Certain attributes of the newly created tuples need to be scaled according to their new (shorter) intervals. “For example, attributes that record total (cumulative) quantities over time, such as project budgets, total sales or total costs, often must be scaled if the timestamp is adjusted” [DBG13], if the relationship has been split into several shorter periods. The scaling of the attributes can be done according to user-defined functions (e.g. uniform or trend scaling).
4. **Evaluation of non-temporal operator(s):** The preparatory steps now make it possible to process the temporal query as a non-temporal query. Therefore, the temporal query has been transformed to a non-temporal query, utilizing standard operators and the two new operators `normalize` and `align`. Subsequently, the non-temporal operators can be evaluated utilizing the standard operators of the DBMS.

Compared to other implementations, this implementation stands out for the following unique advantages [DBGJ16]:

- All operators of a comprehensive sequenced temporal algebra are supported.
- Tight integration into existing non-temporal DBMS, leveraging existing query optimization and indexing techniques. Instead of introducing new evaluation algorithms for temporal operators, temporal queries are reduced to non-temporal statements with minimally invasive changes to the DBMS.
- It supports snapshot reducibility, change preservation, extended snapshot reducibility and attribute value scaling - features being identified to be relevant for temporal data processing by Böhlen and Jensen [BJ09]. “Snapshot reducibility ensures that each snapshot in the result of a temporal operator is equal to the result of the equivalent non-temporal operator evaluated on the corresponding snapshots of the argument relations” [DBG12]. Change preservation ensures that a natural and unique grouping of time points into intervals is not dissolved, even though it would be possible semantically (e.g. merge of two directly successive periods with the same values in other input parameters). Extended snapshot reducibility means that references to interval timestamps can be used along with snapshot reducibility.

Core of the implementation of the reserach prototype are temporal primitives `normalize` and `align` and the respective changes to the query engine. Additionally, an extension on SQL:2011 is proposed, which advances the standard at operator level, so that temporal queries can be rewritten into their non-temporal counterparts. Currently, the following rewrites are supported in the implemented PostgreSQL extension: `SELECT PERIOD DISTINCT`, `GROUP BY PERIOD`, temporal joins except anti-joins, and temporal set operations `UNION`, `EXCEPT` and `INTERSECT`.

Known limitations of the SQL extension are that only binary join operations and distinct non-ambiguous column names can be used for operator `GROUP BY PERIOD`. “The second problem is that [the utilized] Common Table Expressions (CTEs) do not support selection push-down, or similar performance enhancements, because they get parsed, optimized, and executed as independent queries” [Mos16].

## 2.5 Complexity metrics for SQL queries

To retrieve process execution data from a database, respective query statements need to be developed. This might be done during the implementation of a BPM analysis software to offer users a comfortable analysis cockpit. Or, data might be queried directly by an end-user within process analysis if he/she is capable to do so. Irrespective of who is coding those queries: Complex statements lower the readability of query statements (for the user or the software engineer) and increase the susceptibility to errors [BW10]. Therefore the complexity of queries is a key dimension of the benchmark.

Unfortunately, not all software complexity metrics are applicable to be used for SQL statements. A database query is not a usual program code that implements sequential program flows. Code complexity metrics that result from the analysis of a program flow or are based on a detailed interpretation of the program code are not applicable for SQL statements. An early, but well known, example of such non applicable metric is McCabe’s cyclomatic complexity, a quantitative measure of the number of linearly independent paths through a program’s source code [McC76].

When measuring the code complexity of SQL statements, the focus must be on the actual program code (SQL code) itself. Facing a similar problem, Bowen et al. followed the same approach, utilizing Halstead’s program length as complexity measure in their work. They used one dimension of Halstead’s metrics for code complexity, the program length: “The total number of operators and operands in each model query were used to measure complexity. [...] To calculate the complexity value, each operand and operator was given a count of 1. Pairs of symbols, e.g., “()”, were given a count of 1” [BFLR03].

Halstead bases several complexity metrics on program length, but also on program vocabulary [Hal77]. The program length counts the total occurrence of operators and operands, as it was applied by Bowen et al. The program vocabulary counts distinct occurrences of operators and operands. As Halstead bases his metrics on operators and operands, they are less sensitive to the actual code layout as other metrics, such as lines-of-code [Ver19].

An example for the computation of the program length  $N$  and vocabulary  $\eta$  for an SQL query can be found in table 2.5. To make the example easier understandable, a distinction is made between operators (subscript 1) and operands (subscript 2) in a first step to then calculate the metrics for the whole query.

Query	$\eta_1$	$\eta_2$	$N_1$	$N_2$
SELECT NAME, ACTINST.DURATION	3	3	3	3
FROM PROCINST, ACTINST	2	2	2	2
WHERE PROCINST.PROC_INST_ID = ACTINST.PROC_INST_ID	3	4	4	4
AND ACTINST.DURATION > (0.2 * PROCINST.DURATION);	6	5	7	5
Total, differentiated between operators (1) and operands (2):	11	8	16	14
Total query length $N$ and vocabulary $\eta$ :	19		30	

Table 2.5: Example query complexity computation, based on [Hal77].

Other Halstead metrics, namely programme volume, difficulty and effort are calculated based on the determined program length and program vocabulary. Halstead’s basic idea is that an expression is “easier to understand if it is shorter or contains more redundancy of operators and operands because the number of different concepts that a programmer must retain in memory at once will be smaller” [OW10].

Halstead’s volume describes the size of the implementation of an algorithm based on the number of operations performed and operands handled in the algorithm:

$$\text{Program volume } V = N * \log_2(\eta_1 + \eta_2)$$

The difficulty is postulated to be proportional to the number of distinct operators:

$$\text{Program difficulty } D = \frac{\eta_1}{2} * \frac{N_2}{\eta_2}$$

The effort is a function of volume and difficulty and describes the effort to implement or read a code:

$$\text{Program effort } E = V * D$$

## Methodology

This work will identify potential benefits of using a temporal database in Business Process Management. It will focus on potential advantages of temporal databases when working with already existing process execution data, with an emphasis on database operations performed within process analysis and monitoring. This work will not examine potential advantages of using temporal databases for write, update or delete operations during workflow process execution.

Within a benchmark, the performance of a temporal database in query processing will be compared to the performance of its non-temporal equivalent. The following major steps have been carried out:

1. Definition of benchmark dimensions and identification of BPM relevant queries which can be used as basis for the benchmark
2. Setup of a benchmark environment to execute queries and measure query processing performance in a temporal and a non-temporal DBMS
3. Definition of a sample process (BPMN model), whose process execution data can serve as data set to be queried during the benchmark
4. Generation of simulated process execution data for the defined BPMN model

### 3.1 Benchmark approach and setup

In order to compare a temporal and a non-temporal database and their capabilities relevant to BPM as well as their performance in querying workflow control data, appropriate metrics for quantification must be defined. Furthermore, a suitable setup to gather data in order to calculate these metrics need to be specified. Therefore, BPM relevant database queries which are to be used during the benchmark need to be defined. Additionally, since several measurement points are expected per metric and database technology, the statistical handling of the collected data must be defined.

### 3.1.1 Definition of benchmark dimensions and metrics

For process analysis and monitoring, no data manipulation is expected on process execution data. Temporal databases must therefore simplify and/or improve the performance of data retrieval in order to achieve added value in BPM. Consequently, the following benchmark dimensions have been defined along database operations to *retrieve data*, rather than *manipulate data*:

## Complexity of query statements

To determine and compare the complexity of query statements, Halstead complexity metrics will be calculated for the temporal and the non-temporal query statement. Halstead metrics are based on the total number of operands and operators of a statement (program length) and the distinct count of utilized operands and operators (program vocabulary). An example for the computation of the program length and program vocabulary for an SQL query can be found in table 2.5. Other metrics are calculated based on these figures. Details on the different Halstead metrics can be found in chapter 2.5.

To identify operators, the implementation of the complexity calculation will be based on operators and functions of a temporal PostgreSQL 9.6 installation. Therefore, a list of operators has been created based on dictionary tables `pg_proc`, `pg_operator` and `pg_get_keywords` (with category code  $R_{reserved}$ ).

Additional to the extracted operators from dictionary tables, 23 operators<sup>1</sup> have been added to the list based on the documentation of operators in the PostgreSQL user manual. Furthermore, the following symbols have been added as operators, as they have been missing so far: '(', '[', '{', ')', ']', '}', '\*', '::', ':', ';', ',',

To calculate the complexity value, we follow the same approach as Bowen et al. and give each operand and operator a count of 1. Pairs of symbols are seen as one operator [BFLR03]. Implementation wise, only the opening token should be counted while closed tokens should not be considered when counting operators.

<sup>1</sup>Added PostgreSQL operators based on the manual: xip\_list, xmax, xmin, is not distinct from, is distinct from, not in, exists, least, greatest, nullif, coalesce, case when, is document, xmlroot, xmlpi, xmlforest, xmlelement, xmlconcat, at time zone, extract, similar to, trim, !=

### Execution performance of query

For the user it is relevant how long it takes from sending a query to getting results, including processing time within the business logic of his/her administration tool, network transmission costs and I/O conversion costs. Even though the complete processing time is in focus of user experience, this work will only focus on the time to retrieve a result set for a given query statement within the DBMS, as only differences in database technology should be compared.

Usually, the execution time of queries can be split in two parts:

- **Planning time** (or start-up time) before the first row can be returned
- **Execution time** to return all the rows

The query planning time reflects the start-up costs before the first row can be returned. Essentially, it is the time the query planner of a DBMS takes to determine the most cost efficient way to retrieve data. Whenever a full set of data is required, the execution time is of major interest. If only a first row needs to be fetched (e.g. queries containing **EXISTS** operator), the planning time is the more relevant measure [Gro19], as the smallest start-up costs instead of the smallest data retrieval costs are of relevance.

Most DBMS provide tools to measure these two performance metrics. E.g. in PostgreSQL, command **EXPLAIN ANALYZE** can be used to retrieve this information. A sample output of this command can be found in table 3.1.

EXPLAIN ANALYZE command	
<pre>EXPLAIN ANALYZE   SELECT * FROM ACT_HI_PROCINST   JOIN ACT_HI_ACTINST ON ACT_HI_PROCINST.ID=ACT_HI_ACTINST.PROC_INST_ID_</pre>	
EXPLAIN ANALYZE query plan (output)	
1	<pre>Hash Join  (cost=4.22..8.49 rows=11 width=3654) (actual time=0.060..0.099 rows=36 loops=1)   Hash Cond: ((act_hi_actinst.proc_inst_id_)::text = (act_hi_procinst.id_)::text)     -&gt; Seq Scan on act_hi_actinst  (cost=0.00..4.11 rows=11 width=1320) (actual time=0.0...     -&gt; Hash  (cost=4.10..4.10 rows=10 width=2334) (actual time=0.032..0.032 rows=4 loops=1)         Buckets: 1024  Batches: 1  Memory Usage: 9kB         -&gt; Seq Scan on act_hi_procinst  (cost=0.00..4.10 rows=10 width=2334) (actual t...</pre>
2	Planning time: 0.351 ms
3	Execution time: 0.146 ms

Table 3.1: Example for an **EXPLAIN ANALYZE** command and its output

#### 3.1.2 Characteristics of benchmark queries

Usually, process execution data is analyzed in BPM phase process analysis as well as during process monitoring and controlling as described by [DRMR13]. The queries used to measure the differences of query execution performance will be formulated to meet data requirements originating in these BPM phases.

The query statements should be selected so that all features of the temporal database are covered in the benchmark. Also all relevant data requirements on work flow control data within BPM phases process analysis and monitoring should be covered.

The query statement for the temporal database might differ from the one for the non-temporal database, so that temporal features are utilized. However, the defined queries should be similar in structure and approach as well as produce the same result set in the temporal and the non-temporal database. If query statements can be formulated differently, the one with the highest expected execution performance should be chosen. Both queries, the one for the temporal and the one for the non-temporal database, should be constructed as efficient as possible to achieve minimum execution times from a query design point of view. Bad execution times should not result from bad query design, but rather from the database's capabilities in processing it.

#### 3.1.3 Benchmark data evaluation and interpretation

Within the benchmark it is relevant to avoid capturing one-time effects when executing a query. Therefore, queries should be executed more than one time and the average and/or median execution and planning time should be considered.

To determine, if the average execution (or planning) times are significantly different in the temporal and the non-temporal DBMS, a statistical test is going to be applied on the captured data points per query. The two samples will have the same sample size and will be unpaired according to the benchmark setup and are assumed to be normally distributed. If the assumption is correct, a Welch's t-test (t-test for samples with unequal variances) should be applied. This test can be applied to samples with equal and unequal variances without any substantial disadvantages compared to a Student's t-test (t-test for samples with equal variances) [KRM09, RKM11].



## 3.2 Benchmark architecture

This chapter describes the technological basis on which the benchmark was carried out. The choice of the DBMS and the underlying data model are discussed and requirements for the data set of process execution data are defined.

### 3.2.1 DBMS technologies

With the extension of a PostgreSQL database by temporal aspects, the prototype of the University of Bozen-Bolzano enables a direct comparison between the use of a temporal and non-temporal database in BPM. As it is not an entirely new DBMS, but a modification of an already existing DBMS [DBGJ16]. Therefore, the benchmark can focus on the (potential) advantages utilizing the additional temporal functionality in context of BPM.

Thus, two PostgreSQL databases of the same release, one with and one without temporal extension, are used for benchmarking query processing, having the same system resources assigned. To compare query execution on a temporal and non-temporal database, the two databases need to be running under the same conditions, in regards to hardware configuration and setup. In both databases, the same workflow execution data needs to be available. To ensure that, the data of one database will be entirely copied to the other database.

### 3.2.2 Data model for process execution data

In order to make the benchmark as accurate as possible, the data structure of the process execution data and thus the data model of the database used should be as close as possible to databases of “real-life” Workflow Management Systems (WfMSs). So it was our first choice to actually build our prototype on the database of an already existing conventional WfMS and not to design our own database or to use a database of a pure process simulation tool.

Therefore we needed to identify a WfMS fulfilling the following requirements.

- The software is mature enough to be potentially used in business context. In best case, a dedicated business suite is offered.
- A comprehensive Java Application Programming Interface (API) is offered, so that the simulation application can easily utilize the process engine of the WfMS to create simulation process execution data.
- A certain relevance in scientific publications has been achieved.
- It can be used under a cost-free license in scientific context, so that our work can be reproduced easily by the scientific community.

Activiti, as one of the leading open-source BPMN engines, fulfills these needs and does also offer support for various database back-ends, such as PostgreSQL [AS17]. It can be seen as continuation of another business process modelling framework called jBPM [Rad12] and is backed by Alfresco, which does also offer an enterprise version of Activiti. A third viable candidate would have been Camunda, an Activiti fork for which also an enterprise version is offered. As Camunda's significance in literature is rather small compared to Activiti and as the data structures of both are anyhow quite similar, it was decided to use Activiti's database to store process execution data for the benchmark. Other competitors which have been evaluated for our purpose were Intalio and Bonitasoft [BB13]. But as they are more tool-based and not so developer-focused [Rad12], Activiti seemed to be the better option for our purpose.

#### 3.2.3 Execution data and process model characteristics

To measure differences in performance when executing database queries, an appropriate set of process execution data has to be chosen or generated. The data set must be dimensioned large enough so that possible differences in performance can be detected in the two databases. Furthermore, it should be legally possible to publish the data set so that the benchmark can be reproduced on the same basis.

The process execution data's underlying business process needs to be documented in form of a BPMN 2.0 model. To allow various BPM relevant queries, the process needs to at least have the following characteristics:

- Possibility to have process iterations with different durations, correlating with characteristics in input data, resource allocation or process paths
- Possibility to have process iterations following different paths
- Process steps performed by more than one resource with different capacities
- Potential idle time of resources
- Organization of resources in organization units
- Utilization of internal and external resources
- Parallel execution of activities and processes

Unfortunately, no existing Activiti data set was found that meets the requirements defined at the beginning of this chapter. In addition, using an existing dataset would have meant that we would have been limited to the corresponding business process definition. Defining a sample business process specifically for the purpose of the benchmark allows us to define it in such a way that a variety of BPM relevant analysis and a broad range of different temporal queries can be applied to its subsequent process execution data. Therefore we decided to use simulated process execution data. The definition of the corresponding process will be documented in the form of a BPMN diagram.

### 3.3 Requirements on supportive artifacts

Two technological artifacts have been implemented in order to perform the benchmark: A benchmark application for the repeated execution of queries and for the calculation, processing and storage of measurement data. And a simulation application to create different sets of workflow control data which can be used as basis for the benchmark.

#### 3.3.1 Requirements on benchmark application

A user should be able to conduct the benchmark in an easy and fast way. Therefore, a Java application has been implemented, which copies data from an existing database within a non-temporal PostgreSQL database into a temporal PostgreSQL installation to then automatically execute queries in both databases and measure their performance. This Java command line program had to implement the following requirements:

**Repeated execution of queries** When queries are executed only once, the execution performance might be affected by one-time effects, such as short-term CPU load from other processes, but also by the possible execution of database functions that are performed during the initial execution of a query. To avoid these factors influencing the measurement of query performance, each query statement should be executed several times on the same DBMS. Hereby, the count of iterations should be parameterizable by the user.

**Capturing of query execution performance measures** The application executes given query statements in two databases and captures performance measures of the query execution. It documents planning and execution time and detailed execution cost analysis. When measuring the query execution performance, database external factors influencing the execution time (e.g. network latency) should be excluded. This is achieved via using PostgreSQL's command `EXPLAIN ANALYZE`.

**Calculation of query complexity** The benchmark application calculates the queries complexity according to Halstead's complexity measure. Hereby the program length and vocabulary per query statement should be calculated. The special features of the new temporary SQL dialect should be handled accordingly.

**Dynamic load of queries** Query statements whose performance is to be measured shall be provided by the user in form of a comma-separated values (CSV) file. This file should be dynamically read by the benchmark application for the application's resources.

**User configured database connections** The connection properties to both databases should be easily configurable by the user and provided via a properties file in the application's resources.

**Automatic copy of database data** If both databases do not yet have the same data available, the user might facilitate a benchmark application's feature to remove all data from one of the two databases and copy data from the other database.

Constraints defined in the default Activiti database are non-deferrable and therefore checked immediately at every statement. To enable bulk data copies, all constraints should be altered to deferred, so that they are checked not until the transaction is committed.

**Providing results as CSV files** For each simulation run, two CSV files have to be created to document results:

- **aggregated results**, capturing performance measures per query including calculated average and median execution and planning times as well as Halstead complexity measures.
- **detailed results**, capturing performance measures per query as well as entire PostgreSQL's query plan of each query execution.

**Verbose output** Information regarding the status of the benchmark execution should be visible to the user, either via command line or a log file. The level of granularity of information should be configurable as well as the output format.

#### 3.3.2 Requirements on data simulation

To generate workflow control data of for a given business process definition, an application to simulate Activiti process executions has been implemented. It takes a BPMN 2.0 model as input and generates Activiti process execution data, utilizing Alfresco's Activiti process engine. This Java command line program had to implement the following requirements:

**Random start and execution of processes** The application should randomly start and execute processes according to a given process definition. It generates process execution data for one to several process iterations and stores it into a PostgreSQL database.

**Dynamic load of BPMN model** The simulation application should be able to perform simulation runs for any given business process definition. Therefore, it takes a BPMN 2.0 model and a corresponding data simulation class as input. The BPMN model should be defined as specified by the Object Management Group in its BPMN 2.0 standard [Obj11]. The data simulation class adds business logic for the simulation data creation. It should be specified by the user so that it matches to the corresponding business process and its user and service input requirements.

**Provision of CSV mockup data** A mechanism shall be implemented which reads a CSV file from the application's resources and provides the data to the data simulation class via an interface. This makes it possible to easily use mockup data in the data simulation class.

**Configuration of simulation parameters** The count of to-be-started process iteration as well as the overall time frame for the simulation should be parameterizable for each simulation run. Furthermore, a minimum and a maximum execution time of one process execution should be configurable.

**Verbose output** Information regarding the status of the process execution should be visible to the user, either via command line or a log file. The level of granularity of information should be configurable as well as the output format. Additionally, the process execution history captured by the Activiti process engine should be printed to the command line on user's demand.



# Implementation

This chapter documents the implementation of the simulation and the benchmark application and describes how they can be used. It does also describe the BPMN 2.0 model of the process for which process execution data is simulated. Additionally, the queries which are used within the benchmark are defined.

## 4.1 Application architecture and shared resources

Although both applications, the simulation and the benchmark application, work independently, they share configuration files and some generic helper classes. This was the reason to implement them within one Java Maven multi module project persistence-OfWorkflowControlData. The Java project contains three modules, implemented as sub-projects:

- **Simulation Application** (pwcSimulation) - Executes a BPMN process utilizing the Activiti process engine, simulating user- and service tasks
- **Benchmark Application** (pwcBenchmark) - Compares the performance of query executions in two Activiti databases. To do so, the content of database 1 is copied to database 2 beforehand.
- **Shared** (pwcShared) Shared classes, such as PostgreSQL and CSV file handlers, but also shared resources, like SQL scripts and the `log4j.properties` file.

