Grip on software: understanding development progress of SCRUM sprints and backlogs

Helwerda, L.S.

Chapter 3

Database construction

GROS DB: Designing and deploying a database model using extensible and flexible query templates as part of a research-oriented data pipeline architecture

Portions of this chapter are also published in the following article:

- Leon Helwerda et al. “Query compilation for feature extraction in MonetDB”, 2024. Pending submission.

Abstract of Chapter 3

Introduction: Software development teams use several systems that form a development ecosystem to keep track of data related to their projects. We use a data acquisition pipeline to collect records from these systems for research into patterns and outcomes of the process. We store this information in a consolidated MonetDB database with column-based storage. This setup allows us to perform extensible queries.

Research questions: How can we reliably collect data regarding SCRUM software development practices and consolidate the resulting artifacts inside a central database that constantly grows and allows adaptable queries?

Modeling: We define a data model which contains discrete portions corresponding to systems that provide the data, such as issue trackers, version control systems and quality control dashboards. The model consists of entities and relationships that describe states, events and dependencies. We enhance the model with links between different portions, which are not available in the separate systems.

Architecture: We enhance the data selection through the introduction of a query template compiler. Data from different systems and organizations is properly selected and combined into a single data set by adjusting the query parameters for the appropriate context on a high level. We implement more functionality in the database component to handle backups and exports to a central instance of the pipeline.

Experiments: Our initial selection of MonetDB is put to the test. We establish a representative workload of six query templates, each with a refined version and an older version without the changes. We measure the performance of the MonetDB database by comparing two scenarios, one where the queries have been seen by the database several times and one where the database has not created auxiliary internal structures based on the queries. Each version of each query is run in both scenarios ten times. The results show that both our refinement of the query templates and the optimized scenario for MonetDB improve the run times of the queries, justifying the selection for MonetDB and making frequent analysis of the data set during SCRUM sprints feasible.

3.1 Introduction

In order to consistently, reproducibly and frequently perform analytical studies regarding patterns and outcomes of SCRUM software development processes, we design a consolidated location for the data acquired from several systems used by multiple software development projects. As part of the pipeline that we introduced in Chapter 2, we identify a need for a highly-available storage location that allows for a steady stream of additions in order to reflect the situation as experienced by the development teams. Furthermore, the structures and events described by the data should be available for data analysis using various attributes in order to output different reports for involved people.

There is a necessity for a model which describes the interactions between the different artifacts during SCRUM sprints, including product backlogs, code repositories, quality reports and release builds. Often, developers are able to navigate between these items in their current state, but they are hindered by design flaws of the systems when tracking down what they looked like at a specific date, during a sprint from months ago. It is complicated to understand what a metric refers to when it is stored at a separate system without a proper reference point. Thus, our data storage for Grip on Software should have a temporal aspect, where time is a first-class data type with many available functions. It should further allow filters or aggregate operations to determine which factors define a sprint most pertinently.

We consider existing data stores which already implement features that we desire. A *relational database management system* (RDBMS) offers a table-based storage with rows and columns, where some entries refer to other columns in another table. In addition, the RDBMS provides operations in order to add, remove, adjust and search within the relations, allowing manipulation as well as selection. These operations support combining tables in a *query*, with additional arithmetic or programming operators to alter which data points are added, adjusted or retrieved.

This sort of system enables much flexibility towards modeling complex relationships of data structures that reflect dynamics of software development ecosystems. Both the mapping from artifacts to table-based storage as well as the operations involved in manipulating the data are part of the *modeling problem*, which differs for each subject area.

Another side of using a database system is the management and architecture of technical components in the pipeline. This technical aspect largely consists of ensuring that the database has enough hardware resources in order to perform its operations, including establishing connectivity with other components. This further enables exports, backups, migrations and upgrades to take place. These maintenance tasks should not affect the typical performance of the pipeline. The database needs to remain consistent and resilient, with no loss of data due to downtime, for example.

When building a database that should function at more than one organization, there are additional concepts surrounding the *architecture problem*. We consider the integration of the database component in the pipeline and the overall development ecosystem that exists at an organization, the centralization of data of multiple organizations using a separate research pipeline, the administration of privacy-sensitive data and the optimization of queries performed on the database. Meanwhile, the solution needs to support a complex data model based on multiple specialized development platforms. It is important that the solutions for these considerations do not affect the model's restrictions on the outcome of the query operations, among other things.

We summarize the pertinent points of a database system that we find useful for our purposes by proposing a number of objectives for the usefulness of the technical database component:

- We should be able to design a single data model that addresses several complex SCRUM ecosystems, allowing us to easily recognize the original entities and relationships, but also introduce new links between data from previously separate systems.
- The query language allows us to utilize the data model to retrieve relevant data, spread across tables for entities and relationships, with many existing functions for grouping and aggregation as well as opportunities for extending the queries, for example through user-defined functions (UDFs).
- The RDBMS should be efficient enough to perform under high load of parallel updates from independent data acquisition agents, while allowing frequent analysis with complex queries.
- The database component should function as a part of a larger pipeline which operates in different environments, without hindering the existing use of the software development ecosystem.
- Typical guarantees by database systems, such as atomicity of transactions and resilience to crashes, are considered a prerequisite to an operational database, along with management interfaces.

These desirable qualities of a database bring us back to the questions regarding the need and usability of such a system in the overall Grip on Software pipeline, aimed at collecting and understanding metrics regarding SCRUM software development processes. We therefore consider the addition of the following two sub-questions to the research question, mentioned before in Section 2.1, regarding the data acquisition and consolidation pipeline:

RQ1 How can we reliably collect data regarding SCRUM software development practices and consolidate the resulting artifacts inside a central database that constantly grows and allows adaptable queries?

RQ1c How can we model the relationships between different data artifacts acquired from dynamic systems regarding SCRUM software development in order to properly deduce information about their state during different sprints?

RQ1d Which technical challenges are relevant when deploying a database component as part of a pipeline for multiple organizational ecosystems?

The remainder of this chapter focuses on this two-sided problem of database design and architecture. First, we consider existing modeling structures and management systems in Section 3.2. In Section 3.3, we introduce the Grip on Software database (GROS DB), including the data model in Section 3.3.1. The entities and relationships are shown step-by-step, based on subdivisions of the model. The relations between different originating systems are considered in Section 3.3.2. For the architecture problem, we describe how we administer, optimize, engineer and adjust the behavior of the database system in Section 3.4, including a novel query language overloading extension. Section 3.5 presents some experiments regarding optimization and performance of resource usage. Finally, we discuss the entirety of the Grip on Software database as a component of the data acquisition pipeline in Section 3.6.

3.2 Relevant work

Databases are used in a variety of applications and come in many forms. In the scientific field, an RDBMS is often used to persistently store data using a known model. This way, further analysis profits from efficient lookups of the data.

An area of interest within database architectures is the use of column-based storage. This technique helps with efficiently retrieving attributes of a large number of entities stored in tables. Such queries occur regularly when performing analytical research with large amounts of data.

One column-based database store of interest is MonetDB [38], which is used in various research contexts. We find that there are applications of MonetDB for machine learning [40] and statistical analysis [41]. Novel functionalities to database stores allow streaming updates for processing live data, such as in software quality analytics [42].

The architecture underlying MonetDB integrates with programming languages in order to create UDFs [43]. Support for languages such as Python and R has led to seamless integration with more software packages used in data science [44], such as reuse of existing columnar vector formats in libraries like TensorFlow [45].

Some methods for improving the efficiency of in-database operations avoid copying data between memory locations. Such zero-copy integration has shown to work well for statistical problems [46]. Extensions with other systems to provide better query filters using multi-dimensional indexes also exhibit performance improvement [47].

With all these improvements on integration, extensibility and memory usage, it is often another challenge to provide experimental results on database performance. It is relevant to test similar, reproducible configurations. Further, there should be additional tests with baseline situations where the database system has not had the chance use caches and to process the type of query workload [48]. For specific applications, providing a reasonable representation of the query workload serves as a relevant, localized benchmark [49].

We also look into the use of database systems in the context of analysis of SCRUM software development data. We find that there exist models that look into integration of data from multiple sources, such as quality metrics [50]. Similar approaches with transformations of originating data lead to a model which allows generating dashboards for monitoring performance for SCRUM teams [51]. Sometimes, the goal of data-driven software development is reached through the use of an extensive ontology describing the aspects of SCRUM [52].

3.3 Method

We introduce the technical component which handles database storage, imports, exports and management. This component is a continuation of the data acquisition pipeline introduced in Section 2.3. This pipeline is deployed in an existing software development ecosystem at an organization that is involved in the Grip on Software research. An outline of the pipeline is shown in Figure 3.1, with the components of interest for this chapter highlighted with red and blue colors. Multiple instances of this pipeline reside independently at different organizations. Another version of the database plus the remaining machine learning and information visualization components are placed in a central location. Other pipelines send encrypted backups of the database contents—without any readable personal information—to this central database, so that multiple organizations are combined into one data set for further generalized analysis.