Figure 4.1 visualizes the application architecture schematically. This section describes classes, resources and input files which are shared between both applications. The following sections describe details of classes, resources and output files of the two implemented applications.

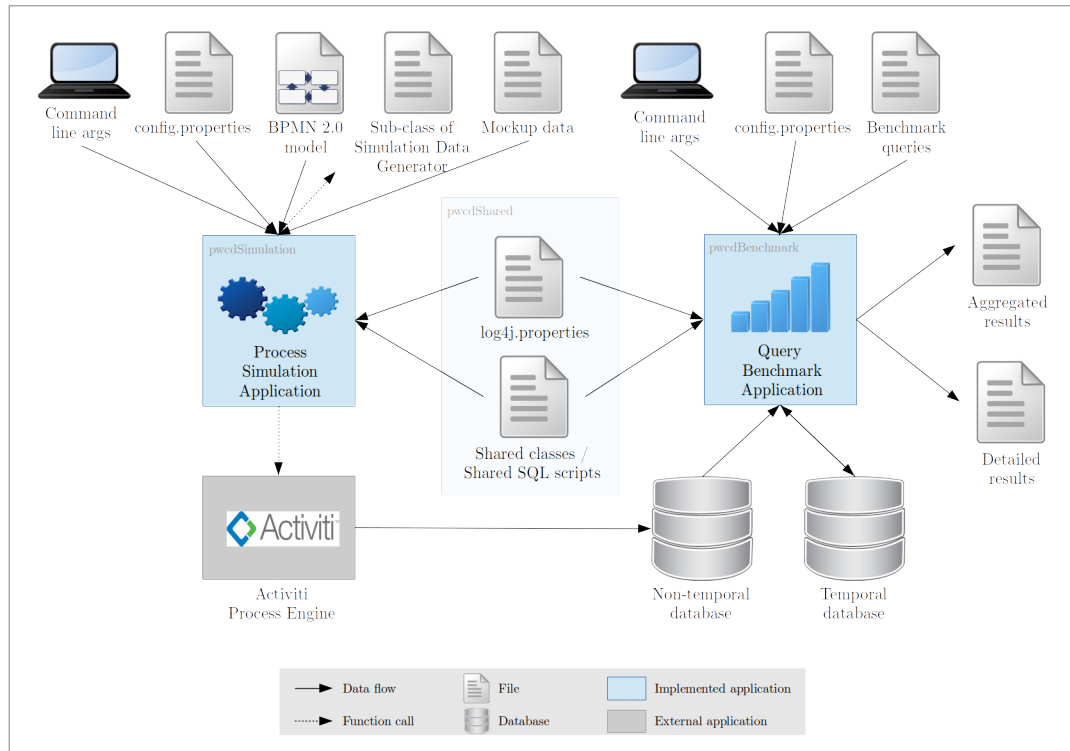


Figure 4.1: Schematic visualization of the application architecture

#### 4.1.1 Technical setup and external libraries

The applications were developed according to Java 11 and tested with PostgreSQL databases in version 9.6. The following external Java libraries are used:

- **Activiti** 6.0.0 Activiti BPMN engine
- **Log4j** 1.7.21 Apache Framework for logging
- **PostgreSQL** 42.2.5 PostgreSQL JDBC driver
- **OpenCSV** 4.6 Library to parse CSV files
- **CommonsCLI** 1.4 as API for parsing command line options

The applications use an Activiti database in version 6.0.0, which has been installed on a standard PostgreSQL database installation and a temporal PostgreSQL installation, by using the respective setup scripts provided by Alfresco [AS17].



### 4.1.2 Shared classes

Similar to external libraries, a common project module named `pwdShared` offers context free, generically implemented classes with mostly static methods to both applications. These classes are grouped in Java package `com.ext` and intended to be used by the simulation and the benchmark application but maintained within the shared project module at a single point of truth. Partly, they contain methods provided by external third parties. The following classes are contained in the package:

#### **`com.ext.PostgreSQLHelper`**

Contains static helper methods for interacting with a PostgreSQL database. It provides methods for initializing a `DataSource` and checking its functionality. It also provides methods to read SQL scripts from a file in the application context to then execute them in a PostgreSQL database. It provides methods to copy data from one database to another database. And there is a method which compares two `ResultSets` to determine if they are equal.

#### **`com.ext.CsvFileHelper`**

Contains static helper methods to read from and write to CSV files, utilizing the external library `OpenCSV`.

The method to read CSV data returns an `ArrayList` whose elements represent rows of the CSV file. Technically, a row is represented as `HashMap` with the CSV column header name as key. Therefore, files to be written require to contain a header. The write method requires a two dimensional `String` array, representing rows and columns of the CSV file.

### 4.1.3 Shared scripts

As both applications offer a database clean-up functionality, respective database scripts are maintained in the shared project module in resource folder `activiti_postgres_database`. The following scripts are used to remove data from an Activiti 6 database, to prepare the database structure for bulk data removals, but also to improve database query capabilities by adding indexes to relevant columns.

#### **`alterConstraintsToDeferrable.sql`**

Alters all constraints in Activiti database to `deferrable`, so that they are checked not until a transaction is committed. This enables e.g. bulk data copies.

#### **`addIndexes.sql`**

Alters tables `ACT_HI_PROCINST`, `ACT_HI_ACTINST`, `ACT_HI_TASKINST` to add indexes to columns of data type `timestamp`.

Additionally, indexes are added to the following columns acting as foreign keys in the benchmark setup: `ACT_HI_TASKINST.assignee_` and `ACT_HI_ACTINST.task_id_`.

### **removeAllData.sql**

Removes data from all Activiti database tables. It does not consider the correct order to avoid foreign key violations and therefore needs to be executed within one transaction and deactivated constraint checks on statement level.

### **4.1.4 Shared configuration files**

To avoid the same configurations being maintained in parallel in both applications, configuration files of the two applications might be maintained centrally in context of the shared module. By default, however, only the `log4j.properties` is maintained in the resource folder of the maven module `pwcdShared`.

### **log4j.properties**

Configuration file as to be interpreted by Apache's logging service framework Log4j. The two applications utilize logging levels `DEBUG`, `INFO`, `WARN` and `ERROR`. Among other log4j properties, the output format can be configured. Default values as configured in the default project setup are log level `INFO` with output on console. The `ConversionPattern` has been slightly adapted to get a better readable output.

## 4.2 Simulation application: Generating Activiti process execution data

Goal of the application is to simulate process executions of a given business process. Therefore, the application takes a BPMN process model as input to then start a specified number of process executions within a given time frame. If the process model contains user or service tasks, the simulation application will automatically create data for input requirements of these tasks instead of asking a user for input or calling external services for data. Additionally, the application might also simulate task processing times, blocking the execution of a process for a certain period of time. In case of user tasks, the respective user will also not work on other tasks in parallel (as he/she is blocked as a resource).

To enable the simulation application to specifically create data for a defined business process, a second file needs to be provided to the simulation application. This file defines how the data of user and service tasks should be created. Therefore, a subclass of `SimulationDataGenerator` needs to be developed, which implements two methods: One method to define the generation of simulation data for user and service tasks of the process. And another method which defines processing times of the tasks.

For the process execution, the BPMN engine Activiti is utilized. Generated process execution data is stored in a corresponding Activiti PostgreSQL database. The runtime of the application equals the time frame of process executions, which means, if simulated process execution data of one week should be generated, the application needs to run for one week.

This chapter documents the application and its capabilities. Chapter 4.3 describes how the application has been used to generate data for the benchmark of a temporal database. It can also be seen as example on how to use the simulation application.

### 4.2.1 Installation, prerequisites and configuration

The application is part of a Java Maven multi module project called `persistenceOf-WorkflowControlData` and is dependent on another module called `pwcdShared` which contains configuration files, script files and generic helper classes. Basically, the maven project is configured in a way that commands like `mvn clean verify` and `mvn compile package` work as to be expected.

To successfully compile the project, dependent libraries need to be available in java class path. The libraries have been listed in chapter 4.1.1 (see also the respective `pom.xml` of the root project and the simulation module). To successfully run the application, an Activiti database in version 6 within a PostgreSQL database installation is required to be running. Information on how to set up the database is available in the Activiti User Guide [AS17].

If a user wants to generate data for his/her own BPMN process, he/she needs to make the following files available as resource in the application context:

- Process definition file (BPMN 2.0 XML)
- Implementation of Java class `SimulationDataGenerator` to generate simulated data
- Optionally, mock-up data (CSV file) to be used in the `SimulationDataGenerator` implementation

Before running the application, the configuration file of the project needs to be adapted. Among other things, the database credentials must be specified and references to the user files must be set accordingly. All available configuration parameters which are to be set for the simulation module can be found in tables 4.1. The settings in file `config.properties` are assumed to be stable over time, containing general settings and process related, stable configurations.

Property name	Description
<code>db.*</code>	Configuration of a database connection to an Activiti database instance in PostgreSQL
<code>process.file</code>	Path to the BPMN 2.0 model definition. The file needs to be available in the application's context
<code>process.key</code>	Key of the BPMN model, as specified in the respective XML tag <code>id</code>
<code>process.simulationDataGeneratorClass</code>	Full qualified name of a class implementing class <code>SimulationDataGenerator</code> to generate input data for task fields

Table 4.1: Mandatory config.properties for simulation application

Table 4.2 lists optional parameters which can be provided by the simulation application's user. All optional properties are related to the process whose executions should be simulated. The properties allow the use of extended functionality when implementing the `SimulationDataGenerator` class. E.g., the minimum and maximum process execution time can be retrieved or mock-up-data provided within a CSV file can be easily accessed. If provided, the simulation application ensures the existence of a user / group hierarchy in the Activiti database. However, please mind, if no users are specified, only a single user is simulated who is not processing any tasks in parallel.

Additional details on how single parameters change the behaviour of the application can be found in the subsequent sections describing the capabilities and functionality of the simulation application in detail (e.g. section 4.2.3).

Property name	Description
<code>process.maxProcessExecutiontimeInSec</code>	Maximum run time of a single process execution in seconds; The time frame to start processes will be lowered by the maximum process run time.
<code>process.minProcessExecutiontimeInSec</code>	Minimum process run time in seconds
<code>process.userGroupMemberships</code>	List of users and their group membership(s) to assign user tasks; Format of configuration property: <code>userA,groupOfUserA,userB,groupOfUserB,userC...</code>
<code>mockup.csvfile</code>	Path to a CSV file containing mockup data. The file needs to be available in the application's context
<code>mockup.csvSeparator</code>	Separator used in the CSV file

Table 4.2: Optional config.properties for simulation application

In the default installation of the project, two sample processes including classes to generate simulated data are already available to be able to run a simulation out-of-the-box, if the database connection has already been configured correctly:

- BPMN process definition `creditApproval.bpmn20.xml` and the corresponding data generator class `com.pwcd.dat.CreditApprovalSimulationData`
- BPMN process definition `onboarding.bpmn20.xml` and the corresponding data generator class `com.pwcd.dat.OnboardingRequestSimulationData`

#### 4.2.2 Running the app and command line arguments

To run a simulation, the `main` method of class `ProcessSimulation` needs to be called. To easily start the application, the generated jar is configured to be executable, having the `ProcessSimulation` class set as manifest's `mainClass`.

The command line arguments can be used to set additional execution-specific parameters like the count of to be simulated processes or the time frame within the processes should be started. All available command line arguments are documented in table 4.3.

In table 4.4 two example command line calls of the simulation application are listed. In row one, an application call can be found, which is recommended after a new Activiti database has been setup to initially adapt it to the simulation application's needs. In row two, a usual application call to simulate a simulation is shown.

Argument	Description
e	Adjustments to the Activiti database after initial installation. The application alters constraints to <b>deferrable</b> , to enable database resets.
r	Reset of the Activiti database. All data of the database will be deleted without any further warning.
g	Generate process simulation data, executing i iterations of the process in s seconds.
i <num>	Option to specify count of process iterations (default 4)
s <num>	Option to specify maximum time frame to start and end process executions in seconds (default 30). The approach of working with a time frame specified via command line arguments makes it easy to define this parameter at runtime, without having the need to rewrite any code or process definition.
h	To show Activiti history for simulated process runs

Table 4.3: Simulation application: Command line arguments

Description
1 Initial adaptations to Activiti database (to be executed once)
2 Removing all data from Activiti database
3 Generating data of 10 process iterations, starting within a timeframe of 30 sec.
Command
1 <code>java -jar pwcdSimulation/target/ pwcdSimulation-&lt;version&gt;-jar-with-dependencies.jar -e</code>
2 <code>java -jar pwcdSimulation/target/ pwcdSimulation-&lt;version&gt;-jar-with-dependencies.jar -r</code>
3 <code>java -jar pwcdSimulation/target/ pwcdSimulation-&lt;version&gt;-jar-with-dependencies.jar -gh -i 10 -s 30</code>

Table 4.4: Simulation application: Examples for command line calls

### 4.2.3 Documentation of classes and program flow

This section contains details on the implementation of the simulation application. The classes specifically implemented for the simulation application are grouped in Java package `com.pwcd.sim`. For user provided implementations of class `SimulationDataGenerator` it is recommended to use package `com.pwcd.dat`.

#### **`com.pwcd.sim.ProcessSimulation`**

This is the process simulation application's main class to configure, start and monitor the creation of process execution data and its persistence in an Activiti database. The class contains the `main` method which will be called by the user. The following major steps are processed within the main method:

1. **Reading configuration properties and command line arguments**  
Configurations set in `config.properties` as well as command line arguments are parsed, checked and prepared to be used by other methods and classes.
2. **Establishing a database connection**  
A connection to a PostgreSQL database is established according to configuration properties with prefix `stddb`.
3. **Setup of database**  
Depending on whether the corresponding command line argument has been set, a setup script is executed on the database to change constraints to `deferrable`. The executed commands are listed in files `alterConstraintsToDeferrable.sql`. This enables the command to reset the database and remove all data without facing any restrictions caused by constraints during a database transaction.
4. **Reset of database**  
Depending on whether the corresponding command line argument has been set existing data in the Activiti database is being removed from all tables. This is by use of SQL script defined in file `removeAllData.sql`.
5. **Setup of user-group hierarchy**  
If a user list has been defined in `config.properties` file, the application checks the existence of each user, group and membership in the Activiti database. Missing users, groups and/or memberships will be created.
6. **Start of process executions**  
At first, process execution starts of all iterations are timed. Therefore, random future timestamps are generated within a time frame from now until the maximum simulation time reduced by the maximum execution time of one process iteration (specified in the `config.properties` file).  
Then, the process definition is deployed to the Activiti process repository and an instance of Activiti's process engine is built. Activiti is configured, so that the

specified PostgreSQL database is used to capture process execution data.

Each process execution is started at the randomly defined start time. For each process execution, an instance of a sub-class of `SimulationDataGenerator` is generated. If a user list has been defined in `config.properties`, for each user an corresponding thread to handle user tasks is started. Otherwise, one catch-all-user-tasks thread is started.

After all iterations have been successfully completed, the Activiti process history might be written to the logfile on log level INFO.

### **com.pwcd.sim.UserTaskHandler**

For each user, one thread to handle user tasks is created by the main process. The `UserTaskHandler` processes open user tasks of all process instances utilizing the process engine's `taskService`. The thread terminates after the process execution of all iterations has ended.

To process the task, methods `generateSimulatedTaskData` and `generateSimulatedTaskTimeInMilliseconds` of the `SimulationDataGenerator` implementations are called. Task variables are set accordingly and, to simulate the task execution time, the user task handler thread sleeps for the calculated simulation task time before the task is reported to the engine as completed.

### **com.pwcd.sim.ServiceTaskHandler**

The `ServiceTaskHandler` is intended to be called by the Activiti engine in case of service tasks are to be processed. Therefore, the user needs to register this class in his/her BPMN model accordingly. The service task needs to have task type "Java class" and the `ServiceTaskHandler` needs to be set as class name.

Method `execute` of the implemented interface `JavaDelegate` processes the execution of one task similar as it is done in the `UserTaskHandler`. The respective `SimulationDataGenerator` instance is loaded via the Activiti's process id to generate data and determine the sleep time.

To also be able to handle asynchronous service tasks during process execution, `async-ExecutorActivate` of Activiti's `ProcessEngineConfiguration` is set to `true`.

### **com.pwcd.sim.SimulationDataGenerator**

Child-classes of class `SimulationDataGenerator` are to be implemented specifically for one BPMN model and contain the business logic for the generation of simulated data. These classes are thought to be provided by the user which defined the BPMN model.

To create an instantiable sub-class, two methods need to be implemented. Both receive the Activiti's activity task id as parameter to identify the kind of the requested data. Method `generateSimulatedTaskData` returns a map containing values for variables which need to be set in the respective task. Method `generate-Simulated-Task-Time-In-Milliseconds` returns the simulated execution duration to process the task.



The retrieved map containing simulated data will be added as task variables to the process instance by the calling function. And, to simulate execution times, the thread of the calling function will sleep for the simulated execution duration to process the task.

To implement a sub class of `SimulationDataGenerator`, the super class offers some supportive methods which might be called by sub-classes:

- **Getter for min/max process execution time** The super class offers methods to determine the user provided minimum and maximum execution time of a process iteration. This is crucial, as the actual simulated execution time of single tasks is determined in the user implemented `SimulationDataGenerator` subclass. The compliance with the user configured min/max process execution times depends on whether the implementation of `SimulationDataGenerator` takes them into account.
- **Getter for random CSV mockup data** Method to easily retrieve data from mock-up data CSV files in the application's resources.

The user's implementation of `SimulationDataGenerator` might keep track of already generated data within the process iteration to implement stateful simulation data creation. When creating data for a task, already created data for previous tasks can be accessed and interpreted. This might be of use, if the values creation in later tasks depends on values in previous tasks.

From an application architecture point of view it is recommended to put all user created sub-classes of `SimulationDataGenerator` into package `com.pwcd.dat`.

### 4.2.4 Documentation of resources

The following files are loaded from the simulation application's resource context:

#### Process definition file (BPMN 2.0 XML)

The business process which should be executed within the simulation application needs to be defined in form of a BPMN 2.0 model. A corresponding XML file is to be created. It is recommended to use Activiti's designer provided within the Activiti web server application or within Activiti's Eclipse plugin.

For all user tasks data is generated within the simulation application by calling the respective method in the subclass of `SimulationDataGenerator`. If service tasks are modeled and data should be generated / simulated within the simulation application, class `ServiceTaskHandler` needs to be registered as handler. Please mind that it was decided to not handle Activiti business rule tasks by the simulation application, as the integration of the business-rule-management-system Drools in Activiti is still experimental[AS17].

It is recommended to store BPMN models in folder `bpmn_models`. This is not mandatory, as the path to the BPMN model is configurable in file `config.properties`.

### **Mock-up data in CSV file**

Super class `SimulationDataGenerator` provides functionality to load random data from CSV file columns. To utilize this feature, a corresponding CSV file needs to be provided in the application context. The first row of the CSV file is required to be the header of the data set.

Several internet platforms, such as Mockaroo [Moc19], provide quite decent sets of mockup data in the requested format. Due to restrictions in the terms of use, no such file could be provided out-of-the-box as part of the standard installation of the simulation application.

It is recommended to store CSV mockup data in folder `mockup_data`. This is not mandatory, as the path to the mockup data is configurable in file `config.properties`.

### 4.3 Simulation app input: Business process definition (BPMN model)

The simulation application of project `persistenceOfWorkflowControlData` will be utilized to create a data set which is used to perform the benchmark of the temporal DBMS. To generate simulated process execution data with help of the simulation application, a process definition and an implementation of class `SimulationDataGenerator` are needed. Both types of resources are described in section 4.2.

In order to be able to conduct different queries during the benchmark, the selected BPMN process must be correspondingly manifold. Rademakers and Weske [Rad12, Wes12] use a loan /credit request model as examples in their books. As they are a good choice to use different BPMN elements and show various functionalities, this work will use their processes as basis to model another credit approval process. By doing so, a process example was chosen which has the highest practical relevance, as the optimization of lending processes is highly relevant in the financial services sector.

Please find details on the process definition, but also about the corresponding `SimulationDataGenerator` implementation in the following sections. The sources of both are contained in project package `persistenceOfWorkflowControlData` and can be (re-)used out of the box by setting the corresponding `config.properties`.

#### 4.3.1 Process idea and model definition

Let's assume, a financial services institution wants to improve their credit application and approval process to be fit for the future. Therefore, they standardized their processes and made them partly digital, dependent on the actual credit amount and the creditworthiness of the applicant. The process was designed as shown in figure 4.2.

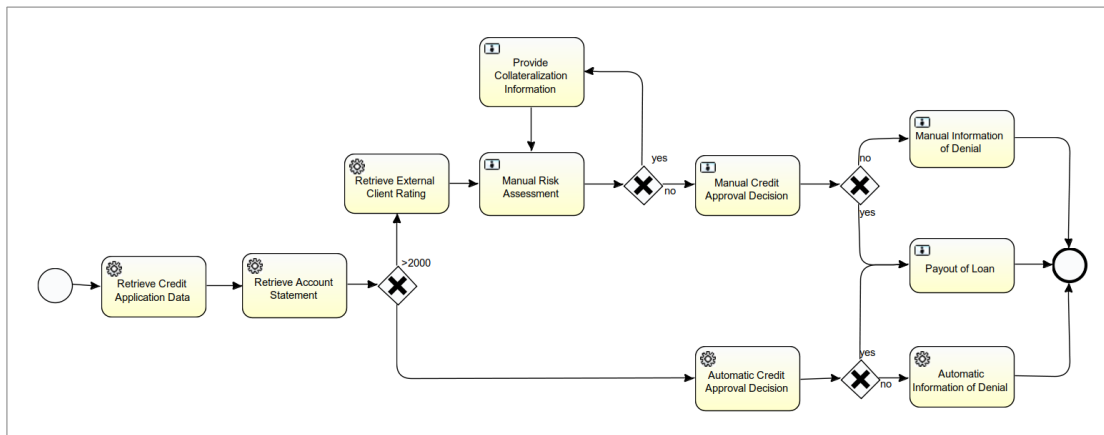


Figure 4.2: Defined sample business process in BPMN 2.0 notation (visualization in Activiti editor)

### 4.3.2 Process steps, input data fields and identity candidate groups

The following process steps have been defined:

#### **Retrieve Credit Application Data** (asynchronous service task)

A credit application might reach the financial institution via several channels, such as mobile app, website, phone hotline or branch office. The credit approval process starts with retrieving this information to start assessing the application. The following data fields are being loaded:

- `fullName` Name of the applicant
- `monthlyIncome` Monthly income (according to application)
- `requestedCreditAmount` Requested loan amount
- `requestedCreditEndDate` Requested end date of the loan

#### **Retrieve Account Statement** (synchronous service task)

At first, his/her bank account will be read and analyzed to check whether the income is equal to the specified one. The calculated income is stored in field `monthlyIncome-AccountStatement`.

#### **Retrieve External Client Rating** (asynchronous service task)

In case of applications for loans with a nominal amount higher than 5000, an external service will be utilized to check his/her creditworthiness. A rating is stored in field `externalClientRating`.

#### **Manual Risk Assessment** (user task; group loanSpecialist)

Based on the data in the application form and in the rating of the external service, an internal rating will be calculated and stored in field `internalManualClientRating`. If additional collateralization is needed, a flag is set by the user(`collateralCheckRequired`).

#### **Provide Collateralization Information** (user task; group collateralSpecialist)

In case of additional collateralization is needed, a specialist is checking for collaterals and captures their nominal value (`collateralAmountNominal`) and the actual collateral value, which is the reduced nominal amount to represent resale costs, risk factors e.g. in case of sureties (`collateralAmountWeighted`). The flag `collateralCheckPerformed` is set to indicate that collateralization information has been provided.

#### **Manual Credit Approval Decision** (user task; group management)

Based on the internal rating a credit approval decision is taken by the head of department for larger loans `manualApprovalDecision`.

**Manual Information of Denial** (user task; group loanSpecialist)

If a loan cannot be granted, an officer will get in contact with the applicant. This is done in personal contact to lower the risk losing the client. A flag indicates, if this step has been performed (`manualInformationDone`).

**Payout of Loan** (user task; group loanSpecialist)

In any case, the payout of the loan will be done manually. A flag indicates, if this step has been performed `payoutDone`. Additionally the `payoutAmount` is captured;

**Automatic Credit Approval Decision** (asynchronous service task)

For smaller loans, an automatic credit decision is taken. Therefore, the share of the loan amount in the monthly salary is calculated for this and stored in field `internalAutomaticClientRating`. Based on this, an automatic credit decision is taken and stored in (`automaticApprovalDecision`). The credit will be approved if the amount of the credit does not exceed twice the monthly income.

**Automatic Information of Denial** (asynchronous service task)

If a loan cannot be granted, an automatic message will be sent to the applicant for smaller loans. The mail informs the client and offers to call a service line in case of any questions. A flag indicates, if this step has been performed (`automaticInformationDone`).

#### 4.3.3 Configuration of BPMN process related `config.properties`

The settings set in `config.properties` can be found in table 4.5. The BPMN file and the corresponding `SimulationDataGenerator` class are specified. The user and their group memberships as well as the minimum and maximum process execution time can be adapted according to your own needs. However, the process definition would require at least one user in groups management, loanSpecialist and collateralSpecialist.

As always, it is recommended to keep the min/max process execution times in relation to the overall simulation time frame and the count of process iterations, which are to be passed as command line arguments.

Description
1 Path to the BPMN 2.0 process model XML
2 Process definition key as set in the BPMN XML
3 <code>SimulationDataGenerator</code> implementation to be called for data creation
4 Maximum time of a process execution in seconds
5 Minimum time of a process execution in seconds
6 Activiti users and their group memberships: Cecilia Chef and Bert Boss are in group management, Kunibert Knowledge and Evelyn Expert are in group loanSpecialist and Susanne Stuff is in group collateralSpecialist.
Properties
1 <code>process.file=bpmn_models/creditApproval.bpmn20.xml</code>
2 <code>process.key=creditApproval</code>
3 <code>process.simulationDataGeneratorClass=com.pwcd.dat.CreditApprovalSimulationData</code>
4 <code>process.maxProcessExecutiontimeInSec=20</code>
5 <code>process.minProcessExecutiontimeInSec=10</code>
6 <code>process.userGroupMemberships = Cecilia Chef,management,Bert Boss,management, Kunibert Knowledge,loanSpecialist,Evelyn Expert,loanSpecialist, Susanne Stuff,collateralSpecialist</code>

Table 4.5: `Config.properties` for credit approval process

#### 4.3.4 Implementation of `SimulationDataGenerator`

To create simulated process execution data for the credit approval process, an implementation of class `SimulationDataGenerator` was developed. The class offers functionality to simulate user and service task execution times and to create simulated input data for these tasks.

##### Calculation of task execution times

Within the simulation application, a minimum and maximum process execution time can be configured in the `config.properties`. The values are then available in class `SimulationDataGenerator`. To make process simulation adaptive to the indicated process execution times, no fixed task times or task time ranges are set. Random task execution times between a minimum and a maximum task execution time are calculated based on user input.

To not have (on average) the same execution times per task, time shares of tasks on the overall process execution times are distributed irregularly. E.g. certain user tasks will take longer than some service tasks. Therefore single task's average time shares of the overall process execution under consideration of the actual possible process path is defined. The actual execution time is then calculated according to the logic as shown in algorithm 4.1.

In a perfect world, the task share of all executed tasks within a process execution would add up to 100%. However, this cannot be assured, as the complete journey of tasks cannot be predicted beforehand (e.g. loops or skipping tasks according to decision criteria in BPMN model).

---

**Listing 4.1** Calculation of task time

---

```
1 taskExecutionTime =  
2   ThreadLocalRandom.current().  
3   nextInt(minProcessExecutionTime * taskShare, minProcessExecutionTime * taskShare);
```

---

#### Generation of task data

Input data for task execution will be created by generating random values. As some input data is semantically dependent on others, such as a credit approval decision on the corresponding risk rating, the generation of input data will be done under consideration of values generated in previous steps. Therefore, the class will also keep track of already generated values.

## 4.4 Benchmark application: Querying process execution data

The benchmark application compares the execution of SQL queries in two different databases. Therefore, the performance of the query execution is measured and compared. Additionally, measures for the complexity of the query statements are determined, as the query statements for the two database technologies may differ despite semantic equivalence.

The initial idea of this application was to compare the performance of a temporal database to the performance of a standard installation of PostgreSQL. Therefore the two databases will be called `stddb` and `tempdb` in the following chapter, even though any PostgreSQL based database technology might be compared with help of the implemented benchmark application.

The application takes a CSV file with queries as input and executes them in two databases. Each statement is executed several times to avoid any incorrect measurement. The query planning and execution times of each query execution are captured to then calculate median and average values per query and database. All data manipulations are rolled back after the query has been executed. Results might be exported as CVS file(s).

### 4.4.1 Installation, prerequisites and configuration

The application is part of a Java Maven multi module project called `persistenceOf-WorkflowControlData` and is dependent on another module called `pwcdShared` which contains configuration files, script files and generic helper classes. Basically, the maven project is configured in a way that commands like `mvn clean verify` and `mvn compile package` work as to be expected.

To successfully compile the project, dependent libraries need to be available in java class path. The libraries have been listed in chapter 4.1.1 (see also the corresponding `pom.xml` of the root project and the simulation module). To successfully run the application, Activiti databases in version 6 are required to be running both databases.

In context of this work, the following two database technologies are being used:

- Standard PostgreSQL 9.6.9 installation
- Temporal PostgreSQL installation in version 9.6beta3 by university of Bozen-Bolzano. Information on how to install a Temporal PostgreSQL can be found in the installation package [Dig18].

As basis for the benchmark, a CSV file containing benchmark queries needs to be available as resource in the application context. The file contains two queries per row, one for each database technology. Both queries are thought to be semantically equal and subsequently called query set.



Before running the application, the configuration files of the project need to be adapted. Among other things, the database credentials must be specified and references to the user provided file containing benchmark queries must be set accordingly. Additionally, the count of executions per query can be specified. The application does also offer the option to compare the result sets of two queries within a query set to warn the user or stop the benchmark if they are not equal.

All available configuration parameters which are specifically to be set for the benchmark module can be found in table 4.6. Some more details on how single parameters change the behaviour of the application can be found in the subsequent sections describing the capabilities and functionality of the benchmark application in detail 4.4.3 and 4.4.4.

Property name	Description
db1.*	Configuration of a database connection to an Activiti database, e.g. in a non temporal PostgreSQL DBMS
db2.*	Configuration of a database connection to an Activiti database, e.g. in a temporal PostgreSQL DBMS
benchmark.csvfile	Path to a CSV file containing benchmark queries. The file needs to be available in the application's context
benchmark.csvSeparator	Separator used in the CSV file
benchmark.queryRepetition	Count of executions per query and database. Executing a query only once leads to doubtful-leading performance measures. Therefore, queries are executed more often and average/median performance measures are calculated.
benchmark.outputfolder	Path of output folder for result files within the application's context
benchmark.checkSameResultset	Indicates if the application should check if two queries of a query set return the same result set before starting the benchmark; Options: IGNORE, WARN, STOP

Table 4.6: Mandatory `config.properties` for benchmark application

#### 4.4.2 Running the app and command line arguments

To run the application, the `main` method of class `DbBenchmark` needs to be called. To easily start the application, the generated jar is configured to be executable, having the `DbBenchmark` class set as manifest's `mainClass`.

While the settings in file `config.properties` are static, the command line arguments can be used to set additional execution-specific parameters. All available command line arguments are documented in table 4.7.

In table 4.8 two example command line calls of the benchmark application are listed. In row one, an application call can be found, which is recommended after new Activiti databases have been setup to initially adapt them to the benchmark application's needs. In row two, a usual application call to copy data from one to another database and then benchmark the query processing is shown.

Argument	Description
<b>e</b>	Adjustments to the Activiti databases after initial installation. The application alters constraints to <b>deferrable</b> to enable database resets and adds indexes to columns with type <b>timestamp</b> in tables <b>ACT_HI_PROCINST</b> , <b>ACT_HI_ACTINST</b> and <b>ACT_HI_TASKINST</b> .
<b>c</b>	Copies all data from default database to the temporal database. Therefore, all existing data in temporal database will be removed.
<b>v</b>	Performs PostgreSQL vacuum command, a garbage-collect feature, in both databases.
<b>b</b>	Running the benchmark determining query complexity and measuring query execution performance in both databases for user specified SQL queries.
<b>f</b>	Writes aggregated results to CSV file; Option will be ignored if no benchmark has been performed.
<b>d</b>	Writes detailed results to CSV file; Option will be ignored if no benchmark has been performed.

Table 4.7: Benchmark application: Command line arguments

	Description
1	Initial adaptations to Activiti databases
2	Example for a usual application call to benchmark query processing incl. copying data from the default to the temporal database
3	Example for a usual application call to benchmark query processing without copying data beforehand
	Command
1	<code>java -jar pwcdBenchmark/target/pwcdBenchmark-&lt;version&gt;-jar-with-dependencies.jar -e</code>
2	<code>java -jar pwcdBenchmark/target/pwcdBenchmark-&lt;version&gt;-jar-with-dependencies.jar -cvbfd</code>
3	<code>java -jar pwcdBenchmark/target/pwcdBenchmark-&lt;version&gt;-jar-with-dependencies.jar -bfd</code>

Table 4.8: Benchmark application: Examples for command line calls

#### 4.4.3 Documentation of classes and program flow

This section contains details on the implementation of the benchmark application. The classes specifically implemented for the simulation application are grouped in Java package `com.pwcd.benchmark`.

**com.pwcd.benchmark.DbBenchmark**

This is the benchmark application's main class to copy data from a conventional to a temporal Activiti database and to start and monitor database query benchmarks. The class contains the `main` method which is the entry point to the application and will be called by the user. The following major steps are processed within the `main` method:

**1. Reading configuration properties and command line arguments**

Configurations set in `config.properties` as well as command line arguments are parsed, checked and prepared to be used by other methods and classes.

**2. Establishing database connections**

A connection to a standard installation of a PostgreSQL database is established according to configuration properties with prefix `stddb`. Another connection to a temporal installation of a PostgreSQL database is established according to configuration properties with prefix `tempdb`.

**3. Optional: Setup of databases**

Depending on whether the respective command line arguments have been set, two setup scripts are executed on the databases to change constraints to `deferrable` and to add indexes to columns with type `timestamp` in tables `ACT_HI_PROCINST`, `ACT_HI_ACTINST` and `ACT_HI_TASKINST`. The executed commands are listed in files `alterConstraintsToDeferrable.sql` and `createIndexes.sql`.

**4. Optional: Copying data**

To perform the query execution performance measurement on the same basis, the benchmark application offers a feature to remove all data from the Activiti database in the temporal PostgreSQL installation (usage of script in `removeAllData.sql`) to then copy all data from the Activiti database in the standard PostgreSQL installation.

This should be done after every change to the data stored in the standard Activiti database, e.g. after a new simulation run has been performed.

**5. Optional: Performing vacuum**

Before the benchmark, PostgreSQL's garbage-collect functionality `VACUUM` might be called in both databases.

**6. Performing benchmark**

The query CSV file contains two SQL queries per row, one for each database. The SQL queries of a row are semantically identical and should return the same result set. These pairs of queries are called query sets. If the corresponding `config.properties` entry is set, the application checks whether the two result sets are actually the same and skips processing the row if not.

Each query statement is executed as often as indicated in the command line arguments. PostgreSQL's function `EXPLAIN ANALZE` is used to measure the planning

and execution time. The measures of all executions are stored and per query a median and an average planning and execution times are calculated. Additionally, complexity measures according to Halstead are calculated for each query.

#### 7. Optional: Export of result data files

Beside of a verbose output of the benchmark results to console or a log file, performance indicators and the calculated query complexity measure might also be exported to CSV files. A detailed file contains information on aggregated level, a detailed CSV file contains data about every single query execution.

#### Java classes encapsulating data

The application uses three types of Plain Old Java Objects (POJOs) to process the benchmark and store result data.

**com.pwcd.benchmark.DbBenchmarkQuerySet** It represents one row in the input file providing queries for the benchmark. The encapsulated data follows the CSV data structure. Therefore, each query set has an ID, a comment explaining the intended meaning of the queries and a pair of SQL queries, one for each database technology. The single queries are represented as object of **DbBenchmarkQuery**.

Additionally, the class offers functions to generate an array of strings, containing aggregated performance measures for the set of queries, to be used as basis for a CSV export.

**com.pwcd.benchmark.DbBenchmarkQuery** It represents a single query and encapsulates an SQL statement as String. Additionally, it offers to set a query type. In the benchmark application, the type is populated with column name in the corresponding CSV file and is used to differentiate between **stddb** and **tempdb** queries.

Additionally, a list of execution performance measures can be stored, represented as list of **DbBenchmarkQueryExecutionResult** instances. The class does also offer functions to calculate average and median values of all execution performance measures of this query. These aggregated measures are also represented in form of an object of **DbBenchmarkQueryExecutionResult**.

The class does also offer functionality to calculate complexity measures for the query, based on Halstead's definition [Hal77].

**com.pwcd.benchmark.DbBenchmarkQueryExecutionResult** It stores query execution measures and therefore encapsulates results of PostgreSQL's command **EXPLAIN ANALYZE**. The object encapsulates planning time and execution time as numerical values (row 2 and 3) as well as the complete query plan as text (row 1).

#### 4.4.4 Documentation of resources

The following files are loaded from the benchmark application's resource context:

##### Benchmark SQL queries (CSV file)

CSV file containing benchmark queries and at least the following headers described in table 4.9. The separator can be freely chosen, but needs to be configured in `config.properties`. The queries of the query set are intended to be semantically equivalent and should therefore return the same result when executed on the same data. Their syntax may differ due to different query languages of the utilized DBMS.

Column	Description
query_set_id	Unique ID for the set of queries. The uniqueness is not checked and also not mandatory for benchmark application but facilitates any subsequent use of results.
query_stddeb	Query to be executed on the database configured as stddeb.
query_tempdeb	Query to be executed on the database configured as tempdeb.
comment	Comment explaining the meaning or intention of the query set.

Table 4.9: Columns of SQL queries CSV file

#### SQL operators

Files `postgresql_operator.csv` and `postgresql_operator_pure.csv` contain operators of the DBMS used in the benchmark. They are needed as basis to calculate Halstead's complexity measures. Both files contain operators sorted by operator length in descent order. The second file contains operators which can also be used as separators between two operands (or operators).

#### 4.4.5 Documentation of result files

The benchmark application exports benchmark results to CSV files. All files are written to a folder defined in the `config.properties` (default folder name is `results`).

##### Results aggregated (CSV file)

The file with aggregated result data is similar in structure to the SQL queries input file. Actually, all columns of the input file are also exported with slightly changed order, but enriched with in table 4.10 listed additional columns containing benchmark result data.

Column	Description
execution_average	Average execution time of query
execution_median	Median execution time of query
planning_average	Average planning time of query
planning_median	Median planning time of query
iterations_count	Count of execution results
halstead_length	Halstead complexity measure: program length
halstead_vocabulary	Halstead complexity measure: program vocabulary
halstead_volume	Halstead complexity measure: program volume
halstead_difficulty	Halstead complexity measure: program difficulty
halstead_effort	Halstead complexity measure: program effort

Table 4.10: Columns per database, exported to `results_aggregated` CSV file

### Results detailed (CSV file)

In addition to the aggregated results file, the results can also be exported in a higher granularity. Whereby the detailed results file is structured differently. Each row represents one execution of one query in one database. For example, if a benchmark with 100 iterations is performed, 200 lines per line in the aggregated results file would be exported to the detailed results file (100 query executions per query and database).

For details on the exported columns, see the table 4.11.

Column	Description
query_set_id	Unique ID for the query set, according to the benchmark query input file
comment	Comment on the query set, according to the benchmark query input file
type	Type of query, indicating the database in which it was executed
iteration_nr	Counter for query executions per query and database
execution	Total execution time of query
planning	Planning time of query
query_plan	Textual query_plan output from PostgreSQL's <code>EXPLAIN ANALSE</code> command

Table 4.11: Columns in `results_detailed` CSV file

## 4.5 Benchmark app input: BPM relevant database queries

To compare the time-related query capabilities and the query execution performance of a non-temporal and a temporal PostgreSQL installation, not only workflow control data needs to be available, but also sample queries that can be used to perform the benchmark.

At first we will show which kind of queries on process execution data are of special interest for the benchmark. In a second step, concrete queries will be defined for the in chapter 4.3 defined credit approval process. The last section of this chapter discusses the design of period data queries in the non-temporal database.

### 4.5.1 Benchmark relevant query types

As described in chapter 2.4.2, the temporal PostgreSQL database by the Free University of Bozen-Bolzano offers extended functionality to handle queries for period data. This functionality should be evaluated in comparison to a standard installation. When defining the queries it should be ensured that all additional features of the temporal database are utilized. All queries on the temporal database should utilize the preliminary SQL extension which enables the user to simpler query period data in SQL. “It supports temporal aggregation, distinct, inner and outer joins and set operations, by using manual query rewriting to the two new operators” [Mos16]. The implemented temporal primitives, namely `NORMALIZE` and `ALIGN`, will not be used in the benchmark, as they would add unnecessary complexity to the temporal query statement.

At the current state of implementation, some limitations are known which need to be considered when defining our benchmark queries. Currently, the temporal DBMS can only process periods which are already closed, as “unbound ranges and `NULL` values are not supported yet” [Mos16]. Therefore it is not possible to utilize the temporal SQL features on data of ongoing process execution, e.g. to monitor process execution. Furthermore the temporal DBMS requires input relations of set operations (`UNION`, `EXCEPT`, `INTERSECT`) to be disjoint [Mos16]. Therefore the processing of `DISTINCT PERIOD` on non-disjoint input relations is required before calling set operations, even if the pure set theory driven perspective would not infer that.