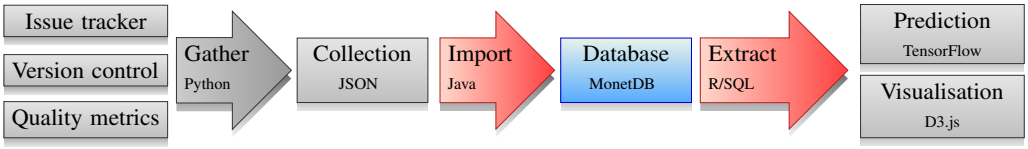

Figure 3.1: High-level overview of the Grip on Software pipeline, with the database system shown as a blue box and relevant technical components as red arrows.

The database uses MonetDB for storage and low-level administration [38]. MonetDB focuses on storage of large-scale analytical information. In contrast to many database management systems that existed at the time of selection, MonetDB is a column-based RDBMS. This means that in this database, all data of a column is stored consecutively, instead of storing row data together. Efficient retrieval of attributes is a main factor in initially selecting MonetDB. Since MonetDB's introduction, other databases such as MariaDB introduced options for column-based storage.

MonetDB is similar to other relational database management systems in that it supports the Structured Query Language (SQL). This query language implements the operations needed for retrieving and manipulating data provided by the database. We consider three main types of invocable queries: (a) data definition statements using `CREATE` as the initial keyword, (b) data manipulation using `INSERT` or `UPDATE`, and (c) data access with a `SELECT` statement. Thus, we first *define* the tables representing entities and relationships, with columns for attributes of different numerical, textual and temporal data types. Then, during the operation of the data pipeline, we *manipulate* the database by filling it with fresh data and altering old data to reflect the most recent situation. We finally *access* the data, selecting subsets for further analysis using combinations of tables through `JOIN` operations and other expressions that filter, order, aggregate, partition and combine data. Such a selection is performed in one go, without having to execute several queries. These features are provided by MonetDB, extending the common SQL language.

The database connection protocols that we use for importing data also support prepared statements in SQL. This allows another system to indicate a query that is to be performed multiple times with different parameters. This reduces the overhead of sending the statement to the database, compiling and optimizing it every single time it is used. For example, we use multiple prepared statements that verify if data did not already exist or needs an update. Then, we respectively insert new rows and perform batched updates for fresh instances of entities.

In addition to query language support, an important feature of MonetDB is integration with other programming languages. The data analysis component of the pipeline connects to the database in order to perform queries. Through UDF integration, a language like R [X] augments the queries with operations specific for our purposes, reducing the need for post-processing. At the moment of implementation, we chose to use R. Since then, MonetDB extended support to Python UDFs as well [53]. Nevertheless, they also continued integration with R [54].

As mentioned in Section 3.2, MonetDB offers more options for extensions as well as in-depth performance optimizations. However, by default, the column-based storage already has compression and dictionary encoding for frequently stored textual values with specified or arbitrary lengths. In this way, the lookup speed and reduction of memory usage are balanced, enabling the modeling of large data sets. The SQL queries themselves are the input of an internal optimizer pipeline, which transforms it to a MonetDB-specific assembly language supported by a relational algebra that works on the internal table structure [55].

3.3.1 Data model

We design a model for GROS DB, a database containing events, artifacts and other relevant elements from SCRUM software development processes. This model is based on the representation of the same entities and relationships as in the systems that we have extracted them from.

An overview of the most relevant entities has been shown during the introduction of the pipeline, in Table 2.1. There, we described from which type of system they originate. This includes issue trackers, version control systems, associated code review, quality control dashboards and build platforms. The aforementioned overview did not mention entities that only exist in relation to another or provide context to a development project, e.g., a software component that an issue applies to. In our survey, we found that some systems contain similar data, such as developers having accounts on multiple services. These systems often refer to each other with an indirect, temporal association: a code commit takes place during a certain sprint.

Software development organizations that use Agile approaches do not always have the same development ecosystem; the same applies to their development teams. As an example, we take the two organizations of our study. At ICTU, there is a frequent use of Jira [I] as an issue tracker, GitLab [III] for version control and code review, SonarQube [V] plus self-made quality dashboards alongside the BigBoat [VII] and Jenkins [IV] build platforms. For Wigo4it, the workflow of refining and implementing stories takes place on TFS/VSTS/Azure DevOps [VIII], with SonarQube for quality control. We summarize the distribution of systems in the two organizations in Table 3.1. Other potential systems in use by such organizations are, e.g., GitHub [XIII] for open source projects or TOPdesk [XV] for internal project asset management.

SYSTEM	ICTU	WIGO4IT
Issue tracker	Jira	Azure DevOps
Version control	Git (some Subversion and TFS)	TFS (Git/TFVC)
Code review	GitLab (most projects)	TFS/VSTS/Azure DevOps
Quality control	Quality time, SonarQube	SonarQube
Build platform	BigBoat, Jenkins	Azure Server
Personnel records	LDAP, Seat counts	Azure DevOps

Table 3.1: Overview of systems in software development ecosystems at two organizations.

In order to model the entities and their interactions in these diverse ecosystems, we first design a rudimentary entity–relationship (ER) diagram. In Figure 3.2, the entities are shown in a fundamental model, excluding attributes, but with most relationships. In order to obtain some "shortcut" relationships, we follow a chain of them, such as an issue belonging to a certain project via the sprint entity. Specific entities, including those based on the issue tracking systems of Jira and Azure DevOps/VSTS/TFS, as well as entities from various types of code repositories, are considered implementations of a class diagram relationship; here, they are condensed into a generic form.

The focus of this section is to model the entities in such a way that we are able to extract a database schema, which defines how the tables are created. One relevant matter is the nature of the relationships between entities. In the diagram, we indicate the type of mapping between entities using *cardinalities* as labels of the relationship's lines. A *one-to-one* relationship, indicated by two 1's on either side, is similar to a bijective function, whereas a *one-to-many* relationship,

where one side is not limited and includes an asterisk * in its cardinality indicator, is like a function with no specific properties, its domain originating from the entity on the side with the 1. A *many-to-many* relationship is best regarded as a multivalued function to either side. Some cardinalities are *optional*, when a side is shown with a $0/1$ or simply an asterisk as indicator. Then, some instances of the entity on an optional side may not participate in a relationship. Table 3.2 provides an overview of the cardinalities by means of examples that are visible in Figure 3.2.

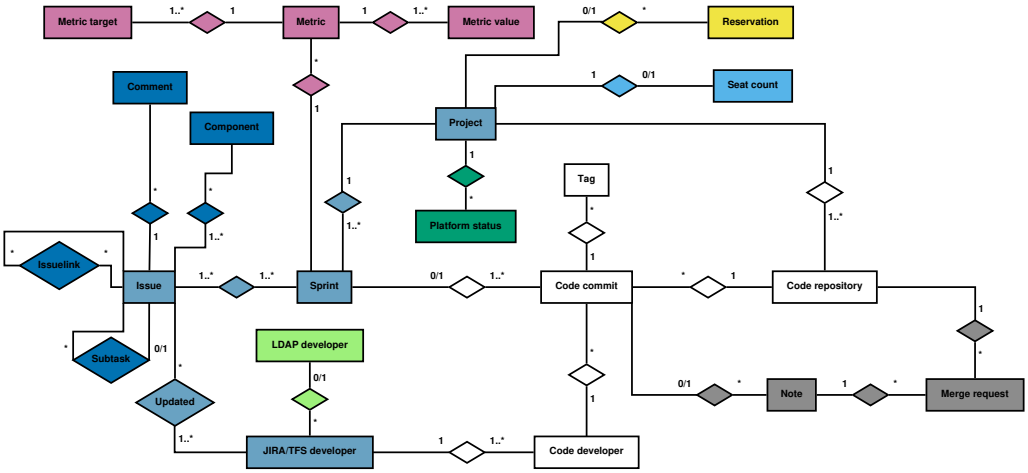

Figure 3.2: An entity-relationship diagram of the main entities and primary relations that are stored in the database. Colors used in this diagram and others indicate the sources of groups of entities: ■ dark blue are retrieved from Jira, ■ grayish blue from either Jira or TFS/VSTS/Azure DevOps (■ light gray in other diagrams), □ white from version control systems, ■ gray from code review systems, ■ pink from quality dashboards, ■ green from build platforms, ■ light green from LDAP, ■ yellow from TOPdesk and ■ light blue from spreadsheets. Not shown here are entities for source tracking and database consistency, colored ■ purple and ■ orange in other diagrams, respectively.

CARDINALITY	EXAMPLE	TARGET	REFERENCE
One-to-one ($1 \leftrightarrow 1$)	Encryption of a project	`project`	■ `project_salt`
Optional one-to-one ($1 \leftrightarrow 0/1$)	Seat count of a project	`project`	■ `seats`
One-to-many ($1 \leftrightarrow 1..^\star$)	Sprints during a project	`project`	■ `sprint`
Optional one-to-many ($1 \leftrightarrow ^\star$)	Comments on an issue	`sprint`	■ `comment`
One-to-many optional ($0/1 \leftrightarrow 1..^\star$)	Commits during a sprint	`sprint`	□ `commits`
One-to-many all-optional ($0/1 \leftrightarrow ^\star$)	Reservations of projects	`project`	■ `reservation`
Many-to-many ($1..^\star \leftrightarrow 1..^\star$)	Issues in sprints	`sprint`	■ `issue`
Many-to-many optional ($1..^\star \leftrightarrow ^\star$)	Issues updated by users	`developer`	■ `issue`
Many-to-many all-optional ($^\star \leftrightarrow ^\star$)	Issues linked to issues	`issue` (2×)	■ `issuelink`

Table 3.2: Cardinalities of relationships that are noticeable in the ER diagrams of the data model. For each cardinality, an example is shown with the entity table that is the target of the reference and the table where we store the reference attribute.