### 4.5.2 Benchmark relevant BPM data requirements

In order to perform the benchmark in the context of Business Process Management, queries should be relevant to the respective data requirements. In chapters 2.1.2 and 2.1.3 time related process execution data being relevant to Business Process Management and its availability in WfMSs has been discussed. To reveal performance differences of query processing, we will focus on queries where we expect an improved query processing in the temporal database, which offers enhanced performance in querying for data “over time”

or extracting a status within “at a certain period of time”. So, not time but specifically period related data requirements are of interest for the benchmark.

To retrieve the duration of a process, activity or task execution from the Activiti database, no feature of the temporal DBMS needs to be utilized. So, even though this kind of data requirements have been identified as relevant to BPM, the respective queries are not temporal in sense of reflecting developments “over time”. Queries for events and durations are therefore not of interest for our benchmark as no differences are to be expected in the performance of a non-temporal and a temporal database.

Whenever values within a certain period or developments over time are part of the data requirement, the respective queries for process execution data are expected to benefit from temporal database features. This is e.g. the case for periods discussed in the following paragraphs.

**Execution periods** In Activiti, data about execution periods is available on process, activity and task level. To determine e.g. the count of active processes or activity executions over time, period data needs to be aggregated to process definition or activity definition level. This aggregation over periods is supported by the temporal database.

**Utilization periods** To determine e.g. the count of tasks allocated to or processed by a resource or service, the perspective needs to change from execution to resource utilization. Period data has to be rearranged accordingly, which is supported by the temporal database.

**Idle times of resources or processes** To identify idle times of resources, services or periods in which no process execution is active, periods need to be identified for which no execution data is available. The temporal database supports these kind of queries.

**Validity periods of data** Queries considering the validity of data would be of relevance. However, Activiti does not keep track of the validity periods of historic values of variables or ID information.

### 4.5.3 Dealing with period data in non-temporal SQL

The temporal database not only promises faster processing of temporal queries. It also provides an extension to standard PostgreSQL to better formulate temporal queries. While defining the temporal queries for the benchmark it became obvious that “natural queries are very difficult to formulate in SQL, but are easy to formulate in these temporal query languages” [DBGJ16].

Particularly it is challenging to identify the respective period start and end times in order to adjust periods. It is necessary to identify a set of shortest possible periods, covering the entire course of time being relevant to a certain fact. E.g. if the count of active process executions should be determined, a new period needs to be defined



whenever a process starts or ends. Therefore, the set of all start and end times of the complete dataset of interest defines the period borders. Listing 4.2 provides an example for identifying all times of either starting or ending process executions. The statement generates a dataset with one column, which contains the start time of the current period which is equal to the end time of the previous period.

---

**Listing 4.2** Identification of maximum set of period borders

---

```
1 SELECT start_time_ as border FROM ACT_HI_PROCINST
2     UNION SELECT end_time_ as border FROM ACT_HI_PROCINST
3 ORDER BY border;
```

---

To have both, start and end time of a period, available in one dataset row, a second column needs to be joined as it is shown in listing 4.3. This complicates the query significantly. At first, all period borders need to be identified (line 3+4), similar to as it is shown in listing 4.2. The first column can be seen as start times of periods. To add end times, the same data (line 7+8) must be added, but offset by one row (line 9). In PostgreSQL this can be achieved via joining the row number to the two datasets representing start and end times (line 2+6).

---

**Listing 4.3** Identification of periods

---

```
1 SELECT period_start.border AS p_start, period_end.border AS p_end FROM
2     ((SELECT ROW_NUMBER() over (order by border) r, border FROM
3     (SELECT start_time_ as border from ACT_HI_PROCINST
4     UNION SELECT end_time_ as border FROM ACT_HI_PROCINST) AS period_borders) period_start
5 JOIN
6     ((SELECT ROW_NUMBER() over (order by border) r, border FROM
7     (SELECT start_time_ as border from ACT_HI_PROCINST
8     UNION SELECT end_time_ as border FROM ACT_HI_PROCINST) AS period_borders) period_end
9 ON period_end.r=(period_start.r+1)
10 ORDER BY period_start.border;
```

---

In some cases, not all periods are relevant to all analysis / query dimensions. E.g. if the count of tasks being assigned to a user over time needs to be determined, period borders of tasks being not assigned to him/her are not of relevance, as they will not change the count of assigned tasks for this user. To avoid too large intermediate result sets, this must be taken into account. Only relevant periods should be identified, e.g. querying periods per user in the case of the given example.

#### 4.5.4 Queries for defined BPMN process

Based on the identified types of benchmark relevant period related data requirements, concrete queries for process execution data have been implemented for the in chapter 4.3 defined credit approval process. Queries are defined to be executed in an Activiti database.

Each query is written in two forms: Once in standard SQL for a non temporal PostgreSQL database installation, once in form of a query involving symbols of the preliminary SQL extension utilizing features of the temporal PostgreSQL installation. Basically, the temporal and standard SQL query should return the same data in the same structure to make their query execution comparable.

For the benchmark, 13 queries have been selected so that all features of the temporal database are utilized at least once. Furthermore, we aimed to query all period data available in Activiti at least once and to extract data on all three available levels of detail: process, activity and task. For queries at task level, we also differentiate between the view by users and groups.

Variations of the same query (e.g. by applying additional filter criterias) were not considered. Additionally, if two queries are identified to be almost equal in their extraction logic and queried data fields, only one is used for the benchmark.

Table 4.12 lists all queries, their level of data granularity and the utilized temporal feature in the temporal SQL statement. Queries with ID-prefix Q can be seen as *basic type* queries, covering assignment, active and non-active periods per granularity level. Queries with ID-prefix S are additional queries which are specifically added to the benchmark set to utilize not yet covered temporal features of the temporal DBMS. References to other queries in column EQ mean that those queries have been identified as almost equal. These queries have been excluded from the benchmark, as two queries of the same kind would not add value to the benchmark.

The following paragraphs describe intention, business meaning and expected results of the queries. In addition, the idea and structure of the corresponding SQL queries are described and specifics are highlighted. The actual source code of all queries can be found in appendix A as well as in the digital appendix in format of a *Benchmark SQL queries CSV file* which can be used as input for the benchmark application.

Level	Query	ID	Temp. feature	EQ
<b>Process</b> (PROCINST)	Count of open processes over time (o.t.)	Q01	GROUP BY PERIOD	
	Count of active processes o.t.	Q02	GROUP BY PERIOD	
	Idle periods of processes	Q03	EXCEPT PERIOD	
	Periods with no open processes	Q04	EXCEPT PERIOD	
<b>User</b> (TASKINST)	Count of assigned tasks o.t.	Q05	GROUP BY PERIOD	
	Count of claimed tasks o.t.	Q06	GROUP BY PERIOD	
	Count of not-yet claimed tasks o.t.	Q07	GROUP BY PERIOD	
	Periods with no assigned tasks	Q08	EXCEPT PERIOD	
<b>Department</b> (TASKINST)	Distinct periods with claimed tasks	S01	DISTINCT PERIOD	Q06 Q07 Q08
	Parallel processing of specific user tasks	S02	INTERSECT PERIOD	
	Count of assigned tasks o.t.	Q09	GROUP BY PERIOD	
	Count of claimed tasks o.t.	Q10	GROUP BY PERIOD	
<b>Service</b> (ACTINST)	Count of not-yet claimed tasks o.t.	Q11	GROUP BY PERIOD	Q08
	Periods with no assigned tasks	Q12	EXCEPT PERIOD	
	Count of assigned tasks o.t.	Q13	GROUP BY PERIOD	
	Periods with no assigned tasks	Q14	EXCEPT PERIOD	
	Parallel processing of activities o.t.	S03	PERIOD JOIN	

Table 4.12: Overview: Benchmark queries and their targeted level of granularity in workflow execution data as well as their utilized temporal SQL feature.

### Q01 Count of open processes over time

At any given time, how many processes have been started but not yet ended? This helps to identify seasonal fluctuations in count of credit applications. Discovering a higher demand at certain times within a day, week or year helps to ensure the availability of resources.

The data should show time periods and the respective count of active processes. Whenever the total count of active processes has changed (e.g. an additional process has been started or a process has been completed) a new row should be created in the result set. Period borders are start and end times of processes (stored in table `ACT_HI_PROCINST`). All processes whose start time is equal or earlier and whose end time is equal or later than a period's start and end time are counted.

Query in standard SQL for the conventional PostgreSQL database: appendix listing A.1

Query in temporal SQL for the temporal PostgreSQL database: appendix listing A.3

Appendix listing A.2 illustrates how the standard SQL query would look like if it was sufficient to have period start times available only (with the respective period end time shown in the next result set row as start time of the subsequent period). This option of a simplified SQL statement has been discussed in chapter 4.5.3. However, it will not be used during the benchmark to base the comparison on same conditions for both database types.

### Q02 Count of active processes over time

At any given time, how many processes exist on which users are currently working or services operating on? In comparison to Q01, this shows processes which utilize internal or external resources.

The data should show distinct count of processes over time, which have either active tasks claimed by a user (period from `claim_time_` to `end_time_` in table `ACT_HI_TASKINST`) or ongoing activities (period from `start_time_` to `end_time_` in table `ACT_HI_ACTINST`, excluding activities with a duration of 0). Activities which are not tasks can be identified as having a `task_id_` of `NULL`.

Query in standard SQL for the conventional PostgreSQL database: appendix listing A.4  
Query in temporal SQL for the temporal PostgreSQL database: appendix listing A.5

The for the benchmark chosen standard SQL statement avoids a costly `GROUP BY` for the `DISTINCT` process count via starting the `SELECT` statement on process level. A similar approach can not be used for the temporal statement, as a `GROUP BY` cannot be avoided as it is needed to determine the set of periods. To further improve the execution performance of the non temporal statement, a CTE has been used for the periods.

Appendix listing A.6 shows a standard SQL statement following the same approach as the temporal statement. As the execution performance was as bad as the execution performance of the temporal statement, a different approach was chosen.

### Q03 Idle periods of processes over time

At any given time, processes are not handled by a user or a service. In these periods, no progress is to be expected in the processing of the credit application. These periods are caused by waiting times from the assignment of a task to an employee until the actual start of working on that task. They are also caused by the time which the WfMS needs to assign tasks or to call external services.

Both queries use the total process execution periods as basis, from which active user task periods and active service periods are deducted. In the temporal statement, temporal SQL symbol `EXCEPT PERIOD` simplifies the query a lot. The standard SQL statement determines at first the set of shortest periods to then join active process times. If no active process time could be joined (`proc_inst_id_` is `NULL` in the interim result data set), the process is idle.

Query in standard SQL for the conventional PostgreSQL database: appendix listing A.7  
Query in temporal SQL for the temporal PostgreSQL database: appendix listing A.8

### Q04 Periods with no open processes

In which periods does not a single credit approval process run? This helps to identify seasonal fluctuations in count of credit applications. Discovering periods with no demand

at certain times within a day, week or year helps to reallocate the availability of resources to periods with a higher demand.

The approach querying for the desired data is similar to Q03. Whereas the temporal query can utilize `EXCEPT PERIOD`, the standard SQL statement starts from the set of shortest periods to check if there is at least one active process running at each period. If this is not the case (`ACT_HI_PROCINST.id_` is `NULL` in the interim result data set), the period is identified as having no active process running.

Query in standard SQL for the conventional PostgreSQL database: appendix listing A.9

Query in temporal SQL for the temporal PostgreSQL database: appendix listing A.10

#### **Q05 Count of assigned tasks to user over time**

At any given time, how many tasks are assigned to an employee? This is of interest, as data about the development of task assignments over time enables the recognition of high and low utilization periods of a resource.

The query identifies a set of shortest periods of task assignments per user. Period borders are start and end times of task assignments (stored in table `ACT_HI_TASKINST`). For each period, all task assignments per user (with start and end times within the period) are counted.

Query in standard SQL for the conventional PostgreSQL database: appendix listing A.11

Query in temporal SQL for the temporal PostgreSQL database: appendix listing A.12

#### **Q6 Count of claimed user tasks over time**

At any given time, on how many tasks is an employee working on. The claim of a user task indicates that an employee is working on that task, or at least started to do so. To get knowledge about the actual utilization, not the number of assigned tasks, but the number of already accepted (claimed) tasks is of relevance.

The query identifies a set of shortest periods of task claims per user. Period borders are claim and end times of task assignments (stored in table `ACT_HI_TASKINST`). For each period, all claimed tasks per user (with claim and end times within the period) are counted. In case of users are not working in parallel on several tasks, the count of accepted tasks can be seen as indicator if or if not he/she is working on a task.

Query in standard SQL for the conventional PostgreSQL database: appendix listing A.13

Query in temporal SQL for the temporal PostgreSQL database: appendix listing A.14

#### **Q7 Count of not-yet claimed tasks per user**

At any given time, how many tasks are assigned to an employee which he/she did not yet claim and is therefore not working on (e.g. because he/she is working on other tasks, making a break or is on vacation). Such periods should be avoided, as no progress is to be expected in the processing of the credit application.

Period borders are start times of task assignments and claim times of tasks (stored in table `ACT_HI_TASKINST`). For each period, all not yet claimed tasks of employees (task with start and claim times within the period) are counted.

Query in standard SQL for the conventional PostgreSQL database: appendix listing A.15

Query in temporal SQL for the temporal PostgreSQL database: appendix listing A.16

### **Q08 Periods with no assigned tasks to user**

In which period of times does an employee not have any tasks assigned? This helps to identify gaps in the utilization of resources and thus to recognize savings potential.

The query identifies a set of shortest periods of task assignments per employee. Period borders are start and end times of task assignments (stored in table `ACT_HI_TASKINST`). For each period, all task assignments per employee (with start and end times within the period) are counted.

In standard SQL, the count result needs to again be joined to the periods to determine rows with no assignments. Additionally, periods from the very start of process executions to the first task assignment and from the last task assignment to the end of the last process also taken into account as non-utilization period.

The temporal SQL statement can utilize the functionality of symbol `EXCEPT` to simply deduct task assignment times from the total process execution time distance.

Query in standard SQL for the conventional PostgreSQL database: appendix listing A.17

Query in temporal SQL for the temporal PostgreSQL database: appendix listing A.18

### **Q09 Count of assigned tasks to department**

At any given time, how many tasks are assigned to a department? This might help to recognize savings potential or identify the need to enlarge the workforce of a department.

The query is similar to Q05, but groups periods not by employee but by department. Therefore, a join to tables `ACT_ID_USER` and `ACT_ID_MEMBERSHIP` which store the organizational structure is necessary before grouping the periods and counting task assignments.

Query in standard SQL for the conventional PostgreSQL database: appendix listing A.19

Query in temporal SQL for the temporal PostgreSQL database: appendix listing A.20

### **Q10 / Q11 / Q12 Further queries for periods on department level**

Queries Q10, Q11 and Q12 are to query period data on department level and have been identified as being almost equal to Q06, Q07 and Q08 querying for periods on user level. The only difference is the level of grouping. Q09 shows how this can be achieved. Since these queries are not to be expected to add any additional value to the benchmark, they have not been implemented.

**Q13 Count of assigned tasks to service over time**

At any given time, how many activities are operated by an internal or external service? Therefore data about the development of service utilization over time enables the recognition of high and low utilization periods. This is e.g. of interest if the count of parallel service utilization should be avoided to e.g. reduce the count of licences to use an external service, such as to externally check the creditworthiness of an applicant.

The query identifies a set of shortest periods of activity assignments to services. Period borders are start and end times of activities of type `serviceTask` stored in table `ACT_HI_ACTINST`). For each period, all service calls per activity name (with start and end times within the period) are counted.

Query in standard SQL for the conventional PostgreSQL database: appendix listing A.21

Query in temporal SQL for the temporal PostgreSQL database: appendix listing A.22

**Q14 Period count per service**

At any given time, how many user tasks are executed within the same department in parallel? This might help to recognize savings potential (if such periods are rather rare).

The query was identified as almost equal to Q06, except the joins to the department, which is shown in Q09.

**S01 Distinct periods with claimed tasks per user**

In which periods does a user work on at least one task? Similar to Q06, the data provide information on employee utilization.

This additional query was added to the benchmark set to show the usage of temporal operator `DISTINCT PERIOD`. The standard SQL follows the same design as the non-temporal SQL statement of Q06, but has no count of assigned tasks.

Query in standard SQL for the conventional PostgreSQL database: appendix listing A.23

Query in temporal SQL for the temporal PostgreSQL database: appendix listing A.24

**S02 Parallel processing of specific user tasks**

How often is a particular task processed in parallel with another specific task? This helps e.g. to identify if there is a possibility to pool resources (human resources, rooms, etc.). In the benchmark we query how often 'manualRiskAssessment' is executed in parallel to 'manualCreditApprovalDecision'. Currently, the two consecutive tasks are handled by two different departments. It is assumed that both could be handled by the same department in order to reduce the number of handovers. However, the number and duration of the parallel execution periods of these tasks must be as low as possible in order to avoid additional idle times in the second process resulting from the reorganization.

This additional query was added to the benchmark set to show the usage of temporal operator `INTERSECT PERIOD`. The standard SQL determines the set of shortest periods for task executions of tasks with a `task_def_key_` of 'manualRiskAssessment' or 'manual-CreditApprovalDecision' to then join twice to the task table (once for each task) to check whether there are executions of both tasks in parallel.

Query in standard SQL for the conventional PostgreSQL database: appendix listing A.25

Query in temporal SQL for the temporal PostgreSQL database: appendix listing A.26

### **S03 Parallel processing of activities**

In which periods of time are services operated in parallel within a process execution? If there are no dependencies, service calls can be executed in parallel which saves time and speeds up the process execution. The query shows how often this possibility was seized.

This additional query was added to the benchmark set to show the usage of temporal operator `PERIOD JOIN`. Additionally it shows that not always a period-centric query approach is of advantage. The standard SQL statement does not determine the number of periods to reduce it in a second step, but uses a classic join with corresponding criteria.

Query in standard SQL for the conventional PostgreSQL database: appendix listing A.27

Query in temporal SQL for the temporal PostgreSQL database: appendix listing A.28



# Benchmark: Evaluation of temporal database handling queries on workflow control data

The following chapter describes how the actual benchmark was performed to compare the capabilities and performance of executing periodic queries in a temporal and a non-temporal DBMS. First, the overall benchmark setup and the data sets generated by the simulation application are described. These data sets were queried using the benchmark application. In the second part of this chapter benchmark results are presented and commented.

## 5.1 Benchmark configuration and technical setup

For the benchmark, we use a ThinkPad T410s in the following configuration:

- Operating system: Ubuntu 18.04 64 bit
- CPU: Intel Core i5 CPU, 2.67GHz
- Memory: Total size of 3.65 GB

On the notebook, two PostgreSQL 9.6.9 DBMSs have been installed. One as out-of-the-box installation (hereinafter referred to as standard database) and one with temporal extensions by university of Bozen-Bolzano (hereinafter referred to as temporal database). Both run in parallel on two different listening ports.

The DBMSs have been configured equally. All parameters of the PostgreSQL servers are kept to default values, besides the following four values in the `postgres.conf` which have been set to best meet the hardware configuration of the notebook:

- Postgres `effective_cache_size` = 2.00 GB
- Postgres `shared_buffers` = 512 MB
- Postgres `work_mem` = 8 MB

The data which is used to benchmark the query executions is kept in instances of the Activiti 6.0.0 default database which includes indexes on some columns. To not bias our benchmark by patchy configured indexes, we added additional indexes to `ACT_HI` table columns of data type `timestamp` and to columns which act as foreign keys. Additionally, constraints have been configured to be deferrable, which should not affect the processing of `SELECT` statements. Apart from that, no changes have been made to the databases.

The queries are executed on three different datasets originating in 250, 1,000 and 4,000 simulated process iterations. The period timestamps are recorded at the granularity of milliseconds and have durations up to 4 minutes. Beginning and end of a period are stored in two different columns of data type `timestamp` (data type `tsrange` is not used in Activiti database). Table 5.1 lists detailed information on the benchmark data sets and the size of benchmark relevant tables.

	<b>simulation A</b>	<b>simulation B</b>	<b>simulation C</b>
Process iterations	250	1,000	4,000
Rows in <code>ACT_HI_PROCINST</code>	250	1,000	4,000
Rows in <code>ACT_HI_ACTINST</code>	3029	11,999	48,308
Rows in <code>ACT_HI_TASKINST</code>	934	3,659	14,852
Rows in <code>ACT_ID_USER</code>	5	5	5
Rows in <code>ACT_ID_GROUP</code>	3	3	3
Rows in <code>ACT_ID_MEMBERSHIP</code>	5	5	5
Count query executions	200	100	50

Table 5.1: Figures of databases used in benchmark

To avoid potential influences on the benchmark resulting from inadequate data organization, a clean-up is performed beforehand. Therefore PostgreSQL’s command `VACUUM` was executed prior to every benchmark. “`VACUUM` reclaims storage occupied by dead tuples. [...] Therefore it’s necessary to do `VACUUM` periodically, especially on frequently-updated tables” [Gro19].