As a typical example, we introduce two additional data sources—meant to validate data from our primary sources—for meeting reservations and numbers of full-time equivalents (FTEs), including seat counts. These entities show two different cardinalities in their relationship: a project has any number of reservations—and a meeting reservation is related to at most one project—while a singular seat count is related to a unique project, if known. These are respectively a one-to-many all-optional and an optional one-to-one relationship. A many-to-many all-optional relationship is found in issues possibly linking to each other; here, domain and codomain are equal, self-links excluded.

The complex issue entity has other many-to-many relationships: the issue could be worked on during different sprints, by various developers and for certain software components. Entities having multiple versions exhibit such complicated interactions. This is also dependent on the level of detail that the originating system provides in the earlier versions of the entity. Similarly, attributes for an issue from Jira differ from work items in TFS/VSTS/Azure DevOps.

All these entities with attributes and relationships have a specific role. Despite the complexity, it is important to model the GROS DB in such a way that it reflects reality, describing the situation as it is in other systems as genuinely as possible. If, early on, we would discard attributes and elaborate relationships, then the analysis that we perform in later stages is more cumbersome. We find it easier to leave out sensitive data—such as personal information and project names—during an export, so shrinking to the necessary fields afterward is a better trade-off.

In Figure 3.3, the entities with all attributes and relationships are shown[¶]. The colors of tables correspond to the grouping in the smaller ER diagram. Some tables now have specialized implementations for their source systems. Attributes in this diagram have key indicators for *primary keys* that uniquely indicate the entity. Yellow icons are local while red icons indicate a *reference* to another table. Non-key attributes have diamond icons, with only an outline if the value is optional. A line indicates a relationship; the line is dashed if no primary key is involved in the references between the two tables. The origin of the relationship—where the reference attribute is stored—is indicated with a triangle at the endpoint. The endpoint is a circle if the reference is a temporal data type.

We introduce relationships which involve multiple attributes in order to properly reference the tables. When primary keys are concerned, the relationship is usually one-to-one or one-to-many. In this visualization, we intentionally left out some lines, with only the endpoints remaining in the diagram. This either means that this relationship is a shortcut or that multiple attributes refer to the same table. This reduces the complexity of the representation, which would otherwise have many more crossing lines. The full diagram is obtainable from the database administration pipeline component [d] using MySQL Workbench [XVI], allowing more inspection and reuse in other database systems.

These additional references make it easier for us to design queries that combine data from multiple entities, while preserving database normalization at the necessary level to fulfill the details of the entities. For example, we could assume that a story is always worked on in the project that a sprint belongs to. However, there may be situational differences in which the story was moved between projects. This discrepancy could not be modeled without tracking the project in each version of the issue. Thus, we track this data through more direct references.

We address specific parts of the larger, encompassing diagram in more detail in the following subsections. In particular, we examine the entities and relationships from Jira, version control systems with their associated code review, TFS/VSTS/Azure DevOps and quality control systems.

[¶]The diagram is also available at `https://gros.liacs.nl/database-model.pdf`

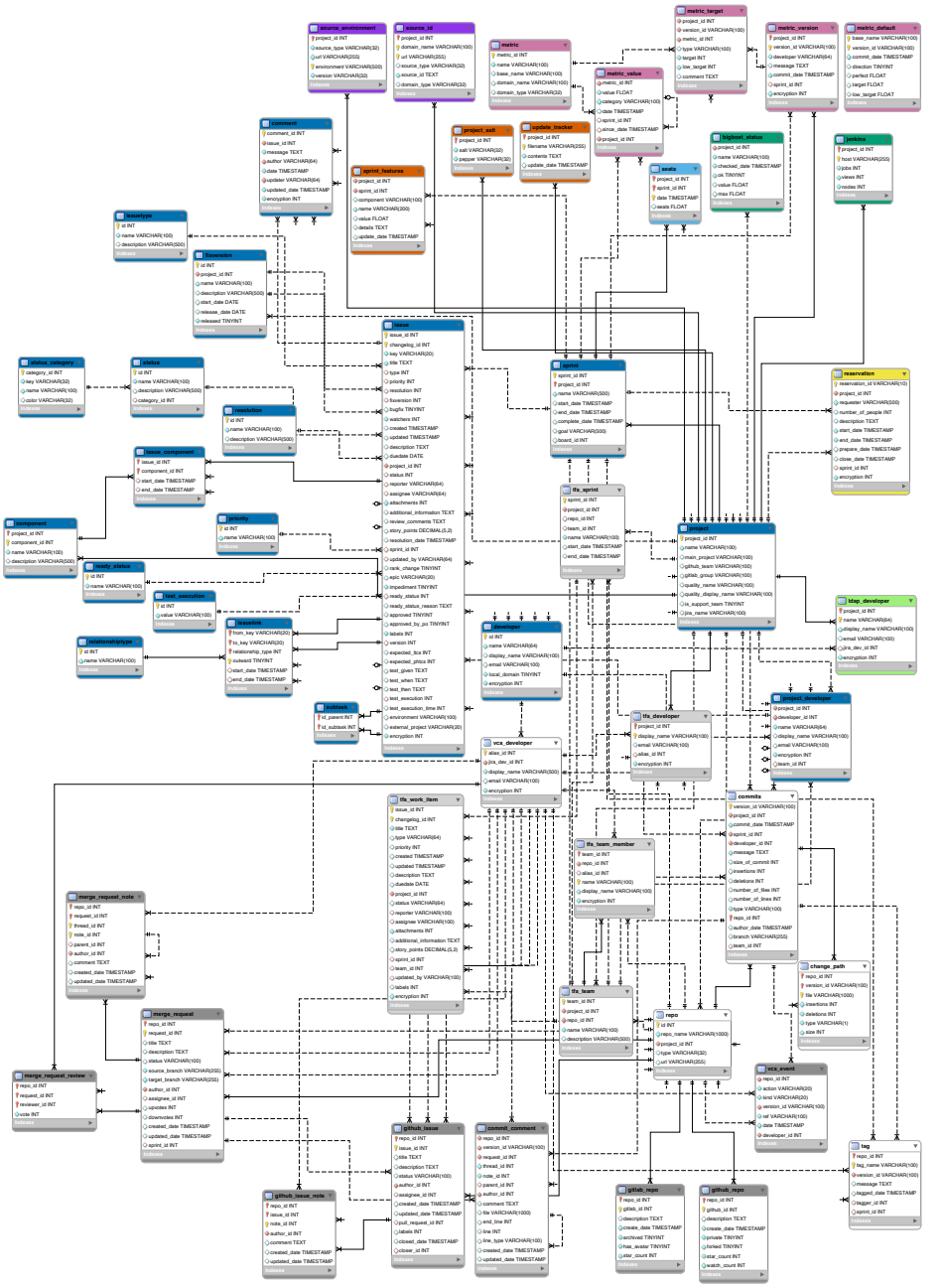

Figure 3.3: Complete ER diagram of the entities and relationships, based on the table schema, depicted as UML classes.

Many software projects arrange their development process using these systems. Some of the smaller portions of the model are not mentioned in full detail, while additional tables for source tracking and internal up-to-dateness checks are out of scope here. Connections between data retrieved from separate data sources are described in Section 3.3.2. Some entities include sensitive data fields that are considered for pseudonymization or excluded from exports. We discuss how we encrypt these specific attributes in Section 3.4.

Jira

The core of our database model consists of entities and relationships extracted from the Jira issue tracker. This system is used by most of the SCRUM software development projects that participate in our research, with the primary goal of tracking backlogs of stories and other types of tasks—commonly known as issues—during sprints. All projects at ICTU use Jira, while Wigo4it instead uses the work item tracking provided by TFS/VSTS/Azure DevOps. Our design approach initially focused on the Jira system, but expansions to the model allowed us to store data acquired from other systems as well. Figure 3.4 displays the central portion of the model that describes the data from Jira, including most of the references between the involved entities.