To avoid any benchmark external temporary system utilization to affect our benchmark, each query has been executed several times over a longer period of time. Due to shorter expected query execution times on lower volume of data in database of simulation A, queries have been executed more often than e.g. in database of simulation C.

## 5.2 Results and analysis

The following sections describe and comment on the results of the standard to temporal DBMS comparison. Results are mainly presented in diagrams. Blue bars visualize values that refer to the standard PostgreSQL installation, while blue is used for temporal database-related metrics. Detailed figures can be found in appendix B.

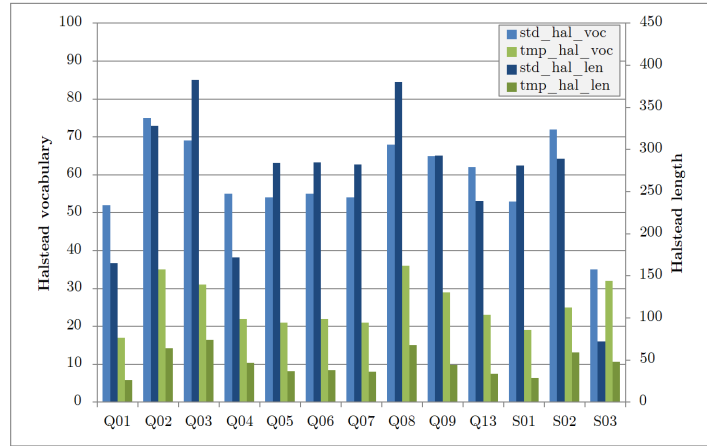
### 5.2.1 Query complexity

At the beginning of the benchmark we compare the query statements themselves. Figure 5.1 shows differences in Halstead metrics of all benchmark queries. The actual values can be found in appendix table B.1. Details on the definition of Halstead metrics in chapter 2.5.

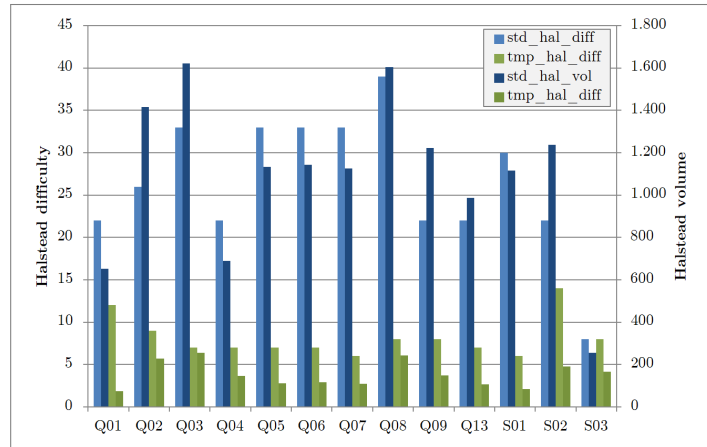
The formulation of statements in standard SQL that query periodic data requires the use of significantly more operators and operands in the total number, but also as distinct symbols, compared to statements in preliminary proposed temporal SQL. This is visualized in figure 5.1a as programme length and vocabulary according to Halstead. Both influence Halstead's volume, which describes the size of the implementation and Halstead's difficulty, a metric to describe error proneness (both shown in 5.1b). The effort to implement or understand a program is proportional to the volume and to the difficulty level of the program and defined as Halstead effort (see figures on logarithmic scale in diagram 5.1c).

In summary, the temporal database allows a much easier retrieval of period related data. The database queries are significantly shorter and less complex and therefore simpler to formulate. If technically experienced end users are to query process execution data directly in the database, the use of a temporal database is worth considering. But also in the development, extension or customization of BPM software, the possibility of simplified querying of period data can be beneficial to the implementation.

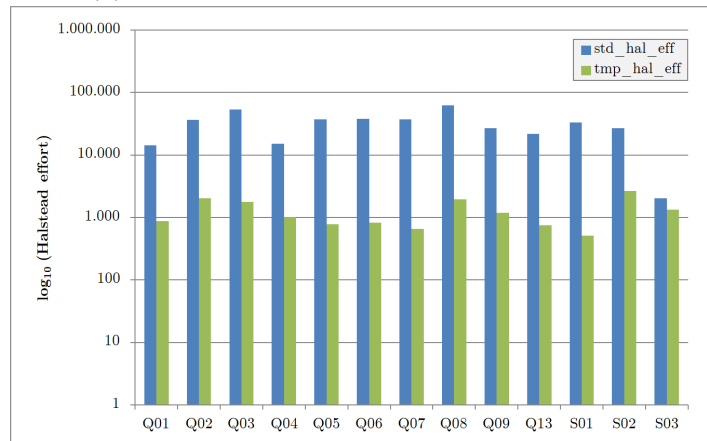
## 5. BENCHMARK: PROCESSING QUERIES ON WORKFLOW CONTROL DATA



(a) Halstead program length and vocabulary



(b) Halstead program volume and difficulty



(c) Halstead program effort

Figure 5.1: Benchmark result: Halstead metrics

### 5.2.2 Query planning performance

Since all defined benchmark queries can also occur as sub queries which only need the first row to be fetched (e.g. usage of `EXISTS` operator), we also take a look at the planning times of the queries.

Figure 5.2 visualizes planning times of queries being executed in a standard and a temporal database. Queries with non-significant differences in the average planning times when executed in a temporal and a non-temporal database are marked with '\*'. Detailed figures can be found in appendix B. In the majority of the query benchmarks, the less complex temporal statements also lead to shorter planning times. In two cases, the standard query statement outperformed the temporal statement independent from the size of the queried data set:

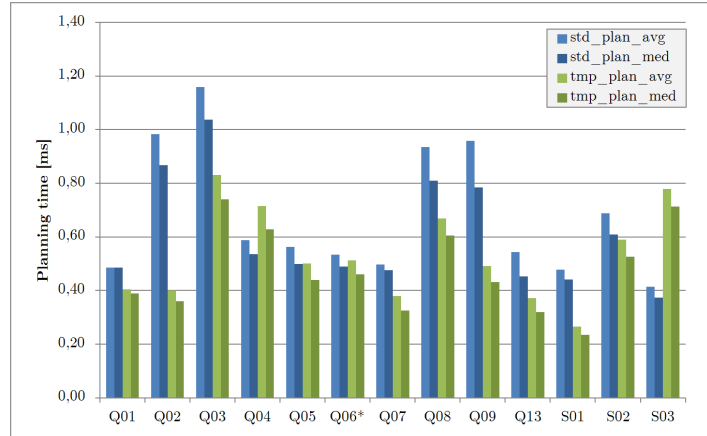
**Disjoint periods** In some queries we had to perform `DISTINCT PERIOD` before applying set operations such as `EXCEPT PERIOD` or `INTERSECT PERIOD`. This is due to a known limitation of the temporal database, which requires input relations of set operations (`UNION`, `EXCEPT`, `INTERSECT`) to be disjoint [Mos16]. The additional step does not only increase the query execution time but also the planning time.

The planning time of the temporary database develops negatively with increasing size of the queried database compared to the planning time in the non-temporary database. As Q04 is in both dialects rather short, the use of `DISTINCT PERIOD` did cause the temporal statement to underperform compared to the standard SQL query independent from the size of the queried database. In the case of S02 the planning time becomes significantly shorter in the non-temporal database when querying the data set of simulation C.

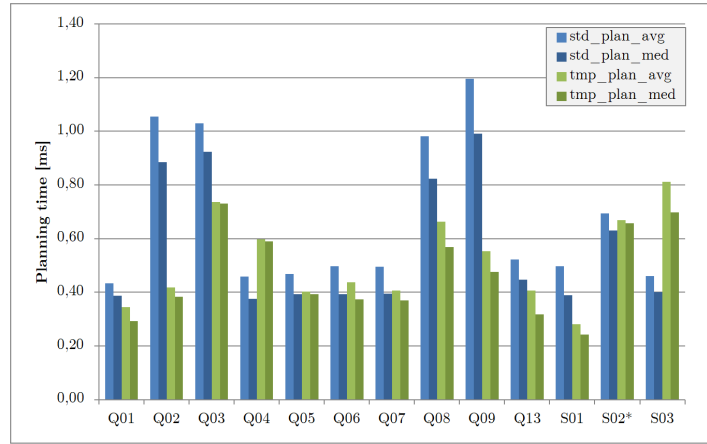
As Q04 is in both dialects rather short, the use of `DISTINCT PERIOD` did cause the temporal statement to underperform compared to the standard SQL query independent from the size of the queried database. Also in the case of S02, the planning time of the temporary database develops negatively with increasing size of the queried database compared to the planning time in the non-temporary database and becomes significantly better in the non-temporal database when querying the data set of simulation C.

**Non period centric alternatives** In S03, a different, non-period focused and considerably simpler query structure has been chosen in standard SQL, which lead to a significantly lower planning time.

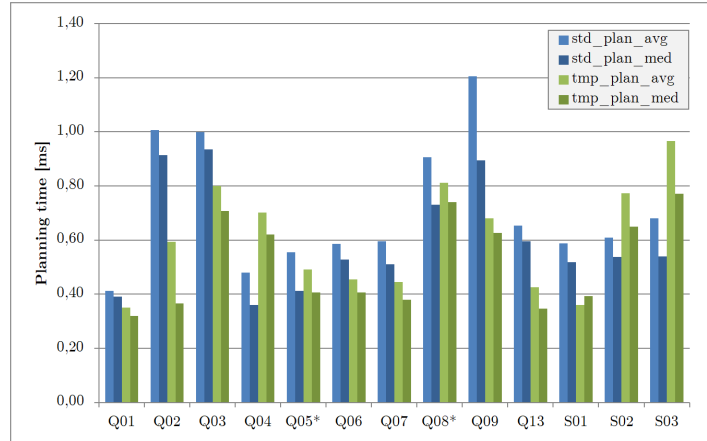
In most of the cases, lower query complexity of temporal statements is also reflected in lower planning times. In one of the two significant cases where this could not be observed we faced a known limitation in the temporal database (and open topic for implementation). In the second case, we followed a complete different query approach in the standard query, which could anyhow also be used having a temporal database in place.



(a) Simulation A - 250 process iterations



(b) Simulation B - 1,000 process iterations



(c) Simulation C - 4,000 process iterations

Figure 5.2: Benchmark results: Average and median query planning times of query processings on different sized databases.

### 5.2.3 Query execution performance

Diagrams in figure 5.4 show differences in the performance of query executions in a standard and temporal PostgreSQL database. Queries having no significant differences in the average execution times in both databases are marked with '\*'. Figure 5.3 compares the average execution times of queries in the temporal database relative to the execution times in the standard database on all three simulation data sets.

Compared to the benchmark results for complexity and planning times of queries, the results regarding the query execution times have no general tendency for the temporal or standard database to perform better. Even though almost all query execution metrics show a significant difference between the performance of the two databases, there is no principle advantage or overall trend. In 7 out of 13 queries, the standard DBMS was able to retrieve data significantly faster from all three different sized databases. However, the other results differ from query to query and depend on the data volume of the queried database. The following paragraphs try to isolate and name indicators and reasons which cause differences the performance of both DBMSs to the disadvantage of the temporal database:

**Limited query fine tuning** The temporal DBMS offers dedicated features for querying period data. However, independent from the utilized DBMS technology, it is always important to adapt and fine tune a query based on the expected volume of data and query planner results. In that respect, the abstraction of determining the set of relevant, disjoint periods in temporal SQL can be a hindrance as early delimitation and filtering of data cannot be done.

As an example we would like to take a closer look at the outlier Q02: When processing Q02 the planner chooses a costly **Merge Join** in the temporal database. This would have been also the case in the standard database when using the alternative statement A.6, as we did in our first attempt. Rewriting the statement to A.4 avoids an **INNER JOIN** and reduces the execution time dramatically. A similar approach was not possible when utilizing temporal SQL for period related parts of the temporal statement, as it always comes to a **GROUP BY** to determine the set of smallest disjoint periods.

**Periods in CTEs** The temporal DBMS makes unexceptional use of CTEs for generated periods. "CTEs do not support selection push-down, or similar performance enhancements, because they get parsed, optimized, and executed as independent queries" [Mos16]. CTEs are not generally detrimental, as in some cases they also speed up query processing. Querying period data in standard SQL allows this decision to be made individually from query to query.

**Disjoint periods** The temporal DBMS requires input relations of set operations (**UNION**, **EXCEPT**, **INTERSECT**) to be disjoint [Mos16] and therefore requires the processing of **DISTINCT PERIOD** on non-disjoint input relations beforehand. This is not of relevance if input relations are disjoint by design (e.g. periods of responsibility for departments

in organizations). However, in BPM related data set, overlapping periods are rather common. An example of the need to use **INTERSECT** is query S02.

**Non period-centric alternatives** In some cases, data requirements for periods can also be satisfied by querying data in a conventional, non period-centric way. Instead of determining all possible periods to reduce them by a criteria or intersect them with other periods as it is done in period-centric statements, a classical approach of querying data might be more applicable and more performant. This was e.g. the case in query S03. It shows, that a blind utilization of temporal features is not always the best way, as period-centric queries do not necessarily advance other approaches.

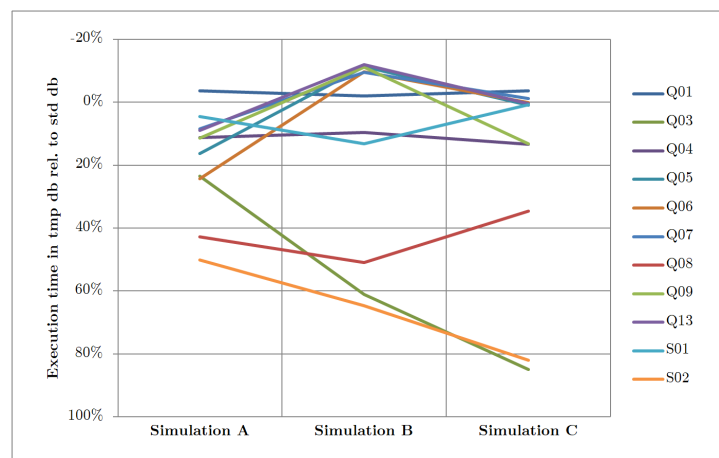
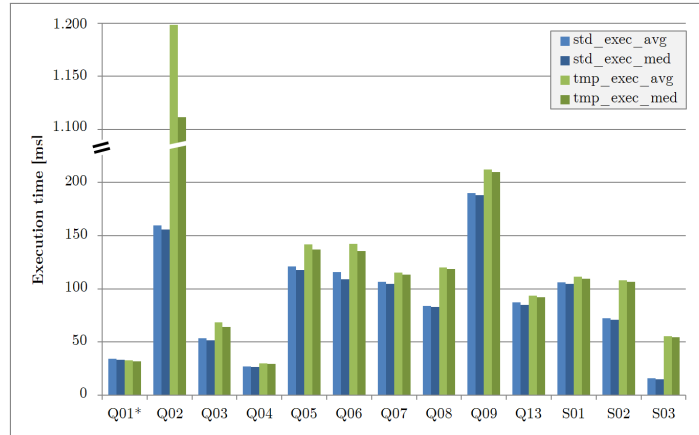
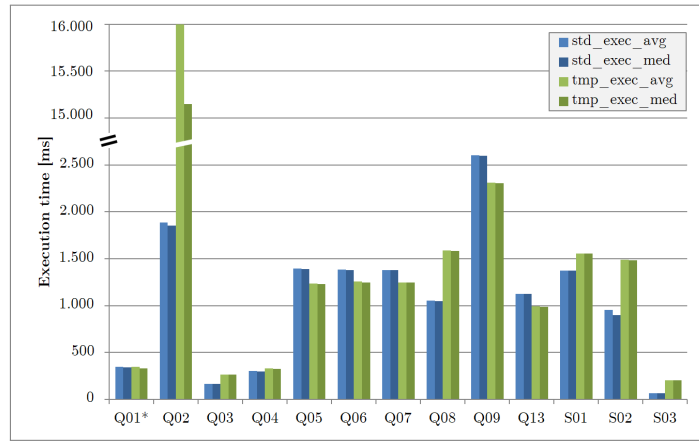


Figure 5.3: Development average execution times

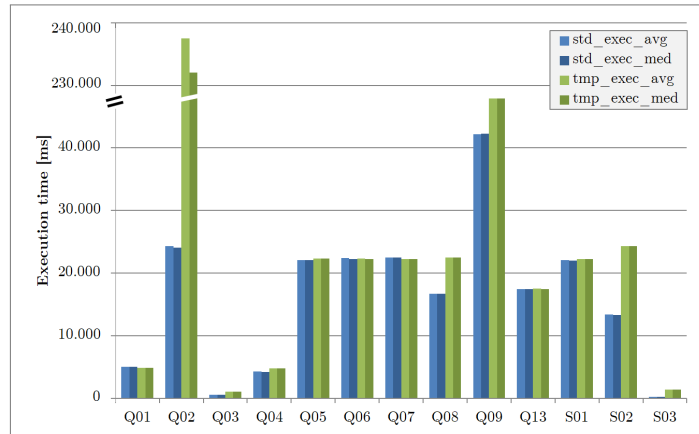




(a) Simulation A - 250 process iterations



(b) Simulation B - 1,000 process iterations



(c) Simulation C - 4,000 process iterations

Figure 5.4: Benchmark results: Average and median query execution times of query processings on different sized databases.

### 5.3 Summary of benchmark results

The performance of a standard and a temporal DBMS in processing queries for periodic data in context of BPM has been evaluated. Therefore, queries have been formulated to serve BPM relevant data requirements as comprehensively as possible. However, the selection of queries used in the benchmark is not exhaustive. More complex special queries were dispensed with, as they would anyway only represent a specialisation of the basic queries. Therefore, the benchmark result in terms of the number of outperforming queries cannot be regarded as a quantitative measure to judge if one of the two DBMS technologies works better.

The benchmark could show that the use of the temporal extension of PostgreSQL DBMS offers support to write considerably simpler query statements. This is beneficial for the development of BPM tools but also for ad-hoc queries on workflow control data. Simpler statements also lead to lower planning times during query executions, for most of the benchmarked queries.

Limited possibilities to fine tune temporal queries due to the abstraction of period generation as well as known issues and limitations of the research prototype (documented in [Mos16]) are reasons causing the temporal DBMS to not outperform the DBMS in execution times. E.g. as periods in workflow control data are rarely disjoint, conditions on input relations for set operators are of high relevance for the application of the temporal DBMS in context of BPM.

The benchmark did not only compare the standard and temporal DBMS capabilities, but also the query design itself. Thereby, a consistently weaker query design at the burden of one of the DBMSs would have biased the result. Therefore, the goal was to write the statements equally, except the use of temporal symbols for period related operations in the temporal DBMS and advanced possibilities to fine tune period generation in the standard DBMS. Two queries are an exception of this principle: S02 and S03 show that it can be beneficial to also consider a non-period centric querying approach.

As all standard SQL statements could have also been processed in the temporal DBMS, the usage of temporal PostgreSQL is advantageous in the sense that it can be decided individually during the query design if to utilize temporal or standard SQL.

## Conclusion

This work evaluates potential benefits of using a temporal database in context of BPM. Therefore, typical time-related queries executed on process execution data during process analysis have been identified. These queries have been executed on a temporal and a non-temporal database to compare query execution performance of both DBMSs. Furthermore, the complexity to develop these query statements in standard and temporal SQL has been measured and compared.

As supportive artifacts, a simulation and a benchmark application have been implemented. The simulation application generates workflow execution data for any user specified business process. It reads a given BPMN 2.0 model and simulates a given number of process iterations which are started within a given time frame and executed in an Activiti 6.0 process engine.

The benchmark application repeatedly executes user provided query statements on a PostgreSQL database and reads the server-side execution and planning times by parsing the result of PostgreSQL command `EXPLAIN ANALYZE`. Furthermore, query complexity metrics according to Halstead are calculated for all provided queries.

For the benchmark, thirteen query statements have been written in standard and temporal SQL. The queries in temporal SQL were significantly less complex compared to the ones in standard SQL. The proposed SQL extension built on top of the two new temporal operators `ALIGN` and `NORMALIZE` makes shorter and simpler query statements possible. Subsequently, the statements have been executed in a standard and temporal PostgreSQL installation on differently sized data sets. The query planning and execution performance did not clearly indicate that the temporal database is outperforming the standard database. Potential reasons are known limitations in the current implementation of the temporal DBMS as well as limited support of query optimizations for CTEs, which are commonly used during temporal query processing.

In addition, data models from databases that store workflow execution data make it difficult for the temporal database to leverage its strengths. Tables, storing process execution data, usually have a hierarchical relationship to each other in terms of the period times to be expected. E.g. process execution periods will always start before related activities are started and will never end before all related activities have been executed. In this data setup, period joins are rarely of use. Furthermore, periods within one table are often overlapping as processes or tasks are often executed in parallel. Therefore, periods often have to be disjoint before further processing in the temporal query logic is possible.