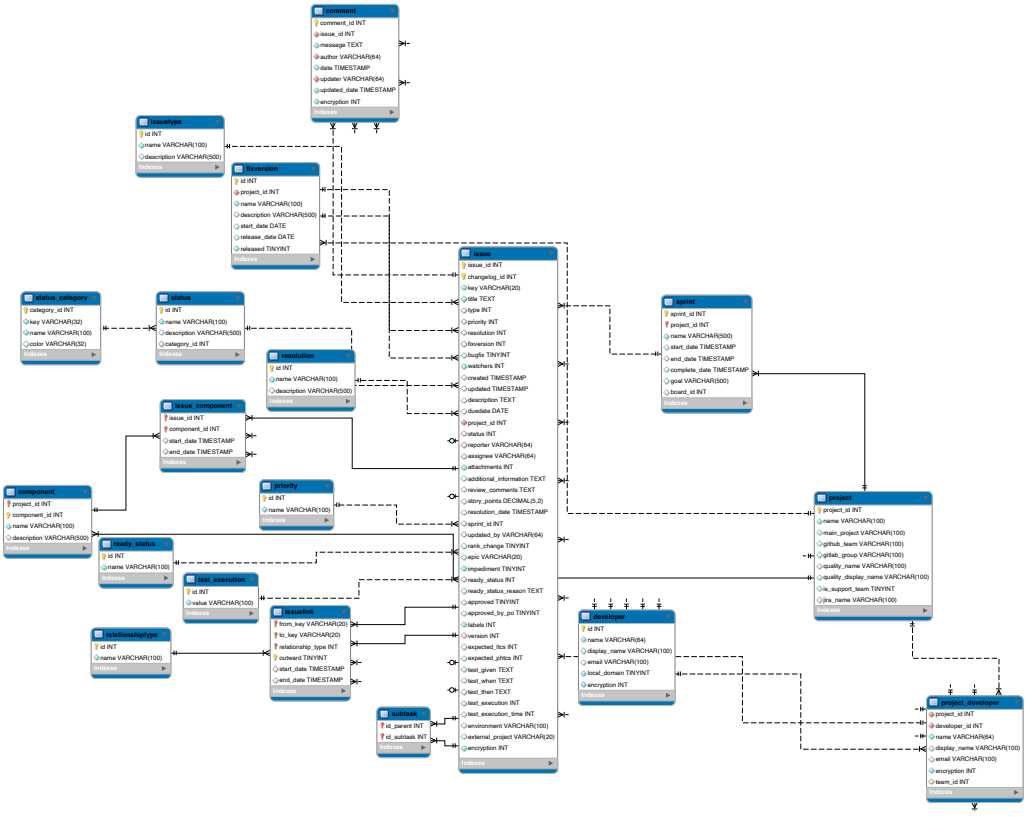

Figure 3.4: Entities and relationships that model data retrieved from Jira.

Jira is the prime data provider for a common entity that other portions of the model heavily rely on: the *project*. The main attributes describe how each project is named in the issue tracker, such as the prefix used by the issue numbering system and the human-readable name. The table also tracks dependent relationships to a parent project, although this is formally a weak reference, since the parent might even be missing from the data set. Through external means, we have more advanced methods of combining projects worked on by the same development team. The project entity is augmented to include attributes deduced from other sources, for more ways to refer to the project or its main assets.

The second pivotal entity is the *sprint*, which tracks important events. The start and end dates are predefined by the Product Owner (PO) when determining which stories to work on. Meanwhile, the completion date of a sprint is deduced from the moment that the team finishes the last of its stories, possibly before or after the planned end date. Other attributes are the description of the sprint goal (SG) and an identifier to the board that we use to construct a URL to the sprint at the issue tracker.

Jira is also a source for determining which *developer* works on which project. Personal data from the user profile, such as name and email address, is stored and encrypted in this table. A second table acts as a relationship between developer and project, using a project-specific encryption key for the duplicated attributes with personal information. This allows later imports and reports to cross-reference the existing data, as long as the encryption keys are available. Section 3.3.2 describes the encryption and linking of personal data in more detail.

The largest entity of the model is the *issue*. Each version of an issue is stored, containing many attributes where a few have been altered compared to the previous version, such as the description or the estimated story points. Some of these are complex, meaning that additional attributes about each of them are stored in a separate table. This includes what type of issue it is, what software component it is for, priority-related attributes, which release version it is or will be fixed in, the status it is in and—if it is done—what resolution it was given. An organizational instance of Jira is often extended with custom fields. The attributes currently modeled are sometimes specific to ICTU, but configuration allows treating the fields differently for another organization. The data acquisition component [a] of the Grip on Software pipeline provides a mapping from custom fields to the recognized attributes. Beyond that, we acquire the comments made on issues, with metadata on developer and time. Finally, developers link issues to others through different types of relationships, with one specialized relation for subtasks. Some of the complex attributes and relationships track their own metadata and chronological information.

Version control

Software development uses generic systems like Git [II] and Subversion [XIV] for version control. Despite differences in how these two systems track changes to code, they are conceptually similar enough to facilitate an abstraction in our data model. In Figure 3.5, we show the entities and relationships that make up this portion of the model.

A version control system provides development teams with the possibility to store and alter code collaboratively. Code related to the same software component is placed in a *repository*, sometimes abbreviated as repo. This entity is central in our modeling of this portion of the database schema. A URL attribute helps with tracking where the code is hosted.

In the context of a repository, a change to the code is considered a *commit*, with increments of commits leading to the most recent version of the code upon a branch. Usually, there is one

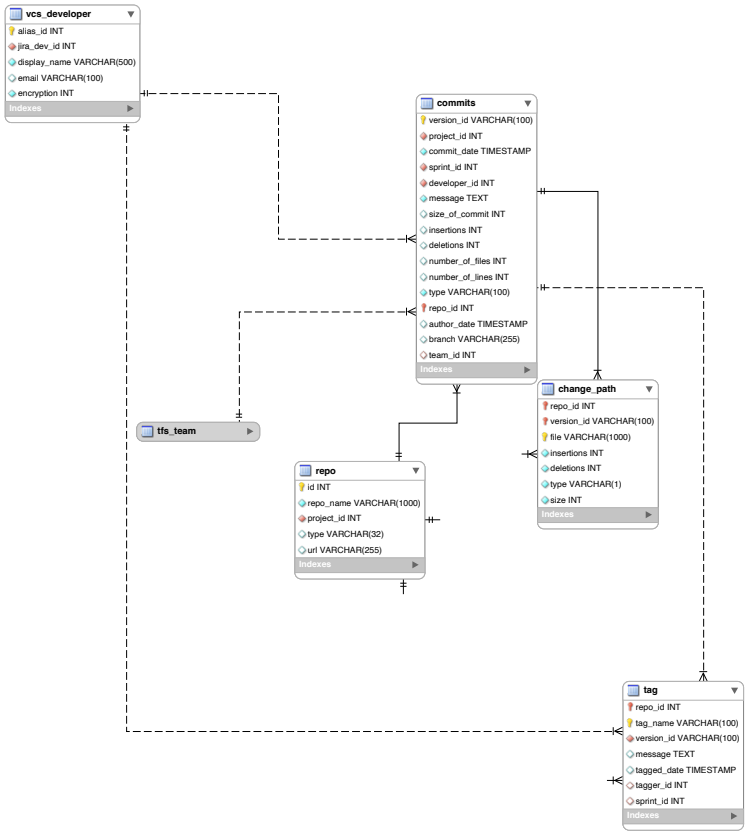

Figure 3.5: Entities and relationships that model data retrieved from version control systems.

main branch which is meant to remain stable, while code for different stories is often developed and tested on separate branches. The branch that the code is initially worked on is stored as an attribute in our model, along with other metadata about the date, message and size of commit. Separately, we track the path to a file that was changed by the commit as an entity related to the commit, with specific statistics similar to the size of the commit.

A commit can be given a *tag* with a version number, which is an indicator that the version of the code is used as a release version or for other purposes. We store the tag as an entity with attributes for the tag's name, message, the developer made the tag and when it was made. Both a commit and a tag are linked to the sprint in which the specific entity was made.

An author of a commit or tag is considered a local *developer* to the version control system. We decide to encrypt personal data like name and email address using a project-specific encryption key, cf. Section 3.3.2.

Code review

Recent version control systems are often accompanied by web applications that allow development teams to propose and review code changes. They are often tightly integrated with the repository

and provide additional data about the development process. We consider three systems that provide the review functionality for their code repositories: GitHub, GitLab and TFS/VSTS/Azure DevOps. In Figure 3.6, the entities and relationships are shown for these specializations, with some entities performing a generalized role for multiple systems.