Nevertheless, the much simpler query design makes the use of temporal DBMS worth considering. Especially if a large number of queries are to be developed or if they are developed ad-hoc by the (experienced) end user himself / herself. Also for developers who set up and customize WfMS for many, often smaller end customers, the simpler development of queries in a temporal database can offer a significant advantage.

## 6.1 Limits of work and results

This work only considers advantages or disadvantages of using a temporal database when querying historic workflow control data during process analysis. As the current implementation of the temporal database does not support open intervals, the query performance was not considered when monitoring currently running processes. It must also be considered that not only the performance in querying data but also the performant manipulation of data can be relevant in context of BPM. This dimension has also not been considered in this work.

The results of the benchmark does not necessarily indicate a difference in performance in real world production use-cases, as a single query is executed several times on the same data. The execution might profit from caching effects in this benchmark environment much more than a production environment. The benchmark has been performed on a local work station which might also affect the applicability of the result. WfMSs usually run on more performant server environments. Furthermore, the performance of query executions is dependent on results of PostgreSQL's query planning and the chosen execution algorithms, which takes system and database configurations into account. The overall execution performance is therefore dependent on the chosen database configurations, which may have influenced the results.

## 6.2 Future work

Goal of the first prototype of the temporal database was a tight integration into an already existing DBMS. "The implementation focuses on achieving a cost effective (i.e., minimal changes to the host DBMS) and tightly integrated solution that leverages the services of an existing DBMS" [DBGJ16]. The authors name the topic of performance improvements of single components as a subsequent topic of research. But not only non-functional, also

functional improvements can be expected in the further development of the prototype. An update of the benchmark might be of interest after further improvements have been made, such as fixing known limitations resulting from the not yet final state of the implementation or being related to the selected (already outdated) PostgreSQL version.

A new version of Activiti was released during this work was written. The new version offers, among other new features, the integration of external personnel management systems or directory services. These systems usually also keep track of changes in personnel and responsibilities - they store historic developments over time. This data adds value to explain workflow control related data. In previous Activiti versions, company hierarchies were stored statically. Only the personell structure at the time of query execution could be used, without taking any (historic) changes over time into account.

For Dignös et al, it was of “interest to apply the proposed temporal DBMS in various application contexts”[DBGJ16]. This work did the first step to contextualize their research work. An application of the database in other contexts is still relevant.

In order to extend the range of potential benefits of the temporal database, the research prototype could be extended by further temporal features. The introduction of temporal indexes might speed up query processing [LYSY11]. Combi and Pozzi discussed on how the support for temporal data at trigger level can be beneficial for WfMS [CP04]. Furthermore, “commercial workflow systems are usually rather limited in their ability to specify temporal conditions for each individual activity or for the global plan and do not provide temporal reasoning” [BWJ02], which might be a promising area of further development. Additionally, not only temporal queries, but also temporal updates might be supported, to e.g. enable addition or subtraction of x per month, aligned to the according interval lengths.



## Query Code Listings

This appendix chapter lists queries which have been used during the benchmark. See more information about the queries in chapter 4.5.4. Each query is once written in standard SQL and once in temporal SQL.

## A.1 Query Q01 Count of open processes over time

---

**Listing A.1** Query Q01 Count of open processes o.t. - standard db

---

```
1 SELECT periods.p_start, periods.p_end, COUNT(*) FROM ACT_HI_PROCIINST
2 LEFT OUTER JOIN
3 (SELECT period_start.border AS p_start, period_end.border AS p_end FROM
4 (SELECT ROW_NUMBER() OVER (ORDER BY border) r, border FROM
5 (SELECT start_time_ AS border FROM ACT_HI_PROCIINST
6 UNION SELECT end_time_ AS border FROM ACT_HI_PROCIINST) period_borders) period_start
7 JOIN
8 (SELECT ROW_NUMBER() OVER (ORDER BY border) r, border FROM
9 (SELECT start_time_ AS border FROM ACT_HI_PROCIINST
10 UNION SELECT end_time_ AS border FROM ACT_HI_PROCIINST) period_borders) period_end
11 ON period_end.r=(period_start.r+1)) periods
12 ON ACT_HI_PROCIINST.start_time_ <= periods.p_start AND ACT_HI_PROCIINST.end_time_ >= periods.p_end
13 GROUP BY periods.p_start, periods.p_end
14 ORDER BY periods.p_start;
```

---

---

**Listing A.2** Query Q01 Count of open processes o.t. - standard db (alternative)

---

```
1 SELECT period_borders.border, COUNT(*) FROM ACT_HI_PROCIINST
2 LEFT OUTER JOIN
3 (SELECT start_time_ as border FROM ACT_HI_PROCIINST
4 UNION SELECT end_time_ as border FROM ACT_HI_PROCIINST) period_borders
5 ON ACT_HI_PROCIINST.start_time_ <= period_borders.border
6 AND ACT_HI_PROCIINST.end_time_ > period_borders.border
7 GROUP BY period_borders.border
8 ORDER BY period_borders.border;
```

---

---

**Listing A.3** Query Q01 Count of open processes o.t. - temporal db

---

```
1 SELECT start_time_, end_time_, COUNT(*)
2 FROM ACT_HI_PROCIINST
3 GROUP BY PERIOD WITH(start_time_, end_time_)
4 ORDER BY start_time_;
```

---



## A.2 Query Q02 Count of active processes over time

**Listing A.4** Query Q02 Count of active processes o.t. - standard db

```

1 WITH periods AS
2 (SELECT period_start.border AS p_start, period_end.border AS p_end FROM
3 (SELECT ROW_NUMBER() OVER (ORDER BY border) r, border FROM
4 (SELECT claim_time_ AS border FROM ACT_HI_TASKINST
5 UNION (SELECT start_time_ AS border FROM ACT_HI_ACTINST
6 WHERE duration_ > 0 AND task_id_ IS NULL)
7 UNION SELECT end_time_ AS border FROM ACT_HI_TASKINST
8 UNION (SELECT end_time_ AS border FROM ACT_HI_ACTINST
9 WHERE duration_ > 0 AND task_id_ IS NULL)) period_borders) period_start
10 JOIN
11 (SELECT ROW_NUMBER() OVER (ORDER BY border) r, border FROM
12 (SELECT claim_time_ AS border FROM ACT_HI_TASKINST
13 UNION (SELECT start_time_ AS border FROM ACT_HI_ACTINST
14 WHERE duration_ > 0 AND task_id_ IS NULL)
15 UNION SELECT end_time_ AS border FROM ACT_HI_TASKINST
16 UNION (SELECT end_time_ AS border FROM ACT_HI_ACTINST
17 WHERE duration_ > 0 AND task_id_ IS NULL)) period_borders) period_end
18 ON period_end.r=(period_start.r+1)
19 )
20 SELECT periods.p_start, periods.p_end, COUNT(*) FROM ACT_HI_PROCIINST
21 LEFT OUTER JOIN periods
22 ON ACT_HI_PROCIINST.start_time_ <= periods.p_start
23 AND ACT_HI_PROCIINST.end_time_ >= periods.p_end
24 RIGHT OUTER JOIN
25 (SELECT start_time_ AS p_start, end_time_ as p_end, proc_inst_id_ FROM ACT_HI_ACTINST
26 WHERE duration_ > 0 AND task_id_ IS NULL
27 UNION SELECT claim_time_ AS p_start, end_time_ AS p_end, proc_inst_id_ FROM ACT_HI_TASKINST
28 ) active_times
29 ON active_times.proc_inst_id_=ACT_HI_PROCIINST.id_
30 AND periods.p_start >= active_times.p_start AND periods.p_end <= active_times.p_end
31 GROUP BY periods.p_start, periods.p_end
32 ORDER BY periods.p_start;
```

**Listing A.5** Query Q02 Count of active processes o.t. - temporal db

```

1 SELECT act_start_time_, end_time_, COUNT(DISTINCT proc_inst_id_) FROM
2 (SELECT proc_inst_id_, start_time_ as act_start_time_, end_time_ FROM ACT_HI_ACTINST
3 WHERE task_id_ IS NULL AND duration_ > 0
4 UNION
5 SELECT PROC_INST_ID_, claim_time_ as act_start_time_, end_time_ FROM ACT_HI_TASKINST) active_times
6 GROUP BY PERIOD WITH(act_start_time_, end_time_)
7 ORDER BY act_start_time_;
```

---

**Listing A.6** Query Q02 Count of active processes o.t. - standard db (alternative)

---

```

1 WITH all_periods AS
2 (SELECT period_start.border AS p_start, period_end.border AS p_end FROM
3  (SELECT ROW_NUMBER() OVER (ORDER BY border) r, border FROM
4  (SELECT claim_time_ AS border FROM ACT_HI_TASKINST
5   UNION (SELECT start_time_ AS border FROM ACT_HI_ACTINST
6    WHERE duration_ > 0 AND task_id_ IS NULL)
7   UNION SELECT end_time_ AS border FROM ACT_HI_TASKINST
8   UNION (SELECT end_time_ AS border FROM ACT_HI_ACTINST
9    WHERE duration_ > 0 AND task_id_ IS NULL)) period_borders) period_start
10 JOIN
11 (SELECT ROW_NUMBER() OVER (ORDER BY border) r, border FROM
12  (SELECT claim_time_ AS border FROM ACT_HI_TASKINST
13   UNION (SELECT start_time_ AS border FROM ACT_HI_ACTINST
14    WHERE duration_ > 0 AND task_id_ IS NULL)
15   UNION SELECT end_time_ AS border FROM ACT_HI_TASKINST
16   UNION (SELECT end_time_ AS border FROM ACT_HI_ACTINST
17    WHERE duration_ > 0 AND task_id_ IS NULL)) period_borders) period_end
18 ON period_end.r=(period_start.r+1)
19 )
20 SELECT all_periods.p_start, all_periods.p_end, COUNT(*) AS count_assignments FROM all_periods
21 JOIN
22 (SELECT start_time_ AS p_start, end_time_ as p_end, proc_inst_id_ FROM ACT_HI_ACTINST
23  WHERE duration_ > 0 AND task_id_ IS NULL
24  UNION SELECT claim_time_ AS p_start, end_time_ as p_end, proc_inst_id_ FROM ACT_HI_TASKINST
25 ) active_times
26 ON all_periods.p_start >= active_times.p_start
27 AND all_periods.p_end <= active_times.p_end
28 GROUP BY all_periods.p_start, all_periods.p_end
29 ORDER BY all_periods.p_start;

```

---

## A.3 Query Q03 Non-active (idle) periods per process

**Listing A.7** Query Q03 Non-active (idle) periods per process - standard db

```

1 SELECT all_periods.p_start, all_periods.p_end, all_periods.proc_inst_id_ FROM
2   (SELECT period_start.proc_inst_id_, period_start.border AS p_start,
3     period_end.border AS p_end FROM
4     (SELECT ROW_NUMBER() OVER (ORDER BY proc_inst_id_, border) r, proc_inst_id_, border FROM
5       (SELECT proc_inst_id_, claim_time_ AS border FROM ACT_HI_TASKINST
6         UNION (SELECT proc_inst_id_, start_time_ AS border FROM ACT_HI_ACTINST
7           WHERE duration_ > 0 AND task_id_ IS NULL)
8         UNION SELECT proc_inst_id_, start_time_ AS border FROM ACT_HI_PROCIINST
9         UNION SELECT proc_inst_id_, end_time_ AS border FROM ACT_HI_PROCIINST
10        UNION SELECT proc_inst_id_, end_time_ AS border FROM ACT_HI_TASKINST
11        UNION (SELECT proc_inst_id_, end_time_ AS border FROM ACT_HI_ACTINST
12          WHERE duration_ > 0 AND task_id_ IS NULL)) period_borders) period_start
13 JOIN
14   (SELECT ROW_NUMBER() OVER (ORDER BY proc_inst_id_, border) r, proc_inst_id_, border FROM
15     (SELECT proc_inst_id_, claim_time_ AS border FROM ACT_HI_TASKINST
16       UNION (SELECT proc_inst_id_, start_time_ AS border FROM ACT_HI_ACTINST
17         WHERE duration_ > 0 AND task_id_ IS NULL)
18       UNION SELECT proc_inst_id_, start_time_ AS border FROM ACT_HI_PROCIINST
19       UNION SELECT proc_inst_id_, end_time_ AS border FROM ACT_HI_PROCIINST
20       UNION SELECT proc_inst_id_, end_time_ AS border FROM ACT_HI_TASKINST
21       UNION (SELECT proc_inst_id_, end_time_ AS border FROM ACT_HI_ACTINST
22         WHERE duration_ > 0 AND task_id_ IS NULL)) period_borders) period_end
23 ON period_end.r=(period_start.r+1)
24 AND period_start.proc_inst_id_=period_end.proc_inst_id_) all_periods
25 LEFT OUTER JOIN
26   (SELECT start_time_ AS p_start, end_time_ as p_end, proc_inst_id_ FROM ACT_HI_ACTINST
27     WHERE duration_ > 0 AND task_id_ IS NULL
28     UNION SELECT claim_time_ AS p_start, end_time_ as p_end, proc_inst_id_ FROM ACT_HI_TASKINST
29   ) active_times
30 ON all_periods.p_start >= active_times.p_start
31 AND all_periods.p_end <= active_times.p_end
32 AND all_periods.proc_inst_id_=active_times.proc_inst_id_
33 WHERE active_times.proc_inst_id_ IS NULL
34 ORDER BY all_periods.proc_inst_id_, all_periods.p_start;

```

**Listing A.8** Query Q03 Non-active (idle) periods per process - temporal db

```

1 SELECT start_time_, end_time_, ID_ FROM ACT_HI_PROCIINST
2 EXCEPT PERIOD WITH (start_time_, end_time_)
3 SELECT PERIOD DISTINCT WITH (start_time_, end_time_) start_time_, end_time_, proc_inst_id_ FROM
4   (SELECT start_time_, end_time_, proc_inst_id_ FROM ACT_HI_ACTINST WHERE task_id_ IS NULL
5   UNION
6   SELECT claim_time_, end_time_, proc_inst_id_ FROM ACT_HI_TASKINST) AS distinct_active_periods
7 ORDER BY id_, start_time_;

```

## A.4 Query Q04 Periods with no open processes

---

**Listing A.9** Query Q04 Periods with no open processes - standard db

---

```
1 SELECT periods.p_start, periods.p_end FROM ACT_HI_PROCIINST
2 RIGHT OUTER JOIN
3 (SELECT period_start.border AS p_start, period_end.border AS p_end FROM
4  (SELECT ROW_NUMBER() OVER (ORDER BY border) r, border FROM
5  (SELECT start_time_ AS border FROM ACT_HI_PROCIINST
6  UNION SELECT end_time_ AS border FROM ACT_HI_PROCIINST) period_borders) period_start
7 JOIN
8  (SELECT ROW_NUMBER() OVER (ORDER BY border) r, border FROM
9  (SELECT start_time_ AS border FROM ACT_HI_PROCIINST
10 UNION SELECT end_time_ AS border FROM ACT_HI_PROCIINST) period_borders) period_end
11 ON period_end.r=(period_start.r+1)) periods
12 ON ACT_HI_PROCIINST.start_time_ <= periods.p_start
13 AND ACT_HI_PROCIINST.end_time_ >= periods.p_end
14 WHERE ACT_HI_PROCIINST.ID_ IS NULL
15 GROUP BY periods.p_start, periods.p_end, ACT_HI_PROCIINST.ID_
16 ORDER BY periods.p_start;
```

---

---

**Listing A.10** Query Q04 Periods with no open processes - temporal db

---

```
1 SELECT min(start_time_) AS period_start_time, max(end_time_) AS period_end_time_
2 FROM ACT_HI_PROCIINST
3 EXCEPT PERIOD WITH (period_start_time, period_end_time_)
4 SELECT PERIOD DISTINCT WITH (start_time_, end_time_) start_time_, end_time_ FROM ACT_HI_PROCIINST
5 ORDER BY period_start_time;
```

---

## A.5 Query Q05 Count of assigned tasks to user over time

---

**Listing A.11** Query Q05 Count of assigned tasks to user o.t. - standard db

---

```

1 SELECT periods.p_start, periods.p_end, COUNT(*), ACT_HI_TASKINST.assignee_ FROM ACT_HI_TASKINST
2 LEFT OUTER JOIN
3 (SELECT period_start.assignee_, period_start.border AS p_start, period_end.border AS p_end FROM
4 (SELECT ROW_NUMBER() OVER (ORDER BY assignee_, border) r, assignee_, border FROM
5 (SELECT assignee_, start_time_ AS border FROM ACT_HI_TASKINST
6 UNION SELECT assignee_, end_time_ AS border FROM ACT_HI_TASKINST) period_borders) period_start
7 JOIN
8 (SELECT ROW_NUMBER() OVER (ORDER BY assignee_, border) r, assignee_, border FROM
9 (SELECT assignee_, start_time_ AS border FROM ACT_HI_TASKINST
10 UNION SELECT assignee_, end_time_ AS border FROM ACT_HI_TASKINST) period_borders) period_end
11 ON period_end.r=(period_start.r+1) AND period_end.assignee_=period_start.assignee_) periods
12 ON ACT_HI_TASKINST.assignee_ = periods.assignee_
13 AND start_time_ <= periods.p_start AND end_time_ >= periods.p_end
14 GROUP BY ACT_HI_TASKINST.assignee_, periods.p_start, periods.p_end
15 ORDER BY ACT_HI_TASKINST.assignee_, periods.p_start;

```

---



---

**Listing A.12** Query Q05 Count of assigned tasks to user o.t. - temporal db

---

```

1 SELECT start_time_, end_time_, COUNT(*), assignee_
2 FROM ACT_HI_TASKINST
3 GROUP BY PERIOD WITH(start_time_, end_time_) assignee_
4 ORDER BY assignee_, start_time_;

```

---

## A.6 Query Q06 Count of claimed user tasks over time

---

**Listing A.13** Query Q06 Count of claimed user tasks o.t. - standard db

---

```
1 SELECT ACT_HI_TASKINST.assignee_, periods.p_claim, periods.p_end, COUNT(*) FROM ACT_HI_TASKINST
2 LEFT OUTER JOIN
3 (SELECT period_claim.assignee_, period_claim.border AS p_claim, period_end.border AS p_end FROM
4 (SELECT ROW_NUMBER() OVER (ORDER BY assignee_, border) r, assignee_, border FROM
5 (SELECT assignee_, claim_time_ AS border FROM ACT_HI_TASKINST
6 UNION SELECT assignee_, end_time_ AS border FROM ACT_HI_TASKINST) period_borders) period_claim
7 JOIN
8 (SELECT ROW_NUMBER() OVER (ORDER BY assignee_, border) r, assignee_, border FROM
9 (SELECT assignee_, claim_time_ AS border FROM ACT_HI_TASKINST
10 UNION SELECT assignee_, end_time_ AS border FROM ACT_HI_TASKINST) period_borders) period_end
11 ON period_end.r=(period_claim.r+1) AND period_end.assignee_=period_claim.assignee_) periods
12 ON ACT_HI_TASKINST.assignee_=periods.assignee_
13 AND claim_time_ <= periods.p_claim AND end_time_ >= periods.p_end
14 GROUP BY ACT_HI_TASKINST.assignee_, periods.p_claim, periods.p_end
15 ORDER BY ACT_HI_TASKINST.assignee_, periods.p_claim;
```

---

---

**Listing A.14** Query Q06 Count of claimed user tasks o.t. - temporal db

---

```
1 SELECT assignee_, claim_time_, end_time_, COUNT(*)
2 FROM ACT_HI_TASKINST
3 GROUP BY PERIOD WITH(claim_time_, end_time_) assignee_
4 ORDER BY assignee_, claim_time_;
```

---

## A.7 Query Q07 Count of not-yet claimed tasks per user

---

**Listing A.15** Query Q07 Count of not-yet claimed tasks per user - standard db

```

1 SELECT ACT_HI_TASKINST.assignee_, periods.p_start, periods.p_claim, COUNT(*) FROM ACT_HI_TASKINST
2 LEFT OUTER JOIN
3 (SELECT period_start.assignee_, period_start.border AS p_start, period_claim.border AS p_claim FROM
4 (SELECT ROW_NUMBER() OVER (ORDER BY assignee_, border) r, assignee_, border FROM
5 (SELECT assignee_, start_time_ AS border FROM ACT_HI_TASKINST
6 UNION SELECT assignee_, claim_time_ AS border FROM ACT_HI_TASKINST) period_borders) period_start
7 JOIN
8 (SELECT ROW_NUMBER() OVER (ORDER BY assignee_, border) r, assignee_, border FROM
9 (SELECT assignee_, start_time_ AS border FROM ACT_HI_TASKINST
10 UNION SELECT assignee_, claim_time_ AS border FROM ACT_HI_TASKINST) period_borders) period_claim
11 ON period_claim.r=(period_start.r+1) AND period_claim.assignee_=period_start.assignee_) periods
12 ON ACT_HI_TASKINST.assignee_=periods.assignee_
13 AND start_time_ <= periods.p_start AND claim_time_ >= periods.p_claim
14 GROUP BY ACT_HI_TASKINST.assignee_, periods.p_start, periods.p_claim
15 ORDER BY ACT_HI_TASKINST.assignee_, periods.p_start;