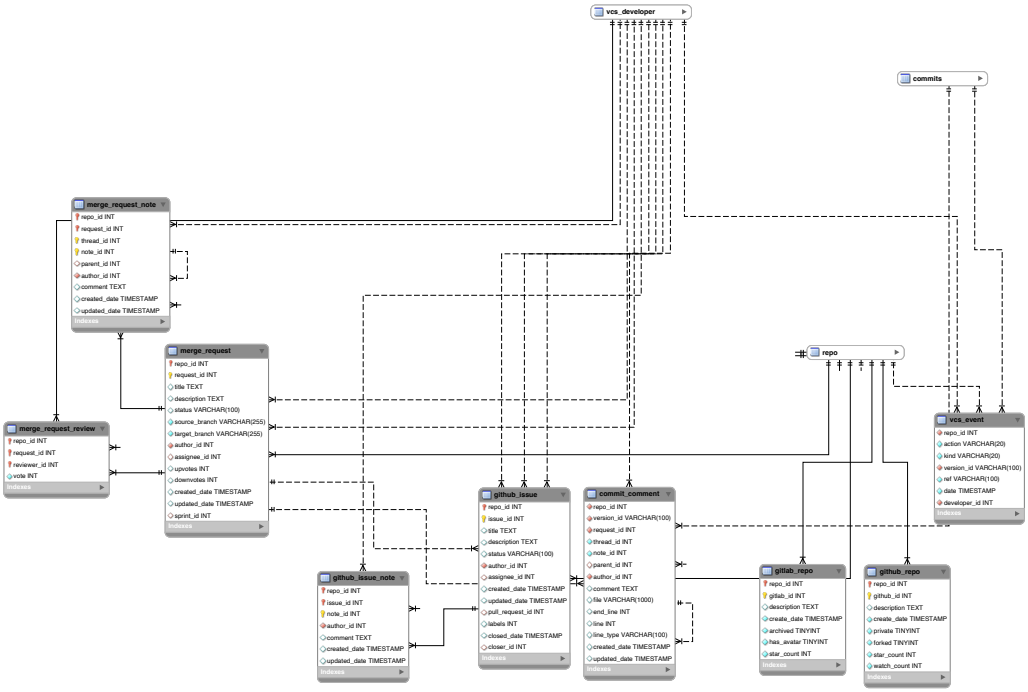

Figure 3.6: Entities and relationships that model data retrieved from code review systems.

We acquire additional metadata regarding the code repositories at GitLab and GitHub. These attributes are stored in specialized tables accompanying the existing entity. We obtain the description, creation date, access restrictions and social interactions such as the number of people giving a star to the repository, as a measure of importance.

The code review systems also contain additional information on interactions such as pushes of commits, tags and branches to the central repository. These *events* are usually not tracked in the Git repository itself, but the time difference between a developer finishing a local code change and publishing it has some relevance for our analysis.

The remaining entities from the code review systems are focused on comments on code, branch merges and standalone issues. A *commit comment* can be made for a code commit in order to discuss one or more lines of code, which were possibly changed by that commit. It is usually related to a *merge request*, which is a discussion regarding the merge of a branch to another—usually main—branch. Depending on the system, a reviewer is assigned to a merge request, who then votes on whether the code that was changed on the branch is ready. Involved developers and bots connected to the build platform and quality control systems sometimes also leave notes on the merge request itself. Additional metadata attributes are acquired for these requests and comments.

These review systems are also able to track issues. This functionality is, however, not used by most teams at ICTU, where they instead track issues in Jira. Issue tracking is done for projects that use GitHub, in particular a few open source projects from support teams at ICTU that are of interest to us. Wigo4it does use TFS/VSTS/Azure DevOps as a work item tracker, as explained in the next portion of the model.

Although limited in use at the organizations in our research project, we consider tracking all types of issues from such sources—including GitHub—to be helpful for the generalizability of our data acquisition pipeline to other development ecosystems, where the workflow of preparing and reviewing tasks may be conducted differently. Thus, we model those issues and notes as entities with similar metadata as the merge requests and notes.

TFS/VSTS/Azure DevOps

The integrated system provided by Azure DevOps Server, previously known as Team Foundation Server (TFS) and Visual Studio Team System (VSTS), encompasses version control via Git—although some older versions only provide its own TFVC protocol—with code review, build platforms and project management through work item tracking. This provides development teams with various options for tracking development process and delivering product increments.

Some organizations use certain functionality of this system or use it differently. At ICTU, TFS was only used by a few projects for version control and code review, with most preferring GitLab. Meanwhile, Wigo4it utilizes more from TFS/VSTS/Azure DevOps by performing sprint planning through work items, next to the version control.

We consider the portion that focuses on the work items to be separate from the version control and code review, which is modeled as part of the other specialized review systems that our schema supports. In Figure 3.7, we display the main entities and relationships that are extracted from the process of work item tracking.

The Azure DevOps system groups developers into teams with their own work item boards. These teams have team members, which have personal data that are encrypted with a common encryption key. Because some people are not part of a team but still interact with the work items—and conversely, we will not find every developer through work item updates—we also acquire developers through information from the work items. This information is stored in a secondary table, with personal data encrypted using a project-specific key. These two entities fulfill analogous roles to the developer tables from Jira, with data acquisition taking place in a similar way for that system.

Moreover, we acquire information regarding sprints from Azure DevOps. Due to differences in the attributes of entities compared to sprints from Jira, we store them separately. The same applies to the work items that stem from this system. Because the focus of our analysis was more clear once we adjusted our data acquisition pipeline to collect the work items, we extract fewer attributes from them. Some of the attributes also have different semantics compared to Jira issues, so the records for work items are stored in a separate table from that system.

Due to this, some of the entities have a dedicated role. When we later want to collect attributes or statistics from these entities, we have to select which data source is relevant to us and adjust the queries based on the definition of the attributes, such as what the status or resolution means to the work left on a story. We discuss some of the intricacies of this process for the way the queries are built in Section 3.3.2.

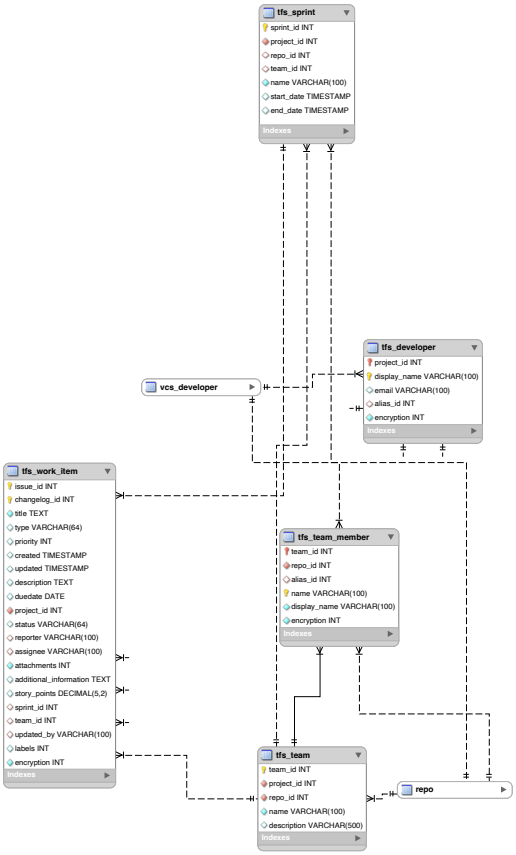

Figure 3.7: Entities and relationships that model data related to work item planning retrieved from TFS/VSTS/Azure DevOps.

Quality control

Code should be regularly inspected for code style issues, vulnerabilities and other indicators that technical debt are present due to maintainability issues. A system like SonarQube [V] automatically performs such checks. ICTU created an additional system to combine the history of those checks as well as reports made by security scanners, build systems and Jira in a central place that is easily accessible to team members. This system, Quality-time [VI], provides measurements in a form that we also model in GROS DB—while remaining compatible with SonarQube directly—as shown by the entities and relationships in Figure 3.8.

The quality control system formats the checks and measurements as a report that displays statistics grouped by software component. Each *metric* has a name, based on what is being measured and which domain object is involved. This domain object refers to other systems and artifacts, such as a code repository, document, build server or Jira board.

The metrics are checked at high frequency for updated information from external systems. A metric value comes with metadata about when the check happened, when the value has most

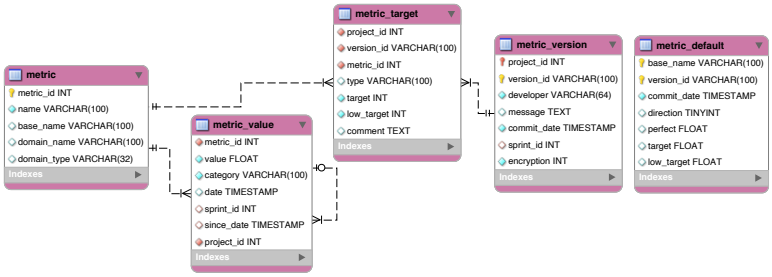

Figure 3.8: Entities and relationships that model data retrieved from quality systems.

recently changed before then and how problematic the value is. This severity category is defined by the target of the metric, which determines whether a lower or higher value is better. The target of a metric also has attributes for threshold values, which determine when the metric should be reported as acceptable, problematic or critical.

The target of a metric can be changed by a development team member or quality manager, usually if it is considered less or more important. Metadata on the version of the metric target is modeled as an entity. Finally, we track what the default value of each target is, based on the version of the Quality-time system, since the default could too have been altered. Combined, these entities allow us to deduce the context of a metric's value at an earlier date. Namely, we check why it was in a certain severity category back then and whether it would still be problematic now.

3.3.2 Linking data sources

The GROS DB contains data regarding events and instances of entities from different points in time, but also from different facets regarding several SCRUM software development processes. Work is undertaken on stories from a product backlog by refining the task description and enriching other attributes and links. Meanwhile, the developers write code that implements the expected change to functionality. The updated code is reviewed by their peers, in additional to quality control systems checking if the software remains maintainable.

While the process itself appears streamlined, one hindrance is that the steps are often separated, with limited linking between the entities and the systems they originate from. This reduces the perceptibility of potential bottlenecks and other issues that emerge during the process, because the ecosystem lacks a single view in which every team member is able to spot those problems. The lack of linking limits the capability for further analysis of patterns in development projects, as these relations are needed to gain a complete picture of the state of a project at any given moment.