```

---



---

**Listing A.16** Query Q07 Count of not-yet claimed tasks per user - temporal db

```

1 SELECT assignee_, start_time_, claim_time_, COUNT(*)
2 FROM ACT_HI_TASKINST
3 GROUP BY PERIOD WITH(start_time_, claim_time_) assignee_
4 ORDER BY assignee_, start_time_;

```

---

## A.8 Query Q08 Periods with no assigned tasks to user

---

**Listing A.17** Query Q08 Periods with no assigned tasks to user - standard db

---

```
1 SELECT all_periods.assignee_, all_periods.p_start, all_periods.p_end FROM
2 (SELECT period_start.assignee_, period_start.border AS p_start, period_end.border AS p_end FROM
3 (SELECT ROW_NUMBER() OVER (ORDER BY assignee_, border) r, assignee_, border FROM
4 (SELECT claim_time_ AS border, assignee_ FROM ACT_HI_TASKINST
5 UNION SELECT MIN(start_time_) AS border, ACT_ID_USER.id_ AS assignee_ FROM ACT_HI_PROCIINST
6 CROSS JOIN ACT_ID_USER GROUP BY ACT_ID_USER.id_
7 UNION SELECT MAX(end_time_) AS border, ACT_ID_USER.id_ AS assignee_ FROM ACT_HI_PROCIINST
8 CROSS JOIN ACT_ID_USER GROUP BY ACT_ID_USER.id_
9 UNION SELECT end_time_ AS border, assignee_ FROM ACT_HI_TASKINST
10 ) period_borders) period_start
11 JOIN
12 (SELECT ROW_NUMBER() OVER (ORDER BY assignee_, border) r, assignee_, border FROM
13 (SELECT claim_time_ AS border, assignee_ FROM ACT_HI_TASKINST
14 UNION SELECT MIN(start_time_) AS border, ACT_ID_USER.id_ AS assignee_ FROM ACT_HI_PROCIINST
15 CROSS JOIN ACT_ID_USER GROUP BY ACT_ID_USER.id_
16 UNION SELECT MAX(end_time_) AS border, ACT_ID_USER.id_ AS assignee_ FROM ACT_HI_PROCIINST
17 CROSS JOIN ACT_ID_USER GROUP BY ACT_ID_USER.id_
18 UNION SELECT end_time_ AS border, assignee_ FROM ACT_HI_TASKINST
19 ) period_borders) period_end
20 ON period_end.r=(period_start.r+1) AND period_end.assignee_=period_start.assignee_) all_periods
21 LEFT OUTER JOIN
22 (SELECT claim_time_ AS p_start, end_time_ as p_end, assignee_ FROM ACT_HI_TASKINST) active_times
23 ON all_periods.p_start >= active_times.p_start
24 AND all_periods.p_end <= active_times.p_end
25 AND all_periods.assignee_=active_times.assignee_
26 WHERE active_times.assignee_ IS NULL
27 ORDER BY all_periods.assignee_, all_periods.p_start;
```

---

---

**Listing A.18** Query Q08 Periods with no assigned tasks to user - temporal db

---

```
1 SELECT ACT_ID_USER.id_,
2 min(start_time_) AS period_start_time_, max(end_time_) AS period_end_time_
3 FROM ACT_ID_USER
4 CROSS JOIN ACT_HI_PROCIINST
5 GROUP BY ACT_ID_USER.id_
6 EXCEPT PERIOD WITH (period_start_time_, period_end_time_)
7 SELECT PERIOD DISTINCT WITH (claim_time_, end_time_) assignee_, claim_time_, end_time_
8 FROM ACT_HI_TASKINST
9 ORDER BY id_, period_start_time_;
```

---



## A.9 Query Q09 Count of assigned tasks to department

---

**Listing A.19** Query Q09 Count of assigned tasks to department - standard db

---

```

1 SELECT ACT_ID_MEMBERSHIP.group_id_, periods.p_start, periods.p_end, COUNT(*) FROM ACT_HI_TASKINST
2 JOIN ACT_ID_MEMBERSHIP ON ACT_ID_MEMBERSHIP.user_id = ACT_HI_TASKINST.assignee_
3 LEFT OUTER JOIN
4 (SELECT period_start.group_id_, period_start.border AS p_start, period_end.border AS p_end FROM
5 (SELECT ROW_NUMBER() OVER (ORDER BY group_id_, border) r, group_id_, border FROM
6 (SELECT group_id_, start_time_ AS border FROM ACT_HI_TASKINST
7 JOIN ACT_ID_MEMBERSHIP ON act_id_membership.user_id = assignee_
8 UNION SELECT group_id_, end_time_ AS border FROM ACT_HI_TASKINST
9 JOIN ACT_ID_MEMBERSHIP ON act_id_membership.user_id = assignee_) period_borders) period_start
10 JOIN
11 (SELECT ROW_NUMBER() OVER (ORDER BY group_id_, border) r, group_id_, border FROM
12 (SELECT group_id_, start_time_ AS border FROM ACT_HI_TASKINST
13 JOIN ACT_ID_MEMBERSHIP ON act_id_membership.user_id = assignee_
14 UNION SELECT group_id_, end_time_ AS border FROM ACT_HI_TASKINST
15 JOIN ACT_ID_MEMBERSHIP ON act_id_membership.user_id = assignee_) period_borders) period_end
16 ON period_end.r=(period_start.r+1) AND period_start.group_id_=period_end.group_id_) periods
17 ON ACT_ID_MEMBERSHIP.group_id_=periods.group_id_
18 AND start_time_ <= periods.p_start AND end_time_ >= periods.p_end
19 GROUP BY ACT_ID_MEMBERSHIP.group_id_, periods.p_start, periods.p_end
20 ORDER BY ACT_ID_MEMBERSHIP.group_id_, periods.p_start;

```

---



---

**Listing A.20** Query Q09 Count of assigned tasks to department - temporal db

---

```

1 SELECT group_id_, start_time_, end_time_, COUNT(*)
2 FROM ACT_HI_TASKINST
3 JOIN ACT_ID_MEMBERSHIP ON act_id_membership.user_id = assignee_
4 GROUP BY PERIOD WITH(start_time_, end_time_) group_id_
5 ORDER BY group_id_, start_time_;

```

---

## A.10 Query Q13 Count of assigned tasks to service

---

**Listing A.21** Query Q13 Count of assigned tasks to service - standard db

---

```
1 SELECT ACT_HI_ACTINST.act_name_, periods.p_start, periods.p_end, COUNT(*) FROM ACT_HI_ACTINST
2 LEFT OUTER JOIN
3 (SELECT period_start.act_name_, period_start.border AS p_start, period_end.border AS p_end FROM
4 (SELECT ROW_NUMBER() OVER (ORDER BY act_name_, border) r, act_name_, border FROM
5 (SELECT act_name_, start_time_ AS border FROM ACT_HI_ACTINST
6 WHERE act_type_='serviceTask'
7 UNION SELECT act_name_, end_time_ AS border FROM ACT_HI_ACTINST
8 WHERE act_type_='serviceTask') period_borders) period_start
9 JOIN
10 (SELECT ROW_NUMBER() OVER (ORDER BY act_name_, border) r, act_name_, border FROM
11 (SELECT act_name_, start_time_ AS border FROM ACT_HI_ACTINST
12 WHERE act_type_='serviceTask'
13 UNION SELECT act_name_, end_time_ AS border FROM ACT_HI_ACTINST
14 WHERE act_type_='serviceTask') period_borders) period_end
15 ON period_end.r=(period_start.r+1) AND period_end.act_name_=period_start.act_name_) periods
16 ON ACT_HI_ACTINST.act_name_ = periods.act_name_
17 AND start_time_ <= periods.p_start AND end_time_ >= periods.p_end
18 WHERE act_type_='serviceTask'
19 GROUP BY ACT_HI_ACTINST.act_name_, periods.p_start, periods.p_end
20 ORDER BY ACT_HI_ACTINST.act_name_, periods.p_start;
```

---

---

**Listing A.22** Query Q13 Count of assigned tasks to service - temporal db

---

```
1 SELECT act_name_, start_time_, end_time_, COUNT(*)
2 FROM ACT_HI_ACTINST
3 WHERE act_type_='serviceTask'
4 GROUP BY PERIOD WITH(start_time_, end_time_) act_name_
5 ORDER BY act_name_, start_time_;
```

---

## A.11 Query S01 Distinct periods with claimed tasks per user

---

**Listing A.23** Query S01 Distinct periods with claimed tasks per user - standard db

---

```
1 SELECT ACT_HI_TASKINST.assignee_, periods.p_claim, periods.p_end FROM ACT_HI_TASKINST
2 LEFT OUTER JOIN
3 (SELECT period_claim.assignee_, period_claim.border AS p_claim, period_end.border AS p_end FROM
4 (SELECT ROW_NUMBER() OVER (ORDER BY assignee_, border) r, assignee_, border FROM
5 (SELECT assignee_, claim_time_ AS border FROM ACT_HI_TASKINST
6 UNION SELECT assignee_, end_time_ AS border FROM ACT_HI_TASKINST) period_borders) period_claim
7 JOIN
8 (SELECT ROW_NUMBER() OVER (ORDER BY assignee_, border) r, assignee_, border FROM
9 (SELECT assignee_, claim_time_ AS border FROM ACT_HI_TASKINST
10 UNION SELECT assignee_, end_time_ AS border FROM ACT_HI_TASKINST) period_borders) period_end
11 ON period_end.r=(period_claim.r+1) AND period_end.assignee_=period_claim.assignee_) periods
12 ON ACT_HI_TASKINST.assignee_=periods.assignee_
13 AND claim_time_ <= periods.p_claim AND end_time_ >= periods.p_end
14 GROUP BY ACT_HI_TASKINST.assignee_, periods.p_claim, periods.p_end
15 ORDER BY ACT_HI_TASKINST.assignee_, periods.p_claim;
```

---

---

**Listing A.24** Query S01 Distinct periods with claimed tasks per user - temporal db

---

```
1 SELECT PERIOD DISTINCT WITH(claim_time_, end_time_) assignee_, claim_time_, end_time_
2 FROM ACT_HI_TASKINST
3 ORDER BY assignee_, claim_time_;
```

---

## A.12 Query S02 - Special: Parallel processing of specific user tasks

---

**Listing A.25** Query S02 - Parallel processing of specific user tasks - standard db

---

```
1 WITH periods AS
2 (SELECT period_claim.border AS p_claim, period_end.border AS p_end FROM
3  (SELECT ROW_NUMBER() OVER (ORDER BY border) r, border FROM
4  (SELECT claim_time_ AS border FROM ACT_HI_TASKINST
5   WHERE task_def_key_ IN ('manualRiskAssessment', 'manualCreditApprovalDecision')
6   UNION SELECT end_time_ AS border FROM ACT_HI_TASKINST
7   WHERE task_def_key_ IN ('manualRiskAssessment', 'manualCreditApprovalDecision')
8   ) period_borders) period_claim
9  JOIN
10 (SELECT ROW_NUMBER() OVER (ORDER BY border) r, border FROM
11  (SELECT claim_time_ AS border FROM ACT_HI_TASKINST
12   WHERE task_def_key_ IN ('manualRiskAssessment', 'manualCreditApprovalDecision')
13   UNION SELECT end_time_ AS border FROM ACT_HI_TASKINST
14   WHERE task_def_key_ IN ('manualRiskAssessment', 'manualCreditApprovalDecision')
15   ) period_borders) period_end
16  ON period_end.r=(period_claim.r+1)
17 )
18 SELECT
19 periods.p_claim, periods.p_end
20 FROM ACT_HI_TASKINST task1
21 CROSS JOIN ACT_HI_TASKINST task2
22 RIGHT JOIN periods
23  ON task1.claim_time_ <= periods.p_claim AND task1.end_time_ >= periods.p_end
24  AND task2.claim_time_ <= periods.p_claim AND task2.end_time_ >= periods.p_end
25 WHERE task1.task_def_key_ = 'manualRiskAssessment'
26  AND task2.task_def_key_ = 'manualCreditApprovalDecision'
27 GROUP BY periods.p_claim, periods.p_end
28 ORDER BY periods.p_claim;
```

---

---

**Listing A.26** Query S02 - Parallel processing of specific user tasks - temporal db

---

```
1 SELECT PERIOD DISTINCT WITH(claim_time_, end_time_) claim_time_, end_time_
2 FROM ACT_HI_TASKINST
3 WHERE task_def_key_ = 'manualRiskAssessment'
4 INTERSECT PERIOD WITH (claim_time_, end_time_)
5 SELECT PERIOD DISTINCT WITH(claim_time_, end_time_) claim_time_, end_time_
6 FROM ACT_HI_TASKINST
7 WHERE task_def_key_ = 'manualCreditApprovalDecision'
8 ORDER BY claim_time_;
```

---

## A.13 Query S03 - Parallel processing of activities

---

**Listing A.27** Query S03 - Parallel processing of activities - standard db

---

```
1 SELECT *,
2 GREATEST(act1.start_time_, act2.start_time_) AS parallel_period_start_,
3 LEAST(act1.end_time_, act2.end_time_) AS parallel_period_end_ FROM ACT_HI_ACTINST act1
4 JOIN ACT_HI_ACTINST act2
5 ON act1.proc_inst_id=act2.proc_inst_id AND act1.id_<>act2.id_ AND
6 (act1.start_time_<act2.end_time_ AND act1.end_time_>act2.start_time_)
7 ORDER BY parallel_period_start_;
```

---

---

**Listing A.28** Query S03 - Parallel processing of activities - temporal db

---

```
1 SELECT *
2 FROM (ACT_HI_ACTINST a1 PERIOD
3 JOIN WITH (start_time_, end_time_, start_time_, end_time_)
4 AS (parallel_period_start_, parallel_period_end_) ACT_HI_ACTINST a2
5 ON a1.proc_inst_id=a2.proc_inst_id AND a1.id_ <> a2.id_) parallel_activities
6 ORDER BY parallel_period_start_;
```

---



# APPENDIX B

## Results

This appendix chapter contains tables with detailed benchmark results. Information about the benchmark setup, comments on results and graphics visualizing the benchmark results can be found in chapter 5.

## B.1 Halstead metrics of queries

QID	db	length	vocab.	volume	difficulty	effort
Q01	std	165	52	652	22	14343
	tmp	26	17	74	12	884
Q02	std	328	75	1416	26	36820
	tmp	64	35	228	9	2048
Q03	std	383	69	1622	33	53515
	tmp	74	31	254	7	1779
Q04	std	172	55	689	22	15164
	tmp	47	22	145	7	1017
Q05	std	284	54	1133	33	37385
	tmp	37	21	113	7	789
Q06	std	285	55	1142	33	37689
	tmp	38	22	117	7	822
Q07	std	282	54	1125	33	37121
	tmp	36	21	110	6	658
Q08	std	380	68	1603	39	62533
	tmp	68	36	244	8	1949
Q09	std	293	65	1223	22	26908
	tmp	44	29	148	8	1185
Q13	std	239	62	986	22	21700
	tmp	34	23	107	7	746
S01	std	281	53	1116	30	33470
	tmp	29	19	85	6	512
S02	std	289	72	1236	22	27191
	tmp	59	25	190	14	2659
S03	std	72	35	256	8	2048
	tmp	48	32	166	8	1331

Table B.1: Benchmark metrics: Query complexity according to Halstead



## B.2 Execution times querying data of simulation A

QID	db	exec <sub>avg</sub>	exec <sub>med</sub>	variance	$P_{(F \leq f)}$	t stat	$P_{(T \leq t)}$
Q01	std	33,98	33,05	111,84	0,00	1,58	0,12
	tmp	32,58	31,83	44,03			
Q02	std	159,82	155,99	235,94	0,00	-29,76	0,00
	tmp	1198,64	1111,63	243483,28			
Q03	std	53,28	51,78	131,21	0,00	-11,77	0,00
	tmp	68,34	63,94	196,00			
Q04	std	26,92	26,27	30,05	0,01	-5,80	0,00
	tmp	29,85	29,22	20,88			
Q05	std	121,09	117,76	139,12	0,00	-12,42	0,00
	tmp	141,80	136,98	417,05			
Q06	std	115,81	108,99	337,39	0,00	-12,11	0,00
	tmp	142,49	135,47	633,08			
Q07	std	106,79	104,77	36,94	0,00	-15,63	0,00
	tmp	115,37	113,58	23,26			
Q08	std	83,89	83,01	17,16	0,00	-79,53	0,00
	tmp	120,34	118,49	24,84			
Q09	std	190,22	188,41	16,40	0,00	-35,22	0,00
	tmp	212,35	209,95	62,59			
Q13	std	87,21	84,74	30,25	0,00	-14,05	0,00
	tmp	93,79	92,31	13,60			
S01	std	106,21	104,65	11,84	0,00	-12,18	0,00
	tmp	111,65	109,45	28,08			
S02	std	72,40	70,86	54,72	0,03	-51,65	0,00
	tmp	108,26	106,41	41,74			
S03	std	15,89	14,95	5,55	0,00	-148,82	0,00
	tmp	55,29	54,25	8,47			

Table B.2: Benchmark measures: average and median query execution times in ms on data of simulation A; Results of F-test and unequal variance T-test (Welch-Test)

### B.3 Execution times querying data of simulation B

QID	db	exec <sub>avg</sub>	exec <sub>med</sub>	variance	$P_{(F \leq f)}$	t stat	$P_{(T \leq t)}$
Q01	std	348,01	339,47	621,12	0,00	0,53	0,60
	tmp	345,76	332,54	1201,44			
Q02	std	1883,50	1854,30	11107,32	0,00	-108,01	0,00
	tmp	16013,10	15148,41	1700192,08			
Q03	std	165,57	162,29	170,88	0,25	-52,58	0,00
	tmp	266,17	261,37	195,24			
Q04	std	302,35	296,45	261,45	0,00	-13,57	0,00
	tmp	329,50	325,06	139,05			
Q05	std	1393,48	1388,13	776,35	0,00	46,69	0,00
	tmp	1233,34	1231,67	400,17			
Q06	std	1382,00	1376,80	477,08	0,00	26,28	0,00
	tmp	1259,35	1244,51	1701,11			
Q07	std	1378,16	1375,55	186,30	0,00	54,94	0,00
	tmp	1248,30	1243,78	372,29			
Q08	std	1051,70	1048,61	135,50	0,00	-259,70	0,00
	tmp	1587,59	1583,43	290,28			
Q09	std	2602,34	2596,40	1034,89	0,28	65,84	0,00
	tmp	2311,39	2307,14	917,96			
Q13	std	1125,19	1122,19	250,40	0,06	54,68	0,00
	tmp	991,80	988,72	344,67			
S01	std	1374,19	1370,91	218,34	0,00	-97,22	0,00
	tmp	1553,55	1552,15	122,06			
S02	std	956,26	900,52	22266,47	0,00	-35,33	0,00
	tmp	1486,22	1482,68	239,00			
S03	std	66,83	65,66	15,77	0,00	-182,46	0,00
	tmp	205,68	203,69	42,15			

Table B.3: Benchmark measures: average and median query execution times in ms on data of simulation B; Results of F-test and unequal variance T-test (Welch-Test)

## B.4 Execution times querying data of simulation C

QID	db	exec <sub>avg</sub>	exec <sub>med</sub>	variance	$P_{(F \leq f)}$	t stat	$P_{(T \leq t)}$
Q01	std	5014,31	5007,94	5014,31	0,00	10,87	0,00
	tmp	4859,01	4830,46	4859,01			
Q02	std	24314,82	24076,80	24314,82	0,00	-76,64	0,00
	tmp	237478,16	231992,39	237478,16			
Q03	std	598,33	589,29	598,33	0,00	-97,40	0,00
	tmp	1095,51	1090,11	1095,51			
Q04	std	4255,79	4247,94	4255,79	0,00	-48,26	0,00
	tmp	4820,22	4818,21	4820,22			
Q05	std	22104,26	22094,79	22104,26	0,13	-15,20	0,00
	tmp	22318,16	22317,64	22318,16			
Q06	std	22403,62	22219,55	22403,62	0,00	2,40	0,02
	tmp	22291,05	22255,24	22291,05			
Q07	std	22476,84	22484,47	22476,84	0,00	6,25	0,00
	tmp	22228,79	22216,97	22228,79			
Q08	std	16723,87	16709,73	16723,87	0,00	-446,66	0,00
	tmp	22483,98	22486,82	22483,98			
Q09	std	42218,21	42287,68	42218,21	0,00	-203,55	0,00
	tmp	47885,06	47876,19	47885,06			
Q13	std	17427,70	17416,53	17427,70	0,00	-4,96	0,00
	tmp	17491,74	17489,08	17491,74			
S01	std	22045,84	22034,54	22045,84	0,00	-7,18	0,00
	tmp	22225,40	22208,11	22225,40			
S02	std	13416,76	13343,45	13416,76	0,00	-429,88	0,00
	tmp	24283,09	24280,46	24283,09			
S03	std	275,72	269,57	275,72	0,00	-195,56	0,00
	tmp	1422,30	1416,30	1422,30			