Nevertheless, we automatically deduce relationships between different entities, which are then used to combine previously unlinked data in our queries. During our data acquisition, we define beforehand which systems, code repositories and other primary entities are in use by a certain project. We collect the data on a project-by-project basis, keeping this identifier in mind when importing the entities into the database. This gives the project entity a central role in our model. While this is sufficient for some analysis, we prefer to give teams a more relevant role than projects, when multiple projects are worked on by the same team, thus giving a better understanding of their total workload. For such purposes, we combine them through data analysis afterward or use the team entities from TFS/VSTS/Azure DevOps.

Next, in order to understand what is happening during a single sprint, we compare timestamps from the originating systems. For any event not yet linked to a sprint, such as a commit, metric measurement or adjustment to a metric target, we determine in which sprint it took place by finding the sprint with the closest start date before the event's timestamp, provided that the sprint did not end before then. It is beneficial to store this explicit link rather than attempt to find the relevant sprint during a query, because the latter approach would possibly find multiple or even false-positive connections, due to overlapping sprint dates. A "smarter" solution utilizing bisection properly handles these situations by identifying the most relevant sprint.

There are still limitations to automated relationship inference. When used on user-generated data, such practice becomes error-prone. For example, we wish to find relationships between code commits and the stories that they are being made for. One convention is that the developer writes the issue key or work item number in the commit message, but extracting the relevant identifier from free-form text could lead to matching unrelated substrings or mentions of irrelevant stories. There is also not a fixed custom for this, so it does not guarantee proper results for other projects and organizations. This obscures the semantics and proper use of a complex many-to-many relationship. Results from further analysis using this link would be incomplete, hard to understand or incorrect. Therefore, we have decided not to extract relationships from unstructured, textual attributes.

Another way to find what work is being done by the team in different systems is to determine simultaneous actions by the same developer. Initially, our model shows that personal data regarding each developer is scattered across portions of different systems. By linking developer entities across systems when they refer to the same person, we find the developer's affiliated activities in certain time frames. This method is potentially inexact and ambiguous when combining the data, but it allows us to filter out work that is not relevant to the developer through other means.

Problems with a person-based link arise when a developer chooses to use another name or email address in a version control system (VCS), thus having separate profiles. One example is a legal name in the employee records, stored in, e.g., LDAP, which differs from the name used on a day-to-day basis. Therefore, we utilize a mapping that links developers with different combinations of name and email address. The mapping is further employed to detect non-human accounts, such as automated code changes, by explicitly omitting a linked primary account. We then ignore those 'bot' accounts in further analysis.

Another important matter using personal data is the consideration of privacy. We prefer not to include information that can be traced back to a specific individual. We encrypt these attributes with a one-way encoding, in such a way that the developers are given a hash-based pseudonym. The mapping still detects identical developers as long as the encryption keys are available; both the mapping and encryption keys are only stored at the organization that the data originates from. Some data is encrypted with project-specific encryption keys, so developer names are encoded differently across systems if another key is used. For this reason, we have different tables with data using global encryption keys and project-specific keys, as indicated in Table 3.3.

We use `JOIN` operations to involve multiple entities through their relationships, allowing these queries to select attributes from several tables for further analysis. Another method of obtaining data from multiple originating systems is to combine them afterward based on primary identifiers. This way, distinct features have their own queries, and multiple features are combined into one data set based on project and sprint identifiers, for example. This adds processing time after performing the individual queries. Only after this step, the data set is made available to machine learning, where each sample in the data set elaborately describes a sprint.

System	Global encryption table	Project encryption table
Jira	`developer`	`project_developer`
TFS	`tfs_developer`	`tfs_team_member`
VCS	*None*	`vcs_developer`
LDAP	*None*	`ldap_developer`

Table 3.3: Portions of the database with personal data of developers as well as the tables in which we store said data using global encryption keys and project-specific keys.

We have designed another method to select and combine data from our cross-referenced database which takes advantage of the integration of the R programming language with MonetDB. We augment the syntax of SQL with a templating system which allows defining which tables, columns and relationships are involved in a query. This makes it possible to use the same query template for data selection from Jira entities as for those from TFS/VSTS/Azure DevOps, for instance. The columns involved in a relationship are provided separately so that the `JOIN` operation remains flexible. Similarly, we reuse definitions, such as what kinds of issues are considered stories, in more queries for other features or reports, by placing them in a central bank. Technical details are provided in Section 3.4.

After selecting the data from the relevant entities, we also want to report which sources were involved in the query. Tracking these sources is relevant for verifying if the information from this query properly reflects what the originating systems display. This way, there are no unforeseen conflicts between those systems and the database report. This also helps with making the query more insightful for stakeholders, who otherwise only see a number or other attribute without context. The data acquisition component of the pipeline tracks URLs of the systems that it requests data from. Additionally, the quality control system provides metadata to describe the code repositories and other monitored artifacts. Human-readable names make the references more familiar to the developers. For most queries, we directly link to the specific source, which is usually a report in the originating system showing the same information.

3.4 Architecture

The database must work properly and in accordance with our goals. Therefore, in this section, we focus on the technical components that we design and implement, supporting the desirable operation of the database. This includes integration of the database with the Grip on Software pipeline, which ranges from data acquisition to machine learning and information visualization.

The main purpose of the pipeline is to frequently provide new insights into patterns found in the wealth of data regarding Scrum software development processes. The pipeline, including the database, should not cause a huge strain on the existing platforms and processes. We keep in mind that we deploy the database to multiple software development ecosystems. As such, we use data from various issue-tracking project management systems in a similar fashion, for example. In addition, the database component necessitates efficient data backup and recovery functionality, allowing encrypted exports to be uploaded to a central instance for cross-organizational research.

The database administration and import component [d] plays a central role in meeting these objectives. A Java-based program uses Java Database Connectivity (JDBC) as a straightforward

method to interact with the database [XVII]. We import fields from JSON artifacts provided by the data acquisition component. First, the program selects which tasks are relevant to be performed for the provided collections of entities. Then, for each collection, the importer reads the objects for the entities one by one, using a custom buffered line reader for memory efficiency. An entity in the collection is used in a check to determine whether there is an existing row describing the same entity. Depending on the outcome, the database is given an update with the new data using either an `INSERT` or `UPDATE` statement. To sharply reduce the number of statements being sent to the database during the import, the aforementioned checks and updates are performed using *batched statements* [56]. Here, a subset of—or all of—the actions are performed in quick succession on the database's side through the use of a precompiled query template. During the import, MonetDB fills in the sets of values to check for and/or to store in the database, saving time transferring and compiling the query [57].

The importer program has more tasks next to revising the knowledge base on relevant entities. We determine the relationships between data from different systems here, for example when it comes to the sprint in which an event took place. The same applies for developer profiles, as described in Section 3.3.2. Other tasks include normalization of metric names, tracking the involved data sources, aligning changelog numbers and encryption of personal data.

The final encryption step takes place after the linking of developer data has been completed, because it is impractical to find relationships when all the attributes of the involved tables are encrypted. The data acquisition agent already encrypts personal data from version control systems at this point, while the names and email addresses from the issue tracker are still obtainable. Encryption keys use two attributes, a salt and pepper, which are produced by a cryptographically strong pseudorandom number generator. The sensitive data field is then encrypted using the salt, original value and pepper to form a SHA-256 hash [58]. The type of encryption keys used to hash the values is stored in a bit field attribute. The bit values indicate whether the encryption was done with a global encryption key, a project encryption key or both, in a particular order. Double-encryption is not usually done in our database, as seen in Table 3.3.

We also consider the database component's external security. Usually, the database is hosted in an ecosystem with limited access from the internet. To prevent potential dependency problems with separate firewalls, we restrict access to the port on which database connections are made. This does not impact any legitimate uses if all the processing takes place on the same server. This works well for a pipeline that is deployed on VMs, Docker platforms or other virtual network ecosystems, where forwarding firewalls or port mapping further restricts access to the database system.

Using the database administration component, we enable several maintenance tasks to be performed on the database. The database model and its synchronicity with the live system is important. We automatically validate the schema that is used to create the database tables against documentation, which compares properties of columns and primary keys, such as their type. If we want to adjust some of the properties or track a new entity, attribute or relationship, then upgrades to both the schema and the documented model must take place. We use a data-driven approach to determine whether the live database requires a schema change. We indicate for each portion of an upgrade what kind of action will be taken. If the action is feasible, i.e., the table or column can be created or adjusted compared to the current situation, then we perform the action. This avoids performing upgrades that we already applied to the live database. Further, it allows staggered updates which change a column of a table that is created by an earlier upgrade, without interfering with each other. Thus, the schema changes take place in an appropriate order.