Table B.4: Benchmark measures: average and median query execution times in ms on data of simulation C; Results of F-test and unequal variance T-test (Welch-Test)

## B.5 Planning times querying data of simulation A

QID	db	plan <sub>avg</sub>	plan <sub>med</sub>	variance	P <sub>(F&lt;=f)</sub>	t stat	P <sub>(T&lt;=t)</sub>
Q01	std	0,49	0,49	0,49	0,00	5,73	0,00
	tmp	0,40	0,39	0,40			
Q02	std	0,98	0,87	0,98	0,00	17,41	0,00
	tmp	0,40	0,36	0,40			
Q03	std	1,16	1,04	1,16	0,00	9,01	0,00
	tmp	0,83	0,74	0,83			
Q04	std	0,59	0,54	0,59	0,00	-5,00	0,00
	tmp	0,71	0,63	0,71			
Q05	std	0,56	0,50	0,56	0,00	2,74	0,01
	tmp	0,50	0,44	0,50			
Q06	std	0,53	0,49	0,53	0,13	1,21	0,23
	tmp	0,51	0,46	0,51			
Q07	std	0,50	0,48	0,50	0,00	9,07	0,00
	tmp	0,38	0,33	0,38			
Q08	std	0,94	0,81	0,94	0,00	9,82	0,00
	tmp	0,67	0,61	0,67			
Q09	std	0,96	0,78	0,96	0,00	19,09	0,00
	tmp	0,49	0,43	0,49			
Q13	std	0,54	0,45	0,54	0,00	12,18	0,00
	tmp	0,37	0,32	0,37			
S01	std	0,48	0,44	0,48	0,00	18,85	0,00
	tmp	0,27	0,24	0,27			
S02	std	0,69	0,61	0,69	0,00	4,53	0,00
	tmp	0,59	0,53	0,59			
S03	std	0,41	0,37	0,41	0,00	-29,36	0,00
	tmp	0,78	0,71	0,78			

Table B.5: Benchmark measures: average and median query planning times in ms on data of simulation A; Results of F-Test and unequal variance T-test (Welch-Test)

## B.6 Planning times querying data of simulation B

QID	db	plan <sub>avg</sub>	plan <sub>med</sub>	variance	$P_{(F \leq f)}$	t stat	$P_{(T \leq t)}$
Q01	std	0,43	0,39	0,43	0,29	4,79	0,00
	tmp	0,35	0,29	0,35			
Q02	std	1,05	0,89	1,05	0,00	17,10	0,00
	tmp	0,42	0,38	0,42			
Q03	std	1,03	0,92	1,03	0,00	6,31	0,00
	tmp	0,74	0,73	0,74			
Q04	std	0,46	0,38	0,46	0,00	-5,53	0,00
	tmp	0,60	0,59	0,60			
Q05	std	0,47	0,39	0,47	0,01	3,85	0,00
	tmp	0,40	0,39	0,40			
Q06	std	0,50	0,39	0,50	0,14	2,66	0,01
	tmp	0,44	0,37	0,44			
Q07	std	0,50	0,40	0,50	0,00	4,20	0,00
	tmp	0,41	0,37	0,41			
Q08	std	0,98	0,82	0,98	0,00	7,98	0,00
	tmp	0,66	0,57	0,66			
Q09	std	1,20	0,99	1,20	0,00	14,71	0,00
	tmp	0,55	0,48	0,55			
Q13	std	0,52	0,45	0,52	0,44	5,27	0,00
	tmp	0,41	0,32	0,41			
S01	std	0,50	0,39	0,50	0,00	10,51	0,00
	tmp	0,28	0,24	0,28			
S02	std	0,69	0,63	0,69	0,00	0,79	0,43
	tmp	0,67	0,66	0,67			
S03	std	0,46	0,40	0,46	0,00	-12,44	0,00
	tmp	0,81	0,70	0,81			

Table B.6: Benchmark measures: average and median query planning times in ms on data of simulation B; Results of F-Test and unequal variance T-test (Welch-Test)

## B.7 Planning times querying data of simulation C

QID	db	plan <sub>avg</sub>	plan <sub>med</sub>	variance	P <sub>(F&lt;=f)</sub>	t stat	P <sub>(T&lt;=t)</sub>
Q01	std	0,41	0,39	0,41	0,33	3,09	0,00
	tmp	0,35	0,32	0,35			
Q02	std	1,01	0,91	1,01	0,00	2,19	0,03
	tmp	0,59	0,37	0,59			
Q03	std	1,00	0,94	1,00	0,01	3,43	0,00
	tmp	0,80	0,71	0,80			
Q04	std	0,48	0,36	0,48	0,00	-5,28	0,00
	tmp	0,70	0,62	0,70			
Q05	std	0,56	0,41	0,56	0,18	1,64	0,10
	tmp	0,49	0,41	0,49			
Q06	std	0,59	0,53	0,59	0,01	3,81	0,00
	tmp	0,46	0,41	0,46			
Q07	std	0,60	0,51	0,60	0,01	4,17	0,00
	tmp	0,44	0,38	0,44			
Q08	std	0,91	0,73	0,91	0,11	1,63	0,11
	tmp	0,81	0,74	0,81			
Q09	std	1,20	0,90	1,20	0,00	6,76	0,00
	tmp	0,68	0,63	0,68			
Q13	std	0,65	0,60	0,65	0,00	5,59	0,00
	tmp	0,43	0,35	0,43			
S01	std	0,59	0,52	0,59	0,00	7,41	0,00
	tmp	0,36	0,39	0,36			
S02	std	0,61	0,54	0,61	0,00	-3,78	0,00
	tmp	0,77	0,65	0,77			
S03	std	0,68	0,54	0,68	0,10	-4,98	0,00
	tmp	0,97	0,77	0,97			

Table B.7: Benchmark measures: average and median query planning times in ms on data of simulation C; Results of F-Test and unequal variance T-test (Welch-Test)

# List of Figures

2.1	BPM life cycle . . . . .	6
2.2	Overview of the Activiti tool stack . . . . .	12
2.3	Periods in Activiti database . . . . .	15
4.1	Schematic visualization of the application architecture . . . . .	32
4.2	Sample business process in BPMN 2.0 notation . . . . .	43
5.1	Benchmark results: Halstead metrics . . . . .	68
5.2	Benchmark results: Average and median query planning times. . . . .	70
5.3	Development average execution times . . . . .	72
5.4	Benchmark results: Average and median query execution times. . . . .	73





# List of Tables

2.1	Examples for BPMN 2.0 notations . . . . .	11
2.2	Table prefixes in Activiti database . . . . .	14
2.3	History tables in Activiti database. . . . .	14
2.4	Relevant ID tables in Activiti database. . . . .	15
2.5	Example query complexity computation . . . . .	20
3.1	Example for an EXPLAIN ANALYZE command and its output . . . . .	23
4.1	Mandatory config.properties for simulation application . . . . .	36
4.2	Optional config.properties for simulation application . . . . .	37
4.3	Simulation application: Command line arguments . . . . .	38
4.4	Simulation application: Examples for command line calls . . . . .	38
4.5	Config.properties for credit approval process . . . . .	46
4.6	Mandatory config.properties for benchmark application . . . . .	49
4.7	Benchmark application: Command line arguments . . . . .	50
4.8	Benchmark application: Examples for command line calls . . . . .	50
4.9	Columns required in SQL queries file . . . . .	53
4.10	Columns exported to file results_aggregated . . . . .	54
4.11	Columns exported to file results_detailed . . . . .	54
4.12	Overview of queries used in the benchmark . . . . .	59
5.1	Figures of databases used in benchmark . . . . .	66
B.1	Query Halstead metrics . . . . .	96
B.2	Query execution benchmark measures - simulation A . . . . .	97
B.3	Query execution benchmark measures - simulation B . . . . .	98
B.4	Query execution benchmark measures - simulation C . . . . .	99
B.5	Query planning benchmark measures - simulation A . . . . .	100
B.6	Query planning benchmark measures - simulation B . . . . .	101
B.7	Query planning benchmark measures - simulation C . . . . .	102



# Listings

4.1	Calculation of task time . . . . .	47
4.2	Identification of maximum set of period borders . . . . .	57
4.3	Identification of periods . . . . .	57
A.1	Query Q01 Count of open processes o.t. - standard db . . . . .	80
A.2	Query Q01 Count of open processes o.t. - standard db (alternative) .	80
A.3	Query Q01 Count of open processes o.t. - temporal db . . . . .	80
A.4	Query Q02 Count of active processes o.t. - standard db . . . . .	81
A.5	Query Q02 Count of active processes o.t. - temporal db . . . . .	81
A.6	Query Q02 Count of active processes o.t. - standard db (alternative) .	82
A.7	Query Q03 Non-active (idle) periods per process - standard db . . . .	83
A.8	Query Q03 Non-active (idle) periods per process - temporal db . . . .	83
A.9	Query Q04 Periods with no open processes - standard db . . . . .	84
A.10	Query Q04 Periods with no open processes - temporal db . . . . .	84
A.11	Query Q05 Count of assigned tasks to user o.t. - standard db . . . . .	85
A.12	Query Q05 Count of assigned tasks to user o.t. - temporal db . . . . .	85
A.13	Query Q06 Count of claimed user tasks o.t. - standard db . . . . .	86
A.14	Query Q06 Count of claimed user tasks o.t. - temporal db . . . . .	86
A.15	Query Q07 Count of not-yet claimed tasks per user - standard db . . .	87
A.16	Query Q07 Count of not-yet claimed tasks per user - temporal db . . .	87
A.17	Query Q08 Periods with no assigned tasks to user - standard db . . .	88
A.18	Query Q08 Periods with no assigned tasks to user - temporal db . . .	88
A.19	Query Q09 Count of assigned tasks to department - standard db . . .	89
A.20	Query Q09 Count of assigned tasks to department - temporal db . . .	89
A.21	Query Q13 Count of assigned tasks to service - standard db . . . . .	90
A.22	Query Q13 Count of assigned tasks to service - temporal db . . . . .	90
A.23	Query S01 Distinct periods with claimed tasks per user - standard db .	91
A.24	Query S01 Distinct periods with claimed tasks per user - temporal db .	91
A.25	Query S02 - Parallel processing of specific user tasks - standard db . .	92
A.26	Query S02 - Parallel processing of specific user tasks - temporal db . .	92
A.27	Query S03 - Parallel processing of activities - standard db . . . . .	93
A.28	Query S03 - Parallel processing of activities - temporal db . . . . .	93



# Acronyms

**API** Application Programming Interface. 12, 13, 25

**BPM** Business Process Management. xiii, 3, 5–9, 16, 19, 21, 22, 24–26, 55, 56, 67, 72, 74–76

**BPMI** Business Process Management Initiative. 9

**BPMN** Business Process Model and Notation. 3, 5, 9–13, 21, 26, 28, 31, 32, 35–37, 40, 41, 43, 47, 75

**CSV** comma-separated values. 27–29, 31–33, 36, 37, 41, 42, 48–54, 58

**CTE** Common Table Expression. 19, 60, 71, 75

**DBMS** Database Management System. xiii, 2, 7, 16–18, 21, 23–25, 27, 43, 49, 53, 55, 56, 58, 65–67, 71, 74–77

**ERP** Enterprise Resource Planning. 7

**GUI** graphical user interface. 13

**IASB** International Accounting Standards Board. 1

**IDE** integrated development environment. 12

**IFRS** International Financial Reporting Standard. 1

**KPI** Key Performance Indicator. 7, 8

**OMG** Object Management Group. 9

**POJO** Plain Old Java Object. 52

**REST** REpresentational State Transfer. 12

**SQL** Structured Query Language. xi, xiii, 5, 16, 18, 31, 33, 39, 48, 50–53, 58, 75

**UML** Unified Modeling Language. 9

**WfMC** Workflow Management Coalition. 10

**WfMS** Workflow Management System. 1, 2, 5, 7, 8, 10, 12, 25, 55, 60, 76, 77

**WSFL** Web Service Flow Language. 9

**XML** Extensible Markup Language. 12, 36

# Bibliography

- [AS17] Inc. Alfresco Software. *Activiti User Guide v6.0.0*, May 2017. <https://www.activiti.org/userguide>.
- [BB13] K. Baïna and S. Baïna. User experience-based evaluation of open source workflow systems: The cases of bonita, activiti, jbpmp, and intalio. In *2013 3rd International Symposium ISKO-Maghreb*, pages 1–8, Nov 2013.
- [BDGJ18a] Michael H. Böhlen, Anton Dignös, Johann Gamper, and Christian S. Jensen. Database Technology for Processing Temporal Data (Invited Paper). In Natasha Alechina, Kjetil Nørnvåg, and Wojciech Penczek, editors, *25th International Symposium on Temporal Representation and Reasoning (TIME 2018)*, volume 120 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:7, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.
- [BDGJ18b] Michael H. Böhlen, Anton Dignös, Johann Gamper, and Christian S. Jensen. Temporal data management – an overview. In Esteban Zimányi, editor, *Business Intelligence and Big Data*, pages 51–83, Cham, 2018. Springer International Publishing.
- [BFLR03] Paul L Bowen, Colin B Ferguson, Timothy H Lehmann, and Fiona H Rohde. Cognitive style factors affecting database query performance. *International Journal of Accounting Information Systems*, 4(4):251–273, 2003. Third International Research Symposium on Accounting Information Systems.
- [BGJ06] Michael H. Böhlen, Johann Gamper, and Christian S. Jensen. Temporal databases. In *In 0-8493-8597-0/01/0.00+1.50 c 2006 by CRC Press, LLC 1 CHAPTER 59.*, pages 59–1 – 59–39, 2006.
- [BJ09] Michael H Böhlen and Christian S Jensen. Sequenced semantics. In Link Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 2619–2621. Springer, 2009.
- [BW10] Raymond P.L. Buse and Westley R. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, July 2010.

- [BWJ02] Claudio Bettini, X. Sean Wang, and Sushil Jajodia. Temporal reasoning in workflow systems. *Distributed and Parallel Databases*, 11(3):269–306, May 2002.
- [CP03] C Combi and G Pozzi. Towards temporal information in workflow systems. In Olive, A and Yoshikawa, M and Yu, ESK and Genero, M and Grandi, F and VandenHeuvel, WJ and Krogstie, J and Lyytinen, K and Mayr, HC and Nelson, J and Piattini, M and Poels, G and Roddick, J and Siau, K, editor, *Advanced Conceptual Modeling Techniques*, volume 2784 of *Lecture notes in computer science*, pages 13–25. Springer-Verlag Berlin, 2003. 21st International Conference on Conceptual Modeling, Tampere, Finland, Oct 07-11, 2002.
- [CP04] Carlo Combi and Giuseppe Pozzi. Architectures for a temporal workflow management system. In *Proceedings of the 2004 ACM symposium on applied computing*, SAC '04, pages 659–666, March 2004.
- [CP09] Carlo Combi and Giuseppe Pozzi. Temporalities for workflow management systems. In *Handbook of Research on Business Process Modeling*, 2009.
- [DBG12] Anton Dignös, Michael H. Böhlen, and Johann Gamper. Temporal alignment. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 433–444, New York, NY, USA, 2012. ACM.
- [DBG13] A. Dignös, M. Böhlen, and J. Gamper. Query time scaling of attribute values in interval timestamped databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1304–1307, April 2013.
- [DBGJ16] Anton Dignös, Michael H. Böhlen, Johann Gamper, and Christian S. Jensen. Extending the kernel of a relational dbms with comprehensive support for sequenced temporal queries. *ACM Trans. Database Syst.*, 41(4):26:1–26:46, November 2016.
- [DES89] Thomas Davenport and James E. Short. The new industrial engineering: Information technology and business process redesign. *Sloan management Review*, 31, November 1989.
- [Dig18] Anton Dignös. Installation Guide for TPG. <http://tpg.inf.unibz.it/>, August 2018. Part of TPG source code package.
- [DRMR13] Marlon Dumas, Marcello La Rosa, Jan Mendling, and Hajo A. Reijers. *Fundamentals of Business Process Management, First Edition*. Springer, 2013.
- [Gro19] The PostgreSQL Global Development Group. *PostgreSQL 9.6.13 Documentation*, 2019. <https://www.postgresql.org/docs/9.6/index.html>.



- [Hal77] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*, volume 7. Elsevier Science Inc., New York, NY, USA, 1977.
- [Int14] International Accounting Standards Board. IFRS 9 Financial Instruments (replacement of IAS 39). <https://www.ifrs.org/issued-standards/list-of-standards/ifrs-9-financial-instruments/>, July 2014. Draft.
- [ISO11] ISO. *ISO/IEC 9075-1:2011 Information technology — Database languages — SQL*. December 2011.
- [JG18] Christian S. Jensen Johann Gamper, Michael H. Böhlen. Temporal aggregation. In Ling Liu and M. Tamer Zsu, editors, *Encyclopedia of Database Systems*, pages 3899–3909. Springer Publishing Company, Incorporated, 2nd edition, 2018.
- [J606] Tick József. Workflow model representation concepts. In *International Symposium of Hungarian Researchers on Computational Intelligence*, number 7, pages 329–337, January 2006.
- [KLWL09] Ryan K.L Ko, Stephen S.G Lee, and Eng Wah Lee. Business process management (bpm) standards: a survey. *Business Process Management Journal*, 15(5):744–791, September 2009.
- [KM12] Krishna Kulkarni and Jan-Eike Michels. Temporal features in SQL:2011. *SIGMOD Record*, 41(3):34–43, October 2012.
- [KRM09] Klaus D Kubinger, Dieter Rasch, and Karl Moder. Zur Legende der Voraussetzungen des t-Tests für unabhängige Stichproben. *Psychologische Rundschau*, 60(1):26–27, 2009.
- [LYSY11] Hai Liu, Xiaoping Ye, Ming Shi, and Boling Yang. Temporal indexes supporting valid time. In Yong Tang, Xiaoping Ye, and Na Tang, editors, *Temporal Information Processing Technology and Its Application*, pages 151–174. Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg, 2011.
- [McC76] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976.
- [Moc19] LLC Mockaroo. Mock-up data generator. <https://mockaroo.com/>, August 2019.
- [Mos16] Peter Moser. *Temporal PostgreSQL Manual and Report*. Free University of Bolzano/Bozen, July 2016.
- [Obj11] Object Management Group. Business Process Model and Notation (BPMN), Version 2.0. <http://www.omg.org/spec/BPMN/2.0>, January 2011.

- [OW10] A. Oram and G. Wilson. *Making Software: What Really Works, and Why We Believe It*. O'Reilly Media, November 2010.
- [Rad12] Tijs Rademakers. *Activiti in Action: Executable Business Processes in BPMN 2.0*. Manning Publications Company, 2012.
- [RKM11] Dieter Rasch, Klaus D Kubinger, and Karl Moder. The two-sample t test: pre-testing its assumptions does not pay off. *Statistical papers*, 52(1):219–231, 2011.
- [Sha10] Robert Shapiro. Update on BPMN release 2.0. <https://www.slideshare.net/Aamir97/folien>, February 2010. visited on 18.09.2019.
- [Sno95] Richard Thomas Snodgrass. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, Norwell, MA, USA, 1995.
- [Sno00] Richard T. Snodgrass. *Developing Time-oriented Database Applications in SQL*. Data Management Systems Series. Morgan Kaufmann Publishers, 2000.
- [TPLZ11] Yong Tang, Zewu Peng, Dongning Liu, and Wenshen Zhang. From time data to temporal information. In Yong Tang, Xiaoping Ye, and Na Tang, editors, *Temporal Information Processing Technology and Its Application*, pages 3–19. Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg, 2011.
- [Vai13] A. Vaisman. An introduction to business process modeling. *Lecture Notes in Business Information Processing*, 138:29–61, 2013.
- [Ver19] Verifysoft. Measurement of halstead metrics. [https://www.verifysoft.com/en\\_halstead\\_metrics.html](https://www.verifysoft.com/en_halstead_metrics.html), August 2019.
- [WB11] Stephen White and Conrad E Bock. New capabilities for process and interaction modeling in bpmn 2. Technical report, National Institute of Standards and Technology, US Department of Commerce, 2011.
- [Wes12] Mathias Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer, Berlin, 2 edition, 2012.
- [Wor99] Workflow Management Coalition. Terminology and glossary. Document Number WFMC-TC-1011 3.0. [http://www.wfmc.org/docs/TC-1011\\_term\\_glossary\\_v3.pdf](http://www.wfmc.org/docs/TC-1011_term_glossary_v3.pdf), February 1999.
- [Zan08] Carlo Zaniolo. Time versus standards: A tale of temporal databases. In Il-Yeol Song, Mario Piattini, Yi-Ping Phoebe Chen, Sven Hartmann, Fabio Grandi, Juan Trujillo, Andreas L. Opdahl, Fernando Ferri, Patrizia Grifoni, Maria Chiara Caschera, Colette Rolland, Carson Woo, Camille Salinesi, Esteban Zimányi, Christophe Claramunt, Flavius Frasincar, Geert-Jan Houben,

and Philippe Thiran, editors, *Advances in Conceptual Modeling – Challenges and Opportunities*, pages 67–67, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.