Other maintenance tasks focus on database dumps, bulk imports and restoration. These auxiliary systems are configurable for different use cases. This has allowed the dump and import functionality to be reused in a cross-pipeline exchange setup. We do this along with a Java program for database export specifically for our use with MonetDB [e]. This program takes care of tables with encryption fields and large tables, so that the dump size remains manageable. The exported dump uses a hybrid format of CSV data and SQL instructions. Another module further encrypts the exported data using an asymmetric key-sharing setup [f]. A separate, private key of the organization and the public key published by the central pipeline instance are involved in the GPG encryption [IX]. The dump—without any unencrypted personal data or the encryption keys—is then uploaded via HTTPS to the central instance. At this web server [g], the payload is decrypted using the public key of the organization, which is known in advance, as well as the private key of the central pipeline. This ensures that the message cannot be compromised. Automated incremental dumps are thus regularly exported from the organizations and imported in a central database for analysis.

Aside from the bulk importer and export handler, the main pipeline component that accesses the database is the data analysis component [h]. We consider the feature extraction that this component performs in more detail in Chapter 4. From a technical standpoint, we use the query templates and definition banks introduced in Section 3.3.2 in order to determine which entity tables, attribute columns and relationship references are involved in each query, allowing us to build a generic and reusable data set. This integration of the R programming language with the MonetDB database uses an interpolation-based compiler, with recursive steps to expand contextually-defined variables and function calls into correct SQL. In Figure 3.9, the input data and supported functions are summarized.

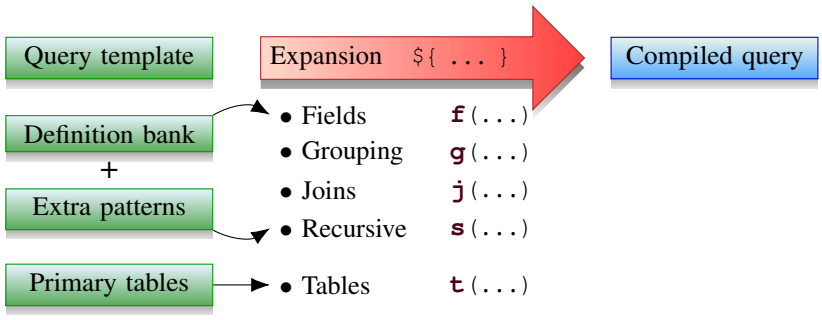

Figure 3.9: Input data (green blocks) and expansion functions of the query template compiler (red arrow), leading to a concrete query (blue block).

We show an example of a query template and a representation of the concrete queries for Jira and TFS/VSTS/Azure DevOps in Figure 3.10. The definition bank is extensible with additional patterns. Recursive expansion of these fields means that any template variables that they contain are further translated. A mapping of tables that are relevant to certain templates allows for translating the specific table where an entity is stored. For example, the issue entity expands to the `issue` table for Jira or `tfs_work_item` for TFS/VSTS/Azure DevOps.

```
1 SELECT DISTINCT ${f(join_cols, "issue")}, ${s(issue_key)} AS
      key
2 FROM gros.${t("issue")}
3 JOIN gros.${t("sprint")} ON ${j(join_cols, "issue",
      "sprint")}
4 WHERE ${s(issue_story)}
5 AND ${t("issue")}.updated > ${s(sprint_open)}
6 AND ${s(sprint_id, "issue")} <> 0
```

(a) Template

```
1 SELECT DISTINCT issue.project_id, issue.sprint_id,
      issue.key AS key
2 FROM gros.issue
3 JOIN gros.sprint ON issue.project_id =
      sprint.project_id AND issue.sprint_id =
      sprint.sprint_id
4 WHERE issue."type" = 7
5 AND issue.updated > COALESCE(CAST(sprint.start_date AS
      TIMESTAMP), CURRENT_TIMESTAMP())
6 AND COALESCE(issue.sprint_id, 0) <> 0
```

(b) Compiled for Jira

```
1 SELECT DISTINCT tfs_work_item.team_id,
      tfs_work_item.sprint_id, CONCAT('#',
      tfs_work_item.issue_id) AS key
2 FROM gros.tfs_work_item
3 JOIN gros.tfs_sprint ON tfs_work_item.team_id =
      tfs_sprint.team_id AND tfs_work_item.sprint_id =
      tfs_sprint.sprint_id
4 WHERE tfs_work_item."type" = 'Product Backlog Item'
5 AND tfs_work_item.updated >
      COALESCE(CAST(tfs_sprint.start_date AS TIMESTAMP),
      CURRENT_TIMESTAMP())
6 AND COALESCE(tfs_work_item.sprint_id, 0) <> 0
```

(c) Compiled for TFS/VSTS/Azure DevOps

Figure 3.10: A query that retrieves issue identifiers in each project's sprints, for a feature that counts the number of user stories, with different concrete versions that are expanded from the template when selecting data from the database.

3.5 Experiments

We test the GROS DB setup with regards to the performance of the database system during the usual workload of the data collection and analysis pipeline. To do this, we propose a number of experiments that look into the performance of MonetDB itself, as well as optimized refinements made to the queries in our template compiler.

Our scope is not to compare MonetDB to other database management systems. We consider the choice for MonetDB to be most reasonable at the moment when we initially created the pipeline. We considered the strong qualities of a column-based storage, relational SQL support and integration with a programming language for its functional deployment in a larger diverse development and research environment. In particular, a performance comparison with another RDBMS would require rethinking the data model for that particular database system. We would have to decide which indexes to include in the model, whereas MonetDB does not require manually-created indexes. Other data type constraints have been made specifically for our feature set and would require a different representation.

Comparisons between database systems are often done with industry-level benchmarks, such as TPC-H [XVIII]. We instead focus on a test set with a small number of queries that we have used during the Grip on Software research. The selected queries should be representative of the usual data collection for further analysis, machine learning and information visualization. Various parts of the data model are involved in the queries. Varying levels of complexity are used in order to provide the feature set and associated details in the query response.

We measure the performance of query templates that we have optimized with our compiler. In order to find out how much these changes improve the efficiency of database system, we go back to older versions of the query stored in our repository and include those in our test set, along with the newer version. We retroactively apply changes that we made to the query in the meantime, but only in case that these changes are not relevant to the performance.

3.5.1 Setup

In our performance experiments, we consider only the data access statements (`SELECT`), not other kinds of queries for data definition or data manipulation. The queries that build the database and fill it are not part of a typical workload in our consideration. In fact, the data import is already covered by other pipeline performance measurements in Section 2.5.

We do involve new database creation and storage in our experiments by testing queries in two situations. First, we run queries one by one on an existing database which has seen the queries before and could create optimized structures for them. This is considered a *hot-start* experiment. We test the same queries again, this time individually where we rebuild the database and import the database, in between each query run. This *cold-start* experiment provides more indication of the inherent complexity of each query for our data model.

We disregard caches in between cold-start runs. In all experiments, we disable the use of tables that hold outcomes of long-running queries. The performance test should not be interrupted by other processes when possible, thus other parts of the pipeline are disabled during the experiments.

Our performance test program reuses portions of the data analysis program that runs the queries [h]. Additionally, it ensures a reproducible and consistent setting for all the queries. We perform multiple runs of each query in order to obtain statistics on deviations. This allows us to determine if the test setup behaves similarly between runs.

The data set of ICTU is used as a representative instance upon which we apply our queries. Some relevant dimensions of the database are listed in Table 4.1, with 192 million measurements of quality control metrics on top of that.

The tests are performed on a Dell PowerEdge 2950 2u rack-mounted server with a Intel Xeon 8-core X5450 CPU at 3GHz, with 2×128KiB L1 cache, 2×12MiB L2 cache, four 4GiB memory cards of DDR2 FB-DIMM RAM of Hynix HYMP351F72AMP4N3Y5 (667MHz) and a 544GiB DELL PERC 6/i SCSI storage disk with LUKS-based 512-bit AES full disk encryption enabled. The database system under test is MonetDB v11.41.5 (Jul2021).

3.5.2 Results

We run six queries related to two large entities in our data model, namely metrics and issues. Most queries make use of related entities, such as sprints. The query templates and compiled versions are provided in Appendix B.

We collect the wall clock time that each query took, from the start of the query request until the data response, as well as the system time that the database used during the full query processing. In addition, we measure the average CPU load during the execution of the query. Finally, we track how many database rows were included in the query. This statistic occasionally differs between refined and original versions, due to a different method of linking with related sprints or by filtering operations. This is because some sample rows are not used in the data set during normal data analysis operations. Only sprints that exist are included in the data set. Thus, the query filters such sprints beforehand.

The performance measurements of the test results on a hot-start database are provided in Table 3.4 for the original queries and Table 3.5 for the refined versions, with 10 runs for each query. We report the mean and standard deviation of the values obtained from those runs. It is clear that the queries that we refined are faster, even if the database also optimizes the queries itself. Our optimization also appears to help with reducing the variance between runs, allowing for queries to run frequently and in a stable rate during research.

QUERY	WALL TIME	RUN TIME	CPU LOAD	ROWS
All metrics (Figure B.1)	11.29 ± 0.32	11.02 ± 0.33	81.3 ± 3.0	1898
Red metrics (Figure B.3)	2.01 ± 0.14	1.74 ± 0.13	72.1 ± 5.7	1775
Team spirit metric (Figure B.5)	10.77 ± 0.34	10.50 ± 0.35	76.4 ± 3.3	1063
Backlog added points (Figure B.7)	1.20 ± 0.03	0.93 ± 0.02	38.3 ± 1.1	16823
Backlog epic points (Figure B.9)	7.98 ± 0.09	7.70 ± 0.09	44.4 ± 0.7	32246
Backlog story points (Figure B.11)	5.61 ± 0.15	5.35 ± 0.13	33.5 ± 0.7	153685

Table 3.4: Results of performance tests of original versions of queries, using a hot-start database system. Times are provided in seconds, CPU load in percentages.

The queries related to metrics use more processing power in order to generate the result. Consequently, they take longer to produce the resulting data set. The queries that calculate product backlog sizes at different moments in time have a large data set, but perform relatively efficiently compared to the metrics queries. With larger data sets from each query, CPU load and in particular query response times decrease. This indicates that query optimization through the template compiler serves a practical use, when queries are rerun many times.

QUERY	WALL TIME	RUN TIME	CPU LOAD	ROWS
All metrics (Figure B.2)	5.76 ± 0.08	5.48 ± 0.09	81.2 ± 1.5	1899
Red metrics (Figure B.4)	1.76 ± 0.04	1.49 ± 0.03	79.3 ± 1.6	1776
Team spirit metric (Figure B.6)	2.54 ± 0.56	2.28 ± 0.04	79.5 ± 1.3	1063
Backlog added points (Figure B.8)	0.88 ± 0.04	0.62 ± 0.02	31.4 ± 1.3	16823
Backlog epic points (Figure B.10)	3.83 ± 0.07	3.57 ± 0.07	26.8 ± 0.9	25833
Backlog story points (Figure B.12)	4.05 ± 0.06	3.79 ± 0.06	27.4 ± 1.2	88032

Table 3.5: Results of performance tests of refined versions of queries, using a hot-start database system. Times are provided in seconds, CPU load in percentages.

We measure the performance of the same queries in a cold-start setup, where the database is restarted and all system caches are removed. The original and refined versions of the six queries in our test set are each run ten times again. The results of these experiment are shown in Tables 3.6 and 3.7.

QUERY	WALL TIME	RUN TIME	CPU LOAD	ROWS
All metrics (Figure B.1)	50.92 ± 0.53	50.86 ± 0.53	20.1 ± 0.6	1898
Red metrics (Figure B.3)	32.03 ± 0.20	31.97 ± 0.21	10.8 ± 0.4	1775
Team spirit metric (Figure B.5)	69.79 ± 0.39	69.80 ± 0.39	15.1 ± 0.3	1063
Backlog added points (Figure B.7)	2.02 ± 0.10	1.97 ± 0.10	22.1 ± 1.1	16823
Backlog epic points (Figure B.9)	8.53 ± 0.09	8.46 ± 0.10	41.2 ± 0.6	32246
Backlog story points (Figure B.11)	6.40 ± 0.11	6.34 ± 0.11	29.6 ± 0.5	153685

Table 3.6: Results of performance tests of original versions of queries, using a cold-start database system. Times are provided in seconds, CPU load in percentages.

QUERY	WALL TIME	RUN TIME	CPU LOAD	ROWS
All metrics (Figure B.2)	49.47 ± 0.49	49.41 ± 0.48	13.2 ± 0.4	1899
Red metrics (Figure B.4)	33.05 ± 0.25	32.99 ± 0.25	10.0 ± 0.0	1776
Team spirit metric (Figure B.6)	59.89 ± 0.70	59.82 ± 0.69	10.0 ± 0.0	1063
Backlog added points (Figure B.8)	1.61 ± 0.15	1.55 ± 0.16	17.5 ± 2.2	16823
Backlog epic points (Figure B.10)	4.45 ± 0.08	4.38 ± 0.08	23.2 ± 0.6	25833
Backlog story points (Figure B.12)	4.93 ± 0.17	4.86 ± 0.18	22.7 ± 0.8	88032

Table 3.7: Results of performance tests of refined versions of queries, using a cold-start database system. Times are provided in seconds, CPU load in percentages.

Again, we observe that the refined variants have a decreased query processing duration and load, although the effect is not as significant compared to the hot-start setup. One query, used to calculate the number of problematic red quality control metrics, is slightly slower.

All queries are faster to process in the hot-start situation than with the cold-start setup. This should be expected, given the presence of additional index-like structures that MonetDB generates

and reuses between the runs of the queries on a hot-start system. The queries that involve the metrics are several times slower in the cold-start setup, while the effect is less extreme for the backlog queries compared to the hot-start test. The average CPU load is however much lower during the cold-start experiment, for all queries. This may be due to longer run times spreading out the actual execution until enough data is obtained from disk. Another potential bottleneck is memory synchronization times between queries, allowing the overall load to decrease. In addition, there is less usage of intricate data structures that help with speeding up the query resolution but use more complex instructions.

Overall, the combination of a hot-start MonetDB database system and the refined queries seems to provide the most benefit to shorter query processing times and advantageous usage of the available resources. The higher load does not impact the system much, given that other processing cores are still available for other portions of the pipeline to be run. This allows us to use the GROS DB at various software ecosystems, which come with different hardware constraints and virtualization options.

3.6 Discussion

In this chapter, we build upon the introduction of the pipeline components in Chapter 2 and further detail the database system as a central component of the pipeline. During the Grip on Software research, there is a need to store information from several systems that are commonly used for SCRUM software development. We focus on modeling and integrating these data sources such that we link previously disconnected entities and perform frequent analysis on the data set.

MonetDB satisfies our purposes and requirements as an extensible RDBMS suitable in academic and corporate environments. The data model of GROS DB makes use of various data types provided by MonetDB, including timestamps and references between entities. MonetDB supports SQL as a core query language, but also enables extensions through the integration with programming languages such as R and Python. As a database system oriented to analytical processing, we find MonetDB to be the most applicable option for our research.

Our data model for the SCRUM software development process has many connections between different portions that roughly correspond to the systems that the data originates from. Initially, systems such as a project's issue tracker, version control system and quality dashboard lack substantial interaction. Through chronological information about events that make changes to entities, we build new relationships, where the sprint acts as a central entity. This model allows us to more easily extract metrics and features regarding this time frame.

Other links exist through the use of personal information, such as when a developer commits some code and resolves a story. We take privacy in mind by encrypting names and other identifiable attributes, with one-way encoding using project-specific encryption keys. We preserve links by using local translation mappings and comparing the hashes.

We enhance the selection of data from the model by introducing a query template compiler. This allows us to write queries that select the same kind of data, such as the number of stories worked in a sprint, while staying agnostic to the underlying portion of the data model used in the actual query. Depending on the organization and project, the templates are compiled to use, e.g., Jira or TFS/VSTS/Azure DevOps as a source. The relevant entities—stories versus work items as well as different types of sprints—plus the references used in `JOIN` operations on their tables are then automatically used in the proper locations.

There are technical challenges to make the database system stable, maintainable, interoperable with other parts of the pipeline and applicable in a wide range of software development ecosystems without many changes or influences on the existing infrastructure. Additional functionalities of the GROS DB allow us to fulfill these requirements, along with tasks for upgrading and exporting data to a central instance of the pipeline.

We test the performance of the database system through the use of a number of queries that are representative of our usual workload, along with older, non-refined versions that perform the query in a different way. We observe that the refined versions use less time to execute. Additionally, we show that MonetDB itself performs optimizations through the use of structures created when queries are run more often, which benefits our resource usage during our research. We easily perform our queries repeatedly whenever new data comes in. This is performed by a scheduled job system that runs hourly or even more frequently, depending on the other pipeline components.

More performance experiments could be done by comparing MonetDB to other potential database systems, such as MySQL, Postgres and SQLite [49]. In particular, MariaDB, as a fork of MySQL that includes options for column-based storage, is a consideration for further research options. Overall, the storage and performance options of MonetDB have proven to be helpful in our data modeling needs, as recognized throughout analytical processing problems.

We also consider an alternate solution where we integrate MonetDB with more Python programming, rather than with R. There exist many modules that make Python work well with column-based feature data, which allows deeper integration with machine learning purposes with fewer intermediate steps. Still, our query template compiler overcomes some issues that would need to be reimplemented in Python. We did purposefully design our query templates to not contain specifics of the language that the compiler is implemented in, in particular regarding the syntax of the expansion functions.

Our data model should be suitable for a large range of software development projects. Some portions are focused on a specification of data originating from specific systems, such as Jira and TFS/VSTS/Azure DevOps. Still, we easily support more integrated issue trackers and review systems, with GitLab and GitHub already part of the database model. Existing entities and relationships are proper templates as a starting ground for such extensions.

In summary, our research objectives are more easily attainable with the GROS DB database system based on MonetDB and extended with our query template compiler. We efficiently extract data sets based on the entities and relationships in our data model, with interconnections between portions based on discrete systems used in SCRUM software development. This data set accommodates different organizations while using the same query template, by taking advantage of the generic and flexible design of our data model and the technical component architecture. Thus, our pipeline remains adaptable to various development ecosystems. The performance of the pipeline component allows involved team members to benefit from recurring, large-scale machine learning and information visualization applications. By making the model inherently more understandable, the goals for improving predictability of the software development process become more feasible as well.